OWASP
Open Web Application Security Project

Common software vulnerabilities: causes and consequences

Ricardo J. Rodríguez – CUD
rjrodriguez@unizar.es – @RicardoJRdez

14 de marzo, 2019

I Jornadas OWASP – ZGZ

$whoami

- **Ph.D. in Computer Sciences** (University of Zaragoza, 2013)
- **Professor in Centro Universitario de la Defensa**, Academia General Militar (Zaragoza)
- Research interests:
  - Performance/dependability/security analysis
  - Model-driven engineering (considering security aspects)
  - Program binary analysis (specially, malware analysis)
  - RFID/NFC security
- Not prosecuted (*yet*) ⌣̈
- Speaker in NcN, HackLU, RootedCON, STIC CCN-CERT, HIP, MalCON, HITB...

Agenda

1. Introduction
2. Common Software Vulnerabilities
3. Conclusions

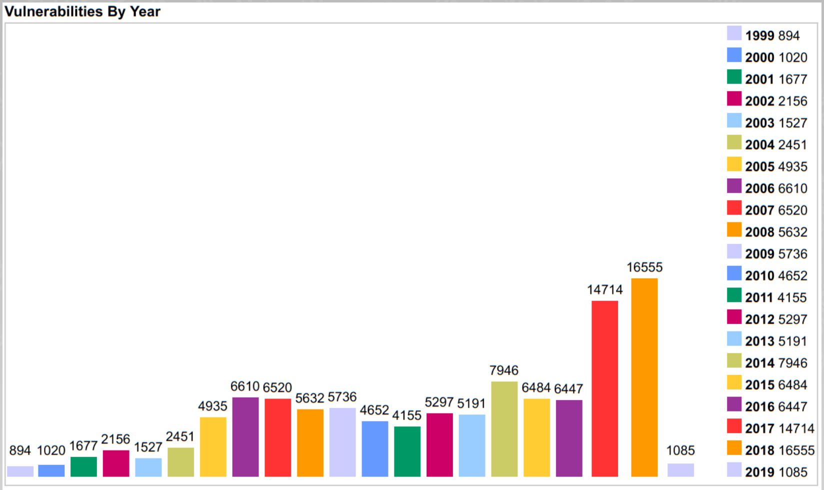
Agenda

1 Introduction

2 Common Software Vulnerabilities

3

Introduction



Credits: `https://www.cvedetails.com/browse-by-date.php`

Introduction

Some definitions of interest

- **Vulnerability**: software flaw
  - An attacker can take advantage of a vulnerability and exploit it
- Average occurrence of faults per Lines of Code (**defect density**)
  - Usually, it depends on the particular software company (different development cycles)

(a) Industry Average: "about 15 - 50 errors per 1000 lines of delivered code." He further says this is usually representative of code that has some level of structured programming behind it, but probably includes a mix of coding techniques.

(b) Microsoft Applications: "about 10 - 20 defects per 1000 lines of code during in-house testing, and 0.5 defect per KLOC (KLOC IS CALLED AS 1000 lines of c ode) in released product (Moore 1992)." He attributes this to a combination of code-reading techniques and independent testing (discussed further in another chapter of his book).

(c) "Harlan Mills pioneered 'cleanroom development', a technique that has been able to achieve rates as low as 3 defects per 1000 lines of code during in-house testing and 0.1 defect per 1000 lines of code in released product (Cobb and Mills 1990). A few projects - for example, the space-shuttle software - have achieved a level of 0 defects in 500,000 lines of code using a system of format development methods, peer reviews, and statistical testing."

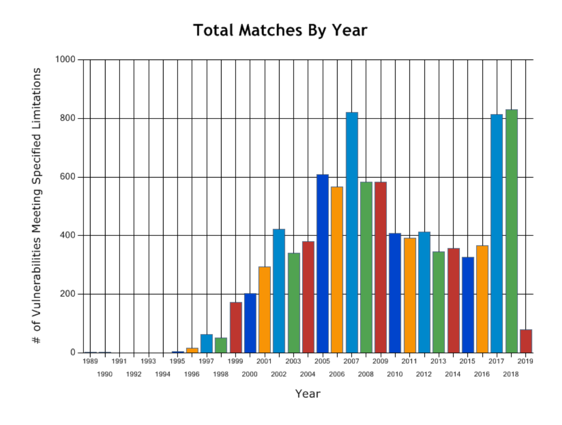
Credits: https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670

Agenda

1. Introduction
2. **Common Software Vulnerabilities**
3.

Common Software Vulnerabilities

Buffer Overflow



Credits: `https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&query=buffer+overflow&queryType=phrase&search_type=all`

