《Exploiting Windows 10 in a Local Network with WPAD/PAC and JScript》译

安全报告：	Exploiting Windows 10 in a Local Network with WPAD/PAC and JScript
报告编号：	B6-2017-122002
报告来源：	360 CERT
报告译者：	c1tas，de4dcr0w
原文作者：	Ivan Fratric, Thomas Dullien, James Forshaw and Steven Vittitoe
原文链接：	https://googleprojectzero.blogspot.com/2017/12/apacolypse-now-exploiting-windows-10-in_18.html
更新日期：	2017 年 12 月 20 日

目录

0x00 aPAColypse now: Exploiting Windows 10 in a Local Network with WPAD/-PAC and JScript . . . 2
0x01 Intro . . . 2
0x02 Web Proxy Auto-Discovery . . . 3
1. Intro . . . 3
2. Attack scenario: Local network via DHCP . . . 3
3. Attack scenario: Remote over the internet via privileged position and DNS 3
4. Attack scenario: Remote over the internet via malicious wpad_tld . . . 3
0x03 Bugs . . . 4
0x04 Exploit . . . 6
1. Understanding JScript VARs and Strings . . . 6
2. Stage 1:Infoleak . . . 7
3. Stage2: overflow . . . 10
4. Stage 3: bypass CFG . . . 15
5. Stage 4: Getting code execution as Local Service . . . 15
6. Stage 5: Privilege escalation . . . 16
7. Tying it all together . . . 17
0x05 Conclusion . . . 18

0x00 aPAColypse now: Exploiting Windows 10 in a Local Network with WPAD/PAC and JScript

by Ivan Fratric, Thomas Dullien, James Forshaw and Steven Vittitoe

0x01 Intro

许多广泛部署的技术在经过 20/20 hindsight(判断方式) 之后发现都是一些奇怪的或不必要的冒险想法

在 IT 里的工程决定往往在不完善的信息和时间压力下完成, 一些古怪的 IT 设置可能被解释为" 在这时候这看起来是个好的方案", 在这篇文章的作者中, 一些人发现 WPAD("Web Proxy Auto Discovery Protocol") 更多被称为 ("Proxy Auto-Config"), 就是一个奇怪的点

在互联网的早期，在 1996 之前，Netscape 的工程师们决定 JavaScript 是一种很好的语言来编写配置文件。其结果是 PAC-配置文件格式，其工作方式如下：浏览器连接到预先配置的服务器，下载 PAC 文件，并执行特定的 JavaScript 函数以确定适当的代理配置。为什么不呢？它肯定比 XML 更具表达性和更少的细节，似乎是向许多客户提供配置的合理方式。

PAC 本身加上一个叫做 WPAD-一个协议，使浏览器不必有一个预配置的服务器用来连接协议。相反，WPAD 允许计算机查询本地网络来确定从哪个服务器下载的 PAC 文件。

不知怎么的，这项技术最终成为了一个 1999 到期的 IETF 的草案，现在，在 2017，每台 Windows 机器都会问本地网络："嘿，我在哪里可以找到一个 JavaScript 文件来执行？"。这可以通过多种机制来实现：DNS、WINS，但也许最有趣的是 DHCP。

近年来，浏览器的开发已经从最初的 DOM 转向直接针对 JavaScript 引擎，所以仅仅提到我们可以在没有浏览器的情况下在网络上执行 JavaScript 是一种激励。初步调查显示 JS 引擎负责执行这些配置文件是 jscript.dll-遗留的 JS 引擎，也带动了 IE7 和 IE8（和仍然可在 IE11 在 IE7 / 8 兼容模式如果使用适当的脚本属性）。这有好有坏——一方面，它意味着不是每一个 Chakra bug 都是自动的本地网络远程攻击，但另一方面，它意味着一些相当旧的代码将负责执行我们的 JavaScript。

早先, 安全研究人员就警告过 WPAD 的危险性, 但这貌似我们知道的第一例, 证明利用 WPAD 进攻击是可以危害到用户机的

windows 不是唯一在使用 WPAD 的软体, 其他的操作系统和软件同样有在使用.Google Chrome 就同样具有 WPAD 解释器, 但是 Chrome 会把 PAC 文件中的 JS 代码放在沙箱中运行, 但其他的操作系统就没有这么做, 这就是 windows 为何是这种攻击最有兴趣的目标

0x02 Web Proxy Auto-Discovery

1. Intro

如上所述，WPAD 将查询 DHCP 和 DNS（按此顺序）以获取要连接的 URL-如果没有来自 DNS 的响应，则显然 LLMNR 和 Netbios 也可以使用。WPAD-over-DNS 的一些特性使得中间人攻击能够出人意料地发挥作用。

2. Attack scenario: Local network via DHCP

在最常见的场景中，机器将使用 option code 252 查询本地 DHCP 服务器。DHCP 服务器返回一个字符串如http://server.domain/proxyconfig.pac，指定一个 URL 的配置文件应该被加载。然后，客户端继续获取该文件，并将内容作为 JavaScript 执行。

在本地网络中，攻击者可以简单地模仿 DHCP 服务器 - 可以通过 ARP 欺骗或通过合法的 DHCP 进行竞争。然后，攻击者可以提供一个恶意 JavaScript 文件所在的 URL。

3. Attack scenario: Remote over the internet via privileged position and DNS

除了当地的网络攻击的情况下，事实上，查找 WPAD 这一请求也可能通过 DNS 查询发生的这一情况, 也就产生了更多攻击场景。许多用户配置他们的计算机执行 DNS 查找某个公共 DNS，全局可见的 DNS 服务器（如 8.8.8.8、8.8.4.4，地址 208.67.222.222 和 208.67.220.220）。在这样的情况下，机器会发出 DNS 查询（如 wpad.local），位于本地网络服务器之外。攻击者在网络上的特权位置（例如网关或任何其他上行主机）可以监视 DNS 查询并欺骗应答，指示客户端下载并执行恶意 JavaScript 文件。

像这样的设置似乎很常见-根据这一维基百科条目，DNS 根服务器所看到的流量中的一大部分是本地请求。

4. Attack scenario: Remote over the internet via malicious wpad_tld

WPAD 的特殊之处在于递归地遍历本地机器名称以查找要查询的域。如果一台机器被称为“laptop01.us.division.company.com”，则按照以下方式查询以下域名：

