Miasm (incomprehensible documentation)

`serpilliere at droids-corp 0rg`
EADS Corporate Research Center — IW/SE/CS
IT sec Lab
Suresnes, FRANCE

logo/eads

Outline

logo/eads

Outline

logo/eads

Elfesteem use

EXE reading

- EXE parsing
- (MZ/PE/sections/Directories)

```
>>> from elfesteem import *
>>> e = pe_init.PE(open('calc.exe', 'rb').read())
#  section          offset   size     addr     flags    rawsize
 0  .text           00000400 0126b0 00001000 60000020 00012800
 1  .data           00012c00 00101c 00014000 c0000040 00000a00
 2  .rsrc           00013600 008a88 00016000 40000040 00008c00
```

logo/eads

Accesses

File view

```
>>> e.content[:4]
'MZ\x90\x00'
```

RVA view

```
>>> e.drva[0x1000:0x1004]
'\xea"\xdaw'
```

Virtual addesses view

```
>>> e.virt[0x1001000:0x1001004]
'\xea"\xdaw'
```

go/eads

EXE attributes

```
>>> e.DirImport
<Directory Import>
 0 <SHELL32.dll>              <W-ImpDesc=76968/4294967295L/4294967295L/77378/4252>
     0 <148, ShellAboutW>
 1 <msvcrt.dll>               <W-ImpDesc=77256/4294967295L/4294967295L/77664/4540>
     0 <82, __CxxFrameHandler>
     1 <71, _CxxThrowException>
     2 <824, wcstoul>
     ...
```

```
>>> e.DirRes
<ID RT_ICON subdir: 90192 None>
    <ID RT_CURSOR subdir: 90528 None>
        <ID 1036 data: 91152 <ResDataEntry=91576/744/1252/0>>
    <ID RT_BITMAP subdir: 90552 None>
        <ID 1036 data: 91168 <ResDataEntry=92320/296/1252/0>>
    <ID RT_ICON subdir: 90576 None>
    ...
```

logo/eads

Common manipulation

EXE generation

- EXE creation
- Default characteristics

```
e = PE()
open('uu.bin', 'wb').write(str(e))
```

logo/eads

Add a section to a binary

- read the binary
- add a section
- generate binary

```
>>> e = PE(open('calc.exe', 'rb').read())
# section          offset   size     addr     flags    rawsize
 0 .text           00000400 0126b0 00001000 60000020 00012800
 1 .data           00012c00 00101c 00014000 c0000040 00000a00
 2 .rsrc           00013600 008a88 00016000 40000040 00008c00
>>> s_XXX = e.SHList.add_section(name='XXX', addr = 0x20000, rawsize = 0x1000)
>>> open('out.bin', 'wb').write(str(e))

>>> PE(open('out.bin', 'rb').read())
# section          offset   size     addr     flags    rawsize
 0 .text           00000400 0126b0 00001000 60000020 00012800
 1 .data           00012c00 00101c 00014000 c0000040 00000a00
 2 .rsrc           00013600 008a88 00016000 40000040 00008c00
 3 XXX             0001c200 001000 00020000 e0000020 00001000
```

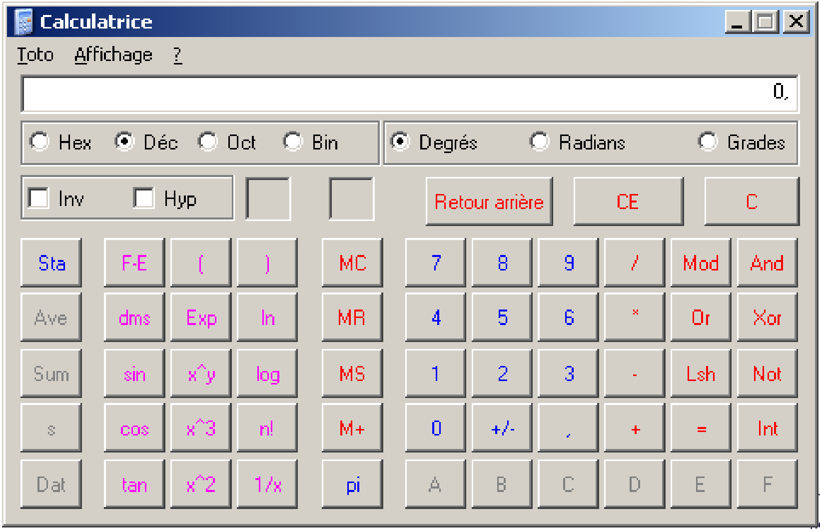
logo/eads

Menu edition

- read DirRes
- find menu
- modify
- generate the binary

```
>>> e = PE(open('calc.exe', 'rb').read())
>>> e.DirRes.resdesc.resentries
 0 <ID RT_ICON subdir: 90192 None>ResEntry
 1 <ID RT_MENU subdir: 90272 None>ResEntry
 ...
>>> menu = e.DirRes.resdesc.resentries[1]
>>> menu.subdir.resentries
 0 <ID 106 subdir: 90720 None>ResEntry
 1 <ID 107 subdir: 90744 None>ResEntry
...

>>> e.Opthdr.Optehdr[pe.DIRECTORY_ENTRY_BOUND_IMPORT].rva = 0
>>> e.Opthdr.Optehdr[pe.DIRECTORY_ENTRY_BOUND_IMPORT].size = 0

>>> e.DirRes.resdesc.resentries[1].subdir.resentries[1].\
    subdir.resentries[0].data.s[8:22:2]
'Edition'
>>> e.DirRes.resdesc.resentries[1].subdir.resentries[1].\
    subdir.resentries[0].data.s[8:22] = "\x00".join([x for x in 'Toto'])+'\x00'
>>> open('out.bin.exe', 'wb').write(str(e))
```

logo/eads


Calculatrice
Toto Affichage ?
0,
Hex Déc Oct Bin
Degrés Radians Grades
Inv Hyp
Retour arrière
CE
C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- , + = Int
Dat tan x^2 1/x pi A B C D E F

Common case

EXE generation

- create an EXE
- default characteristics
- new text section with “xC3” (ret)
- place entry point
- add some imports
- → The binary is ready

logo/eads

```
e = PE()
mysh = "\xc3"
s_text = e.SHList.add_section(name = "text", addr = 0x1000, rawsize = 0x1000, data = mys
e.Opthdr.Opthdr.AddressOfEntryPoint = s_text.addr
new_dll = [({"name":"kernel32.dll",
             "firstthunk":s_text.addr+0x100},
            ["CreateFileA", "SetFilePointer", "WriteFile", "CloseHandle"]
            )
           ,
           ({"name":"USER32.dll",
             "firstthunk":None},
            ["SetDlgItemInt", "GetMenu", "HideCaret"]
            )
           ]
e.DirImport.add_dlldesc(new_dll)
s_myimp = e.SHList.add_section(name = "myimp", rawsize = 0x1000)
e.DirImport.set_rva(s_myimp.addr)
open('uu.bin', 'wb').write(str(e))
```

logo/eads

Ida listing:

```
;*****************************************************************x
;  section 1 <text>                                                x
;  virtual address  00001000  virtual size   00001000              x
;  file offset      00001000  file size      00001000              x
;*****************************************************************x
                                                                   x
;****************************                                      x
;  program entry point                                             x
;****************************                                      x
entrypoint:                                                        x
  ret                                                              x
```

logo/eads

Outline

1. Random manipulations
   - PE
   - Assembleur/Désassembleur
   - Graphe
   - Introduction to intermediate language
2. Langage intermédiaire
   - Description du langage
   - Module de simplification d'expression
   - Symbolic execution
3. Jit compilation
   - Principle
   - Exemples jit
4. Exemples
   - Hooks, gadget finder, ...
   - Conficker
   - mebroot
   - VM Study

logo/eads

Miasm

Goal

- Asm/DisAsm x86/PPC/ARM
- Work on multi sources
- (str/shellcode txt/PE/ELF/.S)
- assembly to intermediate language
- emulate intermediate language (in an environment)
- Snapshot/restore
- library of emulation
- ...

logo/eads

Common cases

Dis/Asm

- work on bytes/asm text/container
- mini integrated linker
- graph generation
- ...

