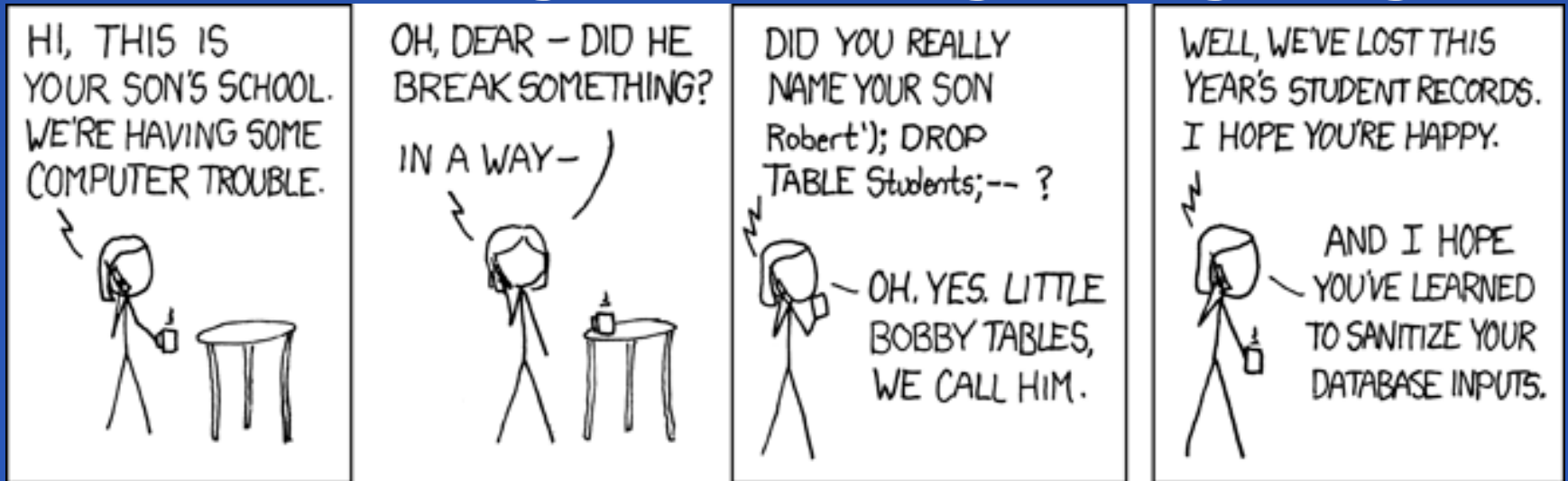


CS 493

Secure Software Systems

Ch 6 Programming languages



XKCD

Dr. Williams

Central Connecticut State University

Objectives

- Common threats in the programming language environment
- The risk of unrestricted user input
- The need to own information where context is understood
- Secure handling of arrays
- Mitigation techniques for common programming language attacks, like buffer overflow
- Risks and best practices in the use of external APIs and libraries

Goals

- Identify common attacks against programming languages.
- Identify mitigation techniques to prevent malicious input.
- Determine the highest risks to the use of an API or library.
- Identify the threats to the most common programming languages.
- Conduct an investigation into the programming languages used in your system.

Programming and Security

- **Programming Securely** To develop code in a secure manner so that the code itself is not a vulnerability that can be exploited by an attacker.
- **Programming Security** To develop code for security-specific functions such as encryption, digital signatures, firewalls, etc.
- In this lecture focus on programming securely:
 - Programming securely – language security and interaction security
 - programming security: security APIs and trust models.

Software Issues

Alice and Bob

- ❑ Find bugs and flaws by accident
- ❑ Hate bad software...
- ❑ ...but must learn to live with it
- ❑ Must make bad software work

Trudy

- Actively looks for bugs and flaws
- Likes bad software...
- ...and tries to make it misbehave
- Attacks systems via bad software

Complexity

- “Complexity is the enemy of security”, Paul Kocher, Cryptography Research, Inc.

System	Lines of Code (LOC)
Netscape	17 million
Space Shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

- A new car contains more LOC than was required to land the Apollo astronauts on the moon

Lines of Code and Bugs

- Conservative estimate: 5 bugs/10,000 LOC
- **Do the math**
 - Typical computer: 3k exe's of 100k LOC each
 - Conservative estimate: 50 bugs/exe
 - So, about 150k bugs per computer
 - So, 30,000-node network has 4.5 billion bugs
 - Maybe only 10% of bugs security-critical and only 10% of those remotely exploitable
 - Then “only” 45 million critical security flaws!

Language Barriers

- **Programming languages** are convenience structures that keep a programmer from needing to speak the native language of the machine; they are written in a combination of variables and near human terms that are similar in meaning to human language.
- Common functions are needed for most programs and are available within libraries for different languages.

Language Barriers

- **Application programming interfaces (APIs)** have been created to allow a system to call existing functionality in another module or system through a specified interface.
- **Compiling** is the process of translating the high-level language into native machine Code.
- An **interpreted language** is one that has a lower layer of machine code that dynamically reads and interprets commands from a higher-level language.

