Helping RE with LLVM

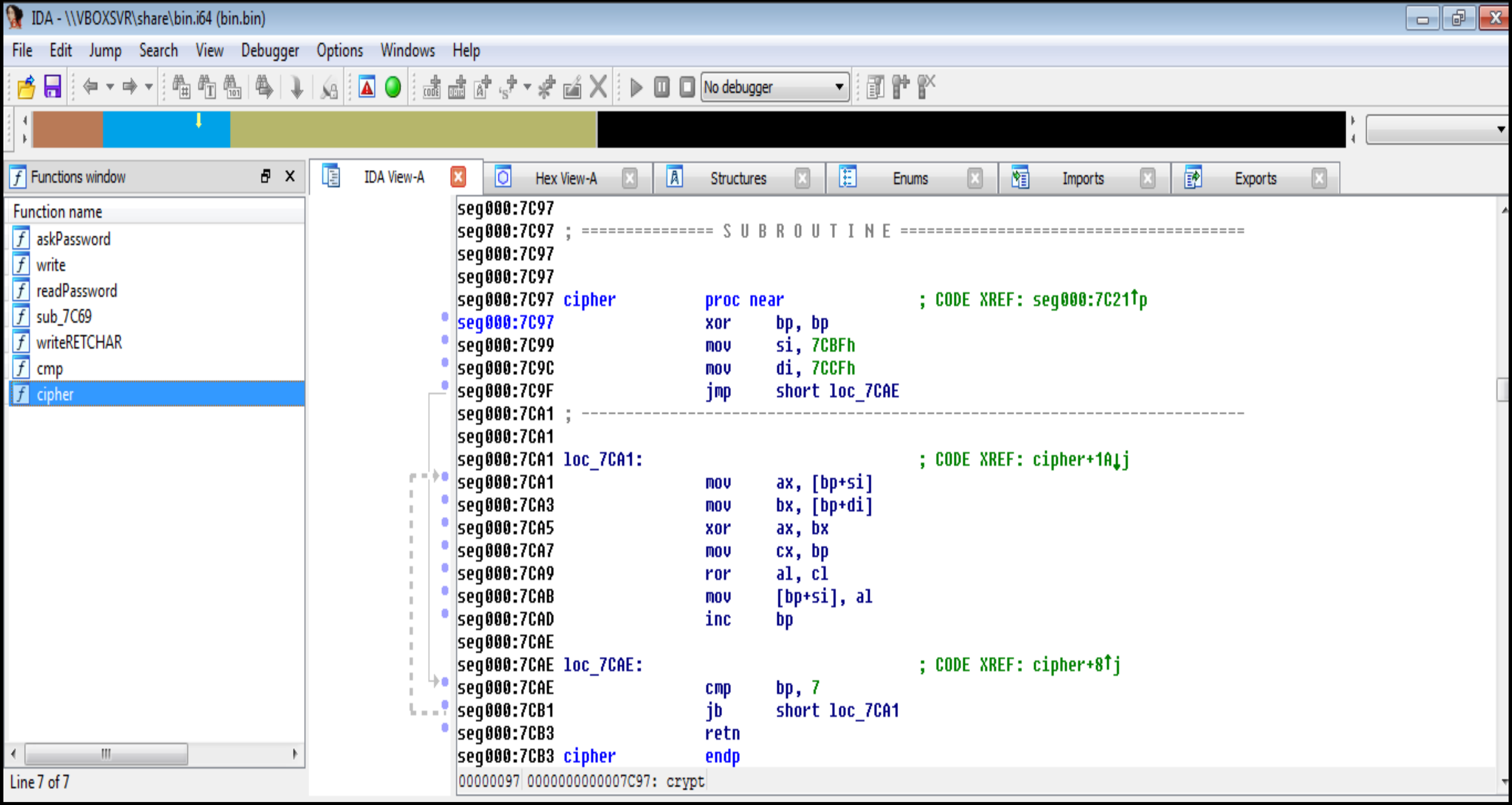
lionel@lse.epita.fr

LSE

Security
System

Laboratory of Epita

1) Reverse Engineering



Function name
- askPassword
- write
- readPassword
- sub_7C69
- writeRETCHAR
- cmp
- cipher

