2. Memory Corruption Exploits
ENEE 657

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

Today's Lecture

- Where we've been
  - Intro to security
- Where we're going today
  - Security principles (discuss reading assignment)
  - Memory corruption exploits
  - Homework #1
- Where we're going next
  - No lecture on Monday (Labor Day)
  - Cryptography review (Wednesday)
  - Homework #1 due (Friday)

2

Logistics

- Choose a hacker handle
  - Examples of famous hackers:
    - Aleph One (buffer overflow exploits)
    - Solar Designer (return-to-libc exploits)
    - Dark Avenger (polymorphic malware)
    - th3j35t3r (https://twitter.com/th3j35t3r)
  - Sign up on our Piazza message board with your new handle
    - Don't use your real name
    - Sign up link at http://ter.ps/enee657
- Reading assignments
    - Read papers, but don't post critiques (for now)
    - More details on critiques later

3

Reading: J. Saltzer & M. Schroeder, SOSP'73

Design Principles for Secure Systems

5

Memory Corruption

Recall: Correctness versus Security

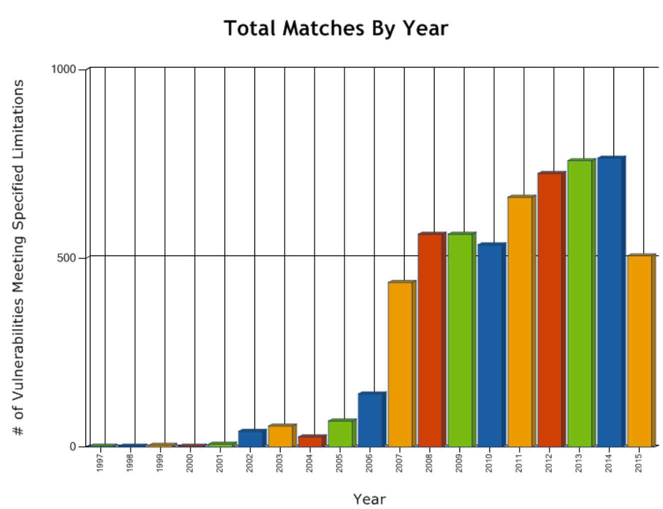
- System **correctness**: system satisfies specification
  - For reasonable input, get reasonable output
- System **security**: system properties preserved in face of attack
  - For unreasonable input, output not completely disastrous
- Main difference: **intelligent adversary trying to subvert system and to evade defensive techniques**

Buffer Errors

- A **buffer** is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of input data
  - The attacker controls the inputs
- What can the attacker do?
  - If the buffer is filled with executable code, the victim's machine may be tricked into executing it (**remote code execution** exploit)
    - First major exploit: 1988 Internet worm (more on this later)
  - Or it may reveal parts of the computer's memory (**information disclosure** exploit)
    - Recent example: Heartbleed (more on this later)
  - Attack can exploit any memory operation
    - Pointer assignment, format strings, memory allocation and de-allocation, function pointers, calls to library routines via offset tables …

8

Buffer Errors – Rate of Discovery



Source: National Vulnerability Database (NVD)