- wpad.us.division.company.com
- wpad.division.company.com

- wpad.company.com
- wpad.com

这（根据这个维基百科条目）过去导致人们注册 wpad.co.uk 并将流量重定向到在线拍卖网站。进一步引用该条目：

通过 WPAD 文件，攻击者可以将用户的浏览器指向自己的代理，拦截并修改所有的 WWW 流量。尽管在 2005 年对 Windows WPAD 处理进行了简单的修复，但它只解决了.com 域的问题。在 Kiwicon 的一个演示显示，世界其他地区仍然严重受到这个安全漏洞的威胁, 在新西兰注册了一个样本域名，用于测试目的，接收来自全国各地的代理请求，速度为几秒钟。几个 wpad.tld 域名（包括 COM，NET，ORG 和 US）现在指向客户端回送环回地址，以帮助防止此漏洞，尽管某些名称仍然被注册（wpad.co.uk）。

因此，管理员应该确保用户可以信任组织中的所有 DHCP 服务器，并且保证组织的所有可能的 WPAD 域都受到控制。此外，如果没有为组织配置 wpad 域，用户将转到域分层结构中具有下一个 wpad 站点的任何外部位置，并将其用于其配置。这允许只要注册了特定国家的 wpad 子域名，通过设置自己作为所有流量或感兴趣的站点的代理，就可以对该国的大部分互联网流量进行中间人攻击。

另一方面，IETF 草案明确要求客户只允许“规范”(例如非顶级域名)。我们还没有调查是否有人在什么程度上实施这个攻击，或者二级域名（如.co.uk）是否是流量重定向的历史案例的罪魁祸首。

无论哪种方式, 如果一个管理员注册 wpad.$TLD 给定组织的 TLD, 前提是该 TLD 没有被客户端实施明确列入黑名单, 那么在 JavaScript 引擎的漏洞就可以被通过互联网远程利用. 鉴于 1999 年的 IETF 草案提到了 1994 年的 TLD 列表（RFC1591），客户不太可能已经更新以反映新 TLD 的扩散。

我们尝试为各种 TLD 注册 wpad.co.$TLD 尚未成功。

0x03 Bugs

我们花了一些时间在寻找jscript.dll, 采用手工分析和 fuzzing,JScript最初的挑战是许许多多的"feature" 能在 javascript engines 触发 bugs 的 tips 无法在其工作, 应为它太陈旧而不支持它们. 举个例子

- 没有多种的 array tpyes(int array, float array etc.). 因此混淆一个数组类型是不可能的

- 没有更多的优化例如 ("fast paths"), 在更新的更快的 Javascript engines 中这些 fast path 的优化常常是 bug 的来源
- 不能对一个通常的 JavaScript 对象定义 getter/setter. 只在 dom 对象中可以调用的定义的属性, 但在 WPAD 进程中没有 DOM. 并且 DOM 对象中大量的 JScript 函数将 fail 并伴随一个消息"JScript object expected"
- 只要在一个对象被创建后, 不允许修改他的属性 (i.e. 这里没有___proto___属性)

然而,JScript 会遭受更多"old school" 的漏洞, 比如 use-after-free. JScript 的垃圾收集器 (a post).Jscript 使用了非世代 (no-generational) 的标记和清理垃圾收集器. 从本质上讲, 无论何时垃圾收集被触发, 它都会标记所有的 JScript 对象. 然后从一组"root" 对象 (有时候被称为"scavengers"(清道夫)) 开始扫描他们, 并从所有的对象中清除标记. 所有仍被标记的对象都将被删除. 一个经常出现的问题是, 默认情况下, 堆栈上的本地变量不会添加到根对象列表中, 这意味着程序员需要记住将他们添加到垃圾收集器的根列表, 特别是这些变量引用对象, 在函数的生命周期中可以被删除的时候.

另一种可能的漏洞类型包括 buffer overflows(缓冲区溢出),uninitialized variables 等等

Fuzzing, 我们使用了基于语法的Domato fuzzing 引擎, 并且写了一套新的 JScript 语法. 我们把发现的有趣的内置的属性和函数添加到这个语法中, 通过观察不同 JScript 对象的EnsureBuildin方法. 被添加的 JScript 语法在 Doamto repository(here)

在 fuzzing 和手工测试中我们发现了 7 个安全漏洞, 整理如下表

Vulnerability class	Vulnerabilities affecting IE8 mode	Vulnerabilities affecting IE7 mode
Use-after-free	1340,1376,1381	1376
Heap overflow	1369,1383	1369,1383
Uninitialized variable	1378	1378
Out-of-bounds read	1382	1382
Total	7	5

在发布这篇文章的时候, 所有的 bugs 已经被修复了

这个表格通过漏洞触发类别, 和所需要的兼容模式来区分.JScript 在 WPAD 中相当于在 IE 兼容模式运行脚本, 这意味我们虽然发现了 7 个漏洞, 但实际只有 5 个引发 WPAD, 然而其他漏洞任然可以被用在 IE(包括 IE11) 投入 IE8 兼容模式的恶意网页

0x04 Exploit

1. Understanding JScript VARs and Strings

在这篇文章中的接下来的部分, 我们将讨论 JScript VARs 和 Strings, 在讨论漏洞工作方式之前, 先讲述这些内容是十分有用的

JScript VAR 是一个 24-byte(64 位) 结构, 这表示一个 JavaScript 变量, 本质上是和 VARIANT 的数据结构相同的 (MSDN described). 在多数情况下 (足以跟踪漏洞) 内存布局如下所示

Offset	Size	Description
0	2	变量类型, 3 对应整形, 5 对应双精度浮点数, 8 对应字符串etc.
8	8	取决于类型, 可能是一个立即数或指针
16	8	大多数类型都不会使用

例如，我们可以用一个 VAR 表示一个双精度数字，它在前 2 个字节中是 5 (表示 double 类型)，后跟一个实际的双值，在偏移量为 8。最后 8 个字节将没有使用，但如果从这个 VAR. 复制另一个 VAR 的值，它们将被复制

一个 JScript 的 String 是 VAR 的一种类型在标志位为 8, 并且指针偏移量是 8. 这个指针指向一个 BSTR 结构体 (described here). 在 64 位的情况下 BSTR 的结构如下

Offset	Size	Description
0	4	未使用
4	4	字符串长度,未包含结束符的长度
8	length+2	以\0结尾的16bit大小的字符串

一个 String VAR 指针直接指向字符数组, 这意味着, 获取一个字符串的长度, 指针需要-4 从那个位置读取长度. 注意,BSTRs 被 OleAut32.dll 的处理, 被分配在独特的堆 (i.e. 意思就是其他的 Jscript 的对象被分配在其他的堆)

