APTs Way: Evading your EBNIDS

Ali Abbasi
Jos Wetzels
Who we are?

• **Ali Abbasi:**
  – PhD student in Distributed and Embedded System Security Group at University of Twente. Researching on embedded systems security related to critical infrastructures. Got M.Sc. at Tsinghua University in China, and was working as head of vulnerability analysis and penetration testing group at Iran National CERT in Sharif University of Technology in Tehran.

• **Jos Wetzels:**
  – M.Sc. Student and a research assistant with the Services, Cyber security and Safety research group at the University of Twente. Currently working on projects aimed at on-the-fly detection and containment of unknown malware and Advanced Persistent Threats, where we focus on malware analysis, intrusion detection, and evasion techniques. Assisted teaching hands-on offensive security classes for graduate students at the Dutch Kerckhoffs Institute for several years.
Plan of Talk

- History of Exploitation and Shellcodes
- Intro to Emulation Based NIDS Approach
- Adaptation
- Detection Techniques and Heuristics
- Evasions
- Questions?
History

• Morris Worm 1988 used Buffer overflow on “finger” service on VAX systems.

• In 1990 first polymorphic virus designed by Washburn

• In 2001 K2 introduced ADMmutate a polymorphic engine to generate shellcodes

• In 2008 Conficker worm with one byte XORed shellcode

Morris fingerd shellcode
pushl $68732f '/sh\0'
pushl $6e69622f '/bin'
movl sp, r10
pushl $0
pushl $0
pushl r10
pushl $3
movl sp,ap
chmk $3b
Signature Based IDS

• Typical Exploit Code:

<table>
<thead>
<tr>
<th>JUNK</th>
<th>Overwrite RET</th>
<th>Padding Instruction (NOP)</th>
<th>Shellcode</th>
</tr>
</thead>
</table>

Exploit the System

Detection based on:

– Return Addresses
– NOP Instructions (\x90)
– Shellcode signatures
– Detecting polymorphic encoder signatures
Limitations of Signature based NIDS

• Attackers change a byte of the payload and evade detection.

• Polymorphic shellcodes with custom encoders/decoders will evade detection.

• You must always update and maintain your signatures.
Emulation-Based NIDS, a Giant Leap

• Emulation-Based NIDSes emulate suspicious payloads.

• Meant to solve the problem of detecting polymorphic shellcodes.

• Emulation-Based NIDSes are a great step forward:
  – Detect polymorphic shellcodes regardless of which type of encoding technique is used.
  – Can detect 0-day exploits.
  – Do not rely on any specific vulnerability (signatures).
  – Uses heuristics, a behavior black listing technique.
How Emulation Based NIDS Works?

1. **Pre-Processing**
   - Network Data
   - Disassembly
     - GetPC
     - Code Detected
     - Yes
     - No

2. **Emulation**
   - Generate Execution Trace
   - Execution Finished?
     - Yes
     - No

3. **Heuristic-based Detection**
   - Match traces with heuristics
   - Traces Matches
     - Yes
     - No

4. **Generate Alert**
Emulation Based Technique Adopted

libemu x86 emu

Honeynet Project

NEMU

dionaea catches bugs

NoAH
Emulation Based NIDS

• Nemu:
  • The state of the art in emulation based network intrusion detection because of its broad range of heuristics.

• Libemu:
  • A simple shellcode detection engine (used in several Honeynet projects).
• Looking for GetPC seeding instruction.
  – Call instructions

```c
jmp startup
Getpc:
  mov (%esp), %eax
ret
startup:
call getpc
```

```c
/* emulate.c Heuristic detection trigger*/
if ((tc[prev_PC].inst.type == INSTRUCTION_TYPE_CALL)
    || (tc[prev_PC].inst.type == INSTRUCTION_TYPE_FSTENV)) {
  has_getpc = 1;
  EXECTRACE_CMD(inst_trace[num_exec].getpc = 1);
}
```

```c
/* 1 if call/fstenv, 2 if PC read, 0 if none */
if (((tc[prev_PC].inst.type == INSTRUCTION_TYPE_CALL) ||
    (tc[prev_PC].inst.type == INSTRUCTION_TYPE_FSTENV))
  {
  has_getpc = 1;
  EXECTRACE_CMD(inst_trace[num_exec].getpc = 1);
}
```

