ECE/CS230
Computer Systems Security

Charalambos (Harrys) Konstantinou

https://sites.google.com/view/ececs230kaust

Program security

Before we start

- Assignment 1 grades are out
- Discussions are usual

Secure programs

- Why is it so hard to write secure programs?

- A simple answer:
  - Axiom (Murphy):
    Programs have bugs
  - Corollary:
    Security-relevant programs have security bugs

Outline

- Flaws, faults, and failures
- Unintentional security flaws
- Malicious code: Malware
- Other malicious code
- Nonmalicious flaws
- Controls against security flaws in programs

Outline

- Flaws, faults, and failures
- Unintentional security flaws
- Malicious code: Malware
- Other malicious code
- Nonmalicious flaws
- Controls against security flaws in programs

Flaws, faults, and failures

- A flaw is a problem with a program
- A security flaw is a problem that affects security in some way
  - Confidentiality, integrity, availability
- Flaws come in two types: faults and failures
- A fault is a mistake “behind the scenes”
  - An error in the code, data, specification, process, etc.
  - A fault is a potential problem

Flaws, faults, and failures

- A failure is when something actually goes wrong
  - You log in to the library's website, and it shows you someone else's account
  - "Goes wrong" means a deviation from the desired behavior, not necessarily from the specified behavior!
  - The specification itself may be wrong

- A fault is the programmer/specifier/inside view
- A failure is the user/outside view

Finding and fixing faults

- How do you find a fault?
  - If a user experiences a failure, you can try to work backwards to uncover the underlying fault
  - What about faults that haven't (yet) led to failures?
  - Intentionally try to cause failures, then proceed as above
    - Remember to think like an attacker!
- Once you find some faults, fix them
  - Usually by making small edits (patches) to the program
  - This is called "penetrate and patch"
  - "Patch Tuesday"* is a well-known example
    https://en.wikipedia.org/wiki/Patch_Tuesday

Problems with patching

- Patching sometimes makes things worse!
- Why?
  - Pressure to patch a fault is often high, causing a narrow focus on the observed failure, instead of a broad look at what may be a more serious underlying problem
  - The fault may have caused other, unnoticed failures, and a partial fix may cause inconsistencies or other problems
  - The patch for this fault may introduce new faults, here or elsewhere!

Unexpected behavior (1/2)

- When a program's behavior is specified, the spec usually lists the things the program must do
  - The `ls` command must list the names of the files in the directory whose name is given on the command line, if the user has permissions to read that directory

- Most implementors wouldn't care if it did additional things as well
  - Sorting the list of filenames alphabetically before outputting them is fine

Unexpected behavior (2/2)

- But from a security / privacy point of view, extra behaviors could be bad!
  - After displaying the filenames, post the list to a public web site
  - After displaying the filenames, delete the files
- When implementing a security or privacy relevant program, you should consider "**and nothing else**" to be implicitly added to the spec
  - "should do" vs. "shouldn't do"
  - How would you test for "shouldn't do"?

Types of security flaws

- One way to divide up security flaws is by genesis (where they came from)
- Some flaws are intentional/inherent
    - Malicious flaws are intentionally inserted to attack systems, either in general, or certain systems in particular
        - If it's meant to attack some particular system, we call it a targeted malicious flaw
    - Nonmalicious (but intentional or inherent) flaws are often features that are meant to be in the system, and are correctly implemented, but nonetheless can cause a failure when used by an attacker
- Most security flaws are caused by unintentional program errors

