A Survey of Obfuscations in Prevalent Packer Tools

Kevin Roundy
Paradyn Project

Paradyn / Dyninst Week
Madison, Wisconsin
March 26, 2012
Types of program analysis

Source code

Friendly binary

Uncooperative binary

Hostile binary

CFG

Function

Basic block

mov eax, *[ebp+8]
leave
ret

A Survey of Prevalent Obfuscations
Analysis building blocks

Analysis steps
1. Extract code bytes
2. Disassemble
3. Identify
4. Build comprehension tools
5. Patch/modify the code
6. Trace code’s execution

Toolkit:

Defensive-mode Dyninst
- Interactive debuggers
- Automated testing
  - Combinatorial testing
  - Code-coverage test generation
- Fault localization
  - Backwards slicing
  - Correlation of statement executions and test failures
- Vulnerability analysis
  - Taint analysis
  - Symbolic evaluation

A Survey of Prevalent Obfuscations
Analysis building blocks

Analysis steps
1. Extract code bytes
2. Disassemble
3. Identify
4. Build comprehension tools
5. Patch/modify the code
6. Trace code’s execution

Toolkit:

Defensive-mode Dyninst
- Interactive debuggers
- Automated testing
  - Combinatorial testing
  - Code-coverage test generation
- Fault localization
  - Backwards slicing
  - Correlation of statement executions and test failures
- Vulnerability analysis
  - Taint analysis
  - Symbolic evaluation
## Binary packing tools

<table>
<thead>
<tr>
<th>Packer</th>
<th>Malware market share*</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERALL</td>
<td>75%-80%</td>
</tr>
<tr>
<td>UPX</td>
<td>9.45%</td>
</tr>
<tr>
<td>PolyEnE</td>
<td>6.21%</td>
</tr>
<tr>
<td>PECompact</td>
<td>2.59%</td>
</tr>
<tr>
<td>Upack</td>
<td>2.08%</td>
</tr>
<tr>
<td>nPack</td>
<td>1.74%</td>
</tr>
<tr>
<td>ASPack</td>
<td>1.29%</td>
</tr>
<tr>
<td>FSG</td>
<td>1.26%</td>
</tr>
<tr>
<td>Nspack</td>
<td>0.89%</td>
</tr>
<tr>
<td>ASPProtect</td>
<td>0.43%</td>
</tr>
<tr>
<td>Armadillo</td>
<td>0.37%</td>
</tr>
<tr>
<td>Yoda’s Prot.</td>
<td>0.33%</td>
</tr>
<tr>
<td>WinUpack</td>
<td>0.17%</td>
</tr>
<tr>
<td>MEW</td>
<td>0.13%</td>
</tr>
</tbody>
</table>


- Open source cross-platform
- Fast, Small, Good
- Anti-reverse engineering

![Open source cross-platform](#)

![Fast, Small, Good](#)

![Anti-reverse engineering](#)
Outline

Analysis steps

1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

---

a. Code packing
b. Code overwriting
Code packing

A Survey of Prevalent Obfuscations
A Survey of Prevalent Obfuscations

Code overwriting

Code packing

Storm Worm

Aspack

malware

Upack

Entry Point

1B - 8KB

Entry Point
Outline

Analysis steps

1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

a. Unresolvable control flow
b. Call-stack tampering
c. Signals and exceptions
d. Ambiguous code & data
e. Disassembler fuzzing

A Survey of Prevalent Obfuscations
Unresolvable control flow

<table>
<thead>
<tr>
<th>invalid target</th>
<th>non-standard indirect</th>
<th>non-standard return</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>call 401000</code></td>
<td><code>jmp eax</code></td>
<td><code>push eax</code></td>
</tr>
<tr>
<td><code>Invalid Region</code></td>
<td><code>call ptr[eax]</code></td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>

Call-stack tampering

Storm Worm

Address 40d002 03 04 05 06 07 08 09 0a 0b 0c 0d
Bytes e8 03 00 00 00 e9 eb 04 5d 45 55 c3

