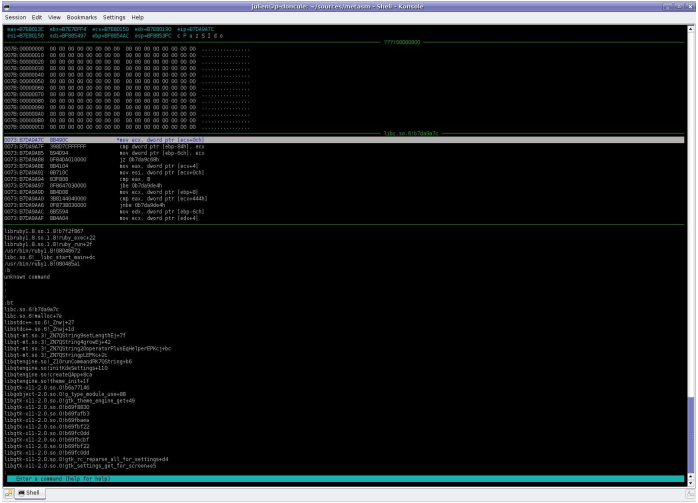
Semi automatic binary deprotection

Alexandre Gazet
Yoann Guillot

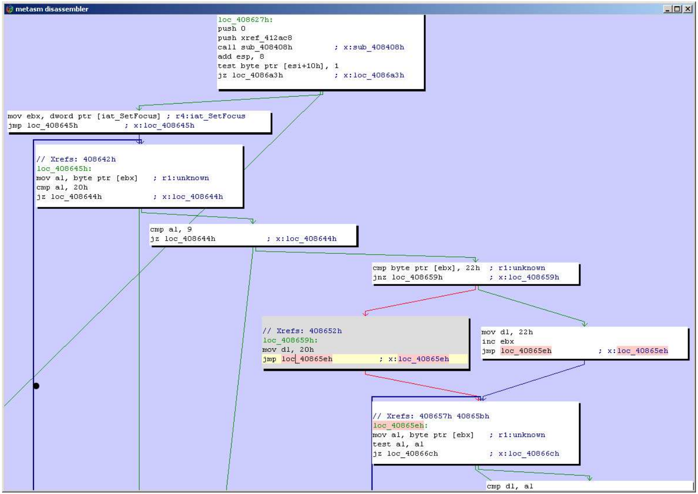
Plan

Plan

Metasm


julien@p-doncule: ~/sources/metasm - Shell - Konsole
Session Edit View Bookmarks Settings Help
eax=B7E8013C ebx=B7E7EFF4 ecx=B7E80150 edx=B7E80190 eip=B7DA9A7C
esi=B7E80150 edi=BF885497 ebp=BF8854AC esp=BF8853FC c P a z S I d o
???!00000000
007B:00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:000000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:000000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
007B:000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
libc.so.6!b7da9a7c
0073:B7DA9A7C 8B490C *mov ecx, dword ptr [ecx+0ch]
0073:B7DA9A7F 398D7CFFFFFF cmp dword ptr [ebp-84h], ecx
0073:B7DA9A85 894D94 mov dword ptr [ebp-6ch], ecx
0073:B7DA9A88 0F84DA010000 jz 0b7da9c68h
0073:B7DA9A8E 8B4104 mov eax, dword ptr [ecx+4]
0073:B7DA9A91 8B710C mov esi, dword ptr [ecx+0ch]
0073:B7DA9A94 83F808 cmp eax, 8
0073:B7DA9A97 0F8647030000 jbe 0b7da9de4h
0073:B7DA9A9D 8B4D08 mov ecx, dword ptr [ebp+8]
0073:B7DA9AA0 3B8144040000 cmp eax, dword ptr [ecx+444h]
0073:B7DA9AA6 0F8738030000 jnbe 0b7da9de4h
0073:B7DA9AAC 8B5594 mov edx, dword ptr [ebp-6ch]
0073:B7DA9AAF 8B4A04 mov ecx, dword ptr [edx+4]
libruby1.8.so.1.8!b7f2f867
libruby1.8.so.1.8!ruby_exec+22
libruby1.8.so.1.8!ruby_run+2f
/usr/bin/ruby1.8!08048672
libc.so.6!__libc_start_main+dc
/usr/bin/ruby1.8!080485a1
:b
unknown command
:
:
:
:bt
libc.so.6!b7da9a7c
libc.so.6!malloc+7e
libstdc++.so.6!_Znwj+27
libstdc++.so.6!_Znaj+1d
libqt-mt.so.3!_ZN7QString9setLengthEj+7f
libqt-mt.so.3!_ZN7QString4growEj+42
libqt-mt.so.3!_ZN7QString20operatorPlusEqHelperEPKcj+bc
libqt-mt.so.3!_ZN7QStringpLEPKc+2c
libqtengine.so!_Z10runCommandRK7QString+b6
libqtengine.so!initKdeSettings+110
libqtengine.so!createQApp+8ca
libqtengine.so!theme_init+1f
libgtk-x11-2.0.so.0!b6a77146
libgobject-2.0.so.0!g_type_module_use+88
libgtk-x11-2.0.so.0!gtk_theme_engine_get+49
libgtk-x11-2.0.so.0!b69f8830
libgtk-x11-2.0.so.0!b69fafb3
libgtk-x11-2.0.so.0!b69fbaea
libgtk-x11-2.0.so.0!b69fbf22
libgtk-x11-2.0.so.0!b69fc0dd
libgtk-x11-2.0.so.0!b69fbcbf
libgtk-x11-2.0.so.0!b69fbf22
libgtk-x11-2.0.so.0!b69fc0dd
libgtk-x11-2.0.so.0!gtk_rc_reparse_all_for_settings+d4
libgtk-x11-2.0.so.0!gtk_settings_get_for_screen+e5
:
Enter a command (help for help)
Shell

