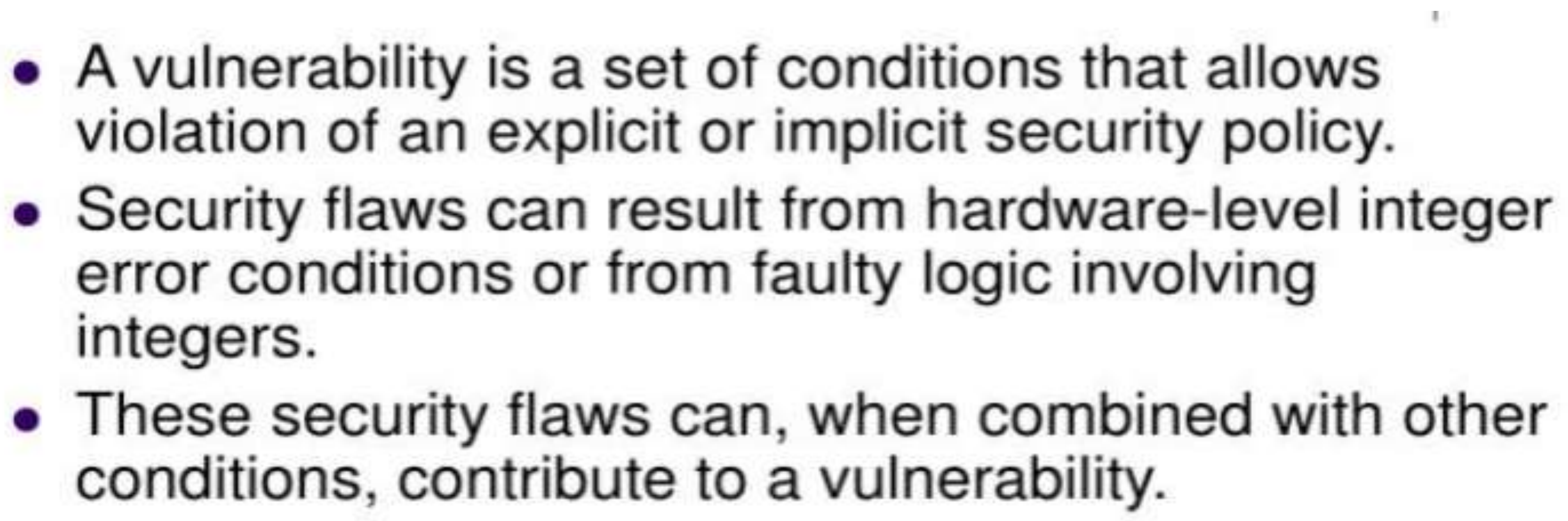
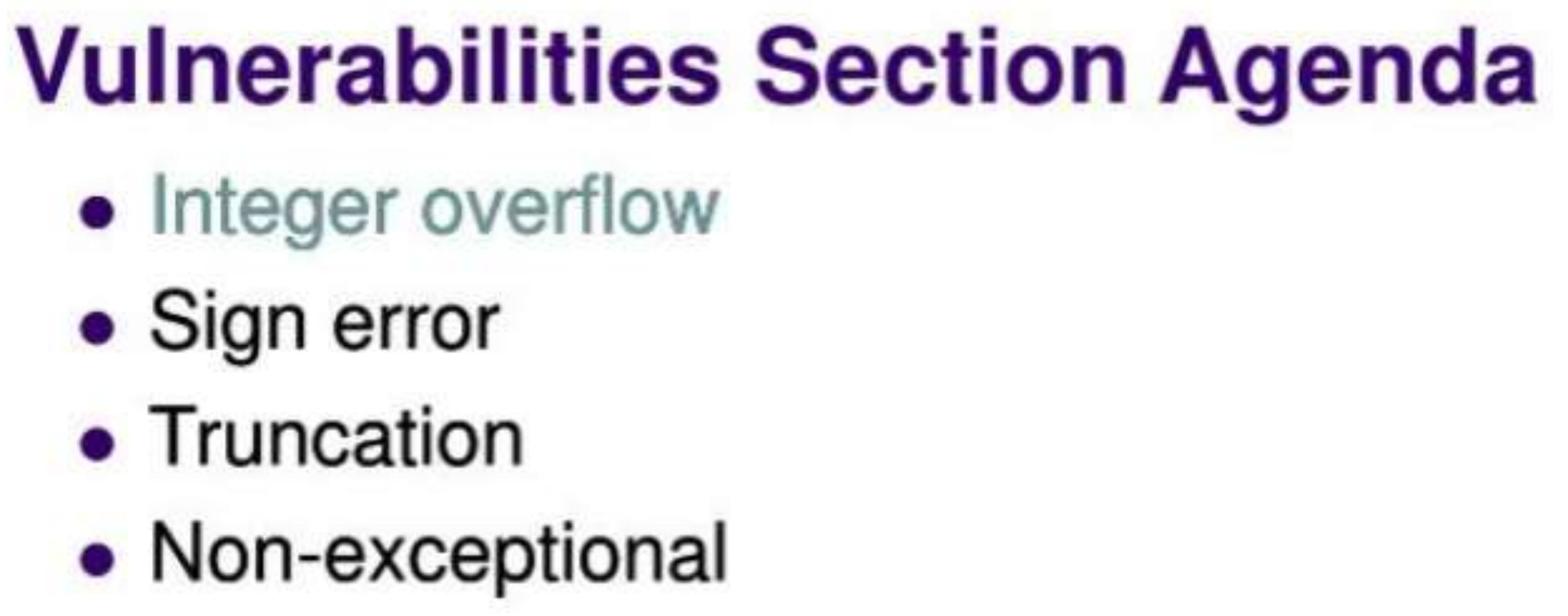
UNIT-4—Integer Security

Integer Security Vulnerabilities



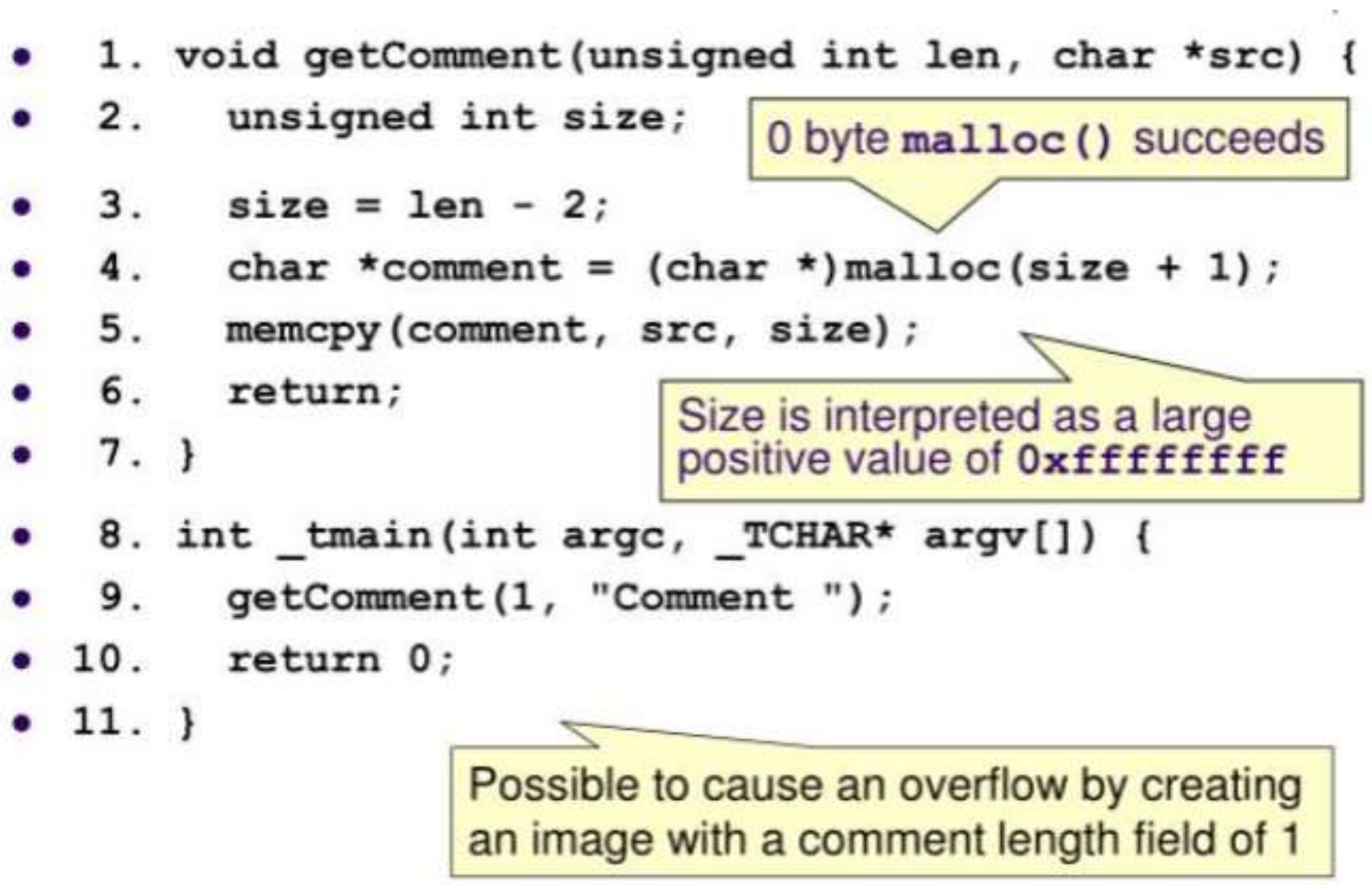
- A vulnerability is a set of conditions that allows violation of an explicit or implicit security policy.
- Security flaws can result from hardware-level integer error conditions or from faulty logic involving integers.
- These security flaws can, when combined with other conditions, contribute to a vulnerability.



Vulnerabilities Section Agenda

- Integer overflow
- Sign error
- Truncation
- Non-exceptional

Integer overflow Example



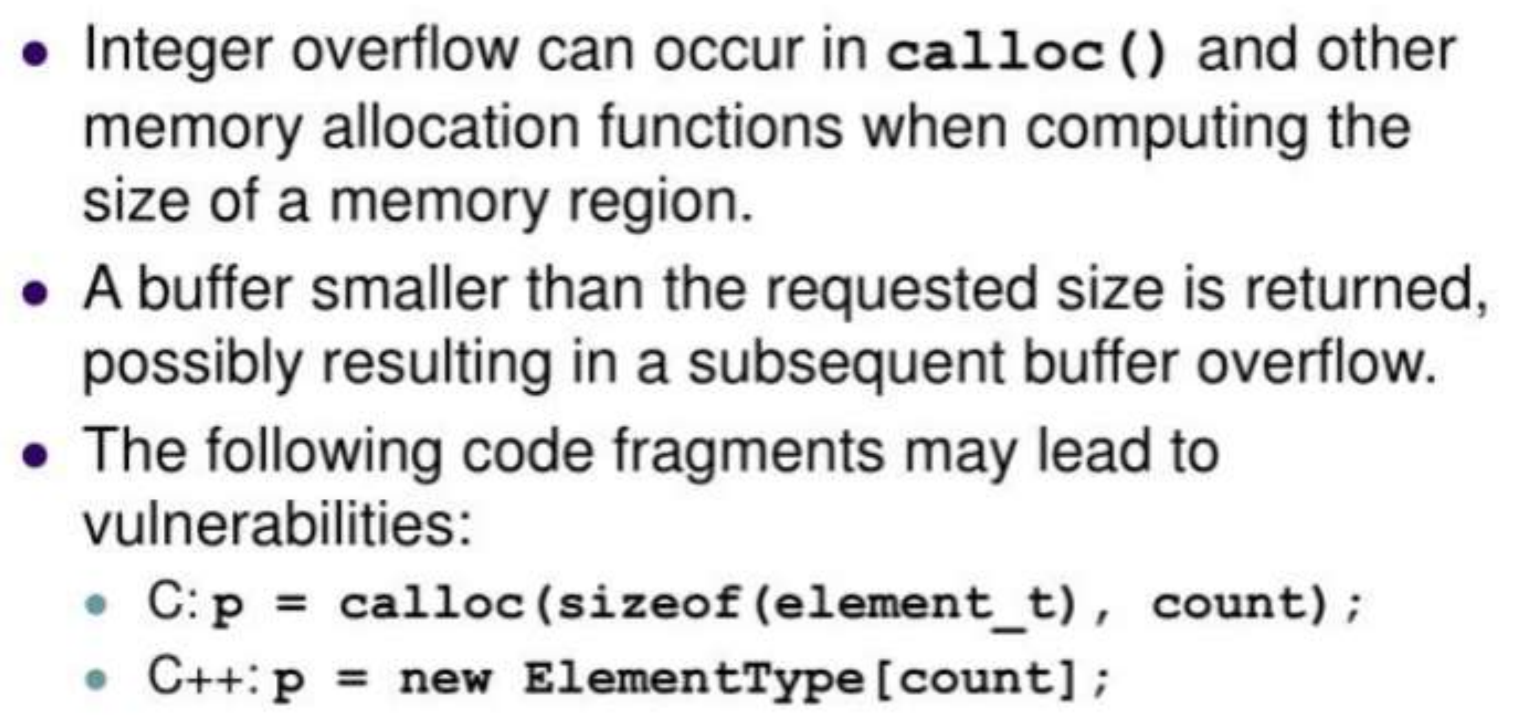
```
void getComment(unsigned int len, char *src) {
   unsigned int size;

   size = len - 2;
   char *comment = (char *)malloc(size + 1);
   memcpy(comment, src, size);
   return;
}

int _tmain(int argc, _TCHAR* argv[]) {
   getComment(1, "Comment ");
   return 0;
}
```

0 byte `malloc()` succeeds

Size is interpreted as a large positive value of `0xffffffff`

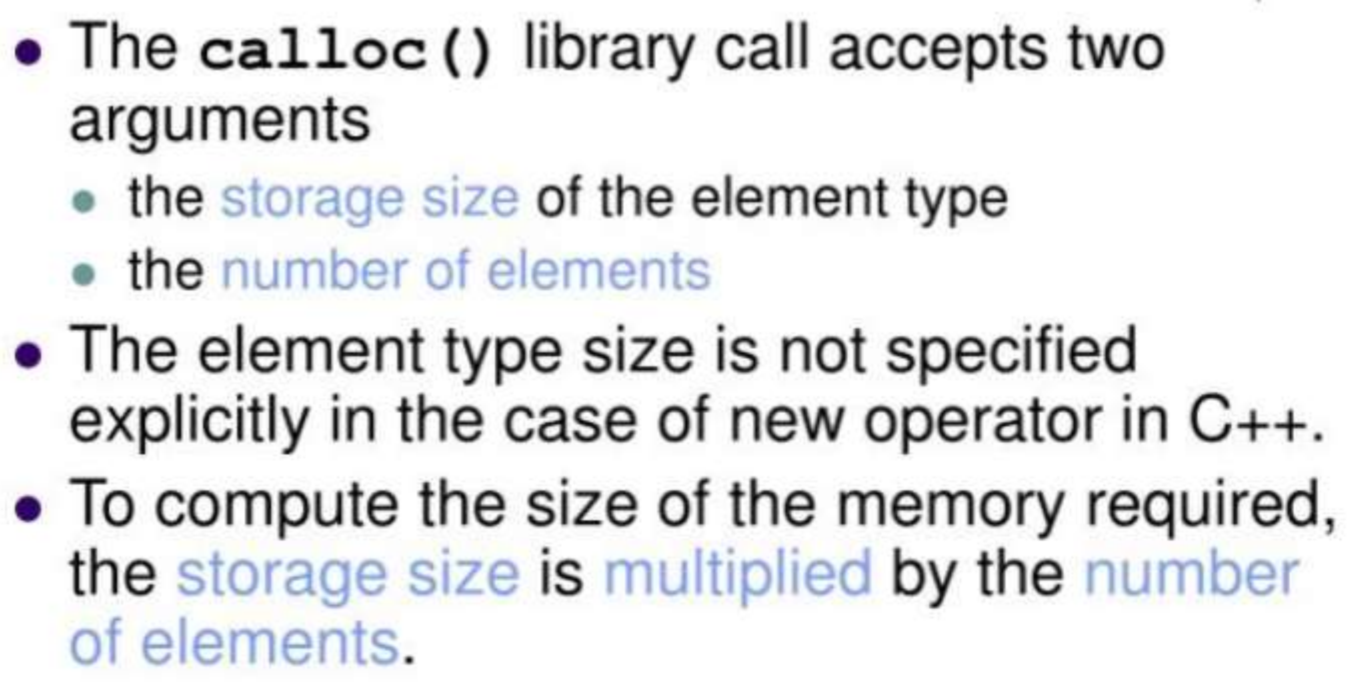
Possible to cause an overflow by creating an image with a comment length field of 1

Memory Allocation Example



- Integer overflow can occur in `calloc()` and other memory allocation functions when computing the size of a memory region.
- A buffer smaller than the requested size is returned, possibly resulting in a subsequent buffer overflow.
- The following code fragments may lead to vulnerabilities:
  - C: `p = calloc(sizeof(element_t), count);`
  - C++: `p = new ElementType[count];`

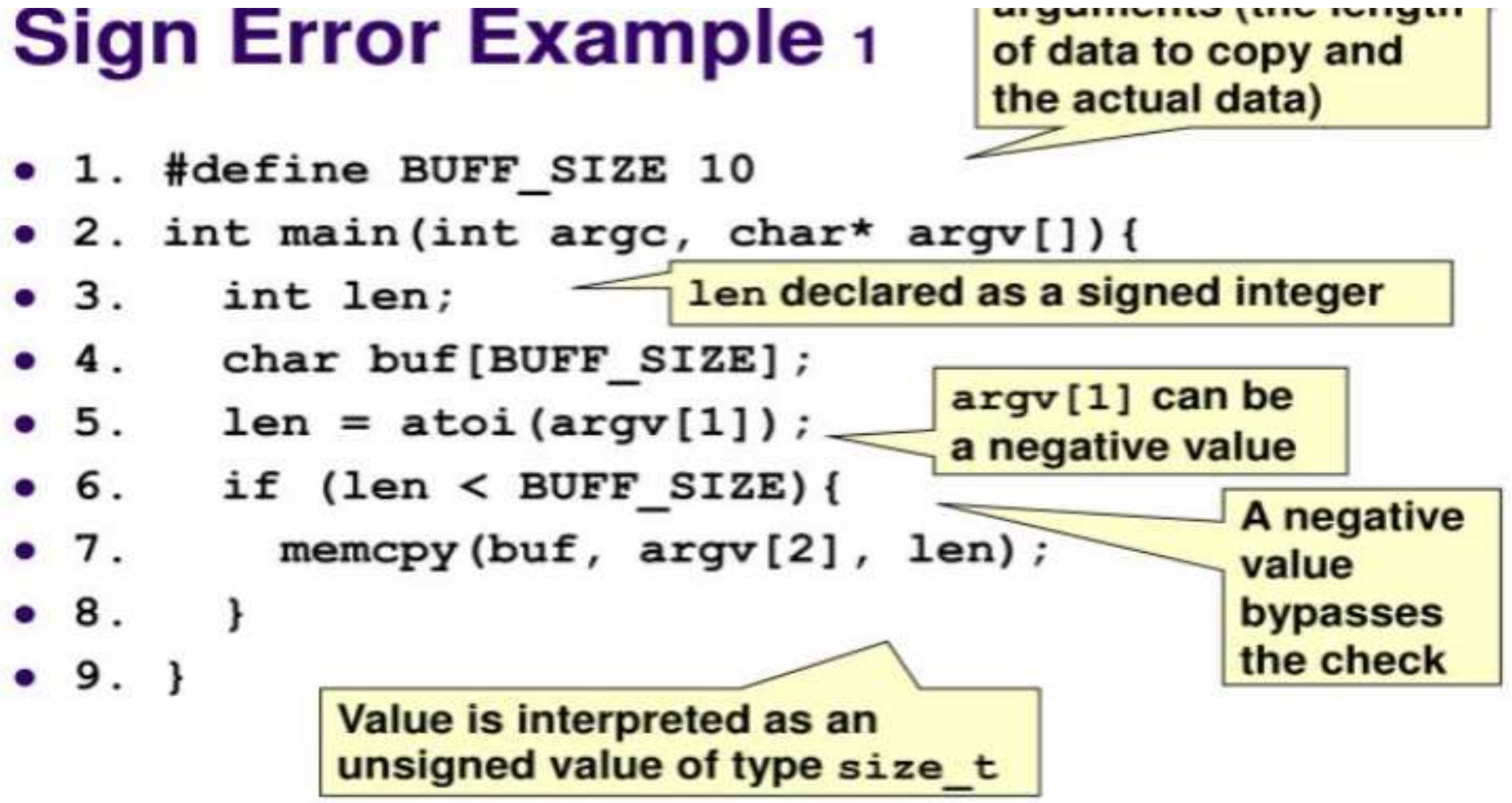
Memory Allocation Example



- The `calloc()` library call accepts two arguments
  - the storage size of the element type
  - the number of elements
- The element type size is not specified explicitly in the case of new operator in C++.
- To compute the size of the memory required, the storage size is multiplied by the number of elements.

Overflow Condition

- If the result cannot be represented in a signed integer, the allocation routine can appear to succeed but allocate an area that is too small.
- The application can write beyond the end of the allocated buffer resulting in a heap-based buffer overflow.



Sign Error Example 1

of data to copy and the actual data)

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
    int len;
    char buf[BUFF_SIZE];
    len = atoi(argv[1]);
    if (len < BUFF_SIZE){
      memcpy(buf, argv[2], len);
    }
}
```

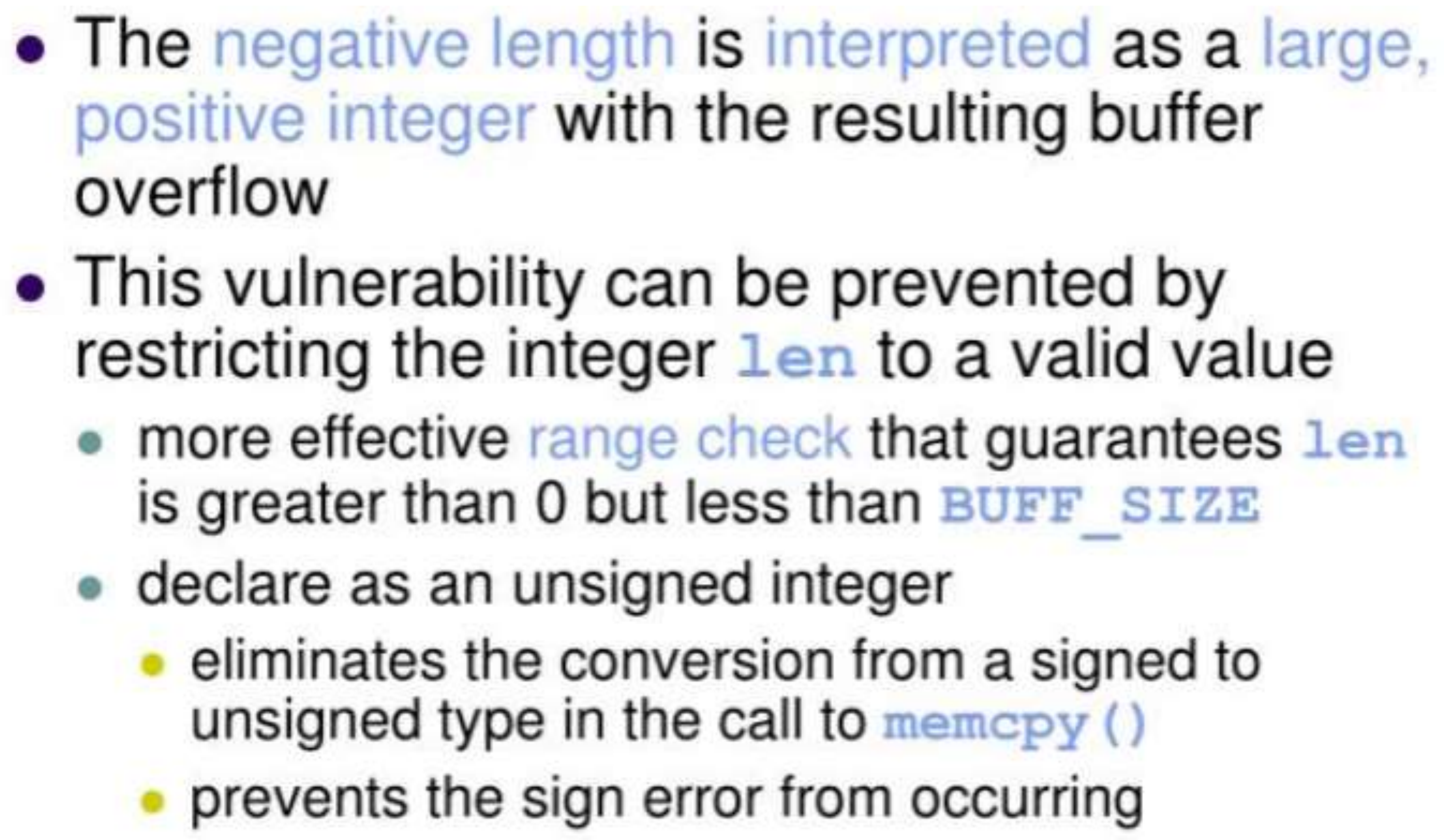
`len` declared as a signed integer

`argv[1]` can be a negative value

A negative value bypasses the check

Value is interpreted as an unsigned value of type `size_t`

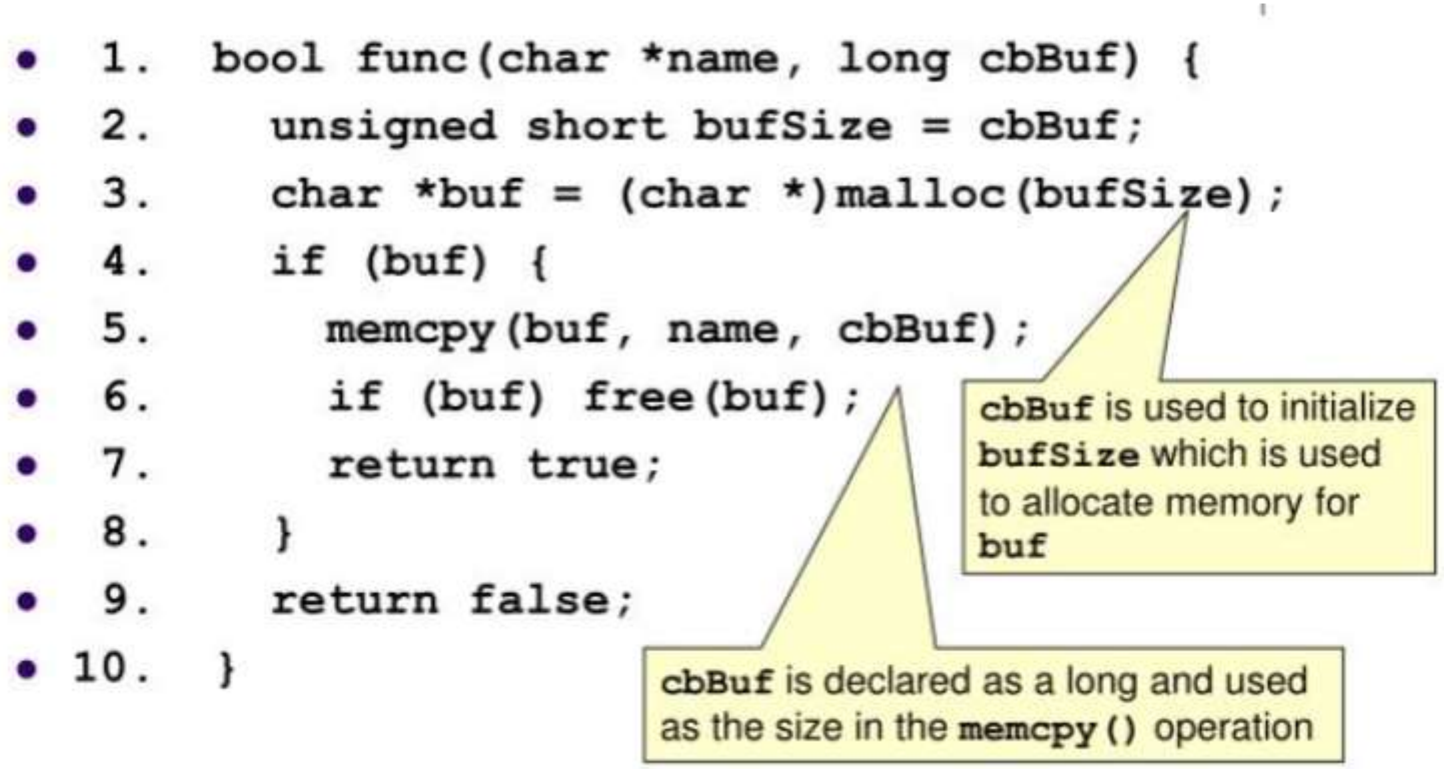
Sign Errors Example-2



- The negative length is interpreted as a large, positive integer with the resulting buffer overflow
- This vulnerability can be prevented by restricting the integer `len` to a valid value
  - more effective range check that guarantees `len` is greater than 0 but less than `BUFF_SIZE`
  - declare as an unsigned integer
    - eliminates the conversion from a signed to unsigned type in the call to `memcpy()`
    - prevents the sign error from occurring

UNIT-4—Integer Security

Truncation Vulnerable Implementation

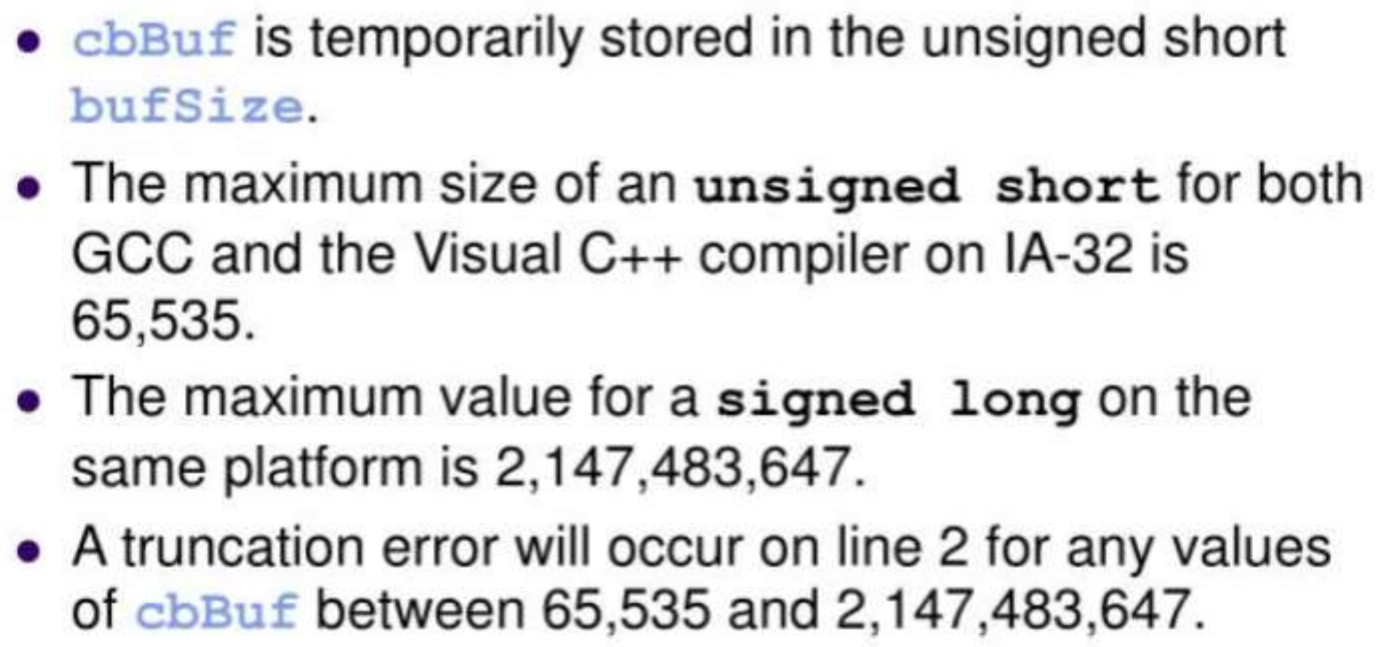


