Defending against Polymorphic Attacks: Recent Results and Open Questions

Michalis Polychronakis
mikepo@ics.forth.gr

Institute of Computer Science
Foundation for Research and Technology – Hellas
Crete, Greece

TERENA Networking Conference, 17 May 2006
What this talk is about

0-day network-level attack detection

- Existing detection methods
- Evasion techniques
- A novel polymorphic attack detector
- Open issues
System Compromise

1. Place the attack code into a buffer

2. Divert the execution path of the vulnerable process (exploitation)
   - Stack/heap/integer overflow
   - Format string abuse
   - Arbitrary data corruption

3. Execute the injected code (shellcode)
   - Performs arbitrary operations under the privileges of the process that has been exploited

\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00
Signature-based Detection (1/2)

➔ Known attacks
  • GET default.ida?NNNNNNNNNNN...

➔ Unknown attacks
  • Generic signatures for suspicious code sequences
  • NOP sleds, system calls, …

```plaintext
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE Linux shellcode"; content:"|90 90 90 E8 C0 FF FF FF|/bin/sh"; classtype:shellcode-detect; sid:652; rev:9;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|"; classtype:system-call-detect; sid:650; rev:8;)
```
Signature-based Detection (2/2)

- Lots of work on automated signature generation
  - Honeycomb, Earlybird, Autograph, PADS, Polygraph, Hamsa, …

- Common idea: find invariant parts among multiple attack instances
  - Then turned into token subsequences ➔ regular expressions

- Limitations
  - Not effective for targeted attacks – only for worm-like attacks
  - Autograph/Polygraph/Hamsa need a pool of “suspect” flows: Which are suspect flows? What if the pool gets intentionally polluted?

