Semi automatic binary deprotection

Alexandre GAZET
Yoann GUILLOT
Plan

1. Metasm
2. Structural manipulation
3. Challenge T2 2007
4. Optimization
Plan

1. Metasm
   - Other disassemblers
   - Binding
   - Backtracing

2. Structural manipulation

3. Challenge T2 2007

4. Optimization
Metasm

Structural manipulation
Challenge T2 2007
Optimization

Other disassemblers
Binding
Backtracing

A. Gazet & Y. Guillot
Semi automatic binary deprotection
Metasm

A. Gazet & Y. Guillot

Semi automatic binary deprotection
The reference: **IDA Pro**

- Very good on *unobfuscated* code: compiled binaries (*Microsoft*)
- Not useful on obfuscated binaries
  - No code interpretation
  - Heavy hypothesis on code behavior

**Hypothesis**

- Both branches of conditionnal jumps are taken
- No overlapping instructions
- The call instruction always returns
Disassemblers

The reference: **IDA Pro**

- Very good on *unobfuscated* code: compiled binaries (Microsoft)
- Not useful on obfuscated binaries
  - No code interpretation
  - Heavy hypothesis on code behavior

**Hypothesis**

- Both branches of conditional jumps are taken
- No overlapping instructions
- The **call** instruction always returns
Hypothesys: call returns

```assembly
.loc_403E9F:
    push    ebp
    push    ecx
    push    ebp
    call    sub_40BECD
    outsb
    cmp     edx, esp
    push    esp
    inc     esi
    add     dword ptr [esp+4], 1
    add     esp, 4
    xor     ebx, edx
    rep     jmp locret_4049F5
```

**CODE XREF:** `.text: loc_40CDEF`
Failure

```asm
push ebp
push ecx
push ebp
call sub_40BECD

.db 6Eh ; n

---------- SUBROUTINE ----------

cmp edx, esp
push esp  sub_40BECD  proc near ; CODE XREF: .text:00403EA2
inc esi
add dword ptr [esp+1], 1
add esp, 4
xor ebx, edx
rep syscall  sub_40BECD  endp

cmp eax, ebp
add dword ptr [esp+0], 1
test ebx, 1E2h
ret 0Ch
```

A. Gazet & Y. Guillot Semi automatic binary deprotection 8/57
Definition

This is how we call an instruction’s semantics, through an array of symbolic expressions.

Instruction **ADD**:  

\[
\begin{align*}
    a &= \text{di.instruction.args.symbolic} \\
    \text{res} &= \text{Expression}([[a[0], :&, mask], :+, [a[1], :&, mask]]) \\
    \text{binding}[a[0]] &= \text{Expression}[[\text{res}, :&, mask]]
    \\
    \text{binding}[:eflag_z] &= \text{Expression}[[\text{res}, :&, mask], :==, 0] \\
    \text{binding}[:eflag_s] &= \text{sign}(\text{res}) \\
    \text{binding}[:eflag_c] &= \text{Expression}[[\text{res}, :>, mask] \\
    \text{binding}[:eflag_o] &= \text{Expression}[[\text{sign}(a[0]), :==, \text{sign}(a[1])], :', :', [\text{sign}(a[0]), :', :', \text{sign}(\text{res})]]
    
\end{align*}
\]
### Binding

**Instruction CALL:**

\[
\text{addr\_ret} = \text{Expression}[\text{di\_address}, :+, \text{di\_bin\_length}].\text{reduce}
\]

\[
\text{binding} = \{
    :\text{esp} \Rightarrow \text{Expression}[:\text{esp}, :-, \text{opsz}],
    \text{Indirection}[:\text{esp}, \text{opsz}] \Rightarrow \text{addr\_ret}
\}
\]

**For exemple:**

\[
dword\text{ ptr}[\text{esp}] = 0x4010CE
\]
\[
\text{esp} = \text{esp} - 4
\]

**Instruction RDTSC:**

\[
\text{binding} = \{
    :\text{eax} \Rightarrow \text{Expression}::\text{Unknown},
    :\text{edx} \Rightarrow \text{Expression}::\text{Unknown}
\}
\]
**Binding**

**Instruction CALL:**

\[
\text{addr\_ret} = \text{Expression} [\text{di\_address}, \text{+}, \text{di\_bin\_length}] . \text{reduce} \\
\text{binding} = \{
    \text{esp} \Rightarrow \text{Expression} [\text{esp}, \text{-}, \text{opsz}], \\
    \text{Indirection} [\text{esp}, \text{opsz}] \Rightarrow \text{addr\_ret}
\}
\]

**For exemple:**

\[
\text{dword ptr [esp]} = 0x4010CE \\
\text{esp} = \text{esp} - 4
\]