释放 BSTR 也不同于大多数对象, 在释放的时候, 不再是直接释放一个 BSTR, 当 SysFreeSting 被调用的时候, 它首先会想一个字符串放在被 OleAut32.dll 控制的 cache 中. 这个机制在Javascript Heap 风水

2. Stage 1:Infoleak

infoleak 的目的在于获取一个完全由我们控制的 string 在内存的地址, 我们在这个点上没获得任何可执行模块的地址, 但接下来会. 然而, 我们的目标是击破 hig-entropy heap randomization(高熵堆随机化) 并且让第二阶段的 exploit 可靠, 而不需要是用堆喷

因此我们主要在 RegExp.lastParen 中使用这个漏洞. 为了理解这个 bug, 我们先来仔细看看 jscript!RegExpFncObj 的内存布局, 以及他对应的 JScript RegExp 对象. 在偏移 OxAC 处, RegExpFncObj 包含 20 个整数的缓冲区. 实际上这是 10 对整数: 每一对的第一个元素的输入字符串的 start index(开始索引), 第二个元素是 end index(结束索引). 只要 RegExp.test,RegExp.exec 或带有 RegExp 参数的 String.search 在匹配正则语法-组 (group) 的时候 (RegExp 中的圆括号), 匹配的开始和结束索引就会存储在这里. 显然的缓冲区中只有 10 对空间, 所以只有前 10 个匹配项会被存在这个缓冲区中

但是, 如果 RegExp.lastParen 被调用, 并且有超过 10 个的捕获组的时候,RegExpFncObj::LastParen 会很高兴的使用捕获组的数量作为缓冲区的索引, 从而导致越界读取

这是一个 PoC:

```
var r= new RegExp(Array(100).join('()'));
''.search(r);
alert(RegExp.lastParen);
```

这两项指标 (我们称之为 start_index, end_index) 读取缓冲区边界之外, 并且可以达到任意大的区域. 假设这第一个越界访问不会造成崩溃, 如果这些索引中的值大于输入字符串的长度, 那将发生第二次越界读取, 这将允许我们读取输入字符串边界之外的内容. 像这样的字符串越界读取, 将字符串内容返回到字符串变量中的可以被检测到的调用处.

这第二次越界读将会是我们所使用的, 但首先我们需要去明白如何获得 start_index, end_index 的控制权. 幸运的是, 在 RegExpFncObj 的布局中, 在 index 缓冲区后的有一段数据我们可以控制:RegExp.input 的值. 通过设置 RegExp.input 为一个整数值, 并且使用一个由 41 组空括号组成的 RegExp(正则表达式), 当 RegExp.lastParen 被调用的时候,start_index 将会变成 0, 然后 end_indx 的值将会是我们写到 RegExp.input 的值

如果我们将一个输入字符串和一个释放字符串相邻, 然后通过越界读取输入字符串的边界外内容, 我们就可以获得堆的 metadata, 比如指向其他空闲堆的指针 (堆块的红黑树的左, 右和父节点参阅window10 堆内部信息获得更多的信息),Image1 展示了相应对象在 infoleak 时的情况

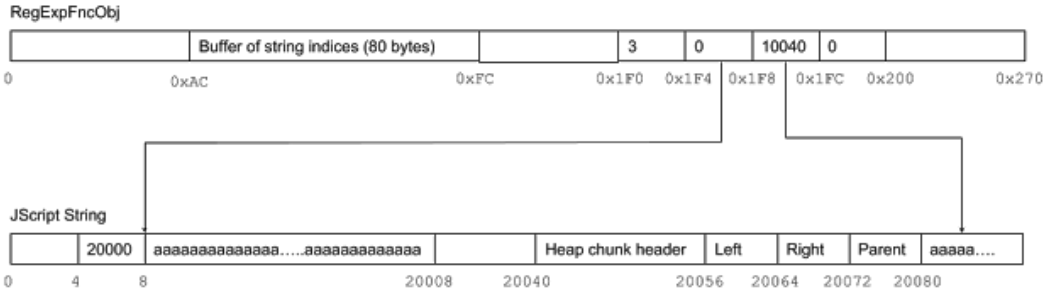

Image 1: Heap infoleak layout

我们将使用 20000 bytes-long 的字符串作输入, 为了不被分配在低碎片堆 (Low Fragmentation Heap)(LFH 只能被用来 16k 字节或更小的分配). 因为 LFH 的堆 metadata 是不同的, 在 win10 的 Segment 堆中不包含有用的指针. 此外,LFH 还引入了随机性, 这会影响到我们把输入字符串被放在释放字符串旁边的构想

通过从返回的字符串中读取堆的 metadata, 我们可以获得一个释放字符串的地址, 那么我们分配一个与释放字符串相同大小的字符串, 它可能被分配在这个地址, 我们实现了我们的目标, 那就是我们知道了一个由我们控制的字符串的地址

这整个 infoleak 的过程大概是:

1. 分配 1000 10000-character 的 String(提示: 10000 characters == 20000 bytes)
2. 释放每第二个 (Free every second one)
3. 触发 infoleak bug, 使用剩余的字符串之一作为输入字符串并越界读取 20080 个 bytes(字节)
4. 分析泄露的字符串并获得指向其中一个释放字符串的指针
5. 使用特定内容分配与释放的字符串 (10000 个 characters) 长度相同的 500 个字符串

特定内容的字符串在这个步骤还不重要, 但在下一阶段将变得十分重要, 因此会在下一个阶段描述它. 还注意, 通过检查堆 metadata, 我们可以很容易的确定进程正在使用的堆 (segment 堆和 NT 堆)

Image2 和 Image3 展示了使用Heap History Viewer 在 infoleak 时生成的堆可视化的图像绿色的条纹表示分配块 (被字符串占用), 灰色的条纹标识分配后, 但是被释放的后再分配的块,(指向的是我们释放后, 又触发 infoleak 后被重新分配到 (the stings we free and then reallocate after triggering the infoleak bug)), 白色的条纹是代表从未被分配的数据 (保护页). 你可以看到随着时间的推移字符串是如何被分配的, 然后一半被释放的 (灰色的), 一段时间后又被分配 (条

纹变成绿色)