Common Software Vulnerabilities

Buffer Overflow

- Also called *buffer overrun*
- **Most prevalent error in C/C++ programs**
- First BOF exploited: **Morris worm** (1988)
  - (BSD-derived) UNIX `fingerd` daemon
  - For curious readers: doi: 10.1145/66093.66095
- **Seminal work of Aleph One in 1996**
  - *Smashing the stack for fun and profit*, Phrack, 7(49), 1996
  - `http://phrack.org/issues/49/14.html`
- Caused when a buffer is overwritten beyond its boundaries
- **Unsafe functions DO NOT check the buffer limits when operating**, then provoking the buffer is overwritten beyond its boundaries
  - Examples of unsafe functions: `gets`, `scanf`, `strcpy`, `strcat`, `sprintf`, ...

Common Software Vulnerabilities

Buffer Overflow

- We can distinguish two kind of buffer overflows:
  - **Stack-based BOF** (`https://cwe.mitre.org/data/definitions/121.html`)
  - **Heap-based BOF** (`https://cwe.mitre.org/data/definitions/122.html`)

Common Software Vulnerabilities

Buffer Overflow

- We can distinguish two kind of buffer overflows:
  - **Stack-based BOF** (`https://cwe.mitre.org/data/definitions/121.html`)
  - **Heap-based BOF** (`https://cwe.mitre.org/data/definitions/122.html`)
- **Which elements are stored in these memory segments?**
  - Stack: stores function parameters, **local variables**, and **caller return address**
  - Heap: dynamic memory (memory allocated by the program – also objects)

Common Software Vulnerabilities

Buffer Overflow

- We can distinguish two kind of buffer overflows:
  - **Stack-based BOF** (`https://cwe.mitre.org/data/definitions/121.html`)
  - **Heap-based BOF** (`https://cwe.mitre.org/data/definitions/122.html`)
- **Which elements are stored in these memory segments?**
  - Stack: stores function parameters, **local variables**, and **caller return address**
  - Heap: dynamic memory (memory allocated by the program – also objects)

Consequences

- **Denial-of-Service** (crashes and resource consumption)
- **Execution of unauthorized code** (or commands)
- **Bypassing of protection mechanisms**
- **Others**

Common Software Vulnerabilities

Buffer Overflow – example + demo

```c
// vuln1.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFLEN 256

void secret()
{
    printf("YOU WIN!\n");
}

void copy_arg(char *s)
{
    char buffer[BUFLEN];

    strcpy(buffer, s);
    printf("Your argument is: %s\n", buffer);
}

int main(int argc, char *argv[])
{
    if(argc != 2){
        fprintf(stderr, "usage error: %s string - echoes
            string argument\n", argv[0]);
        return EXIT_FAILURE;
    }
    copy_arg(argv[1]);

    return EXIT_SUCCESS;
}
```

Common Software Vulnerabilities

Buffer Overflow – example + demo

```
1  // vuln1.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #define BUFLEN 256
7
8  void secret()
9  {
10     printf("YOU WIN!\n");
11 }
12
13 void copy_arg(char *s)
14 {
15     char buffer[BUFLEN];
16
17     strcpy(buffer, s);
18     printf("Your argument is: %s\n", buffer);
19 }
20
21 int main(int argc, char *argv[])
22 {
23     if(argc != 2){
24         fprintf(stderr, "usage error: %s string - echoes
               string argument\n", argv[0]);
25         return EXIT_FAILURE;
26     }
27     copy_arg(argv[1]);
28
29     return EXIT_SUCCESS;
30 }
```

- **L17: `strcpy` is an unsafe function**
  - **Does not check the length** of `buffer`: just copies each byte of `s` to `buffer` until the string terminator (`NULL` character) is reached
  - When size of `s` is greater than `BUFLEN`, the adjacent memory to `buffer` is overwritten
  - What elements were stored in the stack, apart from local variables (such as `buffer`)?