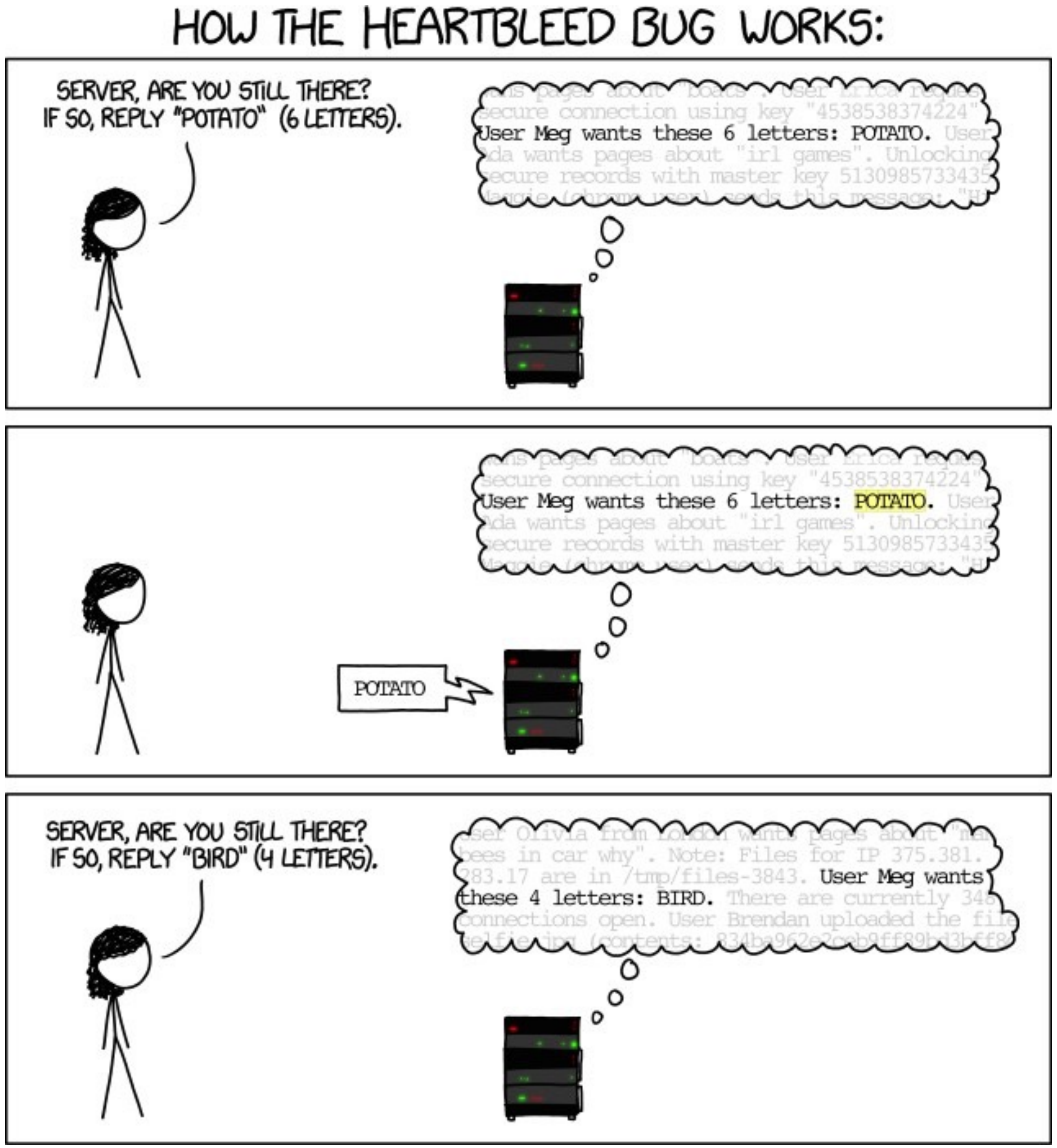
Outline

- Flaws, faults, and failures
- Unintentional security flaws
- Malicious code: Malware
- Other malicious code
- Nonmalicious flaws
- Controls against security flaws in programs

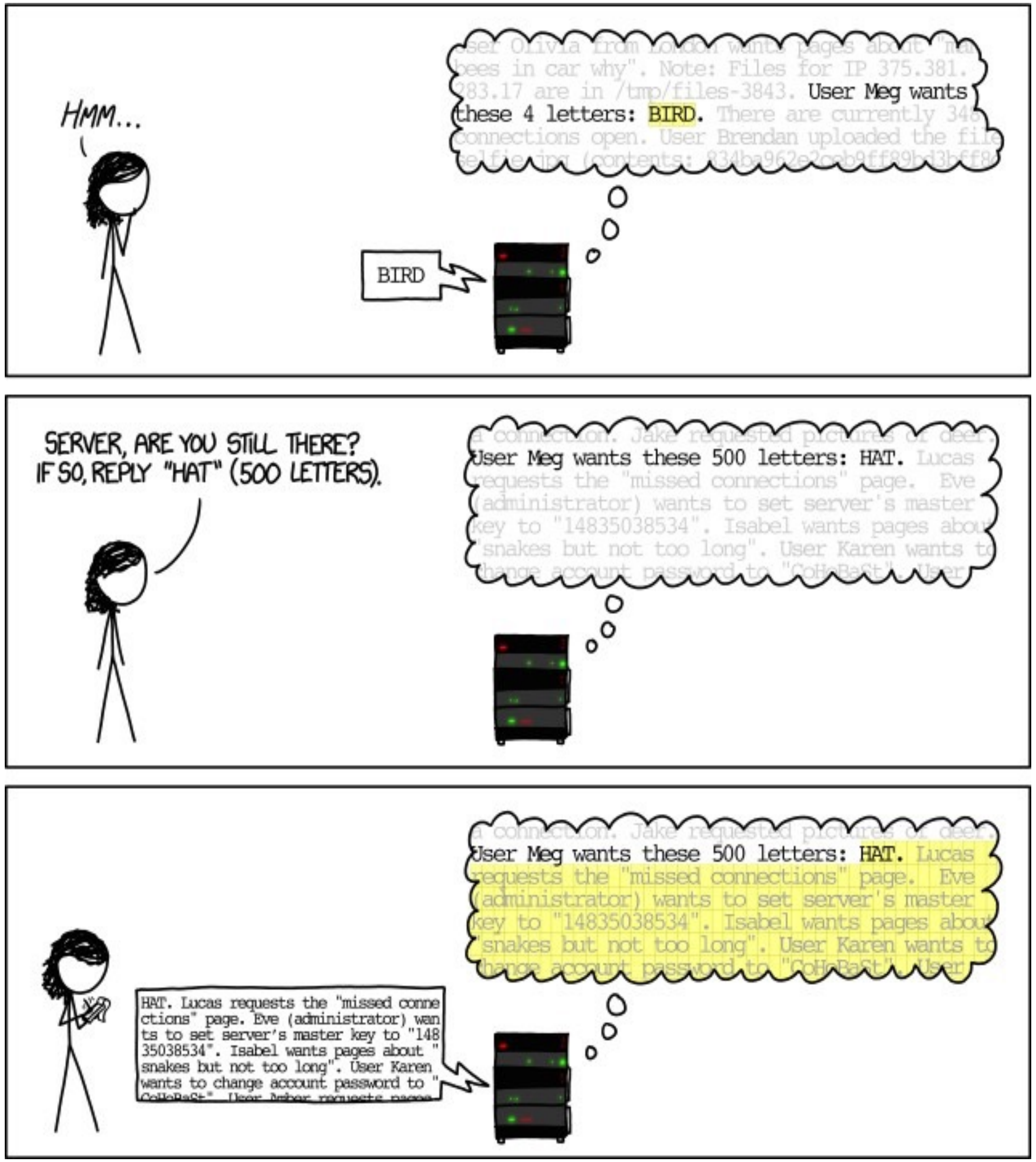
The Heartbleed Bug in OpenSSL (April 2014)