Program Flaws

- An **error** is a programming mistake
 - To err is human
- An error may lead to incorrect state: **fault**
 - A fault is internal to the program
- A fault may lead to a **failure**, where a system departs from its expected behavior
 - A failure is externally observable



Example

```
char array[10];  
for(i = 0; i < 10; ++i)  
    array[i] = 'A';  
array[10] = 'B';
```

- ❑ This program has an **error**
- ❑ This error might cause a **fault**
 - Incorrect internal state
- ❑ If a fault occurs, it might lead to a **failure**
 - ▮ Program behaves incorrectly (external)
- ❑ We use the term **flaw** for all of the above

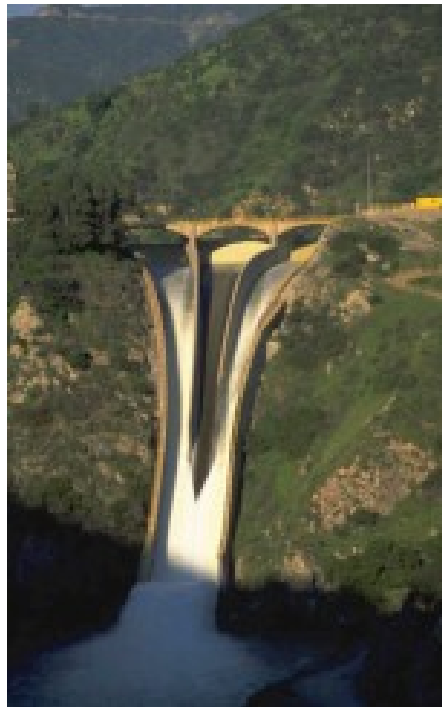
Secure Software

- In software engineering, try to ensure that a program does what is intended
- **Secure** software engineering **requires** that software **does what is intended...**
- **...and nothing more**
- Absolutely secure software is impossible
 - But, absolute security **anywhere** is impossible
- **How can we manage software risks?**

Program Flaws

- Program flaws are **unintentional**
 - But can still create security risks
- We'll consider 2 types of flaws
 - Buffer overflow (smashing the stack)
 - Incomplete mediation
- These are the most common problems

Buffer Overflow



Buffer Bashing

- A **buffer overflow**, in its most general sense, is when more information is written to a location than the location can hold.
- Most buffer overflows in programming result from a lack of constraint on the amount of input that is allowed or the mishandling of pointer variables.

Possible Attack Scenario

- Users enter data into a Web form
- Web form is sent to server
- Server writes data to array called buffer, without checking length of input data
- Data “overflows” buffer
 - Such overflow might enable an attack
 - If so, attack could be carried out by anyone with Internet access

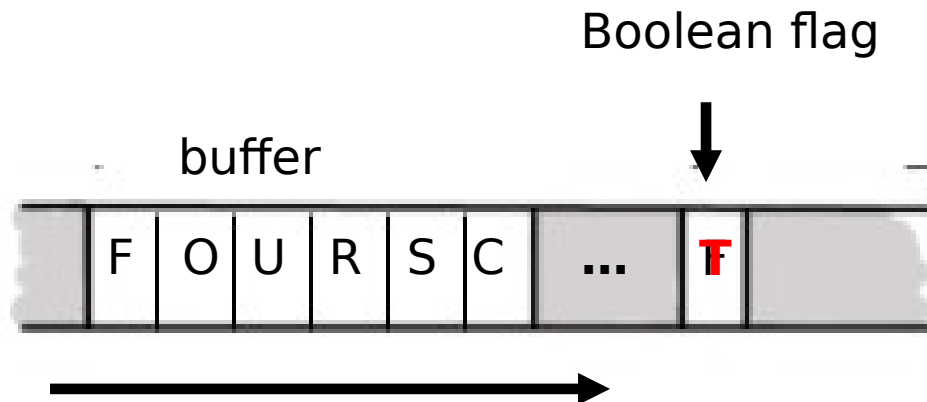
Buffer Overflow

```
int main() {  
    int buffer[10];  
    buffer[20] = 37; }
```

- **Q:** What happens when code is executed?
- **A:** Depending on what resides in memory at location “buffer[20]”
 - Might overwrite **user** data or code
 - Might overwrite **system** data or code
 - Or program could work just fine

Simple Buffer Overflow

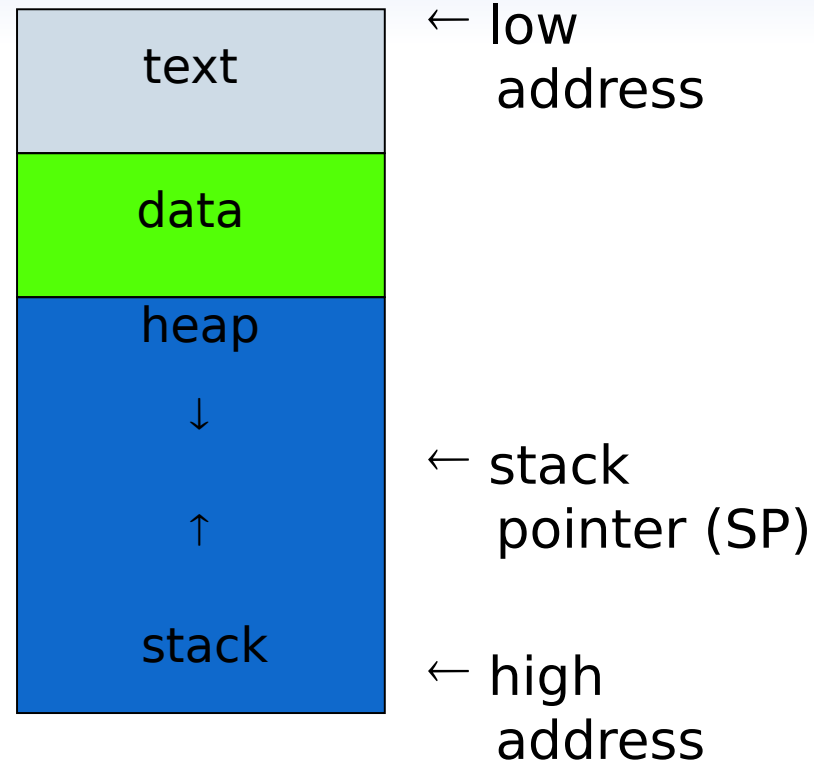
- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate



- ❑ In some cases, Trudy need not be so lucky as in this example

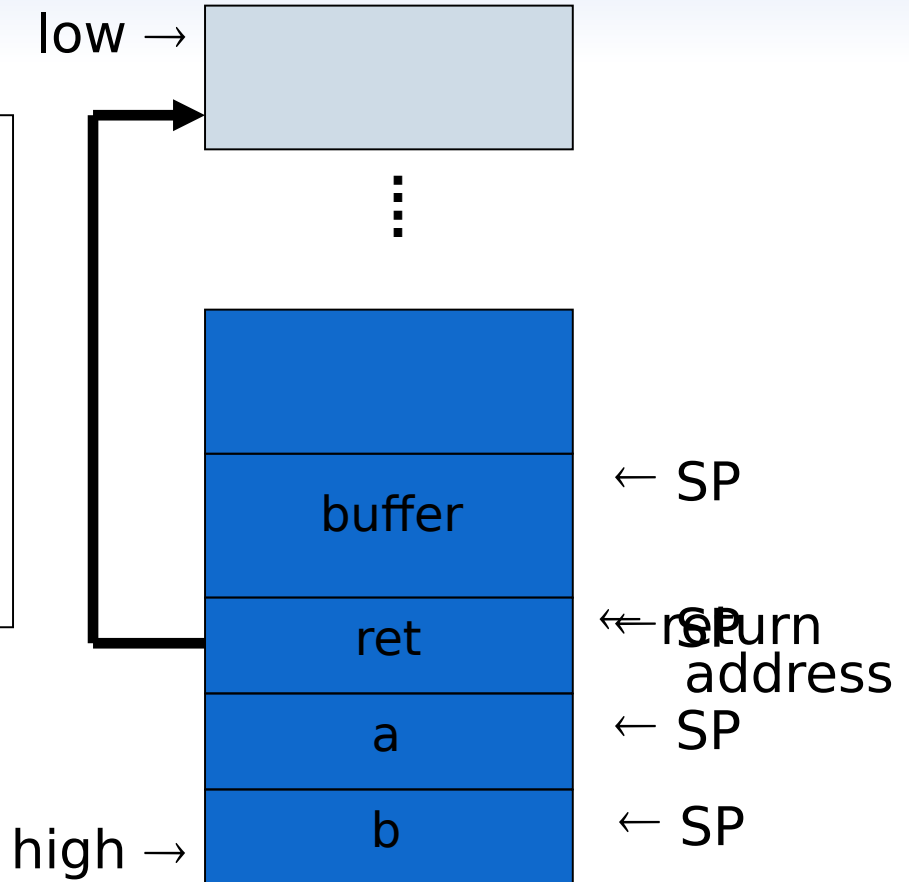
Memory Organization

- **Text** == code
- **Data** == static variables
- **Heap** == dynamic data
- **Stack** == “scratch paper”
 - Dynamic local variables
 - Parameters to functions
 - Return address



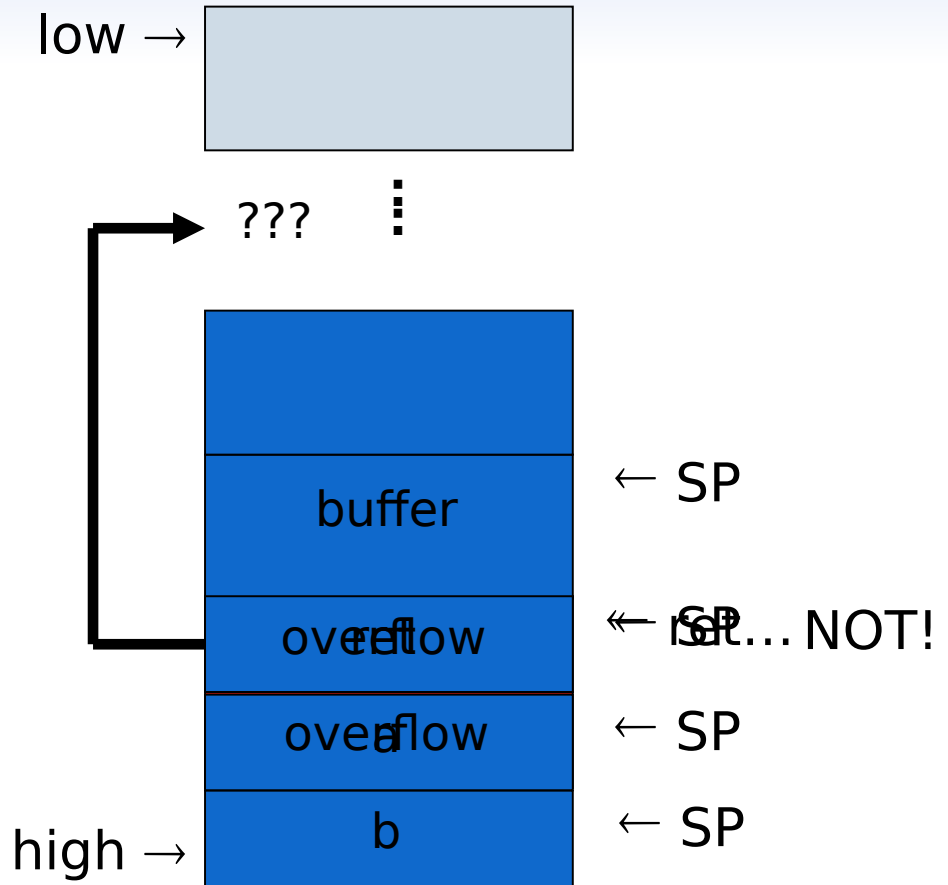
Simplified Stack Example

```
void func(int a, int b){  
    char buffer[10];  
}  
void main(){  
    func(1, 2);  
}
```



