Advanced Software Armoring and Polymorphic Kung-Fu

Nick Harbour

Copyright © 2008 Nick Harbour, All Rights Reserved

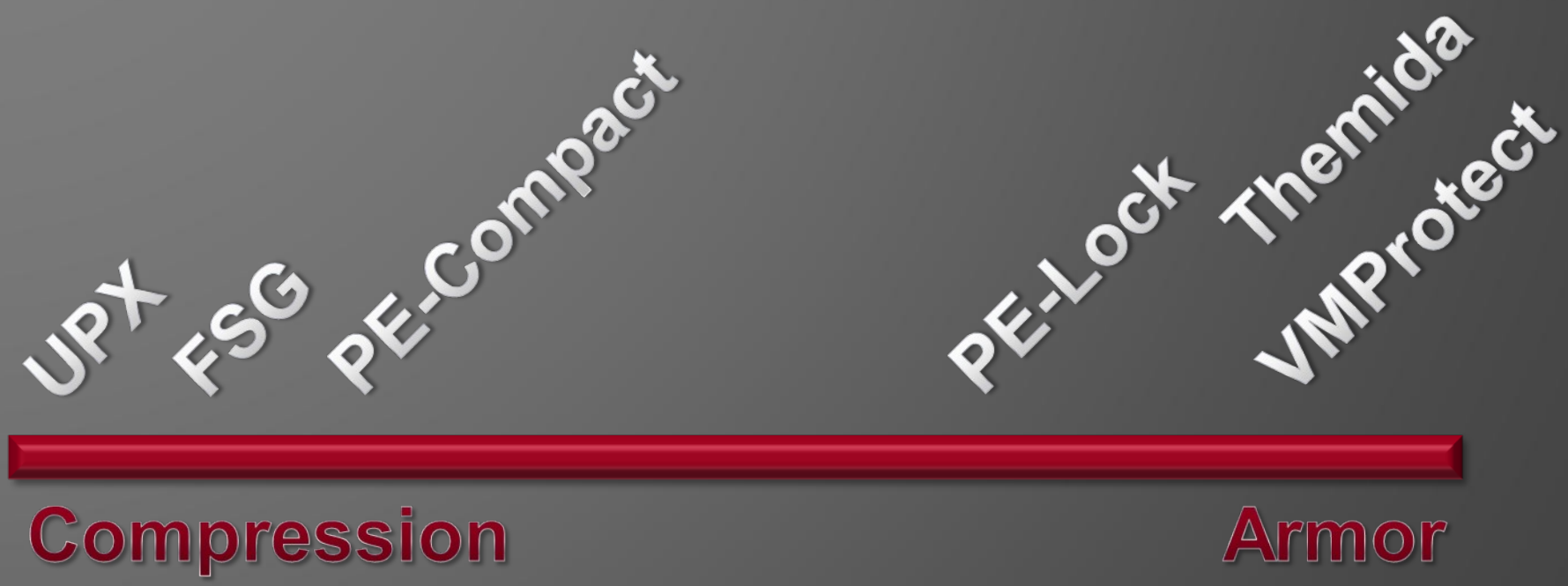
Outline

- 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

Executable Packers

- The Purpose of Packers
  - Compression
  - Protection
    - String Hiding
    - Anti-Reverse Engineering
    - Anti-Anti-Virus
- Frequently a Tradeoff Between Compression and Protection