- The TLS Heartbeat mechanism is designed to keep SSL/TLS connections alive even when no data is being transmitted.
- Heartbeat messages sent by one peer contain random data and a payload length.
- The other peer is supposed to respond with a mirror of exactly the same data.

http://imgs.xkcd.com/comics/heartbleed_explanation.png



http://imgs.xkcd.com/comics/heartbleed_explanation.png



The Heartbleed Bug in OpenSSL (April 2014)

- There was a missing bounds check in the code.
- An attacker can request that a TLS server hand over a relatively large slice (up to 64KB) of its private memory space.
- This is the same memory space where OpenSSL also stores the server's private key material as well as TLS session keys.

Apple’s SSL/TLS Bug (February 2014)

- The bug occurs in code that is used to check the validity of the server’s signature on a key used in an SSL/TLS connection.
- This bug existed in certain versions of OSX 10.9 and iOS 6.1 and 7.0.
- An active attacker (a “man-in-the-middle”) could potentially exploit this flaw to get a user to accept a counterfeit key that was chosen by the attacker.

The Buggy Code

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

What’s the Problem?

- There are two consecutive `goto fail` statements.
- The second `goto fail` statement is always executed if the first two checks succeed.
- In this case, the third check is bypassed and `0` is returned as the value of `err`.

Types of unintentional flaws

- Errors
- Integer overflows
- Buffer overflows
- Format string vulnerabilities
- Incomplete mediation
- TOCTTOU errors

Initialization Errors

- What happens when you use something that hasn't been initialized?
- This is a problem with languages like C/C++ and not Java
- This doesn't need to be a problem with C/C++, the programmer can make sure that they are initialized! (e.g. `int a = 0; vs. int a;`)
- This is a perfect example of trading performance over security/correctness.

Initialization Errors: Uninitialized variables

```
int count;
while(count<100)
{
     cout<<count;
     count++;
}
```

Initialization Errors: variable to an uninitialized value

```
int a, b;
int sum=a+b;
cout<<"Enter two numbers to add: ";
cin>>a;
cin>>b;
cout<<"The sum is: "<<sum;
```

When Run:

```
Enter two numbers to add: 1 3
The sum is: -1393
```

Initialization Errors: fixed

```
int a, b;
int sum;
cout<<"Enter two numbers to add: ";
cin>>a;
cin>>b;
sum=a+b;
cout<<"The sum is: "<<sum;
```

Input Validation Error

- An Input Validation Error occurs when an input is NOT CHECKED to ensure it satisfies the assumptions (specifications)
- This is a general type of error (it overlaps with other types according to CWE)
- This is a very COMMON error

Input Validation Error – Example 1

So what happens to the following pseudo-code for the “myCat” program.

- The user inputs a filename/string. The filename is stored in strFilename.
- Execute the following command “cat [strFilename]” where [strFilename] is whatever the string is.