```
bool func(char *name, long cbBuf) {
  unsigned short bufSize = cbBuf;
  char *buf = (char *)malloc(bufSize);
  if (buf) {
    memcpy(buf, name, cbBuf);
    if (buf) free(buf);
    return true;
  }
  return false;
}
```

cbBuf is used to initialize **bufSize** which is used to allocate memory for **buf**

cbBuf is declared as a long and used as the size in the **memcpy()** operation

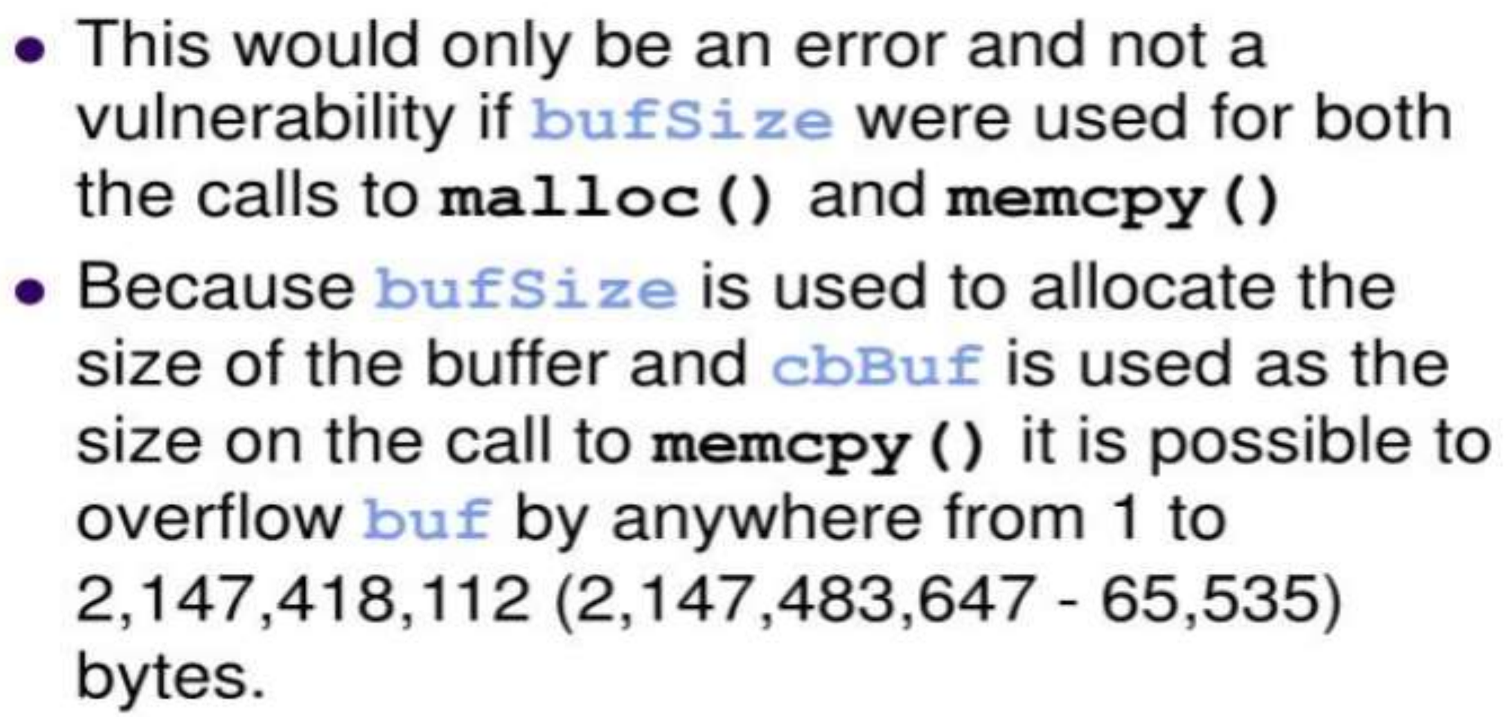
Vulnerability 1



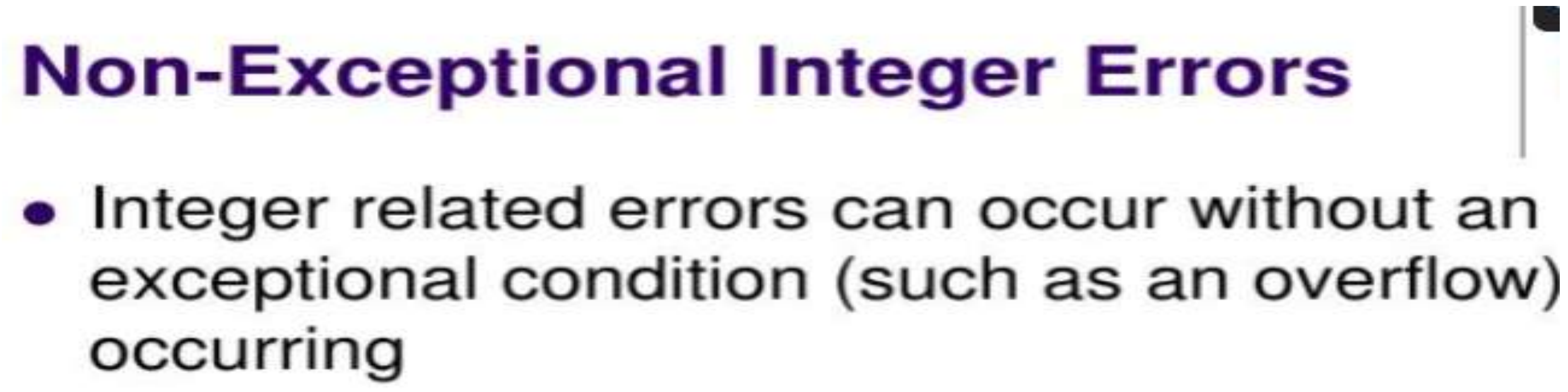
- `cbBuf` is temporarily stored in the unsigned short `bufSize`.
- The maximum size of an **`unsigned short`** for both GCC and the Visual C++ compiler on IA-32 is 65,535.
- The maximum value for a **`signed long`** on the same platform is 2,147,483,647.
- A truncation error will occur on line 2 for any values of `cbBuf` between 65,535 and 2,147,483,647.

UNIT-4—Integer Security

Vulnerability 2



- This would only be an error and not a vulnerability if `bufSize` were used for both the calls to **`malloc()`** and **`memcpy()`**
- Because `bufSize` is used to allocate the size of the buffer and `cbBuf` is used as the size on the call to **`memcpy()`** it is possible to overflow `buf` by anywhere from 1 to 2,147,418,112 (2,147,483,647 - 65,535) bytes.

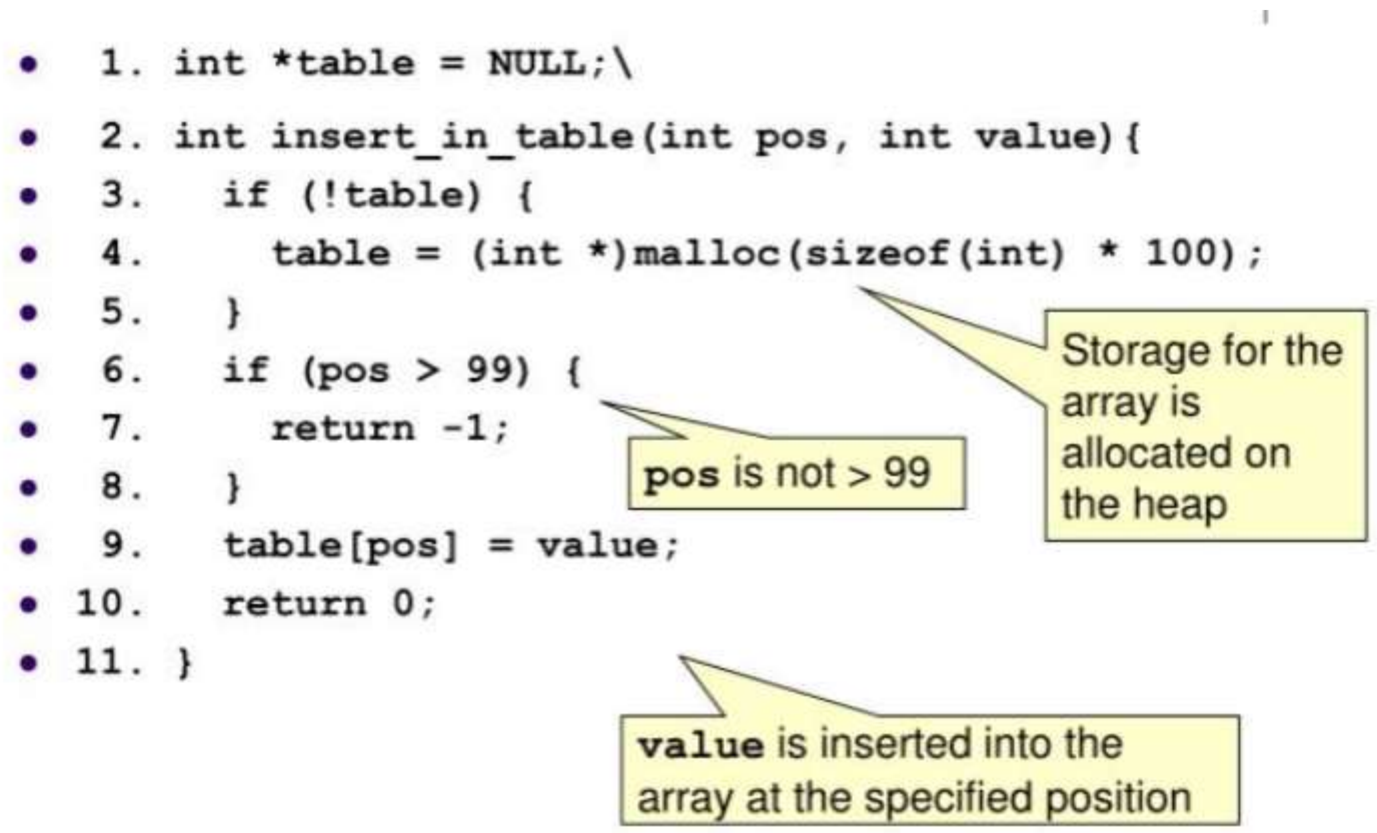


Non-Exceptional Integer Errors

- Integer related errors can occur without an exceptional condition (such as an overflow) occurring

UNIT-4—Integer Security

Negative indices



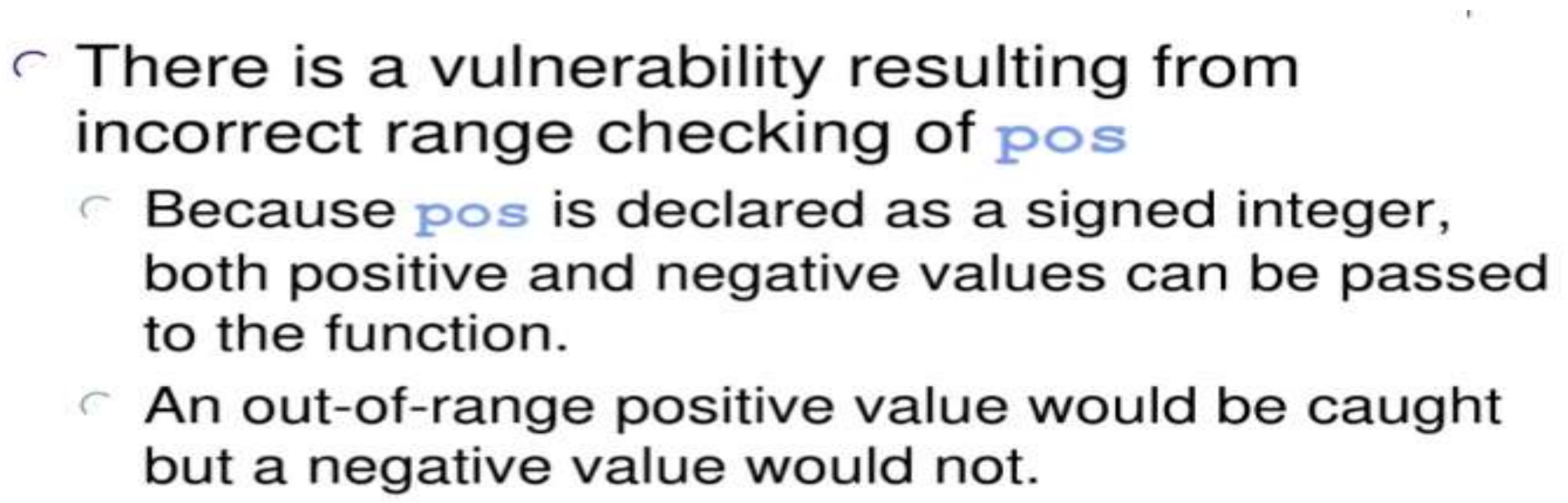
```
int *table = NULL;\
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

Storage for the array is allocated on the heap

pos is not > 99

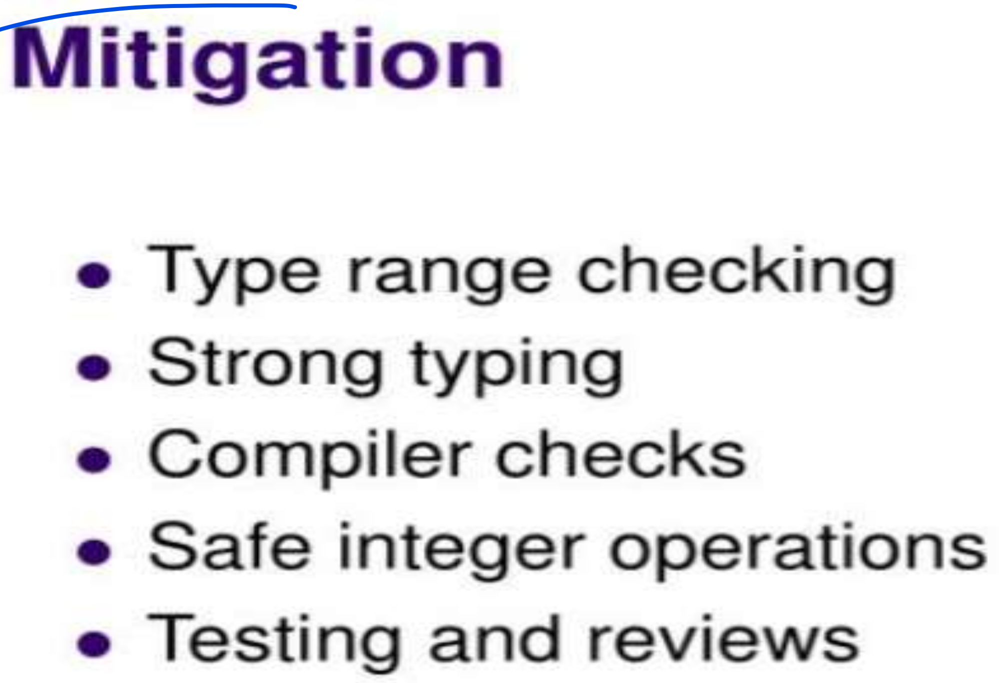
value is inserted into the array at the specified position

Vulnerability



- There is a vulnerability resulting from incorrect range checking of pos
  - Because pos is declared as a signed integer, both positive and negative values can be passed to the function.
  - An out-of-range positive value would be caught but a negative value would not.

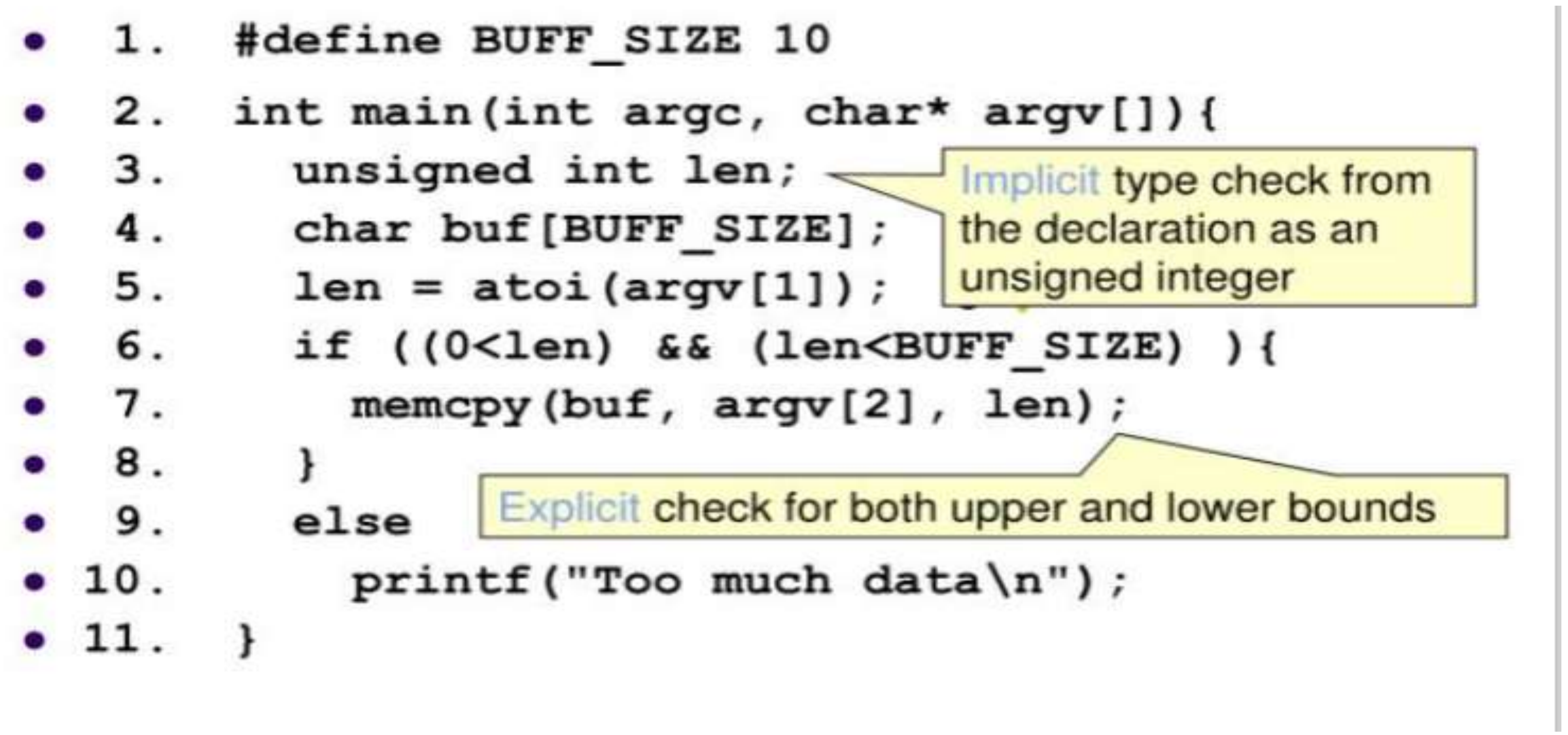
UNIT-4—Integer Security



Mitigation

- Type range checking
- Strong typing
- Compiler checks
- Safe integer operations
- Testing and reviews

Type Range Checking Example

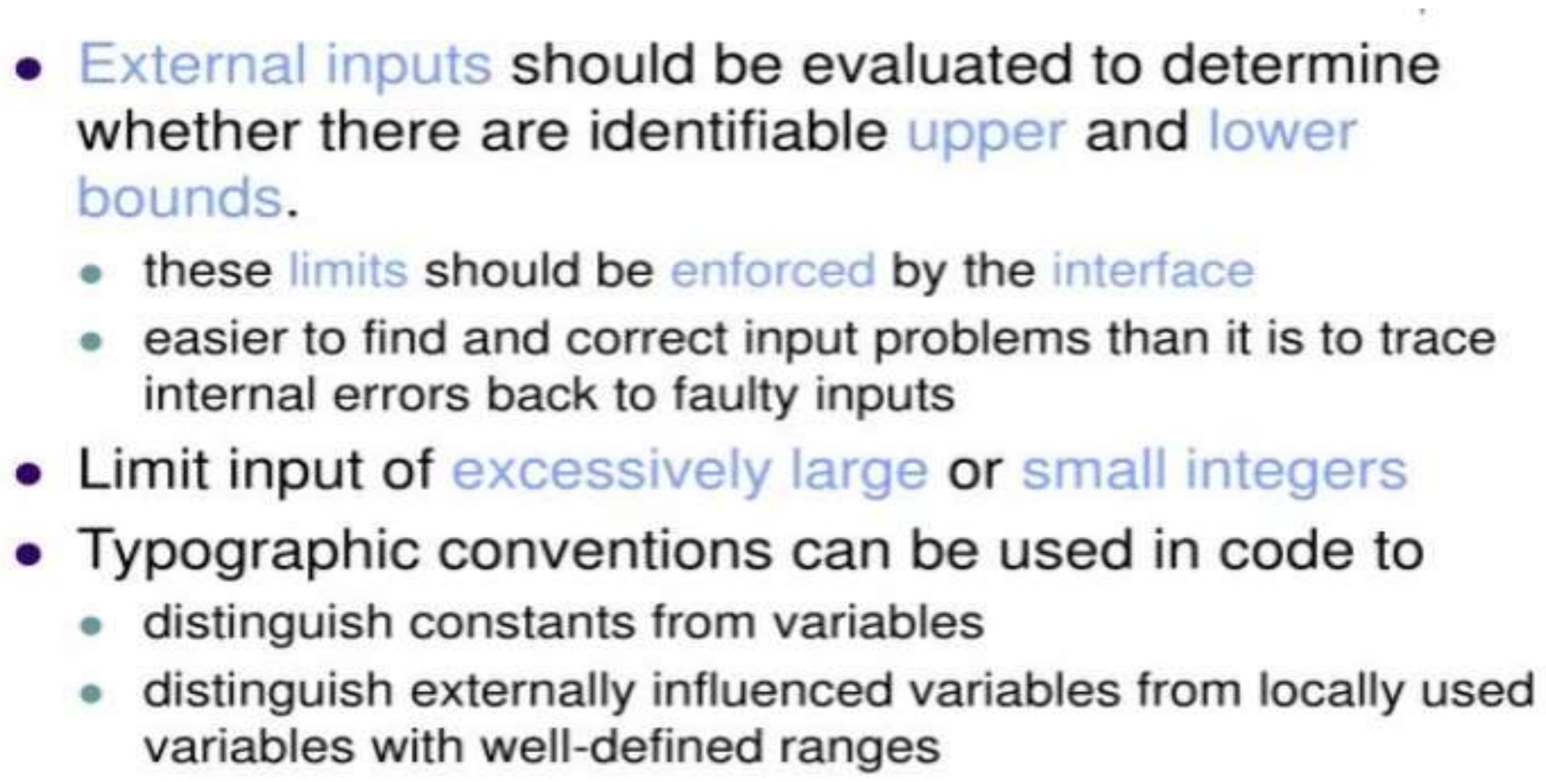


```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  unsigned int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if ((0<len) && (len<BUFF_SIZE) ){
    memcpy(buf, argv[2], len);
  }
  else
    printf("Too much data\n");
}
```

Implicit type check from the declaration as an unsigned integer

Explicit check for both upper and lower bounds

Range Checking

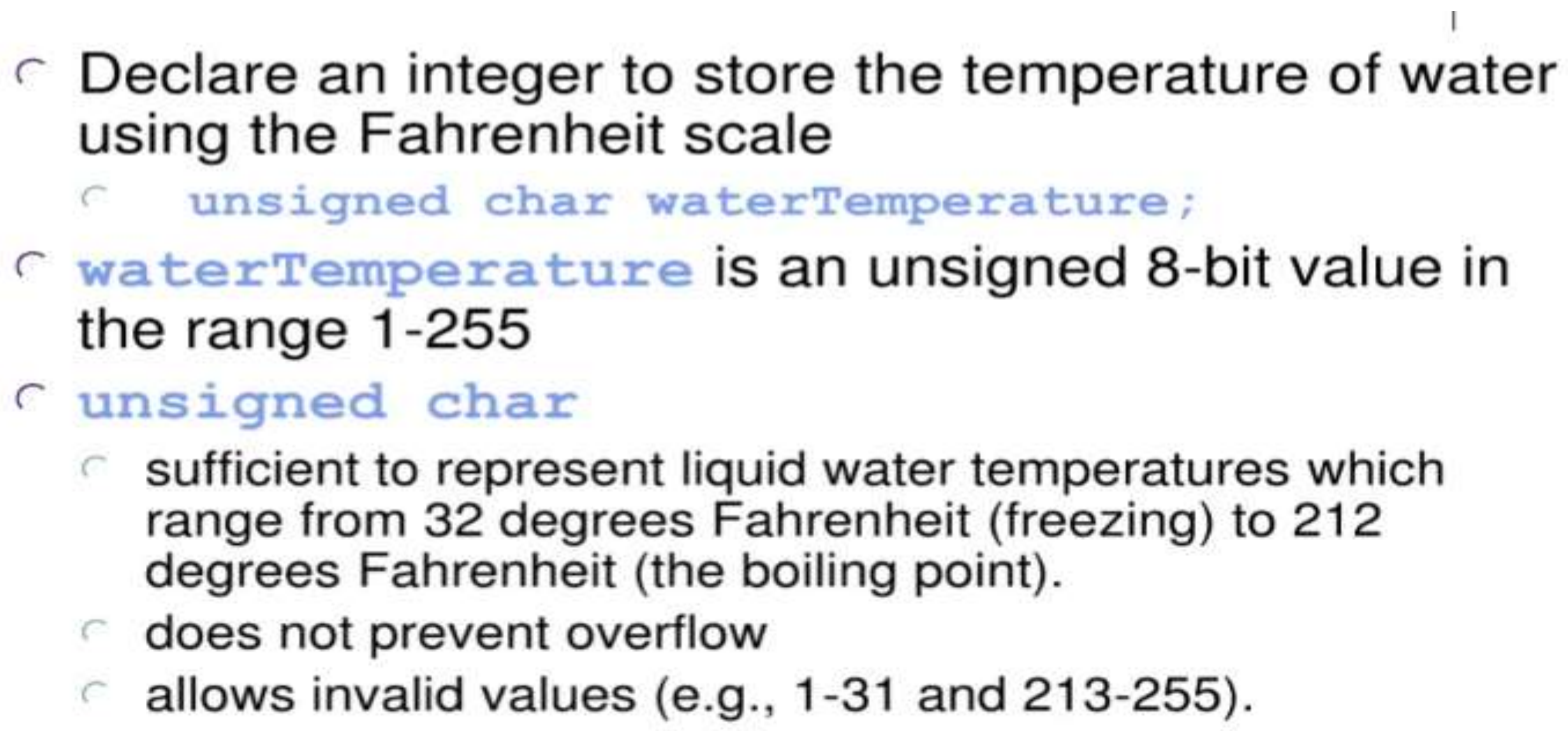


- External inputs should be evaluated to determine whether there are identifiable upper and lower bounds.
  - these limits should be enforced by the interface
  - easier to find and correct input problems than it is to trace internal errors back to faulty inputs
- Limit input of excessively large or small integers
- Typographic conventions can be used in code to
  - distinguish constants from variables
  - distinguish externally influenced variables from locally used variables with well-defined ranges

Strong Typing

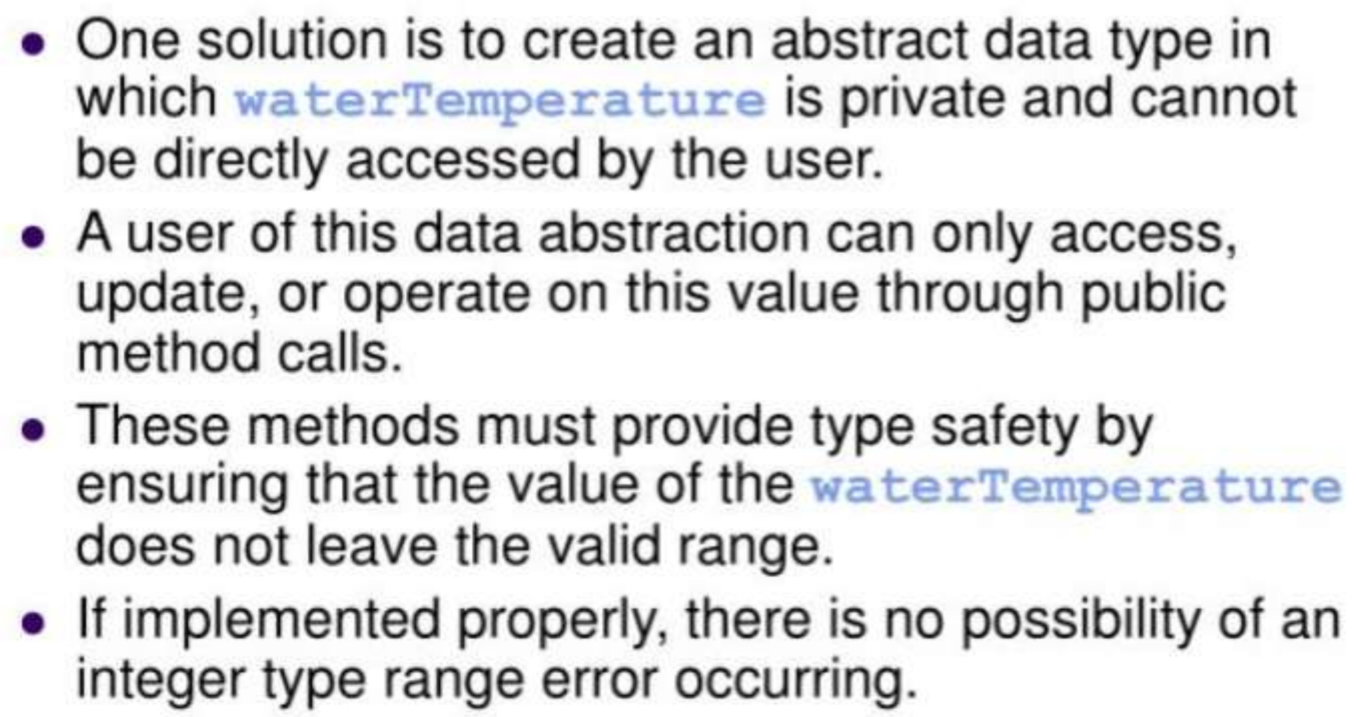
- One way to provide better type checking is to provide better types.
- Using an unsigned type can guarantee that a variable does not contain a negative value.
- This solution does not prevent overflow.
- Strong typing should be used so that the compiler can be more effective in identifying range problems.

