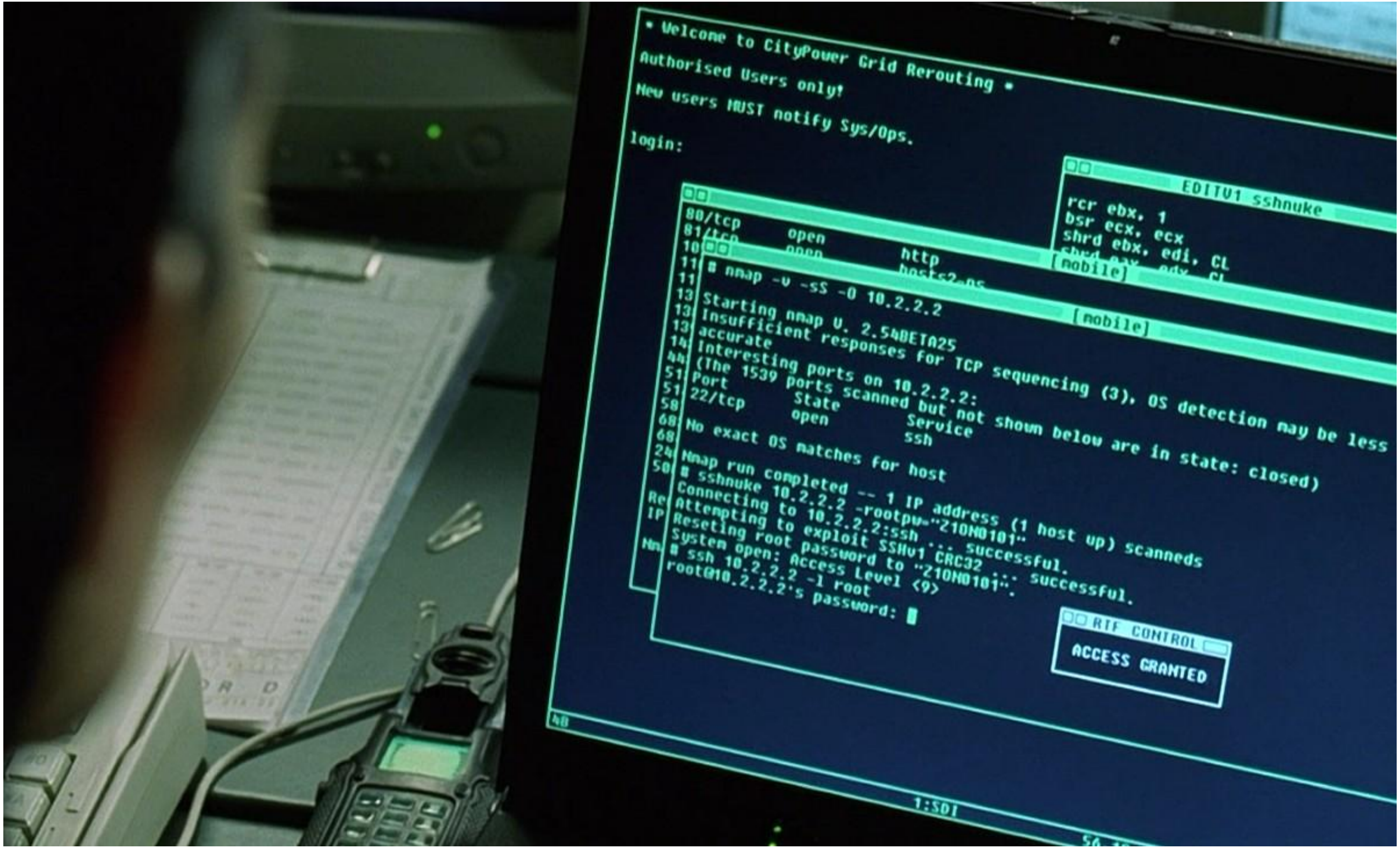
Lecture 11

PWNing 1: Basics

Today

- What is pwn category?
- What went wrong here?
- How to get a shell?

What is “pwn”?


* Welcome to CityPower Grid Rerouting *
Authorised Users only!
New users MUST notify Sys/Ops.
login:
EDITV1 sshnuke
rcr ebx, 1
bsr ecx, ecx
shrd ebx, edi, CL
80/tcp open
81/tcp
http
[mobile]
nmap -v -sS -O 10.2.2.2
Starting nmap V. 2.54BETA25
Insufficient responses for TCP sequencing (3), OS detection may be less accurate
Interesting ports on 10.2.2.2:
(The 1539 ports scanned but not shown below are in state: closed)
Port State Service
22/tcp open ssh
No exact OS matches for host
Nmap run completed -- 1 IP address (1 host up) scanned
sshnuke 10.2.2.2 -rootpw="Z1ON0101"
Connecting to 10.2.2.2:ssh ... successful.
Attempting to exploit SSHv1 CRC32 ... successful.
Reseting root password to "Z1ON0101".
System open: Access Level <9>
ssh 10.2.2.2 -l root
root@10.2.2.2's password:
RTF CONTROL
ACCESS GRANTED
1:SDI

Pwn

- Flag is hidden behind vulnerable service.
- We usually need to:
  - Find vulnerability
  - Make an exploit
  - Use it to get a flag (sometimes by getting access to shell)

```
b33f@Dev:~$ python Desktop/bof.py
[+] Opening connection to pwnable.kr on port 9000: Done
[*] Switching to interactive mode
$
$ id
uid=1008(bof) gid=1008(bof) groups=1008(bof)
$ ls
bof
bof.c
flag
log
log2
super.pl
$ cat flag
```

What’s wrong with this service?

