Week 3: Code Injection
Part 1

Program Hijacking
Top Software Weaknesses for 2020

MITRE, a non-profit organization that manages federally-funded research & development centers, publishes a list of top security weaknesses

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
<td>46.81</td>
</tr>
<tr>
<td>2</td>
<td>Out-of-bounds Write</td>
<td>46.17</td>
</tr>
<tr>
<td>3</td>
<td>Improper Input Validation</td>
<td>33.47</td>
</tr>
<tr>
<td>4</td>
<td>Out-of-bounds Read</td>
<td>26.50</td>
</tr>
<tr>
<td>5</td>
<td>Improper Restriction of Operations within the Bounds of a Memory Buffer</td>
<td>23.73</td>
</tr>
<tr>
<td>6</td>
<td>SQL Injection</td>
<td>20.69</td>
</tr>
<tr>
<td>7</td>
<td>Exposure of Sensitive Information to an Unauthorized Actor</td>
<td>19.16</td>
</tr>
<tr>
<td>8</td>
<td>Use After Free</td>
<td>18.87</td>
</tr>
<tr>
<td>9</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>17.29</td>
</tr>
<tr>
<td>10</td>
<td>OS Command injection</td>
<td>16.44</td>
</tr>
</tbody>
</table>

Hijacking & Injection

Hijacking
Getting software to do something different from what the user or developer expected

• **Session hijacking**
  – Take over someone’s communication session (typically from a web browser)
    • Usually involves stealing a session token that identifies the user and authorizes access

• **Program hijacking**
  – Get a program to execute unintended operations
    – **Command injection**
      • Send commands to a program that are then executed by the system shell
      • Include SQL injection – send database commands
    – **Code injection**
      • Inject code into a program that is then executed by the application
      • Can be used for command injection by running system commands
Examples of Hijacking

• **Session hijacking**
  – Snoop on a communication session to get authentication info and take control of the session

• **Code injection**
  – Overflow input and cause new code to run
  – Provide JavaScript as input that will later get executed (Cross-site scripting)

• **Command hijacking**
  – Provide input that will get interpreted as a system command
  – Change search paths to load different libraries or have different programs run

• **Other forms**
  – Redirect web browser to a malicious site
  – Change DNS (IP address lookup) results
  – Change search engine
Security-Sensitive Programs

• Control hijacking isn’t interesting for regular programs on your system
  – You might as well run commands from the shell

• It is interesting if the program
  – Has escalated privileges (setuid), especially root
  – Runs on a system you don’t have access to (most servers)

Privileged programs are more sensitive & more useful targets
Bugs and mistakes

• Most attacks are due to
  – Social engineering: getting a legitimate user to do something
  – Or bugs: using a program in a way it was not intended
  – Bugs include buggy security policies

• Attacked system may be further weakened because of poor access control rules
  – Violate Principle of Least Privilege

• Cryptography won’t help us!
  – And cryptographic software can also be buggy
Unchecked Assumptions

• Unchecked assumptions can lead to vulnerabilities
  – **Vulnerability**: weakness that can be exploited to perform unauthorized actions

• Attack
  – Discover assumptions
  – Craft an **exploit** to render them invalid … and run the exploit

• **Three common assumptions**
  1. Buffer is large enough for the data
  2. Integer overflow doesn’t exist
  3. User input will never be processed as a command
Buffer Overflow
What is a buffer overflow?