Input Validation Error – Example 1 Continued

- What does the command “myCat hello.txt” result in?
- According to pseudo-code
  - strFilename is now “hello.txt”
  - “cat hello.txt” is run
  - So the contents of hello.txt is printed out onto the screen.
- But remember this is about errors and vulnerabilities. See anything wrong?
- There wasn’t any “input validation”

Input Validation Error – Example 1 Continued

- In Unix commands are separated by ‘;’
- What happens when the command myCat “hello.txt;rm -rf /” (note that the thing in quotes is a single string).
- According to the pseudo-code:
  - strFilename is now “hello.txt;rm -rf /”
  - The command cat hello.txt;rm –rf / is now executed.
  - In actuality it is two commands (separated with ‘;’)
- Now add this with Unix (and Windows) systems not applying principle of least privilege then we have a formula for disaster

Input validation error - preconditions

- Lets take another example, this time with C/C++ code //precondition: iaTemp is a non-null pointer to an array of ints. iLen is the number of elements in iaTemp.

```
void incArray (int* iaTemp, int iLen) {
    for(int i=0;i<iLen;++i) {
        iaTemp[i] = iaTemp[i] + 1;
    }
}
```

- `ia = int array, i = int, str = string`

Input validation error - preconditions

- Preconditions are good, right?
- Any one calling this function should abide by the preconditions right?
  - WRONG! Attackers are successful (most of the time) because they do things that are “unexpected” or beyond the specification
  - They simply don’t, and don’t have to, follow the rules!
  - Of course mistakes are made all the time as well

Input validation error - preconditions

- We need to VALIDATE our preconditions (which includes the assumptions/specifications made on the inputs)
- Add in

```
if (iaTemp == NULL)
  {
    return;
  }
```

Numeric Errors – Example

- A numeric error occurs when you misuse a “type” or “types”
- Example:

```
for (int i = 0; i < strlen(strTemp); ++i)
      {
              printf(“%c”, strTemp[i]);
      }
```

- What is the type of the THING that goes in between [and]? Similarly, what is the type of the THING that is returned by strlen()?
- int? WRONG
- size_t is the right type
  - Follow on question is: size_t = unsigned int?
  - NO! These definitions are library AND platform dependent. REMEMBER OUR ASSUMPTIONS?
  - What if it is unsigned int. Now what?

Numeric Errors – Example Continued

- The previous example is an example of a “type mismatch”
- Now what if size_t = unsigned int?
  - The comparison i < strlen() IS INCORRECT! Since i is “signed” it is only about half as many possible positive values as size_t!
- Due to this, the previous example can also be said to be an example of a “sign mismatch”

Integer Overflows - Example

- Lets say that int and unsigned int are 2 bytes long

```
unsigned int ui = 0xFFFF;
int i = 0x7FFF;
++ui;
++i;
```

- So what is ui?
  - 0x0000//0 in decimal!!!
    There was an overflow!!
- What is `i`?
  - 0x8000//-32768!!!
    There is another overflow, and even worse the sign changed!!!

Integer Overflows - Example Continued

- So what is the big deal?
- It all depends on the code. We can call this a “fault” so if we are prepared for it, then everything is good. If we are not, then things could be bad.
- See some examples in the SAMATE database.
- Ohh this reminds me, what is the opposite of >? (that is the greater than operator)
    - If you answered < you are very wrong and have just introduced a very common numerical error that leads to “off-by-one” errors among other things.
    - Right answer is <=

Integer overflows – summary

- Machine integers can represent only a limited set of numbers, might not correspond to programmer’s mental model
- Program assumes that integer is always positive, overflow will make (signed) integer wrap and become negative, which will violate assumption
  - Program casts large unsigned integer to signed integer
  - Result of a mathematical operation causes overflow
- Attacker can pass values to program that will trigger overflow

Buffer overflows

- Simply, a buffer overflow is an event where more stuff is being written into a buffer than the buffer is meant to hold.

Smashing The Stack For Fun And Profit

- This is a classic (read: somewhat dated) exposition of how buffer overflow attacks work.

- Upshot: if the attacker can write data past the end of an array on the stack, she can usually overwrite things like the saved return address. When the function returns, it will jump to any address of her choosing.

- Targets: programs on a local machine that run with setuid (superuser) privileges, or network daemons on a remote machine

Buffer overflows

- Simply, a buffer overflow is an event where more stuff is being written into a buffer than the buffer is meant to hold.
- Example:

  ```
  char strTemp[10] = “1234567890”;
  ```
- What is the problem? Well lets do some counting.
- We defined the string strTemp to be 10 characters (bytes) long and then we assigned it to a 10 character string. So no problems right?
- No, because “1234567890” is a c-string, which means there is a NULL character at the end. So we are stuffing 11 bytes into a 10 byte buffer.
- This is known as an off-by-one error, but is indeed a buffer overflow

Buffer Overflows Continued

- So then what happens?
- Most of the time, for the example above, absolutely nothing!
- That is because compilers like to “pad” variables to certain boundaries (e.g. 4-byte boundaries).
- This means that the compiler would allocate 12 bytes for that 10 byte buffer. So there is in reality enough room for that extra NULL.
- But what happens when we put in MUCH more than what the buffer can hold? Lets find out.