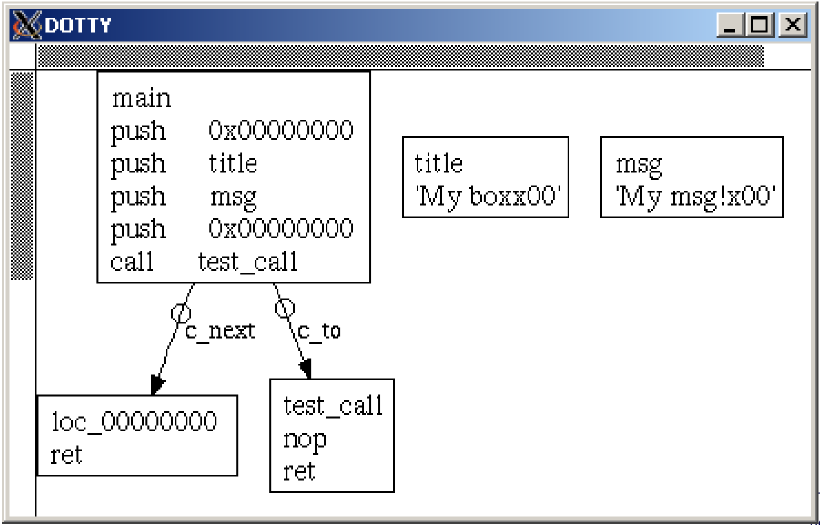
```
>>> from x86_escape.x86_escape import *
>>> l = x86_mn.dis('\x90')
>>> str(l)
'nop      '
>>> x86_mn.asm('nop')
['\x90']
>>> x86_mn.asm('inc eax')
['@', '\xff\xc0']
>>> str(x86_mn.dis('@'))
'inc        eax'
>>> str(x86_mn.dis('\xff\xC0'))
'inc        eax'
```

logo/eads

assembling x86 bloc

```
all_bloc , symbol_pool = parse_asm.parse_txt(x86_escape.x86mnemo, r'''
main:
    push 0
    push title
    push msg
    push 0
    call test_call
    ret
test_call:
    nop
    ret
title:
.string "My box"
msg:
.string "My msg!"
''')


#fix shellcode addr
symbol_pool.add(asmbloc.asm_label('base_address', 0))
symbol_pool.getby_name("main").offset = 0

####graph sc####
g = asmbloc.bloc2graph(all_bloc[0])
open("graph.txt" , "w").write(g)
```

logo/eads


DOTTY
main
push 0x00000000
push title
push msg
push 0x00000000
call test_call
title
'My boxx00'
msg
'My msg!x00'
c_next
c_to
loc_00000000
ret
test_call
nop
ret

logo/eads

shell code generation

```
f = open('out.bin', 'wb')
for p in patches:
    f.seek(p)
    f.write(patches[p])
f.close()
```

logo/eads

Dump hexa

```
x00000000 6a 00 6a 18 6a 10 6a 00-e8 01 00 00 00 c3 90 c3 |j j?j?j ??   ???|
x00000010 4d 79 20 6d 73 67 21 00-4d 79 20 62 6f 78 00    |My msg! My box  |
```

Disassemble

```
x00000000 6a00                    push     0x0
x00000002 6a18                    push     0x18
x00000004 6a10                    push     0x10
x00000006 6a00                    push     0x0
x00000008 e801000000              call     0xe
x0000000d c3                      ret
x0000000e 90                      nop
x0000000f c3                      ret
```

logo/eads

In the next slide...

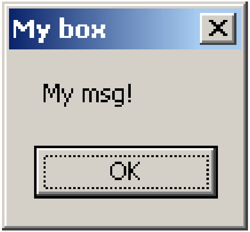
PE generation

- generate a working PE
- with imports
- some code
- which displays a dialog box
- ...

logo/eads

```
#! /usr/bin/env python
from x86_escape import *
from elfesteem import *

e = pe_init.PE()
s_text = e.SHList.add_section(name = "text", addr = 0x1000, rawsize = 0x100)
s_iat = e.SHList.add_section(name = "iat", rawsize = 0x100)
new_dll = [({"name":"USER32.dll","firstthunk":s_iat.addr}, ["MessageBoxA"])]

e.DirImport.add_dlldesc(new_dll)
s_myimp = e.SHList.add_section(name = "myimp", rawsize = len(e.DirImport))
e.DirImport.set_rva(s_myimp.addr)
all_bloc, symbol_pool = parse_asm.parse_txt(x86_escape.x86mnemo,r'''
main:
    push 0
    push title
    push msg
    push 0
    call [MessageBoxA]
    ret
title:
.string "My box"
msg:
.string "My msg!"
''')
symbol_pool.add(asmbloc.asm_label('base_address', 0))
symbol_pool.getby_name("MessageBoxA").offset = e.DirImport.get_funcvirt('MessageBoxA')
symbol_pool.getby_name("main").offset = e.rva2virt(s_text.addr)
resolved_b, patches = asmbloc.asm_resolve_final(x86_escape.x86mnemo, all_bloc[0], symbol
for p in patches:
    e.virt[p] = patches[p]
e.Opthdr.Opthdr.AddressOfEntryPoint = e.virt2rva(symbol_pool.getby_name("main").offset)
open('uu.bin', 'wb').write(str(e))
```

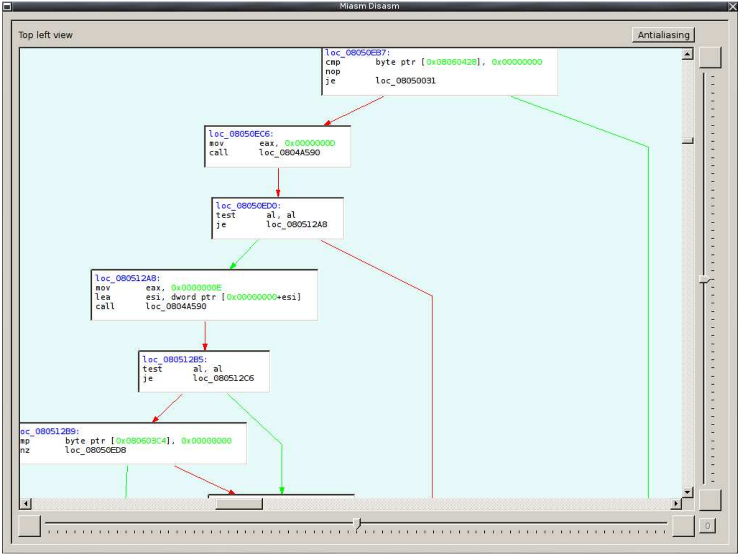
logo/eads



logo/eads

Outline

logo/eads


Miasm Disasm
Top left view
Antialiasing
loc_08050EB7:
cmp byte ptr [0x08060428], 0x00000000
nop
je loc_08050031
loc_08050EC6:
mov eax, 0x0000000D
call loc_0804A590
loc_08050ED0:
test al, al
je loc_080512A8
loc_080512A8:
mov eax, 0x0000000E
lea esi, dword ptr [0x00000000+esi]
call loc_0804A590
loc_080512B5:
test al, al
je loc_080512C6
oc_080512B9:
mp byte ptr [0x080603C4], 0x00000000
nz loc_08050ED8

logo/eads

Outline

logo/eads

Instruction semantic

Intermediate language

- the instruction is composed of operations
- each operation is executed in parallel
- example: cmpsb

```
def cmpsb():
    e= []
    e+=l_cmp(ExprMem(esi, 8), ExprMem(edi, 8))
    e.append(ExprAff(edi, ExprCond(df, ExprOp('+', edi, ExprInt(uint32(1))), ExprOp('-',
    e.append(ExprAff(esi, ExprCond(df, ExprOp('+', esi, ExprInt(uint32(1))), ExprOp('-',
    return e
```

