Computer Security

03. Program Hijacking & Code Injection

Paul Krzyzanowski
Rutgers University
Spring 2019

1

Top vulnerability concerns for 2019

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

Rank	Name	Score
1	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
2	Cross-site Scripting	45.69
3	Improper Input Validation	43.61
4	Information Exposure	32.12
5	Out-of-bounds Read	26.53
6	SQL Injection	24.54
7	Use After Free	17.94
8	Integer Overflow or Wraparound	17.35
9	Cross-Site Request Forgery (CSRF)	15.54
10	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10

https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

2

Hijacking

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

Examples:

- Redirect web browser to a malicious site
- Change DNS (IP address lookup) results
- Change search engine
- Change search paths to load different libraries or have different programs run
- Intercept & alter messages

Code injection

Getting a program to process data in a way that it changes the execution of a program

3

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
- 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 … and often is

4

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
- Three common assumptions
  - Buffer is large enough for the data
  - Integer overflow doesn't exist
  - User input will never be processed as a command

5

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

6

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

7

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, set-top boxes, access points, phone switches
  - Insecure access rights often help with getting more privileges
  - We will continue to write buggy code!

8

Buffer Overflows … still going strong

Just a few of hundreds of vulnerabilities…

- 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/
http://www-01.ibm.com/support/docview.wss?uid=swg22003877
https://landave.io/2017/06/avast-antivirus-remote-stack-buffer-overflow-with-magic-numbers/

9

Buffer Overflows … and going…

- Mar 2018: Exim mailer (used on ~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

357 buffer overflow vulnerabilities posted on the National Vulnerability Database (https://nvd.nist.gov/vuln) from Jan-Sept 2019

10

ars technica

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.

https://arstechnica.com/information-technology/2019/05/whatsapp-vulnerability-exploited-to-infect-phones-with-israeli-spyware/

11

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.
- This was a **zero-day vulnerability**
  - Attackers found & exploited the bug before the company could patch it
- WhatsApp is used by 1.5 billion people
  - Vulnerability discovered in May 2019 while developers were making security improvements

https://arstechnica.com/information-technology/2019/05/whatsapp-vulnerability-exploited-to-infect-phones-with-israeli-spyware/

12

Buggy libraries can affect a lot of code bases

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.

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

https://latesthackingnews.com/2017/07/18/millions-of-iot-devices-are-vulnerable-to-buffer-overflow-attack/

13

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);
}
```


14

Buffer overflow examples

```
void test(void) {
    char name[10];

    strcpy(name, "krzyzanowski");
}
```

That's easy to spot!

15

Another example

How about this?

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

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


16

You might not notice

You made unchecked assumptions on the maximum password length

```
char passwd1[80], passwd2[80];

printf("Enter password: ");
gets(passwd1);
printf("Enter password again: ");
gets(passwd2);
if (strcmp(passwd1, passwd2) != 0) {
    fprintf(stderr, "passwords don't match\n");
    exit(1);
}
...
```


17

Buffer overflow attacks

To exploit a buffer overflow

- Identify overflow vulnerability in a program
  - Inspect source
  - Trace execution
  - Use fuzzing tools (more on that …)
- Understand where the buffer is in memory and whether there is potential for corrupting surrounding data

18

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

19

Taking advantage