Metasm


metasm disassembler
loc_408627h:
push 0
push xref_412ac8
call sub_408408h ; x:sub_408408h
add esp, 8
test byte ptr [esi+10h], 1
jz loc_4086a3h ; x:loc_4086a3h
mov ebx, dword ptr [iat_SetFocus] ; r4:iat_SetFocus
jmp loc_408645h ; x:loc_408645h
// Xrefs: 408642h
loc_408645h:
mov al, byte ptr [ebx] ; r1:unknown
cmp al, 20h
jz loc_408644h ; x:loc_408644h
cmp al, 9
jz loc_408644h ; x:loc_408644h
cmp byte ptr [ebx], 22h ; r1:unknown
jnz loc_408659h ; x:loc_408659h
// Xrefs: 408652h
loc_408659h:
mov dl, 20h
jmp loc_40865eh ; x:loc_40865eh
mov dl, 22h
inc ebx
jmp loc_40865eh ; x:loc_40865eh
// Xrefs: 408657h 40865bh
loc_40865eh:
mov al, byte ptr [ebx] ; r1:unknown
test al, al
jz loc_40866ch ; x:loc_40866ch
cmp dl, al

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 conditionnal jumps are taken
- No overlapping instructions
- The `call` instruction always returns

Disassemblers

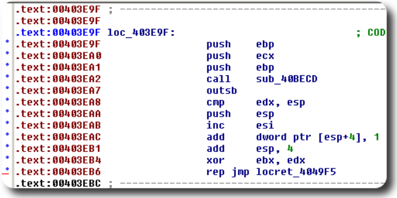
The reference: **IDA Pro**

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

Hypothesis

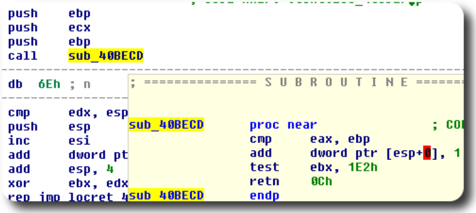
- Both branches of conditionnal jumps are taken
- No overlapping instructions
- The `call` instruction always returns

Hypothesys: call returns




```
.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




```
.text:0040BECD sub_40BECD      proc near      ; CODE XREF: .text:00403EA2

.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

Definition

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

Instruction **ADD**:

```
a = di.instruction.args.symbolic

res = Expression[[a[0], :&, mask], :+, [a[1], :&, mask]]

binding[a[0]] = Expression[res, :&, mask]
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:

```
addr_ret = Expression[di.address, :+, di.bin_length].reduce }
binding = {
   :esp => Expression[:esp, :-, opsz],
   Indirection[:esp, opsz] => addr_ret
}
```

For exemple:

```
dword ptr [esp] = 0x4010CE
esp = esp-4
```

Instruction RDTSC:

```
binding = {
    :eax => Expression::Unknown,
    :edx => Expression::Unknown
}
```

Binding

Instruction CALL:

```
addr_ret = Expression[di.address, :+, di.bin_length].reduce }
binding = {
   :esp => Expression[:esp, :-, opsz],
   Indirection[:esp, opsz] => addr_ret
}
```

For exemple:

```
dword ptr [esp] = 0x4010CE
esp = esp-4
```

Instruction RDTSC:

```
binding = {
    :eax => Expression::Unknown,
    :edx => Expression::Unknown
}
```

Binding

Instruction CALL:

```
addr_ret = Expression[di.address, :+, di.bin_length].reduce }
binding = {
  :esp => Expression[:esp, :-, opsz],
  Indirection[:esp, opsz] => addr_ret
}
```

For exemple:

```
dword ptr [esp] = 0x4010CE
esp = esp-4
```

Instruction RDTSC:

```
binding = {
   :eax => Expression::Unknown,
   :edx => Expression::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 x dword ptr [esp] for 40bedah ret 0ch

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

Securitech 2006 - Challenge 10

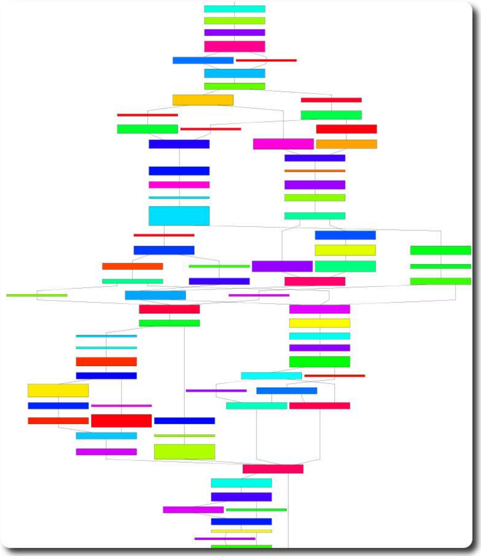
Poeut.exe

- Massively obfuscated binary
- **IDA** overwhelmed
- **Metasm** disassembles correctly, but:
  - Binary blocks are randomely 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

Obscur predicates

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

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

```
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
fild 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

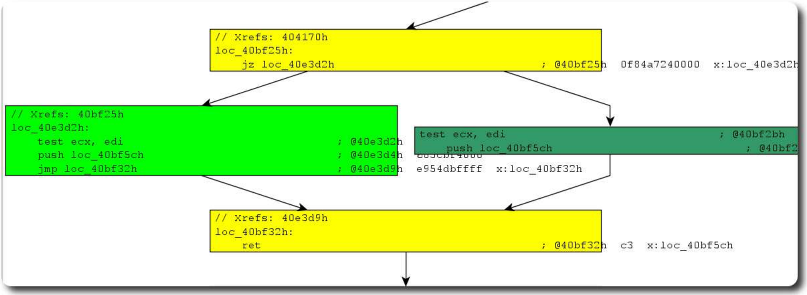
Full randomisation

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

```
if( rand() %2 ) {
    real_code_A;
} else {
    real_code_B;
}
```

Flow duplication

Massive use of random predicates:



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                      ; @402039h  85e3
cmp ebx, edx                       ; @40203bh  39d3
mov byte ptr [edx], al             ; @40203dh  8802
add dword ptr [esp+0], 6           ; @40203fh  8344240006
inc edx                            ; @402044h  42
jmp loc_40b25dh                    ; @402045h  e913920000
```

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

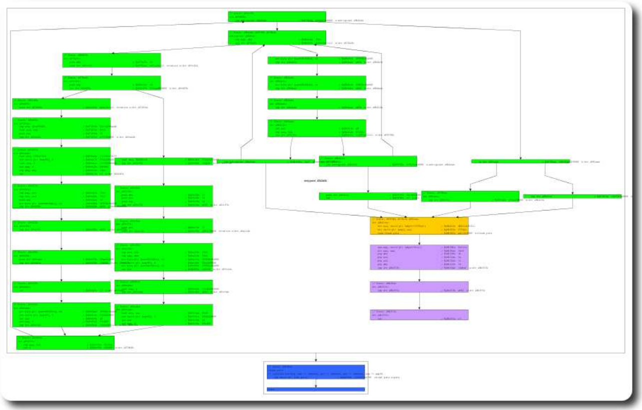
Implémentation :

```
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  83c408
[...]
ret 0ch                 ; @403808h  c20c00  x:loc_40933dh
```