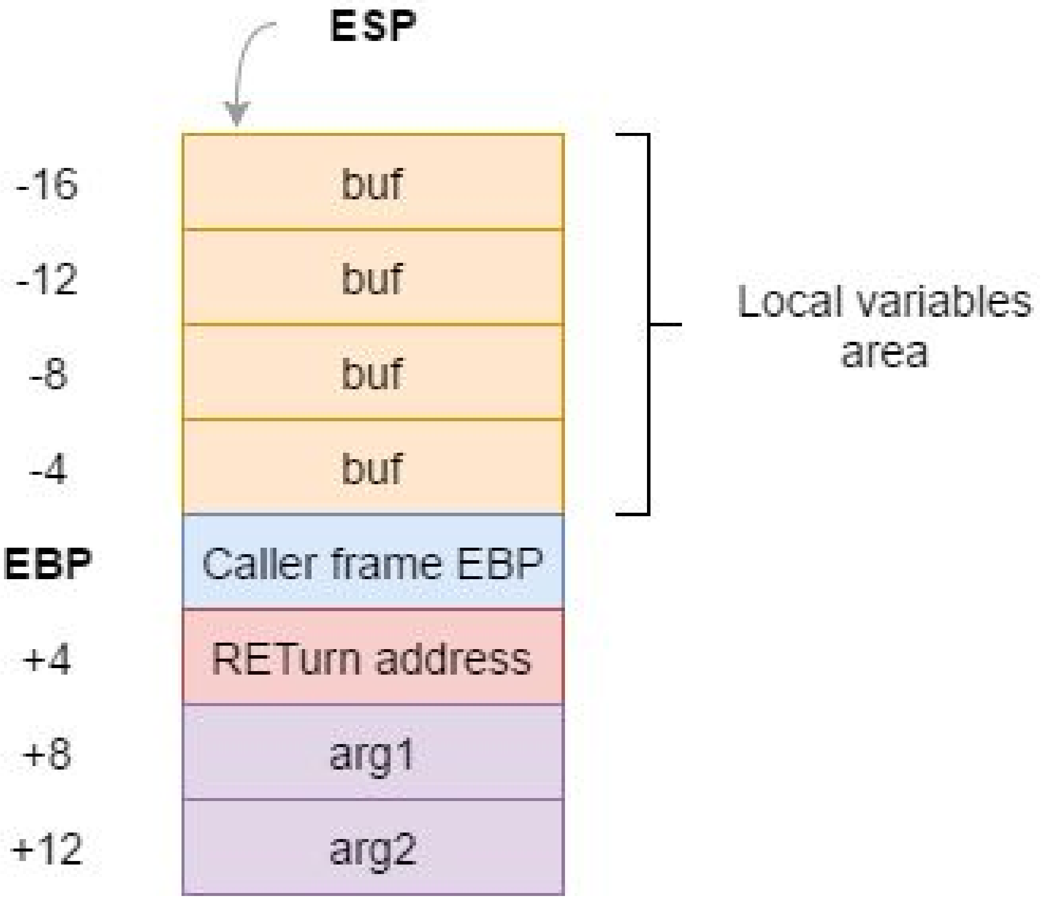
Vulnerability types

- Buffer overflow
- Use-after-free
- Double free
- Format string attack
- Integer problems: overflow/underflow, signedness
- Race conditions
- ...

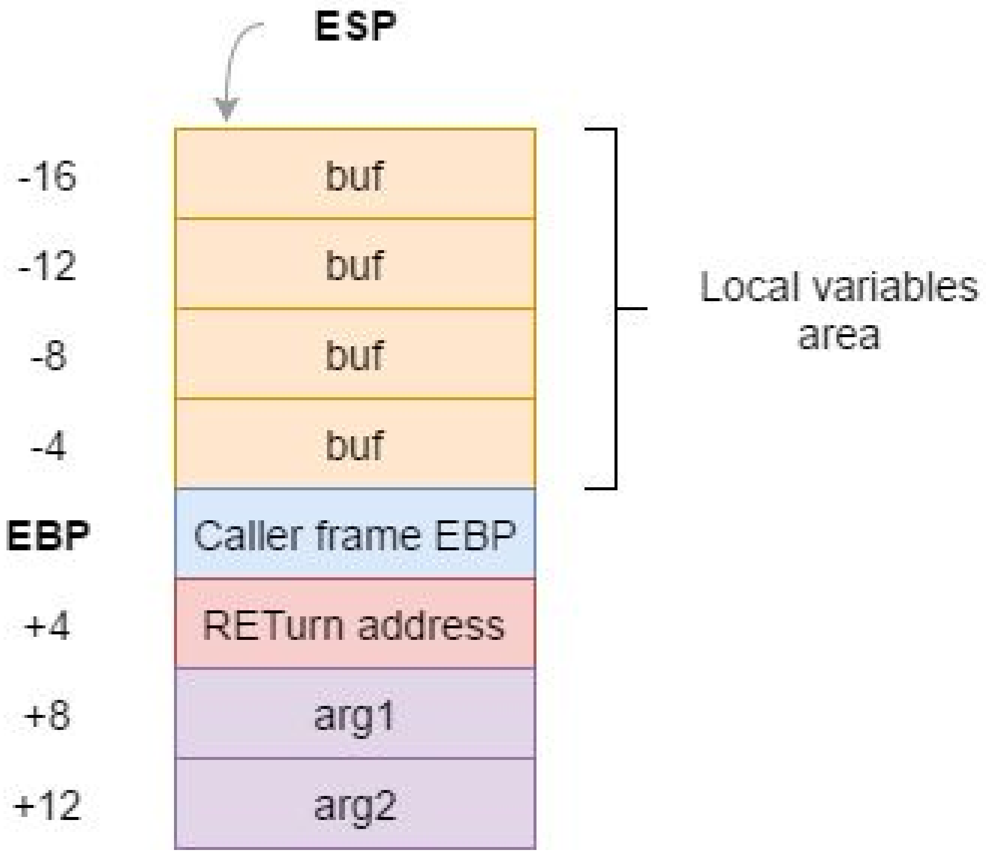
How to get a shell?