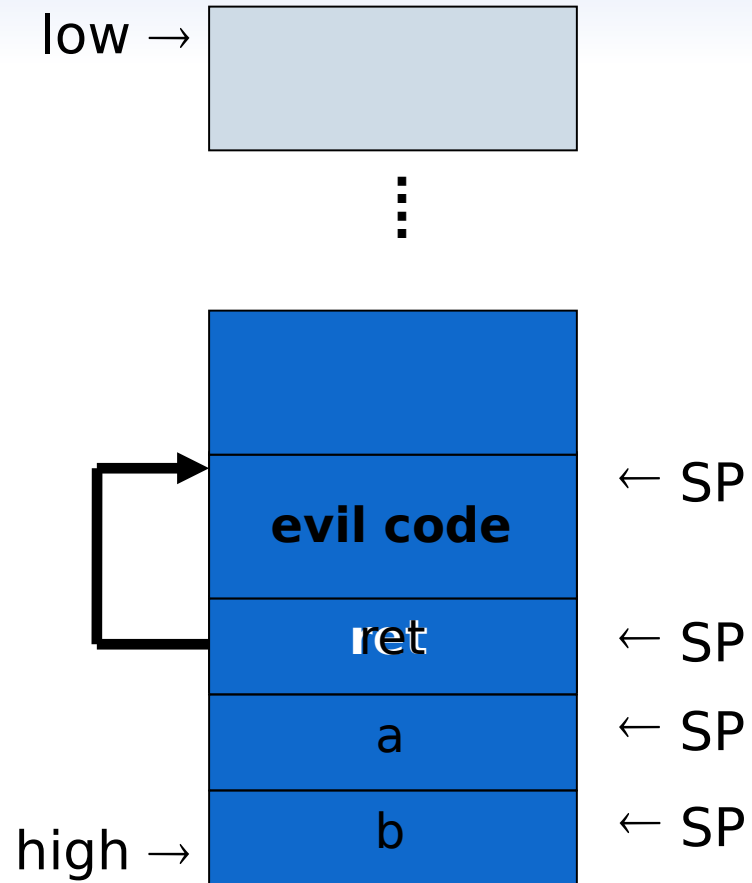
Smashing the Stack

- ❑ What happens if buffer overflows?
- ❑ Program “returns” to wrong location
- ❑ A crash is likely



Smashing the Stack

- ❑ Trudy has a better idea...
- ❑ **Code injection**
- ❑ Trudy can run code of her choosing...
 - ...on your machine

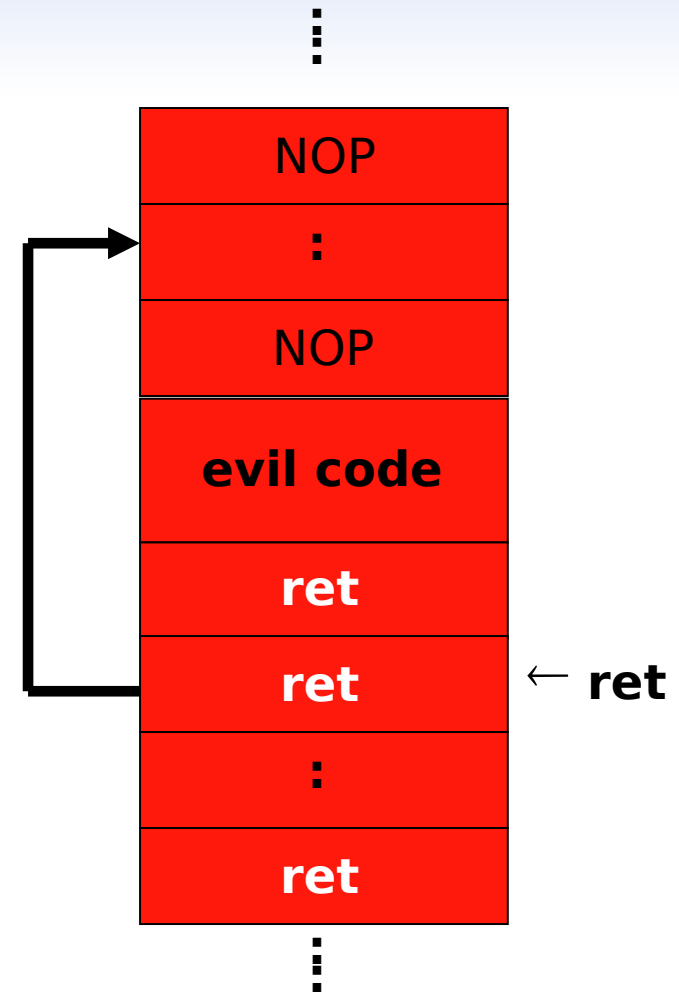


Smashing the Stack

- ❑ Trudy may not know...
 - 1) Address of evil code
 - 2) Location of **ret** on stack

- ❑ Solutions

- 1) Precede evil code with NOP “landing pad”



Stack Smashing Summary

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
 - Things must align properly
- If exploitable, attacker can **inject code**
- Trial and error is likely required
 - Fear not, lots of help is available online
 - Smashing the Stack for Fun and Profit, Aleph One
- Stack smashing is “attack of the decade”
 - Regardless of the current decade
 - Also heap overflow, integer overflow, ...

