

# Binary deprotection with metamsm and stuff

Alexandre Gazet

Sogeti / ESEC R&D

[alexandre.gazet\(at\)sogeti.com](mailto:alexandre.gazet@sogeti.com)

Yoann Guillot

Sogeti / ESEC R&D

[yoann.guillot\(at\)sogeti.com](mailto:yoann.guillot@sogeti.com)



## HITB 2009

# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
- 3 Decompilation

# Metasm

- a pure ruby opensource framework
- assembler/dissassembler
  - ia32 (16/32/64bits), mips
  - Even supports cr7
- debugger
  - linux, windows, remote
- compiler/decompiler (more or less :)
- GUI included !

# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation



# Debugger

| Actions      |              |              |              |                 |  |
|--------------|--------------|--------------|--------------|-----------------|--|
| eax=000003e8 | ebx=00000000 | ecx=00000000 | edx=00000000 | esi=bf8c90f4    |  |
| edi=fffffff  | ebp=bf8c9008 | esp=bf8c9008 | eip=b7ec3553 | c P a Z s I d o |  |
| 0bf8c90f0h   | 01 00 00 00  | 4b 99 8c bf  | ...K...      |                 |  |
| 0bf8c90f8h   | 00 00 00 00  | 57 99 8c bf  | ...W...      |                 |  |
| 0bf8c9100h   | 77 99 8c bf  | 89 99 8c bf  | w.....       |                 |  |
| 0bf8c9108h   | 91 99 8c bf  | a3 99 8c bf  | .....        |                 |  |

```
weak_getuid:  
push ebp  
mov ebp, esp  
mov eax, 0c7h  
↓  
call dword ptr gs:[10h] ; x:unknown  
pop ebp  
ret
```

```
:d esp  
:loadsyms  
loaded 9 symbols from /usr/bin/id  
loaded 13 symbols from libdl.so.2  
loaded 1974 symbols from libc.so.6  
loaded 192 symbols from libselinux.so.1  
loaded 24 symbols from ld-linux.so.2  
:g getuid  
using weak_getuid for getuid  
:bt  
0b7ec3553h libc.so.6!weak_getuid+3  
804954dh /usr/bin/id!_init+95d  
0b7e39775h libc.so.6!__libc_start_main+e5  
8048f31h /usr/bin/id!_init+34l  
:bpm  
bpm: set a hardware memory breakpoint
```

stopped



# Features

- Direct manipulation of the OS primitives
  - `sys_ptrace`
  - `WaitForDebugEvent`
- Very fine & low-level control
- Unified high-level interface
  - Linux, Windows, GDBserver
  - Conditionnal breakpoints, callback. . .



# Plan

- 1 Metasm
  - Debugger
  - **Compiler**
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation



# C Compiler

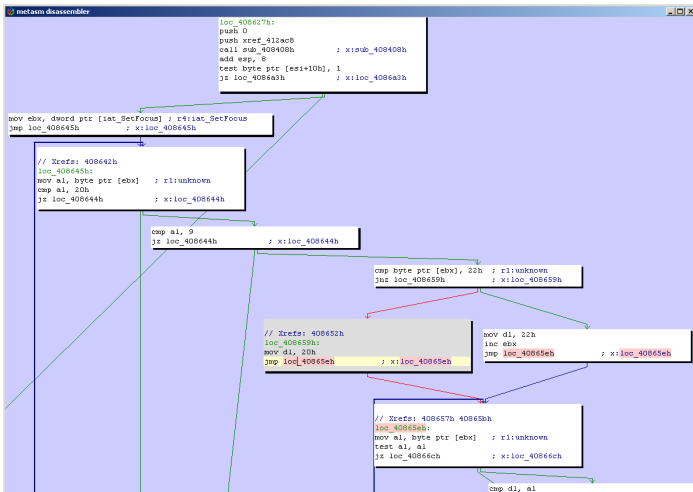
- Rudimentary C compiler
- x86 only
- Framework integration easy to leverage
  - Easy to customize e.g. dynamic symbol resolution



# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation

# Disassembler



# Disassembly

## The reference: **IDA Pro**

- Excellent on unobfuscated binaries
- Not so useful on protected code
  - No code interpretation
  - Strong hypothesis

### Hypothesis

- Both branches are taken on a conditionnal jump
- Two instructions never overlap
- A subfunction call returns