Solution

- Execution flow reconstruction
- Stack emulation
- We have a pattern: return address modification

Epilogue (raw)

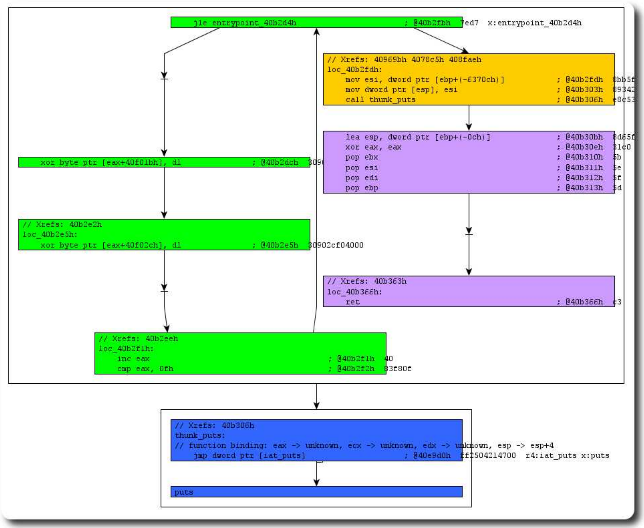


Execution flow analysis

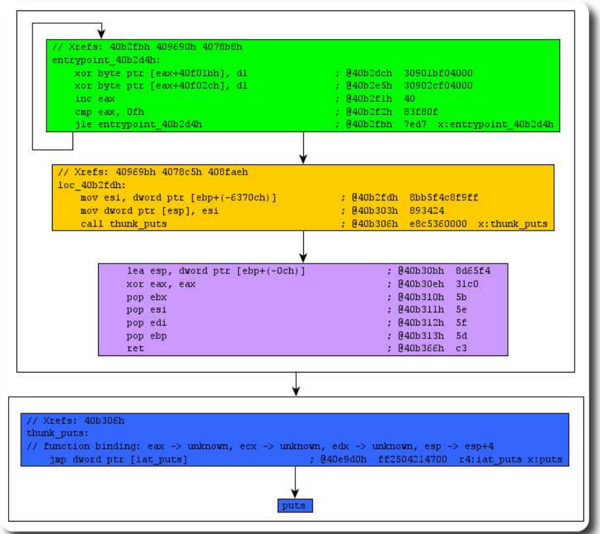
Solution

1. Walk the intenal tree of `InstructionBlock`
2. Inline functions if needed
3. Scan for a diamond pattern
4. Construction, cleaning and comparison of flows
5. Factorisation

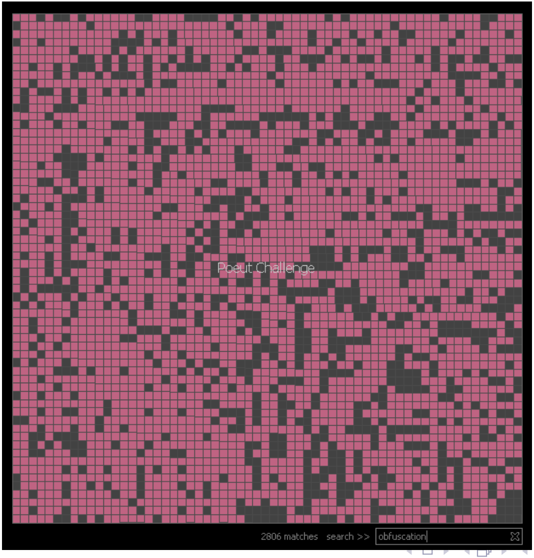
Epilogue (factorized)



Epilogue (reordered)



Epilogue (final)



Last touch

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

Plan

T2 2007

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

junk

```
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_21abah:
 popfd
 pop ebx
 pop esi
```

Virtual machine

Handler structure

- Simple operations
- Similar blueprint
- Operations controlled by [ebp]

[demo/3]