**Instruction RDTSC:**

\[
\text{binding} = \{
    \text{eax} \Rightarrow \text{Expression} :: \text{Unknown} , \\
    \text{edx} \Rightarrow \text{Expression} :: \text{Unknown}
\}
\]
Binding

**Instruction** CALL:

\[
\text{addr\_ret} = \text{Expression} [\text{di\_address}, +, \text{di\_bin\_length}].\text{reduce} \\
\text{binding} = \{
\quad :\text{esp} \Rightarrow \text{Expression}[:\text{esp}, -, \text{opsz}], \\
\quad \text{Indirection}[:\text{esp}, \text{opsz}] \Rightarrow \text{addr\_ret}
\}
\]

For exemple:

\[
\text{dword \_ptr}[\text{esp}] = 0\times4010CE \\
\text{esp} = \text{esp} - 4
\]

**Instruction** RDTSC:

\[
\text{binding} = \{
\quad :\text{eax} \Rightarrow \text{Expression}::\text{Unknown}, \\
\quad :\text{edx} \Rightarrow \text{Expression}::\text{Unknown}
\}
\]
Backtracing, the theory

**Definition**
Symbolic emulation by walking the instruction flow backwards
Backtracing, the facts

Flot d’exécution:

call loc_40becdh ; @403ea2h e826800000
cmp eax, ebp ; @40becdh 39e8
add dword ptr [esp+0], 1 ; @40becfh 8344240001
test ebx, 1e2h ; @40bed4h f7c3e2010000
ret 0ch ; @40bedah c20c00

**Backtracing**

1. backtrace 40becfh add dword ptr [esp+0], 1
dword ptr [esp] => dword ptr [esp]+1

2. backtrace up 40becdh->403ea2h dword ptr [esp]+1

3. backtrace 403ea2h call loc_40becdh
dword ptr [esp]+1 ⇒ 403ea8h

4. backtrace result: 403ea8h
Metasm

Assembler listing produced:

```
loc_403e9fh :
push ebp ; @403e9fh  55
push ecx ; @403ea0h  51
push ebp ; @403ea1h  55
call loc_40becdh ; @403ea2h e826800000 noreturn

db  6eh ; @403ea7h

// Xrefs: 40bedah
loc_403ea8h :
cmp edx, esp ; @403ea8h 39e2
push esp ; @403eaah 54

[ ... ]

// Xrefs: 403ea2h
loc_40becdh :
cmp eax, ebp ; @40becdh 39e8
add dword ptr [esp+0], 1 ; @40becfh 8344240001
test ebx, 1e2h ; @40bed4h f7c3e2010000
ret 0ch ; @40bedah c20c00 x:loc_403ea8h
```
Plan

1. Metasm

2. Structural manipulation
   - Introduction
   - Control graph complexification
   - Neutral element insertion
   - Unprotection

3. Challenge T2 2007

4. Optimization
Securitech 2006 - Challenge 10

**Poeut.exe**
- Massively obfuscated binary
- **IDA** overwhelmed
- **Metasm** disassembles correctly, but:
  - Binary blocks are randomly moved in the binary
  - ⇒ need to write a graphic *front-end*

**yEd - Graph Editor**
- Visualise graphs
- Needs a *graphml* file as input
- We’ll translate **Metasm** internal *InstructionBlock* representation to this format
Raw graph
Obscur predicates

- The predicate function always return true
- Using a way not easily statically analysed
- The conditionnal jump is in fact not conditionnal

```assembly
if ( x^4 * (x-5)^2 >= 0 ) {
    goto real_code;
} else {
    goto nowhere;
}
```

```assembly
fstp qword ptr [esp+8]
fstp qword ptr [esp]
call thunk_pow
fstp qword ptr [ebp-0x20]
mov eax, dword ptr [ebp-0ch]
sub eax, 5
push eax
fld dword ptr [esp]
lea esp, dword ptr [esp+4]
fld qword ptr [xref_8048590h]
fstp qword ptr [esp+8]
fstp qword ptr [esp]
call thunk_pow
fld qword ptr [ebp-20h]
fmulp ST(1)
```
Obscur predicates

Full randomisation

- The predicate function returns randomly true or false
- Both branches after the conditionnal jump are equivalent

```c
if ( rand() % 2 ) {
    real_code_A;
} else {
    real_code_B;
}
```
Massive use of random predicates:

```
// xrefs: 404170h
loc_40bf25h:
    jz loc_40e3d2h ; @40bf25h 0f64a7240000 x:loc_40e3d2h

// xrefs: 40bf25h
loc_40e3d2h:
    test ecx, edi
    push loc_40bf5ch
    jmp loc_40bf32h

// xrefs: 40e3d9h
loc_40bf32h:
    ret ; @40bf32h c3 x:loc_40bf5ch
```

