Week 3: Code Injection
Part 1

Program Hijacking
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
      • Includes SQL injection – send database commands
    – **Code injection**
      • Inject code into a program that is then executed by the application
Examples of Hijacking

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

• **Code injection**
  – Overflow an input buffer and cause new code to run
  – Provide JavaScript as input that will later get executed (Cross-site scripting)
  – Library injection: load different dynamic libraries that cause different versions of code run

• **Command injection**
  – Provide input that will get interpreted and run as a system command
  – Change search paths to run different programs

• **Other forms**
  – Redirect web browser to a malicious site
  – Change DNS (IP address lookup) results
  – Change search engine
• Control hijacking isn’t interesting for regular programs on your system
  – You might as well just run commands from the shell

• It is interesting if the program
  – Has elevated privileges (setuid), especially runs as 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

• Four 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
  4. A file is in a proper format
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 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
  – Insecure access rights often help with gaining access or more privileges
  – We will continue to write buggy code!
Buffer overflows … still going strong

Nov. 19, 2021: NETGEAR meltdown

- Affects 61 different devices
- Allows attackers to execute arbitrary code on routers
- Authentication is not required for exploit
- Bug in UPnP service on TCP port 5000
  - When parsing the uuid request header, the process does not properly validate the length of user-supplied data prior to copying it to a fixed-length stack-based buffer.

Dec. 9, 2021: SonicWall
- Affects SMA (Secure Mobile Access) 100 Series
- Multiple heap-based and stack-based buffer overflows
- Can be accessed by unauthenticated users
- Bug in `fileexplorer` component
  - Unchecked use of `strcpy` with a fixed size buffer
  - Assumes username and password will each be <128 bytes
  - Same bug with the domain name

Feb. 2, 2021: Linux sudo

- Heap-based buffer overflow vulnerability
- An attacker could exploit this vulnerability to take control of an affected system.

- Off-by-one error
  - Can result in a heap-based buffer overflow, which allows privilege escalation to root via "sudoedit -s" and a command-line argument that ends with a single backslash character.

Feb. 5, 2021: Google Chrome
- Buffer overflow vulnerability in V8, Google Chrome’s open-source JavaScript and WebAssembly engine
- Exploits in the wild have been observed
- Allows remote attacker to exploit heap corruption via a crafted HTML page
- Affects Microsoft Edge (Chromium based)

https://www.tenable.com/blog/cve-2021-21148-google-chrome-heap-buffer-overflow-vulnerability-exploited-in-the-wild
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

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

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

CVE-2019-16928: Critical Buffer Overflow Flaw in Exim is Remotely Exploitable

Edge Week 2020: Tenable’s Virtual User Conference. Oct 5th to 9th.

CVE-2019-16928, a critical heap-based buffer overflow vulnerability in Exim email servers, could allow remote attackers to crash Exim or potentially execute arbitrary code.

Background

Exim internet Mailer, the popular message transfer agent (MTA) for Unix hosts found on nearly 5 million systems, is back in the news. Earlier this month, CVE-2019-15846, a critical remote code execution (RCE) flaw, was patched in Exim 4.92.2. In June, we blogged about CVE-2019-10149, another RCE, which saw exploit attempts within a week of public disclosure.

On September 28, Exim maintainers published an advance notice concerning a new vulnerability in Exim 4.92 up to and including 4.92.2. From our analysis of Shodan results, over 3.5 million systems may be affected.

Analysis

CVE-2019-16928 is a heap-based buffer overflow vulnerability due to a flaw in `string_vformat()` found in `string.c`. As noted in the bug report, the flaw was a simple coding error where the length of the string was not properly accounted for, leading to a buffer overflow condition. The flaw can be exploited by an unauthenticated remote attacker who could use a large crafted Extended HELO (EHLO) string to crash the Exim process that receives the message. This could potentially be further exploited to execute arbitrary code on the host. The flaw was found internally by the QAX A-Team, who submitted the patch. However, the bug is trivial to exploit, and it’s likely attackers will begin actively probing for and attacking vulnerable Exim MTA systems in the near future.

Proof of concept
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.

2019 WhatsApp Buffer Overflow Vulnerability

- 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 ⇒ zero-click attack

- 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!

