SOFTWARE SECURITY LABORATORY
HANYANG UNIVERSITY

Formatted Output

2022년 1학기
한양대학교 컴퓨터소프트웨어학부
임을규

Overview

- Formatted Output
- Variadic Functions
- Formatted Output Functions
- Exploiting Formatted Output Functions
- Stack Randomization
- Mitigation Strategies
- Notable Vulnerabilities
- Summary

Formatted Output

- Formatted output functions consist of a format string and a variable number of arguments.
- The format string provides a set of instructions that are interpreted by the formatted output function.
- By controlling the content of the format string a user can control execution of the formatted output function.
- Because all formatted output functions have capabilities that allow a user to violate a security policy, a vulnerability exists

Formatted Output

- Formatted output functions are *variadic*, meaning that they accept a variable number of arguments.
- Limitations of *variadic function* implementations in C contribute to vulnerabilities in the use of formatted output functions

example

- the usage string is built by substituting the %s in the format string with the runtime value of pname

```c
#include <stdio.h>
#include <string.h>
void usage(char *pname) {
    char usageStr[1024];
    snprintf(usageStr, 1024,
              "Usage: %s <target>\n", pname);
    printf(usageStr);
}
int main(int argc, char * argv[]) {
    if (argc < 2) {
        usage(argv[0]);
        exit(-1);
    }
}
```

printf() is called to output the usage information

the name of the program entered by the user (argv[0]) is passed to usage()

Variadic functions

- Variadic functions are implemented using either the UNIX System V or the ANSI C approach.
- Both approaches require that the contract between the **developer** and **user** of the variadic function not be violated by the user

ANSI C Standard Arguments

- In the ANSI C standard argument approach (stdargs), variadic functions are declared using a partial parameter list followed by the ellipsis notation.
- A function with a variable number of arguments is invoked simply by specifying the desired number of arguments in the function call: average(3, 5, 8, -1).

Implementation of average() function using stdargs

```
int average(int first, ...) {
    int count = 0, sum = 0, i = first;
    va_list marker;
    va_start(marker, first);
    while (i != -1) {
        sum += i;
        count++;
        i = va_arg(marker, int);
    }
    va_end(marker);
    return(sum ? (sum / count) : 0);
}
```

ANSI C Standard Arguments

- The variadic `average()` function accepts a single fixed argument followed by a variable argument list.
  - No type checking is performed on the arguments in the variable list.
  - One or more fixed parameters precedes the ellipsis notation, which must be the last token in the parameter list.
- ANSI C provides the `va_start()`, `va_arg()`, and `va_end()` macros for implementing variadic functions.
  - These macros, which are defined in the `stdarg.h` include file, all operate on the va_list data type, and the argument list is declared using the va_list type.

Sample Definition of Variable Argument Macros

```
#define _ADDRESSOF(v) (&(v))

#define _INTSIZEOF(n) \

   ((sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1))

typedef char *va_list;

#define va_start(ap,v) (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v))

#define va_arg(ap,t) (*(t *)((ap+=_INTSIZEOF(t))-_INTSIZEOF(t)))

#define va_end(ap) (ap = (va_list)0)
```

The marker variable is declared as a va_list type

The va_start() macro initializes the argument list and must be called before marker can be used

va_start() is called on line 4 and passed marker and the last fixed argument (first)

Sample Definition of Variable Argument Macros

- The `va_arg()` macro requires an initialized va_list and the type of the next argument.
  - The macro returns the next argument and increments the argument pointer based on the type size.
- The `va_arg()` macro is invoked on line 8 of the `average()` function to get the second through last arguments.
- Finally, `va_end()` is called before the program returns to cleanup.

Sample Definition of Variable Argument Macros

- The termination condition for the argument list is a contract between the programmers who implement and use the function.
- The `average()` function, termination of the variable argument list is indicated by an argument whose value is -1.
- Without this argument, the `average()` function will continue to process the next argument indefinitely until a -1 value is encountered or an exception occurs

Type `va_list` as a Character Pointer

- The figure illustrates how the arguments are sequentially ordered on the stack when average(3,5,8,-1) function is called on these systems.

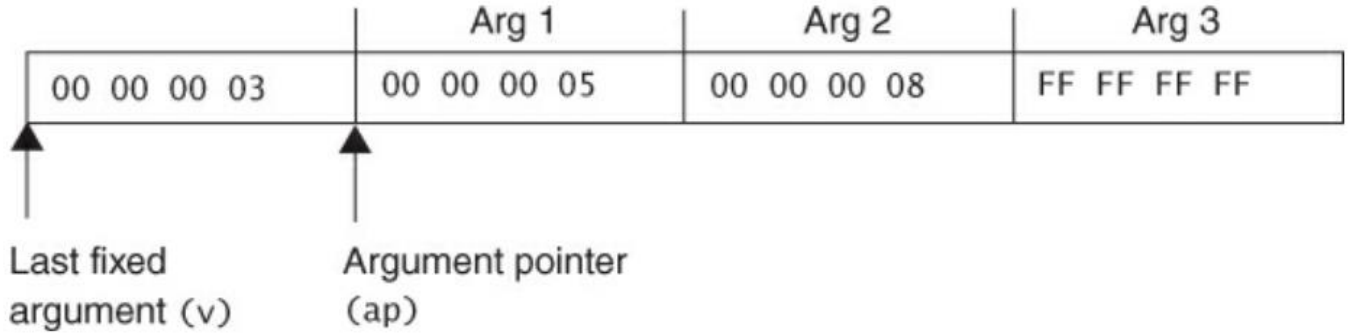


Figure 6.1. Type `va_list` as a character pointer

Type va_list as a Character Pointer

- The character pointer is initialized by `va_start()` to reference the parameters following the last fixed argument.
- The `va_start()` macro adds the size of the argument to the address of the last fixed parameter.
- When `va_start()` returns, va_list points to the address of the first optional argument.

Type va_list as a Character Pointer

- Not all systems define the `va_list` type as a character pointer.
- Some systems define `va_list` as an array of pointers, while other systems pass arguments in registers.
- When arguments are passed in registers, `va_start()` may have to allocate memory to store the arguments.
- The `va_end()` macro is used to free allocated memory

va_arg: average

Example

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

int average (int first, ...) {
        int count = 0, sum = 0, i = first;
        va_list marker;
        va_start(marker, first);
        while (i != -1) {
                sum += i;
                count++;
                i = va_arg(marker, int);
        }
        va_end(marker);
        return (sum ? (sum/count) : 0);
}