```
#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);
}
```

Run on iLab system:
CentOS Linux 7 (3.10)
X86-64: i7-7700 CPU @ 3.60GHz

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


20

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

21

Programming at the machine level

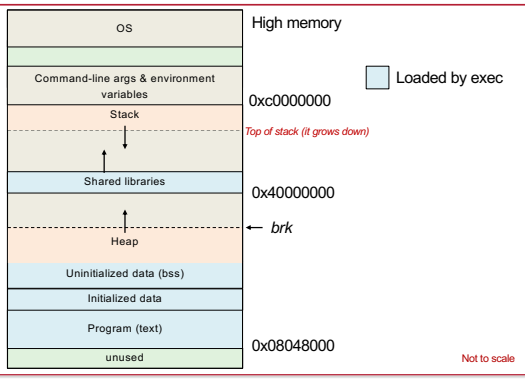
- High level languages (even C) constrain you in
  - Access to variables (local vs. global)
  - Control flows in predictable ways
    - Loops, function entry/exit, exceptions
- At the machine code level
  - No restriction on where you can jump
    - Jump to the middle of a function … or to the middle 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)

22

Stack overflows

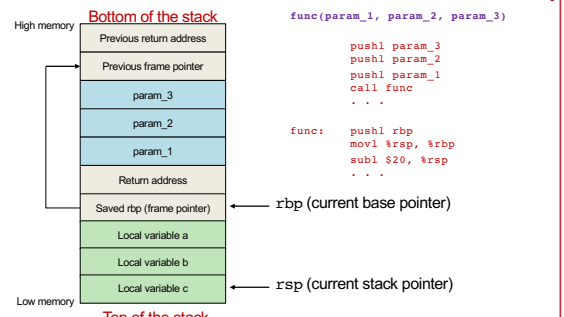
23

Linux process memory map



24

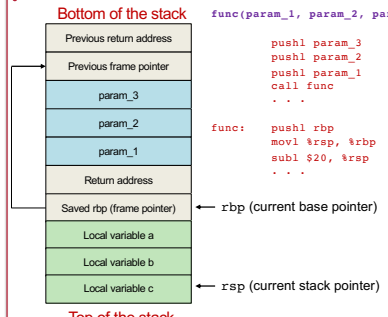
The stack



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

25

The stack: frames, base pointer, & stack pointer



Stack frame:

Refers to the memory on the stack that is used by a function to hold local data.

Base pointer (`rbp`):

Points to the base of the function's stack frame.

Parameters & local variables are referenced as offsets from the base pointer.

The saved frame pointers form a linked list of stack frames

Stack pointer (`rsp`):

Points to the top of the current stack frame.

26

Causing overflow

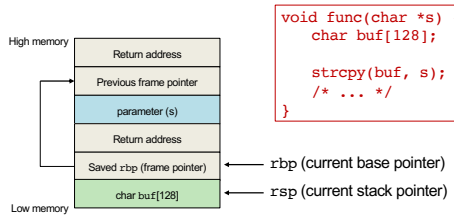
Overflow can occur when programs do not validate the length of data being written to a buffer

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…

27

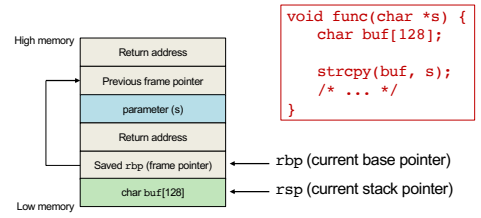
Overflowing the buffer



What if `s` is >128 bytes?

28

Overflowing the buffer



What if `s` is >128 bytes?
You overwrite the saved *rbp* and then the *return address*

29

Overwriting the return address

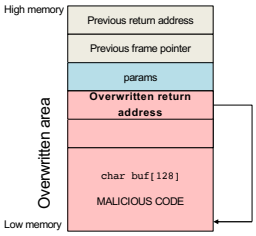
- If we overwrite the return address
  - We change what the program executes when it returns from the function
- "Benign" overflow
  - Overflow with garbage data
  - Chances are that the return address will be invalid
  - Program will die with a SEGFAULT
  - Availability attack

30

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



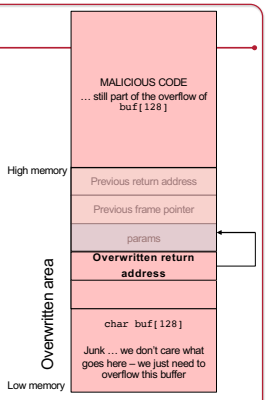
31

Subverting control flow

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



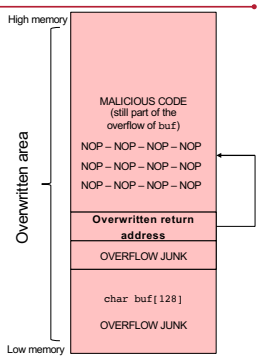
32

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



33

Off-by-one overflows

34

Safe functions aren't always safe

- Safe counterparts require a count
  - *strcpy → strncpy*
  - *strcat → strncat*
  - *sprintf → snprintf*
- But programmers can miscount!

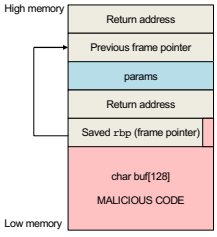
```
char buf[512];
int i;