# Disassembly

## The reference: **IDA Pro**

- Excellent on unobfuscated binaries
- Not so useful on protected code
  - No code interpretation
  - Strong hypothesis

## Hypothesis

- Both branches are taken on a conditionnal jump
- Two instructions never overlap
- A subfunction call returns

# Hypothesis: all call returns

```
.text:00403E9F ; -----  
.text:00403E9F  
.text:00403E9F loc_403E9F: ; CODE  
* .text:00403E9F      push     ebp  
* .text:00403EA0      push     ecx  
* .text:00403EA1      push     ebp  
* .text:00403EA2      call    sub_40BECD  
* .text:00403EA7      outsb  
* .text:00403EA8      cmp     edx, esp  
* .text:00403EA9      push     esp  
* .text:00403EAB      inc     esi  
* .text:00403EAC      add     dword ptr [esp+4], 1  
* .text:00403EB1      add     esp, 4  
* .text:00403EB4      xor     ebx, edx  
* .text:00403EB6      rep    jnp locret_4049F5  
* .text:00403EBC ; -----
```

```
.text:00403E9F loc_403E9F: ; CODE XREF: .text:loc_40CDEF  
.text:00403E9F      push     ebp  
.text:00403EA0      push     ecx  
.text:00403EA1      push     ebp  
.text:00403EA2      call    sub_40BECD  
.text:00403EA7      outsb  
.text:00403EA8      cmp     edx, esp  
.text:00403EAA      push     esp  
.text:00403EAB      inc     esi
```



# Failure

```
push    ebp
push    ecx
push    ebp
call    sub_40BECD

----- SUBROUTINE -----
db  6Eh ; n

cmp     edx, esp
push   esp
inc     esi
add     dword ptr [esp+0], 1
add     esp, 4
xor     ebx, edx
rep    insb

proc near ; CODE XREF: .text:0040BECF
cmp     eax, ebp
add     dword ptr [esp+0], 1
test    ebx, 1E2h
retn    0Ch
endp
```

```
.text:0040BECD sub_40BECD
```

```
proc near ; CODE XREF: .text:0040BECF
```

```
.text:0040BECD
```

```
cmp     eax, ebp
```

```
.text:0040BECF
```

```
add     dword ptr [esp+0], 1
```

```
.text:0040BED4
```

```
test    ebx, 1E2h
```

```
.text:0040BEDA
```

```
retn    0Ch
```

```
.text:0040BEDA sub_40BECD
```

```
endp
```

# Binding

## Our solution:

Express instruction effects through symbolic expressions. This associates semantics to each instruction.

Instruction **ADD**:

```
res = Expression [[a[0], :&, mask], :+, [a[1], :&, mask]]
binding[a[0]] = res
binding[:eflag_z] = Expression [[res, :&, mask], :==, 0]
binding[:eflag_s] = sign[res]
binding[:eflag_c] = Expression [res, :>, mask]
binding[:eflag_o] = Expression [[sign[a[0]], :==, sign[a[1]]],
                                :'&&', [sign[a[0]], :'?!=', sign[res]]]
```





# Binding

## Our solution:

Express instruction effects through symbolic expressions. This associates semantics to each instruction.

## Instruction **ADD**:

```
res = Expression [[a[0], :&, mask], :+, [a[1], :&, mask]]
binding[a[0]] = res
binding[:eflag_z] = Expression [[res, :&, mask], :==, 0]
binding[:eflag_s] = sign[res]
binding[:eflag_c] = Expression [res, :>, mask]
binding[:eflag_o] = Expression [[sign[a[0]], :==, sign[a[1]]],
                                :'&&', [sign[a[0]], : '!=' , sign[res]]]
```



# Binding

## Instruction CALL:

```
binding[:esp] = Expression[:esp, :-, opsz]  
binding[ Indirection[:esp, 4] ] = di.next_addr
```

For exemple:

```
dword ptr [esp] = 0x403EA7  
esp = esp-4
```

## Instruction RDTSC:

```
binding[:eax] = Expression::Unknown  
binding[:edx] = Expression::Unknown
```

# Binding

## Instruction CALL:

```
binding[:esp] = Expression[:esp, :-, opsz]  
binding[ Indirection[:esp, 4] ] = di.next_addr
```

## For exemple:

```
dword ptr [esp] = 0x403EA7  
esp = esp-4
```

## Instruction RDTSC:

```
binding[:eax] = Expression::Unknown  
binding[:edx] = Expression::Unknown
```



# Binding

## Instruction CALL:

```
binding[:esp] = Expression[:esp, :-, opsz]  
binding[ Indirection[:esp, 4] ] = di.next_addr
```

## For exemple:

```
dword ptr [esp] = 0x403EA7  
esp = esp-4
```

## Instruction RDTSC:

```
binding[:eax] = Expression::Unknown  
binding[:edx] = Expression::Unknown
```



# Backtracking, the theory

## Definition

Symbolic emulation by walking the instruction flow backwards.