Stack buffer overflow

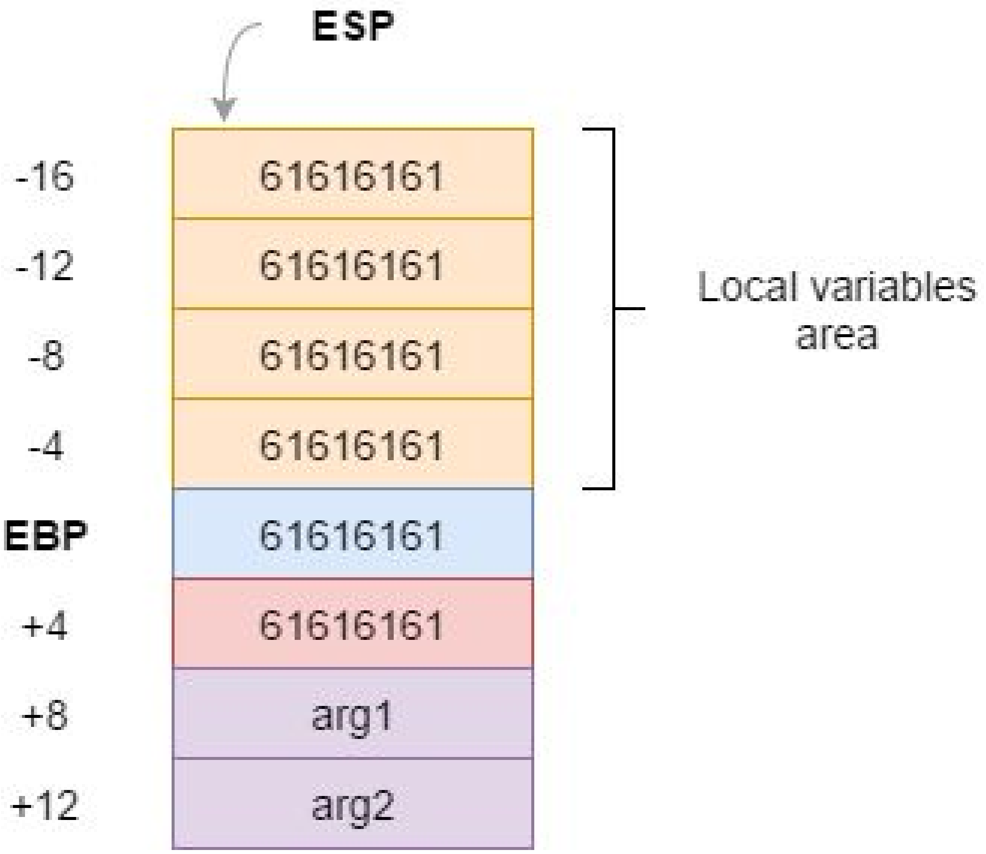
```
int readArgument(int arg1, int arg2)
{
    char buf[16];
    scanf("%s", buf);
    // ...
}
```


ESP
-16
buf
-12
buf
-8
buf
-4
buf
Local variables area
EBP
Caller frame EBP
+4
RETurn address
+8
arg1
+12
arg2

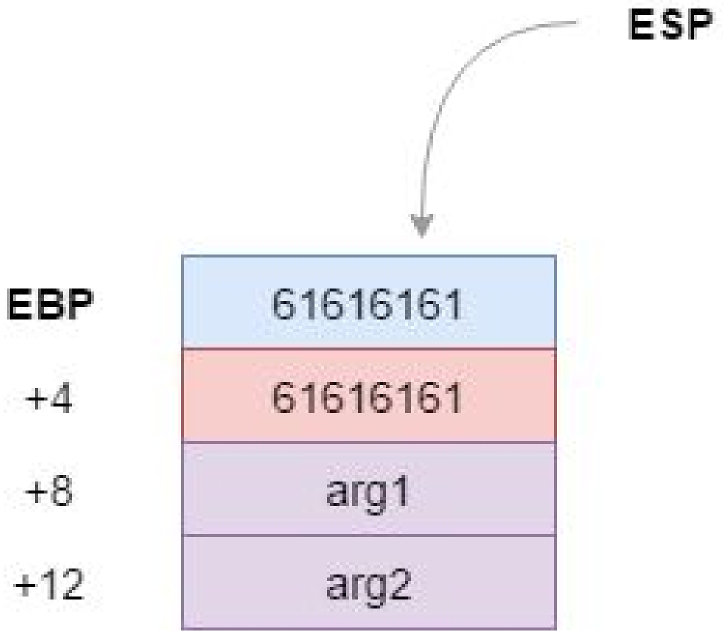
```
lea     eax, [ebp-16]
push    eax
push    addr "%s"
call    scanf
add     esp, 8
// ...
mov     esp, ebp
pop     ebp
ret
```