Strong Typing Example



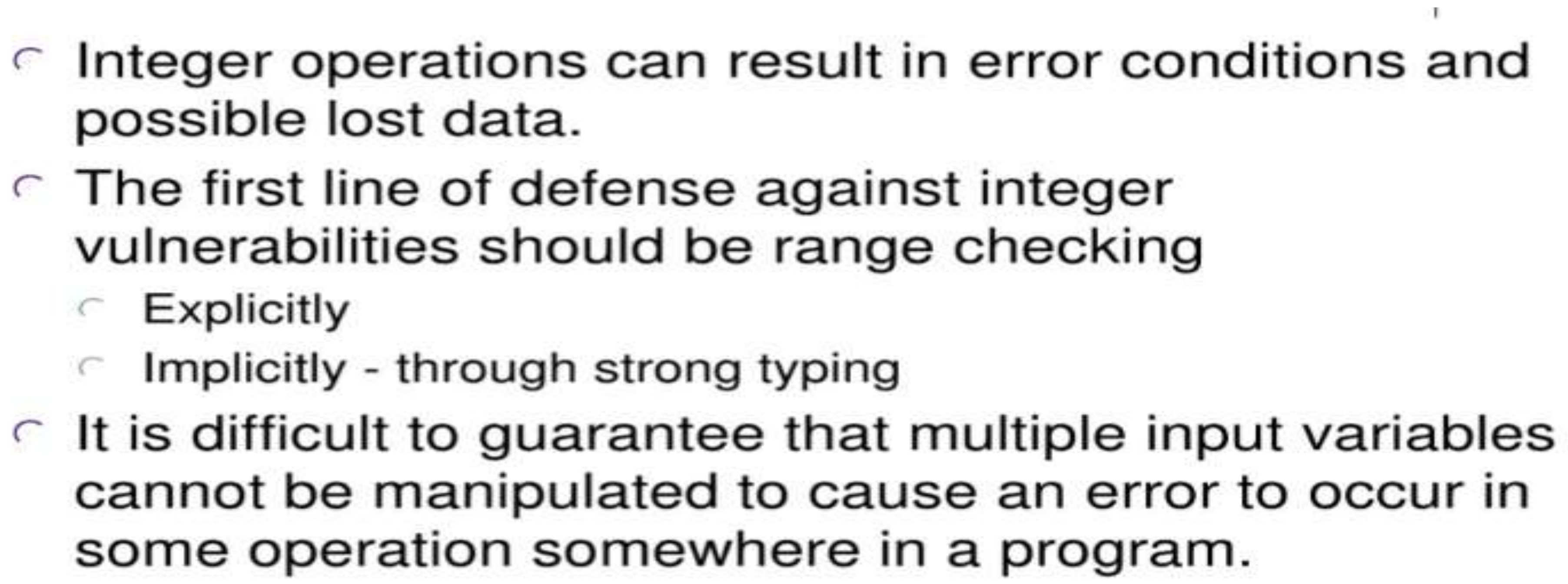
- Declare an integer to store the temperature of water using the Fahrenheit scale
  - `unsigned char waterTemperature;`
- `waterTemperature` is an unsigned 8-bit value in the range 1-255
- `unsigned char`
  - sufficient to represent liquid water temperatures which range from 32 degrees Fahrenheit (freezing) to 212 degrees Fahrenheit (the boiling point).
  - does not prevent overflow
  - allows invalid values (e.g., 1-31 and 213-255).

Abstract Data Type



- One solution is to create an abstract data type in which `waterTemperature` is private and cannot be directly accessed by the user.
- A user of this data abstraction can only access, update, or operate on this value through public method calls.
- These methods must provide type safety by ensuring that the value of the `waterTemperature` does not leave the valid range.
- If implemented properly, there is no possibility of an integer type range error occurring.

Safe Integer Operations-I



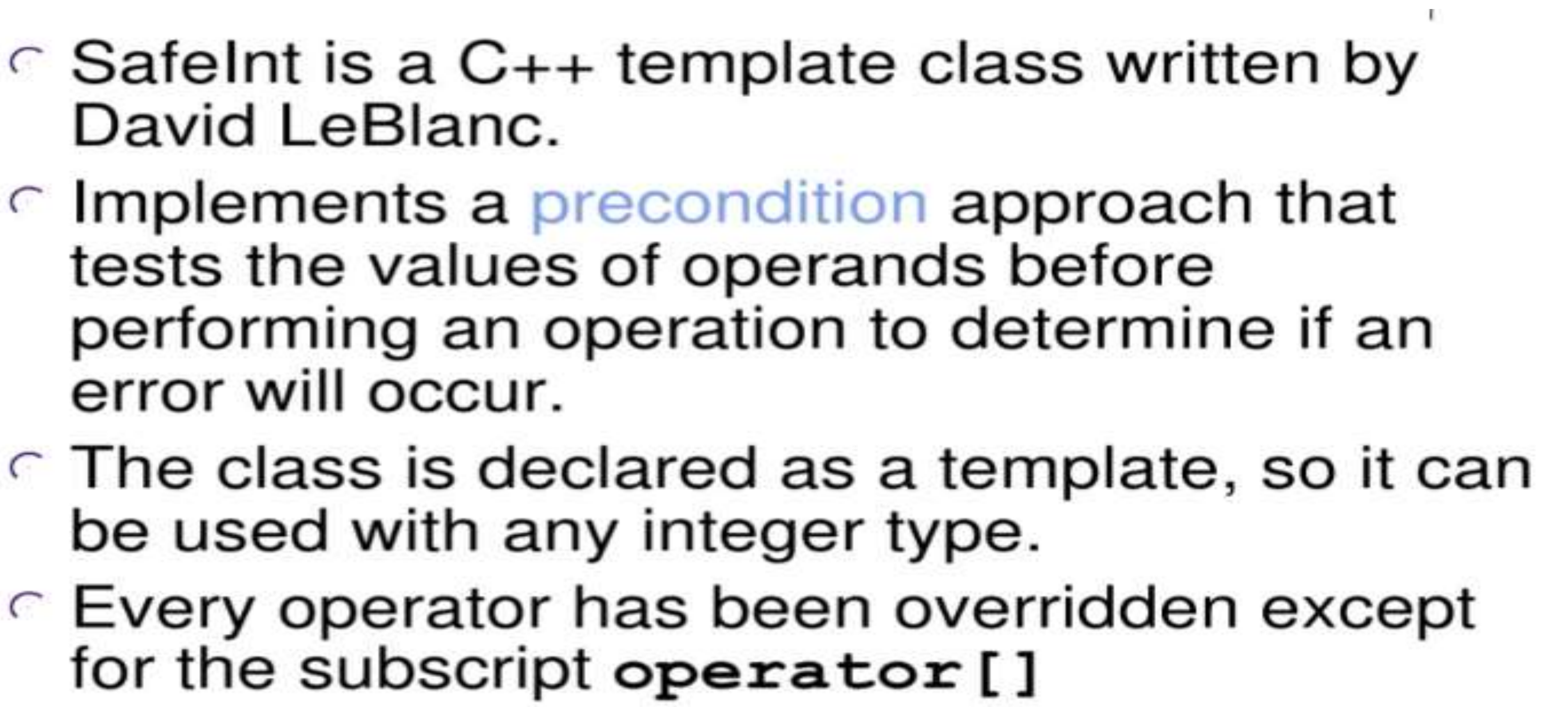
- Integer operations can result in error conditions and possible lost data.
- The first line of defense against integer vulnerabilities should be range checking
  - Explicitly
  - Implicitly - through strong typing
- It is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program.

Safe Integer Operations-II

- An alternative or ancillary approach is to protect each operation.
- This approach can be labor intensive and expensive to perform.
- Use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source.

UNIT-4—Integer Security

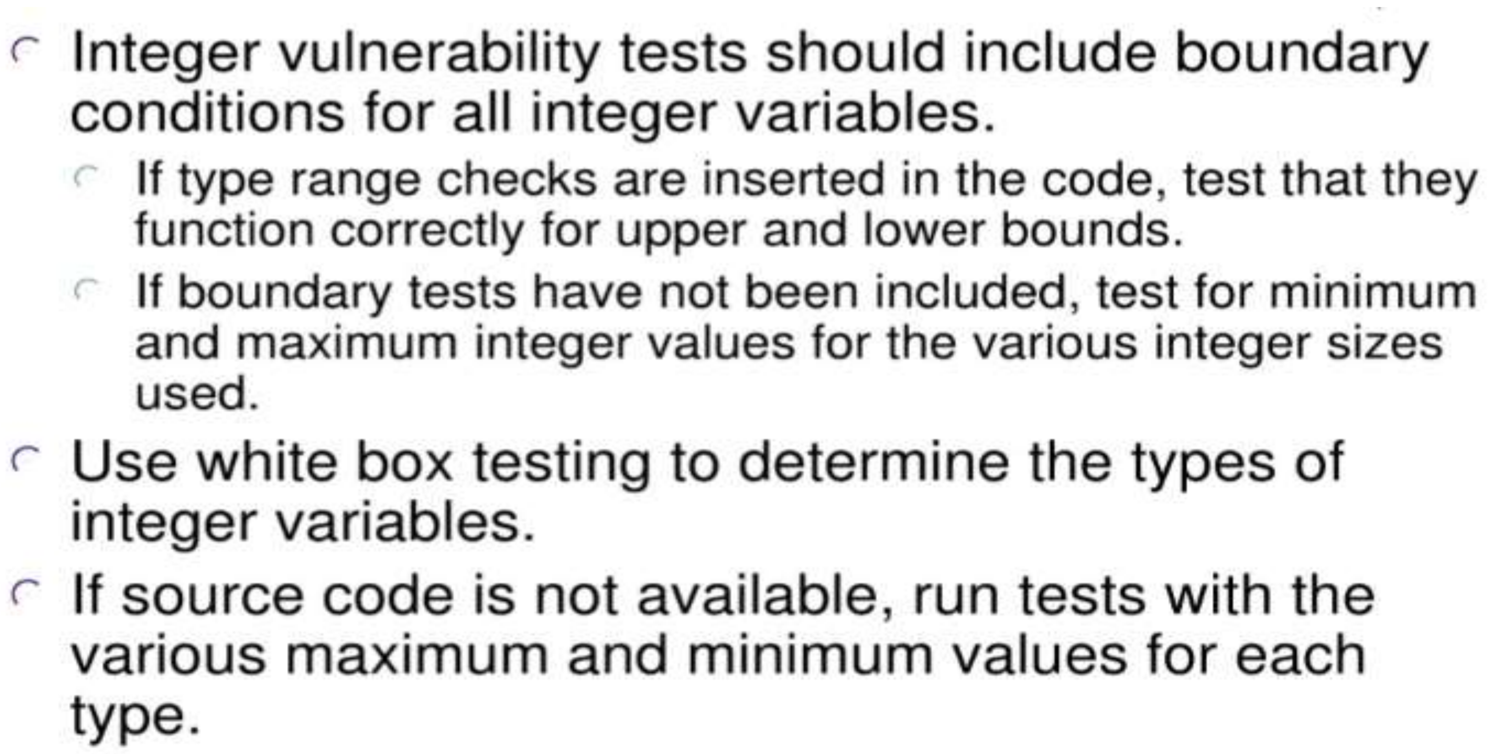
SafeInt Class



- SafeInt is a C++ template class written by David LeBlanc.
- Implements a precondition approach that tests the values of operands before performing an operation to determine if an error will occur.
- The class is declared as a template, so it can be used with any integer type.
- Every operator has been overridden except for the subscript `operator[]`

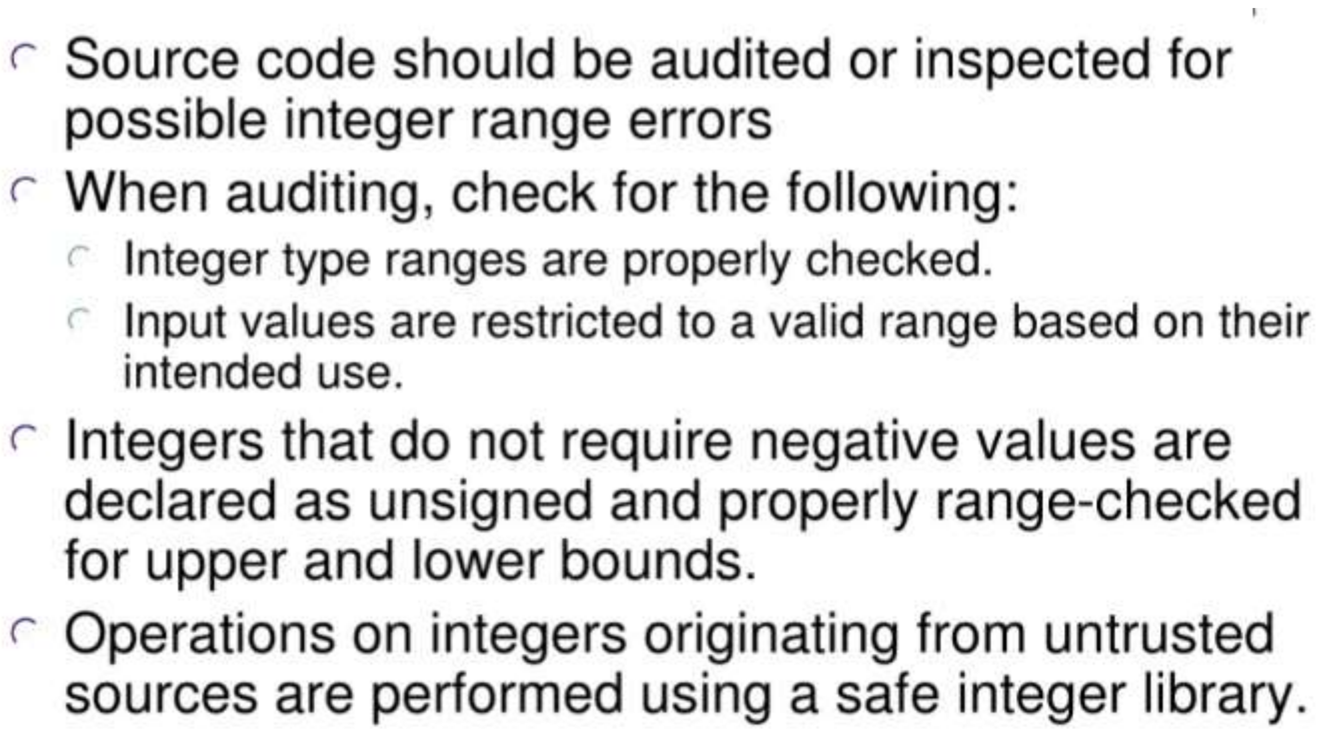
UNIT-4—Integer Security

Testing-I



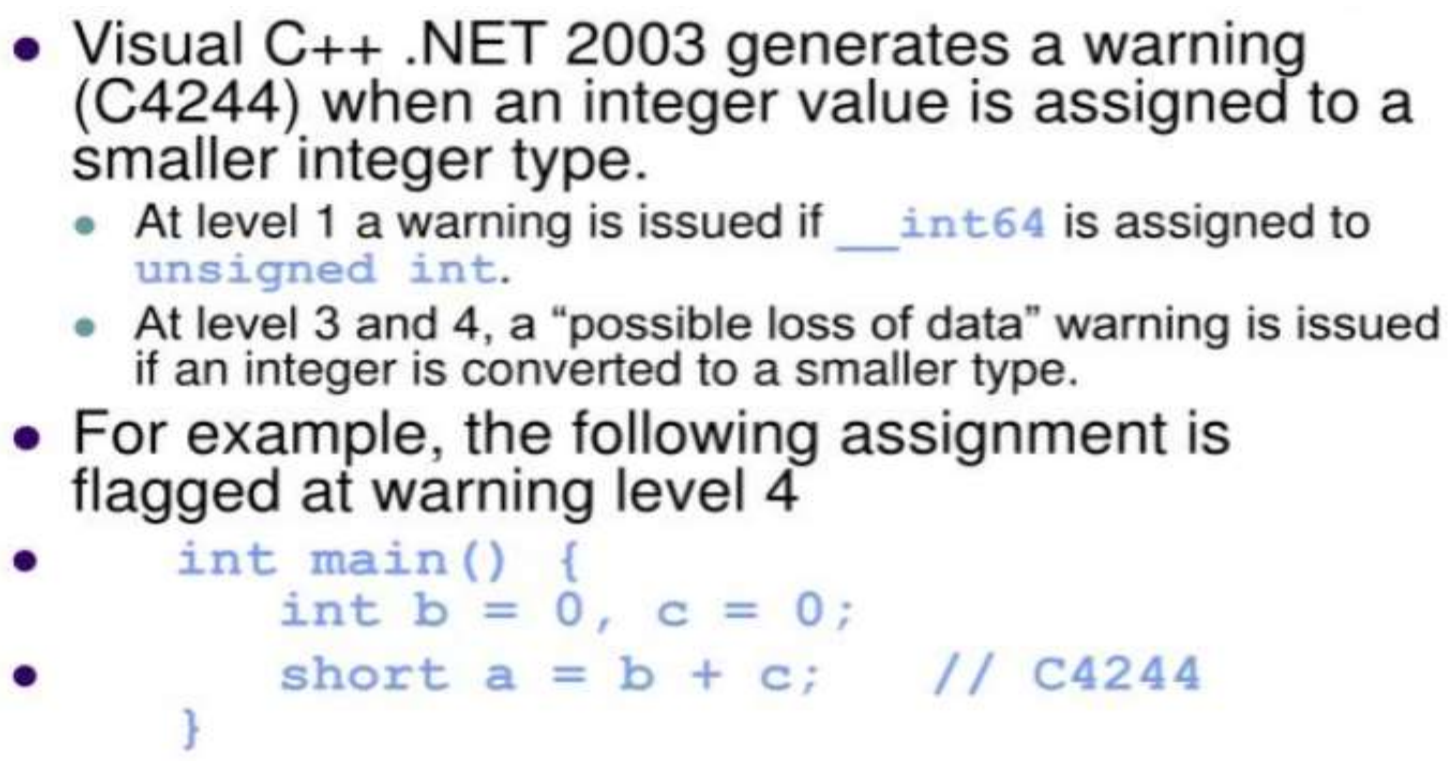
- Integer vulnerability tests should include boundary conditions for all integer variables.
  - If type range checks are inserted in the code, test that they function correctly for upper and lower bounds.
  - If boundary tests have not been included, test for minimum and maximum integer values for the various integer sizes used.
- Use white box testing to determine the types of integer variables.
- If source code is not available, run tests with the various maximum and minimum values for each type.

Source Code Audit



- Source code should be audited or inspected for possible integer range errors
- When auditing, check for the following:
  - Integer type ranges are properly checked.
  - Input values are restricted to a valid range based on their intended use.
- Integers that do not require negative values are declared as unsigned and properly range-checked for upper and lower bounds.
- Operations on integers originating from untrusted sources are performed using a safe integer library.

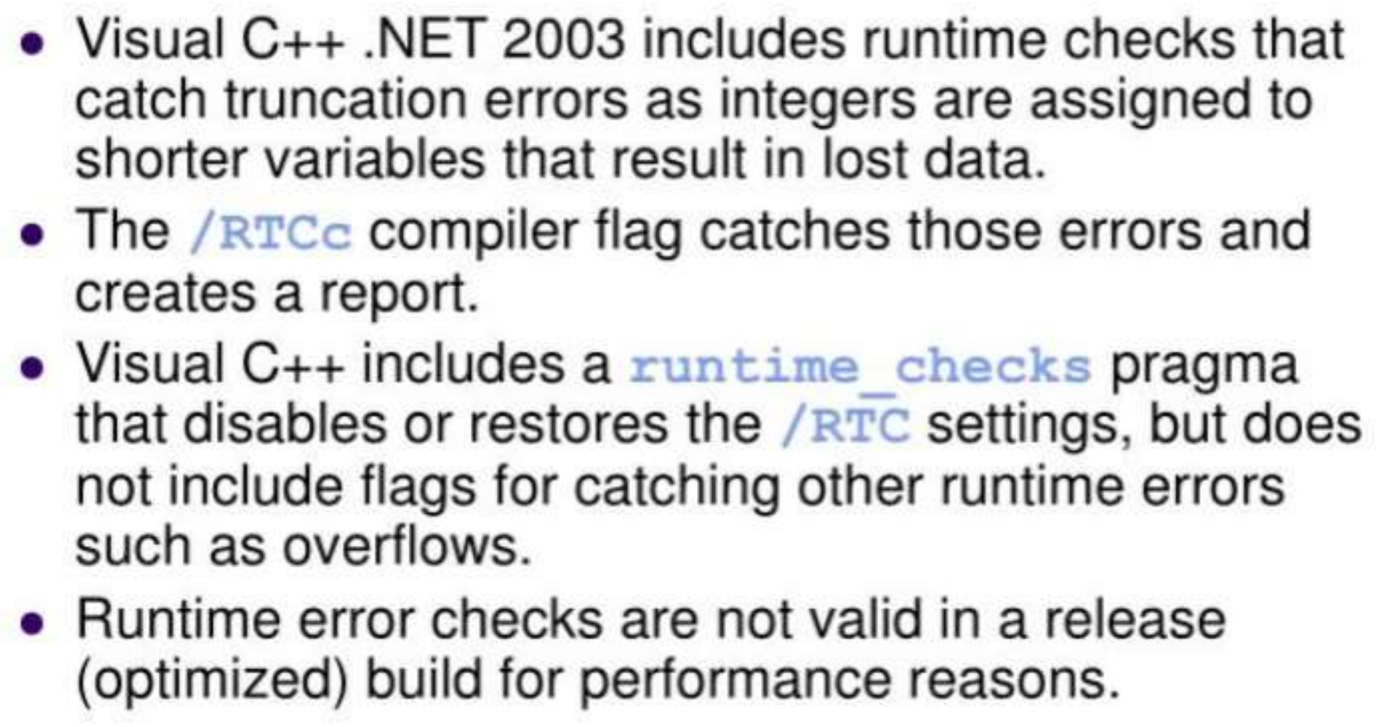
Compiler Checks



- Visual C++ .NET 2003 generates a warning (C4244) when an integer value is assigned to a smaller integer type.
  - At level 1 a warning is issued if `__int64` is assigned to `unsigned int`.
  - At level 3 and 4, a "possible loss of data" warning is issued if an integer is converted to a smaller type.
- For example, the following assignment is flagged at warning level 4

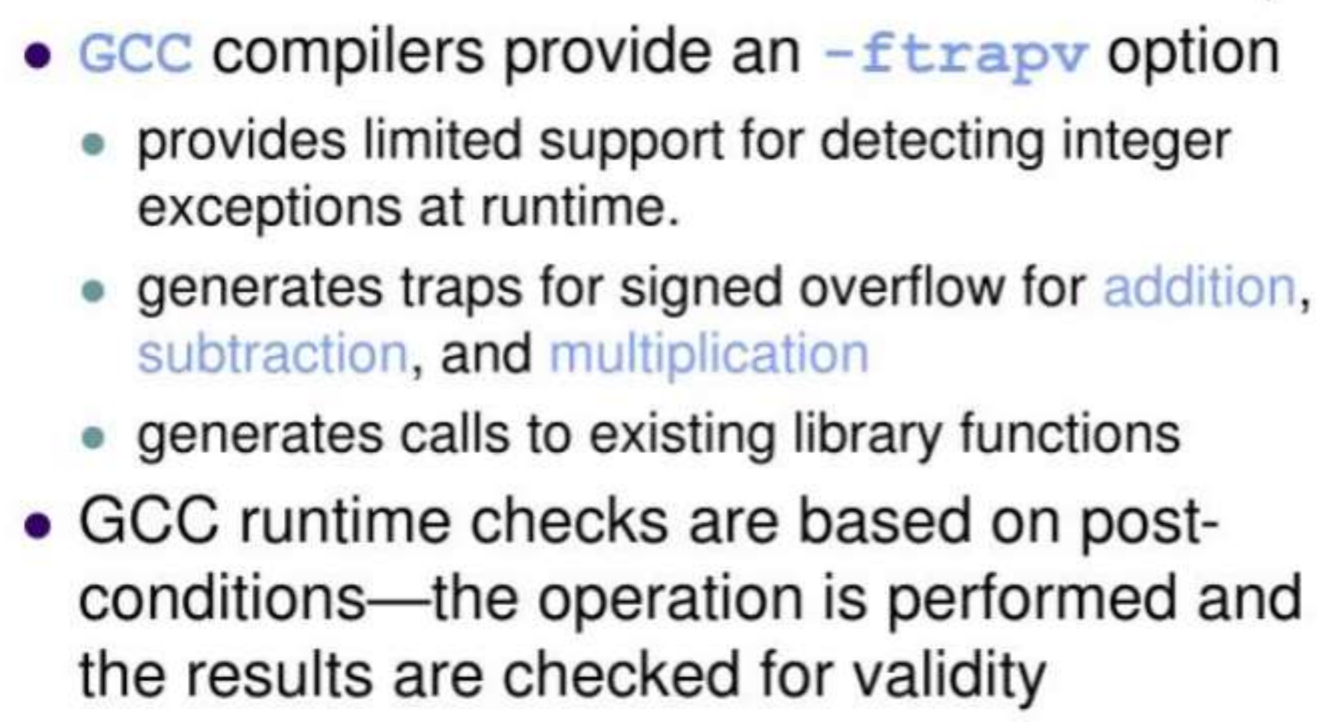
```
•   int main() {
        int b = 0, c = 0;
•       short a = b + c;    // C4244
    }
```

Visual C++ Runtime Checks



- Visual C++ .NET 2003 includes runtime checks that catch truncation errors as integers are assigned to shorter variables that result in lost data.
- The `/RTCc` compiler flag catches those errors and creates a report.
- Visual C++ includes a `runtime_checks` pragma that disables or restores the `/RTC` settings, but does not include flags for catching other runtime errors such as overflows.
- Runtime error checks are not valid in a release (optimized) build for performance reasons.