# Backtracking, the facts

## Execution flow:

```
    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 x dword ptr [esp] for 40bedah ret 0ch

- 1 backtrace 40becfh dword ptr [esp] => dword ptr [esp]+1
- 2 backtrace up 40becdh->403ea2h dword ptr [esp]+1
- 3 backtrace 403ea2h dword ptr [esp]+1 => 403ea8h
- 4 backtrace result: 403ea8h

# Metasm

## Result:

```
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 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation





# Code protection?

- People want to be able to hide the functioning of their code
  - Using code obfuscation
  - Using code virtualization
  - Both



# Defeating code protection

- Need to understand the virtual machine architecture
- The VM interpreter is obfuscated
- Must defeat code obfuscation
- Generate a translator from virtual bytecode to easy to read code
- We'll introduce a framework to assist is those steps



## Previous work: T2 2007

- On the fly deobfuscation with patterns
- Virtual processor generator
  - Automatic semantics analysis



# The target

- A realworld virtualization based protection:
  - Some protected chunks of code are virtualized
  - Virtualized code (bytecode) is executed using an embedded interpreter
  - Each virtual machine instance is unique (polymorphism)
- Code is massively obfuscated
- List of all the instruction handlers is trivial to get



# Starting point

## Need of automation

- What we already have:
  - Filtering processor
  - Control flow graph (CFG) walking
  - Rewriting rules application
  - On-the-fly CFG modification

## Manual analysis of obfuscated code

- Trying to find patterns manually?
  - Painful
  - Loss of genericity
  - Possibly ineffective: polymorphism



# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation

# Compiler approach

## Our needs

- Semantics preservation
- Rewrite code into a simpler form
- Dead code elimination
- etc.

- Compilers already do this kind of jobs: **optimization**
- Our optimization criteria: code conciseness



# Using compiler optimisation

## Proposed approach

- Addition of an optimization module to our CFG walkthrough module
- For each handler:
  - 1 Its code is recovered
  - 2 Then optimized





## Constant propagation

```
cfh mov al , 12h  
67h mov cl , 46h  
69h xor cl , al
```

```
cfh mov al , 12h  
67h mov cl , 46h  
69h xor cl , 12h
```

Figure: Propagation of 12h through *al*.



## Constant folding

```
cfh mov al, 12h  
67h mov cl, 46h  
69h xor cl, 12h
```

```
cfh mov al, 12h  
67h mov cl, 54h
```

Figure: *cl* value folding.



## Operation folding

```
4fh add al, -7fh  
51h add al, bl  
53h add al, -70h
```

```
4fh add al, 11h  
51h add al, bl
```

Figure: *add* operation folding.



# Demonstration

Optimization of a handler



## Next step

- Now we are ready to analyse the virtual machine architecture



# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - **Breaking code virtualization**
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation

## At the beginning

### 2<sup>nd</sup> Futamura projection

*Given two languages  $L_a$  and  $L_b$ , it is possible to find a compiler from  $L_b$  to  $L_a$ , if we know an interpreter of  $L_b$  written in  $L_a$*

- **Once again, compiler approach is the answer!**
- It's only a proof of existence
- How to generate it?



## Semantical analysis of the handlers

- code\_binding: method from the Disassembler object

### Example of an optimized handler

```
lods d
xor eax, ebx
add eax, 859fcfaeh
sub ebx, eax
push eax
```

- Semantics (aka *binding*)

```
dword ptr [esp] := (dword ptr [esi]^ebx)+859fcfaeh
eax := (dword ptr [esi]^ebx)+859fcfaeh
ebx := ebx+-(dword ptr [esi]^ebx)-859fcfaeh
esi := esi+4
esp := esp-4
```





# Semantical analysis

- When a handler is encountered for the first time:
  - Its code is optimized
  - Its semantics is computed and stored
- We progressively build **the description of the semantics of the interpreter !**
- Given the current program state, we are able to emulate the next step of bytecode.