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 10202:10203 (msg:"CA license GCR overflow attempt"; flow:to_server,established; content:"GCR NETWORK<"; depth:12; offset:3; nocase; pcre:"/\S{65}|\S+s+\S{65}|\S+s+\S+\S+\S+s+\S\{65}/Ri"; sid:3520;)
```
Polymorphism (1/2)

→ Naïve encryption

- The decryptor remains the same in each attack instance
- *Easy to fingerprint using typical string signatures*

→ NOP code interspersion

- NOPs’ type/position/number varies in each attack vector
- *Can be fingerprinted using regular expressions*
Polymorphism (2/2)

- Code obfuscation/metamorphism
  - Instruction substitution
  - Code block transposition
  - Register reassignment
  - Dead code insertion
  - *Hard to fingerprint using regexps if applied extensively*

- Combination of all techniques
  - *Signature extraction becomes infeasible*
Recent proposals heuristically identify malicious code inside network flows using static binary code analysis
  - [Kruegel’05], [Chinchani’05], [Payer’05]

Step forward – beyond pattern-matching
  - Do not depend on invariant content

Basic steps
  1. **Disassembly**
  2. **Control Flow Graph extraction**

Initial approaches focused on the shellcodes’ sled component
  - Abstract Payload Execution [Kruegel’02]
    - Pioneer network-level static analysis work
  - Orthogonal to above approaches
Static Analysis Resistant Shellcode (1/4)

- Static binary code analysis is generally accurate for compiled and well-structured binaries

- Shellcode is not normal code!
  - Written/tweaked at assembly level: complete freedom…

- The attacker can specially craft the shellcode to hinder disassembly and CFG extraction
  - Anti-disassembly tricks
  - Indirect addressing
  - Self-modifying code
Running example

- Encrypted shellcode generated by the Countdown engine of the Metasploit Framework
- Slightly modified with a self-modification

Let’s try to figure out what this code does

\x6A\x0F\x59\xE8\xFF\xFF\xFF\xFF\xFF\xC1\x5E\x80
\x46\x0A\xE0\x30\x4C\x0E\x02\xF0
Linear disassembly can be easily tricked

Linear Disassembly

00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFF call 0x7
08  C15E8046   rcr [esi-0x80],0x46
0C  0AE0       or ah,al
0E  304C0E0B   xor [esi+ecx+0xb],cl
12  02FA       add bh,dl
14
... <encrypted shellcode>
93
Linear disassembly can be easily tricked

Linear Disassembly

00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFF  call 0x7
08  C15E8046  rcr [esi-0x80],0x46
0C  0AE0       or ah,al
0E  304C0E0B  xor [esi+ecx+0xb],cl
12  02FA       add bh,dl

... <encrypted shellcode>
93
Static Analysis Resistant Shellcode (3/4)

- Linear disassembly can be easily tricked

<table>
<thead>
<tr>
<th>Linear Disassembly</th>
<th>Recursive Traversal Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 6A7F push byte +0x7f</td>
<td>00 6A7F push byte +0xf</td>
</tr>
<tr>
<td>02 59 pop ecx</td>
<td>02 59 pop ecx</td>
</tr>
<tr>
<td>03 E8FFFFFF call 0x7</td>
<td>03 E8FFFFFF call 0x7</td>
</tr>
<tr>
<td>08 C15E8046 rcr [esi-0x80],0x46</td>
<td>07 FFC1 inc ecx</td>
</tr>
<tr>
<td>0C 0AE0 or ah,al</td>
<td>09 5E pop esi</td>
</tr>
<tr>
<td>0E 304C0E0B xor [esi+ecx+0xb],cl</td>
<td>0a 80460AE0 add [esi+0xa],0xe0</td>
</tr>
<tr>
<td>12 02FA add bh,dl</td>
<td>0e 304C0E0B xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>14</td>
<td>12 02FA add bh,dl</td>
</tr>
<tr>
<td>... &lt;encrypted shellcode&gt;</td>
<td>14 ... &lt;encrypted shellcode&gt;</td>
</tr>
<tr>
<td>93</td>
<td>93</td>
</tr>
</tbody>
</table>
Static Analysis Resistant Shellcode (3/4)

- Linear disassembly can be easily tricked

<table>
<thead>
<tr>
<th>Linear Disassembly</th>
<th>Recursive Traversal Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>00  6A7F</td>
<td>00  6A7F</td>
</tr>
<tr>
<td>02  59</td>
<td>02  59</td>
</tr>
<tr>
<td>03  E8FFFFFF</td>
<td>03  E8FFFFFF</td>
</tr>
<tr>
<td>08  C15E8046</td>
<td>07  FFC1</td>
</tr>
<tr>
<td>0C  0AE0</td>
<td>09  5E</td>
</tr>
<tr>
<td>0E  304C0E0B</td>