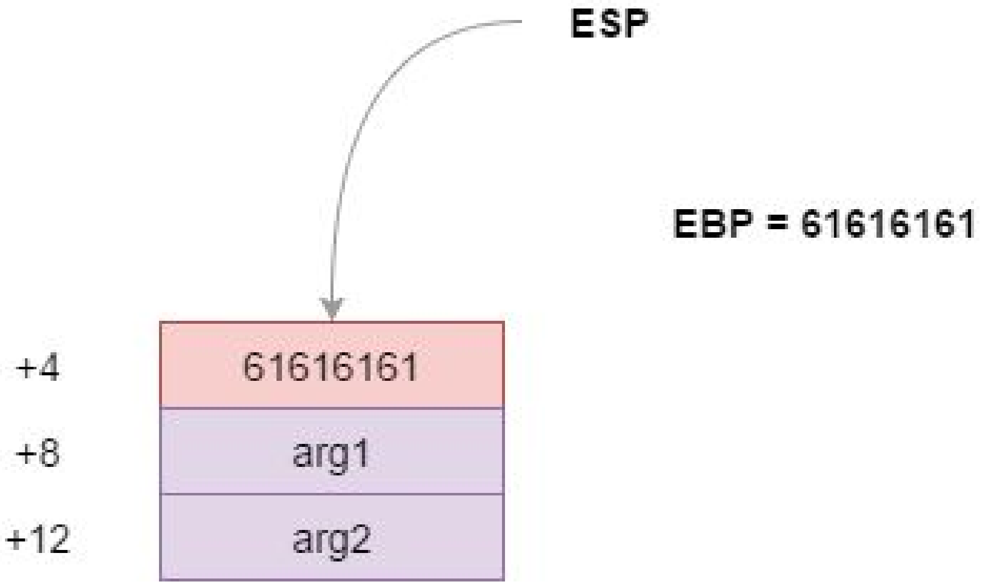
```
lea     eax, [ebp-16]
push    eax
push    addr "%s"
call    scanf
add     esp, 8
// ...
mov     esp, ebp
pop     ebp
ret
```



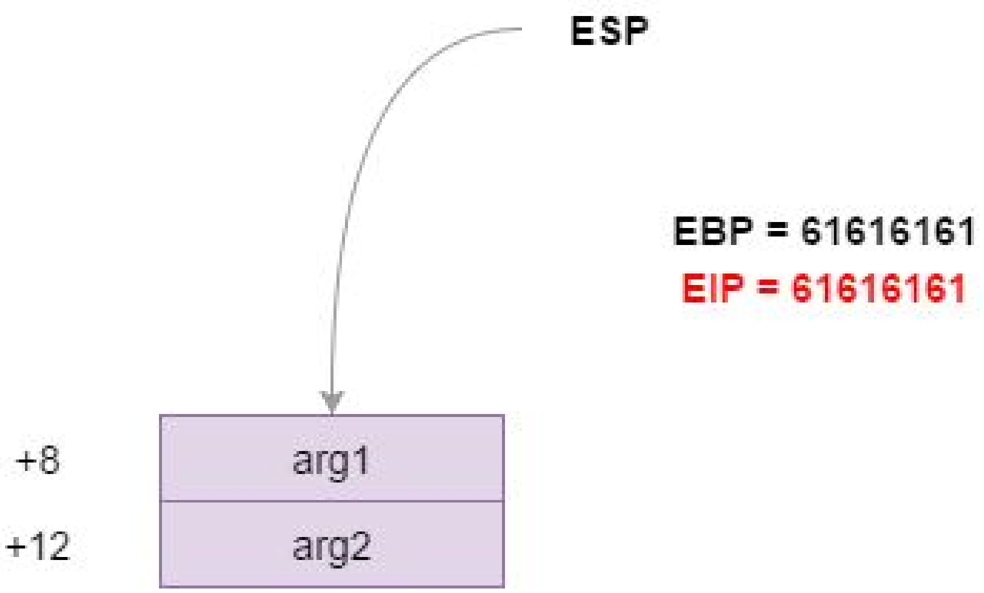
```
lea     eax, [ebp-16]
push    eax
push    addr "%s"
call    scanf
add     esp, 8
// ...
mov     esp, ebp
pop     ebp
ret
```



```
lea     eax, [ebp-16]
push    eax
push    addr "%s"
call    scanf
add     esp, 8
// ...
mov     esp, ebp
pop     ebp
ret
```



```
lea     eax, [ebp-16]
push    eax
push    addr "%s"
call    scanf
add     esp, 8
// ...
mov     esp, ebp
pop     ebp
ret
```



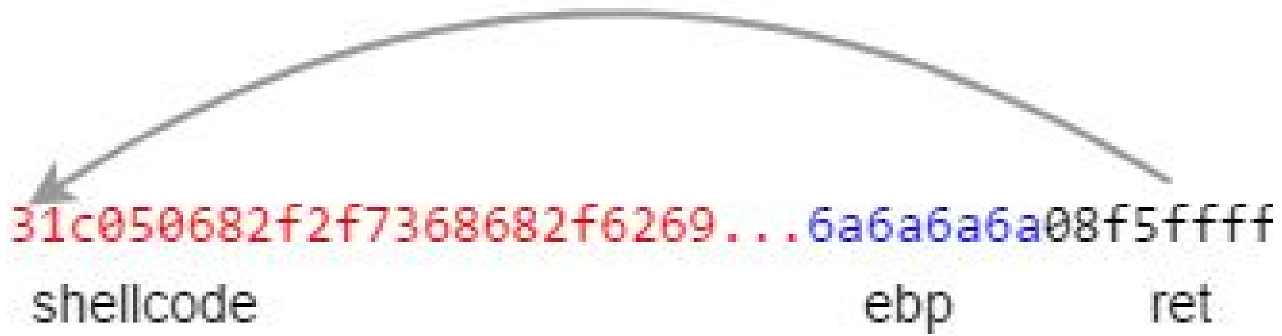
Shellcode anatomy

```
execve("/bin//sh", ["/bin//sh"], nullptr)
```