Common Software Vulnerabilities

Buffer Overflow – example + demo

```
1  // vuln1.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #define BUFLEN 256
7
8  void secret()
9  {
10     printf("YOU WIN!\n");
11 }
12
13 void copy_arg(char *s)
14 {
15     char buffer[BUFLEN];
16
17     strcpy(buffer, s);
18     printf("Your argument is: %s\n", buffer);
19 }
20
21 int main(int argc, char *argv[])
22 {
23     if(argc != 2){
24         fprintf(stderr, "usage error: %s string - echoes
               string argument\n", argv[0]);
25         return EXIT_FAILURE;
26     }
27     copy_arg(argv[1]);
28
29     return EXIT_SUCCESS;
30 }
```

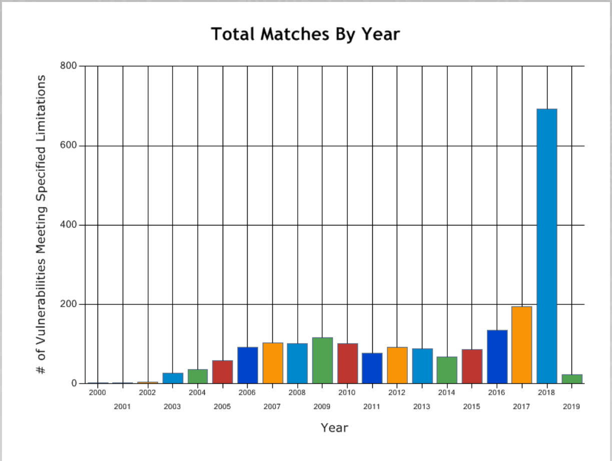
- **L17: `strcpy` is an unsafe function**
  - **Does not check the length** of `buffer`: just copies each byte of `s` to `buffer` until the string terminator (`NULL` character) is reached
  - When size of `s` is greater than `BUFLEN`, the adjacent memory to `buffer` is overwritten
  - What elements were stored in the stack, apart from local variables (such as `buffer`)?

BINGO: return address to `main`

(*let's see a demo about hijacking the program control-flow*)

Common Software Vulnerabilities

Numerical Issues



Credits: https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&query=integer+overflow&queryType=phrase&search_type=all

Common Software Vulnerabilities

Numerical Issues

- **Integer numerical errors**
  - **Overflows**: when the result of an integer expression exceeds the maximum value for its respective type
  - **Underflows**: when the result of an integer expression is smaller than its minimum value, it wraps to the maximum integer for the type. For instance, subtracting 0 – 1 and storing the result in an unsigned 16-bit integer
  - **Signedness error**: when a signed integer is interpreted as unsigned, or vice-versa
  - **Lossy truncations**: when assigning an integer with a larger width to a smaller width
- **Costly and exploitable bugs**
  - Reported in the **top 25 most dangerous software errors** (MITRE 2011)

Common Software Vulnerabilities

Numerical Issues

- **Integer numerical errors**
  - **Overflows**: when the result of an integer expression exceeds the maximum value for its respective type
  - **Underflows**: when the result of an integer expression is smaller than its minimum value, it wraps to the maximum integer for the type. For instance, subtracting 0 – 1 and storing the result in an unsigned 16-bit integer
  - **Signedness error**: when a signed integer is interpreted as unsigned, or vice-versa
  - **Lossy truncations**: when assigning an integer with a larger width to a smaller width
- **Costly and exploitable bugs**
  - Reported in the **top 25 most dangerous software errors** (MITRE 2011)

Consequences

- **Denial-of-Service** (crashes and resource consumption)
- **Execution of unauthorized code** (or commands)
- **Bypassing of protection mechanisms**
- **Logic errors**

Common Software Vulnerabilities

Numerical Issues – example + demo

```c
// vuln2.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXLEN 32 // max passwd length

void store_passwd_indb(char* passwd)
{
    if(passwd != NULL)
    {
        // do stuff...
    }
}

void validate_uname(char* uname)
{
    // do more stuff...
}

void validate_passwd(char* passwd) {
    char passwd_buf[MAXLEN];
    unsigned char passwd_len = strlen(passwd);

    // zeroes the buffer
    bzero(passwd_buf, sizeof(passwd_buf));

    // check length
    if(passwd_len >= 8 && passwd_len <= MAXLEN){
        printf("Valid password\n");
        strcpy(passwd_buf,passwd);
    }else
        printf("Invalid password\n");

    // store it into the DB
    store_passwd_indb(passwd_buf);
}

int main(int argc, char* argv[]) {
    if(argc != 3) {
        printf("usage error: %s username passwd\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    validate_uname(argv[1]);
    validate_passwd(argv[2]);

    return EXIT_SUCCESS;
}
```