<td>0a  304C0E0B</td>
</tr>
<tr>
<td>12  02FA</td>
<td>0e  304C0E0B</td>
</tr>
<tr>
<td>14</td>
<td>12  02FA</td>
</tr>
<tr>
<td>... &lt;encrypted shellcode&gt;</td>
<td>... &lt;encrypted shellcode&gt;</td>
</tr>
<tr>
<td>93</td>
<td>93</td>
</tr>
</tbody>
</table>

- Recursive traversal disassembly is still not enough…

much better, but not the real code that will be eventually executed!
Self-modifying code can hide the real CFG

### Recursive Traversal Disassembly

<table>
<thead>
<tr>
<th>00</th>
<th>6A7F</th>
<th>push byte +0x7f</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>59</td>
<td>pop ecx</td>
</tr>
<tr>
<td>03</td>
<td>E8FFFFFFFFFF</td>
<td>call 0x7</td>
</tr>
<tr>
<td>07</td>
<td>FFC1</td>
<td>inc ecx</td>
</tr>
<tr>
<td>09</td>
<td>5E</td>
<td>pop esi</td>
</tr>
<tr>
<td>0a</td>
<td>80460AE0</td>
<td>add [esi+0xa],0xe0</td>
</tr>
<tr>
<td>0e</td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>12</td>
<td>02FA</td>
<td>add bh,dl</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... <encrypted shellcode>

### Real Code Execution

push byte +0x7f
Static Analysis Resistant Shellcode (4/4)

→ Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFF call 0x7
07  FFC1       inc ecx
09  5E         pop esi
0a  80460AE0   add [esi+0xa],0xe0
0e  304C0E0B   xor [esi+ecx+0xb],cl
12  02FA       add bh,dl
14... <encrypted shellcode>
93

Real Code Execution

push byte +0x7f
pop ecx
ecx = 0x7F
Static Analysis Resistant Shellcode (4/4)

Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>6A7F</td>
<td>push byte +0x7f</td>
</tr>
<tr>
<td>02</td>
<td>59</td>
<td>pop ecx</td>
</tr>
<tr>
<td>03</td>
<td>E8FFFFFFF</td>
<td>call 0x7</td>
</tr>
<tr>
<td>07</td>
<td>FFC1</td>
<td>inc ecx</td>
</tr>
<tr>
<td>09</td>
<td>5E</td>
<td>pop esi</td>
</tr>
<tr>
<td>0a</td>
<td>80460AE0</td>
<td>add [esi+0xa],0xe0</td>
</tr>
<tr>
<td>0e</td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>12</td>
<td>02FA</td>
<td>add bh,dl</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>&lt;encrypted shellcode&gt;</td>
</tr>
<tr>
<td>93</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Real Code Execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>push byte +0x7f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ecx = 0x7F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>call 0x7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(push 0x8)</td>
</tr>
</tbody>
</table>
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
<th>Real Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>6A7F</td>
<td>push byte +0x7f</td>
<td>push byte +0x7f</td>
</tr>
<tr>
<td>02</td>
<td>59</td>
<td>pop ecx</td>
<td>pop ecx</td>
</tr>
<tr>
<td>03</td>
<td>EFFFFFFF</td>
<td>call 0x7</td>
<td>ecx = 0x7F</td>
</tr>
<tr>
<td>07</td>
<td>FFC1</td>
<td>inc ecx</td>
<td>ecx = 0x7F</td>
</tr>
<tr>
<td>09</td>
<td>5E</td>
<td>pop esi</td>
<td>ecx = 0x7F</td>
</tr>
<tr>
<td>0a</td>
<td>80460AE0</td>
<td>add [esi+0xa],0xe0</td>
<td>push 0x8</td>
</tr>
<tr>
<td>0e</td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
<td>call 0x7</td>
</tr>
<tr>
<td>12</td>
<td>02FA</td>
<td>add bh,dl</td>
<td>ecx = 0x80</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>&lt;encrypted shellcode&gt;</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

push byte +0x7f
pop ecx
call 0x7
inc ecx
pop esi

Real Code Execution

push byte +0x7f
pop ecx
ecx = 0x7F
call 0x7
ecx = 0x80
inc ecx
esi = 0x80
pop esi
esi = 0x8

add [esi+0xa],0xe0
xor [esi+ecx+0xb],cl
add bh,dl
...
<encrypted shellcode>
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

Real Code Execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>6A7F</td>
<td>push byte +0x7f</td>
<td></td>
<td>6A7F</td>
<td>push byte +0x7f</td>
</tr>
<tr>
<td>02</td>
<td>59</td>
<td>pop ecx</td>
<td></td>
<td>59</td>
<td>pop ecx</td>
</tr>
<tr>
<td>03</td>
<td>E8FFFFFFF</td>
<td>call 0x7</td>
<td></td>
<td>E8FFFFFFF</td>
<td>call 0x7</td>
</tr>
<tr>
<td>07</td>
<td>FFC1</td>
<td>inc ecx</td>