int main(void) {
        int result;
        result = average(3, 5, 8, -1);
        printf("1 : %d\n", result);
        result = average(3, 5, 8, 1);
        printf("2 : %d\n", result);
}
```

Formatted Output Functions

- `fprintf()` writes output to a stream based on the contents of the format string.
  - The stream, format string, and a variable list of arguments are provided as arguments.
- `printf()` is equivalent to `fprintf()` except that `printf()` assumes that the output stream is `stdout`.
- `sprintf()` is equivalent to `fprintf()` except that the output is written into an array rather than to a stream.
- `snprintf()` is equivalent to `sprintf()` except that the maximum number of characters n to write is specified.
  - If n is non-zero, output characters beyond n-1st are discarded rather than written to the array, and a null character is added at the end of the characters written into the array.

Equivalent Functions

- vfprintf()
- vprintf()
- vsprintf()
- vsnprintf()
- fprintf()
- printf()
- sprintf()
- snprintf()
- These functions are useful when the argument list is determined at runtime

syslog()

- **`syslog()`**: Formatted output function not defined by the C99 specification but included in SUSv2 (Single Unix Specification version 2).
  - It generates a log message to the system logger (`syslogd`) and accepts a priority argument, a format specification, and any arguments required by the format.
- The `syslog()` function:
  - first appeared in BSD 4.2,
  - supported by Linux and UNIX,
  - supported by Windows systems.

Format Strings

- Format strings are character sequences consisting of
  - ordinary characters (excluding %)
  - and conversion specifications.
- *Ordinary characters* are copied unchanged to the output stream.
- *Conversion specifications* convert arguments according to a corresponding conversion specifier, and write the results to the output stream.
  - Conversion specifications begin with a percent sign (%) and are interpreted from left to right.

Format Strings

- If there are **more arguments** than conversion specifications, the extra arguments are ignored.
- If there are **not enough arguments** for all the conversion specifications, the results are undefined.
- A conversion specification consists of:
  - optional field:
    - flags, width, precision, and length-modifier,
  - required fields:
    - conversion-specifier in the following form:
    - %[flags] [width] [.precision] [{length-modifier}] conversion-specifier.

Format Strings

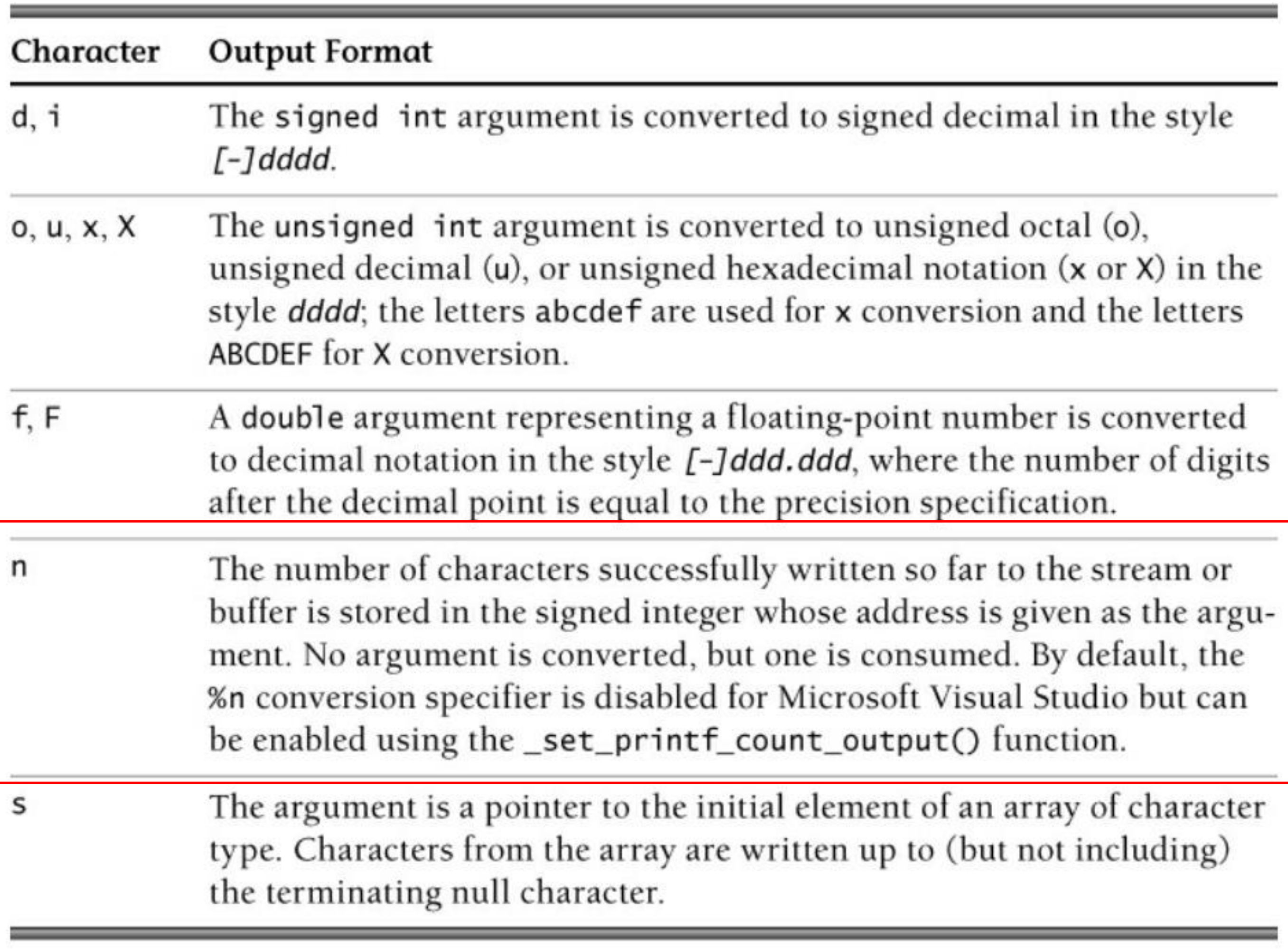
- Example `%-10.8ld`:
  - - is a flag,
  - 10 is the width,
  - 8 is the precision,
  - the letter `l` is a length modifier,
  - `d` is the conversion specifier.
- This conversion specification prints a long int argument in decimal notation, with a minimum of 8 digits left justified in a field at least 10 characters wide.
- The simplest conversion specification contains only % and a conversion specifier (for example, %s).

Conversion Specifier

- Indicates the type of conversion to be applied.
- Is the only required format field, and it appears after any optional format fields.
- Flags justify output and print signs, blanks, decimal points, and octal and hexadecimal prefixes.
- More than one flag directive may appear in a format specification.

Conversion Specifier

Table 6.1. Conversion Specifiers



Character	Output Format
`d`, `i`	The `signed int` argument is converted to signed decimal in the style *[-]dddd*.
`o`, `u`, `x`, `X`	The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion.
`f`, `F`	A `double` argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal point is equal to the precision specification.
`n`	The number of characters successfully written so far to the stream or buffer is stored in the signed integer whose address is given as the argument. No argument is converted, but one is consumed. By default, the `%n` conversion specifier is disabled for Microsoft Visual Studio but can be enabled using the `_set_printf_count_output()` function.
`s`	The argument is a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character.

Width

- Specifies the minimum number of characters to output.
  - If the number of characters output is less than the specified width, the width is padded with blank characters.
- A small width does not cause field truncation.
  - If the result of a conversion is wider than the field width, the field expands to contain the conversion result.
- If the width specification is an asterisk (*), an `int` argument from the argument list supplies the value.
- In the argument list, the width argument must precede the value being formatted.

Precision

- Precision specifies the number of characters to be printed, the number of decimal places, or the number of significant digits.
- The precision field can cause truncation of the output or rounding of a floating-point value.
- If the precision field is an asterisk (*), the value is supplied by an int argument from the argument list.
- In the argument list, the precision argument must precede the value being formatted.

Limits

- Formatted output functions in GCC version 3.2.2 handle width and precision fields up to INT_MAX (2,147,483,647 on IA-32).
- Formatted output functions also keep and return a count of characters outputted as an int.
- This count continues to increment even if it exceeds INT_MAX, which results in a signed integer overflow and a signed negative number.
- If interpreted as an unsigned number, the count is accurate until an unsigned overflow occurs

Length Modifier

- Visual C++ does not support the C99’s h, hh, l, and ll length modifiers.
- It provides an I32 length modifier that behaves the same as the l length modifier and an I64 length modifier that approximates the ll length modifier.
- I64 prints the full value of a long long int but only writes 32 bits when used with the n conversion specifier

Length Modifier

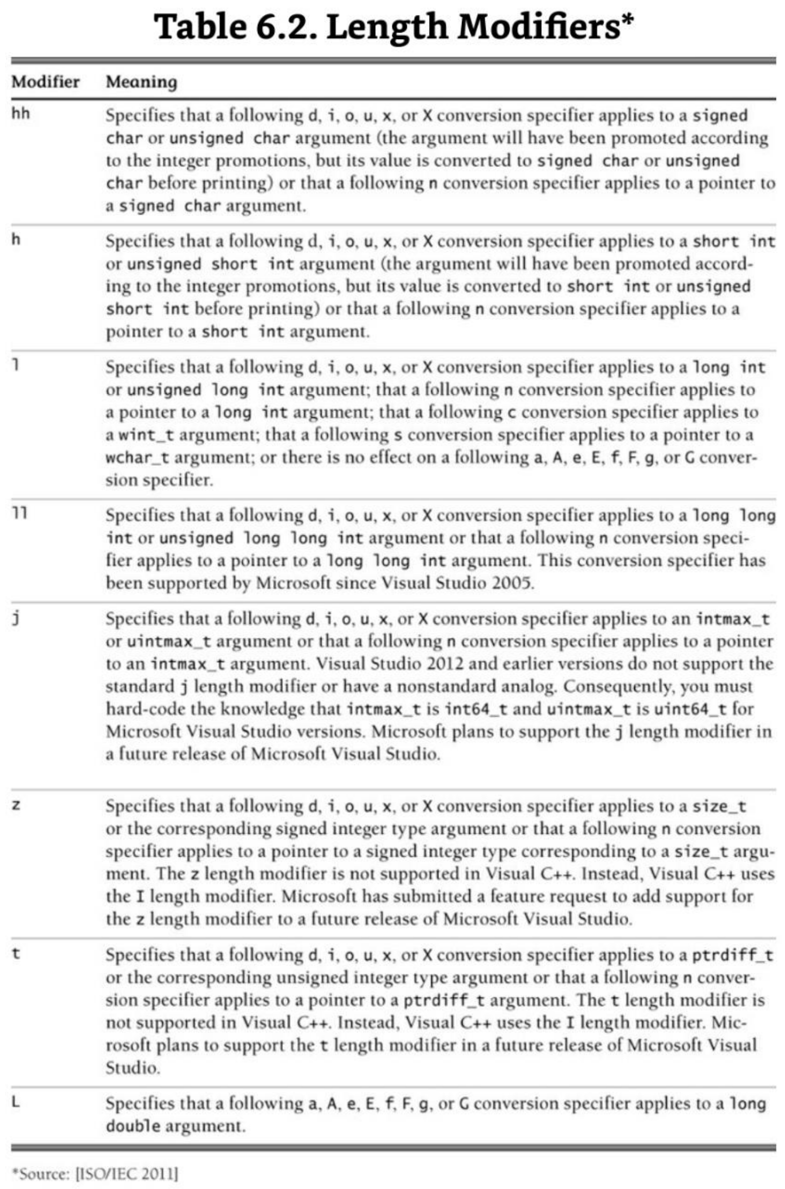


Table 6.2. Length Modifiers*

Modifier	Meaning
`hh`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value is converted to `signed char` or `unsigned char` before printing) or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.
`h`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integer promotions, but its value is converted to `short int` or `unsigned short int` before printing) or that a following `n` conversion specifier applies to a pointer to a `short int` argument.
`l`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; that a following `n` conversion specifier applies to a pointer to a `long int` argument; that a following `c` conversion specifier applies to a `wint_t` argument; that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or there is no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.
`ll`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long int` or `unsigned long long int` argument or that a following `n` conversion specifier applies to a pointer to a `long long int` argument. This conversion specifier has been supported by Microsoft since Visual Studio 2005.
`j`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument. Visual Studio 2012 and earlier versions do not support the standard `j` length modifier or have a nonstandard analog. Consequently, you must hard-code the knowledge that `intmax_t` is `int64_t` and `uintmax_t` is `uint64_t` for Microsoft Visual Studio versions. Microsoft plans to support the `j` length modifier in a future release of Microsoft Visual Studio.
`z`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to a `size_t` argument. The `z` length modifier is not supported in Visual C++. Instead, Visual C++ uses the `I` length modifier. Microsoft has submitted a feature request to add support for the `z` length modifier to a future release of Microsoft Visual Studio.
`t`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument. The `t` length modifier is not supported in Visual C++. Instead, Visual C++ uses the `I` length modifier. Microsoft plans to support the `t` length modifier in a future release of Microsoft Visual Studio.
`L`	Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

*Source: [ISO/IEC 2011]

Exploiting Formatted Output Functions

- Formatted output became important to the security community when a format string vulnerability was discovered in WU-FTP.
- Format string vulnerabilities can occur when a format string is supplied by a user or other untrusted source.
- Buffer overflows can occur when a formatted output routine writes beyond the boundaries of a data structure.

Buffer Overflow

- Formatted output functions that write to a character array assume arbitrarily long buffers, which makes them susceptible to buffer overflows.
  - 1. char buffer[512];
  - 2. sprintf(buffer, "Wrong command: %s\n", user)
- buffer overflow vulnerability using `sprintf()` as it substitutes the %s conversion specifier with a user-supplied string.
- Any string longer than 495 bytes results in an out-of-bounds write (512 bytes - 16 character bytes - 1 null byte)

Stretchable buffer

- The `sprintf()` call cannot be directly exploited because the %.400s conversion specifier limits the number of bytes written to 400.

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
4. sprintf(outbuf, buffer);
```

