

Exploiting ROP attacks with a unique instruction

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 24, 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 expressiveness.

We describe the fundamentals behind ROP attacks, and explain the functioning 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-executable¹, such that an attempt to execute machine code in these regions will cause an exception has largely mitigated classic code injections attacks. It is deployed in virtually all main operating systems: Linux (first via PaX patches), OpenBSD, Windows (since XP SP2), OS X (since 10.5), Android and so on. Hardware now features easy DEP support with an extra flag dedicated to the marking: Intel "XD" and AMD "NX" to name a few.

DEP techniques 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. As no code injection was required, doing so would bypass the defenses we highlighted. A classic target was *libc*² that contained subroutines for powerful system calls that would allow arbitrary code execution. Back then, a common perception was

that removing dangerous *syscalls*³ from standard libraries would suffice to stop return-to-libc attacks. Besides, with the arrival of Intel's new 64bits x86 processors, the subroutines calling convention was changed to require most arguments of a function to be passed in registers instead of on the stack. Shared libraries developers 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 (initially proposed by Shacham [2] at CCS in 2007) capable of arbitrary code execution without relying on shared libraries. *Return Oriented Programming (ROP)* [3] has now become the main technique of choice for nearly all modern exploits of memory-safety vulnerabilities.

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* [4] based on the observation that `mov` instructions are Turing complete [5]. 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*.

The text segment, also known as code segment, 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

¹also referred as $W\oplus X$

²return-to-lib attacks are often referred as return-to-libc for that reason

³`system()` allows to execute any program with the current privileges

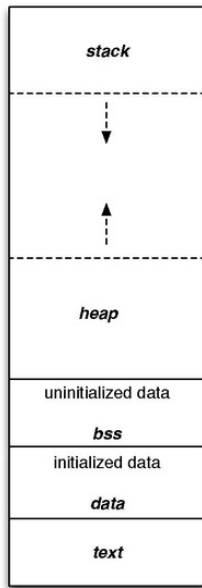


Figure 1: Memory organization

will result in a segmentation violation.

The **data segment** is split in two parts (see Listing 1)

- Initialized 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

The **heap segment** 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.

The **stack segment**, a stack is a contiguous block of memory containing data such as the local variables of functions. 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. In 32bits architectures, 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 possibilities 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 and other memory corruption vulnerabilities still plague many programs and are a main research topic in computer security [6–8].

```
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 existing within the program's space or injected by the attacker. Buffer overflow vulnerabilities typically occur in code that:

- Relies on external data to control its behavior
- Depends upon properties of the data that are enforced outside of the immediate scope of the code
- Is so complex that a programmer cannot accurately predict its behavior

With cautious manipulations, an attacker could grant himself unrestricted privilege to an unprivileged account or user.

Example: *stack-based buffer overflow* [9]. 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 program (see Listing 2), the vulnerability comes from a wrong use of *strcpy()*. We disabled ASLR and Stack Canaries for the sake of simplicity. 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 which would lead to a segmentation fault while the latter is the address we want to return to. Any disassembler tool like *gdb* (see Listing 3) can give us the information we need to perform the overflow and spawn a shell (see Listing 5).

```
$ 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 SFP⁴ with a dummy value and then the target address. Listing 4 shows the state of the stack before and after the 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. In the remainder of this paper, we will assume that we deal with a 32bits architecture.

2.3 Return Oriented Programming

In this section we will introduce the concepts behind *return-oriented programming* [3].

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

<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

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

Spawning a shell !

sh-4.4\$

Listing 5: Spawning a shell

code. It was built to overcome buffer exploit defense mechanisms like ASLR (in a non-trivial way), or DEP. 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 \oplus X defense will not prevent it from running.

Return oriented programming 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 is quite complex and very little research exists to automate the creation of Turing complete exploits.

Example: see Listing 6. We want to give a general understanding on how gadget chaining generally works. Here, we need to concatenate *"/bin/sh"* in a buffer by chaining functions together and finally spawn a shell. Each function requires specific parameters that we need to store on the stack along with the gadget chain. We assume that we have 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 address⁵ is the address