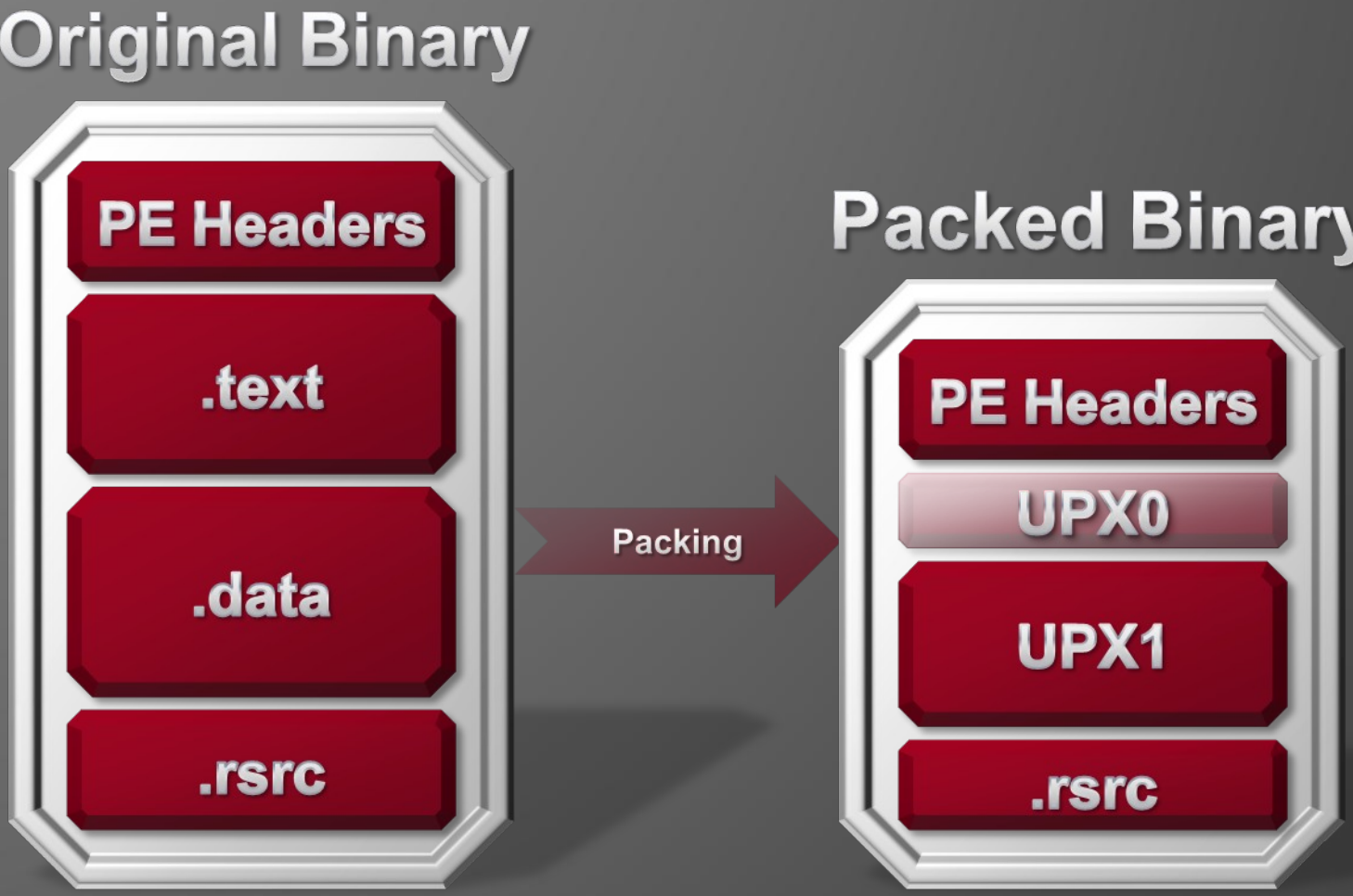
Compression vs. Armor



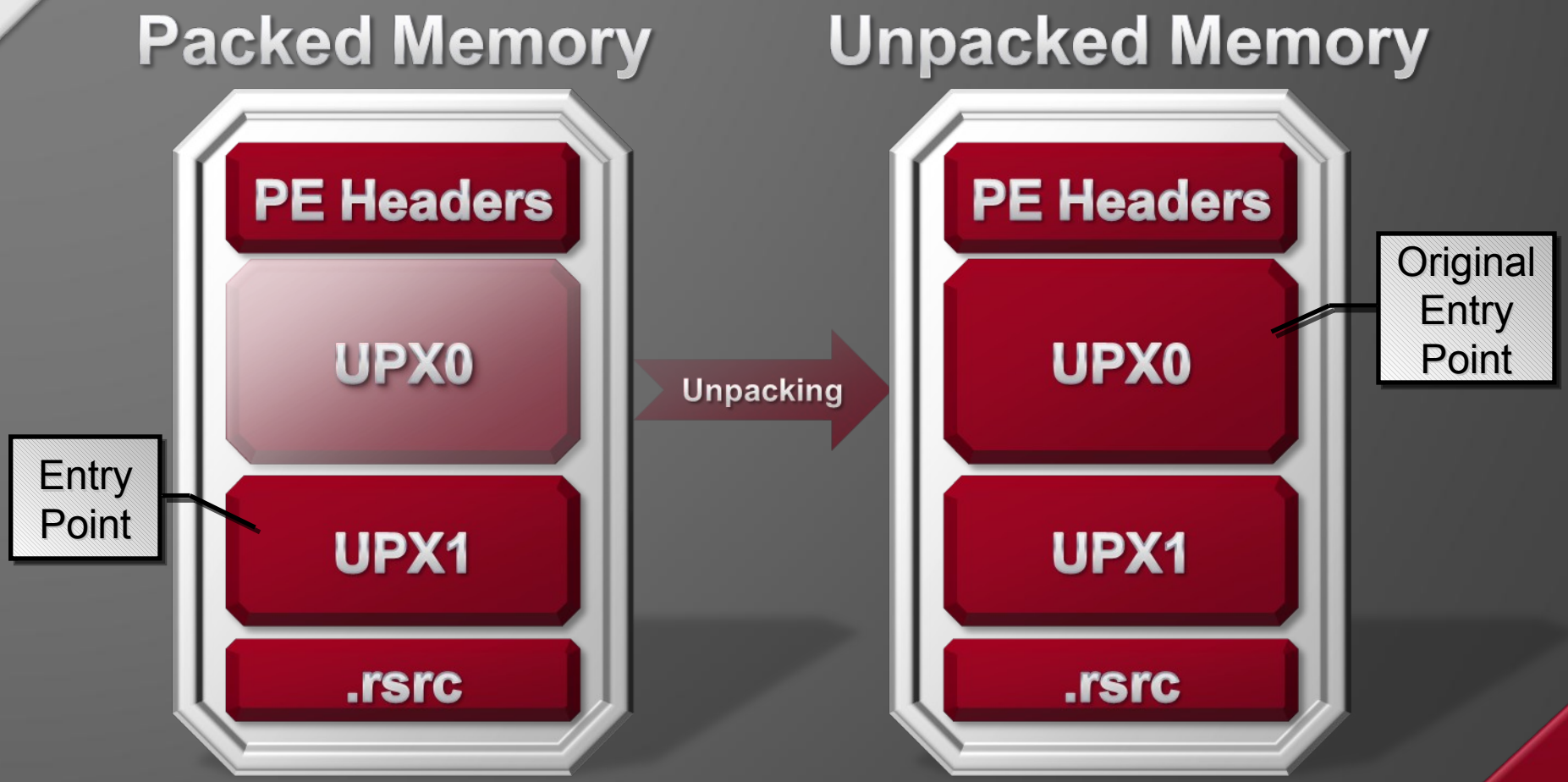
Traditional Packer Design

- 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



Traditional Packer Design



Traditional Limitations

- Usually Easy to Defeat
- Code Unpacks Itself
- Once Unpacked, Original Code Appears Pristine and Unmodified

Kickin' It Up A Notch

- More advanced armoring involves permanently modifying the compiled code