我们可以看到, 每 3 个这样大小的分配后就会有一个保护页. 我们的 exploit 是从来没有想过去接触这些保护页的 (这发生在读取字符串尾少量的数据 (it reads too little data past the end of the string for that to occur)) 但有 1/3 的几率,infoleak 输入的字符串后面讲不会有空字符串, 所以预期的堆 metadata 将会丢失. 但是我们可以很容易的检测到这种情况, 并且使用另一个输入字符串触发 infoleak 错误, 或者终止 exploit(注意: 到目前为止我们没有触发任何内存损坏)

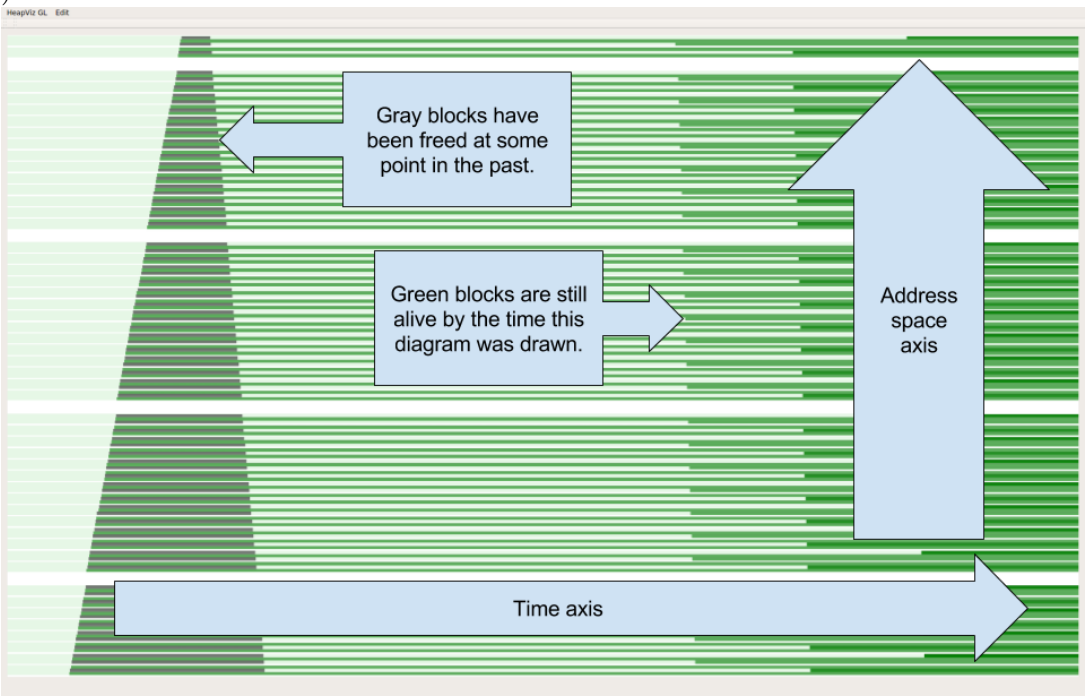

Image 2: Heap Diagram: Showing the evolution of the heap over time

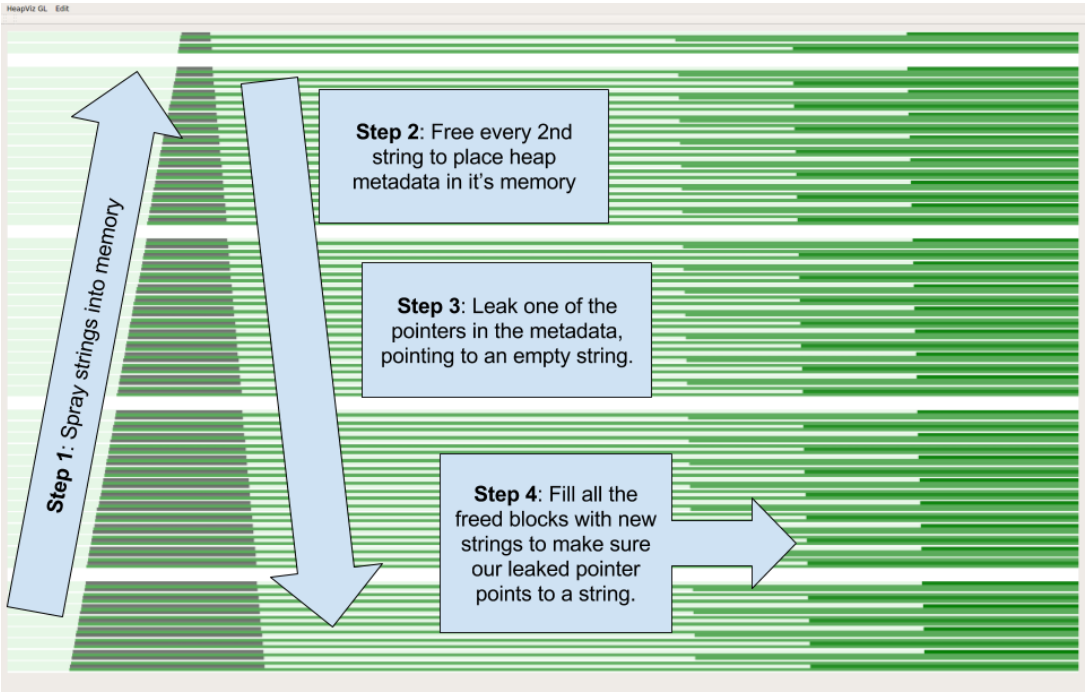

Image 3: Step-by-step illustration of leaking a pointer to a string.

3. Stage2: overflow

在 exploit 的第二个阶段, 我们使用this heap overflow bug 在 Array.sort 中. 如果驶入的数组中元素数量大于 Array.length/2,JsArrayStringHeapSort(如果未指定比较函数则被 Array.sort 调用) 就会分配一个与当前数组元素数量相同大小的临时的缓冲区 (注意: 可比 array.length 小). 然后尝试从 0 到 Array.length 的每一个数组引索检索相应的元素, 如果该元素存在, 则将其添加到缓冲区, 并转化为字符串. 如果数组在 JSArrayStringHeapSort 的生命周期中没有改变, 这将工作正常. 但是,JsArrayStringHeapSort 将数组元素转化为 Strings 将触发 toString() 回调. 如果在其中一个 toString() 回调元素被添加到之前未定义的数组中时, 将发生溢出.