Open Web Application Security Project

Common Software Vulnerabilities

Numerical Issues – example + demo

```c
1  // vuln2.c
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5
6  #define MAXLEN 32 // max passwd length
7
8  void store_passwd_indb(char* passwd)
9  {
10     if(passwd != NULL)
11     {
12         // do stuff...
13     }
14 }
15
16 void validate_uname(char* uname)
17 {
18     // do more stuff...
19 }
20
21 void validate_passwd(char* passwd) {
22     char passwd_buf[MAXLEN];
23     unsigned char passwd_len = strlen(passwd);
24
25     // zeroes the buffer
26     bzero(passwd_buf, sizeof(passwd_buf));
27
28     // check length
29     if(passwd_len >= 8 && passwd_len <= MAXLEN){
30         printf("Valid password\n");
31         strcpy(passwd_buf,passwd);
32     }else
33         printf("Invalid password\n");
34
35     // store it into the DB
36     store_passwd_indb(passwd_buf);
37 }
38
39 int main(int argc, char* argv[]) {
40     if(argc != 3) {
41         printf("usage error: %s username passwd\n", argv[0]);
42         exit(EXIT_FAILURE);
43     }
44     validate_uname(argv[1]);
45     validate_passwd(argv[2]);
46
47     return EXIT_SUCCESS;
48 }
```

- **L23: `passwd_buf` can overflow**
  - `man strlen`:
    `size_t strlen(const char *s);`
- If we provide the program with a crafted input string as password string, **we can control the overflow, bypass the length checking, and reach the unsafe `strcpy`**
- Once reached, **we can easily exploit it** (stack-based buffer overflow)

Common Software Vulnerabilities

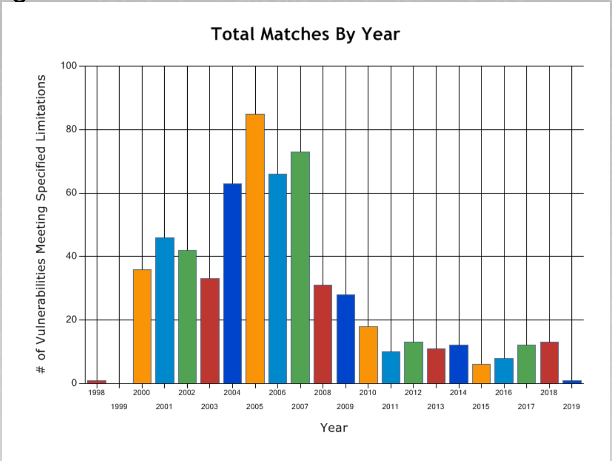
Numerical Issues – example + demo

```
1  // vuln2.c
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5
6  #define MAXLEN 32 // max passwd length
7
8  void store_passwd_indb(char* passwd)
9  {
10     if(passwd != NULL)
11     {
12         // do stuff...
13     }
14 }
15
16 void validate_uname(char* uname)
17 {
18     // do more stuff...
19 }
20
21 void validate_passwd(char* passwd) {
22     char passwd_buf[MAXLEN];
23     unsigned char passwd_len = strlen(passwd);
24
25     // zeroes the buffer
26     bzero(passwd_buf, sizeof(passwd_buf));
27
28     // check length
29     if(passwd_len >= 8 && passwd_len <= MAXLEN){
30         printf("Valid password\n");
31         strcpy(passwd_buf,passwd);
32     }else
33         printf("Invalid password\n");
34
35     // store it into the DB
36     store_passwd_indb(passwd_buf);
37 }
38
39 int main(int argc, char* argv[]) {
40     if(argc != 3) {
41         printf("usage error: %s username passwd\n", argv[0]);
42         exit(EXIT_FAILURE);
43     }
44     validate_uname(argv[1]);
45     validate_passwd(argv[2]);
46
47     return EXIT_SUCCESS;
48 }
```

- **L23: `passwd_buf` can overflow**
  - `man strlen`:
    `size_t strlen(const char *s);`
- If we provide the program with a crafted input string as password string, **we can control the overflow, bypass the length checking, and reach the unsafe `strcpy`**
- Once reached, **we can easily exploit it** (stack-based buffer overflow)

(let's see again a demo here about hijacking the program control-flow)

Common Software Vulnerabilities
Format String



Credits: https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&query=format+string&queryType=phrase&search_type=all

Common Software Vulnerabilities

Format String

```c
void error(char *s)
{
    fprintf(stderr, s);
}
```

What if *s is equal to "%s %s %s %s %s %s"?