logo/eads

cmpsb semantic

```
>>> from x86_escape import ia32_sem
>>> e = ia32_sem.cmpsb()
>>> for x in e:
...     print str(x)
...
zf = (== (- @8[esi] @8[edi]) 0x0)
nf = (& (== 0x1 (>> (- @8[esi] @8[edi]) 0x7)) 0x1)
pf = (parity (- @8[esi] @8[edi]))
cf = (| (& (== (== 0x1 (>> @8[esi] 0x7)) 0x0) (== 0x1 (>> @8[edi] 0x7))) (& (== 0x1 (>>
of = (| (& (== (== 0x1 (>> (- @8[esi] @8[edi]) 0x7)) 0x1) (& (== (== 0x1 (>> @8[esi] 0x7
af = (== (& (- @8[esi] @8[edi]) 0x10) 0x10)
edi = df?((+ edi 0x1),(- edi 0x1))
esi = df?((+ esi 0x1),(- esi 0x1))
```

logo/eads

Manipulation example

registers touched by an instruction

```
>>> from analysis_helper import *
>>> r, w = get_rw(ia32_sem.cmpsb())
>>> r, w
(set_expr[@8[edi], @8[esi], df, edi, esi], set_expr[zf, nf, pf, cf, of, af, edi, esi])
```

logo/eads

Outline

logo/eads

Expressions

- ExprInt: interger
- ExprId: identifier (variable)
- ExprAff: a = b
- ExprCond: a?b:c
- ExprMem: dword ptr [a]
- ExprOp: op(a, b, ...)
- ExprSlice: a[0:8] (bits)
- ExprCompose: slices composition
- ExprSliceTo: position in a composition

That's all.

logo/eads

Some expressions

```
>>> from expression import *
>>> a = ExprId('A', 32)
>>> b = ExprId('B', 32)
>>> c = ExprId('C', 32)
>>> o = a+b
>>> print o
(A + B)
>>> print ExprAff(c, o)
C = (A + B)
```

Definition of some instructions

```
def mov(a, b):
    return [ExprAff(a, b)]

def xchg(a, b):
    e = []
    e.append(ExprAff(a, b))
    e.append(ExprAff(b, a))
    return e
```

```
def update_flag_zf(a):
    cast_int = tab_uintsize[a.get_size()]
    return [ExprAff(zf, ExprOp('==',
                               a,
                               ExprInt(cast_int(0))))]
def update_flag_nf(a):
    return [ExprAff(nf, ExprOp('&',
            get_op_msb(a),
            ExprInt(tab_uintsize[a.get_size()](1))))]
```

Outline

logo/eads

The language has simplification rules

- X + Y - Y : X
- - (-X) : X
- X +int1 + int2 : X+int3 (=int1 + int2)
- ...

Example

```
>>> print o
(A + B)
>>> p = o - b
>>> print p
((A + B) - B)
>>> print expr_simp(p)
A
```

```
>>> q = (a - ExprInt(uint32(1))) + ExprInt(uint32(3))
>>> print q
((A - 0x1) + 0x3)
>>> print expr_simp(q)
(A + 0x2)
```

logo/eads

Outline

logo/eads

Assembly to intermediate language

- expressions representing instructions are execution simultaneously
- affectations are done into a memory representation
- (a dictionary)
- the key is the identifier (non reducible)

interpretation machine

- It defines usable registers

```
machine = eval_abs({esp:init_esp, ebp:init_ebp, eax:init_eax, ebx:init_ebx,
                    ecx:init_ecx, edx:init_edx, esi:init_esi, edi:init_edi,
                cs:ExprInt(uint32(9)),
                zf : ExprInt(uint32(0)), nf : ExprInt(uint32(0)),
                pf : ExprInt(uint32(0)), of : ExprInt(uint32(0)),
                cf : ExprInt(uint32(0)), tf : ExprInt(uint32(0)),...
                tsc1: ExprInt(uint32(0)), dr7:ExprInt(uint32(0)),...},
               mem_read_wrap,
               mem_write_wrap, )
```

/eads

Example

```
>>> l = x86_mn.dis("\x43")
>>> print l
inc       ebx
>>> ex = get_instr_expr(l, eip)
>>> for e in ex:
...   print e
...
zf = ((ebx + 0x1) == 0x0)
nf = ((0x1 == ((ebx + 0x1) >> 0x1F)) & 0x1)
...
ebx = (ebx + 0x1)
```

Example

```
>>> machine = eval_abs(dict(init_state))
>>> print machine.pool[ebx]
init_ebx
>>> my_eip, mem_dst = emul_full_expr(ex, l, ExprInt(uint32(0)), None, machine)
>>> print machine.pool[ebx]
(init_ebx + 0x1)
>>> print machine.pool[zf]
((init_ebx + 0x1) == 0x0)
```

Outline

logo/eads

Memory

- The memory is defined by zones
- size, accesses and data
- (for example a binary section)
- it maps *real addresses* to *virtual addresses*

translation

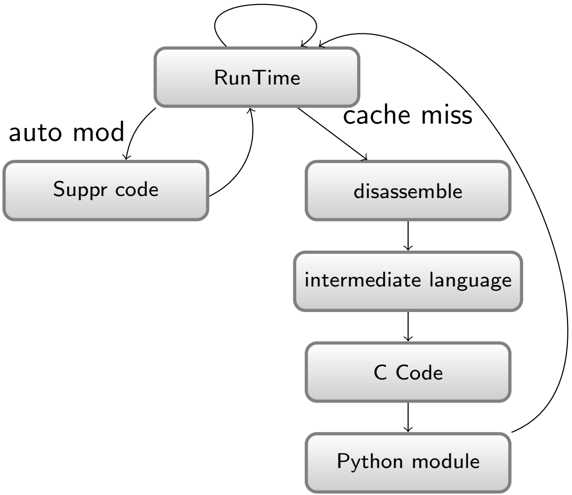
- the code is disassembled
- translated on the fly using semantic representation
- and intermediate code is generated to C code and compiled

logo/eads

Emulation exception

- if a translated bloc is modified, it is deleted from cache
- and regenerated if pc reach it again (cache miss)
- if errors, python is called back to deal it
- for example, we can emulate windows SEH mechanism
- Emulation des api

logo/eads


RunTime
auto mod
cache miss
Suppr code
disassemble
intermediate language
C Code
Python module

logo/eads

C translation

```
unsigned int bloc_0080400B(void)
{
 loc_0080400B:
    //pop        eax
    vmcpu.eax_new = MEM_LOOKUP_32(vmcpu.esp);
    vmcpu.esp_new = (((vmcpu.esp&0xffffffff) + (0x4&0xffffffff))&0xffffffff);

    if (vmcpu.vm_exception_flags > EXCEPT_NUM_UDPT_EIP) {
        vmcpu.eip = 0x80400B;
        return (unsigned int)vmcpu.eip;
    }

    vmcpu.eax = vmcpu.eax_new;
    vmcpu.esp = vmcpu.esp_new;

    if (vmcpu.vm_exception_flags) {
        vmcpu.eip = (vmcpu.vm_exception_flags > EXCEPT_NUM_UDPT_EIP) ? 0x80400B : 0x804
        return (unsigned int)vmcpu.eip;
    }

 loc_0080400C:
    //mov        ebx, eax
    vmcpu.ebx_new = vmcpu.eax;

    if (vmcpu.vm_exception_flags > EXCEPT_NUM_UDPT_EIP) {
        vmcpu.eip = 0x80400C;
        return (unsigned int)vmcpu.eip;
    }
```

logo/eads

translation, memory accesses

Assembly code

```
movzx    eax, ds:byte_410360[ecx]
```

C code

```
//movzx     eax, byte ptr [ecx+{<asmlabel (unsigned int)&tab_00410340[0x20] >: 1}]
eax_new = (((0x0 & (0xFFFFFFFF>>(32-24))) << 8) | ((MEM_LOOKUP(8, (((ecx&0xffffffff) +
        ((unsigned int)&tab_00410340[0x20]&0xffffffff))&0xffffffff)) & (0xFFFFFFFF>>(
eax = eax_new;
```

logo/eads

Outline

logo/eads

Demo: assembly code

```
objdump -D -b binary -m i386   -Maddr32,data32,intel sc_test.bin
sc_test.bin:     file format binary
Disassembly of section .data:
00000000 <.data>:
   0:   b8 ef be 37 13          mov    eax,0x1337beef
   5:   b9 04 00 00 00          mov    ecx,0x4
   a:   c1 c0 08                rol    eax,0x8
   d:   e2 fb                   loop   0xa
   f:   c3                      ret
```

logo/eads

Memory creation

```
code_ad = 0x20000
vm_add_memory_page(code_ad, PAGE_READ|PAGE_WRITE|PAGE_EXEC, open("sc_test.bin").read())
stack_base_ad = 0x1230000
stack_size = 0x10000
vm_add_memory_page(stack_base_ad, PAGE_READ|PAGE_WRITE, "\x00"*stack_size)
dump_memory_page_pool_py()

regs = vm_get_gpreg()
regs['eip'] = code_ad
regs['esp'] = stack_base_ad+stack_size
vm_set_gpreg(regs)
dump_gpregs_py()
```