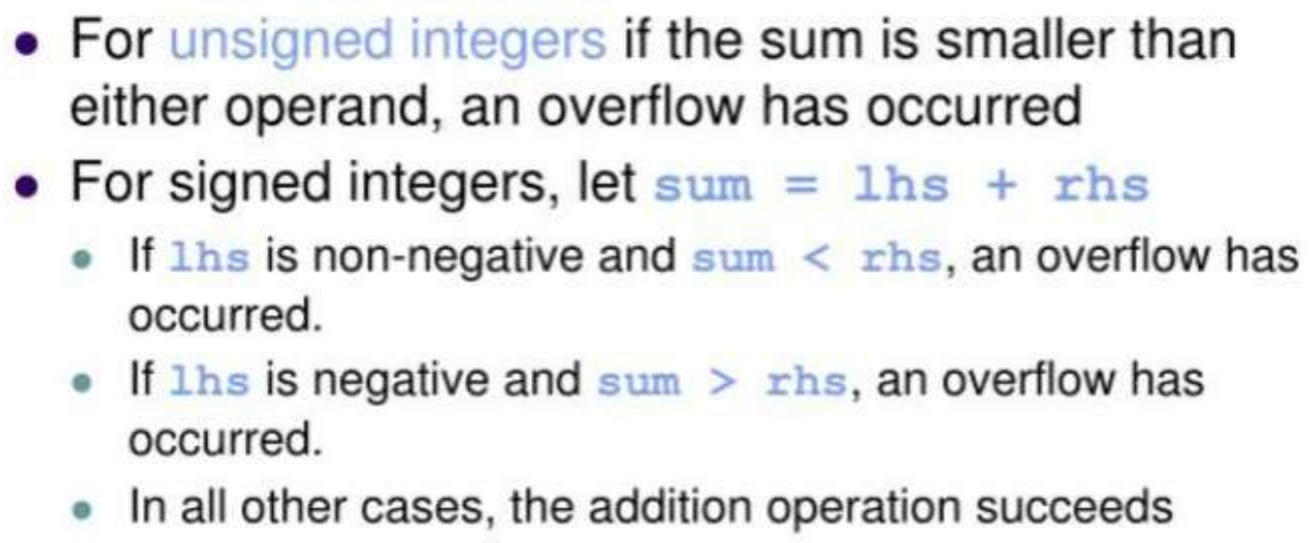
GCC Runtime Checks



- GCC compilers provide an `-ftrapv` option
  - provides limited support for detecting integer exceptions at runtime.
  - generates traps for signed overflow for addition, subtraction, and multiplication
  - generates calls to existing library functions
- GCC runtime checks are based on post-conditions—the operation is performed and the results are checked for validity

UNIT-4—Integer Security

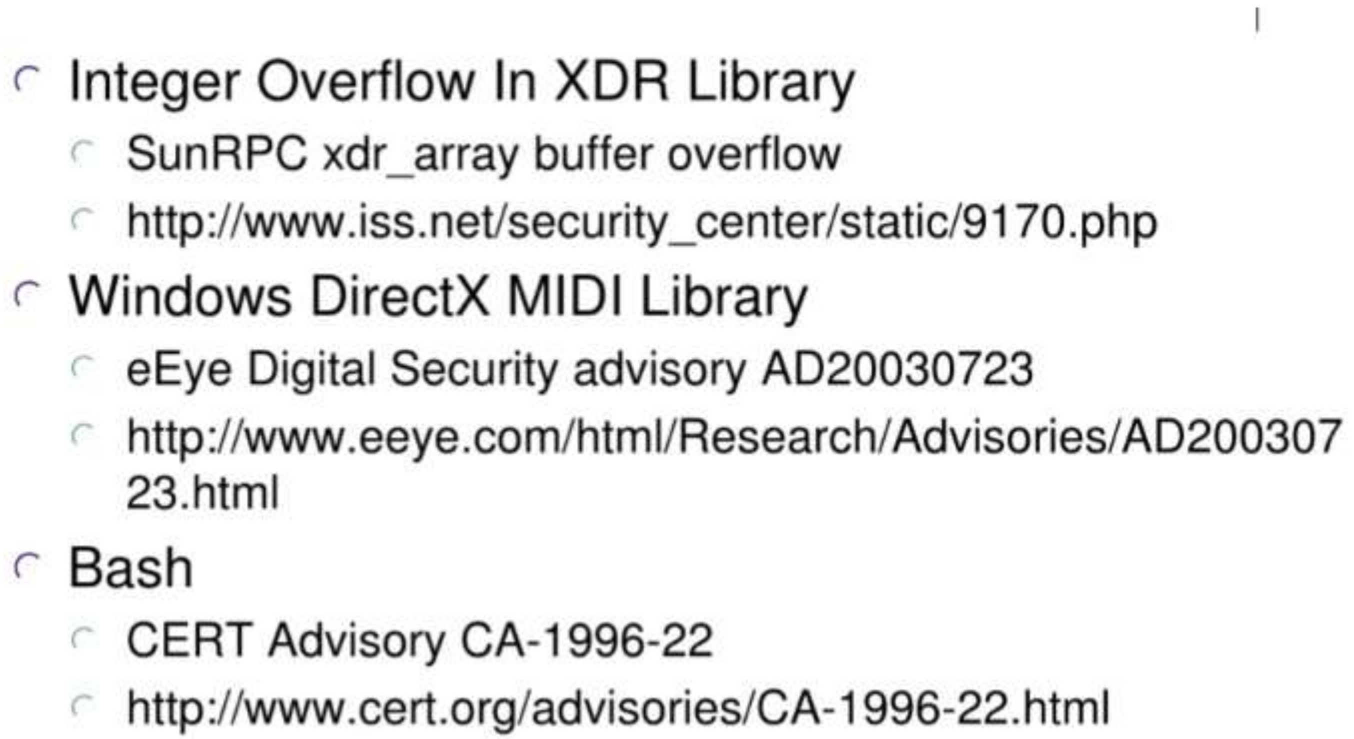
PostCondition



- For unsigned integers if the sum is smaller than either operand, an overflow has occurred
- For signed integers, let `sum = lhs + rhs`
  - If `lhs` is non-negative and `sum < rhs`, an overflow has occurred.
  - If `lhs` is negative and `sum > rhs`, an overflow has occurred.
  - In all other cases, the addition operation succeeds

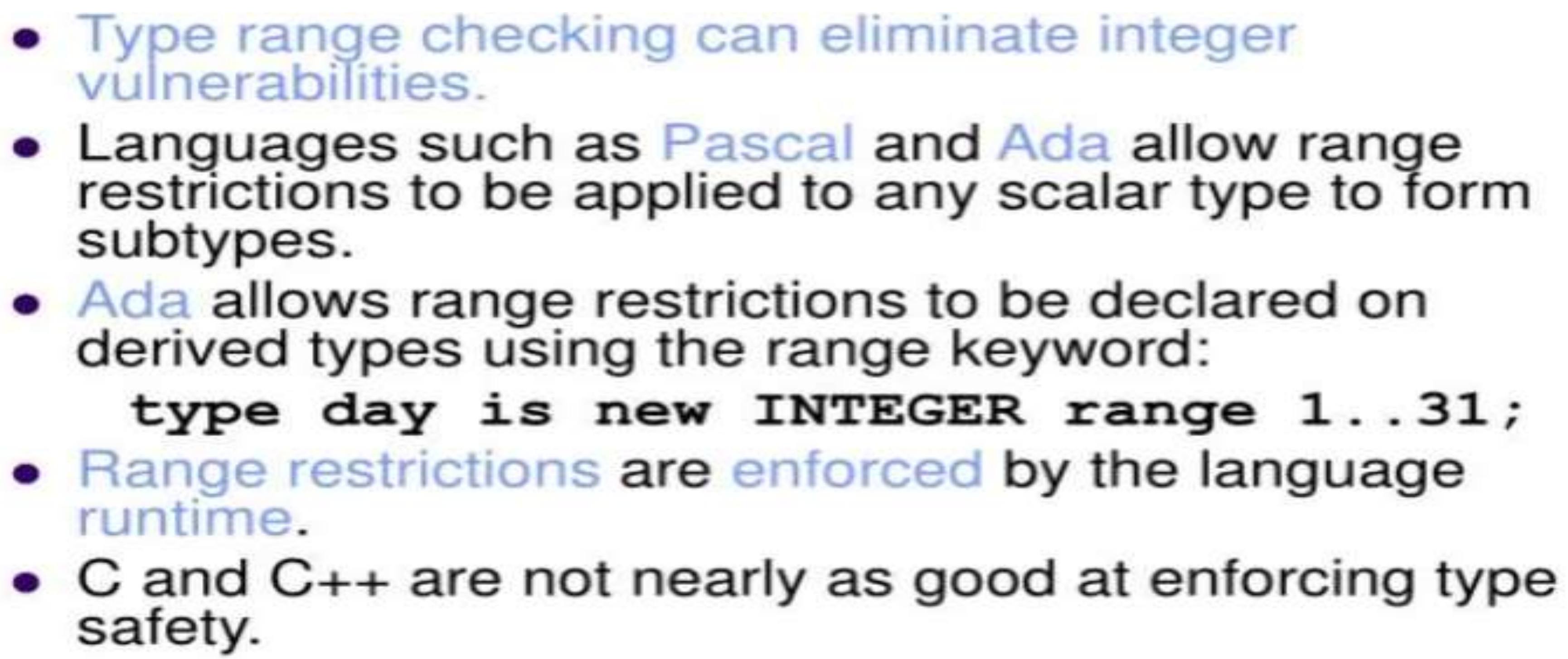
UNIT-4—Integer Security

Notable Vulnerabilities



- Integer Overflow In XDR Library
  - SunRPC xdr_array buffer overflow
  - http://www.iss.net/security_center/static/9170.php
- Windows DirectX MIDI Library
  - eEye Digital Security advisory AD20030723
  - http://www.eeye.com/html/Research/Advisories/AD20030723.html
- Bash
  - CERT Advisory CA-1996-22
  - http://www.cert.org/advisories/CA-1996-22.html

Type Range Checking



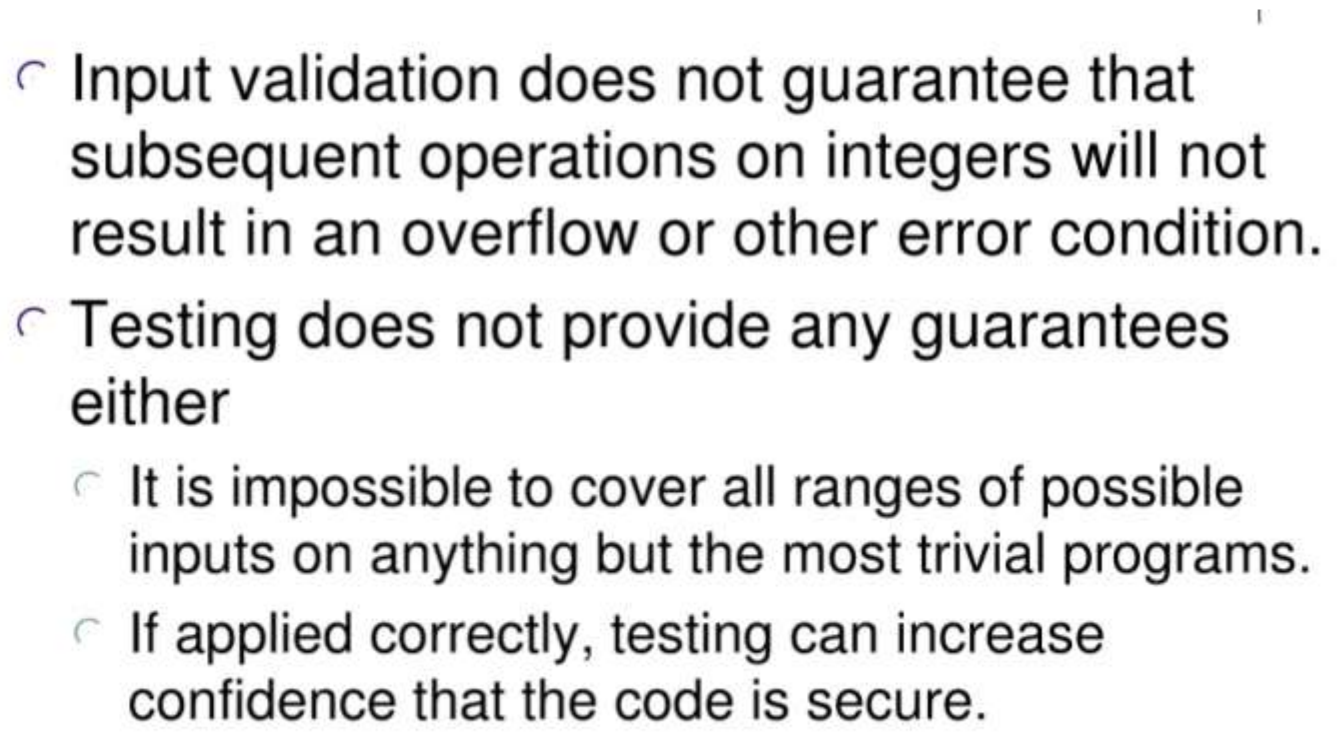
- Type range checking can eliminate integer vulnerabilities.
- Languages such as Pascal and Ada allow range restrictions to be applied to any scalar type to form subtypes.
- Ada allows range restrictions to be declared on derived types using the range keyword:

```
type day is new INTEGER range 1..31;
```

- Range restrictions are enforced by the language runtime.
- C and C++ are not nearly as good at enforcing type safety.

UNIT-4—Integer Security

Testing-I



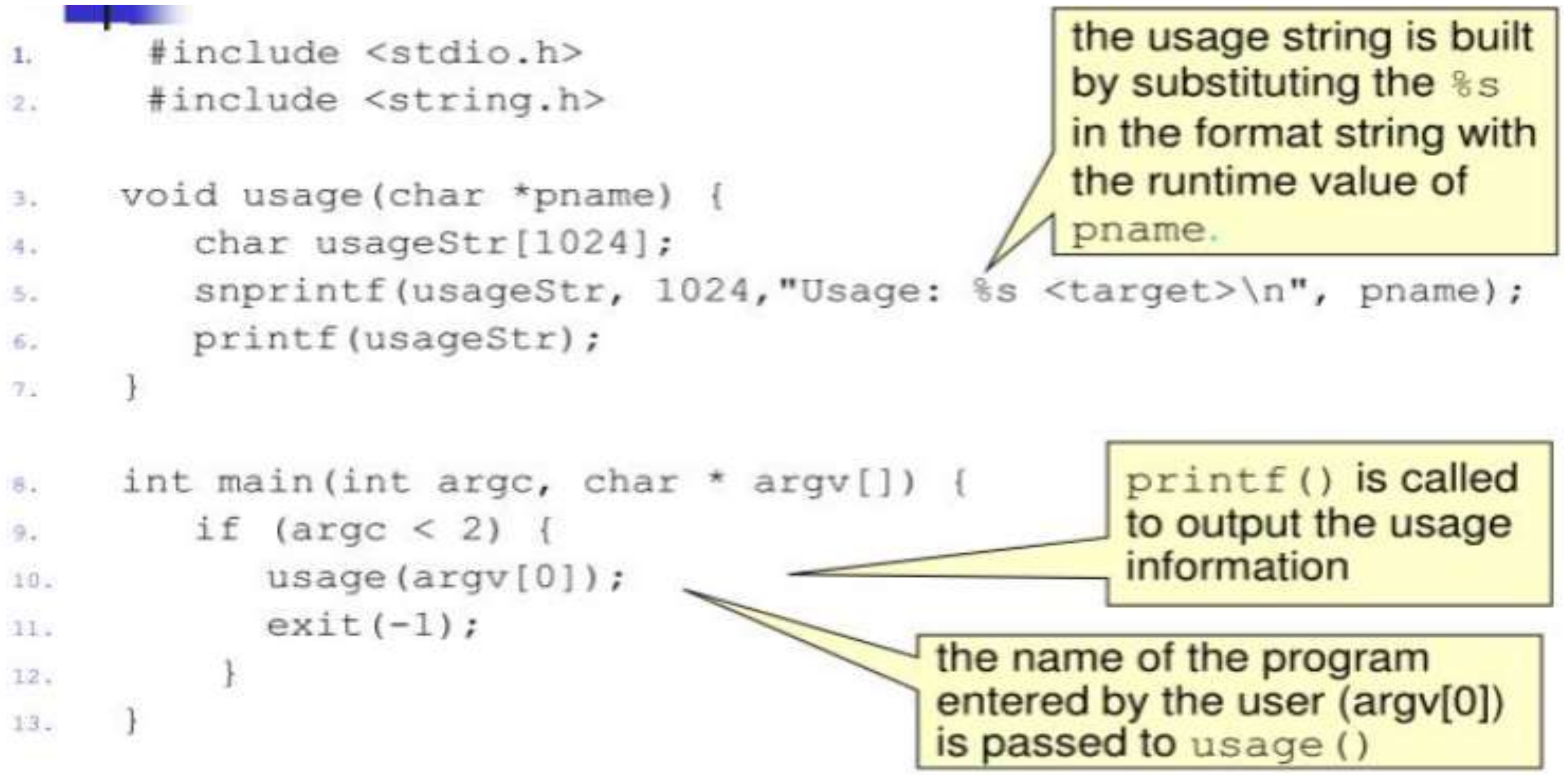
- Input validation does not guarantee that subsequent operations on integers will not result in an overflow or other error condition.
- Testing does not provide any guarantees either
  - It is impossible to cover all ranges of possible inputs on anything but the most trivial programs.
  - If applied correctly, testing can increase confidence that the code is secure.

UNIT-4—Formatted Output

Formatted output functions consist of a format string and a variable number of arguments. The format string, in effect, provides a set of instructions that are interpreted by the formatted output function.
By controlling the content of the format string, a user can, in effect, control execution of the formatted output function.

UNIT-4—Formatted Output

Sample Vulnerable Program



```
#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);
    }
}
```

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

printf() is called to output the usage information

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

UNIT-4—Formatted Output

```
#include<stdio.h>
int main()
{

char smallBuffer[10];
char *text=“Hello, World!”;

int written =snprintf(smallBuffer, sizeof(smallBuffer), “%s”, text);

printf(“Truncated Output %s\n”,smallBuffer);
printf(“Characters Needed: %d\n”, written);

return 0;
}
```

```
Output: Truncated Output: Hello , Wo
Characters Needed: 13
```

UNIT-4—Formatted Output

Variadic Functions

- ❑ variadic function is a function that can accept a variable number of arguments.

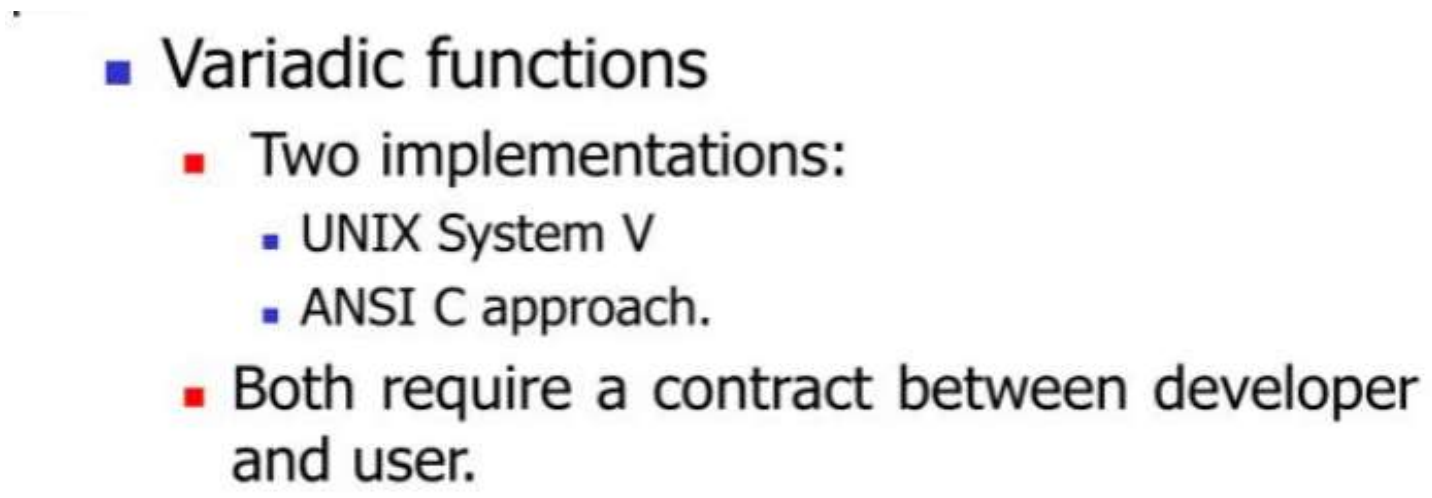
- ❑ In both C and C++, variadic functions are useful when you do not know in advance how many arguments will be passed to the function.

- ❑ The most common use of variadic functions is for functions like printf and scanf, which can accept a different number of arguments depending on the format.

UNIT-4—Formatted Output

Variadic Functions:-

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.

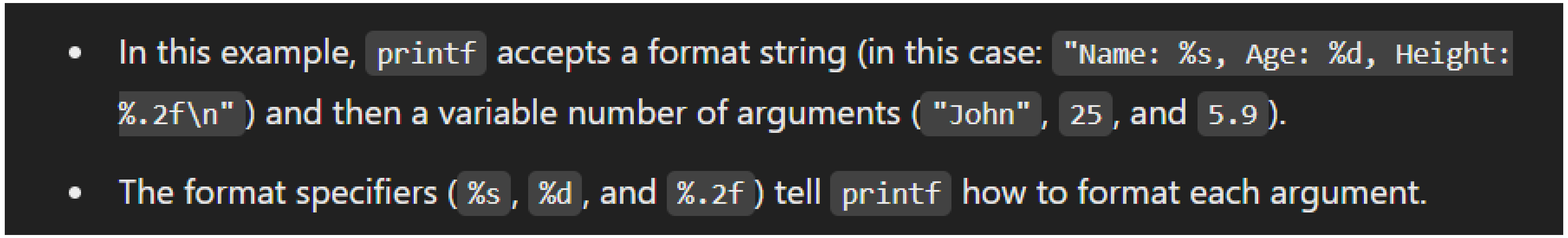


- Variadic functions
  - Two implementations:
    - UNIX System V
    - ANSI C approach.
  - Both require a contract between developer and user.

UNIT-4—Formatted Output

Variadic Functions

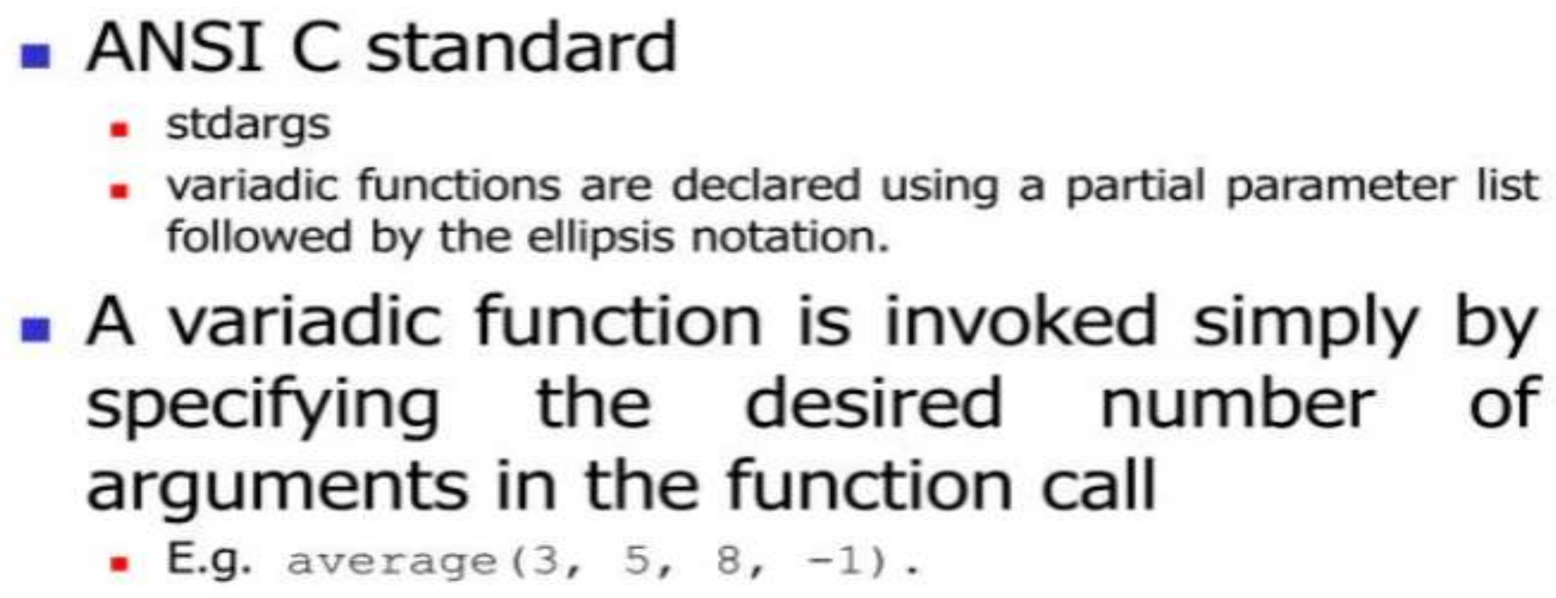
```
#include <stdio.h>

int main() {
    // Printing a string, an integer, and a floating-point number
    printf("Name: %s, Age: %d, Height: %.2f\n", "John", 25, 5.9);
    return 0;
}
```



- In this example, `printf` accepts a format string (in this case: `"Name: %s, Age: %d, Height: %.2f\n"`) and then a variable number of arguments (`"John"`, `25`, and `5.9`).
- The format specifiers (`%s`, `%d`, and `%.2f`) tell `printf` how to format each argument.