CALL 40d00a

JMP 459dd4f7
Exception-based control flow

Ambiguous code and data

- Bytes after call instructions
- Junk after exception-raising instruction
- In-place decryption of unpacked code
Outline

Analysis steps

1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

a. Missing call/ret instructions
b. Extra call/ret instructions
c. Overlapping functions
d. Overlapping basic blocks
Outline

Analysis steps

1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

a. Missing call/ret instructions
b. Extra call/ret instructions
c. Overlapping functions
d. Overlapping basic blocks
Extra call/ret instructions

- `push <targ>`
- `ret`
- `jmp <targ>`

Diagram:

```
push <targ>  jmp <targ>  call <targ>
      |     |     |
      |     |     V
      mov edi,esi
      pop ebp
```

```
      |
      V
call <targ>
      |
      V
mov edi,esi
pop ebp
```

```
      |
      V
call <targ>
      |
      V
jmp <targ2>
pop esi
```
Overlapping functions
Overlapping functions
Overlapping blocks

Address

Block 1
mov eax, ebb907eb
seto bl
or ch, bh
jmp 45402e

Block 2
jmp 45402c

Block 3
jmp 454028

Block 3

Address

Bytes

Block 1
0x454017
b8 eb 07 b9 eb 0f 90 eb 08 fd eb 0b

Block 2
0x45401b

Block 3
0x45401e

Overlapping blocks
### Overlapping blocks

<table>
<thead>
<tr>
<th>Address</th>
<th>454017</th>
<th>18</th>
<th>19</th>
<th>1a</th>
<th>1b</th>
<th>1c</th>
<th>1d</th>
<th>1e</th>
<th>1f</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>e8</td>
<td>03</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>e9</td>
<td>eb</td>
<td>04</td>
<td>5d</td>
<td>45</td>
<td>55</td>
<td>c3</td>
</tr>
<tr>
<td>Block 1</td>
<td>mov eax, ebb907eb</td>
<td>seto bl</td>
<td>or ch,bh</td>
<td>jmp 45402e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 2</td>
<td></td>
<td>jmp 45402c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 3</td>
<td></td>
<td>jmp 454028</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

Analysis steps
1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

- Obfuscated constants
- ABI violations
- Do-nothing code
Outline

Analysis steps
1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

a. Self-checksumming
b. Stolen bytes
c. Anti-unpacking
Self-checksumming

A Survey of Prevalent Obfuscations

Bootstrap code

Payload code

Binary file

Checksum routine

xor eax, eax

add eax, ptr[ebx]
add 4, ebx
cmp ebx, 0x41000
jne .loop

cmp eax, .checksum
jne .fail

pass
fail

process
Stolen bytes

... call ptr [IAT-entry]

Import Address Table
malware.exe

.kloadlibrary
mov edi, edi
push ebp
mov ebp, esp
cmp ptr[ebp+8], 0

... kernel32.dll

... call buffer.stolen

Import Address Table
malware.asprotect.exe

.kloadlibrary
mov edi, edi
push ebp
mov ebp, esp
cmp ptr[ebp+8], 0

... kernel32.dll

... stolen
mov edi, edi
push ebp
mov ebp, esp
cmp ptr[ebp+8], 0

.buffer
Outline

Analysis steps

1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution

- Stolen bytes
- Non-standard API calls
- Anti-debugging
Outline

Analysis steps
1. Extract code bytes
2. Disassemble
3. Identify functions
4. Build comprehension tools
5. Patch/modify the binary
6. Trace code’s execution
Adapting Dyninst for Malware

Analysis tool
Mutator

Dyninst
Control flow analyzer
Data flow analyzer
Instrumenter

program binary
CFG
Adapting Dyninst for Malware

Analysis tool
Mutator

SR-Dyninst

static-dynamic analysis
Control flow analyzer
Data flow analyzer
Sensitivity Resistant Instrumenter

A Survey of Prevalent Obfuscations