- Tools such as *Themida* and *VMProtect* 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

Challenges and Headaches

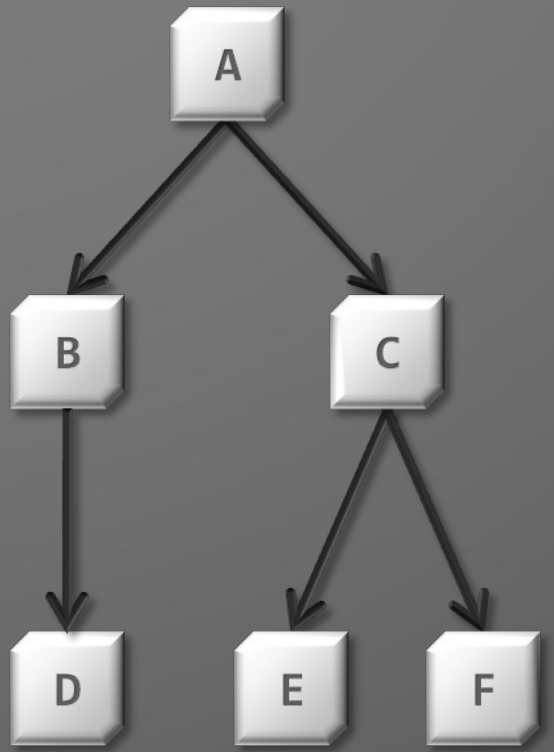
- No margin of error for bad disassembly
- It is often difficult to tell code from data
  - Almost any data can at least partially disassemble into valid x86 instructions
- Disassembler must be pessimistic
- Needs full cross-referencing