February 9, 2022

CS 419 © 2022 Paul Krzyzanowski

522 reported buffer overflow vulnerabilities
Jan 6 2021 – Feb 7, 2022

A buffer overflow vulnerability in libpcap component of libpcap 1.8.3 and older allows an attacker to execute arbitrary code by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CDataFile of the jwriapi component of LibreCAD 2.2.0–rc3 and older allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.

A buffer overflow vulnerability in CVE-2019-15019 allows an attacker to execute arbitrary code execution by writing beyond the bounds of a memory buffer, which may allow an attacker to execute arbitrary code.
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

Millions of IoT devices are vulnerable to buffer overflow attack

A buffer overflow flaw has been found by security researchers (at the IoT-focused security firm Senrio) in an open-source software development library that is widely used by major manufacturers of the Internet-of-Thing devices.

The buffer overflow vulnerability (CVE-2017-9765), which is called “Devil's Ivy” enables a remote attacker to crash the SOAP (Simple Object Access Protocol) WebServices daemon and make it possible to execute arbitrary code on the affected devices.

Security researchers discloses vulnerabilities in TLS libraries and the downgrade Attack on TLS 1.3

The classic buffer overflow bug

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

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

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.\n"
if (!warned) {
    (void) write(STDERR_FILENO, w, sizeof(w) - 1);
    warned = 1;
}
```
```cpp
#include <iostream>

using namespace std;

int main()
{
    char x[4] = "cat";
    char y[4];
    char z[4] = "dog";

    cout << "Enter a word:";
    cin >> y;
    cout << "Read " << strlen(y) << " characters." << endl;
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "z: " << z << endl;
}
```
#include <iostream>

using namespace std;

int main()
{
    char x[4] = "cat";
    char y[4];
    char z[4] = "dog";

    cout << "Enter a word:"
    cin >> y;
    cout << "Read " << strlen(y) << " characters." << endl;
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "z: " << z << endl;
}

$ g++ -o cin cin.cpp
Enter a word:abcdefg
Read 7 characters.
x: efg
y: abcdefg
z: dog

The data in y overflowed to x
x got corrupted
C++ too – and no warnings!

```cpp
#include <iostream>

using namespace std;

int main()
{
    char x[4] = "cat";
    char y[4];
    char z[4] = "dog";

    cout << "Enter a word:";
    cin >> y;
    cout << "Read " << strlen(y) << " characters." << endl;

    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "z: " << z << endl;

    $ g++ -o cin cin.cpp
    Enter a word:abcdefghijklmnopqrstuvwxyz0123456789
    Read 36 characters.
    x: efghijklmnopqrstuvwxyz0123456789
    y: abcdefghijklmnopqrstuvwxyz0123456789
    z: dog
    cout << "Bus error: 10"

    With even more data, 
x got corrupted
    AND the program crashed!
}
void test(void) {
    char name[10];

    strcpy(name, "krzyzanowski");
}

That’s easy to spot!
Another example

How about this?

cchar configfile[256];
cchar *base = getenv("BASEDIR");

if (base != NULL)
    sprintf(configfile, "%s/config.txt", base);
else {
    fprintf(stderr, "BASEDIR not set\n");
}
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**
What’s the harm?

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
Taking advantage of unchecked bounds

```c
#include <stdio.h>
#include <strings.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 my Raspberry Pi
  Raspbian GNU/Linux 10
  5.10.63-v7l+
Or my Mac Mini M1 running macOS 12.2
Or my Intel i7 iMac running macOS 12.2
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 4-5 of popular programming languages**
  - #1 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**

**Loaded by execve**

- **0xc0000000**
- **0x40000000**
- **brk**
- **0x08048000**

*Not to scale*
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?

- Return address
- Previous frame pointer
- Parameter (s)
- Return address
- Saved rbp (frame pointer)
- char buf[128]
- rbp (current frame pointer)
- rsp (current stack pointer)
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`

- **Return address**
- **Previous frame pointer**
- **parameter (s)**
- **Return address**
- Saved `rbp` (frame pointer)
- `char buf[128]`

- `rbp` (current frame pointer)
- `rsp` (current stack pointer)

High memory

Low memory
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

Diagram:
- High memory
- Previous return address
- Previous frame pointer
- `params`
- Overwritten return address
- `char buf[128]`
- MALICIOUS CODE
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
Address Uncertainty

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

**NOP slide = NOP sled = 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?**
Off-by-one errors: frame pointer mangling

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

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?
Memory overflow

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
```

(February 9, 2022)
Memory overflow – filename example

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`!