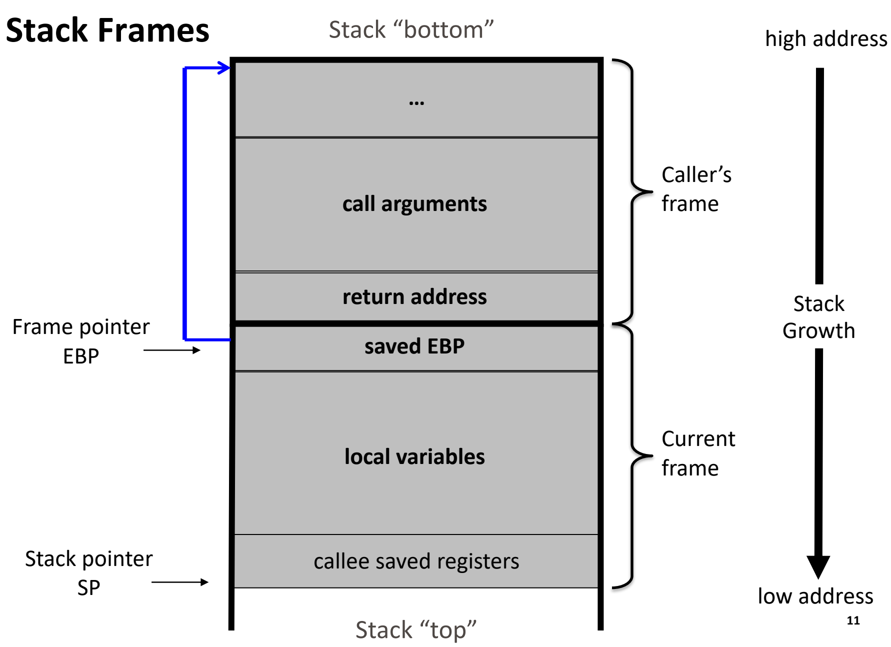
9

What You Need to Know

- Understand C functions and the stack
- Know how system calls are made
- Know the `exec()` system call
- Know the CPU and OS on the target machine
  - Little endian vs. big endian (x86 vs. Motorola)
  - Stack frame structure (Unix vs. Windows)
  - The homework uses x86 (32 bit) running Linux (Ubuntu)

10

Stack Frames



11

C Function Call and Return

- When a C function is called
  - A new stack frame is created
    - Push arguments, return address, EBP of caller frame onto stack
  - Make EBP point to the base of the new frame
  - Jump to the start of the function
    - The function allocates space for local variables by increasing SP
- When a C function returns
  - SP <- EBP
  - Pop the saved frame pointer into EBP
  - Jump to the return address

12

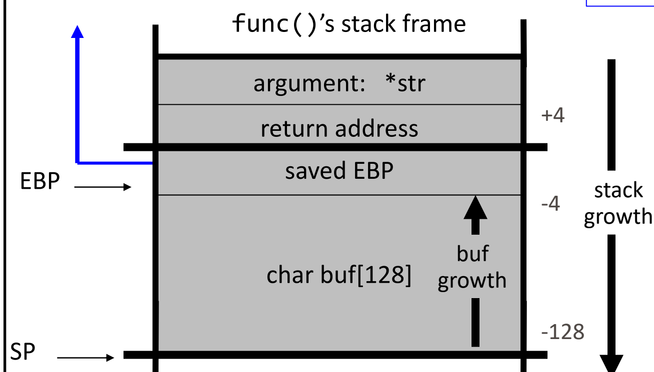
What are Buffer Overflows?

Suppose a web server contains this function:

```
void func(char *str) {
  char buf[128];

  strcpy(buf, str);
  do-something(buf);
}
```

Allocate local buffer (128 bytes reserved on stack)

Copy argument into local buffer



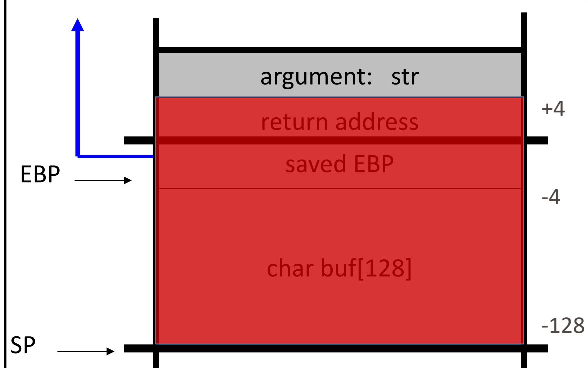
13

What are Buffer Overflows?

What happens when str is 136 bytes long?

```
void func(char *str) {
   char buf[128];

   strcpy(buf, str);
   do-something(buf);
}
```

After `strcpy`:



Problem:
no length checking in `strcpy()`

Basic Stack-Based Overflow

[Aleph One – Smashing the Stack for Fun and Profit]

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- The buffer overflow must do two things:
  - **Hijack the program control**
    - Example: overwrite the **value in the RET position** to point to the beginning of attack assembly code in memory
    - If you return outside the valid address space, the application will crash with a segmentation violation (SEGFAULT)
  - **Ensure that the attack code is stored** somewhere in memory
    - Example: put it **in the buffer**
    - You must correctly guess in which stack position his buffer will be when the function is called
    - You can also achieve this goal without injecting code (more on this later)