Function Call Dispatching

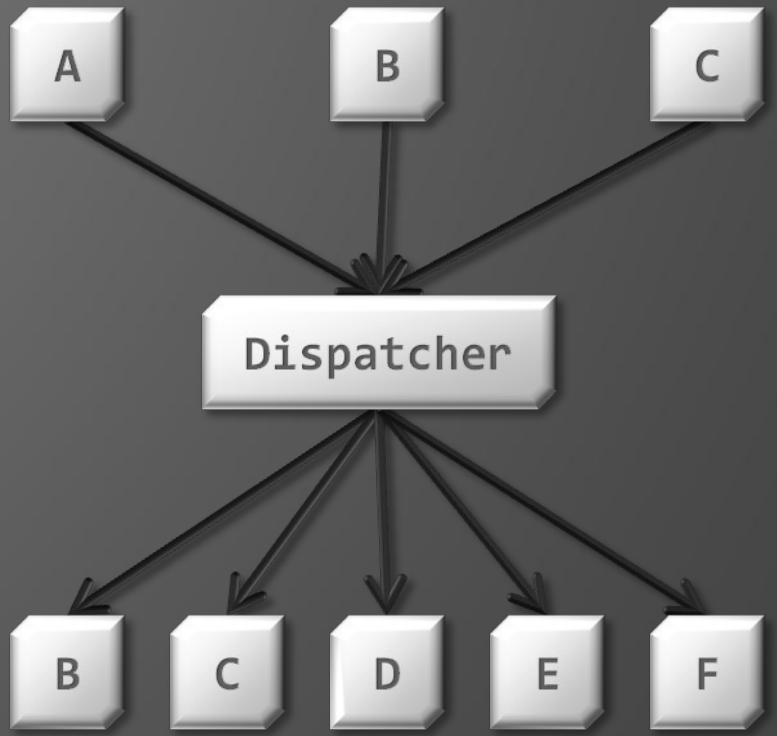
- PE-Scrambler inserts a 65-byte function which dispatches all function calls within the program.
- Finds all internal and external function calls and redirects them to a single Function Call Dispatcher
- The Function Call Dispatcher looks up the intended target based on the return pointer

Function Call Dispatching

Original Call Tree



Call Tree with Dispatcher



```
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

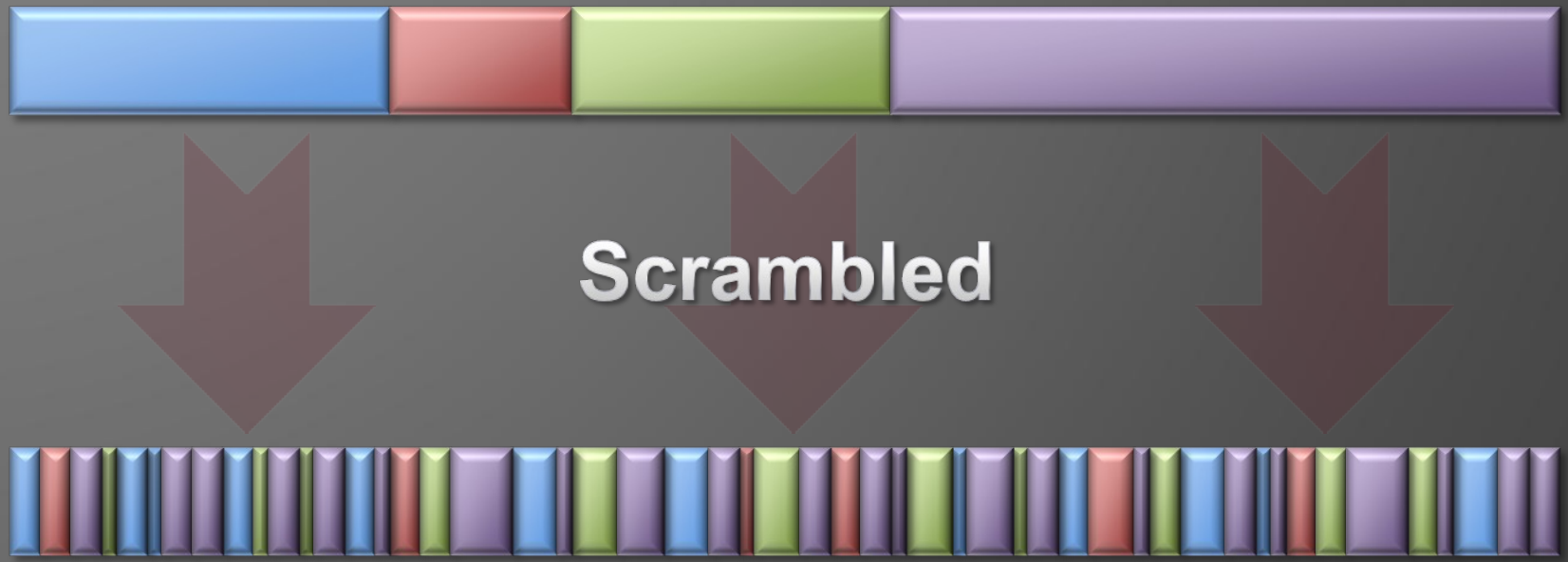
- PE-Scrambler builds a Lookup Table
  - Return Address
  - Intended Target
- The Dispatcher inspects the return address and jumps to the appropriate target

From	To
401050	socket
40107B	printf
401253	401290
40125A	accept
401278	401079
40129F	40443E

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



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
```

```
00401000: PUSH ebp
00401001: MOV  ebp, esp
00401003: CALL sub_4010D8
00401008: JMP  loc_402000
```



```
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.