for (i=0; i<=512; i++)
   buf[i] = stuff[i];
```


35

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?

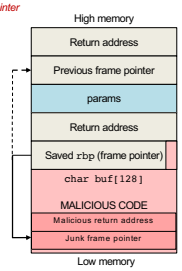


36

Off-by-one errors

- At the end of a function:
  - The compiler resets the stack pointer (%rsp) to the base of the frame (%rbp):
    ```
    mov %rsp, %rbp
    ```
  - and restores the saved base pointer (which we corrupted) from the top of the stack:
    ```
    pop %rbp    pops corrupted base pointer into rbp, the base pointer
    ret
    ```

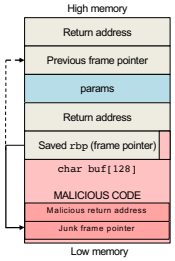
  The program now has the wrong base pointer when the function returns
- The function returns normally (we could not overwrite the return address)
- BUT … when the function that called it tries to return, it will update the stack pointer to what it thinks was the valid base pointer and return there:
  ```
  mov %rsp, %rbp    rbp is our corrupted one
  pop %rbp          we don't care about the base pointer
  ret               return pops the stack from our buffer, so we can jump anywhere
  ```



37

Off-by-one errors

- Stuff the buffer with
  - Local variables
  - "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

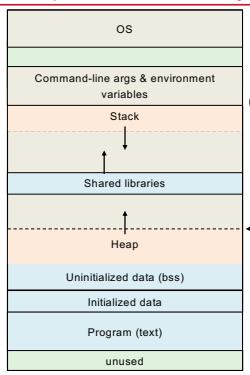


38

Heap & text overflows

39

Linux process memory map



Loaded by exec

- 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?

40

Memory overflow

The program

```
#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
(Linux 4.4.0-59, gcc 5.4.0)

```
a=qrstuvwxyz
b=abcdefghijklmnopqrstuvwxyz
```


41

Memory overflow

- We may be able to overflow a buffer and overwrite other variables in ***higher*** memory
- For example
  - Overwrite a file name

42

Memory overflow

The program

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

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

                                  mybuf can overflow into afile
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/usr/paul/writehere.txt");
     printf("about to open afile=%s\n", afile);
     exit(0);
}
```

The output (Linux 4.4.0-59, gcc 5.4.0)

```
planning to write to /etc/secret.txt
about to open afile=/usr/paul/writehere.txt
```


43

Overwriting variables

- Even if a buffer overflow does not touch the stack, it can modify global or local variables
- Example:
  - Overwrite a function pointer
  - Function pointers are often used in callbacks

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

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


44

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 of the function you really want to call
  - They're strings, so bytes with 0 in them will not work … making this a more difficult attack

45

printf attacks

46

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: print formatted data containing a pointer to argument list
  - vfprintf, vwfprintf: print formatted data containing a pointer to argument list
- Usage

```
printf(format_string, arguments…)
printf("The number %d in decimal is %x in hexadecimal\n", n, n);
printf("my name is %s\n", name);
```


47

Bad usage of printf

Programs often make mistakes with printf

Valid:

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

Also accepted … but not right

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

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

This should be a format string

48

Dumping memory with printf

```
#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);
    }
}
```

```
$ ./tt hello
hello

$ ./tt "hey: %012lx"
hey: 7fffe14a287f
```

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
00000009.00000000.b8875c20.0000000f
```


49

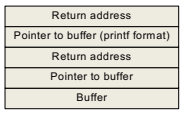
Getting into trouble with printf

- Have you ever used `%n` ?
- Format specifier that will store into memory the number of bytes written so far

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

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 arbitrary memory locations



50

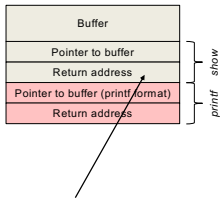
Bad usage of printf

```
#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 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 – e.g., any return address.

51

Printf attacks

- What good is %n when it's just # of bytes written?
  - You can specify an arbitrary number of bytes in the format string

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

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

What happens?

- `%.622404x` = write at least 622404 characters for this value
- Each occurrance 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

52

Defending against hijacking attacks

53

Fix bugs

- Audit software
- Check for buffer lengths whenever adding to a buffer
- Search for unsafe functions
  - Use *nm* (dump the symbol table) and *grep* (search) 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.
```