UNIT-4—Formatted Output

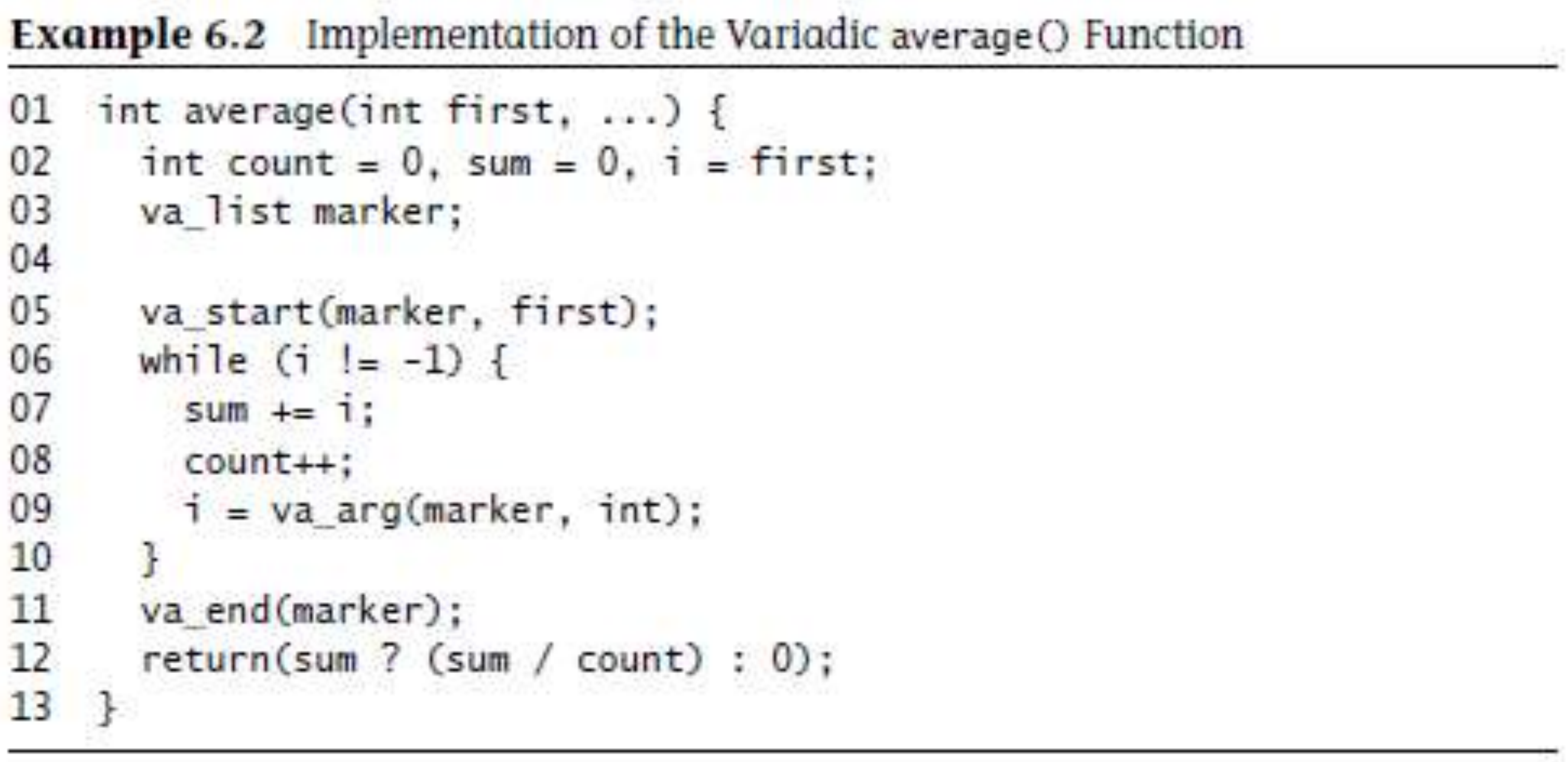
Variadic Functions



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

UNIT-4—Formatted Output

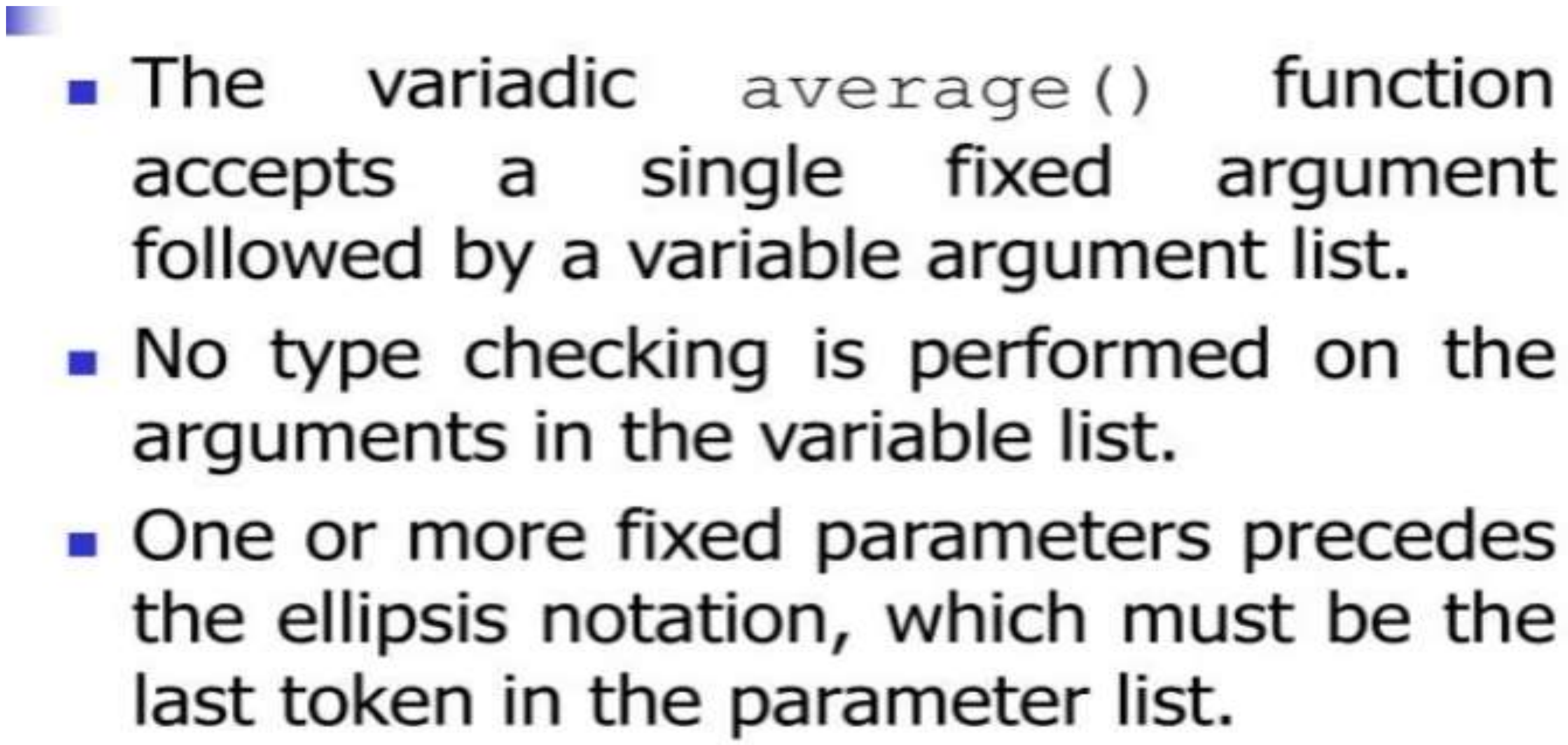
Variadic Functions



Example 6.2 Implementation of the Variadic average() Function

```c
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);
}
```

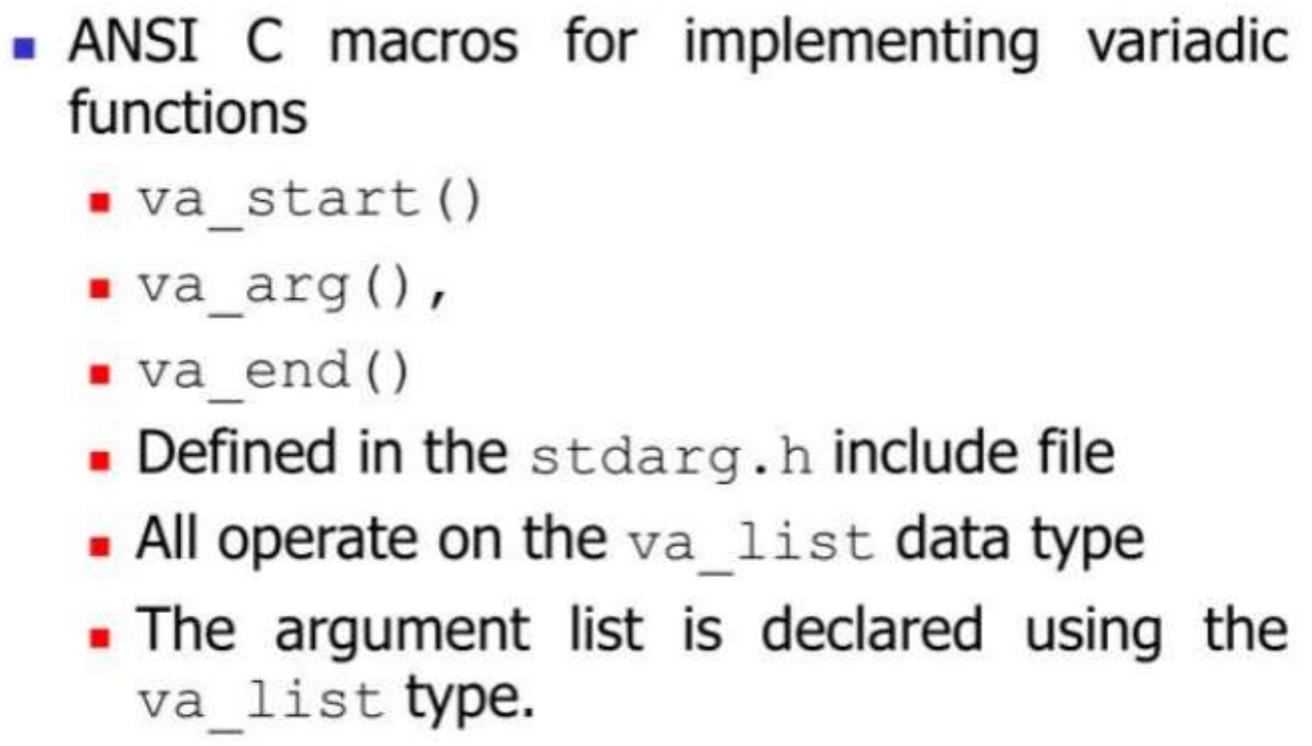
ANSI C



- 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.

UNIT-4—Formatted Output

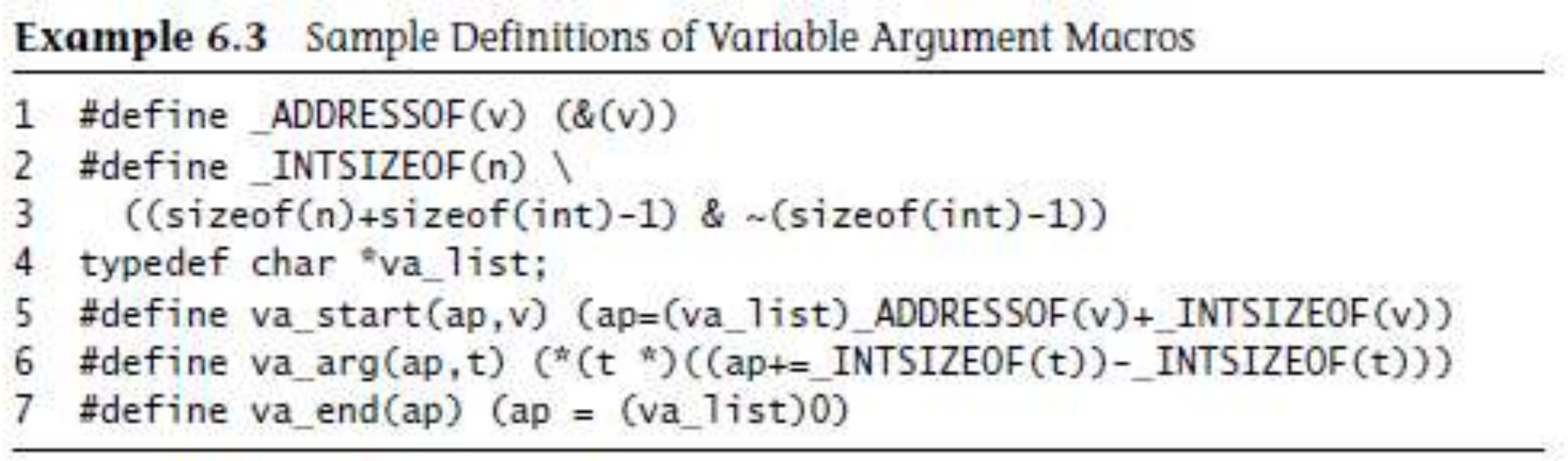
ANSI C



- ANSI C macros for implementing variadic functions
  - `va_start()`
  - `va_arg()`,
  - `va_end()`
  - Defined in the `stdarg.h` include file
  - All operate on the `va_list` data type
  - The argument list is declared using the `va_list` type.

UNIT-4—Formatted Output

Variadic Functions

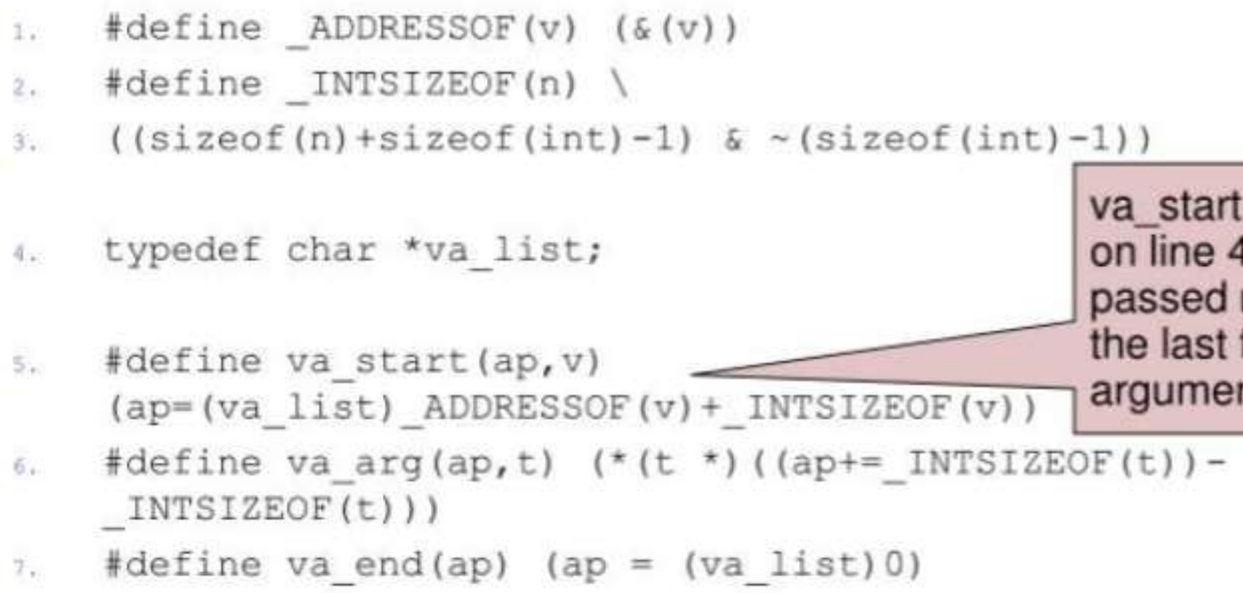


Example 6.3 Sample Definitions of Variable Argument Macros

```
#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)
```

UNIT-4—Formatted Output

Variadic Functions

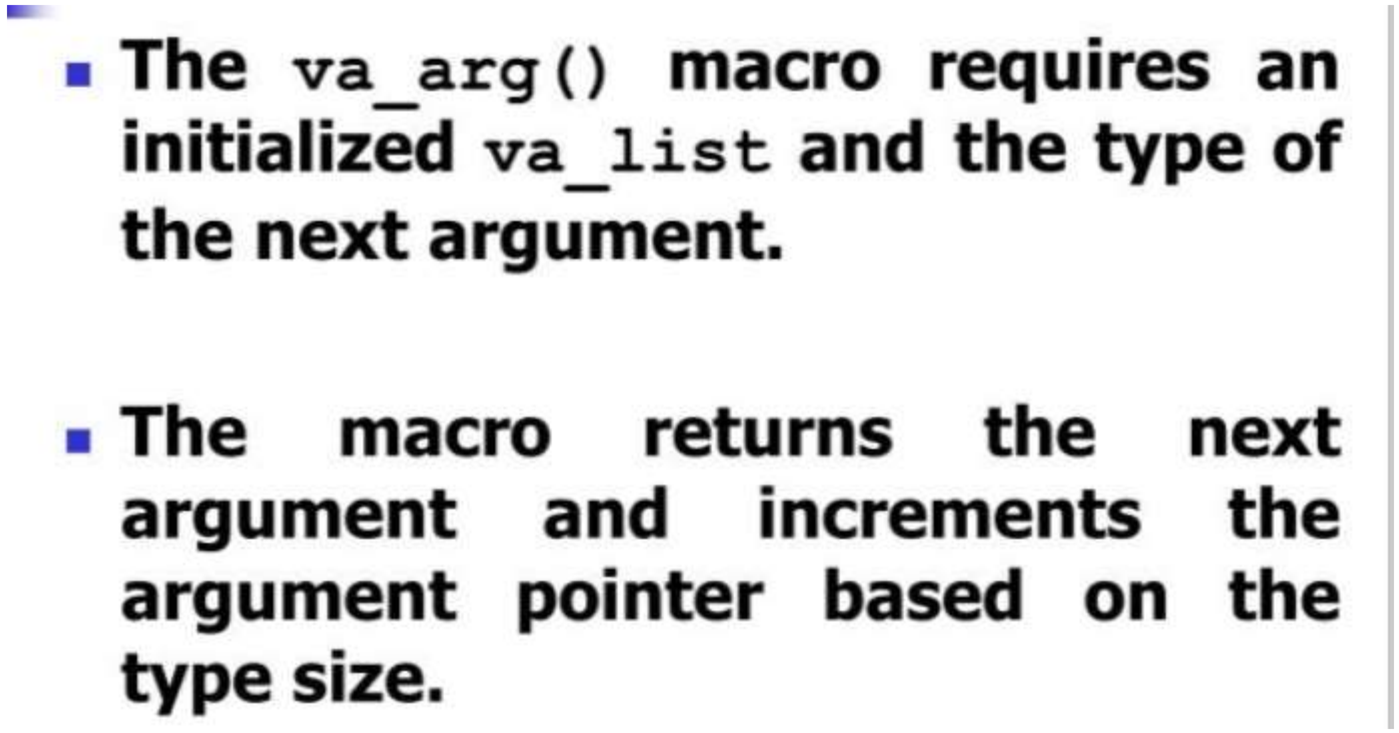


```
1.  #define _ADDRESSOF(v)  (&(v))
2.  #define _INTSIZEOF(n)  \
3.  ((sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1))

4.  typedef char *va_list;

5.  #define va_start(ap,v)
    (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v))
6.  #define va_arg(ap,t)  (*(t *)((ap+=_INTSIZEOF(t))-
    _INTSIZEOF(t)))
7.  #define va_end(ap)  (ap = (va_list)0)
```

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

UNIT-4—Formatted Output

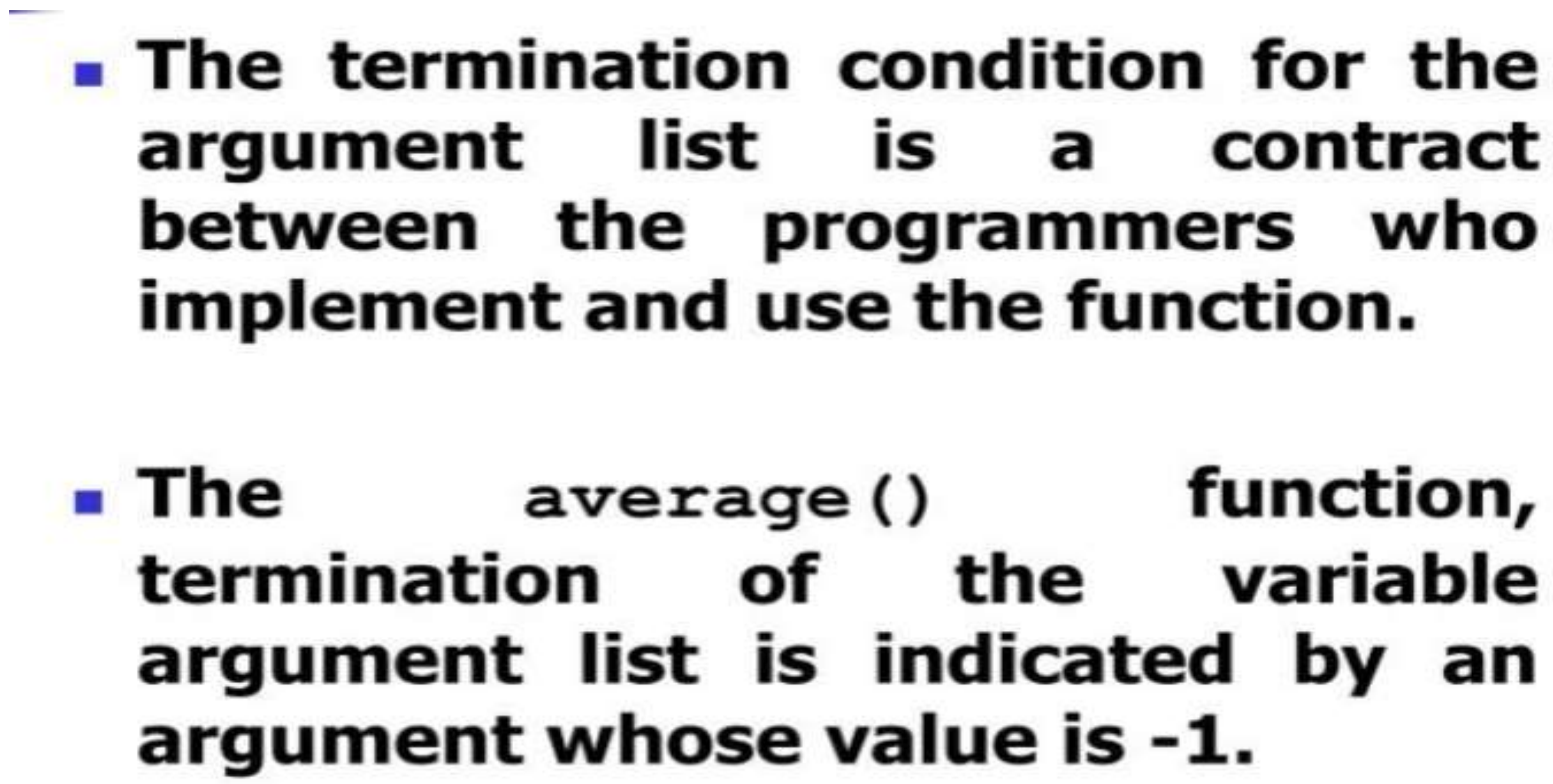
Variadic Functions



- 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.

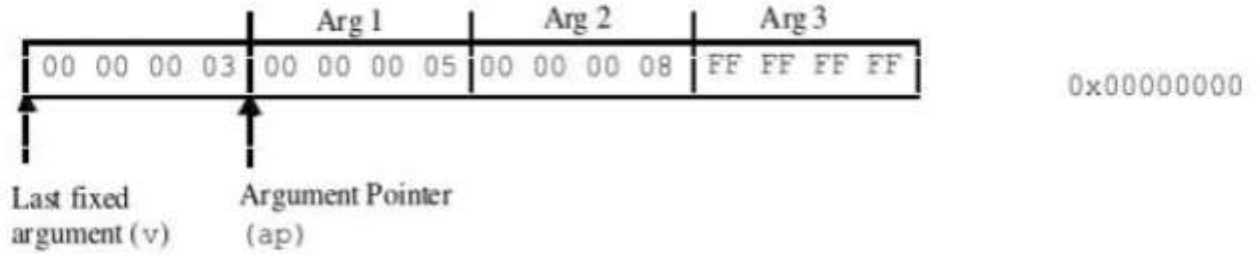
UNIT-4—Formatted Output

Variadic Functions



- 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.

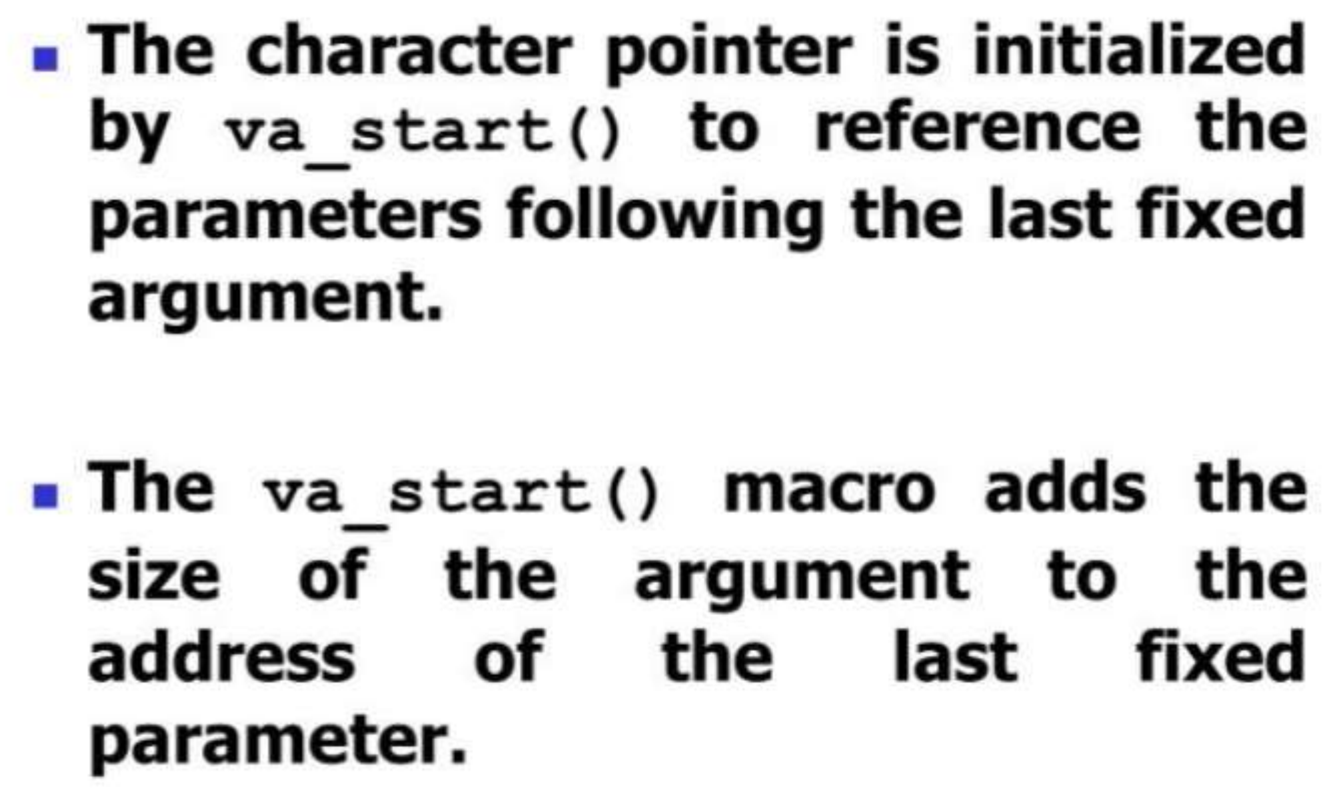
Type va_list as a Character pointer-I



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

UNIT-4—Formatted Output

Type va_list as a Character pointer-2



- 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.

UNIT-4—Formatted Output

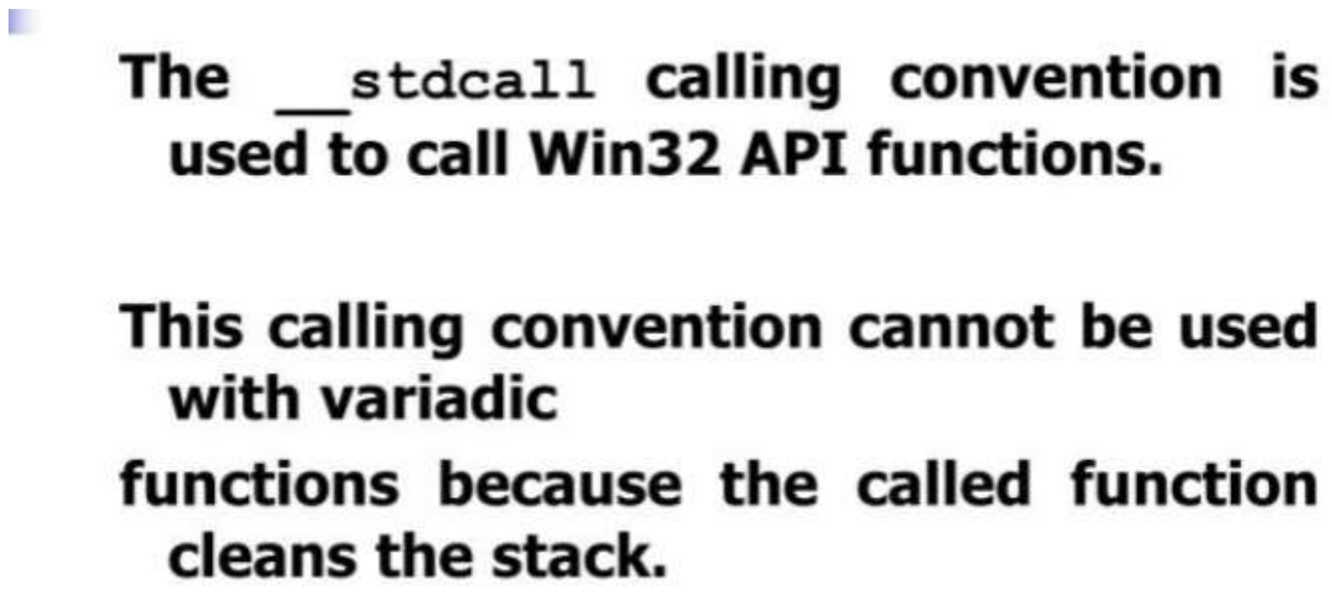
Argument passing and Naming Conventions

_cdecl

- **Parameters are pushed onto the stack in reverse order.**
- **The `__cdecl` calling convention requires that each function call include stack cleanup code.**

UNIT-4—Formatted Output

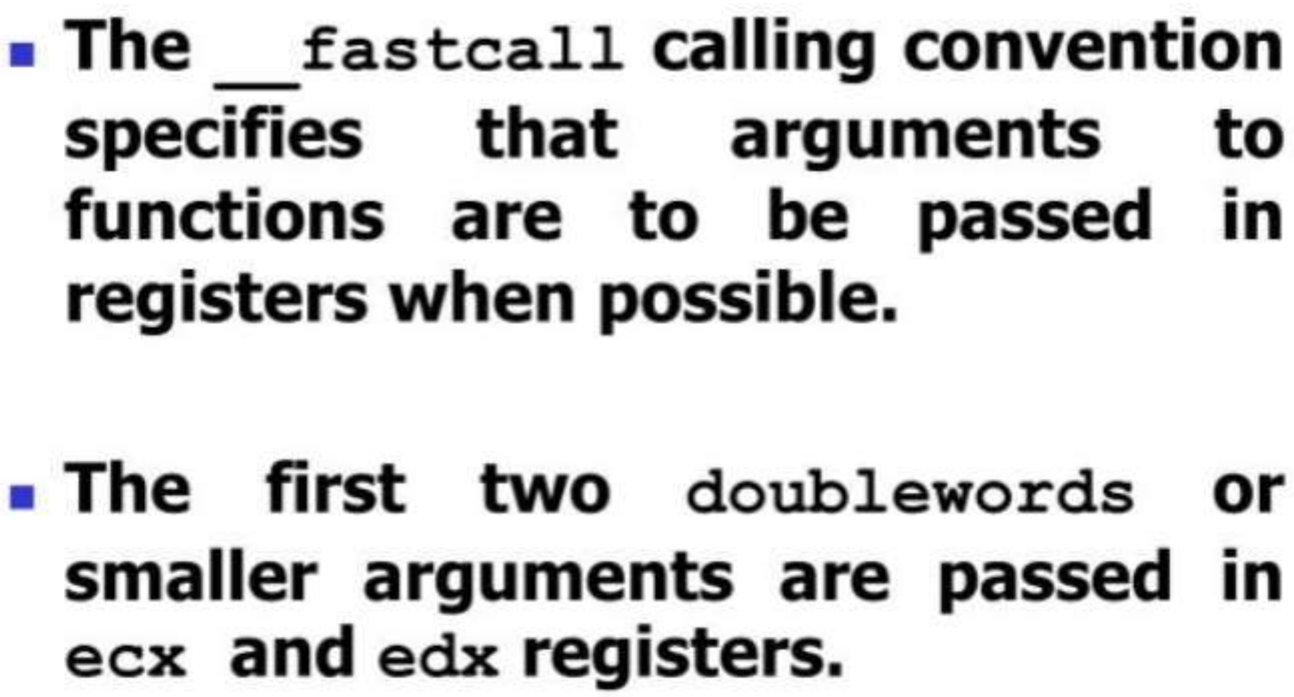
_stdcall



The `__stdcall` calling convention is used to call Win32 API functions.