为了更好的理解这个 bug 以及其可利用性, 我们来仔细看看我们将会溢出的缓冲区结构. 已经提到, 数组的大小和当前正在输入数组中的元素数量相同 (确切的说, 这将是元素数量 +1) 数组的每个元素都是 48 字节 (64 位) 结构如下

Offset	Size	Description
0	8	指针指向一个原始VAR在偏移16之后被转换为的string
8	4	当前元素的index
16	24	VAR保存原来的数组元素
40	4	数字0或1取决于偏移16处VAR类型

在 JsArrayStringHeapSort 期间, 检索索引为<array.length>的数组的每个元素, 并且如果元素被定义, 这会发生以下情况

1. 这个数组元素将被读到偏移 16 的 VAR 中
2. 原始的 VAR 将被转换为一个 String VAR. 指向这个 String VAR 的指针被写到偏移 0
3. 在偏移 8, 这个元素的 index 将被写入
4. 取决于原始 VAR 的类型,0 或 1 将被写入偏移 40

看到临时缓存区的 buffer, 我们不能直接控制大部分. 如果一个数组成员是一个字符串, 那么在它偏移量为 0 和 24 时, 我们将会有一个指针, 当他被解引用的时候, 我们将在偏移 8 处包含另一个指向我们控制数据的指针. 然而, 这是一个间接的在大多数情况下对我们有用的大层次

但是, 如果数组的成员是双精度, 则在偏移量 24(对应原始 VAR 中的偏移量 8) 中, 该数值将被写入, 并直接在我们的控制之下.

如果我们使用与阶段 1 中获得指针相同的双重表示形式创建一个数字, 那么可以使我们的溢出在缓存区结束后的某个地方使用指向我们直接控制的内存的指针来覆盖指针

现在问题变成了, 我们可以使用这个方法改写这个漏洞, 如果我们仔细研究一下对象如何在 JScript 中工作, 那么答案可能就是其中一个

每个对象 (具体的说, 一个 NameList 的 JScript 对象) 将有一个指向 hash 表的指针. 这个 hash 表只是一个指针数组. 当一个对象的成员元素被访问的时候, 元素名称的 hash 被计算. 然后取消对应于 hash 表最低位的偏移处指针. 这个指针指向一个对象元素的链表, 并且这个链表被遍历, 直到我们到达一个与请求元素具有相同名字的元素. 这将在 Image4 中显示

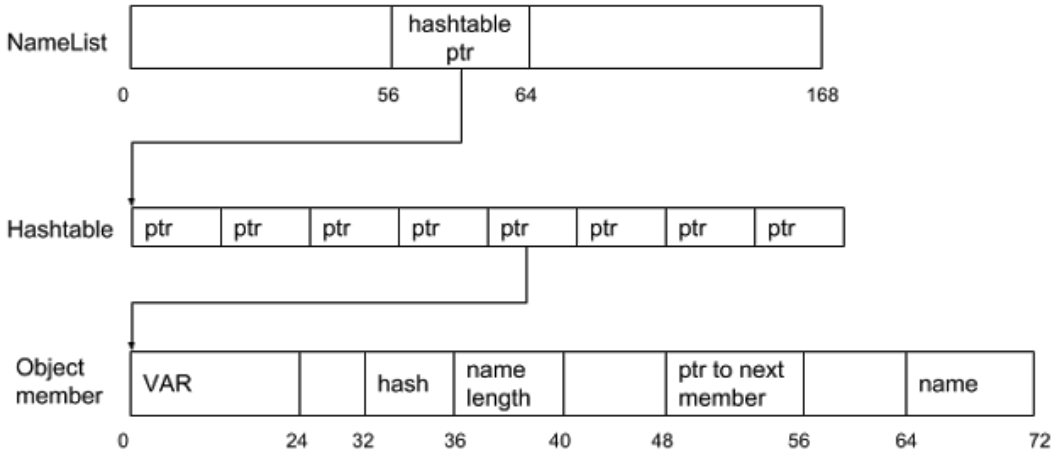

Image 4: JScript Object element internals

请注意, 当元素名称长度少于 4 个字节时, 它存储在与 VAR(元素值) 相同的结构中, 否则将会有个指向元素名称的指针. 名称 <=4 对于我们来说已经足够, 所以我们不需要深入了解这个细节

一个对象的 hash 表是覆盖的很好的候选者, 因为

1. 我们能通过访问相应的成员对象来控制它的那些元素被解引用. 我们用我们不控制的数据覆盖的元素将永远不会被访问
2. 通过控制相应对象有多少个成员, 我们对 hash 表的控制有限. 例如,hash 表以 1024 个字节开始, 但如果我们向对象添加多于 512 个元素,hash 表将被重新分配为 8192 个字节
3. 通过用指向我们控制的数据的指针, 覆盖 hash 表指针, 我们可以在我们控制的数据中创建假的 JScript 变量, 并通过访问相应对象成员来访问它们

要更可靠的覆盖, 我们需要执行以下操作:

1. 分配和释放大小为 8192 的大量内存块. 这就将代开 LFH 以分配 8192 大小的堆. 这将确保我们溢出的缓冲区, 以及我们正在溢出的 hash 表被分配在 LFH 上. 这一点很重要, 因为这意味着附近不会有其他大小的分配来破坏漏洞攻击 (因为 LFH bucket 只能包含特定大小的分配), 这反过来将确保我们将高度可靠地完全覆盖我们想要的东西
2. 创建 2000 个对象, 每个对象包含 512 个成员. 在这种状态下, 每个对象都有一个 1024 字节的 hash 表, 但是向其中一个对象添加一个元素将使其增长到 8192 个字节

3. 添加第 513 个元素到前 1000 个对象中, 这导致 1000 个 8192-byte 的 hash 表被分配

4. 使用长度为 300 包含 170 个元素的数组触发 Array.sort. 这将分配一个 (170 + 1)*48 = 8208bytes. 由于 LFH 的规则, 这个对象将被分配在与 8192 字节 hash 表相同的 LFH bucket 中.

5. 立即在 (第一个数组元素的 toString() 方法中) 向第二个 1000 个对象添加第 513 个元素. 这使得我们非常确定, 现在排序缓冲区是相邻 hash 表之一. 在相同的 toString() 方法中, 还会向数组中添加更多的元素, 这将使它超出边界.