• Programming error that allows more data to be stored in an array than there is space

• Buffer = stack, heap, or static data

• **Overflow** means adjacent memory will be overwritten
  – Program data can be modified
  – New code can be injected
  – Unexpected transfer of control can be launched
Buffer overflows

- Buffer overflows used to be responsible for up to ~50% of vulnerabilities

- We know how to defend ourselves but
  - Average time to patch a bug >> 1 year
  - People delay updating systems … or refuse to
  - Embedded systems often never get patched
    - Routers, cable modems, set-top boxes, access points, IP phones, and security cameras
  - We will continue to write buggy code!
Buffer overflows … still going strong

• July 28, 2020 – SIGRed vulnerability
  – Exploits buffer overflow in Windows DNS Server processing of SIG records
  – Allows an attacker to create a denial-of-service attack (& maybe get admin access)
  – Bug existed for 17 years – discovered in 2020!
    • A function expects 16-bit integers to be passed to it
    • If they are not the proper size, it will overflow other integers
    • Attacker needs to create a DNS response that contains a SIG record > 64KB

Another 17 year-old bug

- **March 4, 2020: Point-to-Point Protocol Daemon**
  - pppd is used for layer 2 (data link) services that include DSL and VPNs
  - Bug existed for 17 years – discovered in 2020!
    - Attacker creates a specially-crafted Extensible Authentication Protocol (EAP) message
    - Incorrect bounds check allows copying an arbitrary length of data

GRUB2 Bootloader

• July 29, 2020: GRUB2 bootloader
  – Used by most Linux systems and many hypervisors and Windows systems that use Secure Boot with the standard Microsoft Third Party UEFI Certificate Authority
  – Vulnerability allows attackers to gain arbitrary code execution during the boot process – even when Secure Boot is enabled
  – Attacker needs to modify the GRUB2 config file
    • But this allows the attack to persist and launch new attacks even before the operating system boots
    – GRUB2 checks a buffer size for a token
      • But does not quit if the token is too large

Exim Mail Server Vulnerability

• September 28, 2019: Exim server
  – Heap-based buffer overflow vulnerability in Exim email
  – Exim mail transfer agent used on 5 million systems
  – Remote code execution possible because of a bug in `string_vformat()` found in `string.c`
  – Length of the string was not properly accounted for
WhatsApp vulnerability exploited to infect phones with Israeli spyware

Attacks used app's call function. Targets didn't have to answer to be infected.

DAN GOODIN - 5/13/2019, 10:00 PM

Attackers have been exploiting a vulnerability in WhatsApp that allowed them to infect phones with advanced spyware made by Israeli developer NSO Group, the Financial Times reported on Monday, citing the company and a spyware technology dealer.

A representative of WhatsApp, which is used by 1.5 billion people, told Ars that company researchers discovered the vulnerability earlier this month while they were making security improvements. CVE-2019-3568, as the vulnerability has been indexed, is a buffer overflow vulnerability in the WhatsApp VOIP stack that allows remote code execution when specially crafted series of SRTCP packets are sent to a target phone number, according to this advisory.
WhatsApp messaging app could install malware on Android, iOS, Windows, & Tizen operating systems

An attacker did not have to get the user to do anything: the attacker just places a WhatsApp voice call to the victim.

This was a zero-day vulnerability
- Attackers found & exploited the bug before the company could patch it

WhatsApp used by 1.5 billion people
- Vulnerability discovered in May 2019 while developers were making security improvements

Many, many more!

303 reported buffer overflow vulnerabilities in 2020 (so far)

683 reported buffer overflow vulnerabilities in 2019

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=%22buffer+overflow%22
A few years earlier...

- **Mar 2018: Exim mailer**
  - (affects ~400,000 Linux/BSD email servers)
  - Buffer overflow risks remote code execution attacks
  - base64 decode function

- **Mar 2018: os.symlink() method in Python on Windows**
  - Attacker can influence where the links are created & privilege escalation

- **May 2018: FTPShell**
  - Attacker can exploit this to execute arbitrary code or a denial of service

- **Jun 2018: Firefox fixes critical buffer overflow**
  - Malicious SVG image file can trigger a buffer overflow in the Skia library (open-source graphics library)

- **Sep 2018: Microsoft Jet Database Engine**
  - Attacker can exploit this to execute arbitrary code or a denial of service

- **Jul 2019: VideoLAN VLC media player**
  - Heap-based buffer overflow vulnerability disclosed
And a year before that…