Lok's Computer 1

- To facilitate this next talk, I will now introduce a new computer so we can readily see what a buffer overflow can result in, AND why these type of errors are so powerful (or devastating if you are on the receiving end)
- Lok's computer has lots of memory that is linear and is separated into bytes (byte-addressing).
  - Address starts at 0x00
  - Address ends at 0x7F (128 bytes total)
  - Data grows up
  - Organization: Data appears first in memory followed by 0xFF and then comes the instructions
    - (See example in a couple slides)

Lok’s Computer 2

- The computer only supports one type: string
  - All strings end with NULL (‘\0’)
  - When defining a string, we can use the [] to specify an initial length.
  - All variable names are pointers (so they contain an address)
- The computer has a very simple instruction set:
  - read VAR. Reads a string from the user and inputs it into the memory space starting at address VAR
    - read has code “0xF1”
  - write VAR. Writes the string starting at memory address VAR to the screen
    - write has code “0xF2”
- The computer runs very simply as well: 1. Find first instruction, take in 2 bytes, first byte is instruction (interpret it) 2nd byte is whatever instruction is expecting. Once complete, move 2 more bytes down memory and process that instruction.

Lok's Computer 3

strHello[12] = "Hello World";

write strHello;

- So how does this look in memory?
  - Lets take first row is main memory
    - Things appearing in quotes are characters, '0' is the character for zero
    - Otherwise the values are in hex, 00 is the hex value zero like NULL
  - Second is the address

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	00	FF	F2	00																															
00															0F	10															1F	20													2D

Lok’s Computer – Example 4

‘H’	‘i’	‘ ’	‘M’	‘y’	00	‘L’	‘o’	‘k’	‘ ’	‘n’	‘a’	‘m’	‘e’	00	‘i’	‘s’	00	FF	F2	00	F2	0A	F2	0F	F2	06																			
00						06				0A					0F	10															1F	20													2D

- So what does the above program do?
  Find the start of the instructions and interpret:
  - We find F2,00 as the first instruction, which is print out the string starting at address 00
  - Then the next instruction is F2,0A which is print out the string starting in address 0A
- So output on the screen is:
  “Hi MynameisLok name”

Lok’s Computer - Example 4 Cont.

- Notice that in the example, the instruction for write (F2) was used in a very strange way.
- F2,0A actually pointed to the MIDDLE of the string!!
- This is a usage that wasn’t expected or at least didn’t seem to have been designed into the computer, but it is indeed possible
- Remember the assumptions!
- Now lets continue with a READ this time.

Example 5

- Lets say we need a program that reads in a name that is less than 10 characters long and says hi.

```
strMessage = “Your Name?”;
strHello = “Hello “;
strName[11]; //one extra character for NULL

write strMessage;
read strName;
write strHello;
write strName;
```

Example 5 Cont.

'Y'	'o'	'u'	'r'	' '	'n'	'a'	'm'	'e'	'?'	00	'H'	'e'	'l'	'l'	'o'	' '	00	00	00	00	00	00	00	00	00	00	00	00	FF	F2	00	F1	12	F2	0B	F2	12								
00						06				0A					0F	10															1F	20													2D

- Above is the before snapshot
- Below is after the user inputs “Lok”

'Y'	'o'	'u'	'r'	' '	'n'	'a'	'm'	'e'	'?'	00	'H'	'e'	'l'	'l'	'o'	' '	00	'L'	'o'	'k'	00	00	00	00	00	00	00	00	FF	F2	00	F1	12	F2	0B	F2	12								
00						06				0A					0F	10															1F	20													2D

Example 5 Cont.

- So! We are talking about buffer overflows right? Lets try to do an overflow and see what happens.
- What will memory look like if the user inputs “BufferOverflowsAreBad”
  So what happens now? The last instruction we processed was F1,12 at address 20. But now

‘Y’	‘o’	‘u’	‘r’	‘ ’	‘n’	‘a’	‘m’	‘e’	‘?’	00	‘H’	‘e’	‘l’	‘l’	‘o’	‘ ’	00	‘B’	‘u’	‘f’	‘f’	‘e’	‘r’	‘O’	‘v’	‘e’	‘r’	‘f’	‘l’	‘o’	‘w’	‘s’	‘A’	‘r’	‘e’	‘B’	‘a’	‘d’	00						
00						06				0A					0F	10															1F	20													2D

- So what happens now? The last instruction we processed was F1,12 at address 20. So our computer here expects the next instruction to be at address 22.
  - The contents of 22 and 23 is now ‘r’ and ‘e’. And ‘r’ is definitely neither F1 nor F2 so?
  - Fault! Computer burns up and we see smoke, and Lok is sad because he paid a lot of money for his computer

Example 5 Cont.