Image5 显示了排序缓存区地址周围的堆可视化 (红线). 你可以看到排序缓存区被大小相近的分配所包围, 这些分配都与 hash 表对应. 你也可以观测到 LFH 随机性, 因为随后分配的不一定在随后的地址上, 然而这对我们的利用没有任何影响

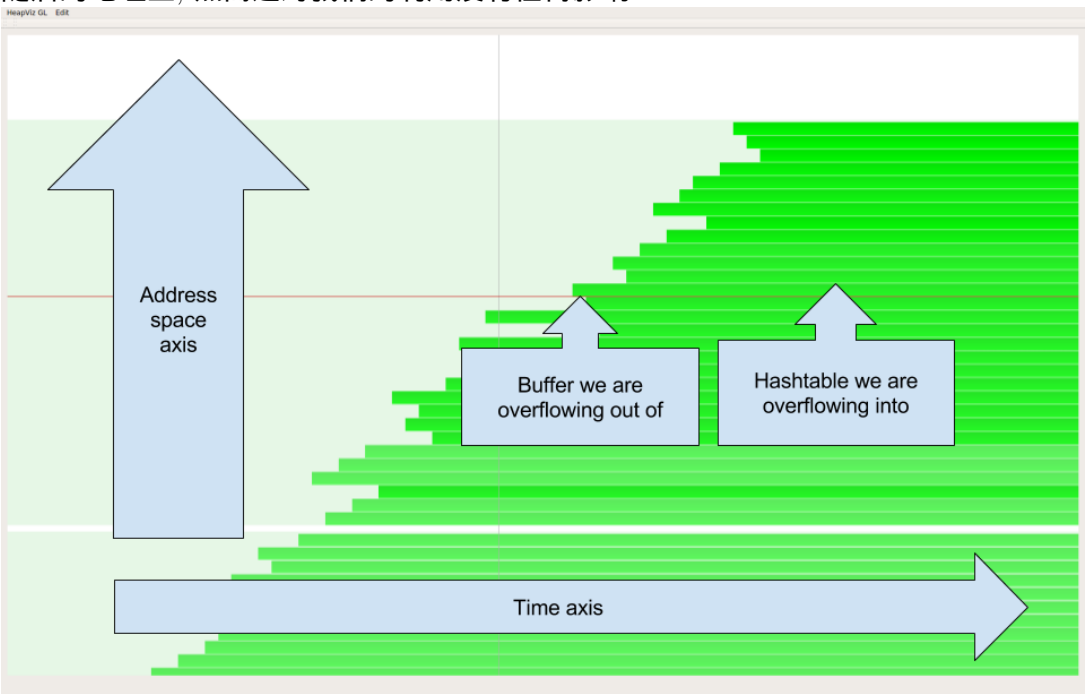

Image 5: Heap visualization around the overflow buffer

正如前面提到的, 我们以这样一种方式制作了我们的溢出: 一个不幸的 JScript 对象的 hash 表指针将会被指向我们控制的数据指针覆盖. 现在我们把这些数据放在什么地方: 我们制作了一个包含 5 个 (假)JavaScript 变量的方法:

1. 变量 1 只包含数字 1337.

2. 变量 2 是特殊类型的 0x400c. 这个类型基本上告诉 Javascript, 实际的 VAR 是有偏移量为 8 的指针指向的, 在读取或写入这个变量之前, 这个指针应该被解引用. 在我们的例

子中, 这个指针指向变量 1 之前的 16 个字节, 这基本上意味着变量 2 的最后 8 字节的 qword 和变量 1 的第一个 8 字节的 qword 重叠

3. 变量 3, 变量 4 和变量 5 是简单的整数. 它们的特殊之处在于它们分别在最后 8 字节包含数字 5,8,0x400c.

Image6 显示了溢出之后被破坏的对象状态

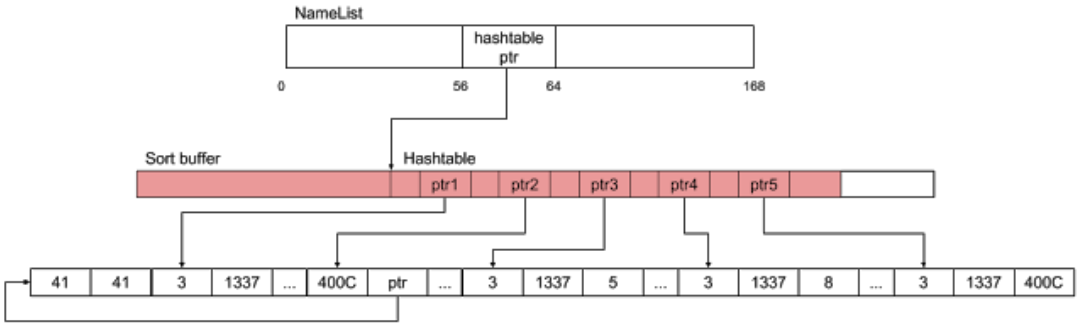

Image 6: State of objects after the overflow. Red areas indicate where the overflow occurred. Each box in the bottom row (except those marked as '...') corresponds to 8 bytes. Data contained in '...'boxes is omitted for clarity

我们可以访问变量 1, 只需要访问正确引索处的已破坏对象 (我们之前称为 Index1), 对于变量 2-5 也是如此. 实际上, 我们可以通过访问所有对象的 Index1 来检测哪个对象被破坏, 并查看哪个对象有现在的特征值 1337

重叠变量 1, 变量 2 的作用是可以将变量 1 的类型 (第一个 word) 改为 5(double),8(字符串), 或 0x400c(指针). 我们通过读取变量 2,3 或 4 然后将读取的值写入变量 2 来实现. 例如语句:

```
corruptedobject[index2] = corruptedobject[index4];
```

具有将变量 1 的类型更改为字符串 (8) 的效果, 而变量 1 的所有其他字段将保持不变

这种布局给了我们几个非常强大的开发原语:

- 如果我们将一个包含指针的变量写入变量 1, 我们可以通过将变量 1 的类型改为 double(5) 并将其读取出来公开该指针的值
- 我们可以通过在改地址上伪造一个字符串来在任意地址上公开 (读取) 内存. 我们可以通过首先将与我们想要读取的地址对应 double 的值写入变量 1, 然后将变量 1 的类型更改为 Stirng(8) 来完成此操作

- 首先将对应于地址的数值写入变量 1, 然后将变量 1 的类型更改为 0x400c(指针), 最后将一些数据写入变量 1, 从而写入任意地址.