• Pre-Processing

```c
if (inst_trace[x].getpc == 1) {
  /* getPC write */
  fprintf(trace_fp, "\033[1;31mw \033[0m";
  }else if (inst_trace[x].getpc == 2) {
  /* getPC read */
  fprintf(trace_fp, "\033[1;31mr \033[0m";
  }
```

00C67000 D9 EE   fldz
00C67002 D9 74 24 F4 fnstenv [esp-0Ch]
00C67006 5B   pop   ebx
Emulation

• Create possibility to track the behavior of the emulated CPU during execution

• Emulate X86 instruction sets

• Emulate FPU Instructions

• make a generic memory image for some local variables
Basic Heuristics Detection

- **GetPC Code:**
  - detect invoking CALL or FPU instructions and check if the emulator started from the seeding GetPC code.

- **Payload Read:**
  - detect polymorphic shellcode by observing in an execution trace some form of GetPC code followed by a number of unique memory reads exceeding so-called PRT.

- **Write-Execute Instructions:**
  - Check in the areas that emulator performed write instructions how many executed X instructions get emulated. If this X instructions pass certain value then the payload will be flagged as Non-self-contained shellcode.
Additional Heuristics

- Kernel32.dll based address resolution
- SEH-based GetPC code
- Process Memory Scanning
PEB Based Kernel32.dll Resolution

```
mov eax, fs:[0x30]; PEB
mov eax,[eax+0x0C]; LoaderData
mov eax,[eax+0x1C]; InInitializationOrderModulelist.flink
lodsd ; Get 2\textsuperscript{nd} entry in list
mov eax,[eax+0x08] ; base address
```
BCKWD Kernel32.dll Resolution

; start walking SEH chain
Mov esi,fs:[0]

;pointer to kernel32.dll
Mov eax,[eax+4]

Is [eax] == 'MZ'? 

;traverse chain
Lods
Mov esi,eax

No

Yes

is [eax] == 0xFFFFFFFF?

Yes

Eax now holds kernel32.dll base address

No

;iterate backwards through memory image
Dec eax
Xor ax,ax
SEH GetPC

; push SEH Handler on stack
push (mov esi,[esi]; jmp esi)
push (popad lods lods lods)

; install custom handler
mov eax,[fs:0]
mov [eax+4], esp

; initiate to faulty address
xor esi, esi

; first execution will trigger access violation
; handler will catch exception, store PC in
; esi and return to instruction
Xor [esi+offset], KEY

[Encoded Body]

Xor [esi+offset+index], KEY
Index++

Index < body_length
Syscall Process Memory Scanning

1. `ebx = 0`
2. `ebx += 1`
3. Use `NtAddAtom` to check if page is valid.
4. Move `eax`, `SYSCALL# (NtAddAtom)`
5. `INT 0x2E`
6. Next page or `bx, 0xffff`
7. `; Was it valid?`
8. `(5 to check for no access violation)`
9. `cmp al, 5`
10. If yes, then:
    - `; is [ebx] == MARKER?`
    - `jmp ebx`
## Evasions

<table>
<thead>
<tr>
<th></th>
<th>Implementation</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Processing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Emulation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heuristics</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Intrinsic Limitations

• Unavailable context data
  – Emulation-based NIDSes cannot have a complete memory image of all possible targets.
  – Context keying.
  – Non-self contained shellcodes.

• Execution threshold
  – The emulator needs to stop at some point, the attacker can wait.