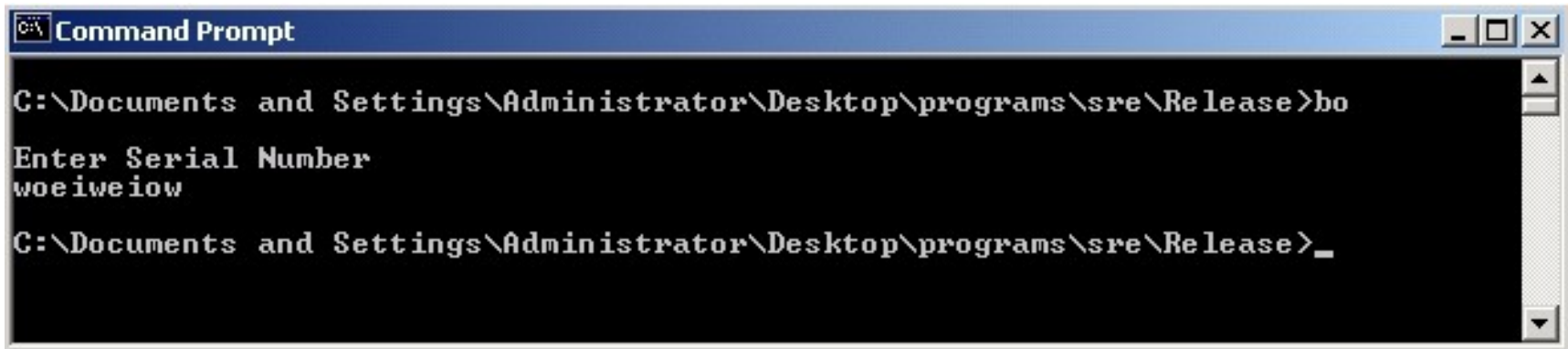
Stack Smashing

- Alice writes the following code to check whether she should allow the user access to her program:

```
main()
{
char in[75];
printf("\nEnter Serial Number\n");
scanf("%s",in);
if (!strncmp(in, "S123N456",8))
{
printf("Serial number is correct.\n");
// Enter program
}
}
```

Stack Smashing Example

- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)