有了这些利用源语, 通常获得代码执行将非常简单, 但由于我们正在利用 win10, 我们首先要绕过 Control Flow Guard(CFG).

4. Stage 3: bypass CFG

我们可以使用一些已知的绕过方式，但是如果攻击者有读写的权限就可以针对 jscript.dll 进行方便绕过。我们想要达到以下的效果：

1. 返回地址没有受 CFG 保护
2. 一些 Jscript 对象已经指向原始的栈

特别地，每一个 NameTbl 对象（在 Jscript 中，所有的 JavaScript 对象都从 NameTbl 继承），在偏移 24 的位置保存一个指向 CSession 对象的指针。CSession 对象偏移 80 的位置保存着指向靠近原始栈顶的指针。

所以，如果可以任意读，通过任何 Jscript 对象的一连串指针，有可能获得指向原始栈的指针。如果可以任意写，就有可能绕过 CFG 来覆盖返回地址。

5. Stage 4: Getting code execution as Local Service

准备好所有攻击的条件，我们现在可以实现代码执行。具体步骤如下：

1. 从任一 JScript 对象的虚表中读取 jscript.dll 的地址
2. 通过读取 jscript.dll 的导入表获取 kernel32.dll 的地址
3. 通过读取 kernel32.dll 的导入表获取 kernelbase.dll 的地址
4. 从 kernel32.dll 中搜索我们需要的 rop gadget
5. 从 kernel32.dll 的导出表中获取 WinExec 函数的地址
6. 根据上文的描述泄露栈地址
7. 准备好 ROP 链并写入栈中，从最接近泄露的栈地址的一个返回地址开始执行 rop

我们需要的 rop 链大致如下：

```
[address of RET]  //needed to align the stack to 16 bytes
[address of POP RCX; RET] //loads the first parameter into rcx
[address of command to execute]
[address of POP RDX; RET] //loads the second parameter into rdx
1
[address of WinExec]
```

通过执行 rop 链我们可以调用 WinExec 函数调用指定的命令。例如：如果我们运行‘cmd’命令，将产生一个命令提示符，和 WPAD 服务运行用户相同的权限。不幸的是，在子进程起的本地服务无法连接网络，但是我们可以将提权代码从内存落地到可写可执行的本地磁盘中。

6. Stage 5: Privilege escalation

当本地服务用户是一个 service 用户时，并没有管理员权限。这意味着 exploit 在进行系统的访问和修改操作时有很大的局限性，特别是攻击过后或者系统重启后的持久化。但是 Windows 上总是有可能找到未修补的提权漏洞，所以我们不需要找一个新的提权漏洞。我们可以滥用一个 built-in 功能实现从本地服务权限到 SYSTEM 权限。下图是 WPAD 中 service 用户被赋予的权限：

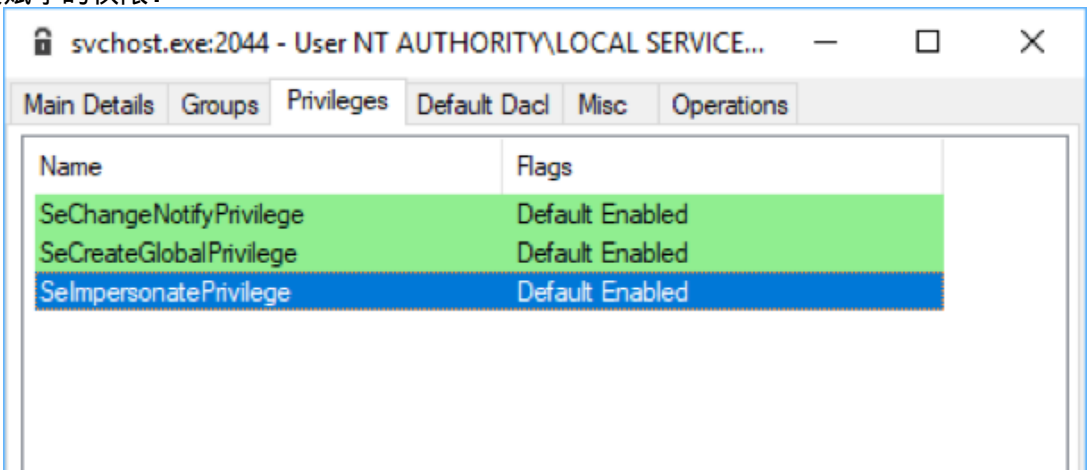

Image 7: Service Access Token's Privileges showing Impersonate Privilege

我们只能得到以上三个权限，但是图中高亮的 SeImpersonatePrivilege 很重要。该权限允许此服务获得本地系统其他用户的权限。Impersonate 权限表明该服务接受本地系统其他用户的请求，可能要代表这些用户执行一些操作。然而只要我们获得这些用户的访问令牌，就可以获得这些用户的全部权限，包括 SYSTEM 用户，这使得我们在本地系统上拥有了管理员权限。

滥用 Impersonate 权限是 Windows 安全模式（你可以通过查找Token Kidnapping获取更多细节）下一个已知的问题。微软已经尝试让它更难获得其他用户的访问令牌，但是不可能避免所有可能的情况。例如，James 发现了一个 Windows DCOM 实现的一个漏洞。该漏洞允许任意用户获得 SYSTEM 用户访问令牌。微软修复直接提权的漏洞，但是没有或者很难修复

token kidnapping 问题。我们可以滥用这个功能来获取 SYSTEM 权限，然后完全控制该系统，比如安装一个提权服务。

这里有一个通过 DCOM 实现的 token kidnapping(RottenPotato)，但是这个实现是被设计用于 Metasploit 框架中 getsystem 命令的，我们无法使用。所以我们用 C++ 实现了更简易的版本，通过 CreateProcessWithToken API 直接产生带有 SYSTEM 令牌的随机进程。我们将它编译成 11KB 大小的可执行程序，比 RottenPotato 小的多，方便落地到本地，执行 rop 链。

7. Tying it all together

当 WPAD 服务请求 PAC 文件时，我们将运行恶意文件来攻击 WPAD 服务，运行 WinExec 来落地和运行提权程序。该程序会在 SYSTEM 权限下执行命令（我们例子中是‘cmd’的硬编码）。

该 exploit 在我们的实验环境中运行得十分稳定，但是 100% 的稳定性并不需要，因为如果该 exploit 使 WPAD 服务崩溃了，一旦用户发起另一个 WPAD 服务请求，就会产生新的进程，所以攻击者可以一直尝试。如果用户没有关闭 Window 错误报告，将不会有图形化的页面显示 WPAD 服务崩溃了。如果开启，将会捕捉到这个崩溃，然后报告给微软。