# From static to (almost) dynamic

- Handler's binding:

```
dword ptr [esp] := (dword ptr [esi]^ebx)+859fcfaeh  
eax := (dword ptr [esi]^ebx)+859fcfaeh  
ebx := ebx-(dword ptr [esi]^ebx)-859fcfaeh  
esi := esi+4  
esp := esp-4
```

- Current context (partial)

```
eax := 93h  
ebx := 0fd8dh  
esi := 100167beh  
[...]
```

- Contextualized binding:

```
dword ptr [esp] := 0c0000001h  
eax := 0c0000001h  
ebx := 4000fd8ch  
esi := 100167c2h  
esp := esp-4
```



## From static to (almost) dynamic (2)

Based on contextualized binding:

- Generate corresponding assembly:

```
push 0c0000001h
```

- **Symbolic execution**: compute context after execution of the current handler
- Follow bytecode execution flow
  - Support virtual calls and jumps (conditionnal or not)

⇒ **Recover the whole chunk of code in native Ia32 assembly**



# Demonstration

Symbolic execution and assembly generation



## Results

- The whole chunk of bytecode is compiled, on-the-fly, into native Ia32 assembly
- Compiled bytecode itself is still obfuscated
- Still many references to the virtual machine's context
- Looks like a stack automaton

### Next steps

⇒ **Re-use optimisation module + inject abstraction**



# Abstraction injection

## Processor extension

```
list = Reg.i_to_s[32].concat( %w[ virt_eax ])
Reg.i_to_s[16].concat( %w[ virt_ax ])
Reg.i_to_s[8].concat( %w[ virt_al ])

Reg.s_to_i.clear
Reg.i_to_s.each { |sz, hh|
  hh.each_with_index { |r, i|
    Reg.s_to_i[r] = [i, sz]
  }
}
Reg::Sym.replace list.map { |s| s.to_sym }
```



# Demonstration

Optimized chunk with virtual registers

## Final pass

- Injection of virtual registers
- Applying optimizations ⇒
  - Stack automaton aspect totally removed
  - Code is expressed using virtual registers only
- Virtual registers are then mapped back to native registers
- Compilation and links edition

⇒ original, unprotected chunk of code is retrieved





# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - **Putting the pieces together**
  - Conclusion(s)
- 3 Decompilation



# Demonstration

Unvirtualized code, mapped into the  
original binary



# Plan

- 1 Metasm
  - Debugger
  - Compiler
  - Disassembler
    - Binding
    - Backtracking
- 2 Analysis of a protection
  - Breaking obfuscation
  - Breaking code virtualization
  - Putting the pieces together
  - Conclusion(s)
- 3 Decompilation



## Conclusion(s) 1/2

- **Optimisation** (rewriting rules)
  - Quite effective
  - Our implementation is limited
    - Local optimizations
    - Lack of an intermediate representation
    - Unsuitable to control flow obfuscation
- **Partial evaluation or specialization**
  - Pre-computation of all the static elements:
    - Data transfers within the obfuscated code
    - Application of the interpreter to the bytecode
  - Generic Approach
  - Require relatively heavy computation



## Conclusion(s) 2/2

- **Integration and re-usability of the deobfuscator**
  - Actual code is still a “prototype”
  - Being integrated into the framework as a plugin
  - Usable on x86 code, with some cross-platform parts



# Plan

- 1 Metasm
- 2 Analysis of a protection
- 3 Decompilation

# Advantages

- Arch-specific code reduced to the minimum
- C code is much more expressive than asm
- Standard loops are simpler to handle
- Semantics is often simpler
  - No side-effects on flags



# Limitations

- Some asm constructs are difficult to express in C
  - *rol, ror*
  - *jmp eax*
- Needs that the code has certain properties
  - Split in function/subfunctions
  - Follows C ABIs/calling conventions
- Those last things can be worked around
  - Custom `__attribute__`





# Demo

## Decompilation

# Reminder

- **Metasm**

<https://metasm.cr0.org/>

- **Blog**

<http://esec.fr.sogeti.com/blog/>

# Conclusion

**Thanks for listening.**

**Questions ?**