- **Mar 2017: Google Nest Camera**
  - Buffer overflow when setting the SSID parameter
- **May 2017: Skype**
  - Remote zero-day stack buffer vulnerability
  - Could be exploited by a remote attacker to execute malicious code
- **Dec 2017: Intel Management Engine**
  - Coprocessor that powers Intel’s vPro admin features
  - Has its own OS (MINIX 3)
  - A computer that monitors your computer” – with full access to system hardware
- **Oct 2017: Windows DNS Client**
  - Malicious DNS response can enable arbitrary code execution
- **June 2017: IBM's DB2 database**
  - Allows a local user to overwrite DB2 files or cause a denial of service
  - Affects Windows, Linux, and Windows implementations
- **June 2017: Avast Antivirus**
  - Remote stack buffer overflow based on parsing magic numbers in files
  - Can exploit remotely by sending someone email with a corrupted file

http://www.vulnerability-db.com/?q=articles/2017/05/28/stack-buffer-overflow-zero-day-vulnerability-uncovered-microsoft-skype-v72-v735
https://www.theregister.co.uk/2017/12/06/intel_management_engine_pwned_by_buffer_overflow/
Buggy libraries can affect a lot of code bases

July 2017 – Devil's Ivy (CVE-2017-9765)

- gsoap open source toolkit
- Enables remote attacker to execute arbitrary code
- Discovered during the analysis of an internet-connected security camera

gets.c from OS X: © 1990,1992 The Regents of the University of California.

gets(buf)
char *buf;

    register char *s;
    static int warned;
    static char w[] = "warning: this program uses gets(), which is unsafe.\r\n";

    if (!warned) {
        (void) write(STDERR_FILENO, w, sizeof(w) - 1);
        warned = 1;
    }

    for (s = buf; (c = getchar()) != '
';)
        if (c == EOF)
            if (s == buf)
                return (NULL);
            else
                break;
        else
            *s++ = c;

    return (buf);
... char name[128]; /* user’s name */ ...

printf("enter your name: ");
if (gets(name) != NULL) {
    printf("your name is \"%s\"\n", name);
The classic buffer overflow bug

gets.c from OS X: © 1990,1992 The Regents of the University of California.

gets(buf)
char *buf;
    register char *s;
    static int warned;
    static char w[] = "warning: this program uses gets(), which is unsafe.\r\n";

    if (!warned) {
        (void) write(STDERR_FILENO, w, sizeof(w) - 1);
        warned = 1;
    }

    for (s = buf; (c = getchar()) != '\n';)
        if (c == EOF)
            if (s == buf)
                return (NULL);
            else
                break;
        else
            *s++ = c;

    *s = 0;
    return (buf);
```c
gets(buf)
char *buf;

register char *s;
static int warned;

static char w[] = "warning: this program uses gets(),
which is unsafe."

if (!warned) {
    (void) write(STDERR_FILENO, w, sizeof(w) - 1);
    warned = 1;
}

for (s = buf; (c = getchar()) != '\n';)
    if (c == EOF)
        if (s == buf)
            return (NULL);
        else
            break;
    else
        *s++ = c;

*s = 0;
return (buf);
```
void test(void) {
    char name[10];
    strcpy(name, "krzyzanowski");
}

That’s easy to spot!
Another example

How about this?

```c
char configfile[256];
char *base = getenv("BASEDIR");

if (base != NULL)
    sprintf(configfile, "%s/config.txt", base);
else {
    fprintf(stderr, "BASEDIR not set\n");
}
```
Buffer overflow attacks

To exploit a buffer overflow

• Identify overflow vulnerability in a program
  – Black box testing
    • Trial and error
    • Fuzzing tools (more on that …)
  – Inspection
    • Study the source
    • Trace program execution

• Understand where the buffer is in memory and whether there is potential for corrupting surrounding data
Execute arbitrary code, such as starting a shell

*Code injection, stack smashing*

- Code runs with the privileges of the program
  - If the program is *setuid root* then you have root privileges
  - If the program is on a server, you can run code on that server