• Cannot deal with fragmented shellcode
  – Send the shellcode payload in multiple (non-consecutive) fragments.
Unavailable Context Data

- Non-self contained shellcodes

- Context Keying
  - CKPE
    - Using CPUID, values present at static memory addresses, system time or file information as a key.
Context Keyed Payload Encoding

Index = 0

;GetPC key
;ebx = GetContextKey1()

;body key
edx = GetContextKey2()

;encoded GetPC on stack
push (ENC(DWORD[mov eax, esp; ret]) ^ ebx)
On stack

;GetPC
call esp

Xor [esi+offset+index], edx
index++

[Encoded Body]
index < body_length
Execution Threshold

• Using time consuming loops to evade the threshold of execution

```c
while (++num_exec < exec_threshold);
STATS_CMD(if (num_exec >= exec_threshold) stop_cond = S_THRESH);
```

<table>
<thead>
<tr>
<th></th>
<th>Opaque loop</th>
<th>Intensive loop</th>
<th>Integrated loop</th>
<th>RDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nemu</td>
<td>9/9</td>
<td>9/9</td>
<td>0/9</td>
<td>0/9</td>
</tr>
<tr>
<td>Libemu</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>
Execution Threshold Random Decryption Algorithm (RDA)

Key candidate for GetPC stub
  ebx = 0
  Push (DWORD([mov eax, esp; ret] ^ KEY) to stack

; esp ^= ebx
  puts PC in eax
  ; Call esp

; body = eax + BODY_Offset
  Different key candidate for body
  ; ebx = 0

next key candidate
  ebx += 1

next key candidate
  ebx += 1

is hash([esp] ^ ebx) = GETPC_STUB_HASH?
  Yes
    encode(body, ebx)
    jmp body
  No
    decode(body, ebx)
    is hash(body) = SHELLCODE_HASH?
    Yes
    decode(body, ebx)
    jmp body
    No
    next key candidate
    ebx += 1
Fragmentation

- Very rare condition
- Shellcode will be sent in two different instances.
- Shellcode have two stage but in one instance
Results

• Context keying
  – Modified version of the Context CPUID Metasploit key generator stub.
  – **Not detected.**

• Non-self contained shellcodes:
  – Dynamically built the entire GetPC code and the shellcode decoder out of ROP gadgets.
  – **Not detected.**

• Execution Threshold
  – Built shellcodes with four types of time-intensive loops.
  – Nemu could **detect half** of the shellcodes (loops were not taking enough time).
  – Libemu could **not detect any**.
Demo

- RDA (Exec Threshold)
- CKPE
Implementation Limitations

• Heuristics are kind of black listing
  – You have to list all possible shellcode behavior patterns, attackers can always find a missing one.

• Runtime difference (Emulator detection)
  – Shellcode can detect if it is being emulated.

• Unsupported instructions

• Detection relies on successful shellcode disassembly
  – Malware already applies anti-disassembly techniques to avoid analysis
Heuristics Evasion

• Kernel32.dll address resolution evasion.

• Evading Payload Read:
  – Use syscalls to execute read operations instead of reading directly in the payload shellcode.

• Evading W-X Instruction:
  – Using Virtual Mapping

• Evasion of Process memory scanning :
  – SEH-walking to evade detection of SEH-based process memory scanning heuristic
  – API-based egg-hunting to evade SYSCALL-based memory scanning heuristic
Kernel32.dll Resolution Heuristic Evasion

• Evading Kernel32.dll heuristic using SEH Chain.

• Evading Kernel32.dll heuristic using Stack Frame pointers (using NtcreateProcess API)
Evading Kernel32.dll Heuristic using SEH Chain
Kernel32.dll Heuristic Evasion using Stack Frame Walking

esi = ebp

esi = [esi]

is [esi] == 0?

Yes  No

esi now holds address of last stackframe dword after that is return address of stack frame
Esi = [esi+4]

esi now points within NtCreateProcess in NTDLL.DLL

esi now holds base address of NTDLL.DLL
aLdrLoadDLL = GetAddress (esi, 'LdrLoadDLL')
Eax = aLdrLoadDLL('Kernel32.dll')

esi -=- 1

is [esi] == 'MZ'?
Payload Read Threshold Heuristic Evasion

SYSCALL-based relocation

```
NtAllocateVirtualMemory(CurrentProcess(),
    &memAddress, 0, shellcodeSize,
    MEM_COMMIT, Pager_ReadWrite_Execute)
    jmp startup
```

startup:
call getBack
[Encoded Body]

```
getBack:
NtReadVirtualMemory(CurrentProcess(),
    [esp], memAddress, shellcodeSize)
    ;decode bytes at address memAddress
    Decode(memAddress)
    ret
```
Stack Constructing Shellcode
GetPC+PRT evasion