Virtual addition

An addition handler

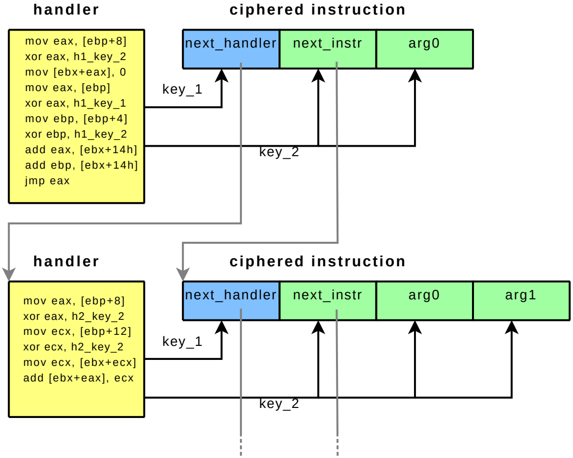
```
loc_15336:
 mov ecx, dword ptr [ebp+0ch]
 xor ecx, 842b1208h
 mov ecx, dword ptr [ebx+ecx]
 mov eax, dword ptr [ebp+8]
 xor eax, 842b1208h
 add dword ptr [ebx+eax], ecx
```

Inter-handler transition

transition to the next handler

```
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 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

```
entrypoint_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)

```
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

Facts

What we have done so far

- Stress was put on binary manipulation : **code rewritting**
- 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 rewritting rules
- Obfuscated code as input
- Clean (obfuscation free) code as output

Means

- What we want to do looks like optimization !
- Extensive litterature 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 knwon patterns with a reduced form

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

We apply them localy, on each basic bloc, spaghetti code has been previously merged.s

Example: constant propagation

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

The constant (0x12) has been propagated through *al*

Example 2: constant folding

Before

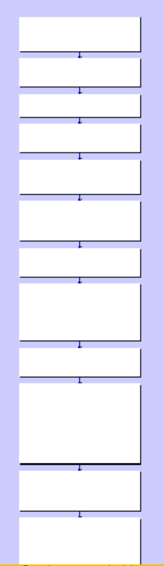
```
100bb5cfh  mov  al, 12h
100bed67h  mov  cl, 46h
100bed69h  xor  cl, 12h
```

After

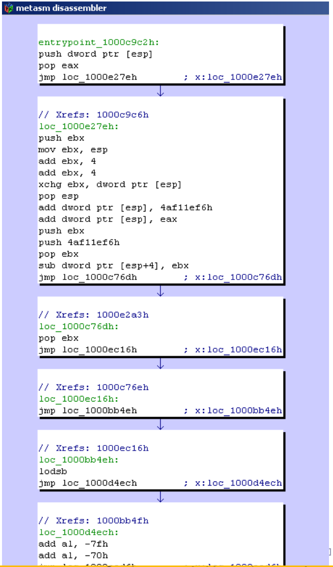
```
100bb5cfh  mov  al, 12h
100bed67h  mov  cl, 54h
```

cl is now assigned with *0x46 xor 0x12h*

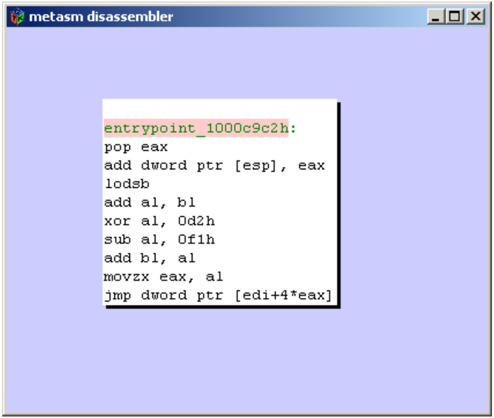
Does it work?



Does it work ?


metasm disassembler
entrypoint_1000c9c2h:
push dword ptr [esp]
pop eax
jmp loc_1000e27eh ; x:loc_1000e27eh
// Xrefs: 1000c9c6h
loc_1000e27eh:
push ebx
mov ebx, esp
add ebx, 4
add ebx, 4
xchg ebx, dword ptr [esp]
pop esp
add dword ptr [esp], 4af11ef6h
add dword ptr [esp], eax
push ebx
push 4af11ef6h
pop ebx
sub dword ptr [esp+4], ebx
jmp loc_1000c76dh ; x:loc_1000c76dh
// Xrefs: 1000e2a3h
loc_1000c76dh:
pop ebx
jmp loc_1000ec16h ; x:loc_1000ec16h
// Xrefs: 1000c76eh
loc_1000ec16h:
jmp loc_1000bb4eh ; x:loc_1000bb4eh
// Xrefs: 1000ec16h
loc_1000bb4eh:
lodsb
jmp loc_1000d4ech ; x:loc_1000d4ech
// Xrefs: 1000bb4fh
loc_1000d4ech:
add al, -7fh
add al, -70h

Does it work ? Actually yes



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 ?