- **Even if you cannot execute code…**
  - You may crash the program or change how it behaves
  - Modify data
  - Denial of service attack

- **Sometimes the crashed code can leave a core dump**
  - You can access that and grab data the program had in memory
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char pass[5];
    int correct = 0;

    printf("enter password: ");
    gets(pass);
    if (strcmp(pass, "test") == 0) {
        printf("password is correct\n");
        correct = 1;
    }
    if (correct) {
        printf("authorized: running with root privileges...\n");
        exit(0);
    }
    else
        printf("sorry - exiting\n");
    exit(1);
}
```

$ ./buf
enter password: abcdefghijklmnop
authorized: running with root privileges...

Run on iLab system:
CentOS Linux 7 (3.10)
X86-64: i7-7700 CPU @ 3.60GHz
It’s a bounds checking problem

• C and C++
  – Allow direct access to memory
  – Do not check array bounds
  – Functions often do not even know array bounds
    • They just get passed a pointer to the start of an array

• This is not a problem with strongly typed languages
  – Java, C#, Python, etc. check sizes of structures

• But C is in the top 4 of popular programming languages
  – Dominant for system programming & embedded systems
  – And most compilers, interpreters, and libraries are written in C
Part 2

Anatomy of overflows
Linux process memory map*

- **OS**
- **High memory**
  - Command-line args & environment variables
  - Stack
  - Shared libraries
  - Heap
  - Uninitialized data (bss)
  - Initialized data
  - Program (text)
  - Unused

*Not to scale
The stack

Note: rbp & rsp are used in 64-bit processors
ebp & esp are used in 32-bit processors

<table>
<thead>
<tr>
<th>High memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous return address</td>
</tr>
<tr>
<td>Previous frame pointer</td>
</tr>
<tr>
<td>param_3</td>
</tr>
<tr>
<td>param_2</td>
</tr>
<tr>
<td>param_1</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Saved rbp (frame pointer)</td>
</tr>
<tr>
<td>Local variable a</td>
</tr>
<tr>
<td>Local variable b</td>
</tr>
<tr>
<td>Local variable c</td>
</tr>
<tr>
<td>Low memory</td>
</tr>
</tbody>
</table>

func(param_1, param_2, param_3)

Calling function:
pushl param_3
pushl param_2
pushl param_1
call func
...

func:
pushl rbp
movl %rsp, %rbp
subl $20, %rsp
...
leave
ret

Note: rbp & rsp are used in 64-bit processors
ebp & esp are used in 32-bit processors
Causing overflow

Overflow can occur when programs do not validate the length of data being written to a buffer

This could be in your code or one of several “unsafe” libraries

- strcpy(char *dest, const char *src);
- strcat(char *dest, const char *src);
- gets(char *s);
- scanf(const char *format, …)
- Others…
void func(char *s) {
    char buf[128];
    strcpy(buf, s);
    /* ... */
}

What if `strlen(s)` is >127 bytes?
Overflowing the buffer

