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

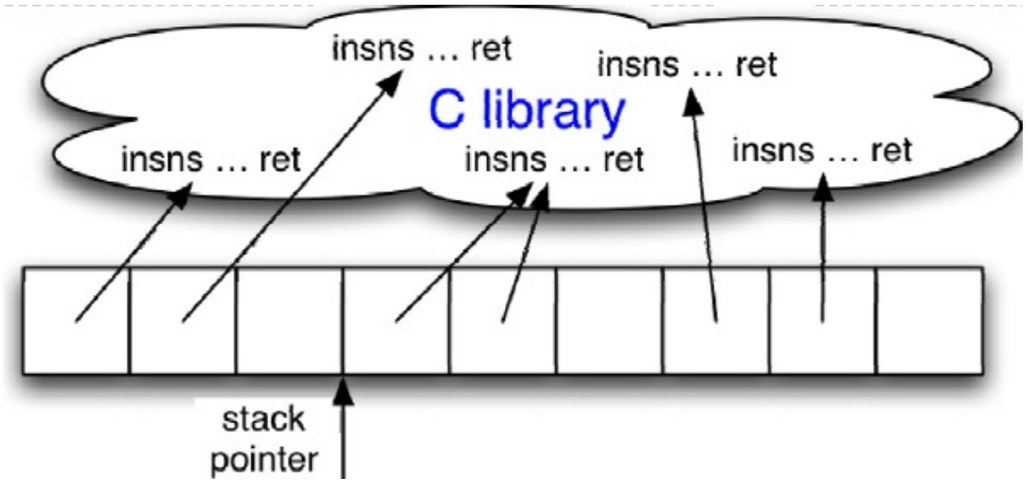
➡️ *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, H.Shacham 2007

➡️ 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

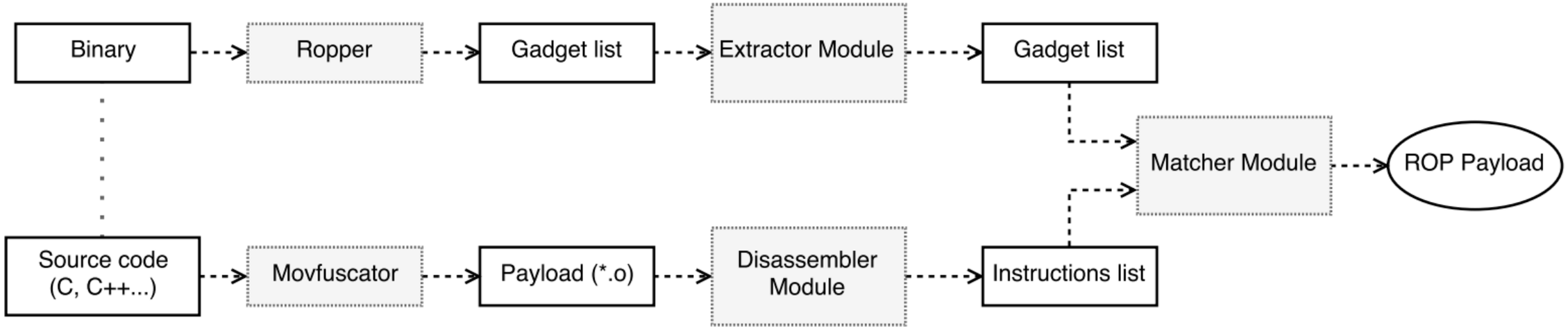
➡️ Automatic gadget extraction and chaining...

➡️ 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

Result 1

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

Result 2

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
- ...

➡️ 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)