Result

```
Memory
ad 00020000 size 00000025 RWX hpad 0x8cd5000
ad 01230000 size 00010000 RW_ hpad 0x8ce8000
Registers
eip 00020000 eax 00000000 ebx 00000000 ecx 00000000 edx 00000000
esi 00000000 edi 00000000 esp 01240000 ebp 00000000
```

ɔ/eads

Memory creation

```
vm_push_uint32_t(0)
vm_push_uint32_t(0)
vm_push_uint32_t(0x1337beef)

symbol_pool = asmbloc.asm_symbol_pool()

known_blocs = {}
code_blocs_mem_range = []

log_regs = True
log_mn = True
must_stop = False
def run_bin(my_eip, known_blocs, code_blocs_mem_range):
    while my_eip != 0x1337beef:
        if not my_eip in known_blocs:
            updt_bloc_emul(known_blocs, in_str, my_eip, symbol_pool, code_blocs_mem_rang
        try:
            my_eip = vm_exec_blocs(my_eip, known_blocs)
        except KeyboardInterrupt:
            must_stop = True
        py_exception = vm_get_exception()
        if py_exception:
            raise ValueEror("except at", hex(my_eip))

print "start run"
run_bin(code_ad, known_blocs, code_blocs_mem_range)
```

/eads

Result

```
                                eax 00000000 ebx 0000...
mov        eax, 0x1337BEEF                          ...
                                eax 1337BEEF ebx 0000...
mov        ecx, 0x00000004                          ...
                                eax 1337BEEF ebx 0000...
rol        eax, 0x00000008                          ...
                                eax 37BEEF13 ebx 0000...
loop       loc_0002000A                             ...
loc_0002000A                                        ...
                                eax 37BEEF13 ebx 0000...
rol        eax, 0x00000008                          ...
                                eax BEEF1337 ebx 0000...
loop       loc_0002000A                             ...
loc_0002000A                                        ...
                                eax BEEF1337 ebx 0000...
rol        eax, 0x00000008                          ...
                                eax EF1337BE ebx 0000...
loop       loc_0002000A                             ...
loc_0002000A                                        ...
                                eax EF1337BE ebx 0000...
rol        eax, 0x00000008                          ...
                                eax 1337BEEF ebx 0000...
loop       loc_0002000A                             ...
loc_0002000F                                        ...
                                eax 1337BEEF ebx 0000...
ret
```

/eads

Interaction

```
>>> vm_get_gpreg()
{'eip': 322420463, 'esp': 19136504, 'edi': 0, 'eax': 322420463,
  'ebp': 0, 'edx': 0, 'ebx': 0, 'esi': 0, 'ecx': 0}
>>> vm_get_str(code_ad, 0x10)
'\xb8\xef\xbe7\x13\xb9\x04\x00\x00\x00\xc1\xc0\x08\xe2\xfb\xc3'
```

logo/eads

Outline

logo/eads

Goal: hook in calc.exe

- spot interesting code
- find characteristic code
- generate a hook

logo/eads

Code search

```
#op code call[XXX]
p = "\xFF\x15"+ struct.pack('L', ad_setwtext)
p = re.escape(p)
candidates = [x.start() for x in re.finditer(p, e.content)]
candidates = [e.off2virt(x) for x in candidates]

#search func setdisplaytext
found = False
for c in candidates:
    #ad = guess_func_start(e, c)
    job_done = set()
    symbol_pool = asmbloc.asm_symbol_pool()
    try:
        all_bloc = asmbloc.dis_bloc_all(x86_mn, in_str, c, job_done, symbol_pool)
    except:
        continue
    #filter on setfocus caller
    for b in all_bloc:
        l = b.lines[-1]
        if not l.m.name == "call" or not x86_afs.imm in l.arg[0]:
            continue
        if l.arg[0][x86_afs.imm] == ad_setfocus:
            found = c
if not found:
    raise ValueError("caanot finc setdisplaytext")
ad = guess_func_start(e, found)
print "setdisplaytext:", hex(ad)
```

logo/eads

Hook example

```
h.add_hook(aes_init_ad,
           {
               "1_aes_init_key":(0x20, "push ecx"),
               })
...
h.add_hook(rsa_enc_priv_ad,
           {
            "1_seckey":(0x140, 'push [esp+0x34]'),
            "2_txt":(0x80, 'push [esp+0x34]'),
            "3_mod":(0x80, 'push [esp+0x34]')
               })
...
h.add_hook(pwd_chk_ad,
           {"1_pwd":('''
             push [ebp+8]
             call [lstrlenA]
             push eax''', 'push ecx'),
            "2_stack":(0x80, 'push [esp+0x30]')})
```

logo/eads

Hook creation

```
h = hooks(in_str, symbol_pool, gen_data_log_code = False)

hname = h.add_hook(ad,
           {
               "1_DONT_LOG":('''
                mov eax, [ebp+8]
                cmp byte ptr [eax], 0x31
                jnz out
                cmp byte ptr [eax+2], 0x33
                jnz out
                cmp byte ptr [eax+4], 0x33
                jnz out
                cmp byte ptr [eax+6], 0x37
                jnz out
                cmp byte ptr [eax+6], 0x2c
                jnz out
                push 0
                push mtitle
                push mtxt
                push 0
                call [MessageBoxA]


            out:
                ''', 'push [esp+0x30]')},
           ['mtitle:\n.string "title"', 'mtxt:\n.string "txt"'])
```

logo/eads

Binary modification

```
all_bloc = h.all_bloc

symbol_pool.add(asmbloc.asm_label('base_address', 0))
symbol_pool.getby_name("MessageBoxA").offset = e.DirImport.get_funcvirt('MessageBoxA')
symbol_pool.getby_name(hname).offset = e.rva2virt(sh_ad)

symb_reloc =
#compilation du patch
resolved_b, patches = asmbloc.asm_resolve_final(x86mnemo, all_bloc[0], symbol_pool, [(0, sh_ad+e.Opthdr.Op
add_rels = []
#ajout des nouvelles relocations
for l, rels in symb_reloc.items():
    for x in rels:
        add_rels.append(e.virt2rva(x+l.offset))
#ajout des nouveaux imports
s_myimp = e.SHList.add_section(name = "myimp", rawsize = len(e.DirImport))
e.DirImport.set_rva(s_myimp.addr)
#patch du binaire
for p in patches:
    e.virt[p] = patches[p]
#reconstruction du binaire
open('calc_mod.exe', 'wb').write(str(e))
```

logo/eads

Go Go Gadget

Goal: find eip/esp control

- binary mapping into memory
- Memory creation
- add context informations to a program
- start interpretation:
  - sweep on each code address range
  - execution maximum 10 instruction (for instance)
  - analyzes symbolic memory
  - filter interesting results

logo/eads

Loading of a binary dump

```
data = open(fname, 'rb').read()
in_str = bin_stream_vm()
init_memory_page_pool_py()
init_code_bloc_pool_py()
vm_add_memory_page(0x10000000, PAGE_READ|PAGE_WRITE, data)
```

variable creation

```
arg1 = ExprId('ARG1', 32, True)
arg2 = ExprId('ARG2', 32, True)
ret1 = ExprId('RET1', 32, True)
```

machine creation

```
machine = eval_abs({esp:init_esp, ebp:init_ebp, eax:init_eax, ebx:init_ebx,
                    ecx:init_ecx, edx:init_edx, esi:init_esi, edi:init_edi,
                cs:ExprInt(uint32(9)),
                zf :  ExprInt(uint32(0)), nf :  ExprInt(uint32(0)),
                of :  ExprInt(uint32(0)), cf :  ExprInt(uint32(0)),
                ...
```

add context execution

```
machine.eval_instr(push(arg2))
machine.eval_instr(push(arg1))
machine.eval_instr(push(ret1))
machine.eval_instr(push(ebp))
machine.eval_instr(mov(ebp, esp))
machine.eval_instr(sub(esp, ExprInt(uint32(0x14))))
machine.eval_instr(mov(eax, ExprMem(ebp + ExprInt(uint32(8)))))
machine.eval_instr(mov(edx, ExprMem(eax + ExprInt(uint32(12)))))
machine.eval_instr(mov(eax, ExprMem(ebp + ExprInt(uint32(12)))))
machine.eval_instr(mov(ExprMem(esp), eax))
machine.eval_instr(push(ExprInt(uint32(0x1337beef))))
```