*diamond* shaped graph.
Neutral elements

**Definition**

Instruction (or group of) having empty semantics: no effect on the execution context.
Neutral elements: apparent randomization

Implementation:

test esp, ebx
cmp ebx, edx
mov byte ptr [edx], al
add dword ptr [esp+0], 6
inc edx
jmp loc_40b25dh

Solving the problem

- Use the instructions’ binding
- Incoherent use of the processor flags seen in the data flow:
  - Two successive writings,
  - or written but never read
Neutral elements: fake subfunctions

Implémentation :

```Assembly
push eax ; @408dadh 50
push ecx ; @408daeh 51
push ebp ; @408dafh 55
[...]
call loc_4037f2h ; @40932fh
[...]
push esp ; @4037f4h 54
push ecx ; @4037f5h 51
[...]
add dword ptr [esp+8], 9 ; @4037f9h 8344240809
add esp, 8 ; @403800h 83 c408
[...]
ret 0ch ; @403808h c20c00 x:loc_40933dh
```

Solution

- Execution flow reconstruction
- Stack emulation
- We have a pattern: return address modification
Epilogue (raw)
Execution flow analysis

Solution

- Walk the internal tree of InstructionBlock
- Inline functions if needed
- Scan for a diamond pattern
- Construction, cleaning and comparison of flows
- Factorisation
Epilogue (factorized)
A. Gazet & Y. Guillot

Semi automatic binary deprotection
Epilogue (final)
Last touch

- Cleanup of the whole program graph
- Output of a clean asm source
- Used to reassemble an unprotected binary
Plan

1. Metasm
2. Structural manipulation
3. Challenge T2 2007
   - Introduction
   - Obfuscation
   - Virtual machine
   - Resolution
4. Optimization
t207.exe

- http://www.t2.fi/challenge/
- Goal: find the password to unlock the program
- Very simple binary [demo/1]
- Loads an obfuscated driver [demo/2]
Deobfuscation

Obfuscation types

- Junk code
- Obfuscated arithmetics
- Ring3 detection
- Code duplication
Junk code

```
ror edi, 0dh
xchg ebx, edi
ror ebx, 13h
xchg ebx, edi
```
Obfuscated arithmetics

bit rotation

```
push eax
push ecx
rol dword ptr [esp+4], cl
pop ecx
pop eax
```
Ring3 detection

test ring0

```
pushfd
push eax
xor eax, eax
mov ax, cs
cmp eax, 9
jle loc_131d5h
rdtsc
imul eax, ecx
jmp eax ; x:unknown

loc_131d5h:
pop eax
popfd
```
Code duplication

duplication

```
push esi
push ebx
pushfd
rdtsc
imul ecx, ebx
cmp cl, 7fh
jnb loc_21aba
popfd
pop ebx
pop esi

loc_21aba:
popfd
pop ebx
pop esi
```
Virtual machine

**Handler structure**
- Simple operations
- Similar blueprint
- Operations controlled by [ebp]

[demo/3]
Virtual addition

An addition handler

loc_15336:

mov ecx, dword ptr [ebp+0ch]
oxor ecx, 842b1208h

mov ecx, dword ptr [ebx+ecx]

mov eax, dword ptr [ebp+8]
oxor eax, 842b1208h

add dword ptr [ebx+eax], ecx
Inter-handler transition

transition to the next handler

```assembly
mov ecx, dword ptr [ebp+0]
xor ecx, 149f0c63h
mov ebp, dword ptr [ebp+4]
xor ebp, 842b1208h
add ebp, dword ptr [ebx+14h]
add ecx, dword ptr [ebx+14h]
jmp ecx ; x:unknown
```
Virtual machine architecture

```
handler
mov eax, [ebp+8]
xor eax, h1_key_2
mov [ebx+eax], 0
mov eax, [ebp]
xor eax, h1_key_1
mov ebp, [ebp+4]
xor ebp, h1_key_2
add eax, [ebx+14h]
add ebp, [ebx+14h]
jmp eax

next_handler next_instr arg0
key_1
key_2

ciphered instruction

handler
mov eax, [ebp+8]
xor eax, h2_key_2
mov ecx, [ebp+12]
xor ecx, h2_key_2
mov ecx, [ebx+ecx]
add [ebx+eax], ecx

next_handler next_instr arg0 arg1
key_1
key_2
```