```c
void func(char *s) {
    char buf[128];
    strcpy(buf, s);
    /* ... */
}
```

What if `strlen(s)` is >127 bytes?
You overwrite the saved `rbp` and then the `return address`
Overwriting the return address

• If we overwrite the return address
  – We change what the program executes when it returns from the function

• “Benign” overflow
  – Overflow with garbage data
  – Chances are that the return address will be invalid
  – Program will die with a SEGFAULT
  – Availability attack
Programming at the machine level

- **High level languages (even C) constrain you in**
  - Access to variables (local vs. global)
  - Control flows in predictable ways
    - Loops, function entry/exit, exceptions

- **At the machine code level**
  - No restriction on where you can jump
    - Jump to the middle of a function … or to the middle of a C statement
    - Returns will go to whatever address is on the stack
    - Unused code can be executed (e.g., library functions you don’t use)
Subverting control flow

Malicious overflow

- Fill the buffer with malicious code
- Overflow to overwrite saved `%rbp`
- Then overwrite saved the `%rsp` (return address) with the address of the malicious code in the buffer
If you want to inject a lot of code

Just go further down the stack (into higher memory)

- Initial parts of the buffer will be garbage data … we just need to fill the buffer
- Then we have the new return address
- Then we have malicious code
- The return address points to the malicious code

MALICIOUS CODE

... still part of the overflow of buf[128]

Previous return address
Previous frame pointer
params
Overwritten return address
char buf[128]
Junk … we don’t care what goes here – we just need to overflow this buffer

Start of buf[128]
Address Uncertainty

What if we’re not sure what the exact address of our injected code is?

**NOP Slide = landing zone**

- Pre-pad the code with a lots of NOP instructions
  - NOP
  - moving a register to itself
  - adding 0
  - etc.
- Set the return address on the stack to any address within the landing zone
Off-by-one overflows
Safe functions aren’t always safe

- **Safe counterparts require a count**
  - `strcpy` → `strncpy`
  - `strcat` → `strncat`
  - `sprintf` → `snprintf`

- But programmers can miscount!

```c
char buf[512];
int i;
for (i=0; i<=512; i++)
    buf[i] = stuff[i];
```
Off-by-one errors

• We can’t overwrite the return address

• But we can overwrite one byte of the saved frame pointer
  – Least significant byte on Intel/ARM systems
    • Little-endian architecture

• What’s the harm of overwriting the frame pointer?
At the end of a function:

- The compiler resets the stack pointer (%rsp) to the base of the frame (%rbp):

\[
\text{mov } %\text{rsp}, %\text{rbp}
\]

- and restores the saved frame pointer (which we corrupted) from the top of the stack:

\[
\text{pop } %\text{rbp} \quad \text{pops corrupted frame pointer into } rbp, \text{ the frame pointer}
\]

The program now has the wrong frame pointer when the function returns.

The function returns normally – we could not overwrite the return address.

BUT … when the function that called it tries to return, it will update the stack pointer to what it thinks was the valid base pointer and return there:

\[
\text{mov } %\text{rsp}, %\text{rbp} \quad \text{rbp is our corrupted one}
\]

\[
\text{pop } %\text{rbp} \quad \text{we don’t care about the base pointer}
\]

\[
\text{ret} \quad \text{return pops the stack from our buffer, so we can jump anywhere}
\]
Off-by-one errors: frame pointer mangling

- **Stuff the buffer with**
  - Malicious code, pointed to by ”saved” %rip
  - ”saved” %rbp (can be garbage)
  - ”saved” %rip (return address)
  - Malicious code, pointed to by ”saved” %rip

- **When the function’s calling function returns**
  - It will return to the “saved” %rip, which points to malicious code in the buffer
Heap & text overflows
Linux process memory map

- Statically allocated variables & dynamically allocated memory (malloc) are not on the stack

- Heap data & static data do not contain return addresses
  - No ability to overwrite a return address

Are we safe?
We may be able to overflow a buffer and overwrite other variables in *higher* memory.

For example, overwrite a file name.

The program:
```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char a[15];
char b[15];

int main(int argc, char **argv)
{
    strcpy(b, "abcdefghijklmnopqrstuvwxyz");
    printf("a=%s\n", a);
    printf("b=%s\n", b);
    exit(0);
}
```

The output:
```
a=qrstuvwxyz
b=abcdefghijklmnopqrstuvwxyz
```

( Linux 4.4.0-59, gcc 5.4.0 )
Memory overflow

The program

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char afile[20];
char mybuf[15];

int main(int argc, char **argv)
{
    strncpy(afile, "/etc/secret.txt", 20);
    printf("Planning to write to %s\n", afile);
    strcpy(mybuf, "abcdefghijklmnop/home/paul/writehere.txt");
    printf("About to open afile=%s\n", afile);
    exit(0);
}
```