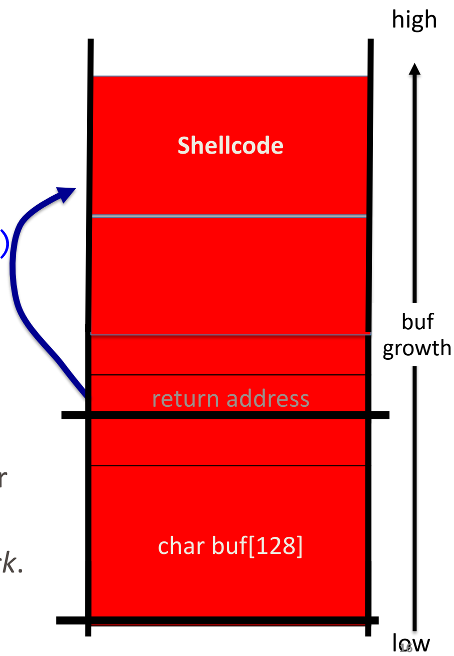
Basic Stack Exploit

Suppose `*str` is such that after `strcpy()` the stack looks like this:

Attack code: `exec("/bin/sh")`
(known as "shellcode")

When `func()` exits, the attacker gets a shell!

Note: the attack code runs *in stack*.



The NOP Sled

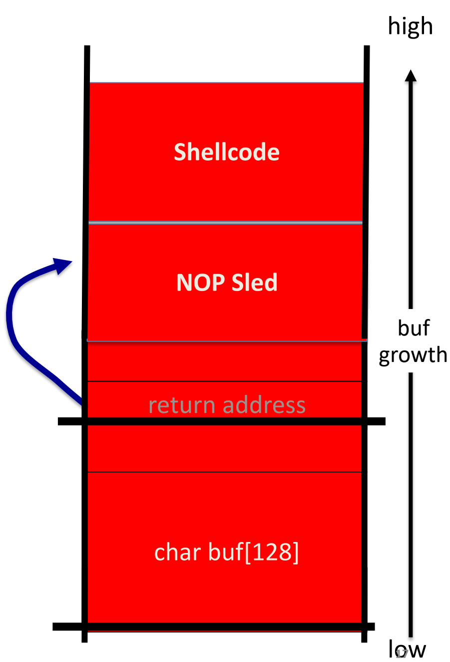
Problem: how does the attacker determine the return address?

Solution: NOP sled

- Guess approximate stack state when func() is called
- Insert many NOP (No OPeration) instructions before the shellcode:

```
nop
xor eax,eax
inc ax; dec ax
…
```

- Jump somewhere in the middle NOP



Some Complications

- The buffer should not contain the '\0' character (**why?**)
  - That means that you cannot have a 0 byte in the shellcode or return address
  - Inspect shellcode and replace with equivalent instructions w/o a 0 byte
  - Set return address to some place in the NOP sled w/o a 0 byte
- Overflow should not crash program before `func()` exits
  - Stack layouts vary across different platforms
  - Make sure you don't copy too many bytes into `buf[]` and run of the valid address space
    - Make sure that your attack input is a properly terminated string (has '\0' **at the end**)
  - Use a NOP sled
  - You can copy the jump target multiple times if unsure of the offset

18

What If You Cannot Inject Code on the Stack?

- Over the years, several defenses against buffer overflow have been proposed
  - Examples: ensure integrity of stack frames ("stack canaries"), randomize memory layout (ASLR), make stack non-executable (DEP, NX bit)
  - These generally target the two necessary steps for buffer overflow
- **Hijack the program control**
  - Overwrite the **value in the RET position** to point to the beginning of attack assembly code in memory
- **Ensure that the attack code is stored** somewhere in memory
  - ~~Put it **in the buffer**~~
  - Jump to code (already present in memory) that does what you want (e.g. the C library functions)

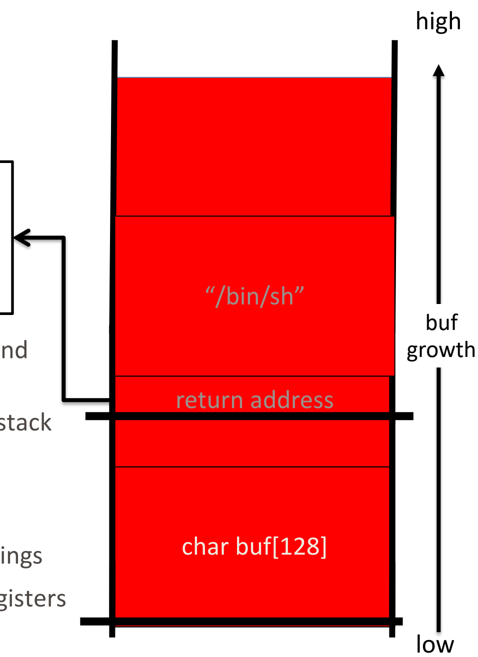
19

Return-to-libc Attack

- Jump to a function in libc

```
int
system(const char *command)
{
        …
}
```

  - `system()` invokes a UNIX command (e.g. `/bin/sh`)
  - You can put the command on the stack



- Limitations
  - 0 bytes to terminate command strings
  - Some functions take args. from registers (**why is this a limitation?**)
  - Overcome by return-oriented programming (more on this later)

What If You Cannot Smash the Return Address?