⁴the saved frame pointer (old content of %ebp) is called SFP

⁵see standard C function calling procedure

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.

```
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 6: ROP exploit example

3 RELATED WORK

Control-flow integrity (CFI) [10] is a property of a program which says that the flow of execution does not divert from the intended or original control flow graph. Many techniques were/are being developed 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 the target address. 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.

<address of exec_command()>	
0xcafebabe <key1>	
0x8badf00d <key2>	
<address of 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 7: Stack prepared with a ROP chain

ShadowStacks: This technique mitigates return address overwrites by keeping a record of the legitimate return address for a function call on a separate stack, 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 but do not fix the underlying vulnerability.

ROP Compiler: Follner et al. presented PSHAPE [11], 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 differs from other compilers as we chose to compile 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 [12] within the Extractor module to extract gadgets from binaries. Gadgets are searched according to a regular expression (see Listing 8).

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

Listing 8: Ropper search engine

The output is then parsed by our program to create a list of *Gadget* objects. Gadgets are defined by a list of *Instruction* objects (see Listing 9) 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 dictionary of opcodes (see Table 1). The opcode is extracted with regular expressions manipulations on Ropper's output.

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	MOV r/m32,imm32	Memory write from immediate val
0xBB		
...		
0xA1	MOV r32,imm32	Memory read from immediate val ex: mov eax, ds:0x0
0x8B1D		
...		
0xA3	MOV imm32,r32	Memory write from 32bits reg to imm val
...

Table 1: Opcode Dictionary

```

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

```

Listing 9: Example instruction object

4.2 Payload Treatment

The source code that we want to translate is firstly compiled to an object file in Intel syntax using *movfuscator*. We use Capstone [13], a multi architecture disassembly framework to analyze each instruction. The Disassembler module searches for all mov instructions and creates Instructions objects similar to Extractor. When created, an Instruction is assigned a mnemonic reference if the dictionary lookup was successful, otherwise it's simply referred to as not supported.

4.3 Instruction Analysis

The Matcher module will create lists containing each type of gadget. We divide them in three categories: memory reads, memory writes, and register moves. The output of the Extractor module is analyzed using an expandable set of rules.

No access on esp: we rule out any instruction accessing, in read or write, esp. Because of the nature of ROP, controlling esp is difficult and it is safer not to use it other than with ret instructions.

Restricted instructions: certain instructions like call or leave introduce jumps at undetermined addresses.

Conflict prevention: gadgets that are not "pure" (i.e: more than one instruction long, excluding ret) have a chance to contain intermediary instructions that can disrupt the purpose of the gadget. For example (see Listing 10), if a value needs to be stored on a register eax and one intermediary instruction modifies the content of said register, then the value is lost. Thus, we rule out any gadget with intermediary instructions that affects the dest register of the first instruction.

We observed that chains can be found even with these

```

X write on ESP
Gadget <0x80a1de3>: g1: mov dword ptr [eax], edx
                  g2: add esp, 8
                  g3: pop ebx
                  g4: ret;

X conflicting instructions
Gadget <0x807af78>: g1: mov eax, dword ptr [eax]
                  g2: sub eax, dword ptr [edx]
                  g3: ret;

```

Listing 10: Rule verification

limitations, while fixing them would require a delicate stack preparation.

4.4 Gadget Chaining

The Matcher module is also in charge of chaining gadgets together mapping to the desired payload instruction.

Firstly, we mapped all possible movements between registers to obtain a general view. On a very simple program (with glibc statically linked) we got the following map (see Listing 11):

```

Possible reg mov:
src: dst (<> = direct, [] = chained)
eax: <edi edx> [eax esi ebx ecx] ebp esp
ebx: <eax> [edi edx esi ebx ecx] ebp esp
ebx: <eax> [edi edx esi ebx ecx] ebp esp
ecx: <eax> [edi edx esi ebx ecx] ebp esp
edx: <eax esi ebx ecx> [edi edx] ebp esp
ebp: <eax> [edi edx esi ebx ecx] ebp esp
edi: <eax edx> [esi ebx edi ecx] ebp esp
esi: <eax esi> [edi edx ebx ecx] ebp esp

```

Listing 11: Map of movement between registers

For example on the first line, the map highlights that at least one gadget that allows to move the content of edi to eax directly⁶. The map is useful when we need to weave gadgets together in order to fit a target instruction.

We ensure that no side-effects occur within the gadget chain (see Algorithm 2). In other words, we check if any

⁶with a "pure" gadget constituted of one instruction followed by a ret

Algorithm 1 IsMovable(src, dst, register_move[])**Require:** *register_move*[] (gadgets moving regA to regB)

```
1: for register accessible from src do
2:   if register ≠ src and register = dst then
3:     for gadget in register_move[] do
4:       if gadget_src = src and gadget_dst = register
          then
5:         if not SearchForConflict() then
6:           return true
7: for register accessible from src do
8:   if register not visited and register ≠ src then
9:     register ← visited
10:  for gadget in register_move[] do
11:    if gadget_src = src and gadget_dst = register
        then
12:      if not SearchForConflict() then
13:        return IsMovable(register, dst, register_move[])
14: return false
```

intermediary instruction inside the gadget chain affects negatively a value modified beforehand. If so, the gadget is discarded.

Algorithm 2 SearchForConflict(gadget, target)