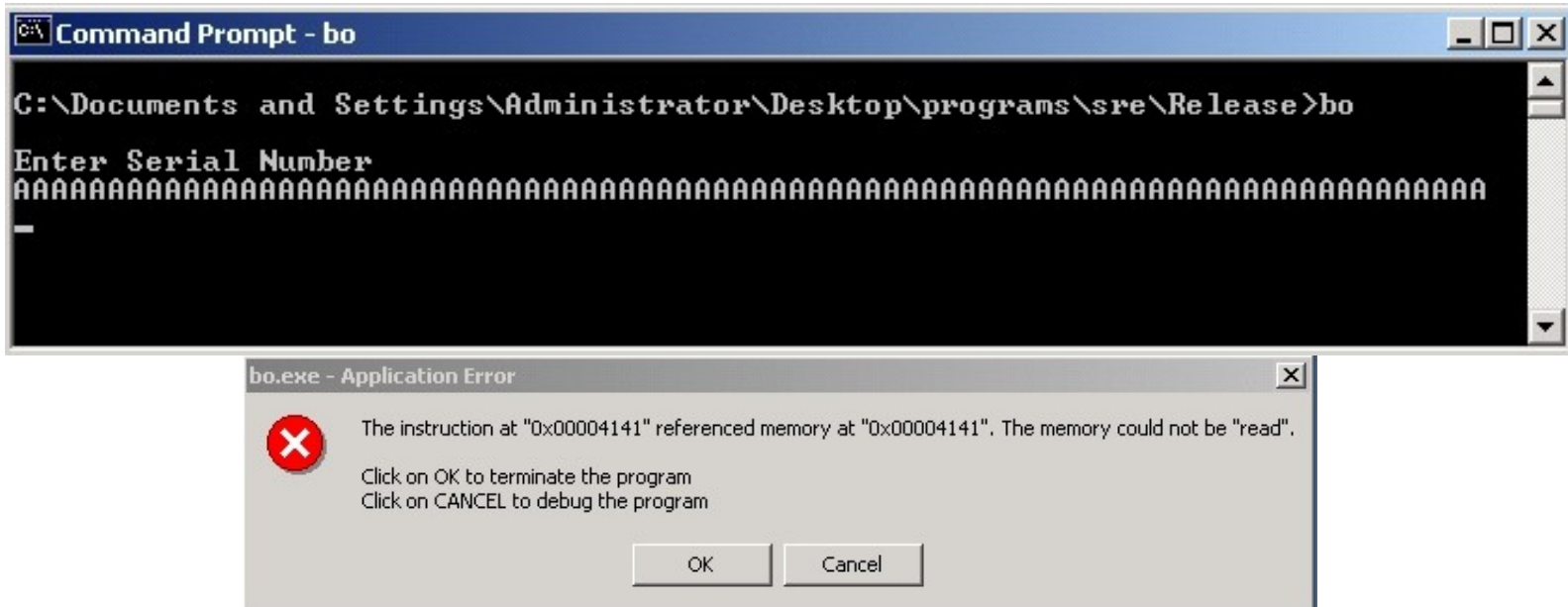


```
Command Prompt
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo
Enter Serial Number
woeiweiw
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

- ❑ Program quits on incorrect serial number

Buffer Overflow Present?

- By trial and error, attacker discovers apparent buffer overflow



- Note that 0x41 is ASCII for "A"
- Looks like **ret** overwritten by 2 bytes!

Disassemble Code

- Next, disassemble bo.exe to find

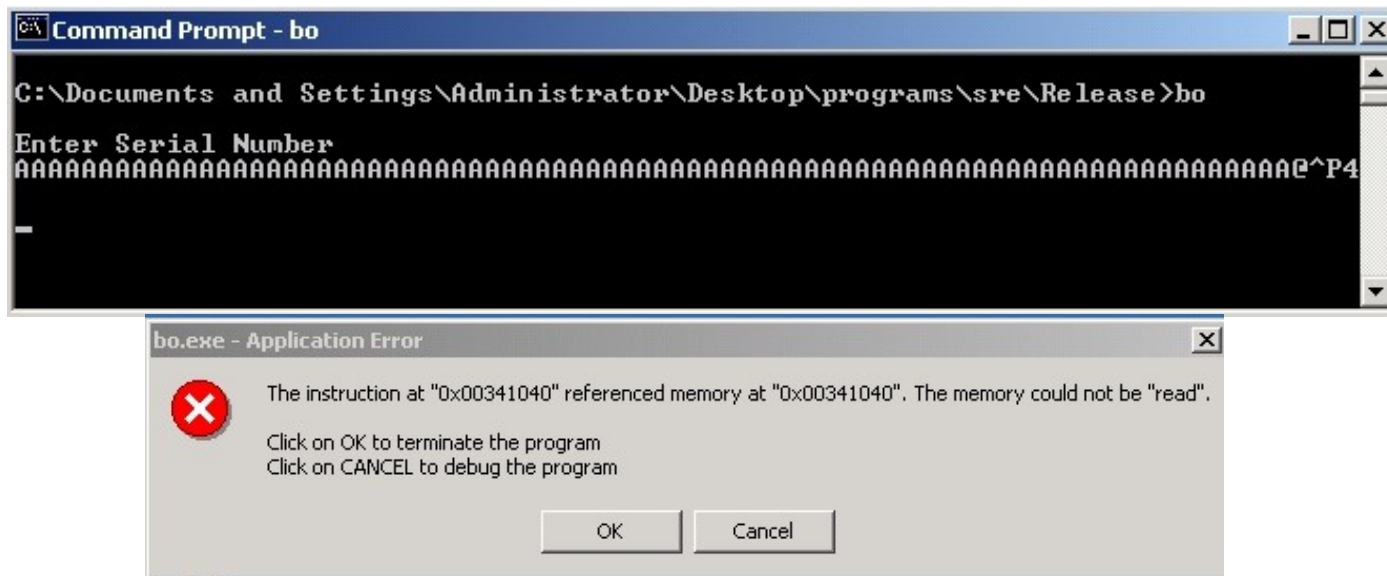
```
.text:00401000
.text:00401000
.text:00401003
.text:00401008
.text:0040100D
.text:00401011
.text:00401012
.text:00401017
.text:0040101C
.text:0040101E
.text:00401022
.text:00401027
.text:00401028
.text:0040102D
.text:00401030
.text:00401032
.text:00401034
.text:00401039
.text:0040103E

sub     esp, 1Ch
push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
call    sub_40109F
lea     eax, [esp+20h+var_1C]
push    eax
push    offset aS                ; "%S"
call    sub_401088
push    8
lea     ecx, [esp+2Ch+var_1C]
push    offset aS123n456 ; "S123N456"
push    ecx
call    sub_401050
add     esp, 18h
test    eax, eax
jnz     short loc_401041
push    offset aSerialNumberIs ; "Serial number is correct.\n"
call    sub_40109F
add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Buffer Overflow Attack

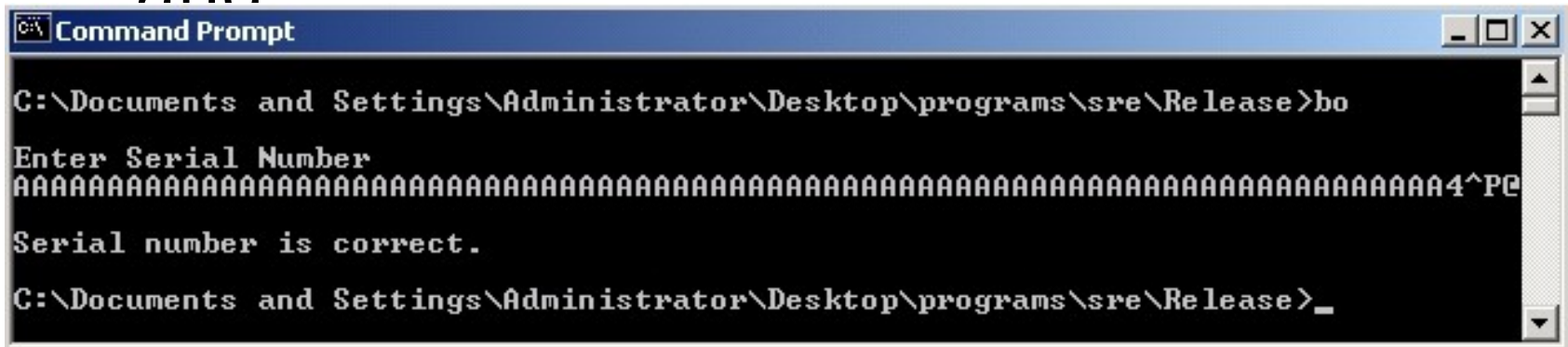
- Find that, in ASCII, 0x401034 is “@^P4”



