Deobfuscator:

An Automated Approach to the Identification and Removal of Code Obfuscation

Eric Laspe, Reverse Engineer
Jason Raber, Lead Reverse Engineer
Overview

• The Problem: Obfuscation
• Malware Example: RustockB
• The Solution: Deobfuscator
• Demonstration
• RustockB: Before & After
• Sample Source Code
• Summary
Overview

• The Problem: Obfuscation
• Malware Example: RustockB
• The Solution: Deobfuscator
• Demonstration
• RustockB: Before & After
• Sample Source Code
• Summary
The Problem: Obfuscated Code

- Malware authors use code obfuscation techniques to hide their malicious code.
- Obfuscation costs reverse engineers time:
  - Complicates instruction sequences
  - Disrupts control flow
  - Makes algorithms difficult to understand
- Manual obfuscation removal is a tedious and error-prone process.
Example: PUSH_POP_MATH

**PUSH** an immediate, then **POP** into a register and do some math on it

Obfuscated code:

```
00401064  push  0E39A3CC0h
00401069  pop  edx
0040106A  xor  edx, 0E3DA2CBBh
00401070  jmp  edx
```

Resolves to:

```
00401064  mov  edx, offset byte_40107B
00401069  [nop]
0040106A  [nop]
0040106B  [nop]
0040106C  [nop]
0040106D  [nop]
0040106E  [nop]
0040106F  [nop]
00401070  jmp  edx
```

Black Hat 2008
Overview

• The Problem: Obfuscation
• Malware Example: RustockB
• The Solution: Deobfuscator
• Demonstration
• RustockB: Before & After
• Sample Source Code
• Summary

Black Hat 2008
Malware Example: RustockB

• Good malware example that implemented obfuscation patterns to hide a decryption routine
• Many useless and confusing instructions
  - Push regs, math, pop regs
  - Pushes and pops in various obfuscated forms
• Control flow obscured
  - Mangled jumps
  - Unnecessary data cross-references

Black Hat 2008
RustockB Control Flow

Black Hat 2008
RustockB Control Flow

Black Hat 2008
Overview

• The Problem: Obfuscation
• Malware Example: RustockB
• The Solution: Deobfuscator
• Demonstration
• RustockB: Before & After
• Sample Source Code
• Summary
The Solution: The Deobfuscator IDA Pro Plug-in

• Combines instruction emulation and pattern recognition
• Determines proper code control flow
• Interprets and transforms instruction sequences to enhance code readability
• Uses a binary injector to make both static and dynamic analysis easier
The plug-in has six modes:

- **Anti-disassembly** - replaces anti-disassembly with simplified code
- **Passive** - simple peep-hole rules
- **Aggressive** - uses aggressive assumptions about memory contents
- **Ultra** - more aggressive assumptions
- **Remove NOPs** - jumps over slack space
- **Collapse** - moves consecutive code blocks together to eliminate NOPs and JMPs
• Deobfuscator plug-in invoked with Alt-Z
• Uses structures created by IDA Pro disassembly and analysis
• Depending on the mode selected, it can:
  - Follow jumps and calls
  - Track registers and emulate the stack
Overview

• The Problem: Obfuscation
• Malware Example: RustockB
• The Solution: Deobfuscator
• Demonstration
• RustockB: Before & After
• Sample Source Code
• Summary
Demonstration

- Demo code protected with anti-disassembly code and obfuscation
- Note the obfuscated jump at the end of this graph
- Run iteratively, the Deobfuscator will remove obfuscation and improve code flow readability

Black Hat 2008
Run 1 - Anti-Disassembly

- Two matching patterns
  - JZ_JMP
  - CALL_MATH
Pattern:  JZ_JMP

Two useless jumps

Before Deobfuscation:

```
00401022 jz short loc_401027
00401024 jmp short loc_401028
00401024 ;
00401026 db 0C7h
00401027 ;
00401027 loc_401027:
00401027 ; CODE XREF: start+11↑j
00401027 nop
00401028
00401028 loc_401028:
00401028 ; CODE XREF: start+10↑j
00401028 xor edx, 131087D0h
```

After Deobfuscation:

```
00401022  nop
00401023  nop
00401024  nop
00401025  nop
00401026  nop
00401027  nop
00401028  xor edx, 131087D0h
```

Black Hat 2008
EDX gets the return address of the CALL $5
Then, there is some math on EDX

Before Deobfuscation:

```
0040102E    call $+5
00401033    pop edx
00401034    xor edx, 0F19h
0040103A    xor ecx, 690A09D8h
```

EDX = 401033

After Deobfuscation:

```
0040102E    mov edx, 0F1D71B16h
00401033    nop
00401034    nop
00401035    nop
00401036    nop
00401037    nop
00401038    nop
00401039    nop
0040103A    xor ecx, 690A09D8h
```

Emulated Result

NOP’d Pop & Math

Black Hat 2008
Output Injection

- A text file is generated by the Deobfuscator plug-in
- Then, we inject the binary with a PERL script

Or just modify the IDA Pro database
• Now, we see the obfuscated code begin to disappear
• The Deobfuscator replaces obfuscation patterns and injects NOPs over useless code to create slack space
Slack Space

- Slack space is useful for patterns that need additional bytes to create a simplified instruction

- Example:

