Advanced Software Armoring and Polymorphic Kung-Fu

Nick Harbour
Outline

- Executable Packers at a Glance
- Introducing PE-Scrambler
- Challenges and Headaches
- Armoring Techniques
  - Function Call Dispatching
  - Code Chunking
  - Anti-Disassembly Tricks
- Polymorphic Code Replacement
- Malware Detection by Disassembly

Advanced Software Armoring and Polymorphic Kung-Fu
Executable Packers

- The Purpose of Packers
  - Compression
  - Protection
    - String Hiding
    - Anti-Reverse Engineering
    - Anti-Anti-Virus

- Frequently a Tradeoff Between Compression and Protection
Compression vs. Armor

Compression
- UPX
- FSG
- PE-Compact

Armor
- PE-Lock
- Themida
- VMProtect
Traditional Packer Design

- Code and Data are Compressed or Encrypted
- A Small Unpacker Stub is inserted
- The Unpacker Stub runs first and Decompresses or Decrypts the original payload
- The Unpacker Stub “Jumps” to the original entry point
Traditional Packer Design

Original Binary

- PE Headers
- .text
- .data
- .rsrc

Packed Binary

- PE Headers
- UPX0
- UPX1
- .rsrc

Advanced Software Armoring and Polymorphic Kung-Fu
Traditional Packer Design

Packed Memory

- PE Headers
- UPX0
- UPX1
- .rsrc

Unpacked Memory

- PE Headers
- UPX0
- UPX1
- .rsrc

Unpacking

Original Entry Point

Advanced Software Armoring and Polymorphic Kung-Fu
Traditional Limitations

- Usually Easy to Defeat
- Code Unpacks Itself
- Once Unpacked, Original Code Appears Pristine and Unmodified
More advanced armoring involves permanently modifying the compiled code.

Tools such as *Themida* and *VMPProtect* translate x86 instructions into a custom byte code.
Introducing PE-Scrambler

- PE-Scrambler manipulates compiled code to defeat or delay Reverse Engineering
- Operates by Disassembling
- Current Features:
  - Function Call Hijacking
  - Code Chunking
  - Anti-RE Wrapping and Weaving
  - Polymorphic Code Substitution

Advanced Software Armoring and Polymorphic Kung-Fu
Challenges and Headaches

- No margin of error for bad disassembly
- It is often difficult to tell code from data
  - Almost any data can at least partially disassemble into valid x86 instructions
- Disassembler must be pessimistic
- Needs full cross-referencing
Function Call Dispatching

- PE-Scrambler inserts a 65-byte function which dispatches all function calls within the program.

- Finds all internal and external function calls and redirects them to a single Function Call Dispatcher.

- The Function Call Dispatcher looks up the intended target based on the return pointer.
Function Call Dispatching

Original Call Tree

A

B

C

D

E

F

Call Tree with Dispatcher

A

B

C

Dispatcher

B

C

D

E

F

Advanced Software Armoring and Polymorphic Kung-Fu
printf(“Hello”);
printf(“World!”);

Compiled:
00401000: PUSH OFFSET “Hello”
00401005: CALL printf
0040100B: ADD esp, 4
0040100F: PUSH OFFSET “World!”
00401015: CALL printf

Instructions:
00401000: PUSH OFFSET “Hello”
00401005: CALL printf
0040100B: ADD esp, 4
0040100F: PUSH OFFSET “World!”
00401015: CALL printf

Stack:
0040101B “World”
Function Dispatch Table

- PE-Scrambler builds a Lookup Table
  - Return Address
  - Intended Target
- The Dispatcher inspects the return address and jumps to the appropriate target

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>401050</td>
<td>socket</td>
</tr>
<tr>
<td>40107B</td>
<td>printf</td>
</tr>
<tr>
<td>401253</td>
<td>accept</td>
</tr>
<tr>
<td>40125A</td>
<td></td>
</tr>
<tr>
<td>401278</td>
<td></td>
</tr>
<tr>
<td>40129F</td>
<td></td>
</tr>
<tr>
<td>40443E</td>
<td></td>
</tr>
</tbody>
</table>