We overwrote the file name `afile` by writing too much into `mybuf`!

mybuf can overflow into afile

The output

(\text{Linux 4.4.0-59, gcc 5.4.0})

Planning to write to /etc/secret.txt
About to open afile=/home/paul/writehere.txt
• Even if a buffer overflow does not touch the stack, it can modify global or static variables

• Example:
  – Overwrite a function pointer
  – Function pointers are often used in callbacks

```c
int callback(const char* msg)
{
    printf("callback called: %s\n", msg);
}
int main(int argc, char **argv)
{
    static int (*fp)(const char *msg);
    static char buffer[16];

    fp = (int(*)(const char *msg))callback;
    strcpy(buffer, argv[1]);
    (int)(*fp)(argv[2]); // call the callback
}
```
The exploit

• The program takes the first two arguments from the command line
• It copies `argv[1]` into a buffer with no bounds checking
• It then calls the callback, passing it the message from the 2nd argument

The exploit
  – Overflow the buffer
  – The overflow bytes will contain the address you really want to call
    • They’re strings, so bytes with 0 in them will not work … making this a more difficult attack
printf attacks
printf and its variants

Standard C library functions for formatted output

- `printf`: print to the standard output
- `wprintf`: wide character version of `printf`
- `fprintf`, `wfprintf`: print formatted data to a FILE stream
- `sprintf`, `swprintf`: print formatted data to a memory location
- `vprintf`, `vfprintf`, `vfprintf`, `vswprintf`, `vfprintf`: print formatted data containing a pointer to argument list

Usage

```c
printf(format_string, arguments ...)
printf(“The number %d in decimal is %x in hexadecimal\n”, n, n);
printf(“my name is %s\n”, name);
```
Bad usage of printf

Programs often make mistakes with printf

Valid:

```c
printf("hello, world!\n")
```

Also accepted ... but not right

```c
char *message = "hello, world\n");
printf(message);
```

This works but exposes the chance that `message` will be changed

*This should be a format string*
Dumping memory with printf

$ ./tt hello
hello

$ ./tt "hey: %012lx"
hey: 7ffe14a287f

printf does not know how many arguments it has. It deduces that from the format string.

If you don’t give it enough, it keeps reading from the stack

We can dump arbitrary memory by walking up the stack

$ ./tt %08x.%08x.%08x.%08x.%08x
6d10c308.6d10c320.85d636f0.a1b80d80.a1b80d80

#include <stdio.h>
#include <string.h>

int show(char *buf)
{
    printf(buf); putchar('\n');
    return 0;
}

int main(int argc, char **argv)
{
    if (argc == 2) {
        show(argv[1]);
    }
}

September 24, 2020 CS 419 © 2020 Paul Krzyzanowski
Have you ever used `%n`?

Format specifier that will store into memory the number of bytes written so far

```c
int printbytes;
printf("paul%n says hi
", &printbytes);
```

Will print

```
paul says hi
```

and will store the number 4 (strlen("paul")) into the variable `printbytes`

- If we combine this with the ability to change the format specifier, we can write to other memory locations
Bad usage of printf: %n

#include <stdio.h>
#include <string.h>

int show(char *buf)
{
    printf(buf);
    putchar('
');
    return 0;
}

int main(int argc, char **argv)
{
    char buf[256];
    if (argc == 2)
    {
        strncpy(buf, argv[1], 255);
        show(buf);
    }
    return 0;
}

printf treats this as the 1st parameter after the format string.
• We can skip ints with formatting strings such as %x
• The buffer can contain the address that we want to overwrite

#include <stdio.h>
#include <string.h>

int show(char *buf)
{
    printf(buf);
    putchar('
');
    return 0;
}

int main(int argc, char **argv)
{
    char buf[256];
    if (argc == 2)
    {
        strncpy(buf, argv[1], 255);
        show(buf);
    }
    return 0;
}
Printf attacks: %n

**What good is %n when it’s just # of bytes written?**

– You can specify an arbitrary number of bytes in the format string

```c
printf("%622404x%622400x%n" . . .
```

Will write the value 622404+622400 = 1244804 = 0x12fe84

What happens?

