Buffer overflow

Top Software Weaknesses for 2020

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

https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

What is buffer overflow

- Buffer
  - stack, heap, data
- Overflow
  - Adjacent memory is overwritten
  - Program data can be modified
  - Code can be injected

- Buffer overflow
  - Bug allows more data to be stored in an array than planned
  - Used to be responsible for ~50% of vulnerabilities

Example buffer overflow

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.

- A zero-day vulnerability
  - Attackers exploited the bug before the company could patch it

Simple Example

```
char buf[80];
void vulnerable() {
    gets(buf);
}
```

- gets() reads all input bytes available on stdin, and stores them into buf[]
- What if input has more than 80 bytes?
  - gets() writes past end of buf, overwriting some other part of memory
  - This is a bug!
- Results?
  - Program crash/core-dump?
  - Much worse consequences possible...

Modified Example

```
char buf[80];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

- A login routine sets authenticated flag only if user enters password
- What's the risk?
  - authenticated stored immediately after buf
  - Attacker writes data after end of buf
- Attacker supplies 81 bytes (81st set non-zero)
  - Makes authenticated flag true!
  - Attacker gains access: security breach!

More Serious Exploit Example

```
char buf[80];
int (*fnptr)();
 void vulnerable() {
    gets(buf);
}
```

- Function pointer fnptr invoked elsewhere
- What can attacker do?
  - Can overwrite fnptr with any address
    - redirecting program execution!
  - Input contains malicious machine instructions, followed by pointer to overwrite fnptr
  - When fnptris next invoked, flow of control re-directed to malicious code
- This is a malicious code injection attack

Buffer overflow vulnerabilities

- 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
- Not a problem with strongly typed languages
  - Java, C#, Python, etc. check sizes of structures

- Overflow can occur when programs do not validate the length of data being written to a buffer
  - strcpy(char *dest, constchar *src);
  - strcat(char *dest, constchar *src);
  - gets(char *s);
  - scanf(constchar *format, ...)

Stack Smashing

```
void vulnerable() {
    char buf[80];
    gets(buf);
}
```

- When vulnerable() is called, stack frame is pushed onto stack
- If input is too-long, saved SP and return addr will be overwritten

- First, attacker stashes malicious code sequence somewhere
- Next, attacker provides carefully-chosen 88-byte sequence
  - Last 4 bytes chosen to hold code's address overwrite saved return address
- When vulnerable() returns, CPU loads attacker's return addr
  - handing control over to attacker's malicious code

Format string vulnerabilities

```
void vulnerable() {
    char buf[80];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

- Printf attack
  - %08x.%08x.%08x.%08x.%08x6d10c308.6d10c320.85d636f0.a1b80d80.a1b80d80

Integer conversion vulnerabilities

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

- If the attacker provides a negative value for len
  - memcpy() will copy a huge amount of memory into buf
- A signed/unsigned or an implicit casting bug
  - hard to spot
  - C compiler doesn't warn about type mismatch between signed int and unsigned int

Integer conversion vulnerabilities

```
typedef unsigned int size_t;
void vulnerable() {
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len+5);
    read(fd, buf, len);
    ...
}
```

- What's wrong with this code?
  - No buffer overrun problems (5 spare bytes)
  - No sign problems (all ints are unsigned)
- But, len+5 can overflow if len is too large
  - If len = 0xFFFFFFFF, then len+5 is 4
  - Allocate 4-byte buffer then read a lot more than 4 bytes
    - buffer overflow!

Defenses & Countermeasures

- Type safe languages (Java, C#, Python). DO NOT use C/C++
  - Legacy Code
  - Practical?
- Secure coding
  - Avoid risky programming constructs
    - Use fgets instead of gets
    - Use strn* APIs instead of str* APIs
    - Use snprintf instead of sprintf and vsprintf
    - scanf & printf: use format strings
  - Never assume anything about inputs

Defenses & Countermeasures

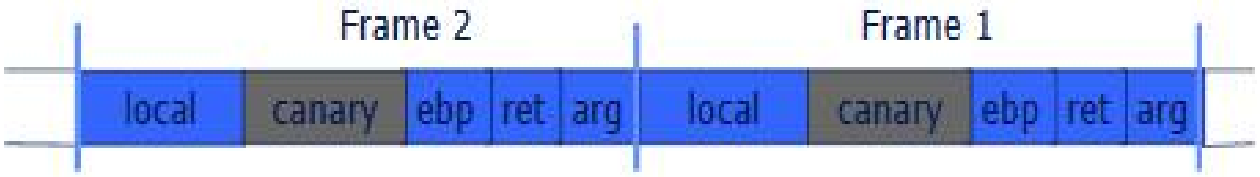
- Mark stack as non-execute
- Run time check for memory safety
- Run time overflow detection
  - Stackguard
- Attack mitigation
  - Randomization techniques

Marking stack as non-executable

- Mark stack segment as non-executable
  - Code patches exist for Linux and Solaris
- Problems
  - Does not block more general overflow exploits
    - Overflow on heap: overflow buffer next to func pointer.
  - Some apps need executable stack (e.g. LISP interpreters)

Run time detection: StackGuard

- StackGuard
  - Run time tests for stack integrity
  - Embed “canaries” in stack frames and verify their integrity prior to function return



- Random canary
  - Choose random string at program startup
  - Insert canary string into every stack frame
  - Verify canary before returning from function
  - To corrupt random canary, attacker must learn the random string

Randomization Techniques

- For successful exploit, the attacker needs to know
  - Stack layout for stack smashing attacks
  - Heap layout for code injection in heap
  - Shared library entry points for exploits using shared library
- Randomization Techniques for Software Security
  - Randomize system internal details
    - Memory layout
    - Internal interfaces
  - Improve software system security
    - Reduce attacker knowledge of system detail to thwart exploit

Randomization Techniques

- Handle a variety of memory safety vulnerabilities
  - Buffer overruns
  - Format string vulnerabilities
  - Integer overflow
- Simple & Efficient
  - Low performance overhead

- Problems
  - Attacks can still happen
    - Overwrite data
    - May crash the program
  - Attacks may learn the randomization secret
    - Format string attacks