54

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

55

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

56

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*

57

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

58

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

59

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")`

60

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 generates gadgets!
    - Example: ROPC – a Turing complete compiler, https://github.com/pakt/ropc

61

Dealing with buffer overflows & ROP: ASLR

Address Space Layout Randomization

- 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
- 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

62

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 ASLD 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

63

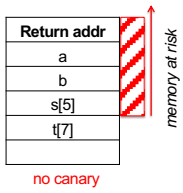
Dealing with buffer overflows: Canaries

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

```
int a, b=999;
char s[5], t[7];

gets(s);
```



64

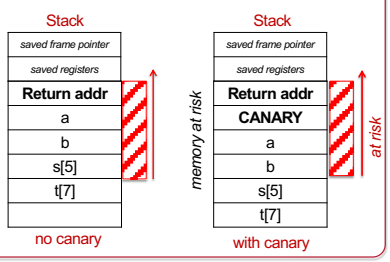
Dealing with buffer overflows: Canaries

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

```
int a, b=999;
char s[5], t[7];

gets(s);
```



65

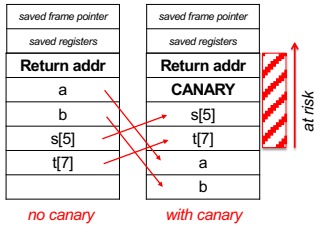
ProPolice

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

```
int a, b=999;
char s[5], t[7];

gets(s);
```



66

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

67

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

68

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

```
(framepointer – destination) > strlen(src)
```


69

Command injection attacks

71

Command injection attacks

- Allows an attacker to inject commands or code into a program or query to:
  - Execute commands
  - Modify a database
  - Change data on a website
- We looked at buffer overflow for code injection and format strings for data reading/modification
  … but there are other forms too

Latest list as of Feb 10 2019
https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

72

Bad Input: SQL Injection

- Let's create an SQL query in our program

```
sprintf(buf,
    "SELECT * WHERE user='%s' AND query='%s';",
    uname, query);
```

- You're careful to limit your queries to a specific user
- But suppose *query* comes from user input and is:

```
foo' OR user='root
```

- The command we create is:

```
SELECT * WHERE user='paul' AND query='foo' OR user='root';
```


73

What's wrong?

We didn't validate our input

– And ended up creating a query that we did not intend to create!

74

Another example: password validation

Suppose we're validating a user's password:

```
sprintf(buf,
"SELECT * from logininfo WHERE username = '%s' AND password = '%s';",
uname, passwd);
```

But suppose the user entered this for a password:

```
' OR 1=1 --
```

The -- is a comment that blocks the rest of the query (if there was more)

The command we create is:

```
SELECT * from logininfo WHERE username = paul AND
password = '' OR 1=1 -- ;
```

1=1 is always true!
We bypassed the password check!

75

Opportunities for destructive operations

https://xkcd.com/327/

Most databases support a batched SQL statement: multiple statements separated by a semicolon

```
SELECT * FROM students WHERE name = 'Robert';DROP TABLE Students; --
```


76

Protection from SQL Injection

- SQL injection attacks are incredibly common because most web services are front ends to database systems
  - Input from web forms becomes part of the command
- Type checking is difficult
  - SQL contains too many words and symbols that may be legitimate in other contexts
- Use escaping for special characters
  - Replace single quotes with two single quotes
  - Prepend backslashes for embedded potentially dangerous characters (newlines, returns, nuls
- Escaping is error-prone
  - Rules differ for different databases (MySQL, PostgreSQL, dashDB, SQL Server, …

Don't create commands with user-supplied substrings added into them

77

Protection from SQL Injection

Use parameterized SQL queries or stored procedures

– Keeps query consistent:
parameter data never becomes part of the query string

```
uname = getResourceString("username");
passwd = getResourceString("password");
query = "SELECT * FROM users WHERE username = @0 AND password = @1";
db.Execute(query, uname, passwd);
```