<td></td>
<td>FFC1</td>
<td>inc ecx</td>
</tr>
<tr>
<td>09</td>
<td>5E</td>
<td>pop esi</td>
<td></td>
<td>5E</td>
<td>pop esi</td>
</tr>
<tr>
<td>0a</td>
<td>80460AE0</td>
<td>add [esi+0xa],0xe0</td>
<td></td>
<td>80460AE0</td>
<td>add [esi+0xa],0xe0</td>
</tr>
<tr>
<td>0e</td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
<td></td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>12</td>
<td>02FA</td>
<td>add bh,dl</td>
<td></td>
<td>02FA</td>
<td>add bh,dl</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFF call 0x7
07  FFC1       inc ecx
09  5E         pop esi
0a  80460AE0   add [esi+0xa],0xe0
0e  304C0E0B   xor [esi+ecx+0xb],cl
12  E2FA       loop 0xe
14
... <encrypted shellcode>
93

Real Code Execution

push byte +0x7f
pop ecx
ecx = 0x7F
(pop 0x8)
call 0x7
ecx = 0x80
inc ecx
pop esi
esi = 0x8
add [esi+0xa],0xe0
ADD [12] 0xE0

Self-modification

0x02FA + 0xEO = 0xE2FA
add bh,dl  loop 0xe
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>6A7F</td>
<td>push byte +0x7f</td>
</tr>
<tr>
<td>02</td>
<td>59</td>
<td>pop ecx</td>
</tr>
<tr>
<td>03</td>
<td>E8FFFFFFF</td>
<td>call 0x7</td>
</tr>
<tr>
<td>07</td>
<td>FFC1</td>
<td>inc ecx</td>
</tr>
<tr>
<td>09</td>
<td>5E</td>
<td>pop esi</td>
</tr>
<tr>
<td>0a</td>
<td>80460AE0</td>
<td>add [esi+0xa],0xe0</td>
</tr>
<tr>
<td>0e</td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>12</td>
<td>E2FA</td>
<td>loop 0xe</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Real Code Execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0e</td>
<td>304C0E0B</td>
<td>xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>93</td>
<td></td>
<td>XOR [93] 0x80</td>
</tr>
</tbody>
</table>
Static Analysis Resistant Shellcode (4/4)

→ Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

- push byte +0x7f
- pop ecx
- call 0x7
- inc ecx
- pop esi
- add [esi+0xa],0xe0
- xor [esi+ecx+0xb],cl
- loop 0xe

Real Code Execution

- push byte +0x7f
- pop ecx
- call 0x7
- inc ecx
- pop esi
- add [esi+0xa],0xe0
- xor [esi+ecx+0xb],cl
- loop 0xe

... <encrypted shellcode>

93
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFE call 0x7
07  FFC1       inc ecx
09  5E         pop esi
0a  80460AE0   add [esi+0xa],0xe0
0e  304C0E0B   xor [esi+ecx+0xb],cl
12  E2FA       loop 0xe
14  

... <encrypted shellcode>

Real Code Execution

push byte +0x7f
pop ecx ecx = 0x7F
call 0x7 (push 0x8)
inc ecx ecx = 0x80
pop esi esi = 0x8
add [esi+0xa],0xe0 ADD [12] 0xE0
xor [esi+ecx+0xb],cl XOR [93] 0x80
loop 0xe (ecx = 0x7F)
xor [esi+ecx+0xb],cl XOR [92] 0x7F
### Static Analysis Resistant Shellcode (4/4)

- **Self-modifying code can hide the real CFG**

<table>
<thead>
<tr>
<th>Recursive Traversal Disassembly</th>
<th>Real Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 6A7F push byte +0x7f</td>
<td>push byte +0x7f</td>
</tr>
<tr>
<td>02 59 pop ecx</td>
<td>pop ecx</td>
</tr>
<tr>
<td>03 E8FFFFFFFE call 0x7</td>
<td>call 0x7</td>
</tr>
<tr>
<td>07 FFC1 inc ecx</td>
<td>inc ecx</td>
</tr>
<tr>
<td>09 5E pop esi</td>
<td>pop esi</td>
</tr>
<tr>
<td>0a 80460AE0 add [esi+0xa],0xe0</td>
<td>add [esi+0xa],0xe0</td>
</tr>
<tr>
<td>0e 304C0E0B xor [esi+ecx+0xb],cl</td>
<td>xor [esi+ecx+0xb],cl</td>
</tr>
<tr>
<td>12 E2FA loop 0xe</td>
<td>loop 0xe</td>
</tr>
<tr>
<td>14</td>
<td>(ecx = 0x7F)</td>
</tr>
<tr>
<td>... &lt;encrypted shellcode&gt;</td>
<td>(push 0x8)</td>
</tr>
<tr>
<td>93</td>
<td>ecx = 0x80</td>
</tr>
<tr>
<td></td>
<td>esi = 0x8</td>
</tr>
<tr>
<td></td>
<td>ADD [12] 0xE0</td>
</tr>
<tr>
<td></td>
<td>XOR [93] 0x80</td>
</tr>
<tr>
<td></td>
<td>(ecx = 0x7F)</td>
</tr>
<tr>
<td></td>
<td>XOR [92] 0x7F</td>
</tr>
<tr>
<td></td>
<td>(ecx = 0x7E)</td>
</tr>
</tbody>
</table>

- **Real Code Execution**
  - push byte +0x7f
  - pop ecx
  - call 0x7
  - inc ecx
  - pop ecx
  - add [esi+0xa],0xe0
  - xor [esi+ecx+0xb],cl
  - loop 0xe
  - xor [esi+ecx+0xb],cl
  - loop 0xe
  - (ecx = 0x7F)
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

Real Code Execution