Stretchable buffer

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
4. sprintf(outbuf, buffer);
```

- This same call can be used to indirectly attack the sprintf() call providing the following value for user:
  - %497d\x3c\xd3\xff\xbf<nops><shellcode>
- The sprintf() call on line 3 inserts this string into buffer.
  - %497d\x3c\xd3\xff\xbf<nops><shellcode>
- The buffer array is then passed to the second call to sprintf() as the format string argument.
  - %497d\x3c\xd3\xff\xbf<nops><shellcode>

Stretchable buffer

- The %497d format instructs sprintf() to read an imaginary argument from the stack and write 497 characters to buffer.
- The total number of characters written now exceeds the length of outbuf by four bytes.
- The user input can be manipulated to overwrite the return address with the address of the exploit code supplied in the malicious format string argument (0xbfffd33c).
- When the current function exits, control is transferred to the exploit code in the same manner as a stack smashing attack

Stretchable buffer

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
4. sprintf(outbuf, buffer);
```

- The programming flaw is that `sprintf()` is being used inappropriately on line 4 as a string copy function when `strcpy()` or `strncpy()` should be used instead
- Replacing this call to `sprintf()` with a call to `strcpy()` eliminates the vulnerability

Sprintf & printf

```
#include <stdio.h>
#define BUFSIZE  64

int main(int argc, char *argv[]) {
        char outbuf[BUFSIZE];
        char buf[BUFSIZE];

        sprintf(buf, "ERR Wrong command: %s", argv[1]);
        printf(buf);
}
```

```
#include <stdio.h>
#define BUFSIZE  64

int main(int argc, char *argv[]) {
        char outbuf[BUFSIZE];
        char buf[BUFSIZE];

        sprintf(buf, "ERR Wrong command: %.40s", argv[1]);
        printf(buf);
}
```

Output Streams

- Formatted output functions that write to a stream instead of a file (such as `printf()`) are also susceptible to format string vulnerabilities:
  - 1. int func(char *user) {
  - 2.   printf(user);
  - 3. }
- If the user argument can be controlled by a user, this program can be exploited to crash the program, view the contents of the stack, view memory content, or overwrite memory.

Crashing a Program

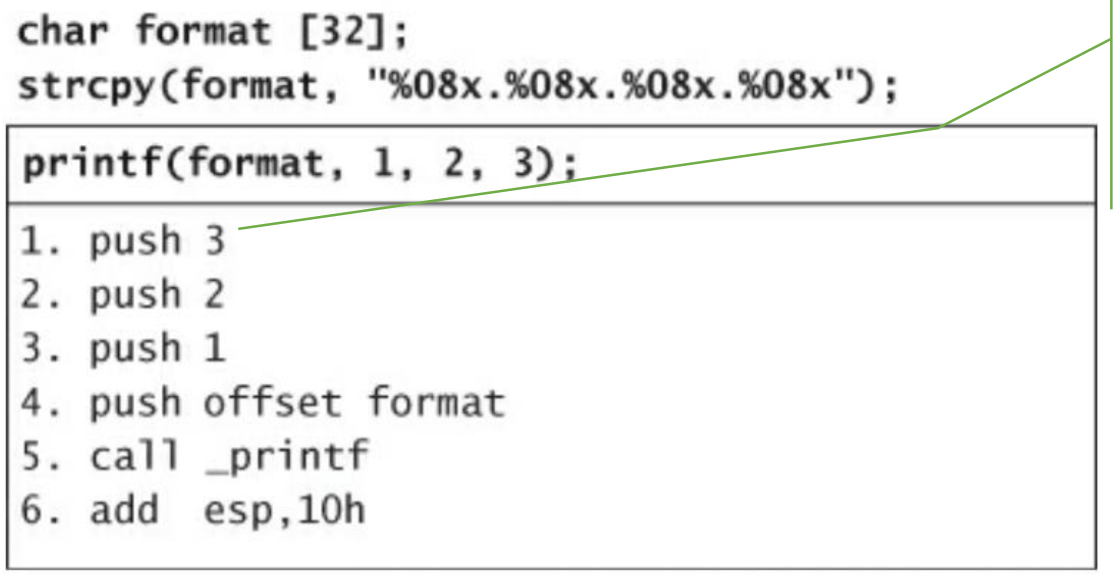
- Format string vulnerabilities are discovered when a program crashes.
- For most UNIX systems, an invalid pointer access causes a SIGSEGV signal to the process.
- Unless caught and handled, the program will abnormally terminate and dump core.
- Similarly, an attempt to read an unmapped address in Windows results in a general protection fault followed by abnormal program termination

Crashing a Program

- An invalid pointer access or unmapped address read can be triggered by calling a formatted output function:
  - printf("%s%s%s%s%s%s%s%s%s%s%s%s");
- The %s conversion specifier displays memory at an address specified in the corresponding argument on the execution stack.
- Because no string arguments are supplied in this example, printf() reads arbitrary memory locations from the stack until the format string is exhausted or an invalid pointer or unmapped address is encountered