This calling convention cannot be used with variadic functions because the called function cleans the stack.

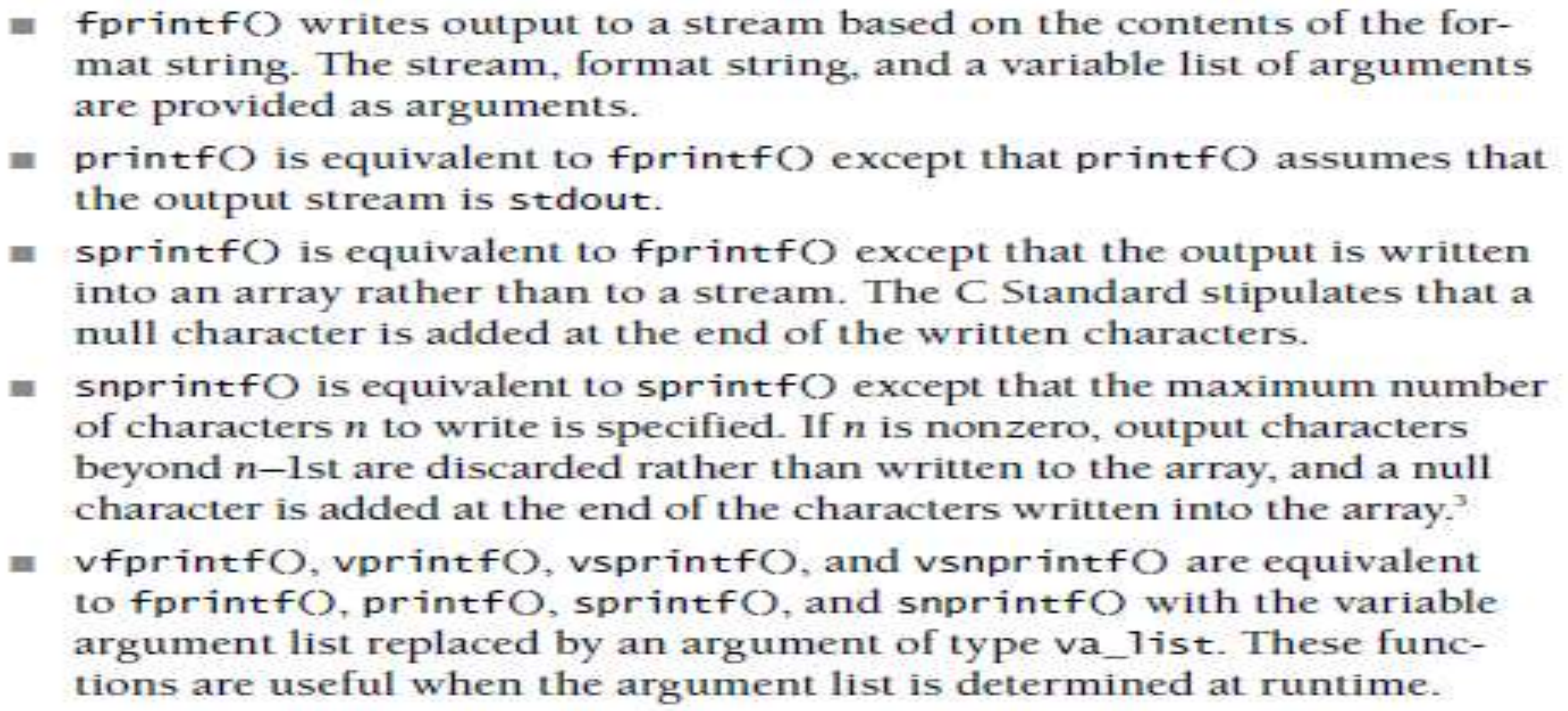
_fastcall



- The `__fastcall` calling convention specifies that arguments to functions are to be passed in registers when possible.
- The first two `doublewords` or smaller arguments are passed in `ecx` and `edx` registers.

UNIT-4—Formatted Output

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. The C Standard stipulates that a null character is added at the end of the written characters.
- snprintf() is equivalent to sprintf() except that the maximum number of characters *n* to write is specified. If *n* is nonzero, 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.[3]
- vfprintf(), vprintf(), vsprintf(), and vsnprintf() are equivalent to fprintf(), printf(), sprintf(), and snprintf() with the variable argument list replaced by an argument of type va_list. These functions are useful when the argument list is determined at runtime.

UNIT-4—Formatted Output

Formatted Output Functions

- ❑ The syslog() function accepts a priority argument, a format specification and any arguments required by the format and generates a log message to the system logger(syslogd).

- ❑ The syslog() function appeared in BSD4.2 and is supported by Linux and other modern POSIX implementations. It is not available in windows system.

UNIT-4—Formatted Output

Formatted Strings:-

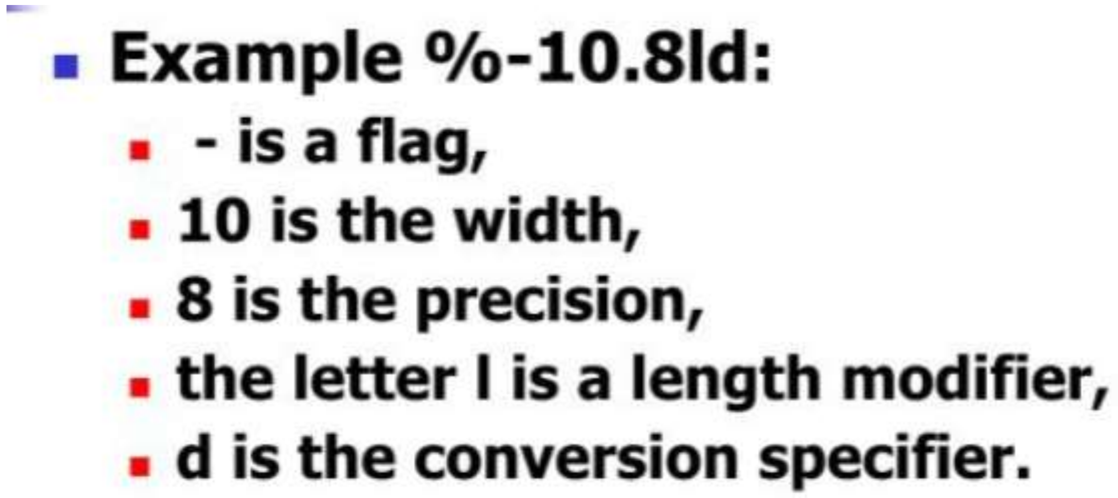
- **In C, format strings are used in functions like printf, sprintf, snprintf, fprintf, etc., to specify how data should be formatted when displayed or written to a string.**

- **A format string consists of plain text and conversion specifiers that dictate how specific data types should be formatted.**

UNIT-4—Formatted Output

Formatted Strings:-

[flags][width][.precision][length-modifier] conversion-specifier



- **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.**

UNIT-4—Formatted Output

Formatted Strings:-

Conversion Specifier: A Conversion specifier indicates the type of conversion to be applied. The conversion specifier character is the only required format field, and it appears after any optional format fields.

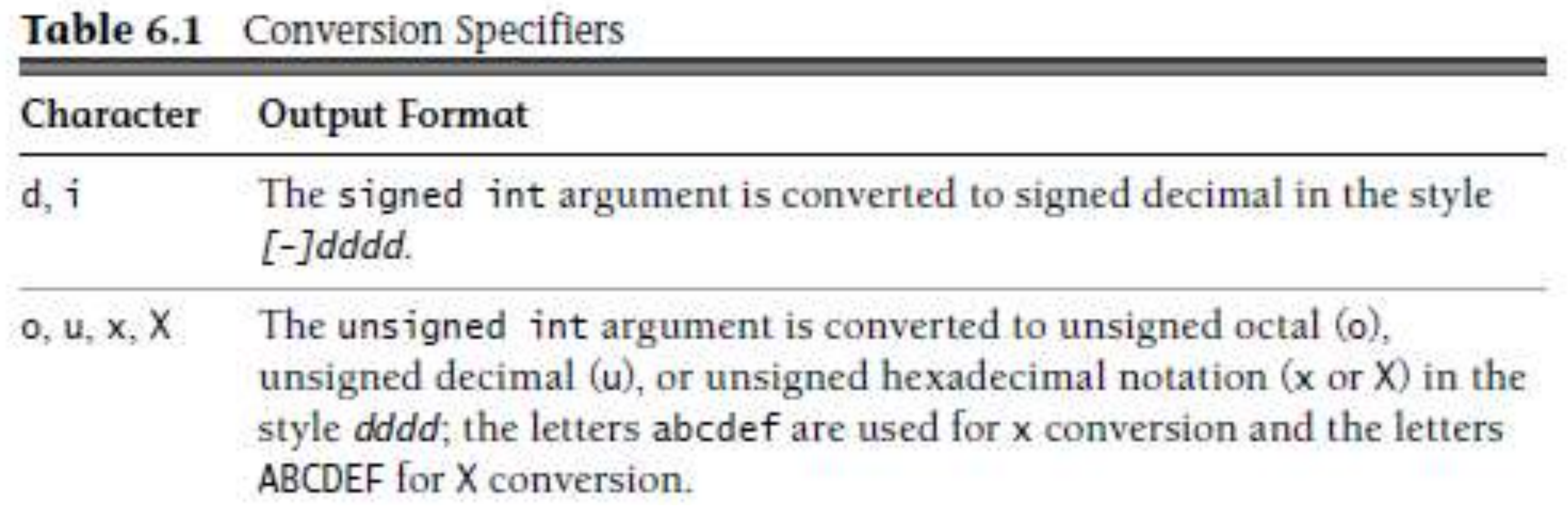


Table 6.1 Conversion Specifiers

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.

UNIT-4—Formatted Output

Formatted Strings:-

Conversion Specifier: A Conversion specifier indicates the type of conversion to be applied. The conversion specifier character is the only required format field, and it appears after any optional format fields.

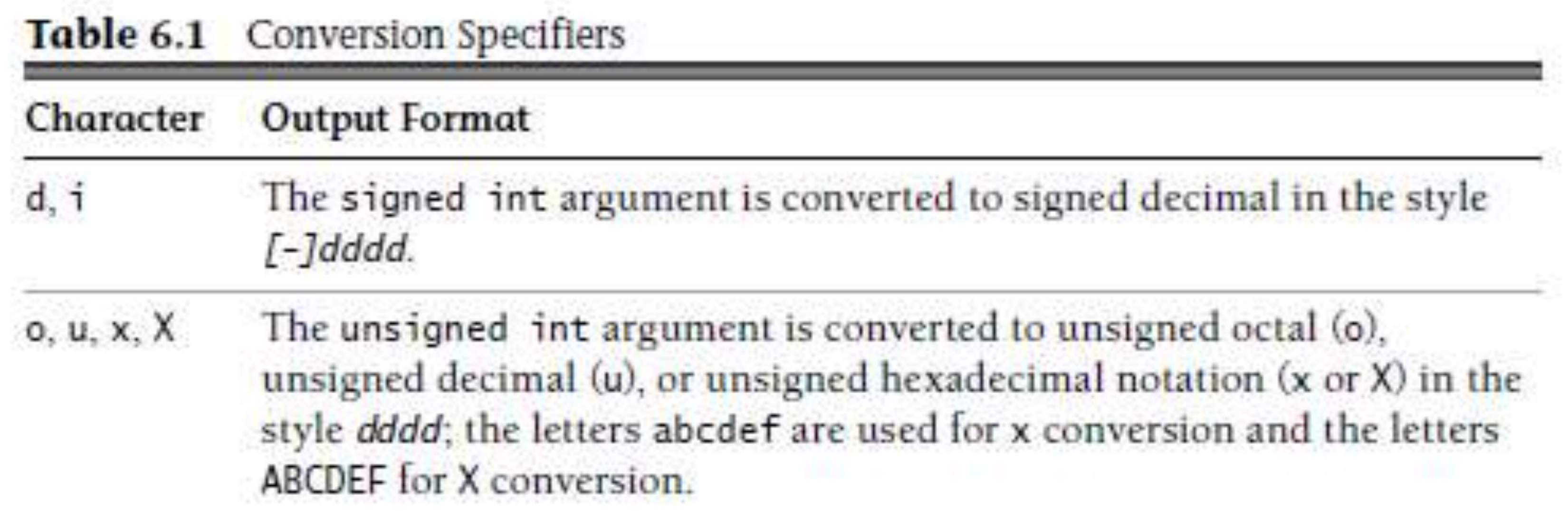


Table 6.1 Conversion Specifiers

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.

UNIT-4—Formatted Output

Formatted Strings:-

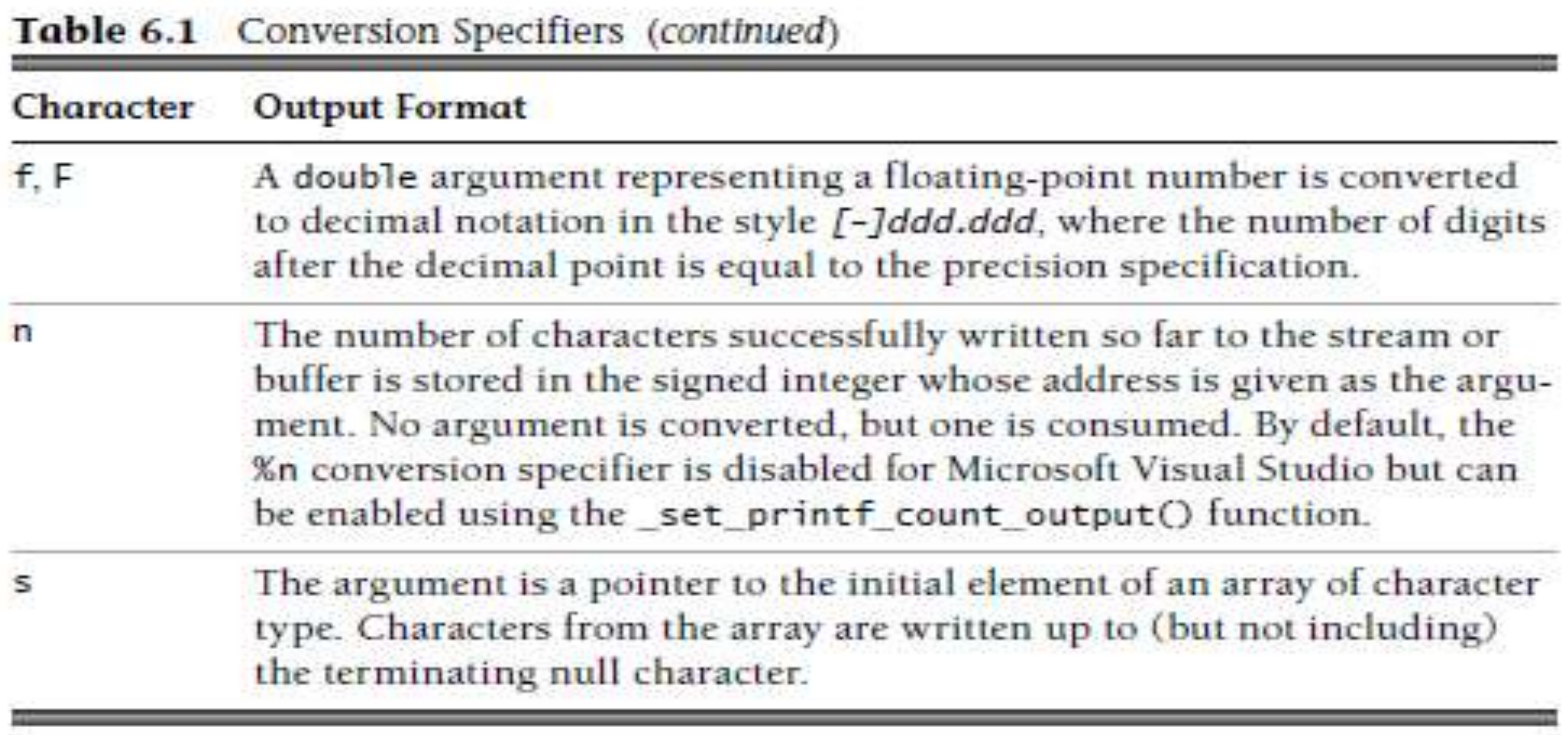


Table 6.1 Conversion Specifiers (*continued*)

Character	Output Format
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.

UNIT-4—Formatted Output

Formatted Strings:-

Flags. Flags justify output and print signs, blanks, decimal points, and octal and hexadecimal prefixes.

Width. Width is a nonnegative decimal integer that 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.

Precision. Precision is a nonnegative decimal integer that specifies the number of characters to be printed, the number of decimal places, or the number of significant digits.

Length Modifier: Length modifier specifies the size of the argument.

UNIT-4—Formatted Output

Exploiting Formatted Output Functions:-

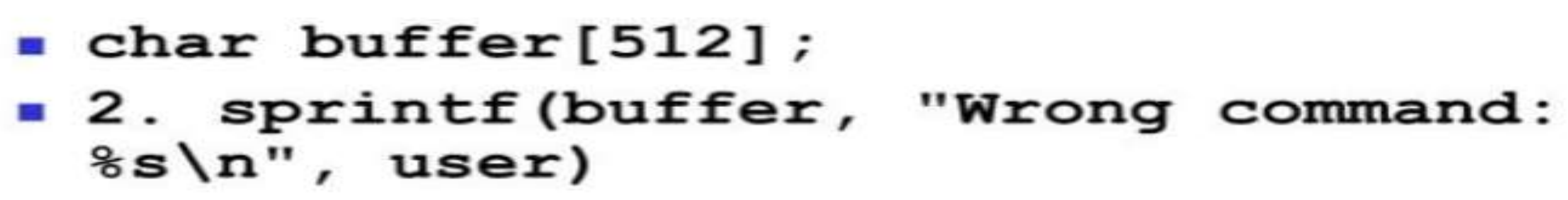
- Formatted Output strings are vulnerable to buffer overflow when formatted output routine writes beyond the boundaries of a data structures.

- Formatted String Vulnerabilities can occur when format string is supplied by the user or other untrusted source.

UNIT-4—Formatted Output

Buffer Overflow:

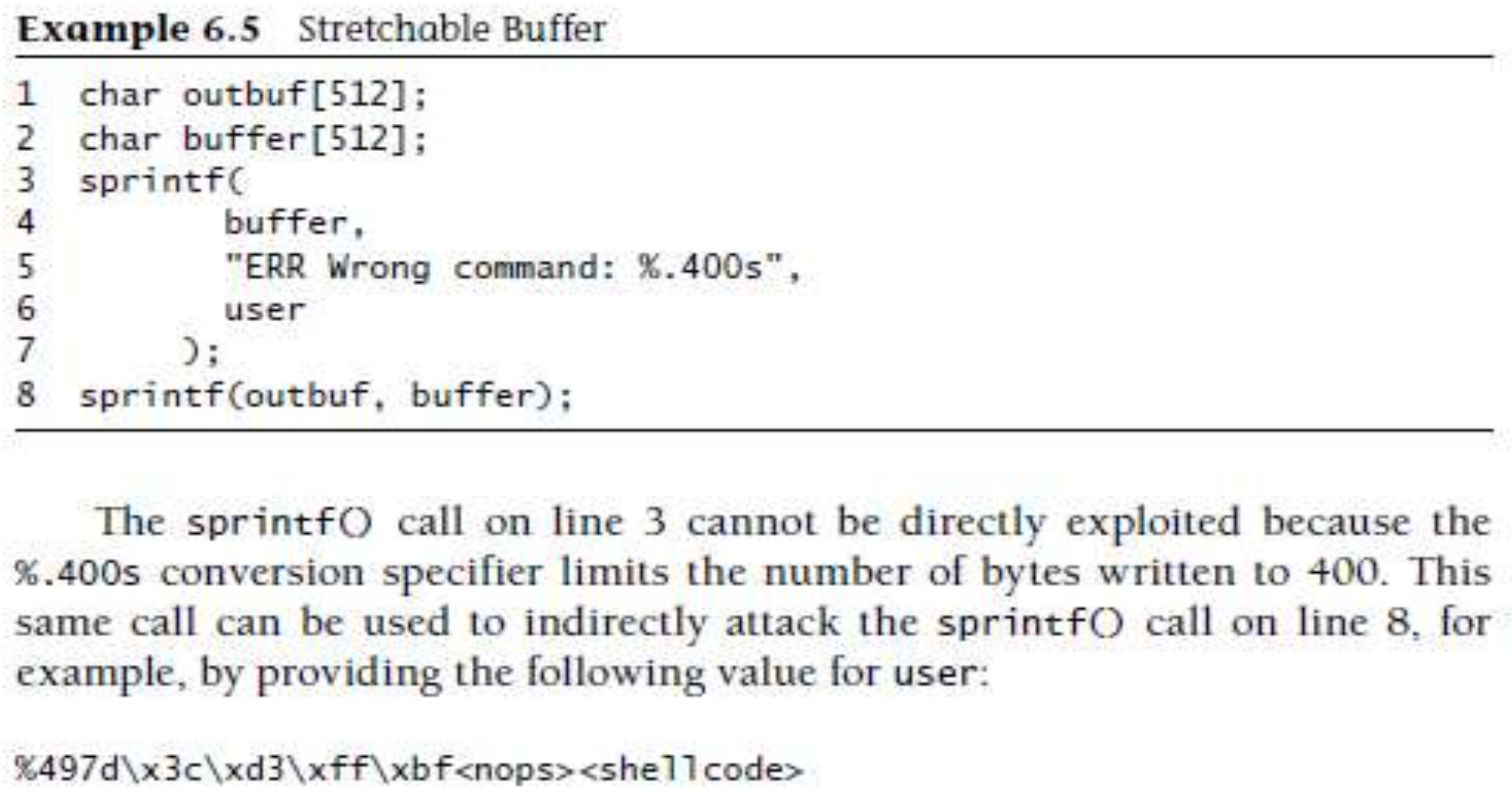
Formatted output functions that write to a character array(for example sprint()) assume arbitrarily long buffers, which males them susceptible to buffer overflows.



```
▪ char buffer[512];
▪ 2. sprintf(buffer, "Wrong command:
  %s\n", user)
```

UNIT-4—Formatted Output

Buffer Overflow:



Example 6.5 Stretchable Buffer

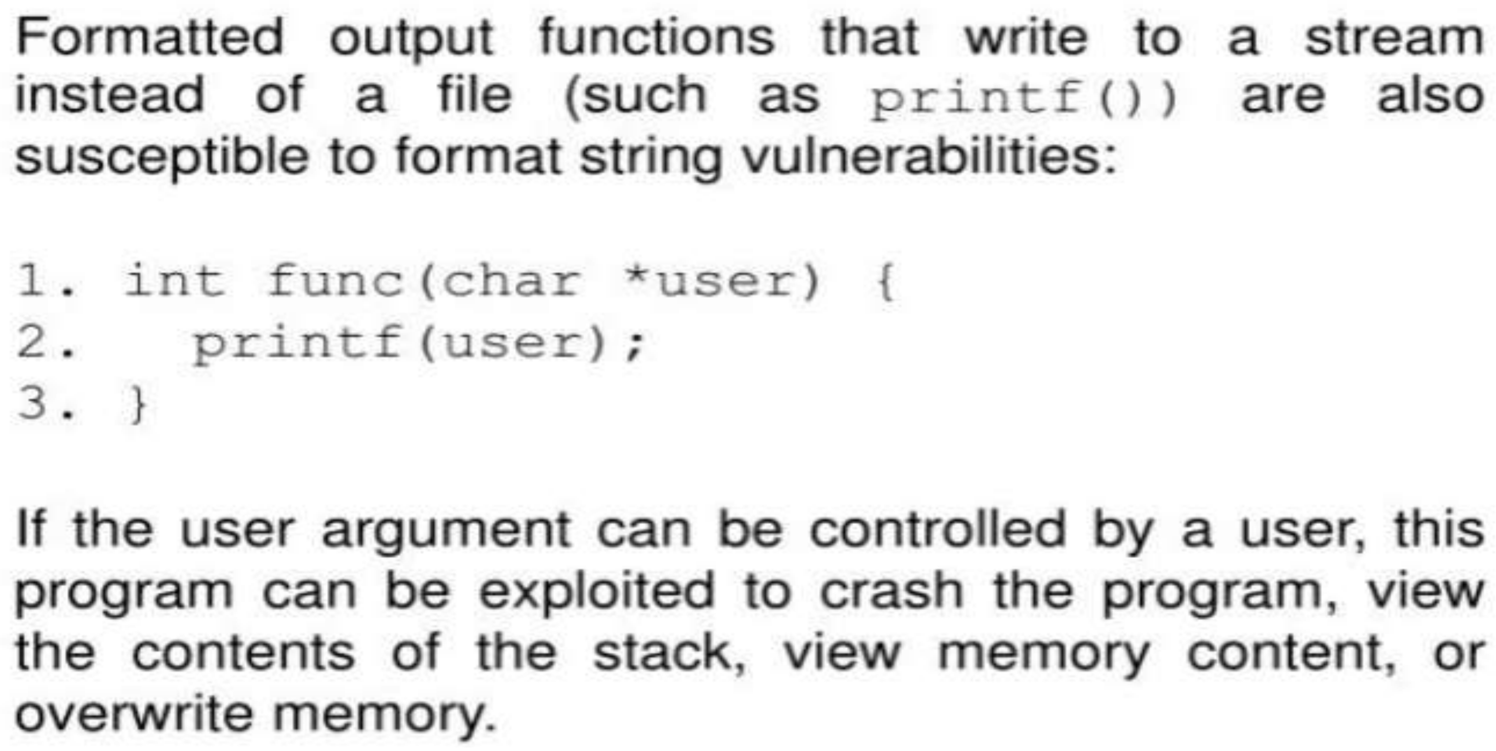
```
1  char outbuf[512];
2  char buffer[512];
3  sprintf(
4         buffer,
5         "ERR Wrong command: %.400s",
6         user
7       );
8  sprintf(outbuf, buffer);
```

The `sprintf()` call on line 3 cannot be directly exploited because the `%.400s` conversion specifier limits the number of bytes written to 400. This same call can be used to indirectly attack the `sprintf()` call on line 8, for example, by providing the following value for user:

```
%497d\x3c\xd3\xff\xbf<nops><shellcode>
```

UNIT-4—Formatted Output

Output Streams:



Formatted output functions that write to a stream instead of a file (such as `printf()`) are also susceptible to format string vulnerabilities:

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

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.

UNIT-4—Formatted Output

Output Streams Exploits:

1. **Accessing Memory (Format String Attack):** If user contains a format specifier like %x or %s, printf will attempt to read data from the stack. This can allow an attacker to:

- Read data from memory (e.g., sensitive variables, return addresses).
- Potentially leak sensitive information such as passwords or cryptographic keys.

Example: func("%x %x %x"); // Could print stack values

2. **Writing to Memory (Dangerous):** If the input contains %n, it can cause printf to write the number of characters printed so far into a variable, which could lead to arbitrary memory writes.

Example:
func("%n"); // Will write number of printed characters to memory

UNIT-4—Formatted Output

Crash a Program:

Format string vulnerabilities are often discovered when a program crashes.

When an attempt to read an unmapped address in windows results in a general protection fault followed by abnormal program termination. An Invalid pointer access or unmapped address read can usually be triggered by calling a formatted output function.

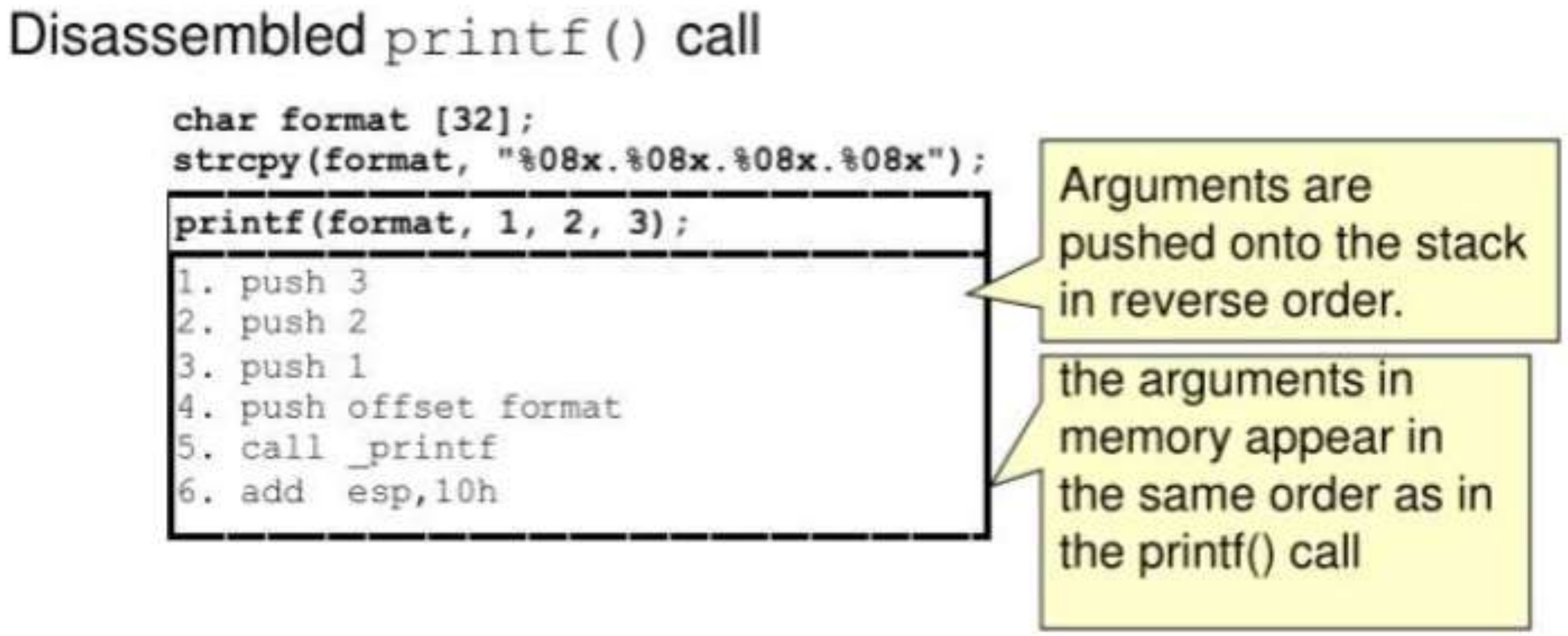
printf(“%s%s%s%s%s”);

The string "%s %s %s%s %s" passed to printf contains multiple format specifiers (%s), which expect corresponding arguments. If the number of arguments provided does not match the number of format specifiers, the behavior of printf is undefined, and this can lead to serious vulnerabilities.