(it could have been emulated as well)

```
print dump_reg(machine.pool)
eax ARG2  ebx init_ebx  ecx init_ecx  edx @32[(+ ARG1 0xC)]
esi init_esi  edi init_edi  esp (init_esp - 0x28)
ebp (init_esp - 0x10)  zf ((init_esp - 0x24) == 0x0)
```

logo/eads

Filter results after symbolic execution

```
myesp = machine.pool[esp]
if not ('ARG' in str(myesp) or 'DATA' in str(myesp)):
    continue
print "eip", my_eip
print "esp", myesp
```

Result

```
0x10002ccf
eip @32[ARG2]
esp (ARG2 + 0x4)
Instructions:
xchg        eax, esp
ret
```

logo/eads

Outline

logo/eads

Binary protection

- packed
- the packer is split with indirect jmps

Result

- Ida knowns each basic bloc
- but cannot graph

Quick Counter measure

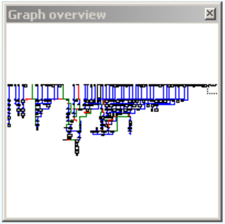
- find each indirect jumps
- find destination
- patch the binary

logo/eads

```
loc_100020A3:
fabs
fstp    dbl_100165D8
mov     eax, dword_10006234
inc     ecx
rol     eax, 18h
mov     dword_10016860, eax
mov     dword_10016498, ecx
mov     edx, dword_100162C0
push    ebx
pop     eax
sub     eax, edx
sub     edx, edx
div     dword ptr [ebp-34h]
cmp     ecx, eax
jmp     off_100163A0
```

```
loc_100020EE:
add     ecx, ebx
mov     ebx, dword_10016438
add     ecx, eax
sub     eax, ebx
push    dword ptr [ecx]
pop     edx
jmp     off_100061B4
```

```
loc_100021F4:                ; lpAddend
push    lpAddend
mov     dword_10016618, 59D90000h
call    ds:InterlockedIncrement
mov     dword_100169E0, eax
mov     ecx, dword_10016288
mov     eax, dword_100164E0
jmp     off_10016368
```



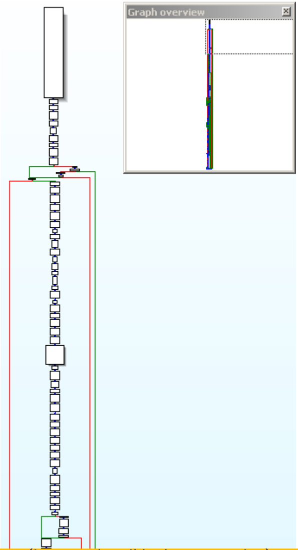
logo/eads

indirect jump patching, binary regeneration

```
e = pe_init.PE(open('conficker.ese', 'rb').read())
s_text = e.getsectionbyname('.text')
s_data = e.getsectionbyname('.data')

for ad in xrange(ad_start, ad_end-15):
    l1 = x86_mn.dis(e.virt[ad:ad+15])
    if not l1: continue
    if not l1.m.name == 'jmp': continue
    if l1.prefix: continue

    a = l1.arg[0]
    if not x86_afs.ad in a or not x86_afs.imm in a:
        continue
    dst_ptr = a[x86_afs.imm]
    if not (data_start <= dst_ptr < data_end): continue
        continue

    dst_ad = struct.unpack('L', e.virt[dst_ptr:dst_ptr+4])[0]
    if not( ad_start <= dst_ad < ad_end): continue

    e.virt[ad] = '\x90'*l1.l
    e.virt[ad] = "\xE9"+struct.pack('l', dst_ad-(ad-1+l1.l))

open('out.bin', 'wb').write(str(e))
```

logo/eads

Result: graph ok, but direct basic blocs are still splitted



logo/eads

Disassembling: main function

```
job_done = []
symbol_pool = asmbloc.asm_symbol_pool()
all_bloc = asmbloc.dis_bloc_all(x86_mn, in_str, 0x100041f4, job_done, symbol_pool, follo

#find call address
for b in all_bloc:
    l = b.lines[-1]
    if l.m.name != 'call': continue

    a = l.arg[0]
    if not x86_afs.symb in a: continue

    dst, off = dict(a[x86_afs.symb]).popitem()
    new_bloc = asmbloc.dis_bloc_all(x86_mn, in_str, off, job_done, symbol_pool, follow_c
    all_bloc+=new_bloc

lbl_start = symbol_pool.getby_offset(oep)
bloc_merge(all_bloc, symbol_pool, [lbl_start])
```

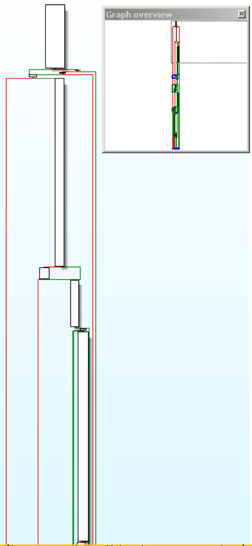
logo/eads

clean binary regeneration

```
#code to PIC
for b in all_bloc:
    del symbol_pool.s_offset[b.label.offset]
    b.label.offset = None

#patch entry point
all_bloc2, symbol_pool2 = parse_asm.parse_txt(x86_mn,r'''
dllentry:
jmp main
''')

#fix shellcode addr
#symbol_pool.add(asmbloc.asm_label('base_address', 0))
symbol_pool2.getby_name("main").offset = 0x10001000
symbol_pool2.getby_name("dllentry").offset = 0x1000434B

#merge our sc and disassembled function
all_bloc+=all_bloc2[0]
for x in symbol_pool2.s.keys():
    symbol_pool.add(symbol_pool2.s[x])
...
open('out.bin', 'wb').write(str(e))
```

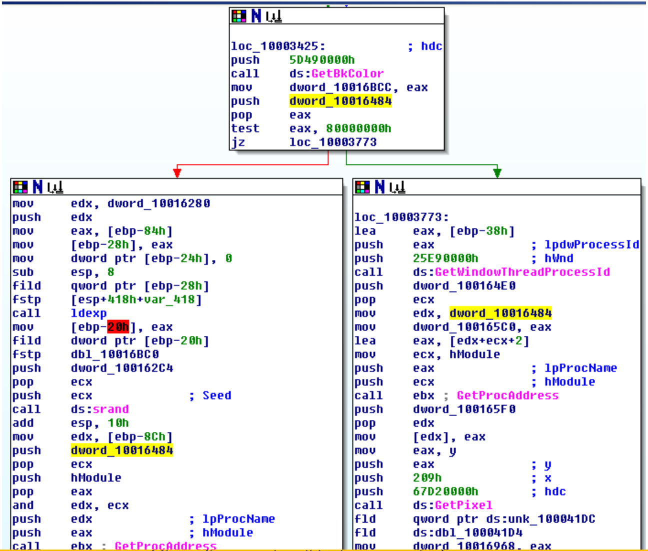
logo/eads

Result



logo/eads

We can analyze import functions loader



logo/eads

Dump and binary reconstruction

```
e = pe_init.PE()
e.Opthdr.Opthdr.ImageBase = 0x3a0000

data = open('_003A0000.mem', 'rb').read()
s_text = e.SHList.add_section(name = "text", addr = 0x0, data = data)
...
e.DirImport.add_dlldesc(new_dll)
s_myimp = e.SHList.add_section(name = "myimp", rawsize = len(e.DirImport))
e.DirImport.set_rva(s_myimp.addr)

open('out.bin', 'wb').write(str(e))
```

logo/eads

binary reconstruction

logo/eads

Outline

logo/eads

Obfuscated packer

```
.text:00404E60 loc_404E60:                             ; CODE XREF: .text:00404ADD↑j
.text:00404E60                 popf
.text:00404E61                 push    0
.text:00404E63                 pushf
.text:00404E64                 push    eax
.text:00404E65                 mov     ax, ds:word_404E6D
.text:00404E65 ; ---------------------------------------------------------------------------
.text:00404E6B                 db 66h
.text:00404E6C                 db 0A9h
.text:00404E6D word_404E6D     dw 2801h                ; DATA XREF: .text:00404E65↑r
.text:00404E6F                 db 58h
```