The output

(Windows 5.10.63, gcc 8.3.0)

| Planning to write to /etc/secret.txt |
| About to open afile=/home/paul/writehere.txt |
Overwriting variables: changing control flow

- 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

```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
}
```
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`, `vwprintf`, `vfprintf`, `vwpfprintf`: 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:

\[
\text{printf(“hello, world!\n”)}
\]

Also accepted … but not right

\[
\text{char *message = “hello, world\n”;} \\
\text{printf(message);} \\
\]

This works but exposes the chance that message will be changed

\[
\text{This should be a format string}
\]
#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]);
    }
}

$ ./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
Getting into trouble with printf

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\n", &printbytes);
```

Will print

```
paul says hi
```

and will store the number 4 (which is the value of `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

`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

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

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

int main(int argc, char **argv)
{
    char buf[256];

    if (argc == 2) {
        strncpy(buf, argv[1], 255);
        show(buf);
    }
}
```
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

\[ \text{printf("%x%x%n"} \ldots \]

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

```plaintext
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

• **Generate semi-random data as input to detect bugs**
  – Locating input validation & buffer overflow problems
  – Enter unexpected input
  – See if the program crashes

• **Enter long strings with searchable patterns**

• **If the app crashes**
  – Search the core dump for the fuzz pattern to find where it died

• **Automated fuzzer tools help with this**
  – E.g., libFuzzer and AFL in C/C++; cargo-fuzz in Rust

• **Or ... try to construct exploits using gdb**
Fuzzing in Go

• Fuzzing available in Go 1.18 (released Feb 2022)

• Goal
  – Make fuzz testing as easy as benchmarking or unit testing
  – No need for custom tools or separate files

• Seed corpus: user-specified set of inputs to a fuzz test
  – Fuzzing engine will mutate these inputs to discover new code coverage

https://go.googlesource.com/proposal/+/master/design/draft-fuzzing.md
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.

• Don't allow user-generated format strings and check results from `printf`
Safer libraries

• Compilers warn against unsafe `strcpy` or `printf`  
• Ideally, fix your code!  
• Sometimes you can’t recompile (e.g., you lost the source)  
• `libsafe`  
  – Dynamically loaded library  
  – Intercepts calls to unsafe functions  
  – Validates that there is sufficient space in the current stack frame: 
    
    \[(\text{framepointer} - \text{destination}) > \text{strlen(src)}\]
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 … every program uses it!`
  – 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
    • Use “borrowed chunks” from a various libraries
  – 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!
    • Example: ROPC – a Turing complete compiler, https://github.com/pakt/ropc
• **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

- Available with the Tiger Lake microarchitecture (mid-2020)

Two mechanisms

1. **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

2. **Indirect branch tracking**
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
Hardware Attacks: Example - Rowhammer

**DDR4 memory protections are broken wide open by new Rowhammer technique**

Researchers build "fuzzer" that supercharges potentially serious bitflipping exploits.

Dan Goodin • 11/15/2021

Rowhammer exploits that allow unprivileged attackers to change or corrupt data stored in vulnerable memory chips are now possible on virtually all DDR4 modules due to a new approach that neuters defenses chip manufacturers added to make their wares more resistant to such attacks.

Rowhammer attacks work by accessing—or hammering—physical rows inside vulnerable chips millions of times per second in ways that cause bits in neighboring rows to flip, meaning 1s turn to 0s and vice versa. Researchers have shown the attacks can be used to give untrusted applications nearly unfettered system privileges, bypass security sandboxes designed to keep malicious code from accessing sensitive operating system resources, and root or infect Android devices, among other things.

[https://arstechnica.com/gadgets/2021/11/ddr4-memory-is-even-more-susceptible-to-rowhammer-attacks-than-anyone-thought/](https://arstechnica.com/gadgets/2021/11/ddr4-memory-is-even-more-susceptible-to-rowhammer-attacks-than-anyone-thought/)
Hardware Attacks: Example - Rowhammer

• **New attack technique discovered**
  – Uses non-uniform patterns that access two or more rows with different frequencies
  – Bypasses all defenses built into memory hardware
  – 80% of devices can be hacked this way
  – Cannot be patched!

• **Sample attacks**
  – Gain unrestricted access to all physical memory by changing bits in the page table entry
  – Give untrusted applications root privileges
  – Extract encryption key from memory
The end
The End
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>