– `%622404x` = write at least 622404 characters for this value
– Each occurrence of %x (or %d, %b, ...) will go down the stack by one parameter (usually 8 bytes). We don’t care what gets printed
– The %x directives enabled us to get to the place on the stack where we want to change a value
– %n will write that value, which is the sum of all the bytes that were written
Part 3

Defending against hijacking attacks
Fix bugs

- Audit software

- Check for buffer lengths whenever adding to a buffer

- Search for unsafe functions
  - Use `nm` and `grep` to look for function names

- Use automated tools
  - Clockwork, CodeSonar, Coverity, Parasoft, PolySpace, Checkmarx, PREfix, PVS-Studio, PCPCheck, Visual Studio

- Most compilers and/or linkers now warn against bad usage

  tt.c:7:2: warning: format not a string literal and no format arguments [-Wformat-security]
  zz.c:(.text+0x65): warning: the 'gets' function is dangerous and should not be used.
Fix bugs: Fuzzing

• Technique for testing for & locating buffer overflow problems
  – Enter unexpected input
  – See if the program crashes

• Enter long strings with well-defined patterns
  – E.g., “$$$$$$$$”

• If the app crashes
  – Search the core dump for “$$” to find where it died

• Automated fuzzer tools help with this

• Or … try to construct exploits using gdb
Don’t use C or C++

• Most other languages feature
  – Run-time bounds checking
  – Parameter count checking
  – Disallow reading from or writing to arbitrary memory locations

• Hard to avoid in many cases
Specify & test code

• If it’s in the specs, it is more likely to be coded & tested

• Document acceptance criteria
  – “File names longer than 1024 bytes must be rejected”
  – “User names longer than 32 bytes must be rejected”

• Use safe functions that check allow you to specify buffer limits

• Ensure consistent checks to the criteria across entire source
  – Example, you might `#define` limits in a header file but some files might use a mismatched number.

• Check results from `printf`
Dealing with buffer overflows: No Execute (NX)

• **Data Execution Prevention (DEP)**
  – Disallow code execution in data areas – on the stack or heap
  – Set MMU per-page execute permissions to no-execute
  – Intel and AMD added this support in 2004

  – Examples
    • Microsoft DEP (Data Execution Prevention) (since Windows XP SP2)
    • Linux PaX patches
    • OS X ≥10.5
No Execute – not a complete solution

• **No Execute Doesn’t solve all problems**
  – Some applications need an executable stack (LISP interpreters)
  – Some applications need an executable heap
    • code loading/patching
    • JIT compilers
  – Does not protect against heap & function pointer overflows
  – Does not protect against printf problems
Return-to-libc

• Allows bypassing need for non-executable memory
  – With DEP, we can still corrupt the stack … just not execute code from it

• No need for injected code

• Instead, reuse functionality within the exploited app

• Use a buffer overflow attack to create a fake frame on the stack
  – Transfer program execution to the start of a library function
  – libc = standard C library
  – Most common function to exploit: system
    • Runs the shell
    • New frame in the buffer contains a pointer to the command to run (which is also in the buffer)
      – E.g., system("/bin/sh")
Return Oriented Programming (ROP)

- **Overwrite return address with address of a library function**
  - Does not have to be the start of the library routine
    - “borrowed chunks”
  - When the library gets to RET, that location is on the stack, under the attacker’s control

- **Chain together sequences ending in RET**
  - Build together “gadgets” for arbitrary computation
  - Buffer overflow contains a sequence of addresses that direct each successive RET instruction

- **It is possible for an attacker to use ROP to execute arbitrary algorithms without injecting new code into an application**
  - Removing dangerous functions, such as system, is ineffective
  - Make attacking easier: use a compiler that combines gadgets!
• **Addresses of everything were well known**
  - Dynamically-loaded libraries used to be loaded in the same place each time, as was the stack & memory-mapped files
  - Well-known locations make them branch targets in a buffer overflow attack