```
1: unavailable ← [target_src, target_dst]
2: if gadget_src = target_dst then
3:   unavailable ← unavailable + target_dst
4: for instruction in gadget ← instructions do
5:   if instruction ← dst in unavailable then
6:     return true
7: return false
```

4.5 Stack Preparation

We provide a visual representation (see Listing 12) of the stack for each payload instruction. We consider that the stack grows downwards, and that every value will be stored on the stack on reverse order.

Each chain is analyzed looking for pop instructions. If present, we need to prepare the stack in a certain way. We distinguish two cases:

Safeguard ret chain: to keep the integrity of the return addresses chain, we store a placeholder value on the stack. That value that will be popped and the function will return to the next gadget address.

Immediate values: certain instructions will read/write to immediate values. These values also need to be present on the stack as we inserted pop gadgets to store them in specific registers.

Gadget addresses: the default case is storing the address of the beginning of each gadget chain.

Target: mov dword ptr [eax], ds:0x0

Gadget <0x806ff3a>: g1: pop edx
 g2: ret;

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

```

                        STACK
+=====+
|          <0x0806ff3a>          | address of G1
+=====+
|          <0x00000000>          | value to be popped
+=====+
|          <0x0808e8ea>          | address of G2
+=====+
|          <0x42424242>          | value to be popped
+=====+
|          <0x42424242>          | value to be popped
+=====+
|          <0x42424242>          | value to be popped
+=====+
```

Listing 12: Stack preparation

4.6 Limitations

Our tool is still a prototype and many things can be improved. We rely on external tools who present some flaws and limitations.

Movfuscator: It was built as toy-compiler for C programs only. In theory, it is possible to overcome this problem by recompiling the source code to an another language using LLVM. An example of translation from C++ to C was demonstrated by the author⁷.

Ropper: is not under active maintenance anymore, one solution would be to implement one from the ground up and adapt our tool. Besides, the team behind PSHAPE demonstrated that gadget extraction still has some room for improvement.

Currently, we can not guarantee a side-effect-free ROP chain as we only search for incompatibilities between a single payload instruction and its associated gadget chain. We do not verify whether the chain for the current payload instruction affects a value stored beforehand. One possibility would be to flag the registers holding important value and restraining from being used until it has been used.

We support x86 32bits and instructions using full 32bits registers. Extending the support should be fairly easy but time-consuming.

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 demon-

⁷see movfuscator GitHub page

strated 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

In its current state, our tool is a prototype in its early stage. It could be improved in two different ways: either we decide to extend the current functionalities or we shift our initial perspective to obtain a more practical program.

Improving the robustness of the current tool using theorem provers. Extending the amount of supported instructions to cover all variations of mov instructions. Support to x86_64 architecture would also be interesting. Another essential feature is a more complete side-effect verification on the gadget chain.

Migrating the tool into an LLVM backend would allow to use any source language compilable to LLVM IR. However, this also means abandoning the idea of using mov instructions only which was one of our principal motivation. That being said, the LLVM project has a big community that keeps growing. To this day, no ROP compilers were integrated into LLVM.

7 REFERENCES

- [1] SolarDesigner, “Return-into-lib(c) exploits.” <http://seclists.org/bugtraq/1997/Aug/63>, August 10, 1997.
- [2] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of CCS 2007* (S. De Capitani di Vimercati and P. Syverson, eds.), pp. 552–61, ACM Press, Oct. 2007.
- [3] 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.
- [4] C. Domas, “M/o/vfuscator, the single instruction compiler.” <http://www.github.com/xoreaxeaxe/movfuscator>.
- [5] S. Dolan, “mov is turing-complete.” <http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf>, July 19, 2013.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] AlephOne, “Smashing the stack for fun and profit.” <http://insecure.org/stf/smashstack.html>, November 08, 1996.
- [10] B. Mulder, “On the subject of Control Flow Integrity: A comparative literature study,” Feb. 2016.
- [11] 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.
- [12] S. Schirra, “Ropper.” <https://github.com/sashs/Ropper>.
- [13] N. A. Quynh, “Capstone: The ultimate disassembler.” <https://github.com/aquynh/capstone>.

A Appendix

```
#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 13: Fibonacci.c

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

```
#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 14: Hanoi_towers.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

```
Dumping target payload <test_programs/fibonacci.o>:
-----
(...)
15c: 8b 04 8d 00 00 00 00  mov  eax,DWORD PTR [ecx
      *4+0x0]
163: 8b 15 00 00 00 00      mov  edx,DWORD PTR ds:0x0
169: 89 10                  mov  DWORD PTR [eax],edx
16b: c7 05 00 00 00 00 00  mov  DWORD PTR ds:0x0,0x0
(...)
```

Listing 15: Sample movfuscator output

```
#!/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 16: ROP exploit in Python