78

General Rule

If you invoke *any* external program,
know its parsing rules

- Converting data to statements that get executed is common in some interpreted languages
  - Shell, Perl, PHP, Python

79

IFS

Shell variable IFS (Internal Field Separator) defines delimiters used in parsing arguments
- If you can change IFS, you may change how the shell parses data
- The default is space, tab, newline

try1.sh

```
#!/bin/bash
while read name password; do
    echo name=\"$name\", password=\"$password\"
done
```

names

```
james password
mary 123456
john qwerty
patricia letmein
robert shadow
jennifer harley
```

output

```
$ ./try1.sh <names
name="james", password="password"
name="mary", password="123456"
name="john", password="qwerty"
name="patricia", password="letmein"
name="robert", password="shadow"
name="jennifer", password="harley"
```


80

IFS

One small change: IFS=+

try1.sh

```
#!/bin/bash
IFS=+
while read name password; do
    echo name=\"$name\", password=\"$password\"
done
```

names

```
james password
mary 123456
john qwerty
patricia letmein
robert shadow
jennifer harley
```

output

```
$ ./try1.sh <names
name="james password", password=""
name="mary 123456", password=""
name="john qwerty", password=""
name="patricia letmein", password=""
name="robert shadow", password=""
name="jennifer harley", password=""
```


81

IFS

It gets tricky for output

try.sh

```
#!/bin/bash

IFS='+'

echo '"$@" expansion'
echo "$@"

echo '"$*" expansion'
echo "$*"
```

```
$ ./try.sh sleepy sneezy grumpy dopey doc
"$@" expansion
sleepy sneezy grumpy dopey doc
"$*" expansion
sleepy+sneezy+grumpy+dopey+doc
```

You really have to know what you're dealing with!

Suppose a program wants to send mail. It might call:

```
FILE *fp = popen("/usr/bin/mail –s subject user", "w")
```

If IFS is set to "/" then the shell will try to execute usr bin mail…
An attacker needs to plant a program named "usr" anywhere in the search path

82

system() and popen()

- These library functions make it easy to execute programs
  - *system*: execute a shell command
  - *popen*: execute a shell command and get a file descriptor to send output to the command or read input from the command
- These both run `sh –c command`
- Vulnerabilities include
  - Altering the search path if the full path is not specified
  - Changing IFS to change the definition of separators
  - Using user input as part of the command

```
snprintf(cmd, "/usr/bin/mail -s alert %s", bsize, user);
f = popen(cmd, "w");
```

What if user = `"paul;rm –fr /home/*"`

```
sh –c "/usr/bin/mail -s alert paul; rm –fr /home/*"
```


83

Other environment variables

- PATH: search path for commands
  - If untrusted directories are in the search path before trusted ones (`/bin`, `/usr/bin`), you might execute a command there.
    - Users sometimes place the current directory (.) at the start of their search path
    - What if the command is a booby-trap?
  - If shell scripts use commands, they're vulnerable to the user's path settings
  - Use absolute paths in commands or set PATH explicitly in a script
- ENV, BASH_ENV
  - Set to a file name that some shells execute when a shell starts

84

Other environment variables

LD_LIBRARY_PATH
- Search path for shared libraries
- If you change this, you can replace parts of the C library by custom versions
  - Redefine system calls, *printf*, whatever…

LD_PRELOAD
- Forces a list of libraries to be loaded for a program, even if the program does not ask for them
- If we preload our libraries, they get used instead of standard ones

You won't get root access with this, but you can change the behavior of programs
- Change random numbers, key generation, time-related functions in games
- List files or network connections that a program does
- Modify features or behavior of a program

85

Function interposition

interpose
(ĭn'tər-pōz')

1. Verb (transitive)
   to put someone or something in a position between two other people or things
   He swiftly interposed himself between his visitor and the door.
2. To say something that interrupts a conversation

- Change the way library functions work without recompiling programs
- Create wrappers for existing functions

86

Example of LD_PRELOAD

random.c

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
  int i;

  srand(time(NULL));
  for (i=0; i < 10; i++)
    printf("%d\n", rand()%100);
  return 0;
}
```

```
$ gcc -o random random.c
$ ./random
9
57
13
1
83
86
45
63
51
5
```


87

Let's create a replacement for rand()

rand.c