- ❑ Byte order is reversed? Why?
- ❑ X86 processors are “little-endian”

Overflow Attack, Take 2

- Reverse the byte order to “4^P@”
and



```
Command Prompt
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo
Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4^P@
Serial number is correct.
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

- ❑ Success! We've bypassed serial number check by exploiting a buffer overflow
- ❑ What just happened?
 - Overwrote return address on the stack

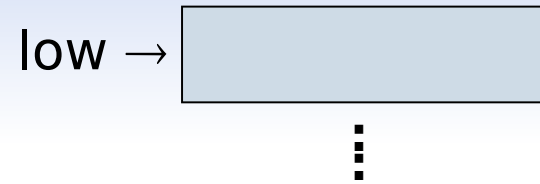
Buffer Overflow

- Attacker did **not** require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Find desired address by trial and error?
 - Necessary if attacker does not have exe
 - For example, a remote attack

Stack Smashing Defenses

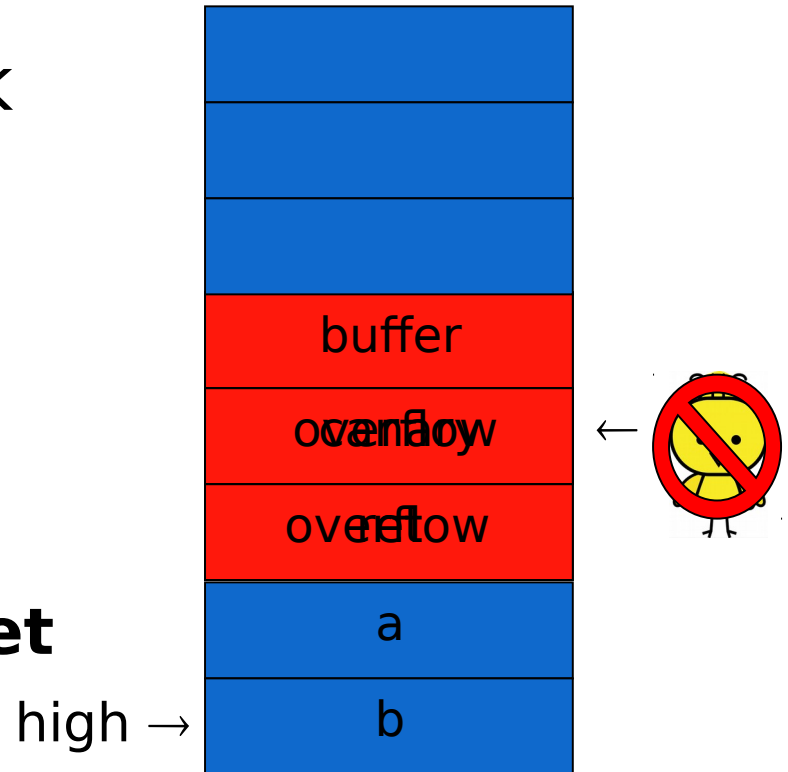
- Employ **non-executable stack**
 - “No execute” **NX bit** (if available)
 - Seems like the logical thing to do, but some real code executes on the stack (Java, for example)
- Use a **canary**
- Address space layout randomization (**ASLR**)
- Use **safe languages** (Java, C#)
- Use **safer C functions**
 - For unsafe functions, safer versions exist
 - For example, strncpy instead of strcpy

Stack Smashing Defenses



- **Canary**

- Run-time stack check
- Push canary onto stack
- Canary value:
 - Constant 0x000aff0d
 - Or may depends on **ret**



Microsoft's Canary

- Microsoft added **buffer security check** feature to C++ with /GS compiler flag
 - Based on canary (or “security cookie”)

Q: What to do when canary dies?

A: Check for user-supplied “handler”

- Handler shown to be subject to attack
 - Claim that attacker can specify handler code
 - If so, formerly “safe” buffer overflows become exploitable when /GS is used!

ASLR

- Address Space Layout Randomization
 - Randomize place where code loaded in memory
- Makes most buffer overflow attacks probabilistic
- Windows Vista uses 256 random layouts
 - So about 1/256 chance buffer overflow works?
- Similar thing in Mac OS X and other OSs
- Attacks against Microsoft's ASLR do exist
 - Possible to “de-randomize”

Buffer Overflow

- A major security threat yesterday, today, and tomorrow
- The good news?
- It is possible to reduced overflow attacks
 - Safe languages, NX bit, ASLR, education, etc.
- The bad news?
- Buffer overflows will exist for a long time
 - Legacy code, bad development practices, etc.

Buffer Bashing

There are defenses against buffer overflow, though none of them are perfect:

- **Array bounding**
- **Pointer handler indirection**
- **Data canaries**
- **Strict read and write limits**

Incomplete Mediation



Input Validation

- Consider: `strcpy(buffer, argv[1])`
- A buffer overflow occurs if
 $\text{len}(\text{buffer}) < \text{len}(\text{argv}[1])$
- Software must **validate** the input by checking the length of `argv[1]`
- Failure to do so is an example of a more general problem: **incomplete mediation**

Good Input

- **Input validation** is a common tool used in developing a system. This asserts that the information entered by a user is of proper format for the system to process.
- **Input validation** at the front end of the application is wonderful for getting legitimate users to line up correctly, but the attacker can circumvent the front end. If the same validation is not done on the back end of the system, your initial input validation is meaningless in terms of attack.

JIT Systems

There are several steadfast rules that should be employed with any system, but most of all with JIT systems:

- **Never execute your input.**
- **Keep a layer between your code and your input.**
- **Map your exceptions.**

Good Output

Output scrubbing is different than input scrubbing because you do not need to worry about format and injection unless you are passing user input through your system into another external component.

- **Encrypt information that is secret.**
- **Include only what is necessary for the external process.**
- **Do not assume trust for the external system.**

Inherent Inheritance and Overdoing Overloads

Two consideration that need to be made when dealing with secure coding:

1. **Inheritance** is the use of a base parent class and defining specific extensions of it.
2. An **overload** is a redefinition of an existing operation for a new class.

The Threatdown

The following list identifies some of the highest-reported vulnerabilities for these languages and what you can do to mitigate their effects.

C: has its main vulnerabilities in susceptibility to buffer overflow and data type mismatch.

C++: One of the main languages used in infrastructure along with C, C++ falls victim to buffer overflow as easily as C. No inherent bound checking occurs in either language. One of the other significant areas of vulnerability for C++ is the boundary conditions on floating point values.

The Threatdown

Java: One of the most popular and powerful languages, Java was designed to have the highest interoperability of any language in existence. The highest risk to Java is the ability to call out external executions and inject OS commands.