UNIT-4—Formatted Output

Viewing Stack Content:

The attackers can also exploit formatted output functions to examine the contents of memory.



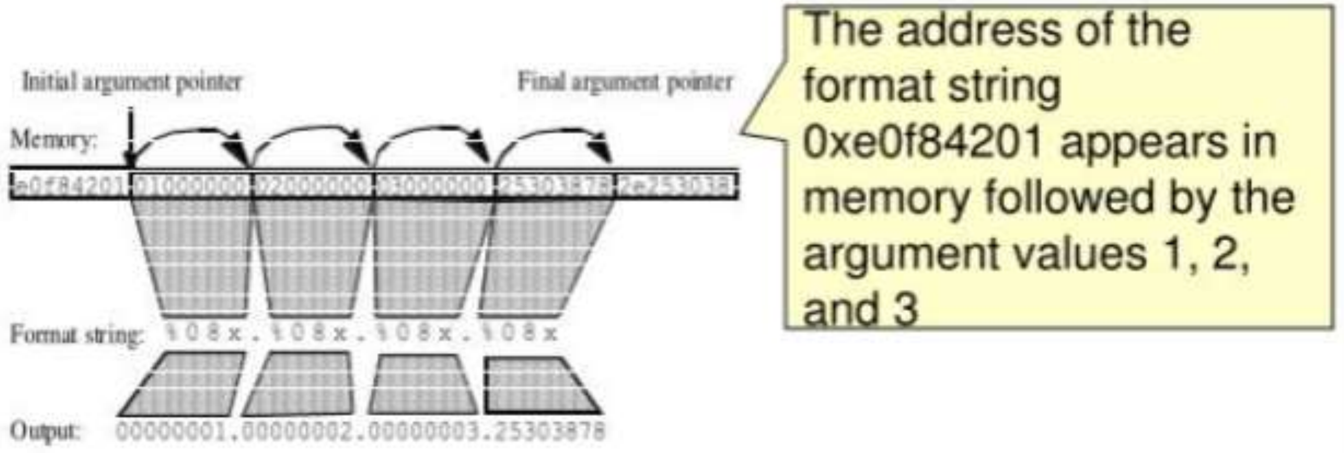
Disassembled `printf()` call

```
char format [32];
strcpy(format, "%08x.%08x.%08x.%08x");
printf(format, 1, 2, 3);
1. push 3
2. push 2
3. push 1
4. push offset format
5. call _printf
6. add  esp,10h
```

Arguments are pushed onto the stack in reverse order.
the arguments in memory appear in the same order as in the printf() call

UNIT-4—Formatted Output

Viewing Stack Content:


Initial argument pointer
Final argument pointer
Memory:
e0f84201
01000000
02000000
03000000
25303878
2e253038
Format string: %08x.%08x.%08x.%08x
Output: 00000001.00000002.00000003.25303878
The address of the format string 0xe0f84201 appears in memory followed by the argument values 1, 2, and 3

Viewing the contents of the stack

UNIT-4—Formatted Output

Viewing Stack Content:

If you pass user input or improperly crafted format strings to printf, you might inadvertently reveal the contents of the stack. This is where a format string vulnerability can arise.

```
#include <stdio.h>
void func() {
    int a = 10;
    int b = 20;
    printf("%x %x %x\n", a, b, 42); // Print stack contents in hexadecimal
}

int main() {
    func();
    return 0;
}
```

UNIT-4—Formatted Output

Viewing Stack Content:

```
#include <stdio.h>

void func(char *user_input) {
    printf(user_input);  // Vulnerable to format string attack
}

int main() {
    char *input = "%x %x %x %x %x";  // Format string
    func(input);  // Calling func with malicious format string
    return 0;
}
```

UNIT-4—Formatted Output

Viewing Memory Content:

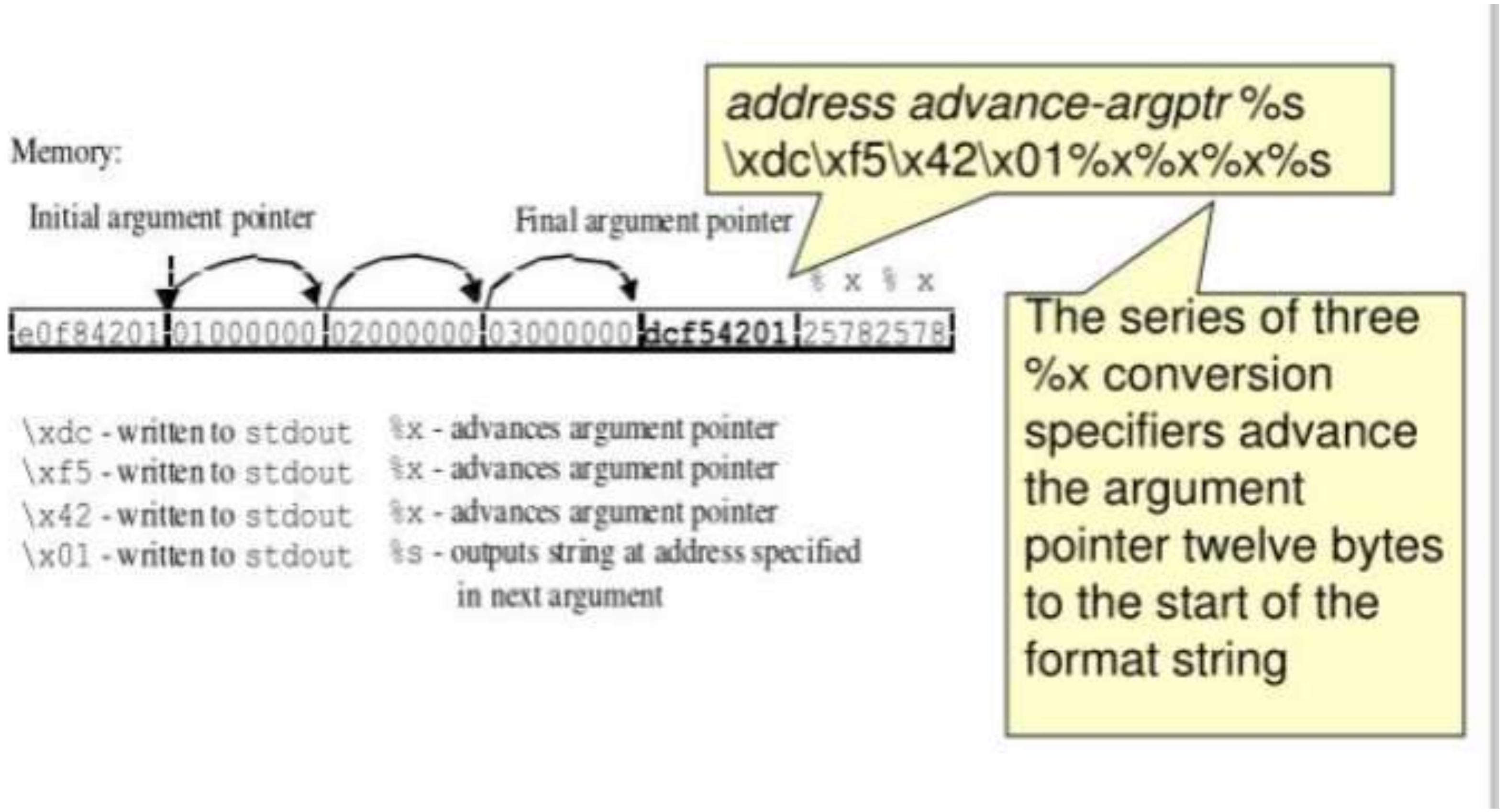
The attacker examine a memory at an arbitrary address by using a format specification that displays memory at a specified address.

For example, the %s conversion specified displays memory at an address specified by the argument pointer as an ASCII string until null byte is encountered.

If an attacker can manipulate the argument pointer to reference a particular address, the %s conversion will output memory at that location.

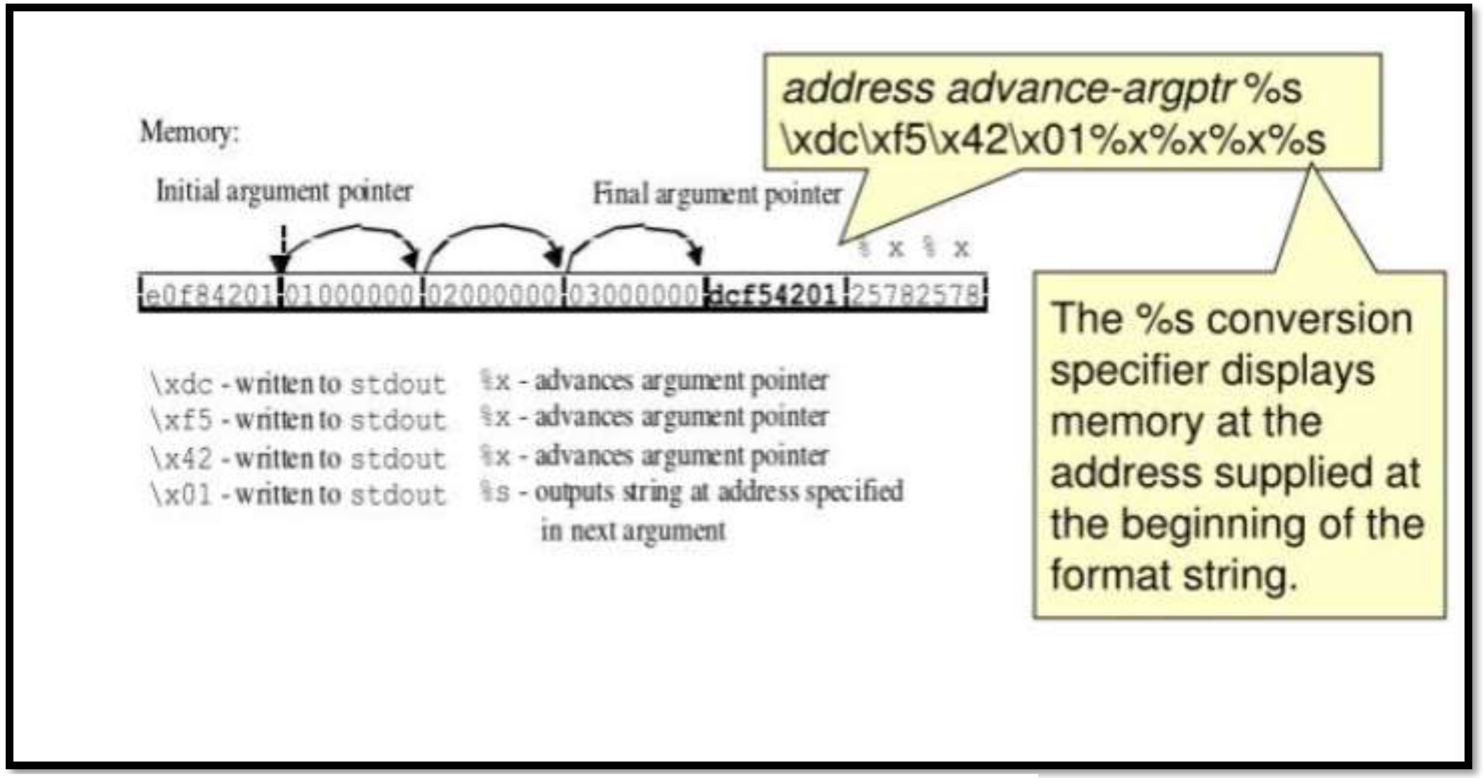
UNIT-4—Formatted Output

Viewing Memory at specific location:


Memory:
Initial argument pointer
Final argument pointer
address advance-argptr %s
\xdc\xf5\x42\x01%x%x%x%s
% x % x
e0f84201
01000000
02000000
03000000
dcf54201
25782578
\xdc - written to stdout
\xf5 - written to stdout
\x42 - written to stdout
\x01 - written to stdout
%x - advances argument pointer
%x - advances argument pointer
%x - advances argument pointer
%s - outputs string at address specified in next argument
The series of three %x conversion specifiers advance the argument pointer twelve bytes to the start of the format string

UNIT-4—Formatted Output

Viewing Memory at specific location:



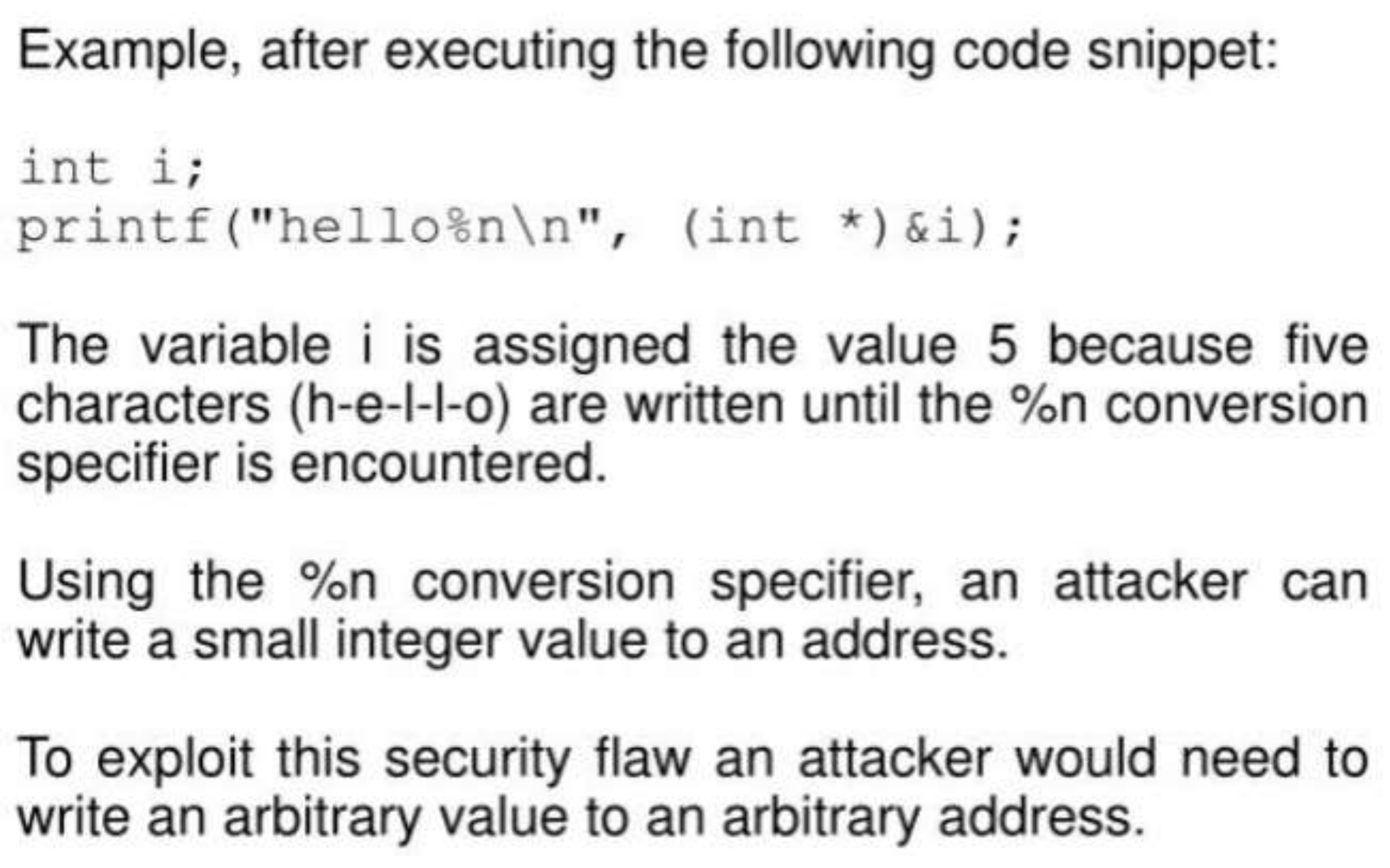
address advance-argptr %s
\xdc\xf5\x42\x01%x%x%x%s
Memory:
Initial argument pointer
Final argument pointer
% x % x
e0f84201
01000000
02000000
03000000
dcf54201
25782578
\xdc - written to stdout
\xf5 - written to stdout
\x42 - written to stdout
\x01 - written to stdout
%x - advances argument pointer
%x - advances argument pointer
%x - advances argument pointer
%s - outputs string at address specified in next argument
The %s conversion specifier displays memory at the address supplied at the beginning of the format string.

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()`.

UNIT-4—Formatted Output

Overwriting Memory:



Example, after executing the following code snippet:

```
int i;
printf("hello%n\n", (int *)&i);
```

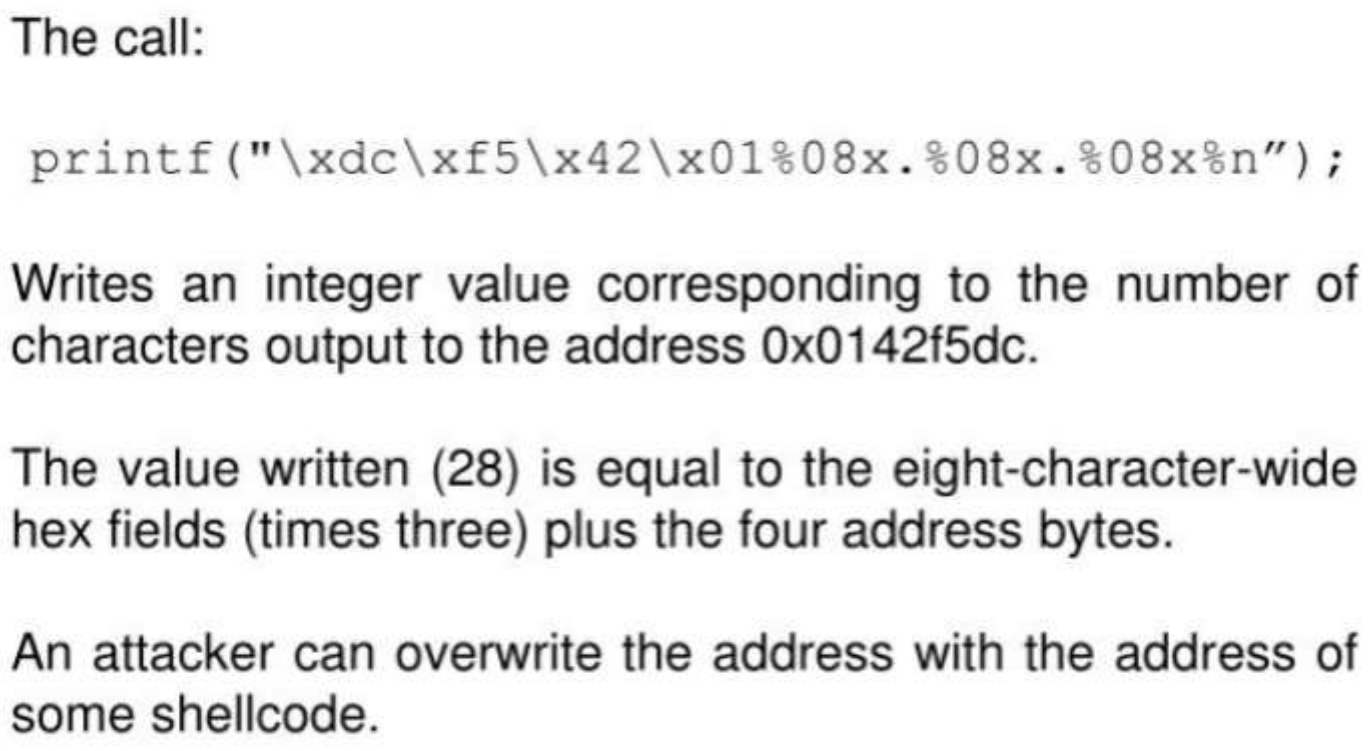
The variable i is assigned the value 5 because five characters (h-e-l-l-o) are written until the %n conversion specifier is encountered.

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.

UNIT-4—Formatted Output

Overwriting Memory:



The call:

```
printf("\xdc\xf5\x42\x01%08x.%08x.%08x%n”);
```

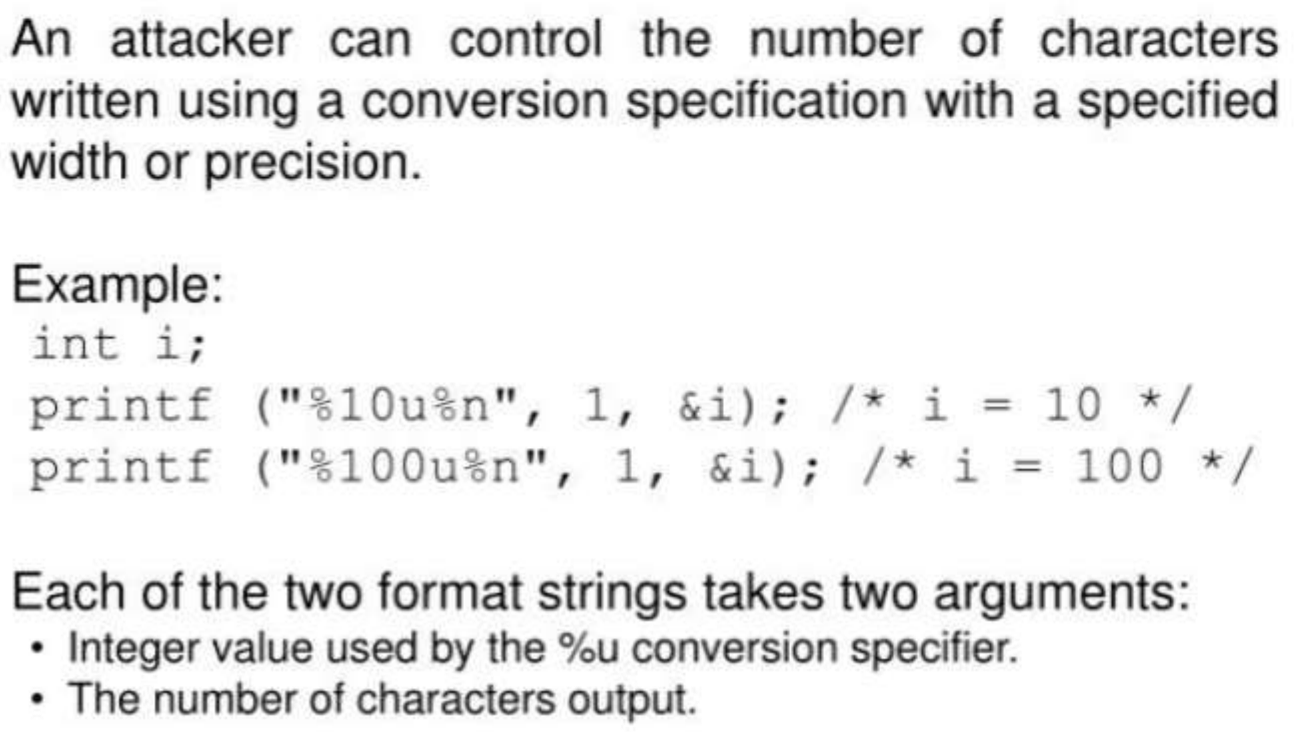
Writes an integer value corresponding to the number of characters output to the address 0x0142f5dc.

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.

UNIT-4—Formatted Output

Overwriting Memory:



An attacker can control the number of characters written using a conversion specification with a specified width or precision.

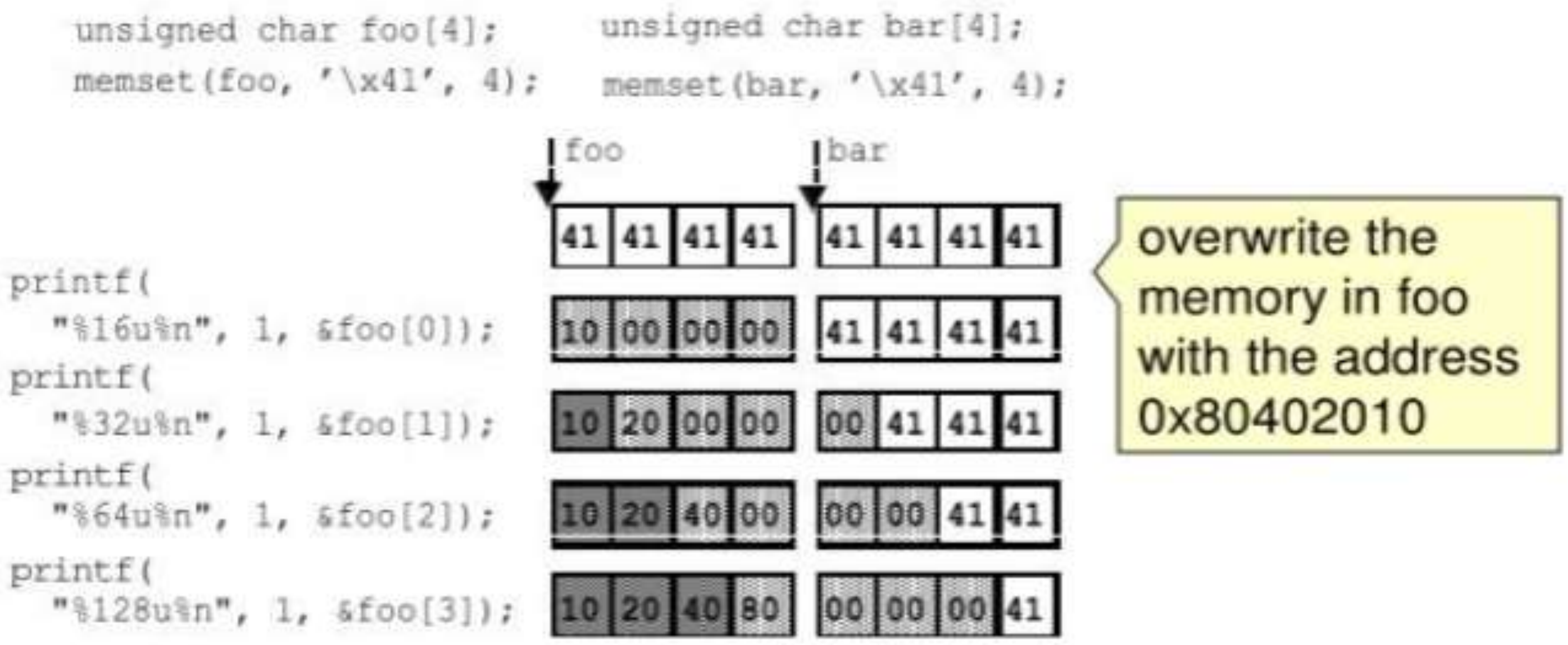
Example:

```
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: Writing an address in four stages

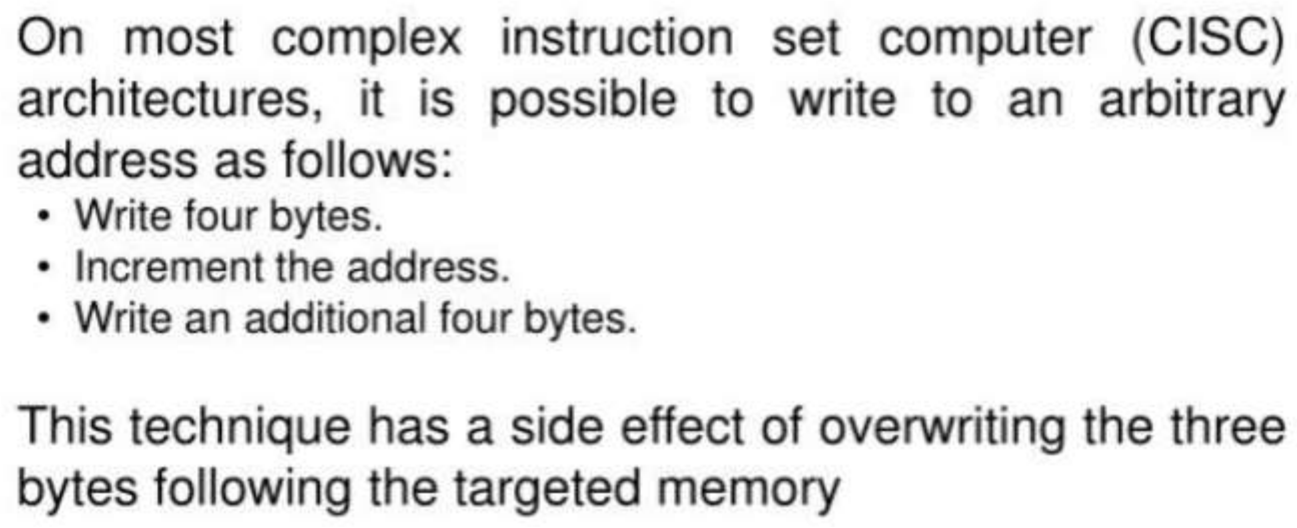