Packer obscurci

```
.text:00404E60 loc_404E60:                              ; CODE XREF: .text:00404ADD↑j
.text:00404E60                 popf
.text:00404E61                 push    0
.text:00404E63                 pushf
.text:00404E64                 push    eax
.text:00404E65                 mov     ax, word ptr ds:loc_404E6B+2
.text:00404E6B
.text:00404E6B loc_404E6B:                              ; DATA XREF: .text:00404E65↑r
.text:00404E6B                 test    ax, 2801h
.text:00404E6F                 pop     eax
.text:00404E70                 jnz     near ptr dword_404CA4+7
.text:00404E76                 popf
.text:00404E77                 mov     edx, [ebx+20h]
.text:00404E7A                 pushf
.text:00404E7B                 push    eax
.text:00404E7C                 mov     ax, ds:word_404E84
.text:00404E7C ; ---------------------------------------------------------------------------
.text:00404E82                 dw 0A966h
```

logo/eads

Reconstruction

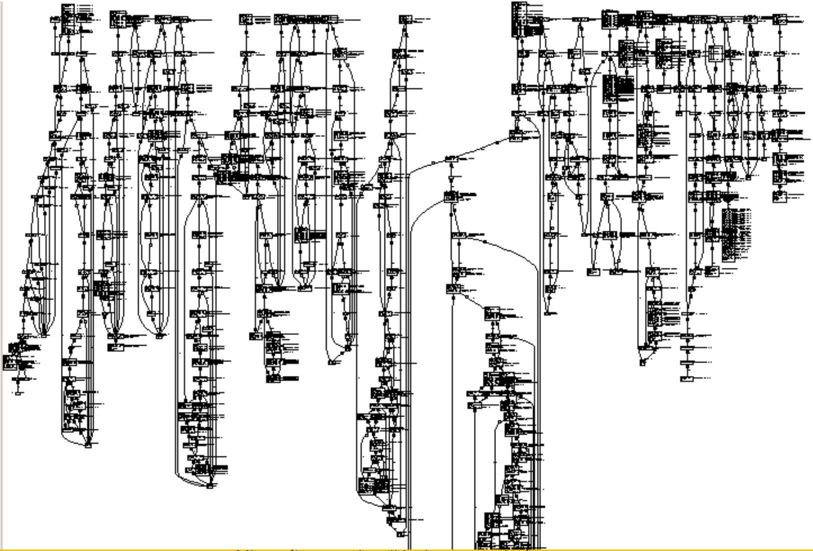
- goal: find non analyzed code
- Disassemble a bloc
- guess fake jcc destination
- group and simplify blocs
- regenerate binary

logo/eads

```
def get_mebbloc_instr(e, b):
    if not b.lines:
        return None
    if b.lines[-1].m.name != "jnz":
        return None
    if b.lines[-2].m.name != "pop":
        return None
    if b.lines[-3].m.name != "test":
        return None
    ...
        asmbloc.dis_bloc(x86mnemo, in_str, b, ad, job_done, symbol_pool, follow_call = F
...
asmbloc.bloc_merge(master_bloc, symbol_pool, call_ad)
```

logo/eads

Code reconstruit



ogo/eads

re generated binary mapping

logo/eads

Entry point

```
public start
start proc near

; FUNCTION CHUNK AT 00401F7B SIZE 00000043 BYTES
; FUNCTION CHUNK AT 00402D34 SIZE 0000002B BYTES

push    0                ; nSize
push    0                ; lpBuffer
push    offset Name      ; "eeacbajjhfggidfhfeicaccjeajcaajjbchgdhh"...
call    ds:GetEnvironmentVariableW
test    eax, eax
pushf
jz      short loc_404AE8
```

```
loc_404AE8:
popf
push    0                ; hTemplateFile
push    0                ; dwFlagsAndAttributes
push    0                ; dwCreationDisposition
push    0                ; lpSecurityAttributes
push    0                ; dwShareMode
push    0                ; dwDesiredAccess
push    offset FileName  ; "ibjecdeiffjdhfaghjcdjhjfcfcjdehjjfchijg"...
call    ds:CreateFileA
test    eax, eax
pushf
jnz     short loc_404B09
```

```
loc_404B09:
popf
push    0                ; lpOverlapped
push    0                ; lpNumberOfBytesWritten
push    0                ; nNumberOfBytesToWrite
push    0                ; lpBuffer
push    929CD202h        ; hFile
call    ds:WriteFile
test    eax, eax
pushf
jz      loc_402D34
```

logo/eads

Analysis

Reconstruction

- The binary brute force its own key (hash)
- Decipher itself
- and decompress

We could borrow its own code

- Disassemble deciphering code
- Disassemble decompression code
- Generate C code
- and execute it on another ciphered code

logo/eads

Deciphering function emulation

```
bsize: 33858
bufs 0x976a000 0xb4bac008
starting...
dyn_call to B4E6B480
dyn_call to B4E6B300
nop func called
...
dyn_call to B4E6B480
dyn_call to B4E6B300
nop func called
dyn_call to B4E6B480
dyn_call to B4E6B300
nop func called
dyn_call to B4E6B480
dyn_call to B4E6B300
nop func called
88B308C5 88B308C5
C4FB632B C4FB632B
AA94D763 AA94D763
5C2BB68F 5C2BB68F
end
ret len 33858
```

logo/eads

Decompression function code

```
bsize: 33858
bufs 0x947b000 0xb46b1008
starting...
dyn_call to B4AFC3F0
func alloc 828B0 (ret B462E008)
dyn_call to B4AFC5F0
dyn_call to B4AFC3F0
func alloc 3E6C (ret 94AF600)
dyn_call to B4AFC9D0
dyn_call to B4AFC350
func free 94AF600 (from 401FB5)
ret: 'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff...'
```

logo/eads

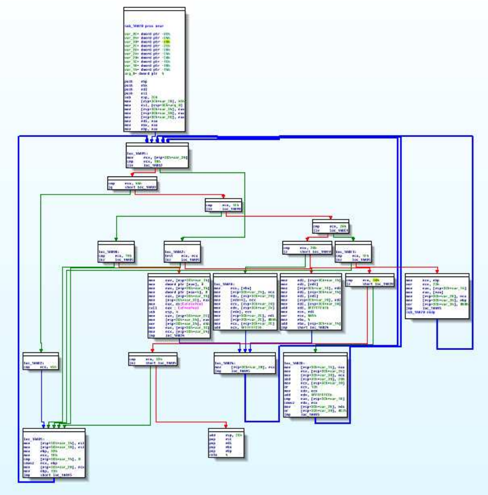
Final step

Second layer

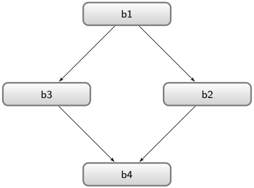
- The binary generate a new binary (driver)
- It is packed as well
- Redo previous steps
- or use previous functions to decipher new binary

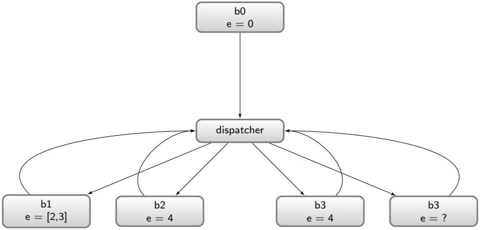
logo/eads

Obfuscated functions



logo/eads


b1
b3
b2
b4

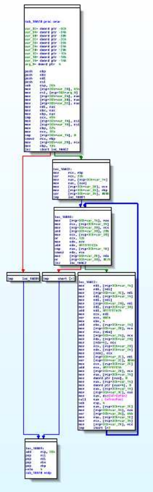

b0
e = 0
dispatcher
b1
e = [2,3]
b2
e = 4
b3
e = 4
b3
e = ?