```
00401000: PUSH ebp
00401001: MOV  ebp, esp
00401003: XOR  eax, eax
00401008: MOV  [esp-4], eax
0040100A: MOV  [esp-8], eax
0040100C: PUSH eax
00401010: PUSH eax
00401014: PUSH 42
00401016: CALL [0x401200]
```

```
00401000: PUSH ebp
00401001: MOV  ebp, esp
00401003: XOR  eax, eax
00401008: JZ   loc_402000
0040100A: ;Fake Code Goes Here
```



```
00402000: MOV  [esp-4], eax
00402004: MOV  [esp-8], eax
00402008: PUSH eax
00402009: PUSH eax
0040200A: PUSH 42
0040200F: CALL [0x401200]
```

XOR eax, eax always sets the Zero Flag to 1

Inserting Conditionals

- Attempt to trip up automated analysis
- Insert conditional tests which are predictable to a human, but not to the machine.

Inserting Conditionals

```
00401000: PUSH ebp
00401001: MOV  ebp, esp
00401003: XOR  eax, eax
00401008: MOV  [esp-4], eax
0040100A: MOV  [esp-8], eax
0040100C: PUSH eax
00401010: PUSH eax
00401014: PUSH 42
00401016: CALL [0x401200]
```

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



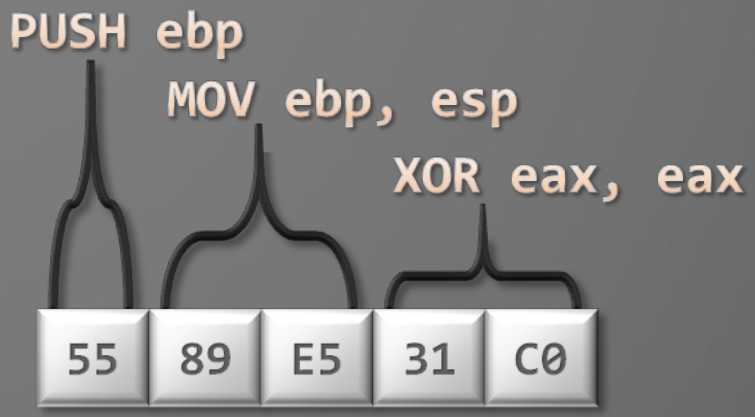
```
00402000: MOV  [esp-4], eax
00402004: MOV  [esp-8], eax
00402008: PUSH eax
00402009: PUSH eax
0040200A: PUSH 42
0040200F: CALL [0x401200]
```

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