- **Hijack the program control**
  - ~~Overwrite the **value in the RET position** to point to the beginning of attack assembly code in memory~~
  - Overwrite other things that will ultimately give you control (e.g. EBP, function pointers, exception handlers)
- **Ensure that the attack code is stored** somewhere in memory
  - Put it **in the buffer**

Off-By-One Overflow

- Home-brewed range-checking string copy

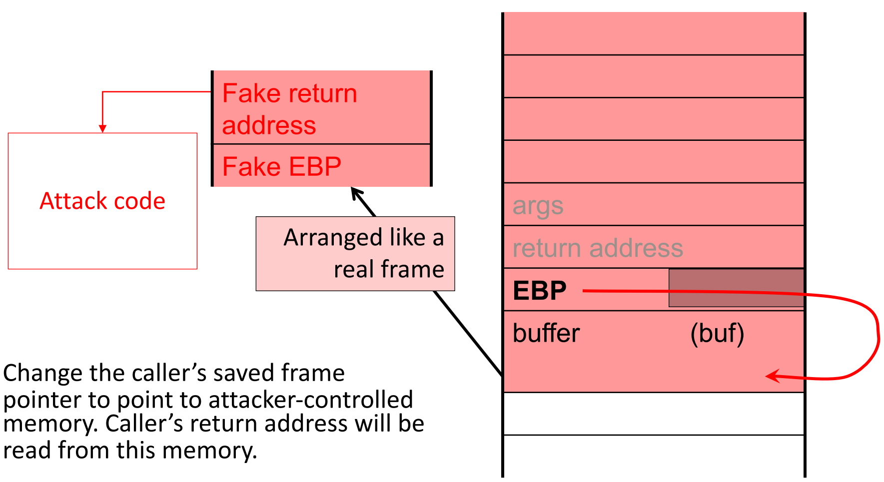
```
 void notSoSafeCopy(int *input) {
     int buffer[512]; int i;

     for (i=0; i<=512; i++)
         buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
     if (argc==2)
        notSoSafeCopy((int*) argv[1]);
}
```

This will copy **513** integers into buffer. Oops!

- 1-int overflow: can't change the return address, but can change saved pointer to **previous** stack frame
  - On little-endian architecture, make it point into buffer
  - The **caller's return address** will be read from the buffer!

22

Smash the Frame Pointer



Change the caller's saved frame pointer to point to attacker-controlled memory. Caller's return address will be read from this memory.

23

Fundamental Causes for Basic Stack Smashing Exploits

- C strings are nul-terminated, rather than specifying the bound
  - Programmer must check the range manually
  - Many unsafe functions in the standard C library
    - strcpy(char *dest, const char *src)
    - strcat(char *dest, const char *src)
    - gets(char *s)
    - scanf(const char *format, …)
    - printf(const char *format, …)
- Stacks grow down and arrays grow up
- Von Neumann architecture: program and data in same memory
  - In addition, for x86: no distinction between executable and readable pages

24

Where Can We Find Buffer Overflows?

- Most operating systems are written in C
  - Internet worms:
    - (1988) Morris worm
    - (2000) Code Red worm
    - (2008) Conficker
    - (2017) WannaCry
- Web browsers
  - (2007) Overflow in Windows animated cursors (ANI). LoadAniIcon()
- Security software
  - (2005) Overflow in Symantec Virus Detection

    test.GetPrivateProfileString "file", **[long string]**
- Cars, embedded devices

25

How Exploits Are Used Today
[Grier et al, CCS 2012]

- Writing successful exploits today requires specialized skills
  - On underground markets, you can buy specialized services and products that provide this function
- Exploit kits
  - Packaged software with a collection of exploits
  - Code for profiling the target and deliver the right exploit
- Exploit services
  - Web sites that exploit vulnerabilities in Web browsers
    - Drive-by-downloads (more on this later)
  - Just redirect your victims to those Web sites

26

Review of Lecture

- What did we learn?
  - Design principles of secure systems
  - Memory corruption attacks: return address, shellcode, stack frames
- Sources
  - Vitaly Shmatikov, Dan Boneh
- What's next?
  - Cryptography review
  - First homework due next Friday

28