  Obfuscated Code
  ```
  PUSH EAX
  *NOP
  NOP
  NOP
  NOP
  NOP
  POP EBX
  ```

  Transformed Code 1
  ```
  MOV EBX, EAX
  NOP
  NOP
  NOP
  NOP
  ```

  Transformed Code 2
  ```
  MOV EBX, IMMED
  NOP
  ```

*Code that was removed by an earlier run of the Deobfuscator*

Black Hat 2008
Run 2 - Passive, Aggressive, & Ultra

- Three matching patterns
  - MOV_MATH
  - MATH_MOV_OR_POP
  - MATH_MOV_OR_POP

---

Begin Deobfuscation
---

Passive, Aggressive, & Ultra 7
Opened jmp.txt
Opened math.txt
Opened ultra.txt
start_addr: 401008  end_addr: 401070
Begin unref
Done unref 0

401009 mov_math
401028 useless_Code_08::math_mov_or_pop
401040 Useless_Code_08::math_mov_or_pop

Black Hat 2008
Move an immediate into **EAX** and **XOR** it with another known register value

**Before Deobfuscation:**

```
00401009  mov eax, dword_402009
0040100E  mov ecx, dword_40200D
00401014  mov edx, dword_402011
0040101A  mov ebx, dword_402015
00401020  xor eax, ebx
```

**After Deobfuscation:**

```
00401009  mov eax, 083769346h
0040100E  mov ecx, dword_40200D
00401014  mov edx, dword_402011
0040101A  mov ebx, dword_402015
00401020  {nop}
00401021  {nop}
```

**Emulated Result**

**NOP’d Math**
Pattern: MATH_MOV_OR_POP

Do math on **EDX**, then **MOV** an immediate or **POP** from the stack into **EDX** before using it again.

**Before Deobfuscation:**

```
00401028 xor edx, 131087D0h
0040102E mov edx, 0F1D71B16h
```

**EDX Math**

**After Deobfuscation:**

```
00401028 nop
00401029 nop
0040102A nop
0040102B nop
0040102C nop
0040102D nop
0040102E mov edx, 0F1D71B16h
```

**NOP’d Math**

Black Hat 2008
Finishing Up

- The Deobfuscator has finished matching obfuscation patterns
- Slack space is no longer needed, so we run one of the clean-up modes to simplify the appearance of the control flow
- “NOP Remove” injects JMPs to remove NOPs from control flow
- “Collapse” mode moves code to slack space to eliminate NOPs and JMPs
Before:

After:

Black Hat
Overview

• The Problem: Obfuscation
• Malware Example: RustockB
• The Solution: Deobfuscator
• Demonstration
• RustockB: Before & After
• Sample Source Code
• Summary
for (i = 7; i > 0; i--)
{
    Address = 0x00401B82 // Starting address of encrypted region
    Key1 = 0x4DFEE1C0 // Decryption key 1
    Key2 = 0x0869ECC5 // Decryption key 2
    Key3 = 0 // Decryption key 3
    Key4 = 0 // Decryption key 4 (Accumulator)
    for (j = 0x44DC; j > 0; j--, Address += 4) // 0x44DC = size of encrypted region
    {
        for (k = 2; k > 0; k--)
        {
            Key4 = k * 4
            XOR Key4, 0x5E57B7DE
            XOR Key4, Key3
            Key4 += Key2
            XOR Key1, k
            [Address] -= Key4
            Key3 += Key1
        }
    }
}

for (i = 0x44DC, Address = 0x00401B82, Sum = 0; i > 0; i--, Address += 4)
    Sum += [Address] // Add up the encrypted region (a DWORD at a time) in EAX

for (i = 0x44DC, Address = 0x00401B82; i > 0; i--, Address += 4)
    XOR [Address], Sum // XOR each DWORD of the encrypted region with the sum in EAX

Black Hat 2008
Overview

- The Problem: Obfuscation
- Malware Example: RustockB
- The Solution: Deobfuscator
- Demonstration
- RustockB: Before & After
- Sample Source Code
- Summary
Sample Source Code

The Simple Solution:
A Simple Problem:

// CALL NULL - A function call that just returns
//-------------------------------------------------------------------------------
int CALL_NULL(insn_t call, FILE *outfile, int *instr_offset){
    if (call.itype == NN_call && call.Operands[0].type == o_near){
        if (!get_next_instruction(call.Operands[0].addr)) return 0;
        insn_t ret = call;
        // Function that just returns
        if (ret.itype == NN_retn){
            "instr_offset = call.size;
            msg("%a CALL_NULL\n", call.ea);

            // NOP the call
            fprintf(outfile, "%X 5 90 90 90 90 90\n", get_fileregion_offset(call.ea));

            // NOP the return
            fprintf(outfile, "%X 1 90\n", get_fileregion_offset(ret.ea));

            return 1;
        }
    }
    return 0;
}

Black Hat 2008
Overview

- The Problem: Obfuscation
- Malware Example: RustockB
- The Solution: Deobfuscator
- Demonstration
- RustockB: Before & After
- Sample Source Code
- Summary

Black Hat 2008
Summary

• Most malware authors that wish to protect their IP use obfuscation techniques
• The Deobfuscator detects and simplifies many of these obfuscation and anti-disassembly patterns
• Over time, the repository of patterns will be developed to characterize most generic cases of obfuscation
Future Development

• Iterative patching of IDA database
Future Development

✓ Iterative patching of IDA database
  • Code collapsing
Future Development

- Iterative patching of IDA database
- Code collapsing
  - Grammar
  - Black-box control flow

Black Hat 2008
Contact

- For more information on this and other tools, contact:

  **Eric Laspe, Reverse Engineer**
  elaspe@rri-usa.org
  937-427-7042

  **Jason Raber, Lead Reverse Engineer**
  jraber@rri-usa.org
  937-427-7085

- Visit us online:
  [http://www.rri-usa.org/isrssoftware.html](http://www.rri-usa.org/isrssoftware.html)