Producing 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



Producing Conflicting Disassembly

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

- IDA Pro disassembles the false branch first

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



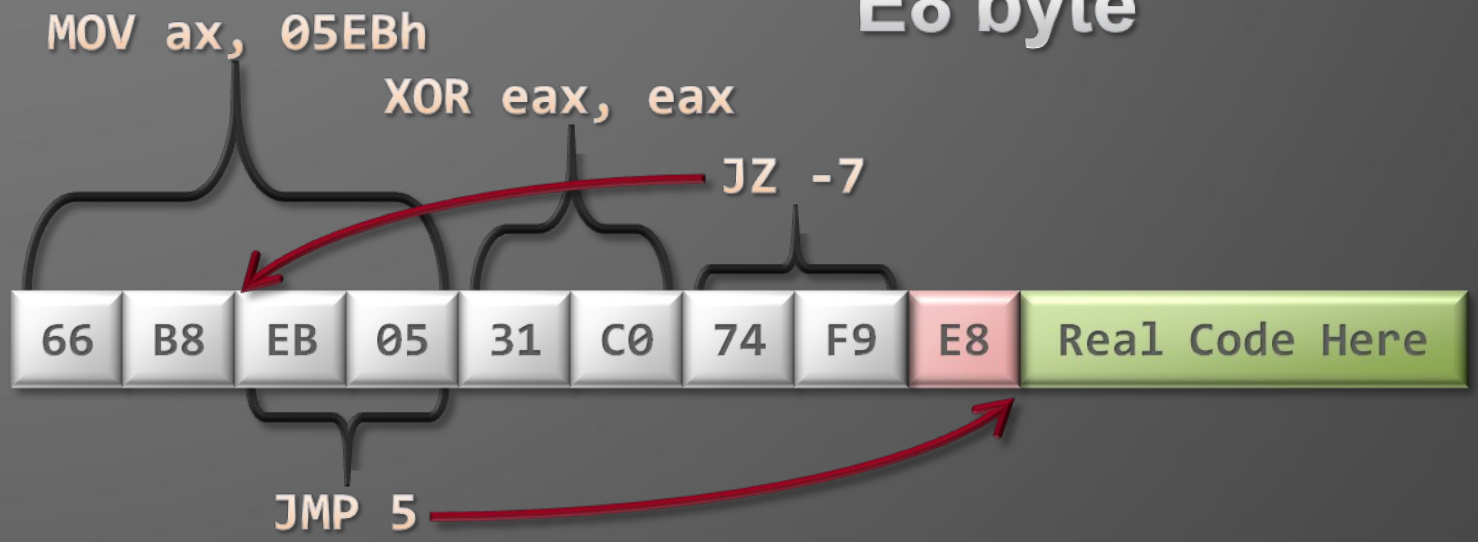
Impossible Disassembly

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



Impossible Disassembly

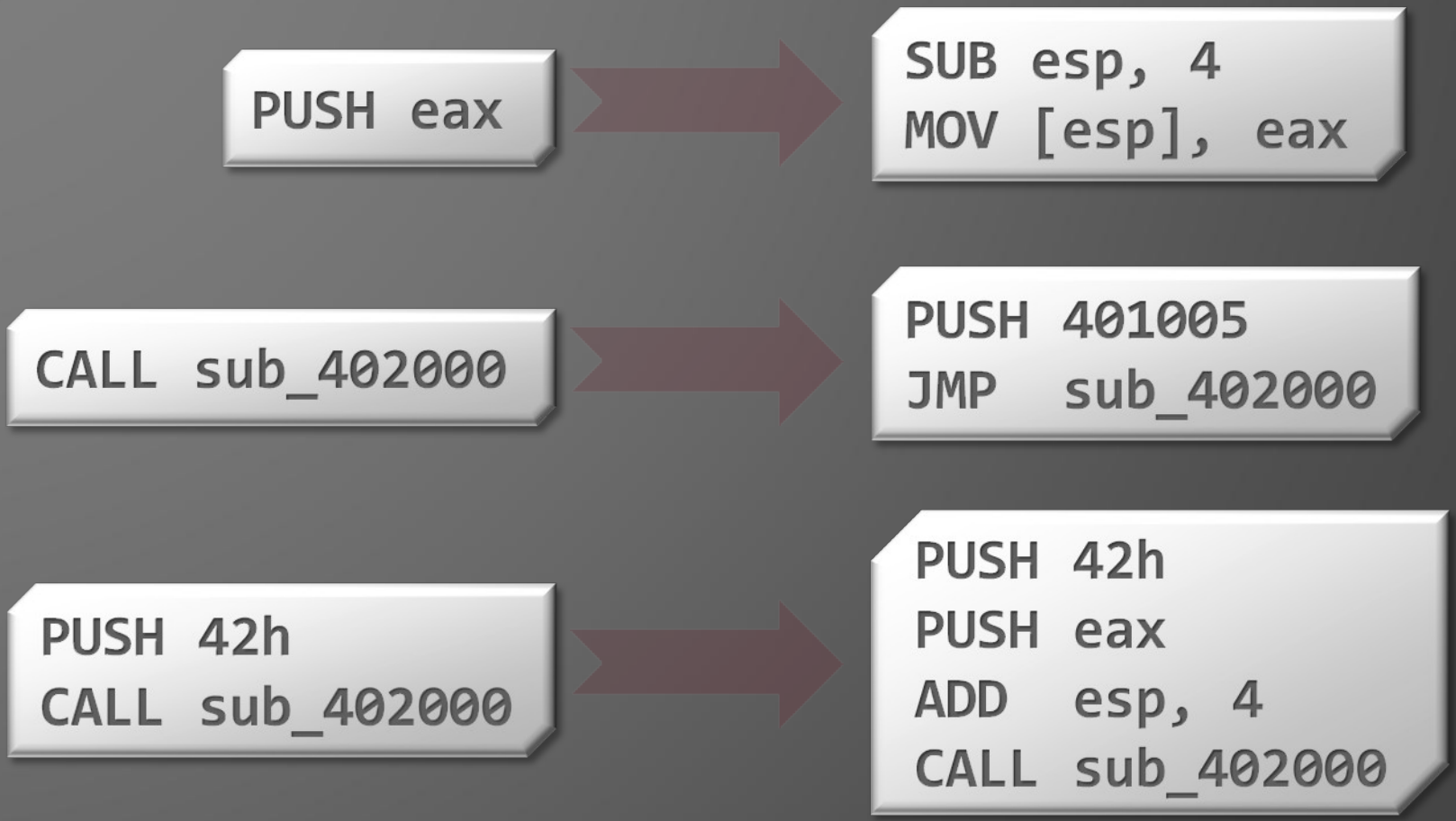
- JZ points to the operand to the MOV instruction
- IDA Pro will continue disassembly at the fake E8 byte



Polymorphic Replacement

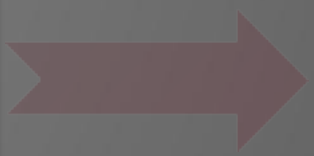
- PE-Scrambler seeks out small algorithms or instructions which may be easily substituted by alternative instructions
- Can insert useless code where able
- Substitutions may occur randomly
- Provides the ability to randomize a binary on the fly

Polymorphic Examples



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

- FindEvil 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

FindEvil Demo

File Options Help

Filename	Size	Reported Code Size	Executable Sections Size	Disasm Size	File to Code Ratio	Code to Disasm Ratio	Notes	Full Path
doWTP_RESTO...	300032	294912	299008	0	98.29%	0%		C:\Documents and