Viewing Stack Content

- Attackers can also exploit formatted output functions to examine the contents of memory.
- Disassembled printf() call



```
char format [32];
strcpy(format, "%08x.%08x.%08x.%08x");
```

```
printf(format, 1, 2, 3);
```

```
push 3
push 2
push 1
push offset format
call _printf
add  esp,10h
```

Arguments are pushed onto the stack in reverse order

Figure 6.2. Disassembled **printf()** call

Example – format string

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

int
main(int argc, char *argv[])
{
        int j = 16;
        int wi = 0;

        if (argc < 2)   exit(-1);

        wi = atoi(argv[1]);
        printf("1: %#*x \n", wi, j++);
        printf("2: %#*X \n", wi, j++);
        printf("3: %#*o \n", wi, j++);
        printf("4: %#.*x \n", wi, j++);
        printf("5: %#.*X \n", wi, j++);
        printf("6: %#.*o \n", wi, j++);
        printf("7: %#.*d \n", wi, j++);
}
```

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

void f(void) {
        char format[32];
        strcpy(format,
           "%08x.%08x.%08x.%08x.%08x.%08x");
        printf(format, 1, 2, 3);
}

int main(void) {
        f();
}
```

Viewing the Contents of the Stack

- The address of the format string 0xe0f84201 appears in memory followed by the argument values 1, 2, and 3

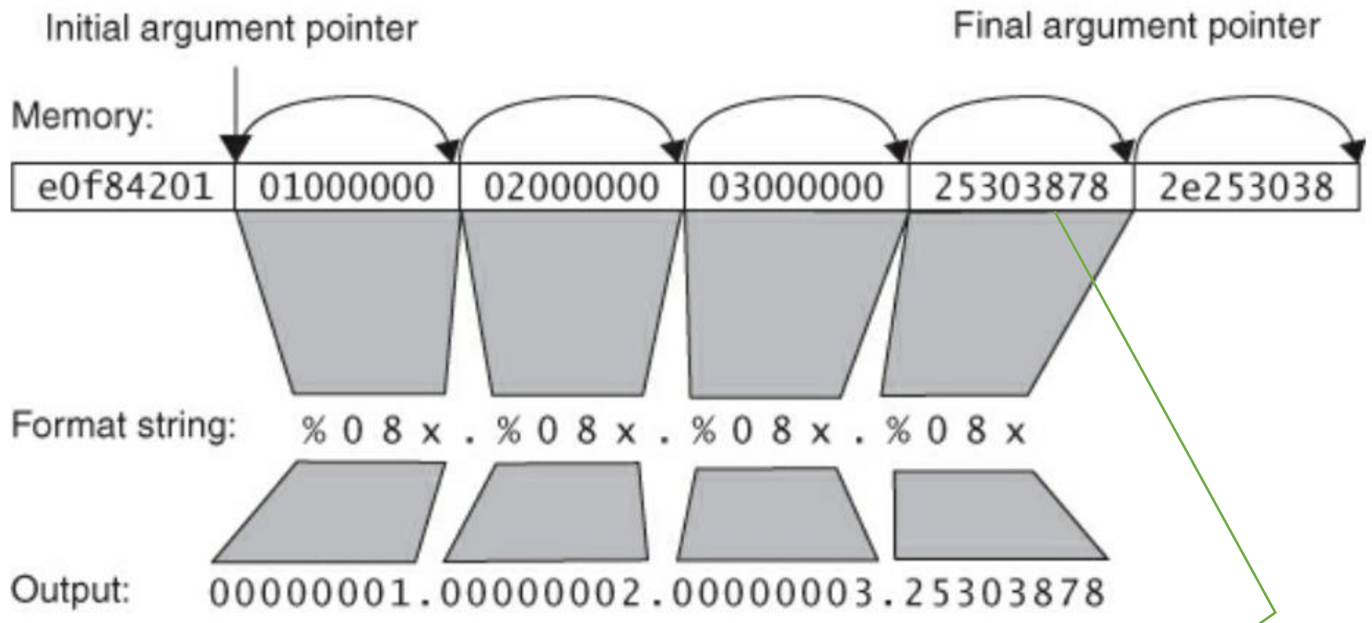


Figure 6.3. Viewing the contents of the stack

The memory immediately following the arguments contains the automatic variables for the calling function, including the contents of the format character array 0x2e253038

Viewing the Contents of the Stack

- The address of the format string 0xe0f84201 appears in memory followed by the argument values 1, 2, and 3

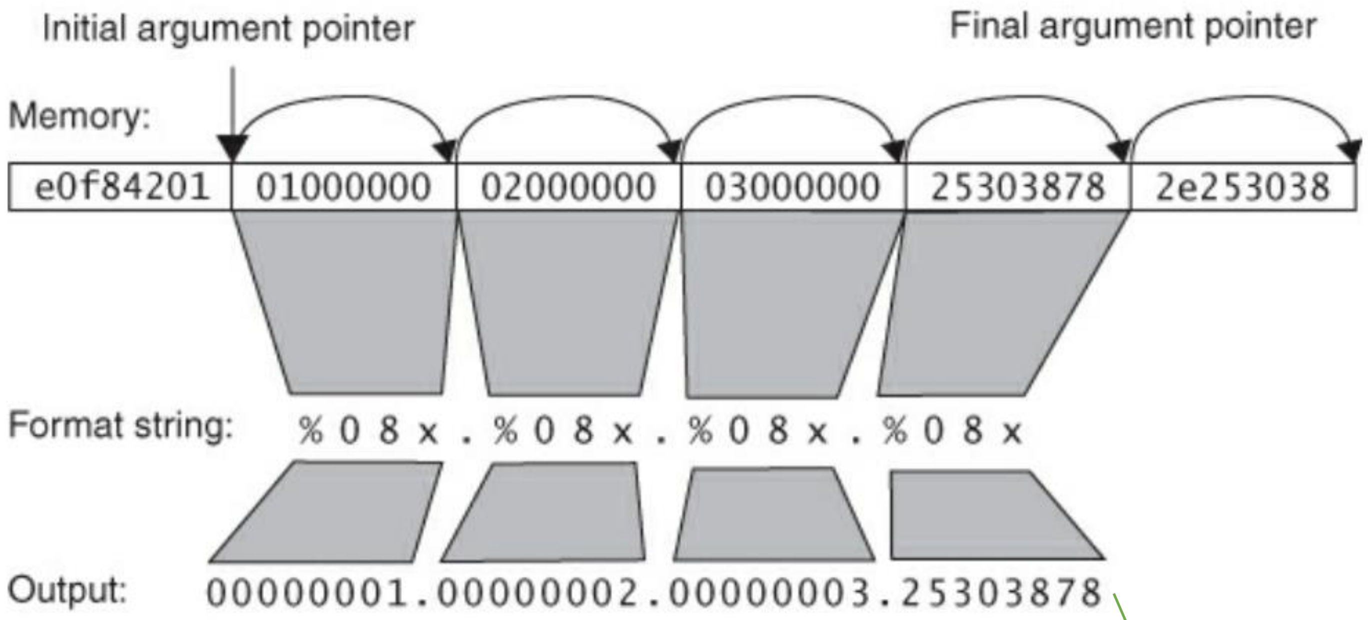


Figure 6.3. Viewing the contents of the stack

The format string %08x.%08x.%08x.%08 instructs printf() to retrieve four arguments from the stack and display them as eight-digit padded hexadecimal numbers

Viewing the Contents of the Stack

- The address of the format string 0xe0f84201 appears in memory followed by the argument values 1, 2, and 3