Reconstruction

- disassemble a function
- get semantic of each bloc
- symbolic execution
- get each bloc result of the automata
- patch jcc
- regenerate binary

logo/eads

Résultat



logo/eads

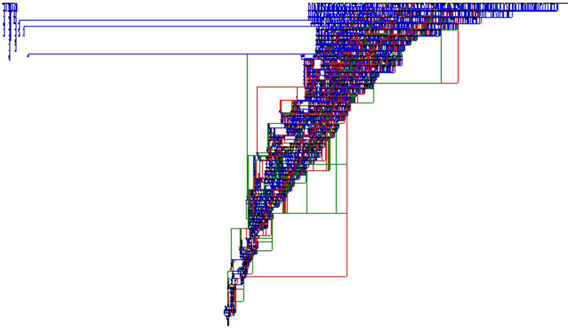
Outline

logo/eads

first layer

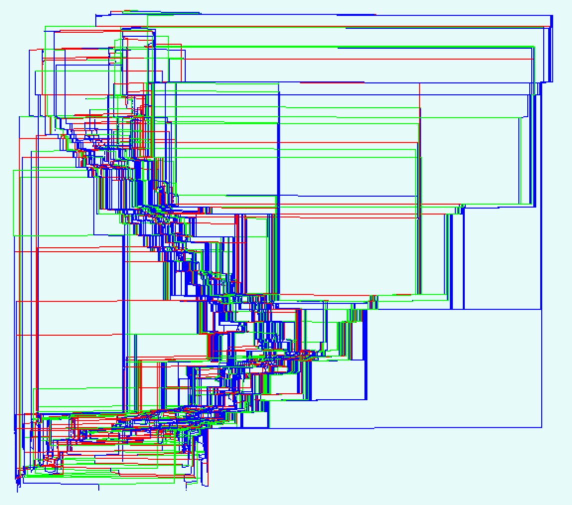
Packer

- The binary is ciphered by layers.
- Just create an environment and emulate it with miasm.



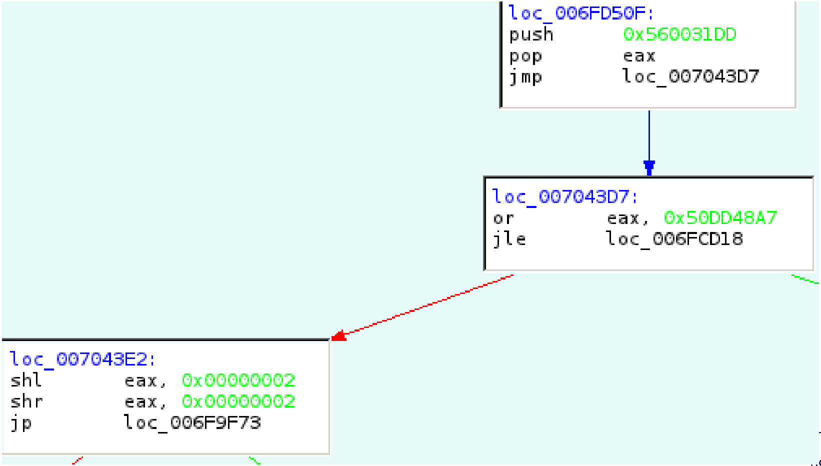
logo/eads

First disassembling of vm mnemonic parsing



logo/eads

Obfuscation

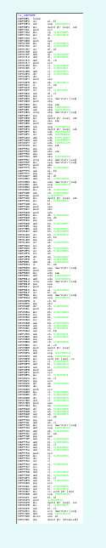


Solution

- Callback is the disassembler engine
- symbolic execution of its parents
- Test if jcc is always true/false
- and delete fake edges

logo/eads

Result: simplified mnemonic parser



logo/eads

End of parser: instruction dispatcher

```
006FF76B   add     cl, 0x0000002A
006FF76E   add     bl, 0x00000005
006FADF6   add     bl, cl
006FADF8   sub     bl, 0x00000005
00706460   pop     ecx
00706461   add     bl, 0x000000D9
00706464   sub     bl, 0x00000070
00706467   mov     cx, word ptr [esp]
0070646B   add     esp, 0x00000002
00706471   sub     bl, al
00706473   push    0x00003F4A
00706478   mov     dword ptr [esp], ecx
0070647B   mov     cl, 0x00000015
0070647D   sub     bl, cl
007020CC   mov     ecx, dword ptr [esp]
00707246   add     esp, 0x00000004
00707249   movzx   eax, al
0070724C   jmp     dword ptr [4*eax+edi]
```

logo/eads

symbolic execution: touched variables

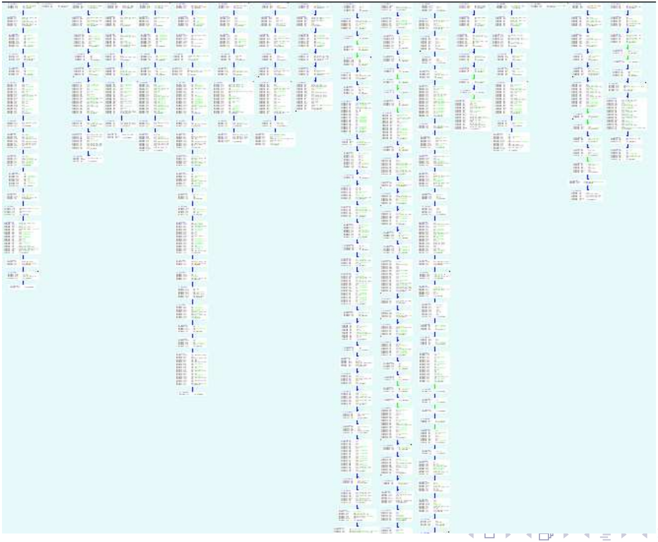
```
eax = ((((@8[init_esi] ^ init_ebx[0:8]) - 0xA8) ^ 0x1)_to[0:8], 0x0_to[8:32])
ebx = ((init_ebx[0:8] - (((@8[init_esi] ^ init_ebx[0:8]) - 0xA8) ^ 0x1))_to[0:8], init_ebx[8:32]_to[8:32])
esi = (init_esi + 0x1)
DST @32[((((((@8[init_esi] ^ init_ebx[0:8]) - 0xA8) ^ 0x1)_to[0:8], 0x0_to[8:32]) * 0x4) + init_edi)]
```

Note

- We need to follow modifications of ebx, esi, edi in each vm mnemonic
- Those modifications are needed to known where disassembling next mnemonic

logo/eads

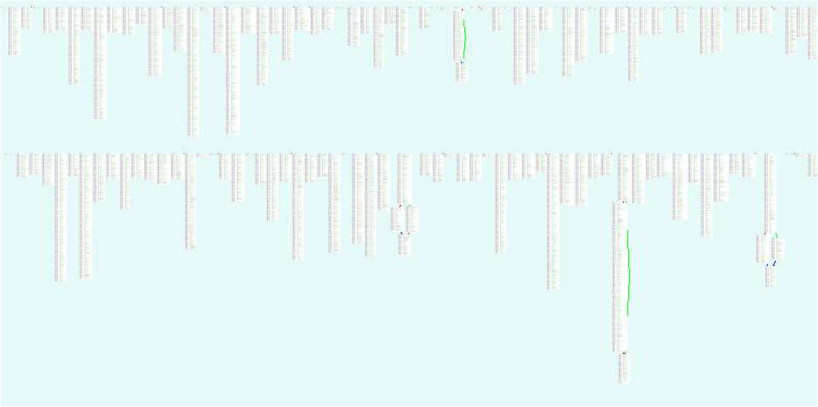
Disassembling correction



Bloc grouping

logo/eads

More than 150 vm mnemonic



logo/eads

Bloc analysis: erf

```
loc_0070AE67:
0070AE67  mov    cx, word ptr [esp]
0070AE6B  push   0x00005001
0070870D  mov    dword ptr [esp], eax
00708710  push   esp
00708711  mov    eax, dword ptr [esp]
00708714  add    esp, 0x00000004
00708717  push   0x00006EA1
0070871C  mov    dword ptr [esp], ecx
0070206C  push   esi
0070206D  mov    esi, 0x0A222CE7
00702072  shl    esi, 0x00000001
00702074  xor    esi, 0x2EC60A82
0070207A  mov    ecx, esi
0070207C  pop    esi
0070207D  push   eax
0070207E  mov    eax, 0x737134A1
006FE368  dec    eax
006FE369  inc    eax
006FE36A  not    eax
006FC2E6  add    eax, 0x62D26028
006FC2EB  sub    eax, 0xB67DF840
006FC2F0  xor    ecx, eax
006FC2F2  pop    eax
006FD440  or     ecx, 0x585D31CE
006FD446  sub    ecx, 0x54EB6CD4
006FD44C  dec    ecx
```