- You thought we were done huh? NOPE.
- We just noticed that we could completely violate the original assumption that the user name was going to be 10 characters long
- We made the computer crash and burn
- We should also notice that if we were slick, we could change our name to some specially crafted string so that instead of ‘r’ and ‘e’ we get a real instruction.

Example 5 Cont.

- So lets do it, what do you think will happen if the following string was inserted as the username? You must pretend that the stuff in [] is actually the character corresponding to the hex value within the []

"BufferOverflowsA"[F2][0E]

- Here is what we will get:
  "Your name?"
  "lo "
- Then the program ends, and everyone is happy

Example 5 Cont.

So what happened?

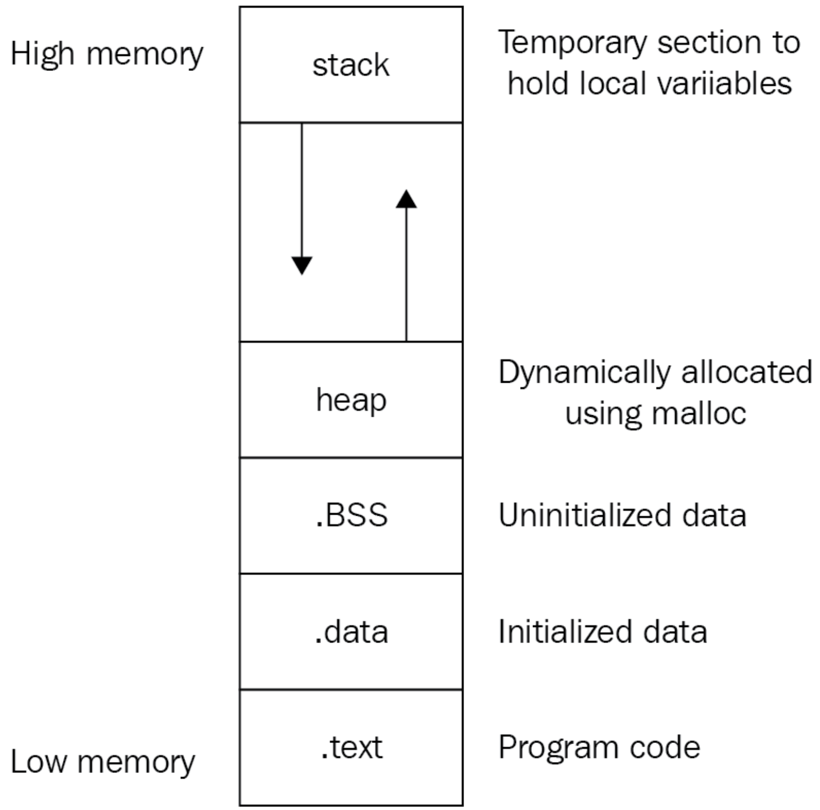
- We made an assumption that the user name was going to be a certain length.
- User was bad and inputted something really long.
- Turns out that it was SO long that it overwrote some instructions
- Lok’s computer just kept going to the next expected instruction
- At first it was junk, but then we got smart and we were able to make Lok’s computer do something the original program didn’t do!
- This last point is the important part! Let’s now move onto some real buffer overflow stuff

Buffer Overflows - Stack

- Now, we will focus on the stack, so stack overflows. The same concepts can be extended, but we don’t talk about those

What is the stack? For x86

- Stack is on top of memory
- It grows down, but buffers go up (strange huh?)
- Stack not only contain local-variables, but also control data, “stack frames” for functions
- The last bullet is why stack overflows can be bad.



Intermission – Function Preamble / Postamble, Finale

- So we need to talk about the function preamble and finale (postamble? That is not a word I know) with respect to C and x86
- The preamble is a setup procedure whenever a function is called, the finale is the takedown

Preamble

When a function is called the following happens

- Function parameters are pushed onto the stack in reverse order
- The function is then called using the CALL instruction - The return address of the next instruction (the one after the function call) is pushed onto the stack. This is known as the return address
- The current stack frame base pointer (where the current frame starts) is pushed onto the stack