- Program **will crash (most likely)**: Denial-of-Service
- Otherwise, **memory content will be printed**: privacy issues
- "Young" vulnerability
- Considered as a **programming bug** instead of a security threat

Common Software Vulnerabilities

Format String

- **Appears when a user can provide a format string to an ANSI C format function** (in part or as a whole)
  - Potentially vulnerable functions: **any function that gets a format string**
- **Attacker capabilities** – two really important things
  - **Arbitrary memory read**
  - **Arbitrary memory write**

Common Software Vulnerabilities

Format String

- **Appears when a user can provide a format string to an ANSI C format function** (in part or as a whole)
  - Potentially vulnerable functions: **any function that gets a format string**
- **Attacker capabilities** – two really important things
  - **Arbitrary memory read**
  - **Arbitrary memory write**

Consequences

- **Denial-of-Service** (crashes and resource consumption)
- **Execution of unauthorized code** (or commands)
- **Bypassing of protection mechanisms**
- **Logic errors**

Common Software Vulnerabilities

Format String – example + demo

```c
// vuln3.c
#include <stdio.h>

#define BUFLEN 256

void secret()
{
    printf("YOU WIN!\n");
}

void echo()
{
    char buffer[BUFLEN];
    char *buf = 0;

    printf("Type your input: ");
    fgets(buffer, sizeof(buffer), stdin); // reads from stdin securely
    printf("Your input is: ");
    printf(buffer);
    printf("\n");
}

int main()
{
    int x;

    x = 0xBAADF00D;
    printf("Address of x: 0x%08x\n", &x); // mem leak -- just for help

    echo();
    if(x == 0x29A)
        secret();

    return 0;
}
```

Common Software Vulnerabilities

Format String – example + demo

```
1  // vuln3.c
2  #include <stdio.h>
3
4  #define BUFLEN 256
5
6  void secret()
7  {
8      printf("YOU WIN!\n");
9  }
10
11 void echo()
12 {
13     char buffer[BUFLEN];
14     char *buf = 0;
15
16     printf("Type your input: ");
17     fgets(buffer, sizeof(buffer), stdin); // reads from
           stdin securely
18     printf("Your input is: ");
19     printf(buffer);
20     printf("\n");
21 }
22
23 int main()
24 {
25     int x;
26
27     x = 0xBAADF00D;
28     printf("Address of x: 0x%08x\n", &x); // mem leak --
           just for help
29
30     echo();
31     if(x == 0x29A)
32         secret();
33
34     return 0;
35 }
```

- **L19: `printf` is printing the string given by the user**
  - But `printf` is a format function!!
  - If we provide the program with a format string as input, `printf` **interprets it and looks for the values specified by the format specifiers**
- **With a crafted input, we can read and write any memory address**

Common Software Vulnerabilities

Format String – example + demo

```
1  // vuln3.c
2  #include <stdio.h>
3
4  #define BUFLEN 256
5
6  void secret()
7  {
8      printf("YOU WIN!\n");
9  }
10
11 void echo()
12 {
13     char buffer[BUFLEN];
14     char *buf = 0;
15
16     printf("Type your input: ");
17     fgets(buffer, sizeof(buffer), stdin); // reads from stdin securely
18     printf("Your input is: ");
19     printf(buffer);
20     printf("\n");
21 }
22
23 int main()
24 {
25     int x;
26
27     x = 0xBAADF00D;
28     printf("Address of x: 0x%08x\n", &x); // mem leak -- just for help
29
30     echo();
31     if(x == 0x29A)
32         secret();
33
34     return 0;
35 }
```

- **L19: `printf` is printing the string given by the user**
  - But `printf` is a format function!!
  - If we provide the program with a format string as input, `printf` **interprets it and looks for the values specified by the format specifiers**
- **With a crafted input, we can read and write any memory address**

(let's see again a demo here about altering the program control-flow)

Agenda

Conclusions

- **Software flaws are common during software development**
  - Some are critical, some others aren't
  - Secure programming MUST be a mantra for any development team
- **Time-to-market cannot lead the development process**
  - The less time we have to develop, the more defect density our code is likely to have
- **SSoftware vulnerabilities may be the entrance door to the whole company's infrastructure**
  - Some systems are important to be attacker-free (e.g., critical infrastructures, business servers)

- **How to ~~avoid~~ minimize software vulnerabilities?**
  - **Know what vulnerabilities are likely to occur and how/why they are produced** (plus consequences)
  - **Follow guidelines for secure software development** (e.g., CERT C Coding Standard)
  - **Apply mechanisms to mitigate impact of exploitation**
  - **Make a source code auditing** (internal auditing process)
  - If enough budget, **make a security auditing** prior release (external auditing process)

OWASP
Open Web Application Security Project

Common software vulnerabilities: causes and consequences

Ricardo J. Rodríguez – CUD
rjrodriguez@unizar.es – @RicardoJRdez

14 de marzo, 2019

I Jornadas OWASP – ZGZ