Shellcode anatomy

```
xor     edx,edx          // envp nullptr
xor     eax,eax
push    eax              NULL-byte
push    0x68732f2f       ("//sh")
push    0x6e69622f       ("/bin")
mov     ebx, esp
push    eax               // NULL-byte
push    ebx               // filename
mov     ecx, esp          // argv address
mov     al, 0x0B          // 0x11 - execve
int     0x80
```

Shellcode anatomy



“My exploit runs locally, but doesn’t work on remote. Hint plz”

ASLR

- Address Space Layout Randomization
- Randomized position of basic process areas:
  - Stack/heap
  - Libraries (libc)
  - Executable base
    (for Position Independent Execs - PIE)

ASLR

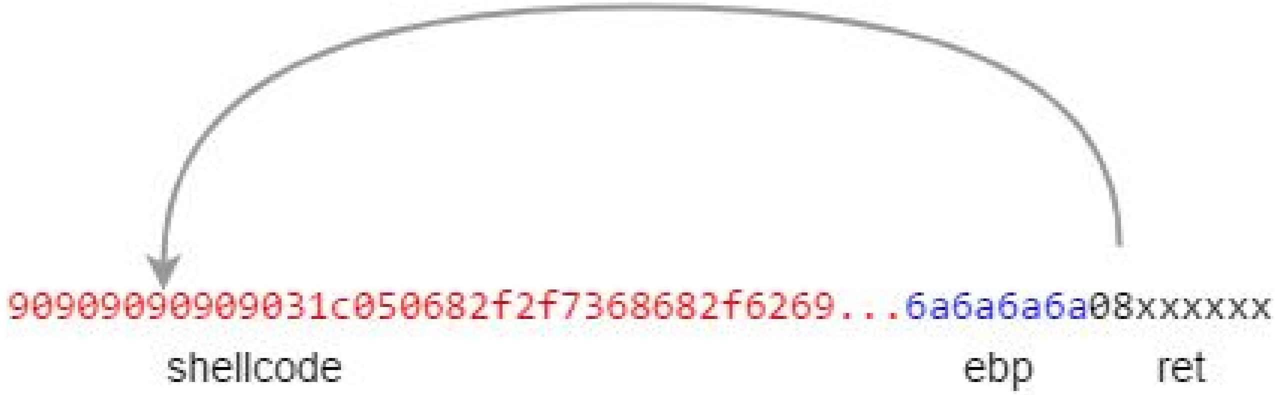
- Usually turned off under GDB.
- But still.. stack position is difficult to predict
- What we can do?
    - Partial return address overwrite
      (page granularity, stack alignment)
    - NOP slides

Stack randomization bypass

31c050682f2f7368682f6269...6a6a6a6a08xxxxxx

shellcode ebp ret

Stack randomization bypass



0x90 - NOP

DEP

- Data Execution Prevention
- Prevents execution of code from non-executable regions (e.g. stack…)
- Support by hardware (NX-bit on memory pages)
- Apart from CPU support, binary must be NX-enabled to enable this feature.

checksec

```
[*] '/home/psrok1/ctf/plaidctf/zamboni_b5c1b58ada23773e86306d2374ce871f'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE
```

How to deal with DEP?

- Now, we’re unable to run code from stack.
- But hey.. we have lots of executable code in process virtual memory (own code, libraries)
- We need to use some **gadgets** in our **ROP chain**

ROP

- Return Oriented Programming
- We can build our shellcode with parts of code, ended with RET instruction (**gadgets**)

ROP

0x00804040	xor eax, eax add eax, ebx **ret**
0x00404222	mov ebx, esp **ret**
0x004043fa	pop eax **ret**
0x00403000	**ret**

```
<buffer>

...
00804040
00404222
004043fa
00000000
00403000
0ff88080
...
```

function return address (00804040)

ROPgadgets

```
psrokl@ubuntu:~/ctf/plaidctf$ ROPgadget --binary zamboni_b5c1b58ada23773e86306d2374ce871f | head
Gadgets information
============================================================
0x0000000000414f48 : adc al, 0 ; add bl, ch ; add ecx, dword ptr [rax - 0x77] ; ret
0x0000000000416eb2 : adc al, 0 ; add byte ptr [rax - 0x77], cl ; ret
0x0000000000416d81 : adc al, 0 ; add byte ptr [rax - 0x77], cl ; ret 0x8d48
0x000000000041a7ca : adc al, 0x11 ; add byte ptr [rax], al ; nop ; leave ; ret
0x000000000042485c : adc al, 0x4b ; add byte ptr [rax], al ; ret 0xff94
0x0000000000415146 : adc al, byte ptr [rax] ; add bl, ch ; add ecx, dword ptr [rax - 0x77] ; ret
```

In tested binary: 1855 gadgets found

+ lots in libc etc.

Return-to-libc

- We should add to our ROP chain some useful libc calls (call to system() or whatever we want)
- But... where is libc?

GOT & PLT

```
                        CONTENTS, ALLOC, LOAD, READONLY, DATA
10 .init                0000001a  0000000000400510  0000000000400510  00000510  2**2
                        CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .plt                 000000a0  0000000000400530  0000000000400530  00000530  2**4
                        CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .text                00002772  00000000004005d0  00000000004005d0  000005d0  2**4
                        CONTENTS, ALLOC, LOAD, READONLY, CODE
21 .got                 00000008  00000000006032a0  00000000006032a0  000032a0  2**3
                        CONTENTS, ALLOC, LOAD, DATA
22 .got.plt             00000060  00000000006032a8  00000000006032a8  000032a8  2**3
                        CONTENTS, ALLOC, LOAD, DATA
23 .data                00000010  0000000000603308  0000000000603308  00003308  2**3
                        CONTENTS, ALLOC, LOAD, DATA
24 .bss                 00000038  0000000000603318  0000000000603318  00003318  2**3
                        ALLOC
25 .comment             00000047  0000000000000000  0000000000000000  00003318  2**0
                        CONTENTS, READONLY
```