A. Gazet & Y. Guillot Semi automatic binary deprotection 39/57
Handler enumeration

Direct enumeration impossible

- No “handler table”
- Need to follow the virtual code flow step by step
- Binary size implies numerous handlers
- Auto-analyse every handler behavior using backtracking

[demo/4]
Analyse result

Handler binding

```
handler_13491h:
// "reg0" <- Expression["reg0", :, "reg1"]
// handler type: add reg, reg
    mov eax, dword ptr [ebp+0ch]
    xor eax, 8d3f5d8bh
    mov eax, dword ptr [ebx+eax]
    mov ecx, dword ptr [ebp+8]
    xor ecx, 8d3f5d8bh
    add dword ptr [ebx+ecx], eax
```
Raw virtual code

The first virtual instructions

```assembly
textpoint_219feh_21ea6h:
    nop
    mov r68, 28h
    add r68, host_esp
    mov r64, dword ptr [r68]
    mov dword ptr [esp], r64
    mov r64, 4
    add esp, r64
    mov r68, 2ch
    add r68, host_esp
    mov r64, dword ptr [r68]
    mov dword ptr [esp], r64
    mov r64, 4
    add esp, r64
    trap
```
Virtual macro-instructions

Higher abstraction level

- Through pattern recognition
- Reconstruction of a higher level assembler
- Apparition of functions (call, ret)
Virtual macro-code

The first instructions (again)

```assembly
entrypoint_219feh_21ea6h:
    mov dword ptr [esp], dword ptr [host_esp+28h]
    add esp, 4
    mov dword ptr [esp], dword ptr [host_esp+2ch]
    add esp, 4
    mov ebp, esp
    add esp, 234h
    mov r64, dword ptr [ebp+200h]
    xor r64, 1
    jrz loc_2d630h_2d8ffh, r64
    syscall_alloc_ptr r64, 0ch
    mov dword ptr [ebp+200h], r64
loc_2d630h_2d8ffh:
```
Decompilation

- The instruction semantic is very simple
- The code patterns reminds C
- Check it out
Validation

- Enabled us to solve the challenge by a pure static approach
Plan

1. Metasm
2. Structural manipulation
3. Challenge T2 2007
4. Optimization
   - Introduction
   - POC
   - Results
   - Prospects
Facts

What we have done so far
- Stress was put on binary manipulation: **code rewriting**
- Add effective methods in Metasm to clean obfuscation on the fly
- At various level: filtering processor, graph manipulation . . .

Drawbacks
- Lack of a higher level of abstraction
- Analysis is still time-consuming and painfull
- Mostly target-specific
Concepts

Objectives
- Generic rewriting rules
- Obfuscated code as input
- Clean (obfuscation free) code as output

Means
- What we want to do looks like optimization!
- Extensive literature on the subject
- Can be easily applied at assembly level using Metasm: we have methods to work on basic blocks & instructions
Proof of concept

Implemented optimizations

- **Declaration cleaning**: remove useless assignments
- **Constant propagation**
- **Constant, operation folding**: apply basic rules of arithmetic
- **Peephole**: replace known patterns with a reduced form

Each of those optimization amounts to one or many rewriting rules, possibly associated with a condition.

We apply them locally, on each basic bloc, spaghetti code has been previously merged.
Example: constant propagation

Before

100 bb5cfh mov al, 12h
100 bed67h mov cl, 46h
100 bed69h xor cl, al

After

100 bb5cfh mov al, 12h
100 bed67h mov cl, 46h
100 bed69h xor cl, 12h

The constant (0x12) has been propagated through al
**Example 2: constant folding**

Before

```
100 bb5cfh mov al, 12h
100 bed67h mov cl, 46h
100 bed69h xor cl, 12h
```

After

```
100 bb5cfh mov al, 12h
100 bed67h mov cl, 54h
```

*cl* is now assigned with $0x46 \timesor 0x12h$
Does it work?
Does it work?

Metasm
Structural manipulation
Challenge T2 2007
Optimization

Introduction
POC
Results
Prospects

Does it work?

A. Gazet & Y. Guillot Semi automatic binary deprotection 54/57
Does it work? Actually yes

```
entrypoint_1000c9c2h:
    pop eax
    add dword ptr [esp], eax
    lodsb
    add al, bl
    xor al, 0d2h
    sub al, 0f1h
    add bl, al
    movzx eax, al
    jmp dword ptr [edi+4*eax]
```
Conclusion and prospect

Conclusion

- Using optimization to defeat obfuscation is very promising
- The kind of obfuscation used in the protection is too weak: mainly based on local constants expansion and affine functions.
- We don’t have a real intermediate representation

Prospect

- We are working on decompilation problems
- We will study the opportunity to use jointly Metasm and LLVM
The end

Thank you for your attention!

Any questions?