事实上，我们的 exploit 一旦运行它的 payload 就会使 WPAD 服务崩溃，所以如果我们在服务被攻击后继续保持恶意的 PAC 文件，将会再次攻击服务。你可以在图片 7 中看到该效果，在攻击服务后的几分钟里，在受害者机器上产生了大量的 HTTP 请求。

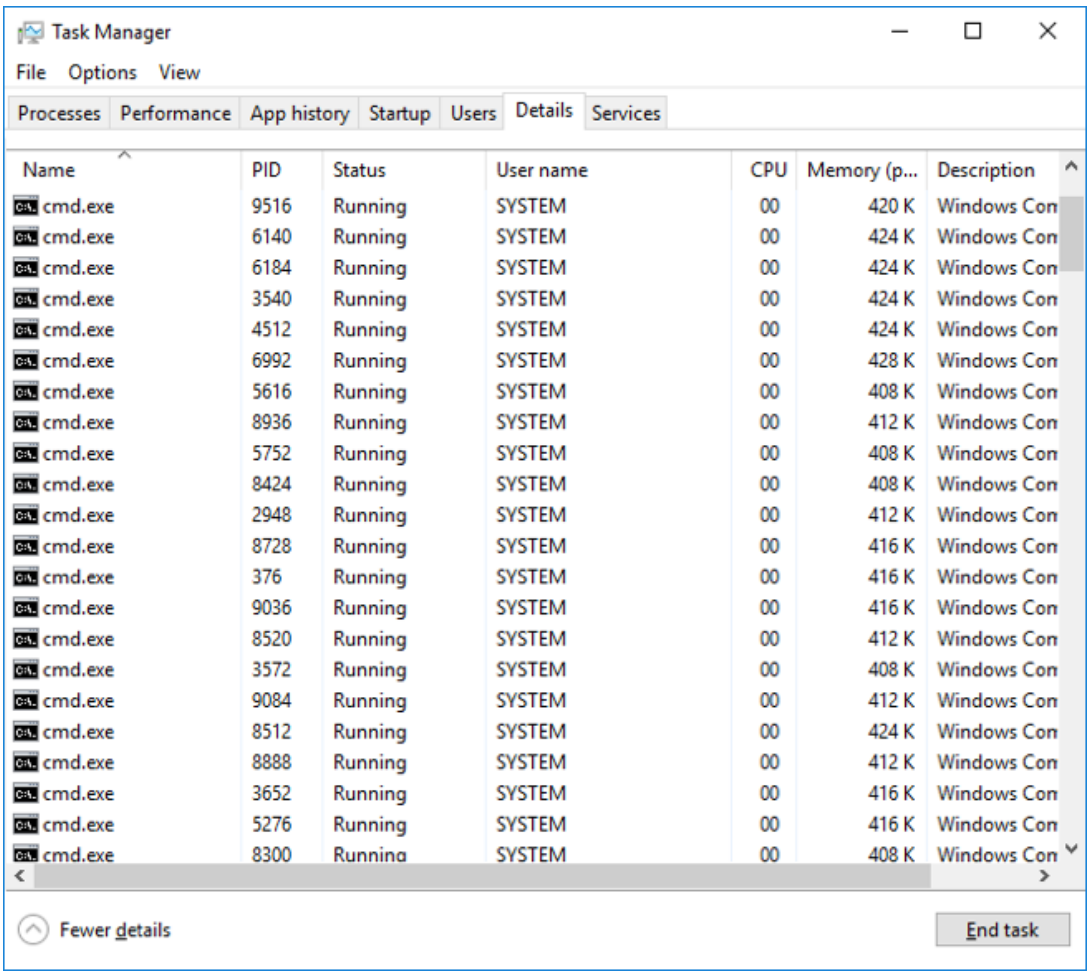

Image 8: Did we leave the exploit running for too long?

我们将很快在 issue tracker 上公布我们的 exploit 源码

0x05 Conclusion

执行不信任的 JavaScript 代码是危险的，在一个没有沙箱的进程里执行更是如此。即使是在像 jscript.dll 这样相对安全的 JavaScript 引擎中也是这样。我们披露了 jscript.dll 7 个安全问题，成功证明了可以从本地网络（或者之外）达到稳定地代码执行。即使是在 Windows 10 64-bit with Fall Creators Update 版本的全补丁（撰写本文之前）也可以实现。

尽管现在 bug 已经被修复了，是否意味我们就可以完成任务，安枕无忧了？事实不然，尽管我们花了相当多的时间，精力和电力成本来发现 jscript.dll 漏洞，但是我们并没有表明我们找到了所有漏洞。事实上，发现了 7 个漏洞，就很有很可能发现第 8 个。所以一些事情不改变，早晚有一天我们很可能会在野外看到这些利用。（这还是乐观地假设攻击者没有这种能力。）

那么，微软能做什么来使未来像这类攻击变的艰难：

1. 默认关闭 WPAD，事实上，虽然其他的操作系统也支持 WPAD，但是 Windows 是唯一默认开启的。

2. 沙箱化在 WPAD 服务中的 Jscript 解释器。当解释器需要执行一个定义好输入输出的 JavaScript 函数时，应该直接沙箱化它。考虑到输入输出模型的简易性，如果微软推出一个类似于 seccomp-strict，有限制性的沙箱：一些进程不需要高于“接收一些数据”、“执行一些计算”、“返回一些数据”的权限。

如果你想要自己解决这些问题，当前防止这类攻击（除了未知的漏洞）的唯一方法似乎就是完全关闭 WinHttpAutoProxySvc 服务。由于有其他服务依赖于 WPAD，所以在 Services 界面里有时不能关闭 WinHttpAutoProxySvc 服务（“Startup type”可能变灰失效），但是可以通过修改相应的注册表项关闭：将“HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\WinHttpAutoProxySvc”中的“Start”值从 3（手动）改成 4（关闭）。

当搜索“disabling WPAD”时，在网上发现一些常见的建议在我们的实验中并不能防范攻击：

1. 在控制面板关闭“Automaticallydetectsettings”

2. 设置“WpadOverride”注册表值

3. 在hosts文件中写入“255.255.255.255 wpad”（这将会关闭 DNS 变量，但是不能关闭 DHCP 变量）