doWTP_RESTO...	784896	624640	624640	10000	79.58%	100%	(Large Amount of Code Detected)	C:\Documents and
ravmon_unpacke...	68096	32768	30352	10000	48.12%	100%	(Large Amount of Code Detected)	C:\Documents and
game.exe	432640	384000	384000	10000	88.76%	100%	(Large Amount of Code Detected)	C:\Documents and
ptanks.exe	28672	11264	11264	9994	39.29%	88.73%		C:\Documents and
Tanks.exe	6732826	45056	45056	0	0.67%	0%		C:\Documents and
Tanks_unpacke...	6778394	75264	75264	10000	1.11%	100%	(Large Amount of Code Detected)	C:\Documents and
0.ex_	116871	0	114176	297	0%	100%	(Large Resources Section Detec...	C:\Documents and
0_unpacked.ex_	270336	0	114176	4	0%	100%	(Large Resources Section Detec...	C:\Documents and
autorun.ex_	44544	8192	43520	7920	18.39%	96.68%		C:\Documents and
crsvc.ex_	49152	36864	36864	10000	75%	100%	(Large Amount of Code Detected)	C:\Documents and
doWTP_RESTO...	300032	294912	299008	0	98.29%	0%		C:\Documents and
doWTP_RESTO...	784896	624640	624640	10000	79.58%	100%	(Large Amount of Code Detected)	C:\Documents and
explorer.ex_	9728	6656	6656	3554	68.42%	53.4%		C:\Documents and

Welcome to FindEvil!



Thank You

Tools Available at
www.nickharbour.com

nickharbour@gmail.com

Copyright © 2008 Nick Harbour, All Rights Reserved