Postamble/Finale

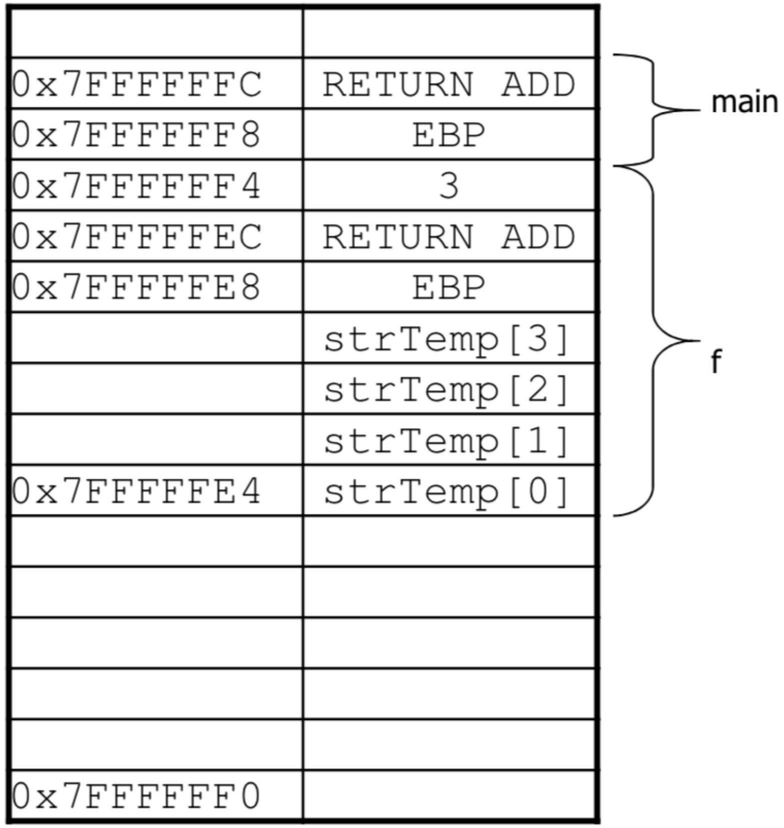
When a function ends the following takes place

- The saved stack frame base pointer is restored to the appropriate register (EBP)
- The Instruction Pointer is set to the saved return address. This way the next instruction will be the one after the function was called.

Stack Frames (Simplified)

```
void f(int a)
{
        char strTemp[4];
}
void main()
{
        f(3);
}
```

- Left column is the address
- Right column is contents
- I made up the addresses, but they are labeled correctly
- RETURN ADD is the saved return address, i.e. the address of the next instruction after the CALL instruction that started the current function
  - The idea is so that when the current function ends, RETURN ADD is the next instruction to run
- EBP holds the base of the previous stack frame (Extended Base Pointer)

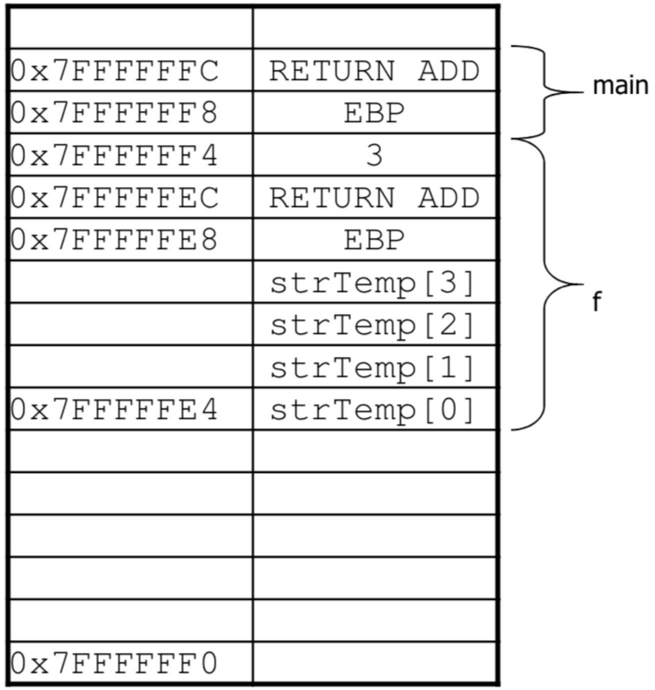


Notice something?

- On Lok’s computer we were able to overwrite the next instruction
- What about in the stack frame from the previous slide?
  - No instructions ☹
  - BUT! There is that return address.. Maybe if we can just overwrite that with another address
  - What other address?
    - Any address
    - What about an address on the stack, like strTemp?

Stack Overflow - Example 6

- So lets now just pretend that strTemp is user inputted.
- What does the user need to input in order to make that first return address point back to the start of strTemp?
- “abcdabcd”[0x7FFFFFE4] is one possibility. There are plenty more.
  - That is interesting...
- What happens when this function ends?
  - The next instruction is now going to be 0x7FFFFFE4.
  - But that is this “abcd” thing, so it will probably just crash. Just like in Lok’s computer.



Example 6 Cont.