```
int rand() {
    return 42;
}
```

```
$ gcc -shared -fPIC rand.c -o newrandom.so   # compile
$ export LD_PRELOAD=$PWD/newrandom.so  # preload
$ ./random
42
42
42
42
42
42
42
42
42
42
```

We didn't have to recompile *random!*

88

File descriptor vulnerabilities

89

File Desciptors

- On POSIX systems
  - File descriptor 0 = standard input (*stdin*)
  - File descriptor 1 = standard output (*stdout*)
  - File descriptor 2 = standard error (*stderr*)
- *open*() returns the first available file descriptor

Vulnerability
- Suppose you close file descriptor 1
- Invoke a setuid root program that will open some sensitive file for output
- Anything the program prints to *stdout* (e.g., via *printf*) will write into that file, corrupting it

90

File Descriptors - example

files.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
  int fd = open("secretfile", O_WRONLY|O_CREAT, 0600);

  fprintf(stderr, "fd = %d\n", fd);
  printf("hello!\n");
  fflush(stdout); close(fd);
  return 0;
}
```

```
$ ./files
fd = 3
hello!
$ ./files >&-
fd = 1
```

Bash command to close a file descriptor
We close the standard output
We just corrupted secretfile

91

Obscurity

Windows CreateProcess function

```
BOOL WINAPI CreateProcess(
  _In_opt_    LPCTSTR               lpApplicationName,
  _Inout_opt_ LPTSTR                lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_        BOOL                  bInheritHandles,
  _In_        DWORD                 dwCreationFlags,
  _In_opt_    LPVOID                lpEnvironment,
  _In_opt_    LPCTSTR               lpCurrentDirectory,
  _In_        LPSTARTUPINFO         lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION lpProcessInformation);
```

- 10 parameters that define window creation, security attributes, file inheritance, and others…
- It gives you a lot of control but do most programmers know what they're doing?

92

Pathname parsing

93

App-level access control: filenames

- If we allow users to supply filenames, we need to check them
- App admin may specify acceptable pathnames & directories
- Parsing is tricky
  - Particularly if wildcards are permitted (*, ?)
  - And if subdirectories are permitted

94

Parsing directories

- Suppose you want to restrict access outside a specified directory
  - Example, ensure a web server stays within `/home/httpd/html`
- Attackers might want to get other files
  - They'll put `..` in the pathnaame
    `..` is a link to the parent directory
    For example:
    `http://pk.org/../../../etc/passwd`
  - The `..` does not have to be at the start of the name – could be anywhere
    `http:// pk.org /419/notes/../../416/../../../../etc/passwd`
  - But you can't just search for `..` because an embedded `..` is valid
    `http:// pk.org /419/notes/some..junk..goes..here/`
  - Also, extra slashes are fine
    `http:// pk.org /419////notes///some..junk..goes..here///`

Basically, it's easy to make mistakes!

95

Application-Specific Syntax: Unicode

Here's what Microsoft IIS did

- Checked URLs to make sure the request did not use `../` to get outside the *inetpub* web folder
  - Prevents http://www.poopybrain.com/scripts/../../winnt/system32/cmd.exe
- Then it passed the URL through a decode routine to decode extended Unicode characters
- Then it processed the web request

What went wrong?

96

Application-Specific Syntax: Unicode

- What's the problem?
  - `/` could be encoded as unicode `%c0%af`
- UTF-8
  - If the first bit is a 0, we have a one-byte ASCII character
    - Range 0..127
      `/` = 47 = `0x2f` = `0010 0111`
  - If the first bit is 1, we have a multi-byte character
    - If the leading bits are 110, we have a 2-byte character
    - If the leading bits are 1110, we have a 3-byte character, and so on…
  - 2-byte Unicode is in the form `110a bcde 10fg hijk`
    - 11 bits for the character # (codepoint), range 0 .. 2047
    - C0 = 1100 0000, AF = 1010 1111 which represents 0x2f = 47
  - **Technically, two-byte characters should not process # < 128**
    … but programmers are sloppy … and we want the code to be fast