GOT & PLT

- .got (Global Offset Table)
  - table of addresses to data section
  - used for simple relocation of static data (relative location of GOT is well-known)
- .plt (Procedure Linkage Table)
- .got.plt (GOT for PLT - part of GOT used by PLT)

Lazy resolution

```
=> 0x804841f <main+20>:    push    $0x80484c0
   0x8048424 <main+25>:    call    0x80482e0 <printf@plt>
   0x8048429 <main+30>:    add      $0x10,%esp
   0x804842c <main+33>:    mov      $0x0,%eax
   0x8048431 <main+38>:    mov      -0x4(%ebp),%ecx
   0x8048434 <main+41>:    leave
   0x8048435 <main+42>:    lea      -0x4(%ecx),%esp
   0x8048438 <main+45>:    ret
```

Lazy resolution

```
(gdb) x/8i 0x80482e0
   0x80482e0 <printf@plt>:               jmp    *0x804a00c
   0x80482e6 <printf@plt+6>:             push   $0x0
   0x80482eb <printf@plt+11>:            jmp    0x80482d0
   0x80482f0 <__libc_start_main@plt>:    jmp    *0x804a010
   0x80482f6 <__libc_start_main@plt+6>:  push   $0x8
   0x80482fb <__libc_start_main@plt+11>: jmp    0x80482d0
   0x8048300:                            jmp    *0x8049ffc
   0x8048306:                            xchg    %ax,%ax
```

Lazy resolution

```
(gdb) x/8i 0x80482e0
   0x80482e0 <printf@plt>:              jmp    *0x804a00c
   0x80482e6 <printf@plt+6>:            push   $0x0
   0x80482eb <printf@plt+11>:           jmp    0x80482d0
   0x80482f0 <__libc_start_main@plt>:   jmp    *0x804a010
   0x80482f6 <__libc_start_main@plt+6>: push   $0x8
   0x80482fb <__libc_start_main@plt+11>: jmp   0x80482d0
   0x8048300:                           jmp    *0x8049ffc
   0x8048306:                           xchg    %ax,%ax

(gdb) x/dx 0x804a00c
   0x804a00c:     0x080482e6
```

Lazy resolution

```
(gdb) x/8i 0x80482d0
   // plt:_dl_runtime_resolve
   0x80482d0:                    pushl 0x804a004
   0x80482d6:                    jmp    *0x804a008
   0x80482dc:                    add    %al,(%eax)
   0x80482de:                    add    %al,(%eax)
   0x80482e0 <printf@plt>:       jmp     *0x804a00c
   0x80482e6 <printf@plt+6>:     push    $0x0
   0x80482eb <printf@plt+11>:    jmp     0x80482d0
   0x80482f0 <__libc_start_main@plt>:    jmp   *0x804a010
(gdb) x/dx 0x804a008
   0x804a008:     0xf7fedf00 <ld:_dl_runtime_resolve>
```

Lazy resolution

```
   0x804841f <main+20>:    push    $0x80484c0
   0x8048424 <main+25>:    call    0x80482e0 <printf@plt>
=> 0x8048429 <main+30>:    add      $0x10,%esp
   0x804842c <main+33>:    mov      $0x0,%eax
   0x8048431 <main+38>:    mov      -0x4(%ebp),%ecx
   0x8048434 <main+41>:    leave
   0x8048435 <main+42>:    lea      -0x4(%ecx),%esp
   0x8048438 <main+45>:    ret
```

Lazy resolution

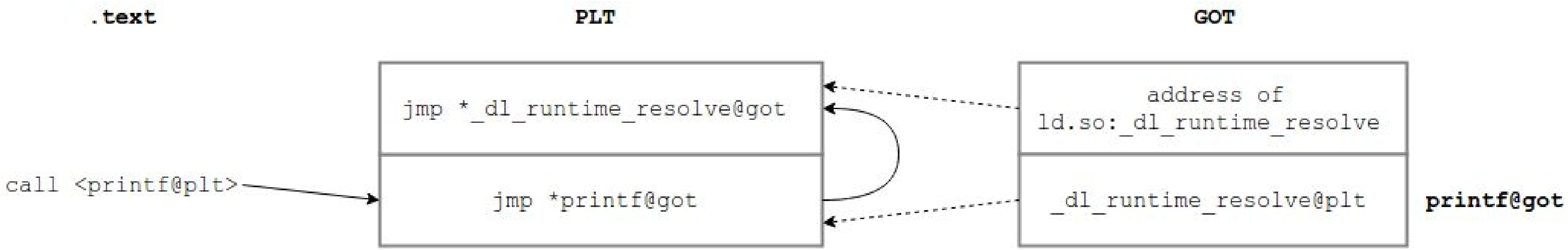
```
(gdb) x/8i 0x80482e0
   0x80482e0 <printf@plt>:              jmp    *0x804a00c
   0x80482e6 <printf@plt+6>:            push   $0x0
   0x80482eb <printf@plt+11>:           jmp    0x80482d0
   0x80482f0 <__libc_start_main@plt>:   jmp    *0x804a010
   0x80482f6 <__libc_start_main@plt+6>: push   $0x8
   0x80482fb <__libc_start_main@plt+11>: jmp   0x80482d0
   0x8048300:                           jmp    *0x8049ffc
   0x8048306:                           xchg    %ax,%ax
```