- So… if we were slick, then we will replace “abcd” with a valid instruction right?!
- Absolutely. Actually, if we were REALLY slick, we will replace it with a whole SERIES of instructions. Like instructions to open a “root shell”
  - A “root shell” is just a command prompt that has root privileges
  - Other possibilities are opening a port to listen to instructions
  - Anything else really

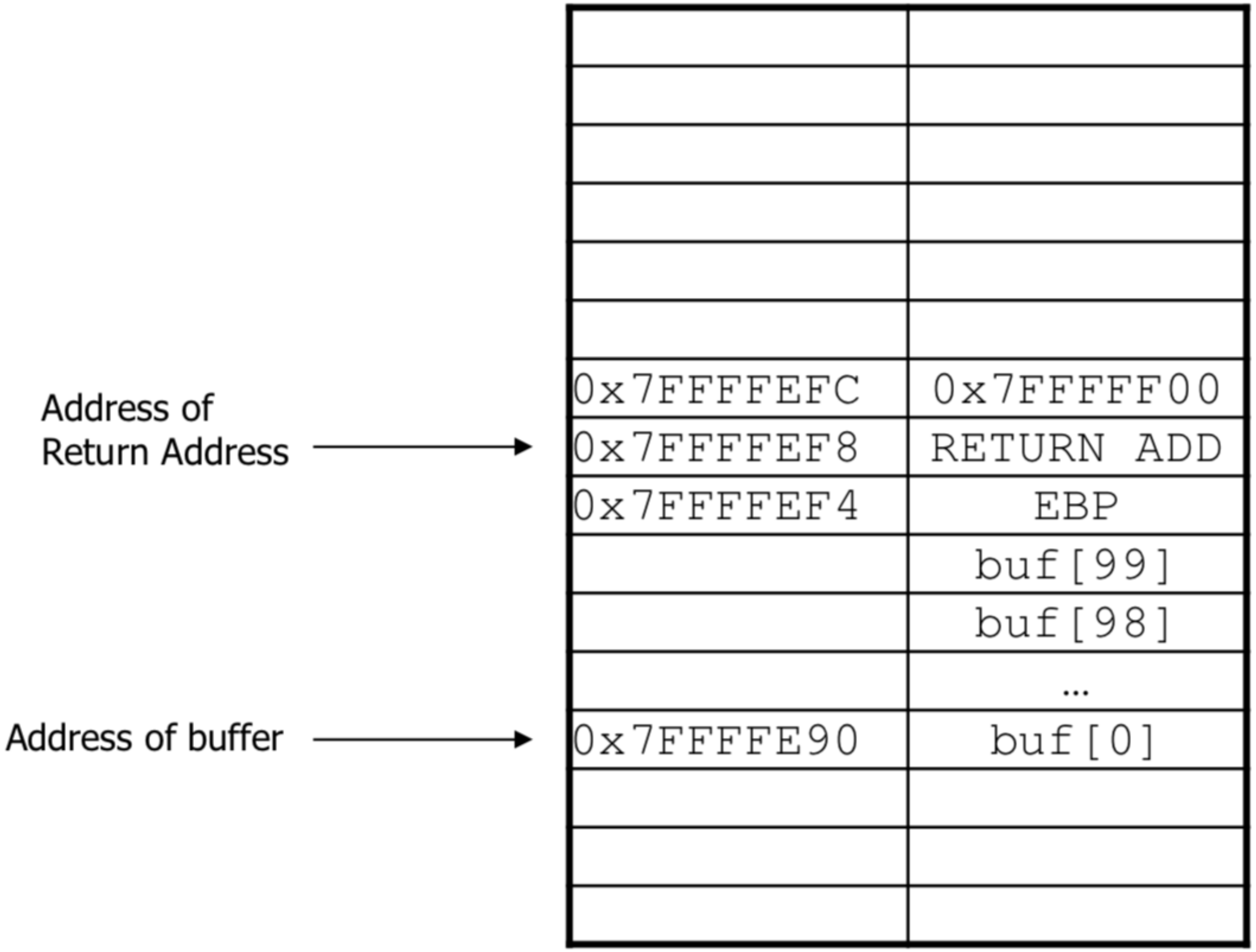
Shellcode

- The code to open a “shell” is known as – SHELLCODE (See http://shell-storm.org/shellcode/ for samples)
- Now that is interesting. But then there are a couple of difficult things.
  - First is how do we even know the address of strTemp in the first place?
  - Second is how do we know where the return address is, so I know how long my string is and also how to align things right?

Address of strTemp

- The address of where your “shellcode” will be is difficult to obtain
  - You might be able to get it through debugging the program
  - You can try trial and error
  - But in general, there are well known ways (we’ll talk about mitigation strategies, one of which is called address space layout randomization)
- Okay, so the start of the “shellcode” is difficult to pin down, but what if there was a way to make the start of the shellcode span a LARGE area?
  - There is. In x86, instruction 0x90 is the NOP instruction, which is exactly 1 byte (which is good) and does absolutely nothing.

Some stack


Address of
Return Address
Address of buffer
0x7FFFFEFC
0x7FFFFF00
0x7FFFFEF8
RETURN ADD
0x7FFFFEF4
EBP
buf[99]
buf[98]
...
0x7FFFFE90
buf[0]

Stack with the buffer filled with 100 bytes of NOPs + SHELLCODE