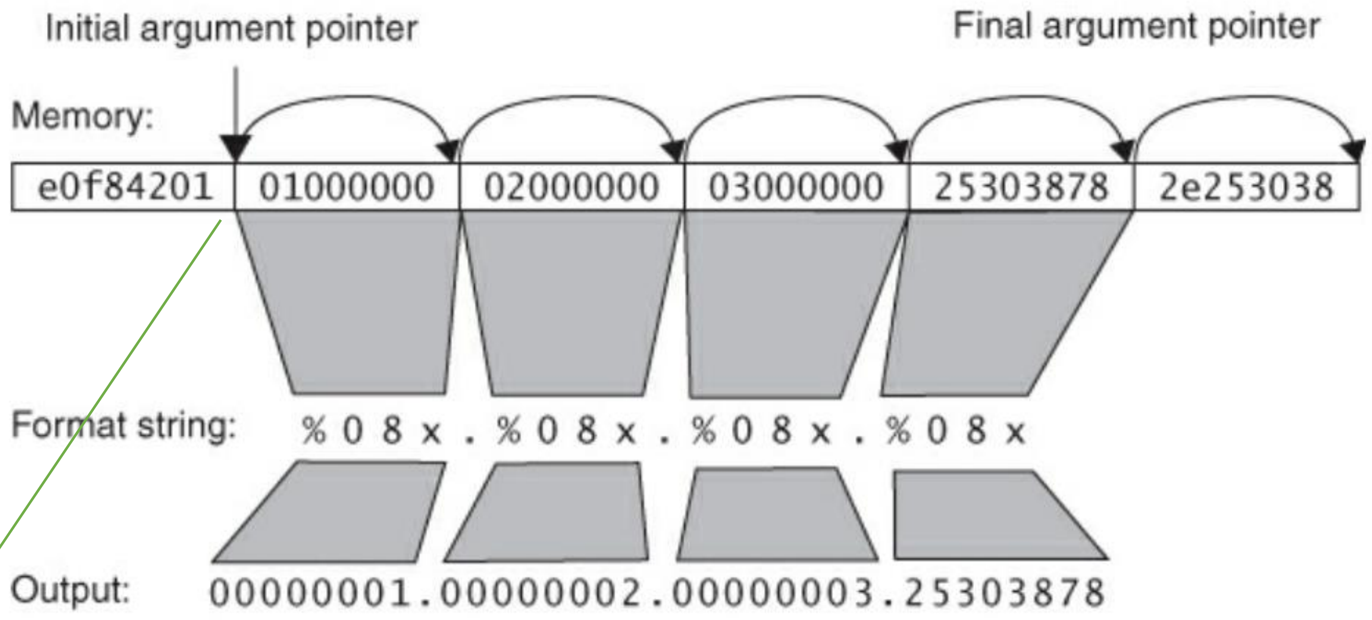


Figure 6.3. Viewing the contents of the stack

As each argument is used by the format specification, the argument pointer is increased by the length of the argument

Viewing the Contents of the Stack

- The address of the format string 0xe0f84201 appears in memory followed by the argument values 1, 2, and 3

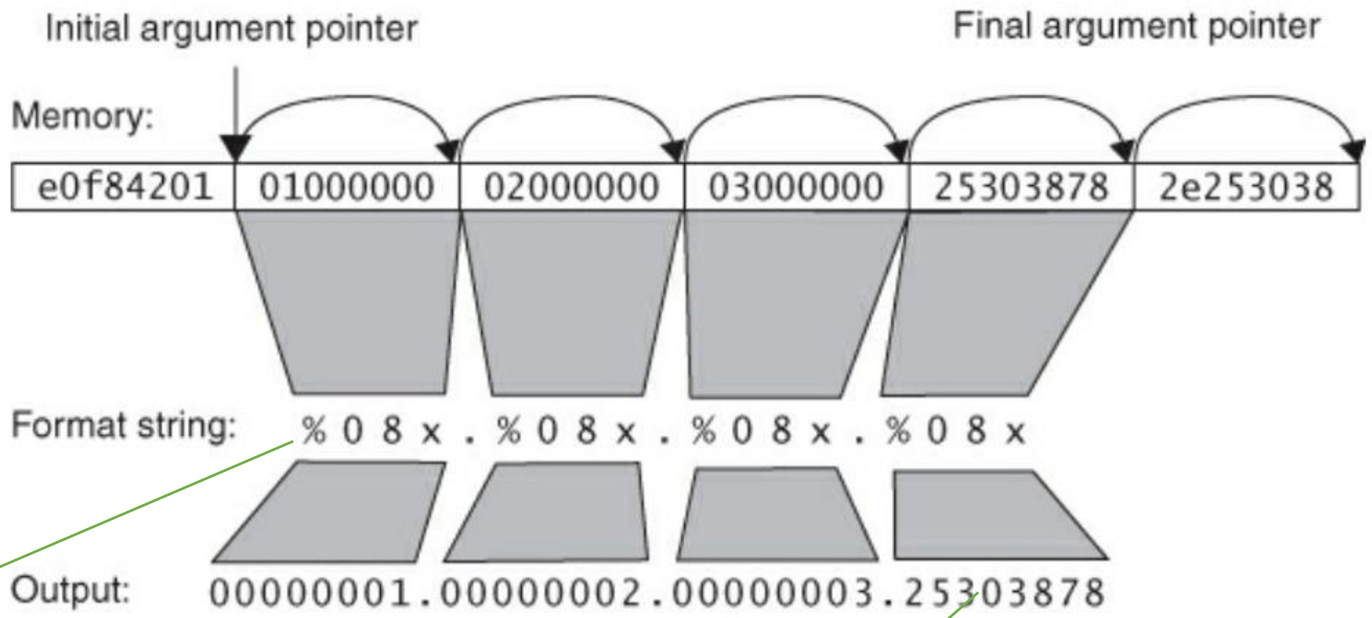


Figure 6.3. Viewing the contents of the stack

Each %08x in the format string reads a value it interprets as an `int` from the location identified by the argument pointer

The fourth “integer” contains the first four bytes of the format string—the ASCII codes for %08x

Viewing the Contents of the Stack

- Formatted output functions including `printf()` use an internal variable to identify the location of the next argument.
- The contents of the stack or the stack pointer are not modified, so execution continues as expected when control returns to the calling program.
- The formatted output function will continue displaying the contents of memory in this fashion until a null byte is encountered in the format string

Viewing the Contents of the Stack

- After displaying the remaining automatic variables for the currently executing function, `printf()` displays the stack frame for the currently executing function
- As `printf()` moves sequentially through stack memory, it displays the same information for the calling function.
- The function that called that function, and so on, up through the call stack.

Viewing the Contents of the Stack

- Using this technique, it is possible to reconstruct large parts of the stack memory.
- An attacker can use this data to determine offsets and other information about the program to further exploit this or other vulnerabilities

Viewing Memory Content

- The %s conversion specifier displays memory at the address specified by the argument pointer as an ASCII string until a null byte is encountered.
- If an attacker can manipulate the argument pointer to reference a particular address, the %s conversion specifier will output memory at that location.
- The argument pointer can be advanced in memory using the %x conversion specifier.
- The distance it can be moved depends on the size of the format string.
- An attacker can insert an address in an automatic variable in the calling function.
- If the format string is stored as an automatic variable, the address can be inserted at the beginning of the string.
- An attacker can create a format string to view memory at a specified address:
  `address advance-argptr %s`

Viewing Memory at a Specific Location

- address advance-argptr %s
- \xdc\xf5\x42\x01%x%x%x%s
- The series of three %x conversion specifiers advance the argument pointer twelve bytes to the start of the format string

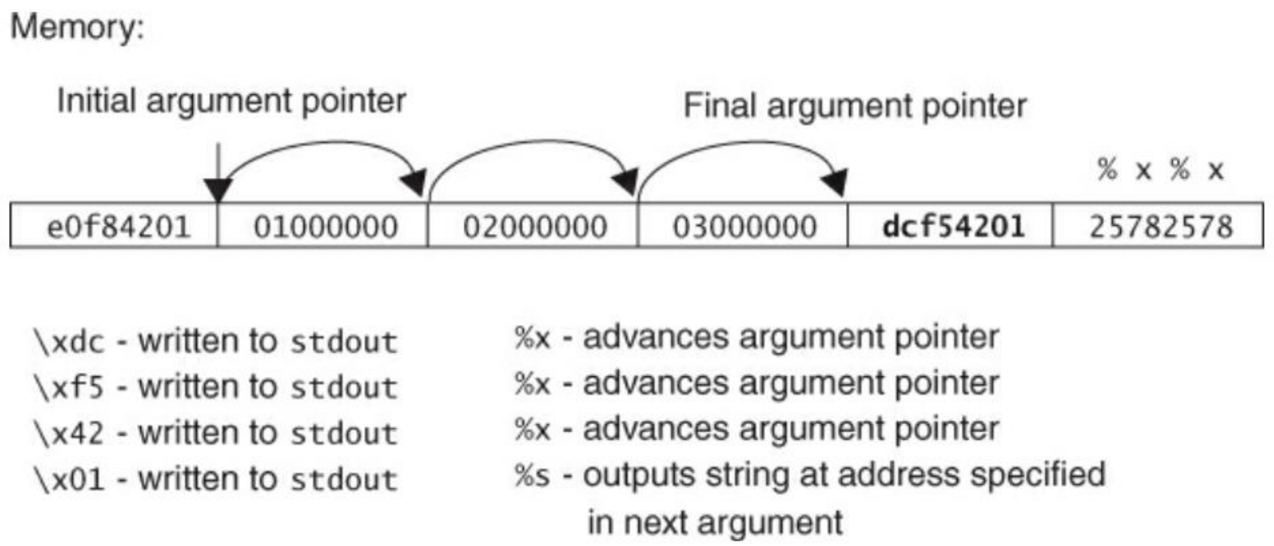


Figure 6.4. Viewing memory at a specific location

Viewing Memory Content

- `printf()` displays memory from 0x0142f5dc until a \0 byte is reached.
- The entire address space can be mapped by advancing the address between calls to `printf()`.
- Viewing memory at an arbitrary address can help an attacker develop other exploits, such as executing arbitrary code on a compromised machine.