<table>
<thead>
<tr>
<th>Code Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push [shellcode[n-4:n] ^ KEY]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Push [shellcode[4:8] ^ KEY]</td>
</tr>
<tr>
<td>Push [shellcode[0:4] ^ KEY]</td>
</tr>
<tr>
<td>;decode bytes at esp</td>
</tr>
<tr>
<td>jmp esp</td>
</tr>
</tbody>
</table>
Egg Hunt (Using API)

```
; ebx = 0

Next page or bx, 0xffff

; Was it valid?
(5 to check for no access violation)
cmp al, 5

Yes

; ebx += 1
; use NtAddAtom to check if page is valid
mov eax, SYSCALL# (NtAddAtom)
INT 0x2E

; is [ebx] == MARKER?

Yes

No

jmp ebx

No

Next page Or bx, 0xffff

; ebx = 0

No

Yes

; do info.permissions indicates page has read+execute?

VirtualQuery(ebx, &info, sizeof(MEMORY_BASIC_INFORMATION))

; is [eax] == MARKER?

Yes

No

jmp (eax+sizeof(MARKER))
```
Heuristics Evasion Demo

• PRT Evasion

• Kernel32 Evasion (Both Techniques)

• Process Memory Scanning Evasion
Emulator Detection

Nemu GP Register detection

Libemu
Timing

;read clock using RDTSC
;execute NOP loop
;read clock using RDTSC
;time1 = difference between clockreads

;read clock using RDTSC
;execute IMUL loop
;read clock using RDTSC
;time2 = difference between clockreads

;eax = time2/time1

jmp startup

startup:
call GetPC

GetPC:
;if eax is smaller than threshold, move esp to ecx

;if eax was smaller, eax(result) now holds return address
Mov eax,[ecx]
ret
Emulator Detection Demo

- Demo
Anti-Disassembly

• Using garbage bytes and opaque predicates
• Flow redirection to the middle of an instruction
• Push/pop-math stack-constructed shellcode
• Code transposition

<table>
<thead>
<tr>
<th></th>
<th>Garbage Byte</th>
<th>Flow Redirect</th>
<th>Push/Pop Math</th>
<th>Code Transposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nemmu</td>
<td>9/9</td>
<td>9/9</td>
<td>8/9</td>
<td>8/9</td>
</tr>
<tr>
<td>Libemu</td>
<td>0/1</td>
<td>1/1</td>
<td>0/1</td>
<td>1/1</td>
</tr>
</tbody>
</table>
Unsupported Instructions

- FPU Instructions (FNSTENV, FNSAVE)
- MMX Instructions
- SSE Instructions
- Obsolete instructions (salc or xlatb)

<table>
<thead>
<tr>
<th></th>
<th>FPU (FNSTENV)</th>
<th>FPU (FNSAVE)</th>
<th>MMX</th>
<th>SSE</th>
<th>OBSOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nemu</td>
<td>9/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
</tr>
<tr>
<td>Libemu</td>
<td>1/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>
Unsupported Instructions

**MMX**
- jmp startup
- Startup: call GetPC
- GetPC:
  - MMX instructions to move [esp] to mm0
  - MMX instructions to move mm0 to mm1
  - MMX instructions to move mm1 to eax (result)
  - ret

**SSE**
- jmp startup
- Startup: call GetPC
- GetPC:
  - SSE instructions to move [esp] to xmm0
  - SSE instructions to copy eax to xmm0
  - Overwrite contents of [esp] with junk bytes
  - SSE instructions to restore [esp] from xmm0
  - ret
Question?

Everything that has a beginning has an end

The Matrix Revolution

Contact Us:
Ali Abbasi: a.abbasi@utwente.nl
Jos Wetzels: a.l.g.m.wetzels@student.utwente.nl