C#: is Microsoft's answer to Java. It exists on the .NET platform from Microsoft, which insulates it from most of the usual suspects when it comes to exploits. The most significant risk in using any .NET application is specifying an unsafe block of code.

The Threatdown

Visual Basic (VB): Although it has lost out for most thick client development, VB has found a new life in Active Server Pages (ASP). Remote code execution is the big item on the list for VB.

PERL: PERL is often called the “Duct Tape of the Internet” because of its unequaled power and ability to compile on demand within a server environment. It is vulnerable to command injection.

The Threatdown

- **PHP:** PHP is another common language used for web applications. In particular, sending malicious-form data could actually corrupt the internal data of a PHP application and run arbitrary code or simply cause a system crash.

Deployment Issues

- There is generally no way to get rid of the system calls that allow for compromise in the myriad network of libraries and system function calls that have been constructed.
- If you are using any portion of an external system, make sure it is running the latest version with all of the security patches in place.
- However, too much information given in an error log on a deployed system can let attackers know just what state they need to trigger for a compromise.

Web security

- Intersection of lots of user input, JIT languages, and interaction with OS and DB environments

Web security: server-side threats

- **Access control:** should prevent certain files being served.
- Complex or malicious URLs
- Denial of service attacks
- Remote authoring and administration tools
- Buggy servers, with attendant security risks
- Server-side scripting languages: C or shell CGI, PHP, ASP, JSP, Python, Ruby, all have serious security implications in configuration and execution. File systems and permissions have to be carefully designed. *That's before any implemented web application is even considered. . .*

Web programming: application security

Many issues

- **Input validation:** to prevent SQL injection, command injection, other confidentiality attacks.
- **Ajax:** beware client-side validation! Understand metacharacters at every point. Use labels/indexes for hidden values, not values themselves.
- **Output filtering:** Beware passing informative error messages.
- **Careful cryptography:** encryption/hashing to protect server state in client, use of appropriate authentication mechanisms for web accounts

Cross site scripting (XSS)

- Inserting code to be run on target server or pages returned by target server
- Common way unprotected database inserts
- Steal cookies, key logging, passwords, credit card, phishing , etc
- See code demo
 - Simple test
 - Key logging
 - Steal authentication cookie...and so much worse

SQL injection

- Inserting SQL to be run in existing SQL calls to database
- Common way unprotected database selects, inserts, updates, deletes
- Insert/update/delete records, potentially drop tables
- Return information you shouldn't be able to access
- See code demo

Summary

- Programming languages are tools, and they can be used well or misused just like any other tool.
- Knowing what you are using will help you avoid the pitfalls that can allow system compromise.
- Scrubbing user input wherever it exists is the main item that should be done for any programming language in any platform.

In class exercises revisited

Bob's Pizza Shack

In groups consider this scenario and identify in general what places you have programming language related security concerns

- Bob is a small business owner of Bob's Pizza Shack and wants to create a website to allow online credit card delivery orders

Alice's Online Bank

In groups consider this scenario and identify in general what places you have programming language related security concerns

- Alice opens Alice's Online Bank (AOB)
- What are Alice's security concerns from a programming perspective?
 - Consider ones in own environment
 - Consider ones interoperating with another bank
 - Consider ones interacting with customer
 - Web
 - Mobile app

Location based social media app

In groups consider this scenario and identify in general what places you have programming language related security concerns

- Open source group wants to create a mobile app to allow groups to communicate/find each other in public demonstrations/protests
 - Communication internet, as well as, P2P (WiFi/Bluetooth) in case internet cut off – so if person you want to contact is on other side of crowd and no internet as long as P2P network can be established with app can reach somebody outside your immediate vicinity via the P2P network
 - Should be able to communicate securely messages and images to people you identify within your group (group as whole or direct)
 - Should be able to share GPS location with people in group (group as whole or direct)
- Broader scope – Who are potential threats and associated ways could attack/weaken system?