Exploiting ROP attacks with a unique instruction

Julien Couvy, ***Vrije Universiteit***

Introduction

- Background information
- Our project with examples
- Limitations
- Results & Conclusion

Return-into-libc and DEP

Data Execution Prevention (aka. W^X)

- Industry response against **code injection** exploits
- Marks all writable locations in a process's address space as **non executable**
- Hardware support in Intel and AMD processors
- Protection available in all modern OS

Return-into-libc

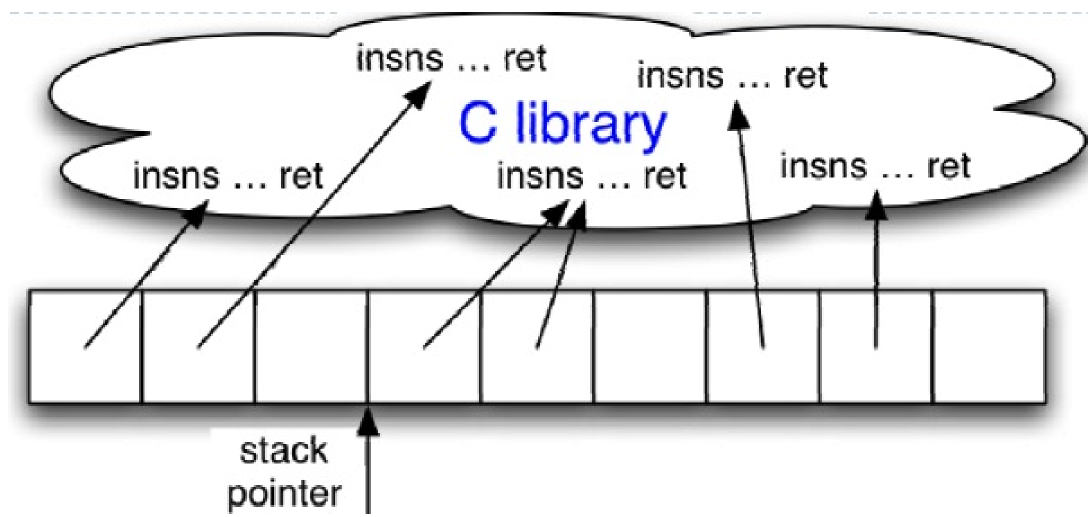
- Evolution of code injection exploits
- No injection necessary, **instead** re-use functions present in shared libraries (libc common target)
- Sensible instructions like `system()` or `printf()`
- Removed from *glibc*, replaced by safe versions like `execve()`

Return oriented programming

Return oriented programming: Overview

- *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, H.Shacham 2007
- Turing-complete exploit language
- Defeats DEP, code signing, and ASLR (non trivial)
- No function call required

Return oriented programming: Machine level



➡️ The *stack pointer* (%esp) determines which instruction sequence to fetch & execute

➡️ Processor doesn't automatically increment %esp; - but the "ret" at the end of each instruction sequence does

Return oriented programming: Gadgets

- Small instruction sequences ending in `ret`
- Already present in the target binary
- Chain of gadgets = attacker payload

Problem

Cratfting payload is complex and time-consumming...

Can we automate it ?

Our idea: *Mov2Rop*

Mov2Rop: Objectives

➡️ Prove that *return oriented programming* can be made more accessible ==> more dangerous

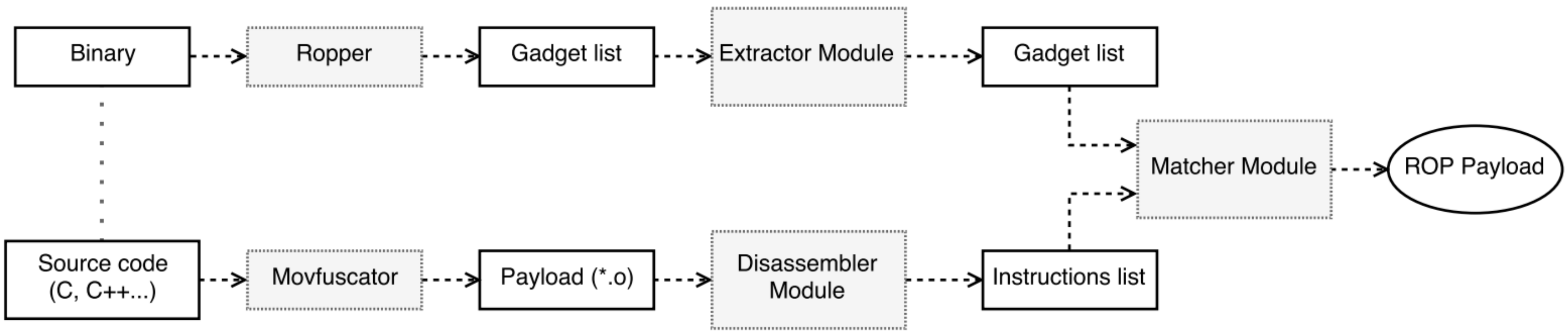
➡️ Automatic gadget extraction and chaining...