Lazy resolution

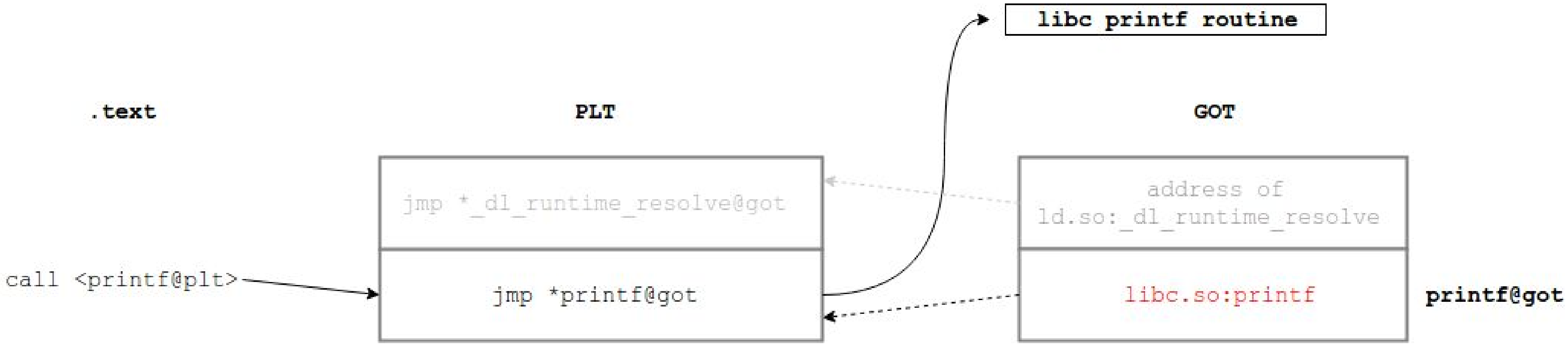
```
(gdb) x/8i 0x80482e0
   0x80482e0 <printf@plt>:                jmp    *0x804a00c
   0x80482e6 <printf@plt+6>:              push   $0x0
   0x80482eb <printf@plt+11>:             jmp    0x80482d0
   0x80482f0 <__libc_start_main@plt>:     jmp    *0x804a010
   0x80482f6 <__libc_start_main@plt+6>:   push   $0x8
   0x80482fb <__libc_start_main@plt+11>:  jmp    0x80482d0
   0x8048300:                             jmp    *0x8049ffc
   0x8048306:                             xchg    %ax,%ax

(gdb) x/dx 0x804a00c
   0x804a00c:     0xf7e3e670 <libc.printf>
```

Lazy resolution



Lazy resolution



Return-to-libc

- .plt/.got sections position is usually well-known
  - unless we have ASLR & PIE executable
- If exploited executable uses some interesting routines, we can directly call them in our ROP chain

Return-to-libc

- Sometimes we need to call routine, which is not directly available from GOT/PLT (e.g. system)
- We need to resolve them manually

Return-to-libc

1. Leak GOT contents
2. Determine libc version and base addr (based on two known routine addresses)
3. Calculate system() address
4. Use buffer overflow once again to fetch second-stage payload
5. Return-to-libc system()... pwned!

libcdb

libcdb.com: the libc data base

/ search /

search

symbolA name: __libc_start_main

symbolA address: 0xb74a43e0

symbolB name: setsockopt

symbolB address: 0xb757c7b0

search

results

results: 2 item(s) found

- Libc: libc-2.15.so
- Libc: libc-2.15_2.so

**** stack smashing detected ***

Alas! Brave adventurer...

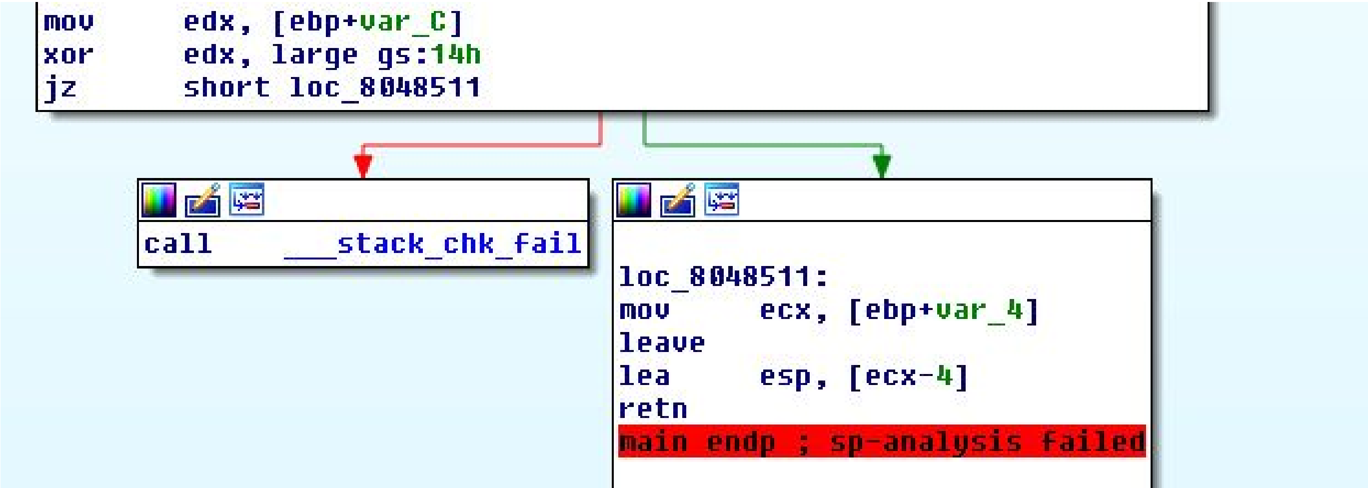
checksec

```
[*] '/home/psrok1/ctf/plaidctf/zamboni_b5c1b58ada23773e86306d2374ce871f'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE
```

Stack canaries

- Now.. we’re “unable” to do buffer overflow on stack
- Compiler protects stack with random, known value. When function returns - canary is checked and if it doesn’t match: program is terminated.