Line 7 of 7

```
seg000:7C97
seg000:7C97 ; =============== S U B R O U T I N E =======================================
seg000:7C97
seg000:7C97
seg000:7C97 cipher          proc near               ; CODE XREF: seg000:7C21↑p
seg000:7C97                 xor     bp, bp
seg000:7C99                 mov     si, 7CBFh
seg000:7C9C                 mov     di, 7CCFh
seg000:7C9F                 jmp     short loc_7CAE
seg000:7CA1 ; ---------------------------------------------------------------------------
seg000:7CA1
seg000:7CA1 loc_7CA1:                               ; CODE XREF: cipher+1A↓j
seg000:7CA1                 mov     ax, [bp+si]
seg000:7CA3                 mov     bx, [bp+di]
seg000:7CA5                 xor     ax, bx
seg000:7CA7                 mov     cx, bp
seg000:7CA9                 ror     al, cl
seg000:7CAB                 mov     [bp+si], al
seg000:7CAD                 inc     bp
seg000:7CAE
seg000:7CAE loc_7CAE:                               ; CODE XREF: cipher+8↑j
seg000:7CAE                 cmp     bp, 7
seg000:7CB1                 jb      short loc_7CA1
seg000:7CB3                 retn
seg000:7CB3 cipher          endp
00000097 0000000000007C97: crypt
```

2) Obfuscation objectives

- confuse tools
    * hack binary loading
    * unaligned instructions

- confuse human
    * junk code
    * proxy calls / vm
    * cipher

3) unaligned instructions

```
// test.s
_start:
    pushl   %ebp
    movl    %esp  %ebp
    subl    $16  %esp
    movl    $32   4 %ebp
    jmp _1
    .byte 0xC7
    .byte 0x45
_1:
    call    0xCAFEBABE
    hlt
```

```
// objdump -dr

push    %ebp
mov     %esp %ebp
sub     $0x10, %esp
movl    $0x20, 0x4(%ebp)
jmp     0x11
movl    $0xcafebaba, 0x18(%ebp)
hlt
```

4) unaligned instructions fails

Against linear disasm algorithm
→ recursive disasm algorithm

Disasm re synchronizes itself after few instructions.

5) Junk code

Pollutes the code with:

Dead Code...

Expand constant values...

Use stack like a VM...

6) proxy calls / vm

Sometimes we found calls like that

```
// some computation on %eax
call    *%eax
```

So function addresses are hard to find

Sometimes full virtual machines are used.
VM uses lots of junk code / proxy calls

7) Cipher block

Parts of code are ciphered (or not here).

Decipher stub use previous tech (6,5,4) to decipher it or grab it from somewhere (network, device).

8) IDA but ...

IDA isn't free (license, expensive non free plugins)

our IDA plugin for deobfuscation?

note:
"junk code looks like unoptimized code!"

9) Our tool

junk code looks like unoptimized code!

Dead Code... DCE, CSE

Expand constant values... Constant folding

Use stack like VM... CFG, SSA, recombination

10) LLVM

LLVM framework provides what we need.

LLVM works with its own IR language for optimization stuff.

We need to convert ASM to LLVM IR !

This mapping is critical! We must fill the semantic gap!

11) POC

Requirements:

- Quick & Dirty -> Python
- Read Elf -> construct 2.5
- Disasm -> distorm 3.3
- Compiler stuff -> LLVM 3 + pyllvm

12) Deobfuscation Chain

- Read Elf
- Disasm
- Remap instructions to LLVM IR
- Do optimization passes
- Obtain simplified asm dump from IR

13) Read Elf

```
from construct.formats.executable.elf32 import *
def LoadElf32Text(fn):
    obj = elf32_file.parse_stream(open(fn, "rb"))
    bincode = None
    for section in obj.sections:
        if section.name == b'.text':
            return section.data.read()
```

14) Disasm

```
from distorm3 import *
# ...
while True:
    one_inst = distorm3.DecodeOne(map_adr, self.bincode, Decode32Bits, idx)
    size_inst = one_inst[1]
    map_adr += size_inst
    idx += size_inst
    # ...
    if one_inst[2] == "HLT":
        break
```

15) Remap instruction

```
from llvm.core import *

class  Reorganize:

    def __init__(self, bincode):
        self.bincode = bincode
        # need a module
        self.module = Module.new("reorg")
        func_type = Type.function(Type.void(), [])
        self.main = Function.new(self.module, func_type, "main")
        self.entry = self.main.append_basic_block("entry")
        # need a builder
        self.builder = Builder.new(self.entry)
# ...
        self.builder.ret_void()
```