Overwriting Memory

- Formatted output functions are dangerous because most programmers are unaware of their capabilities.
  - On platforms where integers and addresses are the same size (such as the IA-32), the ability to write an integer to an arbitrary address can be used to execute arbitrary code on a compromised system.
- The **%n conversion specifier** was created to help align formatted output strings.
  - It writes the number of characters successfully output to an integer address provided as an argument.

Overwriting Memory

- Example, after executing the following code snippet:
  - int i;
  - printf("hello%n\n", (int *)&i);
- The variable `i` is assigned the value 5 because five characters (h-e-l-l-o) are written until the %n conversion specifier is encountered.
- Using the %n conversion specifier, an attacker can write a small integer value to an address.
- To exploit this security flaw an attacker would need to write an arbitrary value to an arbitrary address.

Overwriting Memory

- The call:
  - printf("\xdc\xf5\x42\x01%08x.%08x.%08x%n”);
- Writes an integer value corresponding to the number of characters output to the address 0x0142f5dc.
  - The value written (28) is equal to the eight-character-wide hex fields (times three) plus the four address bytes.
- An attacker can overwrite the address with the address of some shellcode.

Overwriting Memory

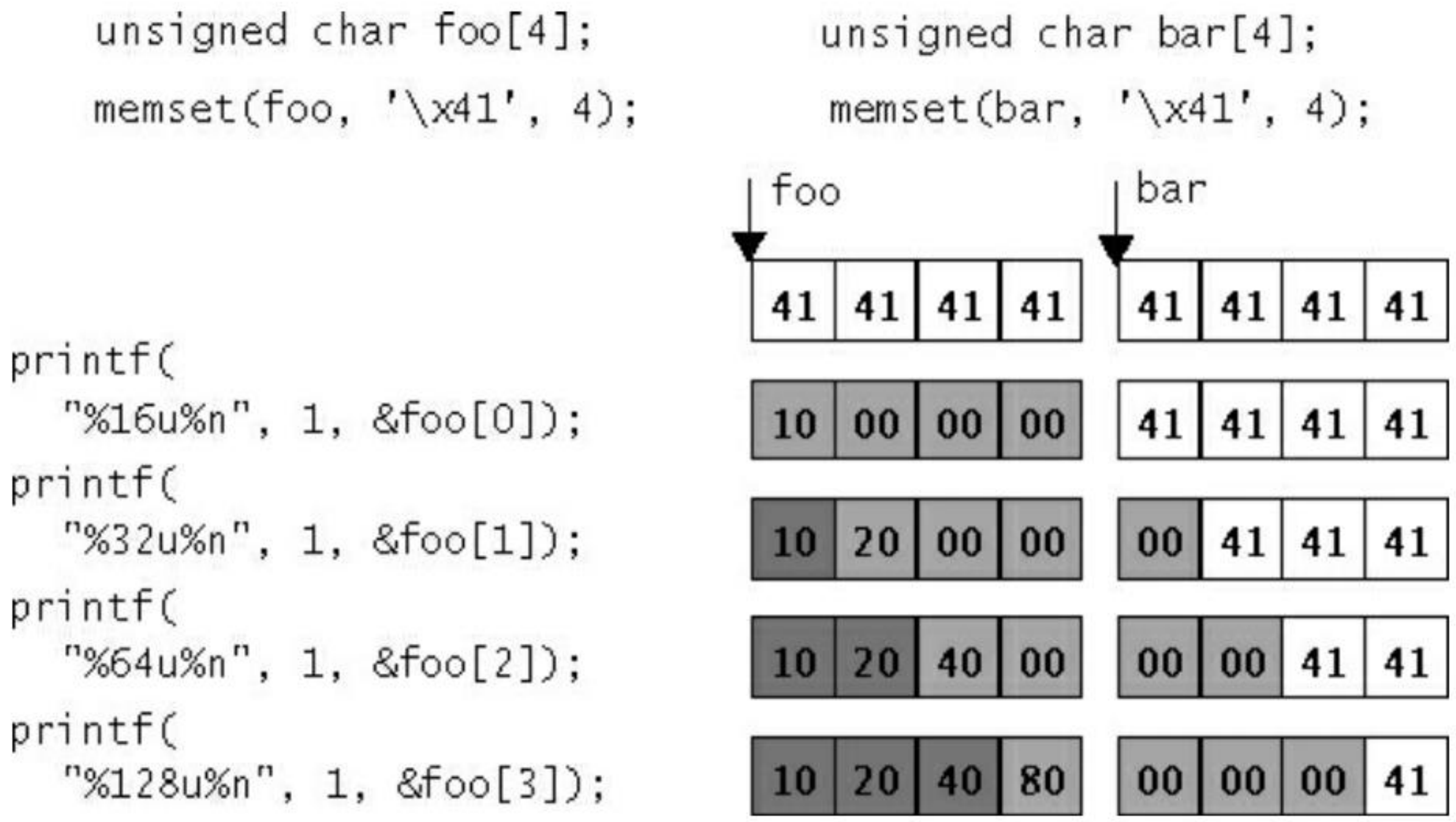
- An attacker can control the number of characters written using a conversion specification with a specified width or precision.
- Example:
  - int i;
  - printf ("%10u%n", 1, &i); /* i = 10 */
  - printf ("%100u%n", 1, &i); /* i = 100 */
- Each of the two format strings takes two arguments:
  - Integer value used by the %u conversion specifier.
  - The number of characters output.

Overwriting Memory

- On most complex instruction set computer (CISC) architectures, it is possible to write to an arbitrary address as follows:
  - Write four bytes.
  - Increment the address.
  - Write an additional four bytes.
- This technique has a side effect of overwriting the three bytes following the targeted memory

Writing an Address in Four Stages

- overwrite the memory in `foo` with the address 0x80402010



```
unsigned char foo[4];          unsigned char bar[4];

memset(foo, '\x41', 4);        memset(bar, '\x41', 4);

printf(
  "%16u%n", 1, &foo[0]);
printf(
  "%32u%n", 1, &foo[1]);
printf(
  "%64u%n", 1, &foo[2]);
printf(
  "%128u%n", 1, &foo[3]);
```

Figure 6.5. Writing an address in four stages

Writing an Address in Four Stages

- Each time the address is incremented, a trailing value remains in the low-memory byte.
- This byte is the low-order byte in a little endian architecture and the high-order byte in a big endian architecture.
- This process can be used to write a large integer value (an address) using a sequence of small (< 255) integer values.
- The process can also be reversed—writing from higher memory to lower memory while decrementing the address.

Writing an Address in Four Stages

- The formatted output calls perform only a single write per format string.
- Multiple writes can be performed in a single call to a formatted output function as follows:
  - printf ("%16u%n%16u%n%32u%n%64u%n",
    - 1, (int *) &foo[0], 1, (int *) &foo[1],
    - 1, (int *) &foo[2], 1, (int *) &foo[3])

Writing an Address in Four Stages

- The only difference in combining multiple writes into a single format string is that the counter continues to increment with each character output.
  - printf ("%16u%n%16u%n%32u%n%64u%n",
- The first %16u%n sequence writes the value 16 to the specified address, but the second %16u%n sequence writes 32 bytes because the counter has not been reset.

Exploit Code to Write an Address

```
static unsigned int already_written, width_field;
static unsigned int write_byte;
static char buffer[256];
already_written = 506;
  // first byte
write_byte = 0x3C8;
already_written %= 0x100;
width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
  // second byte
write_byte = 0x3fA;
already_written += width_field;
already_written %= 0x100;
width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
// third byte
write_byte = 0x442;
already_written += width_field;
already_written %= 0x100;
width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
  // fourth byte
write_byte = 0x501;
already_written += width_field;
already_written %= 0x100;
width_field = (write_byte - already_written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
```

Exploit Code to Write an Address

- The code uses three unsigned integers:

```
already_written
width_field
write_byte
```

- The `write_byte` variable contains the value of the next byte to be written.
- The `already_written` variable counts the number of characters output.
- The `width_field` stores the width field for the conversion specification required to produce the required value for %n.