Stack canaries



Canary bypass

- To exploit buffer overflow with canaries enable:
  - Bypass check (Windows SEH overwrite attack)
  - Use another vuln to leak canary value from stack
  - Use another vuln..

Another ways to pwn

- Buffer overflow
    - ~~Stack-based (killed)~~
    - Heap-based
- Use-after-free
- Double free
- Format string attack
- Integer problems: overflow/underflow, signedness
- Race conditions
- ...

So.. How to get a shell?

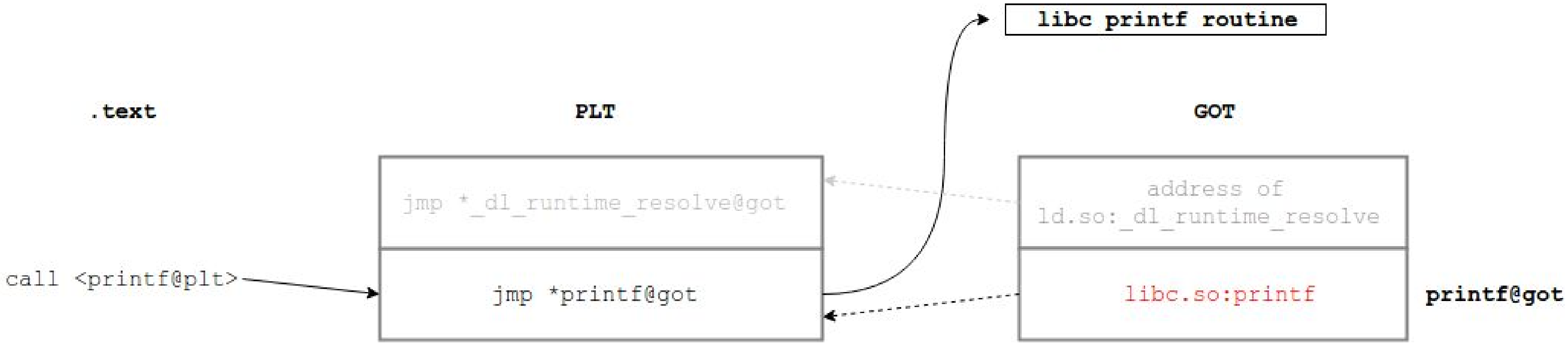
What we want?

- Information leak (e.g. arbitrary read)
  - Leak stack canaries
  - Leak libc base using GOT
  - Leak flag (sometimes)
- Write-what-where condition
  - Maybe we should overwrite something else than stack!

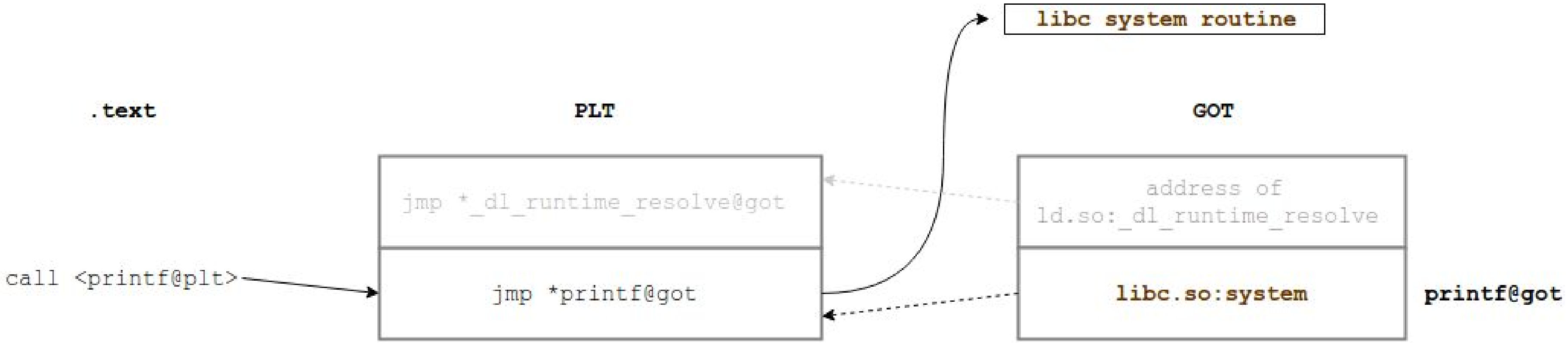
What we can overwrite

- Most interesting are function pointers
- We have lots of function pointers in GOT
- Overwrite GOT?

GOT overwrite



GOT overwrite



But.. what is RELRO?

```
[*] '/home/psrok1/ctf/plaidctf/zamboni_b5c1b58ada23773e86306d2374ce871f'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE
```

RELRO

- RELocation Read-Only
- Linker sets GOT as read-only after relocations
- Partial RELRO:
  - non-PLT GOT is read-only (GOT still writeable)
  - reordered sections (GOT precedes .data sections)
- Full RELRO
  - .got.plt is also read-only
  - … another one bites the dust

Another one bites the dust

- Life is hard
    - ASLR (+PIE)
    - DEP (NX)
    - Stack canaries
    - RELRO
    - …
- We need more tricks in our arsenal!

soon...

Bibliography

"Praktyczna inżynieria wsteczna" - Mateusz Jurczyk, Gynvael Coldwind

“Hacking: The art of exploitation” - Jon Erickson

“RPISEC: Modern Binary Exploitation” course materials
https://github.com/RPISEC/MBE