16) Do optimized passes

```
from llvm.ee import *
from llvm.passes import *
# ...
    def doOrganize(self):
        pass_man = FunctionPassManager.new(self.module)
        pass_man.add(PASS_MEM2REG)
        # Eliminate Common SubExpressions.
        pass_man.add(PASS_GVN)
        # Simplify the control flow graph (deleting unreachable blocks, etc).
        pass_man.add(PASS_DCE)
        pass_man.add(PASS_CONSTPROP)
        pass_man.add(PASS_INSTCOMBINE)
        # finish init pass_man
        pass_man.initialize()
        # optimize block
        pass_man.run(self.main)
```

17) Get the final ASM

```
def getFinalAsm(self):
    # For intel syntax
    import sys, os
    os.environ['LLVMPY_OPTIONS'] = "-x86-asm-syntax=intel"
    parse_environment_options(sys.argv[0], "LLVMPY_OPTIONS")
    # For 32 bit
    tm = TargetMachine.lookup(arch="x86", cpu="i386")
    return tm.emit_assembly(self.module)
```

18) Mapping

```
movl    %eax, $4
call    0xCAFEBABE
```

how to map the stack? push? pop?

LLVM use "alloca" and naming for locals!

how to map EAX ?

LLVM "store" only on local variables previously created by LLVM "alloca"!

how to map call?

LLVM "call" use type informations!

19) Map stack/push/pop

Creates an hidden variable sp as first local
Get its address
Use it as a stack register   .ptrtoint(), .inttoptr()

PUSH -> dec __sp + store
POP -> load + inc __sp

19) Map registers

Create a local and shadow store

```
eax = builder.alloca(Type.int(), "eax")
_eax = builder.load(eax, "_eax")
builder.store(Constant.int(ty_int, 4), _eax)
```

Register are only tmp var, thanks to PASS_MEM2REG, allocations disappears

20) Map calls

We use a local variable to store the address.
LLVM detect the constant propagation.

```
funcadr_type = Type.pointer(Type.function(Type.void(), (), var_arg=True))
funcadr = builder.alloca(ty_int, "funcadr")
builder.store(Constant.int(ty_int, 1234), funcadr)
vfuncadr = builder.load(funcadr, "vfuncadr")
ptrfunc = builder.inttoptr(vfuncadr, funcadr_type, "ptrfunc")
builder.call(ptrfunc, [])
```

All these lines for generate

```
call    1234
```

21) A full example

```
pushl $12
pushl $555
movl (%esp), %eax
addl -4(%esp), %eax
addl 8, %esp
pushl %eax
movl $0x8045600, %eax
call *%eax
```

```
push 567
call    1234
```

```
%sp = alloca i32
%sp2 = alloca i32
%sp3 = alloca i32
%isp = ptrtoint i32* %sp to i32
%isp1 = sub i32 %isp, 4
%sp4 = inttoptr i32 %isp1 to i32*
store i32 12, i32* %sp4
%isp5 = ptrtoint i32* %sp4 to i32
%isp6 = sub i32 %isp5, 4
%sp7 = inttoptr i32 %isp6 to i32*
store i32 555, i32* %sp7
%eax = alloca i32
%isp8 = ptrtoint i32* %sp7 to i32
%isp9 = add i32 %isp8, 8
%tmp = inttoptr i32 %isp9 to i32*
%tmp10 = load i32* %sp7
store i32 %tmp10, i32* %eax
%isp11 = ptrtoint i32* %sp7 to i32
%isp12 = add i32 %isp11, 4
%tmp13 = inttoptr i32 %isp12 to i32*
    = load i32* %tmp13
    = load i32* %eax
%eax14 = add i32      ,
%isp15 = ptrtoint i32* %sp7 to i32
%isp16 = sub i32 %isp15, 4
%sp17 = inttoptr i32 %isp16 to i32*
store i32 %eax14, i32* %sp17
store i32 134501888, i32* %eax
%_eax = load i32* %eax
%ptrfunc = inttoptr i32 %_eax to void (...)*
call void (...)* %ptrfunc()
```

22) Next step

Seems to work with simple cases

* More testing needed
* Find functions parameters
* Inlining
* ISA specific instructions (ldt, SSE x)
* ...

23) Thanks

lionel@lse.epita.fr

soon http://code.google.com/p/py-orgasm