Exploit Code to Write an Address

- The required width is determined by subtracting the number of characters already output from the value of the byte to write modulo 0x100.
- The difference is the number of output characters required to increase the value of the output counter from its current value to the desired value.
- After each write, the value of the width field from the previous conversion specification is added to the bytes already written.

Overwriting Memory

```
1. unsigned char exploit[1024] =       "\x90\x90\x90...\x90";
2. char format[1024];
3. strcpy(format, "\xaa\xaa\xaa\xaa");
4. strcat(format, "\xdc\xf5\x42\x01");
5. strcat(format, "\xaa\xaa\xaa\xaa");
6. strcat(format, "\xdd\xf5\x42\x01");
7. strcat(format, "\xaa\xaa\xaa\xaa");
8. strcat(format, "\xde\xf5\x42\x01");
9. strcat(format, "\xaa\xaa\xaa\xaa");
10. strcat(format, "\xdf\xf5\x42\x01");
11. for (i=0; i < 61; i++) {
12.     strcat(format, "%x");
13. }
            /* code to write address goes here */
14. printf(format)
```

four sets of dummy integer/address pairs instructions to advance the argument pointer instructions to overwrite an address

Lines 3, 5, 7, and 9 insert dummy integer arguments in the format string corresponding to the %u conversion specifications

Lines 4, 6, 8, and 10 specify the sequence of values required to overwrite the address at 0x0142f5dc with the address of the exploit code

Lines 11-13 write the appropriate number of %x conversion specifications to advance the argument pointer to the start of the format string and the first dummy integer/address pair

Internationalization

- Because of internationalization, format strings and message text are often moved into external catalogs or files that the program opens at runtime.
- An attacker can alter the values of the formats and strings in the program by modifying the contents of these files.
- These files should have file protections that prevent their contents from being altered.
  - Set search paths, environment variables, or logical names to limit access.

Stack Randomization

- Under Linux the stack starts at 0xC0000000 and grows towards low memory.
- Few Linux stack addresses contain null bytes, which makes them easier to insert into a format string.
- Many Linux variants include some form of stack randomization.
- It difficult to predict the location of information on the stack, including the location of return addresses and automatic variables, by inserting random gaps into the stack

Thwarting Stack Randomization

- If these values can be identified, it becomes possible to exploit a format string vulnerability on a system protected by stack randomization:
  - address to overwrite,
  - address of the shell code,
  - distance between the argument pointer and the start of the format string,
  - number of bytes already written by the formatted output function before the first %u conversion specification.

Address to Overwrite

- It is possible to overwrite the GOT entry for a function or other address to which control is transferred during normal execution of the program.
- The advantage of overwriting a GOT entry is its independence from system variables such as the stack and heap.

Address of the Shellcode

- A Windows-based exploit assumes that the shellcode is inserted into an automatic variable on the stack.
- This address would be difficult to find on a system that has implemented stack randomization.
- The shellcode could also be inserted into a variable in the data segment or heap, making it easier to find.

Distance

- An attacker needs to find the distance between the argument pointer and the start of the format string on the stack.
- The relative distance between them remains constant.
- It is easy to calculate the distance from the argument pointer to the start of the format string and insert the required number of %x format conversions.

Writing Addresses in Two Words

- The Windows-based exploit wrote the address of the shellcode a byte at a time in four writes, incrementing the address between calls.
- If this is impossible because of alignment requirements or other reasons, it may still be possible to write the address a word at a time or even all at once.

Linux Exploit Variant

- This exploit inserts the shellcode in the data segment, using a variable declared as static

```
1. #include <stdio.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.   static unsigned char shellcode[1024] =
           "\x90\x09\x09\x09\x09\x09/bin/sh";
5.   int i;
6.   unsigned char format_str[1024];
7.   strcpy(format_str, "\xaa\xaa\xaa\xaa");
8.   strcat(format_str, "\xb4\x9b\x04\x08");
9.   strcat(format_str, "\xcc\xcc\xcc\xcc");
10.   strcat(format_str, "\xb6\x9b\x04\x08");
11.   for (i=0; i < 3; i++) {
12.     strcat(format_str, "%x");
13.   }
    /* code to write address goes here */
14.   printf(format_str);
15.   exit(0);
16. }
```

The address of the GOT entry for the exit() function is concatenated to the format string

The same address + 2 is concatenated on line 10

Control is transferred to the shellcode when the program terminates on the call to exit()

Mitigation Strategies

- disabling the %n conversion specifier by default?
  - Because of an existing code base, impossible to change library (remove %n)
- Disallow dynamic format strings.
  - Instead, the dynamic use of static content

Dynamic Format Strings

```
#include <stdio.h>
#include <string.h>
int main(int argc, char * argv[]) {
int x, y;
static char format[256] = "%d * %d = ";
x = atoi(argv[1]);
y = atoi(argv[2]);
if (strcmp(argv[3], "hex") == 0) {
    strcat(format, "0x%x\n");
}
else {
    strcat(format, "%d\n");
}
printf(format, x, y, x * y);
exit(0);
}
```

- This program is secure from format string vulnerabilities.

Restricting Bytes Written

- Buffer overflows can be prevented by restricting the number of bytes written by formatted output functions.
- The number of bytes written can be restricted by specifying a precision field as part of the %s conversion specification.
- Example, instead of
  - sprintf(buffer, "Wrong command: %s\n", user);
- use
  - sprintf(buffer, "Wrong command: %.495s\n", user);
- The precision field specifies the maximum number of bytes to be written for %s conversions.
- In the example:
  - The static string contributes 17 bytes.
  - A precision of 495 ensures that the resulting string fits into a 512 byte buffer.

Restricting Bytes Written

- Use more secure versions of formatted output library functions that are less susceptible to buffer overflows.
- Example,
  - snprintf() better than sprintf().
  - vsnprintf() as alternative to vsprintf()).
- These functions specify a maximum number of bytes to write, including the trailing null byte.

Restricting Bytes Written

- The asprintf() and vasprintf() functions can be used instead of sprintf() and vsprintf().
- These functions allocate a string large enough to hold the output including the terminating null, and they return a pointer to it via the first parameter.
- These functions are GNU extensions and are not defined in the C or POSIX standards.
- They are also available on *BSD systems.

C11 Annex K Bounds-checking interfaces

- Security-enhanced functions:
  - fprintf_s(),
  - printf_s(),
  - snprintf_s(),
  - sprint_s(),
  - vfprintf_s(),
  - vprintf_s(),
  - vsnprintf_s(),
  - vsprintf_s().

Security-enhanced Functions

- They differ from their non-_s counterparts by:
  - not supporting the %n format conversion specifier,
  - making it a constraint violation if pointers are null,
  - the format string is invalid.
- These functions cannot prevent format string vulnerabilities that crash a program or are used to view memory.

iostream vs. stdio

- C++ programmers have the option of using the `iostream` library, which provides input and output functionality using streams.
  - Formatted output using `iostream` relies on the insertion operator <<, an infix binary operator.
  - The operand to the left is the stream to insert the data into.
  - The operand on the right is the value to be inserted.
  - Formatted and tokenized input is performed using the >> extraction operator.
  - The standard I/O streams `stdin`, `stdout`, and `stderr` are replaced by `cin`, `cout`, and `cerr`.