```
00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFF call 0x7
07  FFC1       inc ecx
09  5E         pop esi
0a  80460AE0   add [esi+0xa],0xe0
0e  304C0E0B   xor [esi+ecx+0xb],cl
12  E2FA       loop 0xe
14
... <encrypted shellcode>
93
```

```
push byte +0x7f
pop ecx
ecx = 0x7F
call 0x7
(push 0x8)
inc ecx
ecx = 0x80
pop esi
esi = 0x8
add [esi+0xa],0xe0
ADD [12] 0xE0
xor [esi+ecx+0xb],cl
XOR [93] 0x80
loop 0xe
(ecx = 0x7F)
xor [esi+ecx+0xb],cl
XOR [92] 0x7F
loop 0xe
(ecx = 0x7E)
xor [esi+ecx+0xb],cl
XOR [91] 0x7E
...```
Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

push byte +0x7f
pop ecx
call 0x7

inc ecx
pop esi
add [esi+0xa],0xe0
xor [esi+ecx+0xb],cl
add bh,dl

Real Code Execution

push byte +0x7f
pop ecx
call 0x7

inc ecx
pop esi
add [esi+0xa],0xe0
xor [esi+ecx+0xb],cl
loop 0xe
Attacks – Defenses Coevolution

Plain Shellcode

Simple Obfuscation

Polymorphism

SARS

String Signatures

Regexp Signatures

Static Analysis

?
Attacks – Defenses Coevolution

Plain Shellcode

- String Signatures

Simple Obfuscation

- Regexp Signatures

Polymorphism

- Static Analysis

SARS

- Emulation
Network-level Polymorphic Shellcode Detection

- **Motivation:** Highly obfuscated code will not reveal its actual form until it is executed

- **Main idea:** execute each TCP stream as if it was executable code

- **Goal:** identify the specific behavior inherent in polymorphic shellcodes
Is it possible to execute the shellcode using only information available at the network level?
- No access to the vulnerable host

The execution of a polymorphic shellcode consists of two sequential parts
1. Decryption
2. Actual shellcode execution

Focus on the decryption process
- Generic, independent of the exploit/vulnerability/OS
Network-level Emulation (2/2)

- Polymorphic shellcode engines (so far) produce \textit{self-contained} decryptor code
  - \textbf{Position-independent:} will run from any location in the vulnerable process’ address space
  - \textbf{Mandatory GetPC code:} for finding its absolute address in memory (x86 has no indirect memory addressing)
  - \textbf{Known operand values:} operands are initialized before use (encrypted payload size, decryption key, …)

- Can be executed using merely a CPU emulator
  - Without any host-level information
Detector

- **Input:** reassembled TCP streams or UDP packets
- **CPU emulator**
  - Randomized state before each new execution
- **We don’t know the starting position of the shellcode in the input stream**
  - Start execution from each byte
  - Performance optimization: skip NULL-byte-delimited regions smaller than 50 bytes
- **Execution Threshold**
  - Sometimes *endless* or *infinite* loops occur in random code
  - Dynamic infinite loop detection and squashing
Detection Heuristic

1. GetPC code (just a hint)
   - Execution of a suspicious instruction: `call, fstenv/fnstenv, fsave/fnsave`

2. Memory reads from *distinct* locations of the input buffer (*Payload reads*)
   - Low probability of payload reads in random data (~1KB vs 4GB)
   - For each execution, the buffer is mapped to a random location

![Diagram showing decryption process]
Evaluation: Correct Execution

- Off-the-shelf polymorphic shellcode engines
- Original shellcode is 128 bytes, 1000 mutations with each engine
- In all cases the shellcode is decrypted correctly
Evaluation: False Positives / Heuristic Tuning

- Benign traffic traces and 61GB of random data
  - More than 2 million streams

- Requiring the execution of some GetPC code followed by 7 or more payload reads gives zero false positives
Payload Reads for Complete Decryption