97

Application-Specific Syntax: Unicode

- Parsing ignored %c0%af as / because it shouldn't have been one
- So intruders could use IIS to access *ANY* file in the system
- IIS ran under an IUSR account
  - Anonymous account used by IIS to access the system
  - IUSER is a member of *Everyone* and *Users* groups
  - Has access to execute most system files, including cmd.exe and command.com
- A malicious user had the ability to execute *any* commands on the web server
  - Delete files, create new network connections

98

Parsing escaped characters

Even after Microsoft fixed the Unicode bug, another problem came up

- If you encoded the backslash (\) character (Microsoft uses backslashes for filenames & accepts either in URLs)
- … and then encoded the encoded version of the \, you could bypass the security check

  \ = %5c
  - % = %25
  - 5 = %35
  - c = %63

  For example, we can also write:
  - %%35c => %5c => \
  - %25%35%63 => %5c => \
  - %255c => %5c => \

Yuck!

http://help.sap.com/SAPHELP_NWPI71/helpdata/en/df/c36a376a3a43ceaaa879ab726f0ec8/content.htm

99

These are application problems

- The OS uses whatever path the application gives it
  - It traverses the directory tree and checks access rights as it goes along
    - "x" (search) permissions in directories
    - Read or write permissions for the file
- The application is trying to parse a pathname and map it onto a subtree
- Many other characters also have multiple representations
  - á = U+00C1 = U+0041,U+0301

Comparison rules must be handled by applications and be application dependent

100

Access check attacks

101

Setuid file access

Some commands may need to write to restricted directories or files but also access user's files

- Example: some versions of *lpr* (print spooler) read users' files and write them to the spool directory
- Let's run the program as *setuid* to *root*
  But we will check file permissions first to make sure the user has read access

```
if (access(file, R_OK) == 0) {
    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```


102

Problem: TOCTTOU

```
if (access(file, R_OK) == 0) {

    fd = open(file, O_RDONLY);
    ret = read(fd, buf, sizeof buf);
    ...
}
else {
    perror(file);
    return -1;
}
```

- Race condition:
  TOCTTOU: Time of Check to Time of Use
- Window of time between *access* check & *open*
  - Attacker can create a link to a readable file
  - Run *lpr* in the background
  - Remove the link and replace it with a link to the protected file
  - The protected file will get printed

103

mktemp is also affected by this race condition

Create a temporary file to store received data

```
if (tmpnam_r(filename)) {
  FILE* tmp = fopen(filename, "wb+");
  while((recv(sock, recvbuf, DATA_SIZE, 0) > 0) && (amt != 0))
    amt = fwrite(recvbuf, 1, DATA_SIZE, tmp);
}
```

race condition!

- API functions to create a temporary filename
  - C library: *tmpnam, tempnam, mktemp*
  - C++: *_tempnam, _tempnam, _mktemp*
  - Windows API: *GetTempFileName*
- They create a unique name when called
  - But no guarantee that an attacker doesn't create the same name before the filename is used
  - Name often isn't very random: high chance of attacker constructing it

From https://www.owasp.org/index.php/Insecure_Temporary_File

104

mktemp is also affected by this race condition

If an attacker creates that file first:

- Access permissions may remain unchanged for the attacker
  - Attacker may access the file later and read its contents
- Legitimate code may append content, leaving attacker's content in place
  - Which may be read later as legitimate content
- Attacker may create the file as a link to an important file
  - The application may end up corrupting that file
- The attacker may be smart and call *open* with `O_CREAT | O_EXCL`
  - Or, in Windows: CreateFile with the `CREATE_NEW` attribute
  - Create a new file with exclusive access
  - But if the attacker creates a file with that name, the *open* will fail
    - Now we have *denial of service* attack

From https://www.owasp.org/index.php/Insecure_Temporary_File

105

Defense against mktemp attacks

Use *mkstemp*

- It will attempt to create & open a unique file
- You supply a template
  A name of your choosing with `XXXXXX` that will be replaced to make the name unique

```
mkstemp("/tmp/secretfileXXXXXX")
```

- File is opened with mode 0600: `r-- --- ---`
- If unable to create a file, it will fail and return -1
  - You should test for failure and be prepared to work around it.

106

The main problem: ***interaction***

- To increase security, a program must minimize interactions with the outside
  - Users, files, sockets
- All interactions may be attack targets
- Must be controlled, inspected, monitored

107

The end

108