Advanced Software Armoring and Polymorphic Kung-Fu
Code Chunking

- The ability to find and relocate portions of code
- Obscures the control flow of the program
- Opens the door to more advanced armoring techniques
Scrambled Functions

Scrambled

Advanced Software Armoring and Polymorphic Kung-Fu
Relocation Techniques

- Relocate with a simple jump
- Find instructions that produce predictable flag register states and relocate “conditionally”
- Insert conditional code and jump conditionally
**Simple Jump Relocation**

00401000: PUSH ebp
00401001: MOV ebp, esp
00401003: CALL sub_4010D8
00401008: TEST eax, eax
0040100A: JZ loc_401014
0040100C: ADD eax, [esp+8]
00401010: MOV [esp+10h], eax
00401014: MOV esp, ebp
00401016: POP ebp
00401017: RET

00402000: TEST eax, eax
00402002: JZ loc_401014
00402004: ADD eax, [esp+8]
00402008: MOV [esp+10h], eax
0040200C: MOV esp, ebp
0040200E: POP ebp
0040200F: RET
Known Conditional Reloc.

- Find an instruction guaranteed to set the flag to a certain value
- Insert a conditional jump immediately after

- A Conditional Jump on a guaranteed condition is a really unconditional, but can fool disassemblers
Known Conditional Reloc.

XOR eax, eax always sets the Zero Flag to 1

Advanced Software Armoring and Polymorphic Kung-Fu
Inserting Conditionals

- Attempt to trip up automated analysis

- Insert conditional tests which are predictable to a human, but not to the machine.
A human may know that esp is always less than 80000000h, but a disassembler? Not so much.
Producing Conflicting Disassembly

- When we divert code with a conditional jump, we can add an unconditional jump which causes the disassembly to follow the wrong branch.
- In the event of a conflict, most disassemblers trust the false conditional branch.
- Most disassemblers trust only their first interpretation.
Producng Conflicting Disassembly

- The same sequence of bytes produces two conflicting disassemblies.
- Disassembly at a slightly different offset with a valid opcode "hides" the real code.

Advanced Software Armoring and Polymorphic Kung-Fu
Producing Conflicting Disassembly

This produces the **CALL** instruction which conflicts with the truth branch

IDA Pro disassembles the false branch first

```
00401000: PUSH ebp
00401001: MOV ebp, esp
00401003: XOR eax, eax
00401008: CMP esp, 80000000h
0040100E: JLE loc_402001
00401014: JMP loc_402000
```
Impossible Disassembly

- JMP instruction jumps into itself
- FF byte is part of 2 instructions.

JMP -1

EB FF C0 48

INC eax DEC eax

Advanced Software Armoring and Polymorphic Kung-Fu
Impossible Disassembly

- JZ points to the operand to the MOV instruction
- IDA Pro will continue disassembly at the fake E8 byte

66 B8 EB 05 31 C0 74 F9 E8 Real Code Here

MOV ax, 05EBh
XOR eax, eax
JZ -7
JMP 5
Polymorphic Replacement

- PE-Scrambler seeks out small algorithms or instructions which may be easily substituted by alternative instructions
- Can insert useless code where able
- Substitutions may occur randomly
- Provides the ability to randomize a binary on the fly
Polymorphic Examples

PUSH eax
SUB esp, 4
MOV [esp], eax

CALL sub_402000
PUSH 401005
JMP sub_402000

PUSH 42h
CALL sub_402000
PUSH eax
ADD esp, 4
CALL sub_402000

Advanced Software Armoring and Polymorphic Kung-Fu
Polymorphic Examples

PUSH ebp
MOV esp, ebp
SUB 8, esp

ENTER 8, 0

JMP loc_402000

PUSH loc_402000
RET

TEST eax, eax
JZ loc_402000

CMP eax, 0
JNZ loc_401015
JMP loc_402000
loc_401015:
PE-Scrambler Demo
Malware Detection by Disassembly

- Find Evil is a tool to detect packed malware binaries.
- Uses the same disassembly engine as PE-Scrambler.
- Measures the amount of disassembly in relation to the size of the binary.
- Relies on various ratios to spot packed binaries.
Thank You

Tools Available at
www.nickharbour.com

nickharbour@gmail.com