➡️ Simplified by targetting `mov` instructions only

Mov2Rop: Tools used

- Language: **Python 3**
- Gadget extraction: **Ropper**
- Disassembly: **Capstone**
- Payload translation: **Movfuscator**

Mov2Rop: Outline



Mov2Rop: Disassembler Module

- ➡️ Uses Capstone framework
- ➡️ Seaches for `mov` instructions in an object file
- ➡️ `mov` instructions are stored in custom **Instruction** data structures

Mov2Rop : Instruction example

```
Instruction found at <0x80bbf0e>
-----------------------------------
Mnemonic: MOV r/m32,r32
   Label: mov dword ptr [ecx], eax
    Dest: ecx
     Src: eax
```

Mov2Rop: Extractor Module

- Ropper's engine in a Python script
- Gadgets are identified with *regular expressions*...

```
ropper -f fibonacci --type rop --search "mov e??, e??"
```

- and stored in custom **Gadget** data structures

Mov2Rop: Gadget example

```
✗ write on ESP
Gadget <0x80bbe4f>:      g1: mov eax, dword ptr [eax]
                         g2: add esp, 8
                         g3: pop ebx
                         g4: ret;

✓ potential gadget
Gadget <0x809c35f>:      g1: mov eax, dword ptr [ebx]
                         g2: pop ebx
                         g3: pop esi
                         g4: ret;
```

Mov2Rop: Matcher Module

➡️ Instruction analysis

- Instructions need to pass a set of rules to be validated

➡️ Gadget chaining

- Tries to map a gadget chain with a payload instruction
- Searches for eventual side-effects

➡️ Stack preparation and visualization

- Stores gadget addresses on the stack
- Searches for `pop` instructions

  I. Safekeeping return addresses integrity

  II. Storing immediate values on the stack

Mov2Rop: Chain example

```
Target: mov dword ptr [eax], edx
Gadget <0x808e8ea>:       g1: mov dword ptr [eax], edx
                          g2: xor eax, eax
                          g3: pop ebx
                          g4: pop esi
                          g5: pop edi
                          g6: ret;

             STACK
+================================+
|          <0x0808e8ea>          | address of G1
+================================+
|          <0x42424242>          | value to be popped
+================================+
|          <0x42424242>          | value to be popped
+================================+
|          <0x42424242>          | value to be popped
+================================+
```

Limitations

- Support only x86 32 bits and instructions using 32 bits registers
- Incomplete side-effect management
- External tools are flawed and limited

Results

Total Gadgets	**13124**	**100%**
Mov Gadgets	953	7%
Total Instructions	**1270**	**100%**
Supported	1178	92%
Supported (w/o offsets)	1098	85%
Not supported	92	7%

Table 2: Statistics on fibonacci.c

Total Gadgets	**13153**	**100%**
Mov Gadgets	961	7%
Total Instructions	**2054**	**100%**
Supported	1925	93%
Supported (w/o offsets)	1803	87%
Not supported	129	7%

Table 3: Statistics on hanoi_towers.c

Further work

Two possibilites:

➡️ Improving current protoype

- Cover all `mov` instructions
- In depth side-effect verification
- x86_64 support
- etc...

➡️ Integration as backend in LLVM

- *Pro:* LLVM = growing project with strong community, LLVM IR allows to use any input language easily
- *Con:* Dropping the idea of `mov` only instructions (original motivation)

Thank you for your attention.

Any questions ?