Object dump

```
▷ python Matcher.py test_programs/fibonacci test_programs/fibonacci.o | grep 'BYTE'
     30:       8a 15 00 00 00 00       mov    dl,BYTE PTR ds:0x0
     36:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
     4b:       8a 15 01 00 00 00       mov    dl,BYTE PTR ds:0x1
     51:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
     66:       8a 15 02 00 00 00       mov    dl,BYTE PTR ds:0x2
     6c:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
     81:       8a 15 03 00 00 00       mov    dl,BYTE PTR ds:0x3
     87:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    8eb:       8a 15 00 00 00 00       mov    dl,BYTE PTR ds:0x0
    8f1:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    906:       8a 15 01 00 00 00       mov    dl,BYTE PTR ds:0x1
    90c:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    921:       8a 15 02 00 00 00       mov    dl,BYTE PTR ds:0x2
    927:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    93c:       8a 15 03 00 00 00       mov    dl,BYTE PTR ds:0x3
    942:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    e9e:       8a 15 00 00 00 00       mov    dl,BYTE PTR ds:0x0
    ea4:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    eb9:       8a 15 01 00 00 00       mov    dl,BYTE PTR ds:0x1
    ebf:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    ed4:       8a 15 02 00 00 00       mov    dl,BYTE PTR ds:0x2
    eda:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
    eef:       8a 15 03 00 00 00       mov    dl,BYTE PTR ds:0x3
    ef5:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   1197:       8a 15 00 00 00 00       mov    dl,BYTE PTR ds:0x0
   119d:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   11b2:       8a 15 01 00 00 00       mov    dl,BYTE PTR ds:0x1
   11b8:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   11cd:       8a 15 02 00 00 00       mov    dl,BYTE PTR ds:0x2
   11d3:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   11e8:       8a 15 03 00 00 00       mov    dl,BYTE PTR ds:0x3
   11ee:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   148e:       8a 80 00 00 00 00       mov    al,BYTE PTR [eax+0x0]
   14b4:       8a 15 00 00 00 00       mov    dl,BYTE PTR ds:0x0
   14ba:       8a 84 02 00 00 00 00    mov    al,BYTE PTR [edx+eax*1+0x0]
   14c1:       8a 15 01 00 00 00       mov    dl,BYTE PTR ds:0x1
   14c7:       8a 84 02 00 00 00 00    mov    al,BYTE PTR [edx+eax*1+0x0]
   14ce:       8a 15 02 00 00 00       mov    dl,BYTE PTR ds:0x2
   14d4:       8a 84 02 00 00 00 00    mov    al,BYTE PTR [edx+eax*1+0x0]
   14db:       8a 15 03 00 00 00       mov    dl,BYTE PTR ds:0x3
   14e1:       8a 84 02 00 00 00 00    mov    al,BYTE PTR [edx+eax*1+0x0]
   14e8:       8a 80 00 00 00 00       mov    al,BYTE PTR [eax+0x0]
   1527:       88 15 00 00 00 00       mov    BYTE PTR ds:0x0,dl
   1651:       8a 15 00 00 00 00       mov    dl,BYTE PTR ds:0x0
   1657:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   166c:       8a 15 01 00 00 00       mov    dl,BYTE PTR ds:0x1
   1672:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   1687:       8a 15 02 00 00 00       mov    dl,BYTE PTR ds:0x2
   168d:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
   16a2:       8a 15 03 00 00 00       mov    dl,BYTE PTR ds:0x3
   16a8:       8a 14 11                mov    dl,BYTE PTR [ecx+edx*1]
```

Gadgets found

```
▷ ropper -f test_programs/fibonacci --type rop --search 'mov dl'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: mov dl

[INFO] File: test_programs/fibonacci
0x08096754: mov dl, 0x83; les ecx, ptr [ebx + ebx*2]; pop esi; pop edi; pop ebp; ret;
0x080d683c: mov dl, 0x9f; sub edi, esi; int 0x6b; ret;
0x080ddaa9: mov dl, 0xa; ret;
0x080d3a52: mov dl, 0xb3; mov ah, 0xb5; mov dh, 0xb7; mov eax, 0xbcbbbab9; mov ebp, 0xc1c0bfbe; ret 0xc4c3
;
0x0804cf4d: mov dl, 0xff; inc dword ptr [ebx + 0x508d10c4]; add ecx, ebp; ret;
0x080daa93: mov dl, 1; or cl, byte ptr [esi]; adc al, 0x41; ret;
```

Finding gadgets

```
[INFO] File: test_programs/fibonacci
0x080dcb3c: pop ecx; or cl, byte ptr [esi]; adc al, 0x41; ret;
0x080e6f75: pop ecx; or cl, byte ptr [esi]; adc al, 0x43; ret;
0x080e1bce: pop ecx; or cl, byte ptr [esi]; adc al, 0x46; ret;
0x080b6754: pop ecx; pop ebx; pop ebp; lea esp, dword ptr [ecx - 4]; ret;
0x0806ff61: pop ecx; pop ebx; ret;
0x080acd53: pop ecx; pop es; add byte ptr [eax], al; add esp, 0x2c; ret;
0x080dd2b2: pop ecx; push cs; or byte ptr [ebx + 0x100e4502], al; imul ecx, dword ptr [esi], 8; inc ecx; r
et;
0x0805a563: pop ecx; ret 0xffff;

[INFO] File: test_programs/fibonacci
0x08069600: add eax, ecx; ret;
0x08069e23: add eax, edx; ret;
0x0804cf55: add ecx, ebp; ret;
0x08048983: add ecx, ecx; ret;
0x0809f20c: add esi, ebx; ret;
0x0809bbb2: add esi, esi; ret;

[INFO] File: test_programs/fibonacci
0x0807bb6b: mov eax, dword ptr [eax]; add al, byte ptr [eax]; add bh,
al; ret;
```

Payload instructions

```
    3fd:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    403:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    43c:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    442:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    448:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    477:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    4a6:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    4ac:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    4b2:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    4b8:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    4be:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
    4e3:    8b 80 00 00 00 00       mov    eax,DWORD PTR [eax+0x0]
```

```
            STACK
+=============================+
|        <0x0806ff61>         | address of G1 <pop ecx; pop ebx; ret;>
+=============================+
|        <0x00000000>         | value required in %ecx
+=============================+
|        <0x42424242>         | dummy value for <pop ebx;>
+=============================+
|        <0x08069600>         | address of G2 <add eax, ecx; ret;>
+=============================+
|        <0x0807bb6b>         | address of G3 <mov eax, dword ptr [eax]...>
+=============================+
               ...
```

Thank you for your attention.

Any questions ?