Extremely Insecure `stdio` Implementation

```
1.    #include <stdio.h>
      #include <stdlib.h>
2.   int main(int argc, char * argv[]) {
3.    char filename[256];
4.    FILE *f;
5.    char format[256];
6.    fscanf(stdin, "%s", filename);
7.    f = fopen(filename, "r"); /* read only */
8.    if (f == NULL) {
9.        sprintf(format, "Error opening file %s\n", filename);
10.       fprintf(stderr, format);
11.       exit(-1);
12.   }
13.  fclose(f);
14. }
```

This program reads a filename from `stdin` and attempts to open the file

If the open fails, an error message is printed on line 10

This program is vulnerable to **buffer overflows**

This program is vulnerable to **format string exploits**

Insecure

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

int main(int argc, char *argv[]) {
        char filename[256];
        FILE *f;
        char format[256];

        printf("Enter filename: ");
        fscanf(stdin, "%s", filename);
        if ((f = fopen(filename, "r")) == NULL) {
                // file open error
                sprintf(format, "Error opening file %s\n", filename);
                fprintf(stderr, format);
                exit(-1);
        }

        fclose(f);
}
```

Testing

- It is extremely difficult to construct a test suite that exercises all possible paths through a program.
- A major source of format string bugs comes from error-reporting code.
  - Because such code is triggered as a result of exceptional conditions, these paths are often missed by runtime testing.
- Current versions of the GNU C compiler provide flags -Wformat, -Wformat-nonliteral, and -Wformat-security.

-Wformat

- The -Wformat option is included in –Wall.
- This flag instructs the GCC compiler to:
  - check calls to formatted output functions,
  - examine the format string,
  - verify that the correct number and types of arguments are supplied.
- This feature does not report mismatches between signed and unsigned integer conversion specifiers and their corresponding arguments.

-Wformat-nonliteral

- This flag performs the same function as –Wformat
- Adds warnings if the format string is not a string literal
- Cannot be checked, unless the format function takes its format arguments as a va_list.

-Wformat-security

- This flag performs the same function as -Wformat but adds warnings about formatted output function calls that represent possible security problems.
- At present, this warns about calls to printf() where the format string is not a string literal and there are no format arguments (e.g., printf (foo)).
- This is currently a subset of what -Wformat-nonliteral warns about, but future warnings may be added to -Wformat-security that are not included in -Wformat-nonliteral.

Example

```
gcc –Wformat –Wformat-nonliteral sprintf.c
gcc –Wformat –Wformat-security sprintf.c
```

Static Taint Analysis

- Shankar describes a system for detecting format string security vulnerabilities in C programs using a constraint-based type-inference engine.
  - Inputs from untrusted sources are marked as tainted,
  - Data propagated from a tainted source is marked as tainted,
  - A warning is generated if tainted data is interpreted as a format string.
  - The tool is built on the equal extensible type qualifier framework.
- Tainting is modeled by extending the existing C type system with extra type qualifiers.
- The standard C type system already contains qualifiers such as const.
- Adding a tainted qualifier allows the types of all untrusted inputs to be labeled as tainted.
- Example:

```
tainted int getchar();
int main(int argc, tainted char *argv[])
```

Static Taint Analysis

- The return value from getchar() and the command-line arguments to the program are labeled and treated as tainted values.
- Given a small set of initially tainted annotations, typing for all program variables can be inferred to indicate whether each variable might be assigned a value derived from a tainted source.
- If any expression with a tainted type is used as a format string, the user is warned of the potential vulnerability.

Modifying the Variadic Function Implementation

- Exploits of format string vulnerabilities require that the argument pointer be advanced beyond the legitimate arguments passed to the formatted output function.
  - This is accomplished by specifying a format string that consumes more arguments than are available.
  - Restricting the number of arguments processed by a variadic function to the actual number of arguments passed can eliminate exploits in which the argument pointer needs to be advanced.
  - It is difficult to determine when the arguments have been exhausted by passing a terminating argument.
- The ANSI variadic function mechanism allows arbitrary data to be passed as arguments.
  - Pass the number of arguments to the variadic function as an argument.

Safe variadic Function Implementation

- the va_start() macro has been expanded to initialize a va_count variable to the number of variable arguments.

```
1. #define va_start(ap,v) (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v)); \
     int va_count = va_arg(ap, int)
2. #define va_arg(ap,t) \     (*(t *)((ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
     if (va_count-- == 0) abort();
3. int main(int argc, char * argv[]) {
4.   int av = -1;
5.   av = average(5, 6, 7, 8, -1); // works
6.   av = average(5, 6, 7, 8); // fails
7.   return 0;
8. }
```

The va_arg() macro has been extended to decrement the va_count variable each time it is called

If the count reaches zero, then no further arguments are available and the function fails

The first call to average() succeeds as the -1 argument is recognized by the function as a termination condition

Fails: the user of the function neglected to pass -1 as an argument

Safe variadic Function Binding

- The extra argument containing the count of variable arguments is inserted on line 4

```
av = average(5, 6, 7, 8); // fails
  1. push  8
  2. push  7
  3. push  6
  4. push  4 // 4 var args (and 1 fixed)
  5. push  5
  6. call  average
  7. add   esp, 14h
  8. mov   dword ptr [av], eax
```

- Example of the assembly language instructions that would need to be generated for the call to average() on line 6 to work with the modified variadic function implementation.

Exec Shield

- Exec Shield is a kernel-based security feature for Linux IA32 developed by Arjan van de Ven and Ingo Molnar.
- In Red Hat Enterprise Linux v.3, update 3, Exec Shield randomizes the stack, the location of shared libraries, and the start of the programs heap.
- Exec Shield stack randomization is implemented by the kernel as executables are launched.
- The stack pointer is increased by a random value. No memory is wasted because the omitted stack area is not paged in.

FormatGuard

- FormatGuard injects code to dynamically check and reject formatted output function calls if the number of arguments does not match the number of conversion specifications.
- Applications must be recompiled using FormatGuard for these checks to work.
- FormatGuard uses the GNU C pre-processor (CPP) to extract the count of actual arguments. This count is passed to a safe wrapper function.
- The wrapper parses the format string to determine how many arguments to expect.
- If the format string consumes more arguments than are supplied, the wrapper function raises an intrusion alert and kills the process.
- If the attacker's format string undercounts or matches the actual argument count to the formatted output function, FormatGuard fails to detect the attack.
- it is possible for the attacker to employ such an attack by creatively entering the arguments.

Libsafe

- Libsafe version 2.0 prevents format string vulnerability exploits that attempt to overwrite return addresses on the stack.
  - If an attack is attempted, Libsafe logs a warning and terminates the targeted process.
  - Libsafe includes safer versions of vulnerable functions.
  - Its first task is to make sure that the functions can be safely executed based on their arguments
  - Libsafe executes code that is functionally equivalent (e.g., snprintf() in place of sprintf()).
  - Libsafe is implemented as a shared library that is loaded into memory before the standard library.
  - Some functions are re-implemented to provide the necessary checks.
- Example: formatted output functions are re-implemented to perform two additional checks.
  - The first check examines the pointer argument associated with each %n conversion specifier to determine whether the address references a return address or frame pointer.
  - The second check determines whether the initial location of the argument pointer is in the same stack frame as the final location of the argument pointer.

Static Binary Analysis

- It is possible to discover format string vulnerabilities by examining binary images using the following criteria:
  - Is the stack correction smaller than the minimum value?
  - Is the format string variable or constant?
- The printf() function accepts at least two parameters: a format string and an argument.
- If a printf() function is called with only one argument and this argument is variable, the call may represent an exploitable vulnerability.

Static Binary Analysis

- The number of arguments passed to a formatted output function can be determined by examining the stack correction following the call.
- Example:
  - Only one argument was passed to the printf() function because the stack correction is only four bytes:
    - lea eax, [ebp+10h]
    - push eax
    - call printf
    - add esp, 4

Notable Vulnerabilities: Wu-ftpd

- Washington University FTP daemon (wu-ftpd) is a popular UNIX FTP server shipped with many distributions of Linux and other UNIX operating systems.
- A format string vulnerability exists in the insite_exec() function of wu-ftpd versions before 2.6.1.
- wu-ftpd is a string vulnerability where the user input is incorporated in the format string of a formatted output function in the Site Exec command functionality.

CDE ToolTalk

- The common desktop environment (CDE) is an integrated graphical user interface that runs on UNIX and Linux operating systems.
- CDE ToolTalk is a message brokering system that provides an architecture for applications to communicate with each other across hosts and platforms.
- There is a remotely exploitable format string vulnerability in versions of the CDE ToolTalk RPC database server.

Summary

- Improper use of C99 standard formatted output routines can lead to exploitation ranging from information leakage to the execution of arbitrary code.
- Format string vulnerabilities, in particular, are relatively easy to discover (e.g., by using the -Wformat-nonliteral flag in GCC) and correct.
- Format string vulnerabilities can be more difficult to exploit than simple buffer overflows because they require synchronizing multiple pointers and counters.
- The location of the argument pointer to view memory at an arbitrary location must be tracked along with the output counter when overwriting memory.
- An obstacle to exploitation may occur when the memory address to be examined or overwritten contains a null byte.
- Because the format string is a string, the formatted output function exits with the first null byte.

Summary

- The default configuration for Visual C++ .NET, places the stack in low memory (e.g., 0x00hhhhhh).
- These addresses are more difficult to attack in any exploit that relies on a string operation.
- Recommended practices for eliminating format string vulnerabilities include preferring iostream to stdio when possible and using static format strings when not.
- When dynamic format strings are required, it is critical that input from untrusted sources is not incorporated into the format string.

Q&A