- Benign data: 1-6 accidental payload reads in extremely rare cases
- Polymorphic shellcodes: at least 32 payload reads for a conservatively small 128-byte shellcode
Payload Reads for Complete Decryption

- **Benign data**: 1-6 accidental payload reads in extremely rare cases
- **Polymorphic shellcode**: at least 32 payload reads for a conservatively small 128-byte shellcode

Choose the median

- Even more increased resilience to false positives
Higher XT → longer processing time per input → lower throughput

NULL-byte optimization not effective for port 80 (mostly ASCII data)
  - Could take advantage of other delimiters (CRLF, protocol framing)
Open Issues: Endless Loops

- Evasion by placing endless loops before/within the decryptor code
  - The execution threshold is reached before any sign of polymorphic behavior

- Endless loops occur in less than 5% of the benign traffic
  - Even if loops are used for evasion, useful as a first-level detector
  - Send all traffic reaching XT to a honeypot

- Infinite loop squashing provides some mitigation

- Can we do better than this? Can we also skip the execution of endless (but not infinite) loops?
  - The loop can compute something useful, like the decryption key
  - Static analysis strikes back?
Open Issues: Non-Polymorphic Shellcode

What about plain or completely metamorphic code?
- Does not decrypt its body
- No self modifications

Existing methods: search for exposed system calls, suspicious code sequences, ...

Shellcode “packing” is becoming essential
- Evasion!
- Avoidance of restricted bytes
Open Issues: Non-Self-Contained Code

- Although current polymorphic/encryption engines produce self-contained code, non-self-contained code is possible
- Take advantage of addresses with a priori known contents
  - e.g., initialize registers or jump to existing code
  - Should remain constant across all vulnerable systems (not always feasible)

```
shellcode
...
inc ecx
call 0x40038EF0
add [esi+0xa],0xe0
...
```

Augment the network-level detector with host-level information
  - e.g., invariant parts of the address space of each protected process

\[ \text{vulnerable process's address space} \]
\[ \text{cannot be followed at network-level} \]
Summary

- Pattern matching and static analysis are not enough
  - Highly polymorphic and self-modifying code

- Network-level emulation
  - Detects self-modifying polymorphic shellcode

- Preliminary experimental results are promising
  - Network-level detection of all known off-the-shelf polymorphic shellcode engines at 10-100 Mbps

- Open issues that have to be explored
Defending against Polymorphic Attacks: Recent Results and Open Questions

thank you!

Michalis Polychronakis
mikepo@ics.forth.gr

Institute of Computer Science
Foundation for Research and Technology – Hellas
Crete, Greece
fallback slides
Network-Level Emulation

GET /ind  ex.php HT  TP/1.1 Hos ...

GET /index.php HTTP/1.1 Host: www.foo.com ...

inc edi
inc ebp
push esp
and [edi],ch
imul ebp,[esi+0x64],dword 0x702e7865
push dword 0x54482070
push esp
push eax
das
xor [esi],ebp
xor [eax],esp
...

✓ benign request
Network-Level Emulation

\x6A\x0F\x59 \xE8\xFF\xFF \xFF\xFF\xC1 ...

\x6A\x0F\x59\xE8\xFF\xFF\xFF\xFF\xC1\x5E\x80...

6A07 59 E8FFFFFFFF FFC1 5E 80460AE0 304C0E0B E2FA ...

push byte +0x7f pop ecx call 0x7 inc ecx pop esi add [esi+0xa],0xe0 xor [esi+ecx+0xb],cl loop 0xe xor [esi+ecx+0xb],cl loop 0xe xor [esi+ecx+0xb],cl ...

✗ malicious request!
# Payload Reads vs Execution Threshold

![Graph showing payload reads vs execution threshold for different models: Alpha2, TAPiON, ADMmutate, Clet.](image)

- **Alpha2**: High payload reads at low execution thresholds, with a steep curve.
- **TAPiON**: Intermediary payload reads, slightly below Alpha2.
- **ADMmutate**: Lower payload reads compared to Alpha2, with a more gradual increase.
- **Clet**: Lowest payload reads among the models, showing a slow and steady increase.

---

**Legend**:
- Alpha2
- TAPiON
- ADMmutate
- Clet