logo/eads

But symbolic execution (again :p)

```
Registers after a bloc execution
 eax init_eax
 ebx init_ebx
 ecx (@16[init_esp]_to[0:16], init_ecx[16:32]_to[16:32])
 edx init_edx
 esi init_esi
 edi init_edi
 esp (init_esp-0x2)
 ebp init_ebp

stack modification:
 @16[(init_esp+0x2)]    (@16[(init_esp+0x2)]<<(@8[init_esp]&0x1F))
 @32[(init_esp-0x2)]    ((@8[init_esp]&0x1F)?(0x0,((@16[(init_esp+0x2)]>>(0x10-(@8[init_esp]&0x1F)))&0x1)

Result:
a = pop16
b = pop16
push16(a<<b)
push32(eflag)
```

logo/eads

Another example

```
---------- 12 0x6fab03 ----------
eax = @32[init_esp]
@32[(init_esp+0x4)]    (@32[(init_esp+0x4)]-@32[init_esp])
@32[init_esp]    (flags(@32[(init_esp+0x4)]-@32[init_esp]))
---------- 13 0x6fb100 ----------
eax = @32[init_esp]
esp = (init_esp+0x4)
@32[(init_esp+0x4)]    (@32[(init_esp+0x4)]^@32[init_esp])
---------- 15 0x6fb3b2 ----------
@32[(init_edi+0x1C)]    (@32[(init_edi+0x1C)]&0xFFFFFBFF)
---------- 19 0x6fb6b8 ----------
ecx = @32[init_esp]
esp = (init_esp+0x4)
@32[(init_esp+0x4)]    (@32[(init_esp+0x4)]<<(@8[init_esp]&0x1F))
---------- 21 0x6fb97e ----------
eax = @32[init_esp]
@32[init_esp]    @32[@32[init_esp]]
---------- 24 0x6fc3d1 ----------
eax = ((((@16[init_esi]_to[0:16], init_eax[16:32]_to[16:32])+init_ebx)^0x18EE5784)-0x12C0A81E)
ebx = (init_ebx^((((@16[init_esi]_to[0:16], init_eax[16:32]_to[16:32])+init_ebx)^0x18EE5784)-0x12C0A81E))
edx = (init_edx^((((@16[init_esi]_to[0:16], init_eax[16:32]_to[16:32])+init_ebx)^0x18EE5784)-0x12C0A81E))
esi = (init_esi+0x2)
---------- 28 0x6fc935 ----------
...
esp = (init_esp-0x4)
@32[(init_esp+0x4)]    (@32[(init_esp+0x4)] umul32_hi @32[init_esp])
@32[init_esp]    (@32[(init_esp+0x4)] umul32_lo @32[init_esp])
@32[(init_esp-0x4)]    (0x2_to[0:2], (parity init_edi)_to[2:3], 0x0_to[3:4], (((init_esp+0x4)&0x10)==0x10
```

Instruct the interpreter

- `@32[(init_edi+0x1C)]` is the vm eflagest le eflag de la vm
- we replace `@8[(init_edi+0x28)]`, by REG1, REG2, ...
- we replace `esp+X` by registers `arg32_0`, ...

logo/eads

Instruction

```
known_vm_e = {
    init_edi + ExprInt(uint32(0x1C)): regflag,
    init_edi + ExprInt(uint32(0x20)): reg1,
    init_edi + ExprInt(uint32(0x24)): reg2,
    init_edi + ExprInt(uint32(0x28)): reg3,
    init_edi + ExprInt(uint32(0x2C)): reg4,
    init_edi + ExprInt(uint32(0x30)): reg5,
    init_edi + ExprInt(uint32(0x34)): reg6,
    init_edi + ExprInt(uint32(0x38)): reg7,
    init_edi + ExprInt(uint32(0x3C)): reg8,
    init_edi + ExprInt(uint32(0x40)): reg9,

    ExprMem(init_esp-ExprInt(uint32(4))): argm1_32,
    ExprMem(init_esp-ExprInt(uint32(2))): argm1_16,
    ExprMem(init_esp): arg0_32,
    ExprMem(init_esp+ExprInt(uint32(4))): arg1_32,
    ExprMem(init_esp, size = 16): arg0_16,
    ExprMem(init_esp+ExprInt(uint32(2)), size=16): arg1_16,
    ExprMem(init_esp, size = 8): arg0_08,
    ExprMem(init_esp+ExprInt(uint32(4)), size=8): arg1_08,
}
```

logo/eads

Results

```
---------- 143 0x70a4f5 ----------
esp = (init_esp - 0x4)
arg-1_32 = @32[init_edx]
=> push 32@[edx]
---------- 151 0x70b1e1 ----------
eax = arg1_32
ecx = (arg1_32_to[0:8], ((0x1 == (arg1_08 >> 0x7)) == 0x1)?(0x0,0xFFFFFFFF)_to[8:32])
esp = (init_esp + 0x4)
arg1_32 = (arg1_32_to[0:8], ((0x1 == (arg1_08 >> 0x7)) == 0x1)?(0x0,0xFFFFFFFF)_to[8:32])
=> pop   dum
   pop   X
   movsx X, X8
   push  X
---------- 154 0x70b8b4 ----------
ecx = (arg0_16_to[0:16], init_ecx[16:32]_to[16:32])
esp = (init_esp - 0x2)
arg1_16 = (arg1_16 a>> (arg0_08 & 0x1F))
arg-1_16 = ... flags of op ...
=> push arg1_16 >> arg0_08
   push eflags
---------- 158 0x70bb56 ----------
esp = (init_esp - 0x4)
arg-1_32 = @32[reg4]
=> push @32[reg4]
---------- 155 0x70ba09 ----------
arg0_32 = (! arg0_32)
DST 0x6F88F1
=> not @32[esp]
```

Multi bloc mnemonic

- A mnemonic can have a complex graph
- we can evaluate a bloc, and propagate its state to its sons
- and so on
- No loop for the moment

logo/eads

Result: multiple vm exit

teste si reg7 == 0x0

```
---------- state 1 ----------
eax = @32[(init_esp + 0x1C)]
ebx = @32[(init_esp + 0x10)]
ecx = @32[(init_esp + 0x18)]
edx = @32[(init_esp + 0x14)]
esi = arg1_32
edi = arg0_32
the vm does not unstack args
esp = (init_esp + 0x28)
ebp = @32[(init_esp + 0x8)]
@32[reg5] = 0x0
DST @32[(init_esp + 0x24)]
```

```
---------- state 2 ----------
eax = @32[(init_esp + 0x1C)]
ebx = @32[(init_esp + 0x10)]
ecx = @32[(init_esp + 0x18)]
edx = @32[(init_esp + 0x14)]
esi = arg1_32
edi = arg0_32
the vm unstack reg7 arguments
esp = ((init_esp + @32[reg7]) + 0x28)
ebp = @32[(init_esp + 0x8)]
@32[reg7] = 0x0
@32[reg5] = 0x0
arg-1_32 = ((init_esp + @32[reg7]) + 0x24)
DST @32[(init_esp + 0x24)]
```

logo/eads