• **Address Space Layout Randomization (ASLR)**
  - Position stack and memory-mapped files to random locations
  - Position libraries at random locations
    • Libraries must be compiled to produce position independent code
  - Implemented in
    • OpenBSD, Windows ≥Vista, Windows Server ≥2008, Linux ≥2.6.15, macOS, Android ≥4.1, iOS ≥4.3
  - But … not all libraries (modules) can use ASLR
    • And it makes debugging difficult
Address Space Layout Randomization

• **Entropy**
  – How random is the placement of memory regions?

• **Examples**
  – Linux Exec Shield patch
    • 19 bits of stack entropy, 16-byte alignment > 500K positions
    • Kernel ASLR added in 3.14 (2014)
  – Windows 7
    • 8 bits of randomness for DLLs
      – Aligned to 64K page in a 16MB region: 256 choices
  – Windows 8
    • 24 bits for randomness on 64-bit processors: >16M choices
Dealing with buffer overflows: Canaries

- **Stack canaries**
  - Place a random integer before the return address on the stack
  - Before a return, check that the integer is there and not overwritten: a buffer overflow attack will likely overwrite it

```c
int a, b=999;
char s[5], t[7];
gets(s);
```
Dealing with buffer overflows: Canaries

- **Stack canaries**
  - Place a random integer before the return address on the stack
  - Before a return, check that the integer is there and not overwritten: a buffer overflow attack will likely overwrite it

```c
int a, b=999;
char s[5], t[7];
gets(s);
```
IBM’s ProPolice gcc patches

- Allocate arrays into higher memory in the stack
- Ensures that a buffer overflow attack will not clobber non-array variables
- Increases likelihood that the overflow won’t attack the logic of the current function

```c
int a, b=999;
char s[5], t[7];
gets(s);
```
Stack canaries

• Again, not foolproof

• Heap-based attacks are still possible

• Performance impact
  – Need to generate a canary on entry to a function and check canary prior to a return
  – Minimal degradation ~8% for apache web server
Developed by Intel & Microsoft to thwart ROP attacks
- Availability announced for Tiger Lake microarchitecture (mid-2020)

• Two mechanisms
  1. Shadow stack
  2. Indirect branch tracking

• Shadow Stack
  - Secondary stack
    • Only stores return addresses
    • MMU attribute disallows use of regular store instructions to modify it
  - Stack data overflows cannot touch the shadow stack – cannot change control flow
• **Indirect Branch Tracking**
  – Restrict a program’s ability to use jump tables
  – Jump table = table of memory locations the program can branch
    • Used for switch statements & various forms of lookup tables
  – Jump-Oriented Programming (JOP) and Call Oriented Programming (COP)
    • Techniques where attackers abuse JMP or CALL instructions
    • Like Return-Oriented Programming but use gadgets that end with indirect branches
  – New **ENDBRANCH** (ENDBR64) instruction allows a programmer to specify valid targets for indirect jumps
    • If you take an indirect jump, it has to go to an ENDBRANCH instruction
    • If the jump goes anywhere else, it will be treated as an invalid branch and generate a fault
Heap attacks – pointer protection

• Encrypt pointers (especially function pointers)
  – Example: XOR with a stored random value
  – Any attempt to modify them will result in invalid addresses
  – XOR with the same stored value to restore original value

• Degrades performance when function pointers are used
Safer libraries

• Compilers warn against unsafe \textit{strcpy} or \textit{printf}

• Ideally, fix your code!

• Sometimes you can’t recompile (e.g., you lost the source)

• \texttt{libsafe}
  – Dynamically loaded library
  – Intercepts calls to unsafe functions
  – Validates that there is sufficient space in the current stack frame
    \[(\text{framepointer} - \text{destination}) > \text{strlen(src)}\]
The end
The End