```
unsigned char foo[4];          unsigned char bar[4];
memset(foo, '\x41', 4);        memset(bar, '\x41', 4);

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

overwrite the memory in foo with the address 0x80402010

UNIT-4—Formatted Output

Overwriting Memory:



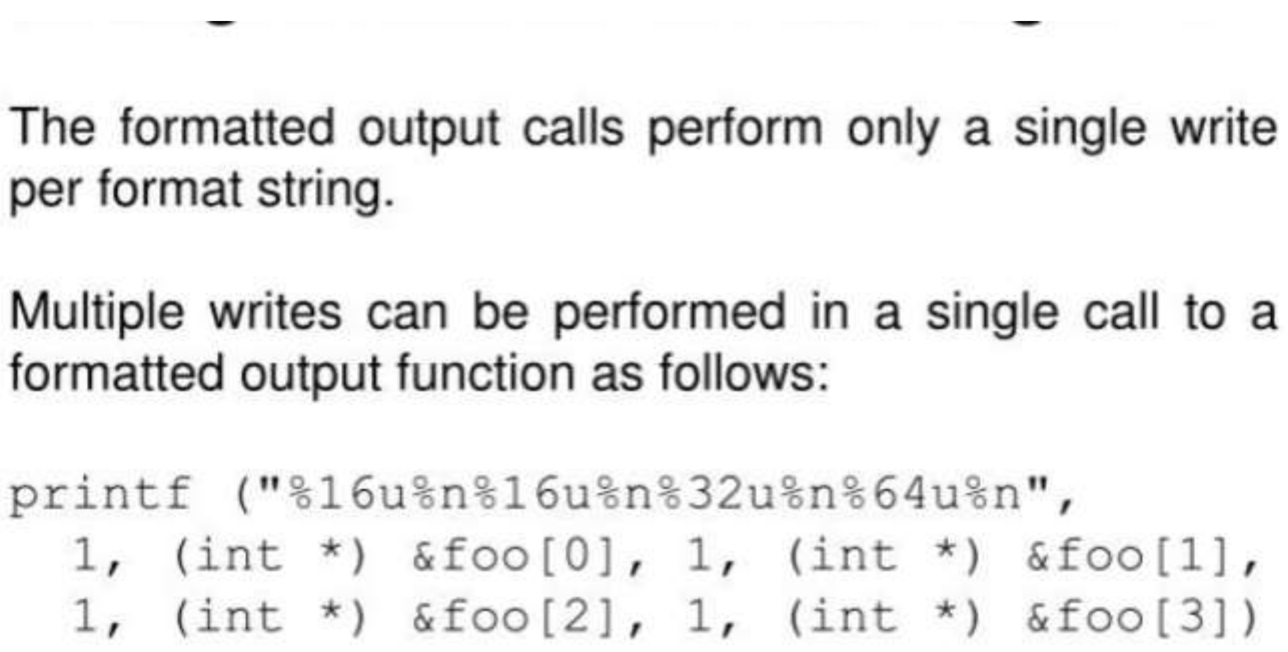
On most complex instruction set computer (CISC) architectures, it is possible to write to an arbitrary address as follows:

- 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

UNIT-4—Formatted Output

Overwriting Memory:

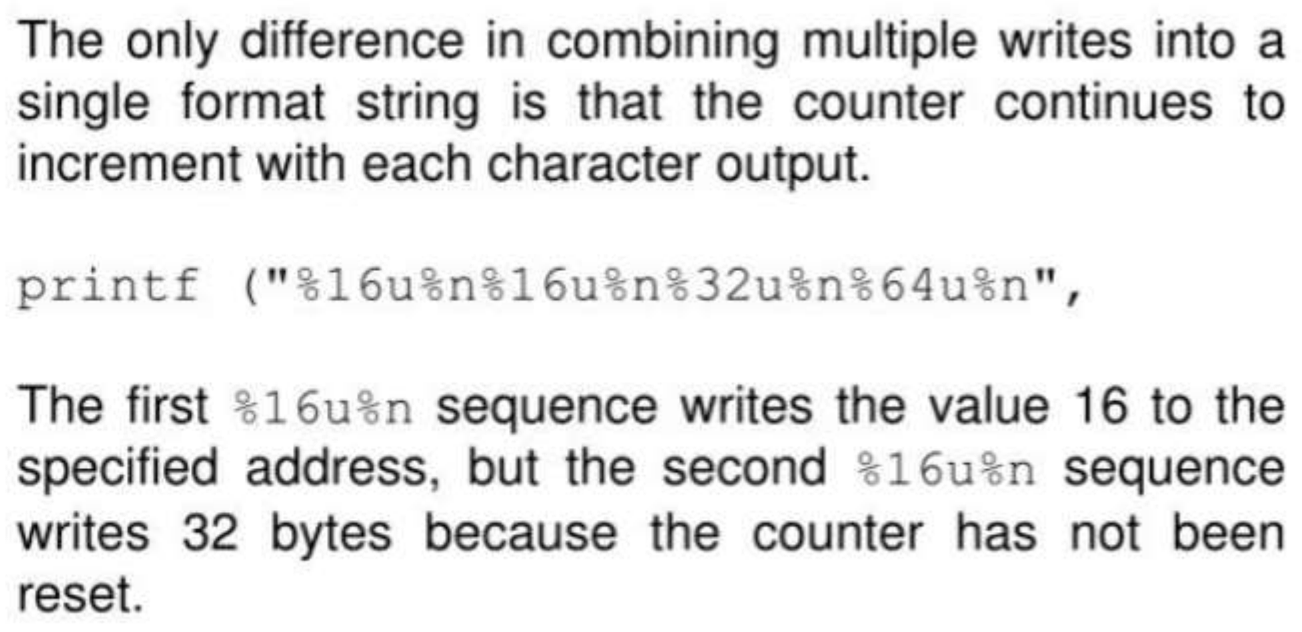


The formatted output calls perform only a single write per format string.

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])
```

Overwriting Memory:



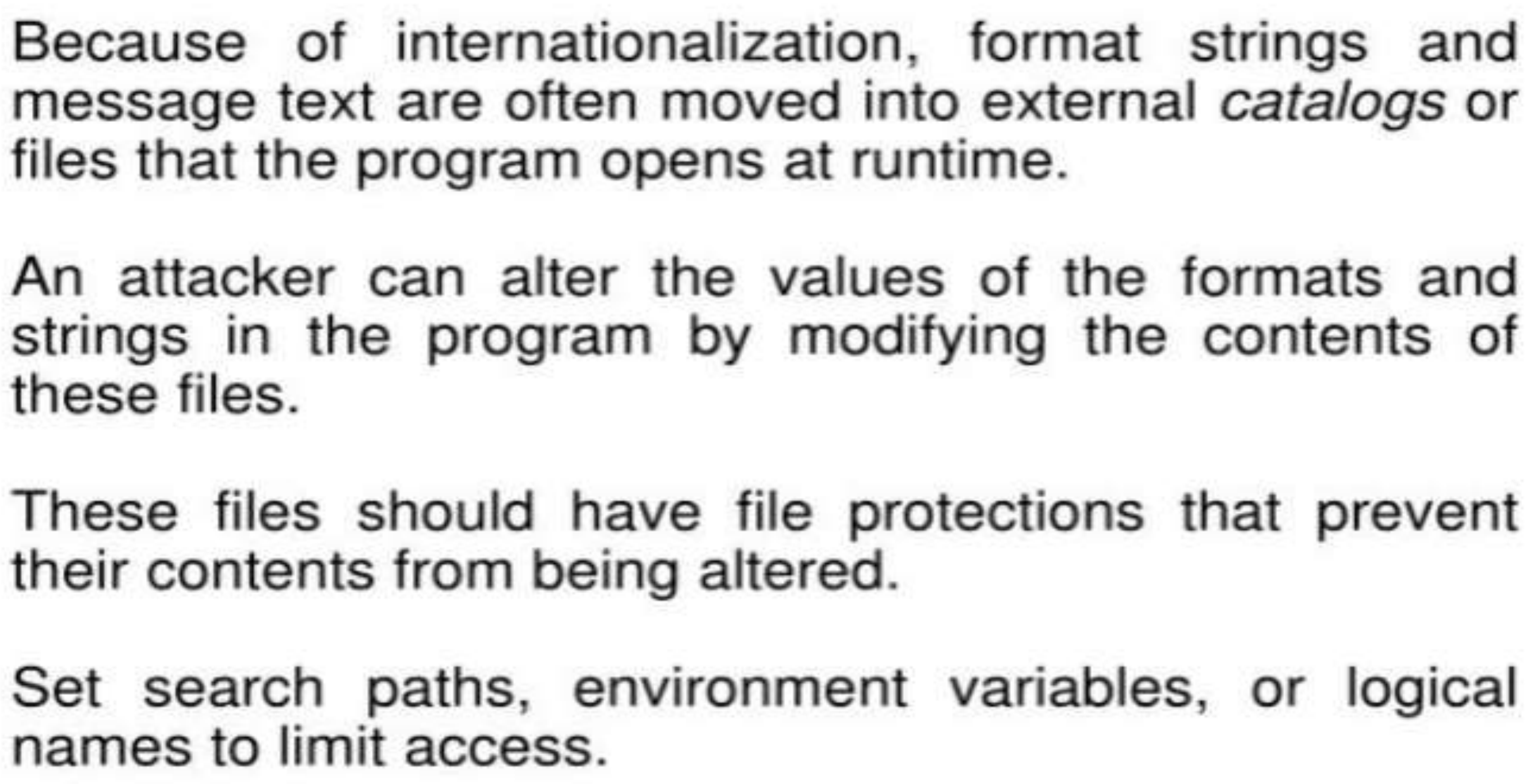
The only difference in combining multiple writes into a single format string is that the counter continues to increment with each character output.

```
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.

UNIT-4—Formatted Output

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.

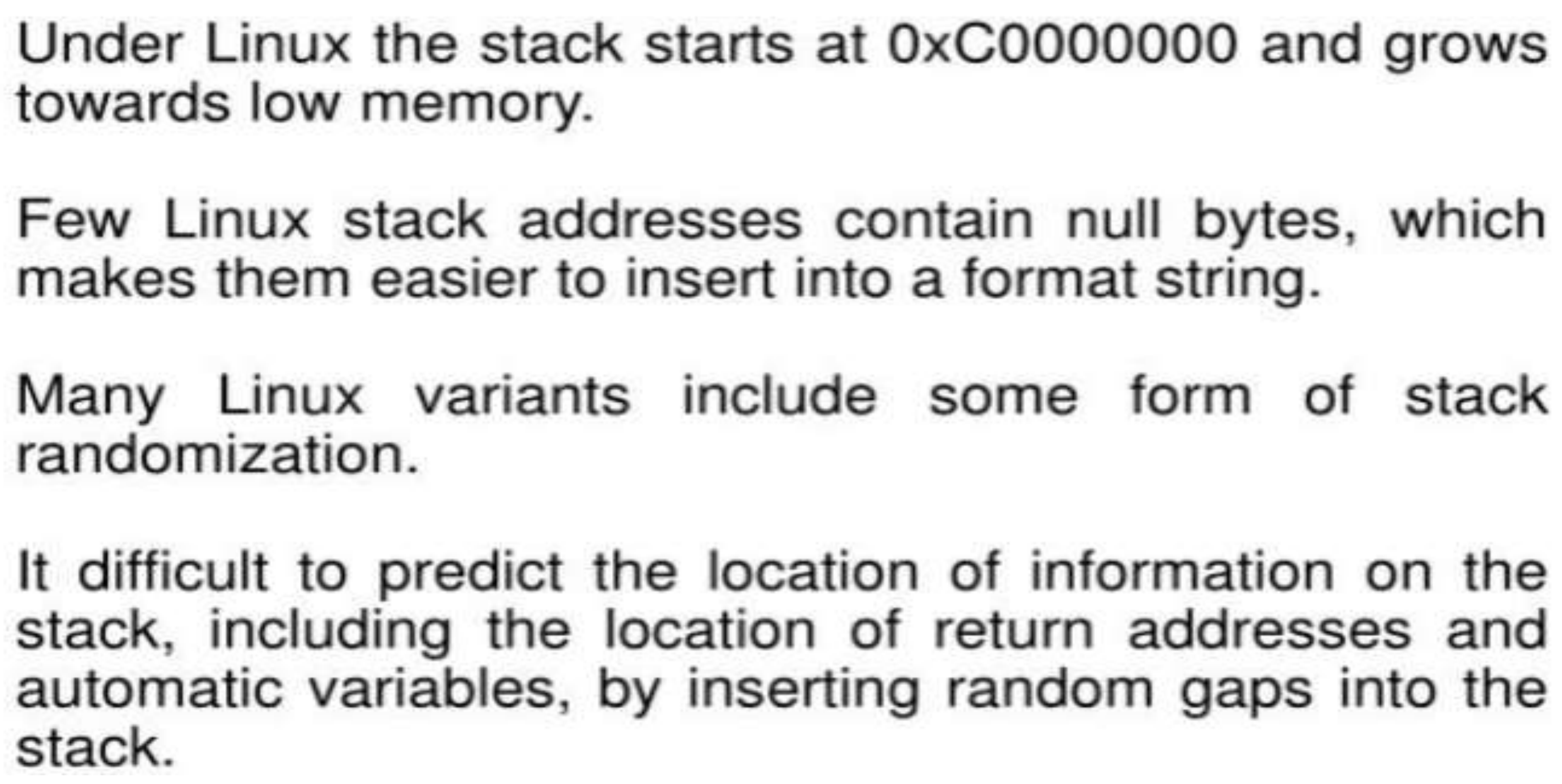
Set search paths, environment variables, or logical names to limit access.

Stack Randomization:

- **Stack Randomization** (often called **Stack Canary** or **Stack Protector**) aims to make it more difficult for attackers to predict the location of key data structures on the stack. By randomizing the location of stack variables (including return addresses and function pointers), it prevents attackers from being able to easily exploit buffer overflows.

UNIT-4—Formatted Output

Stack Randomization:



Under Linux the stack starts at 0xC0000000 and grows towards low memory.

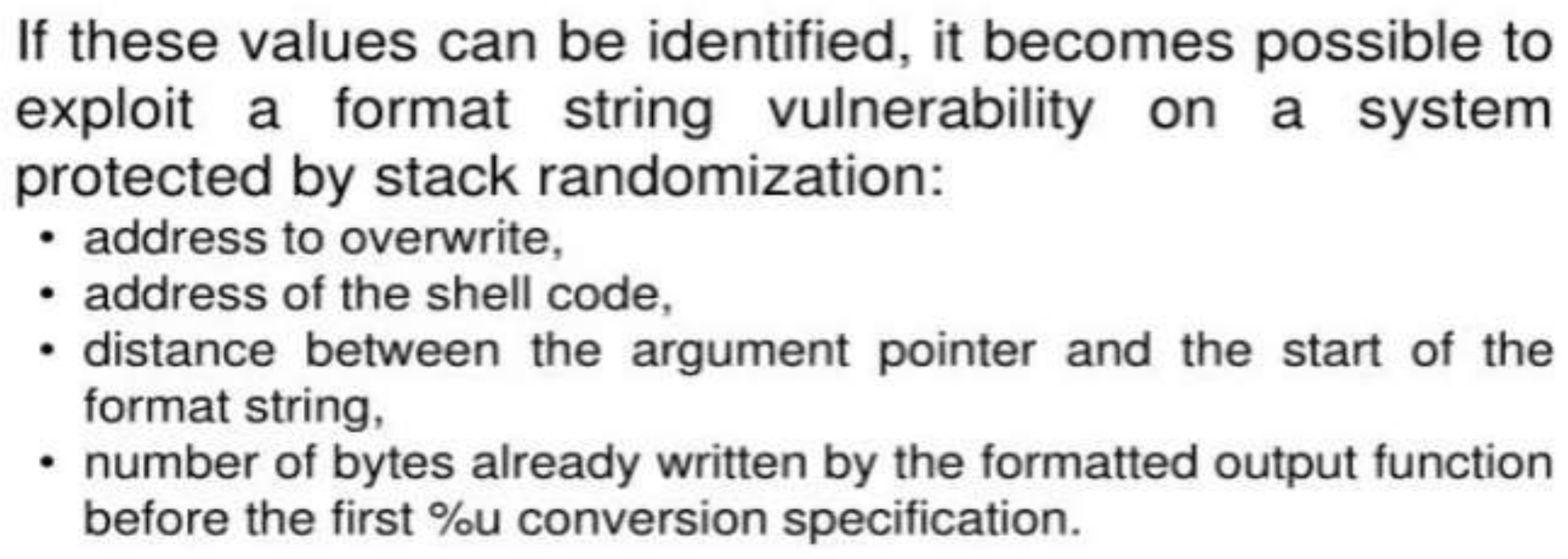
Few Linux stack addresses contain null bytes, which makes them easier to insert into a format string.

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.

UNIT-4—Formatted Output

Thwarting Stack Randomization:

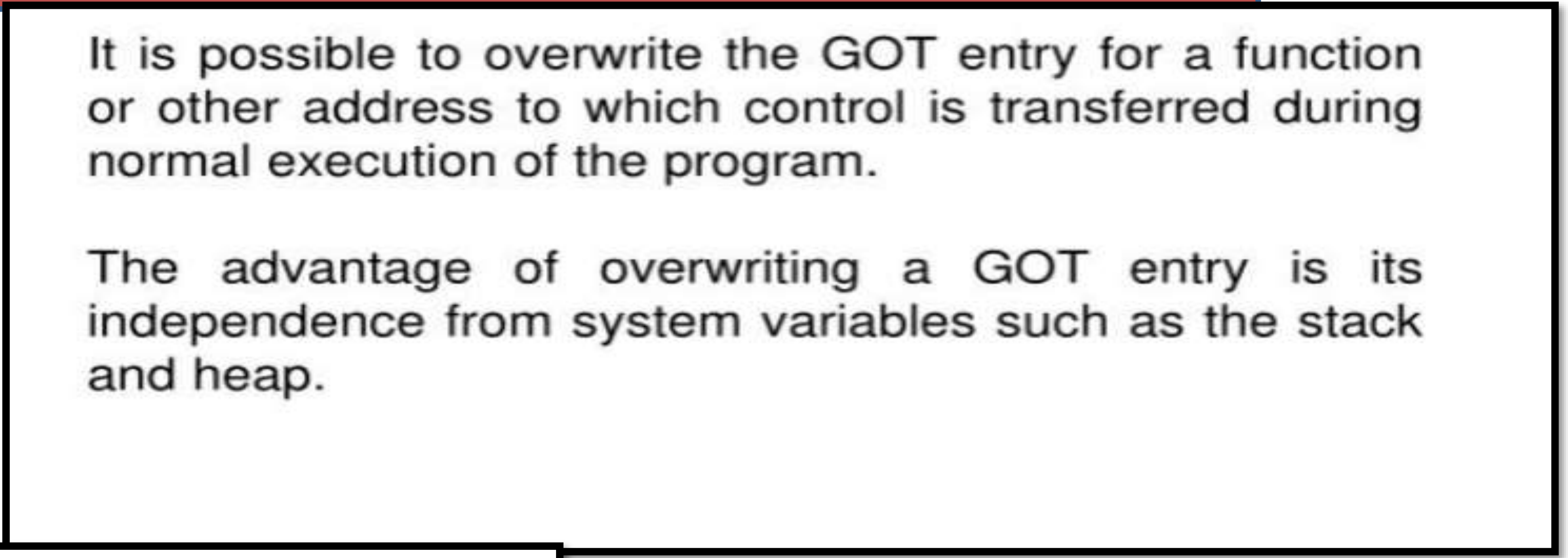


If these values can be identified, it becomes possible to exploit a format string vulnerability on a system protected by stack randomization:

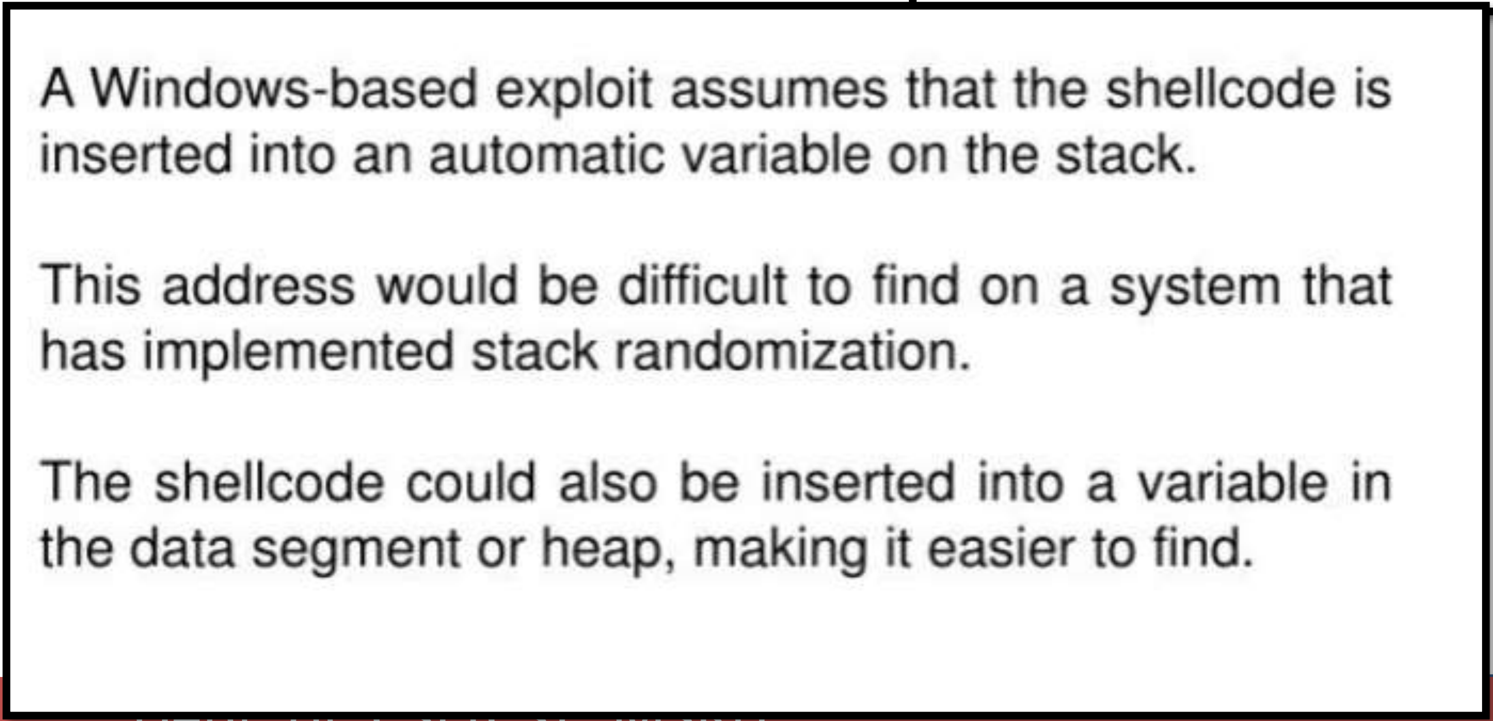
- address to overwrite,
- address of the shell code,
- distance between the argument pointer and the start of the format string,
- number of bytes already written by the formatted output function before the first %u conversion specification.

UNIT-4—Formatted Output

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.


A Windows-based exploit assumes that the shellcode is inserted into an automatic variable on the stack.

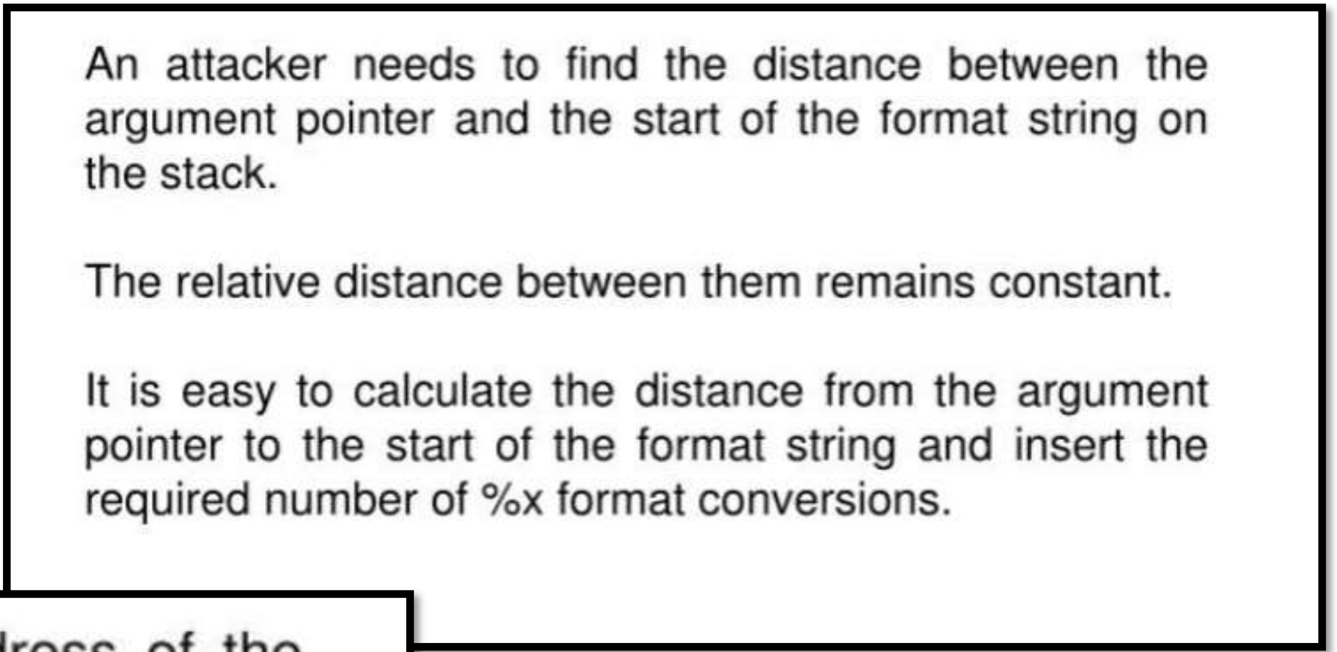
This address would be difficult to find on a system that has implemented stack randomization.

The shellcode could also be inserted into a variable in the data segment or heap, making it easier to find.

Address to Shellcode

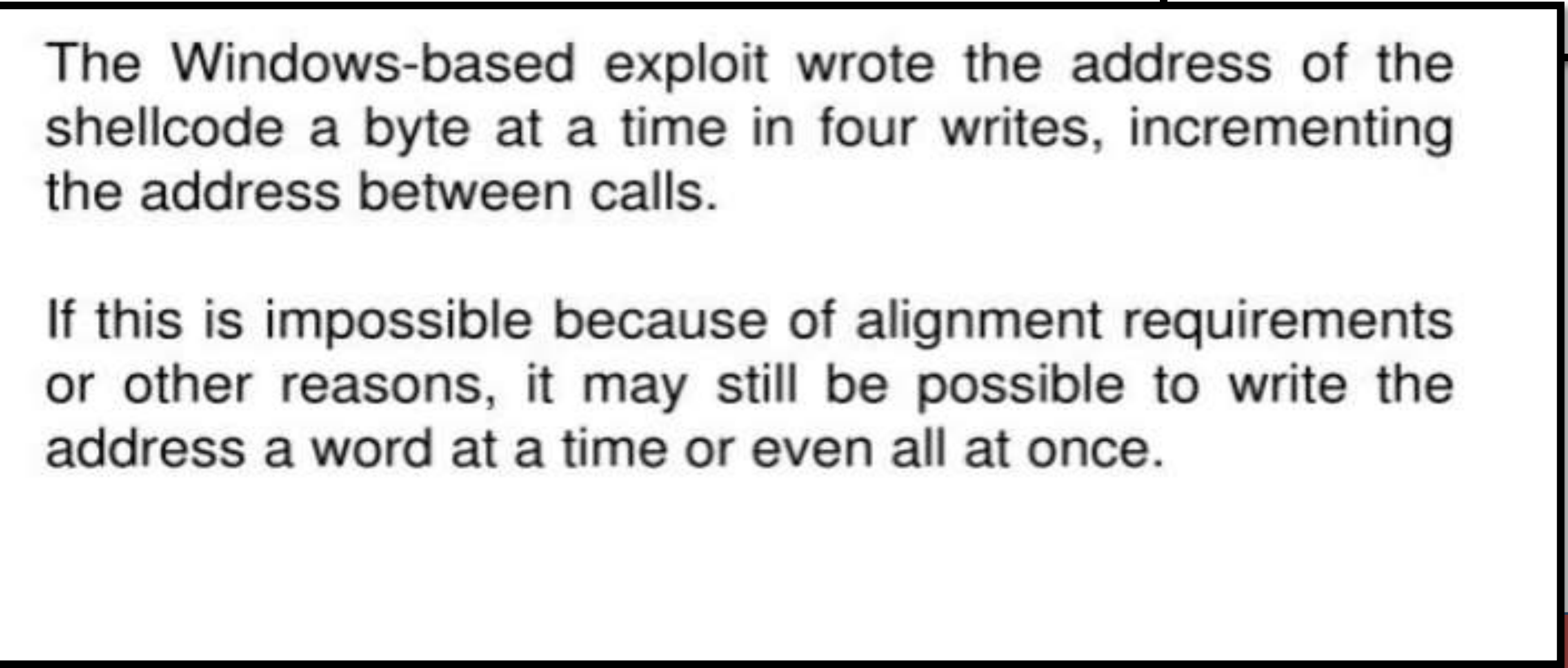
UNIT-4—Formatted Output

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.

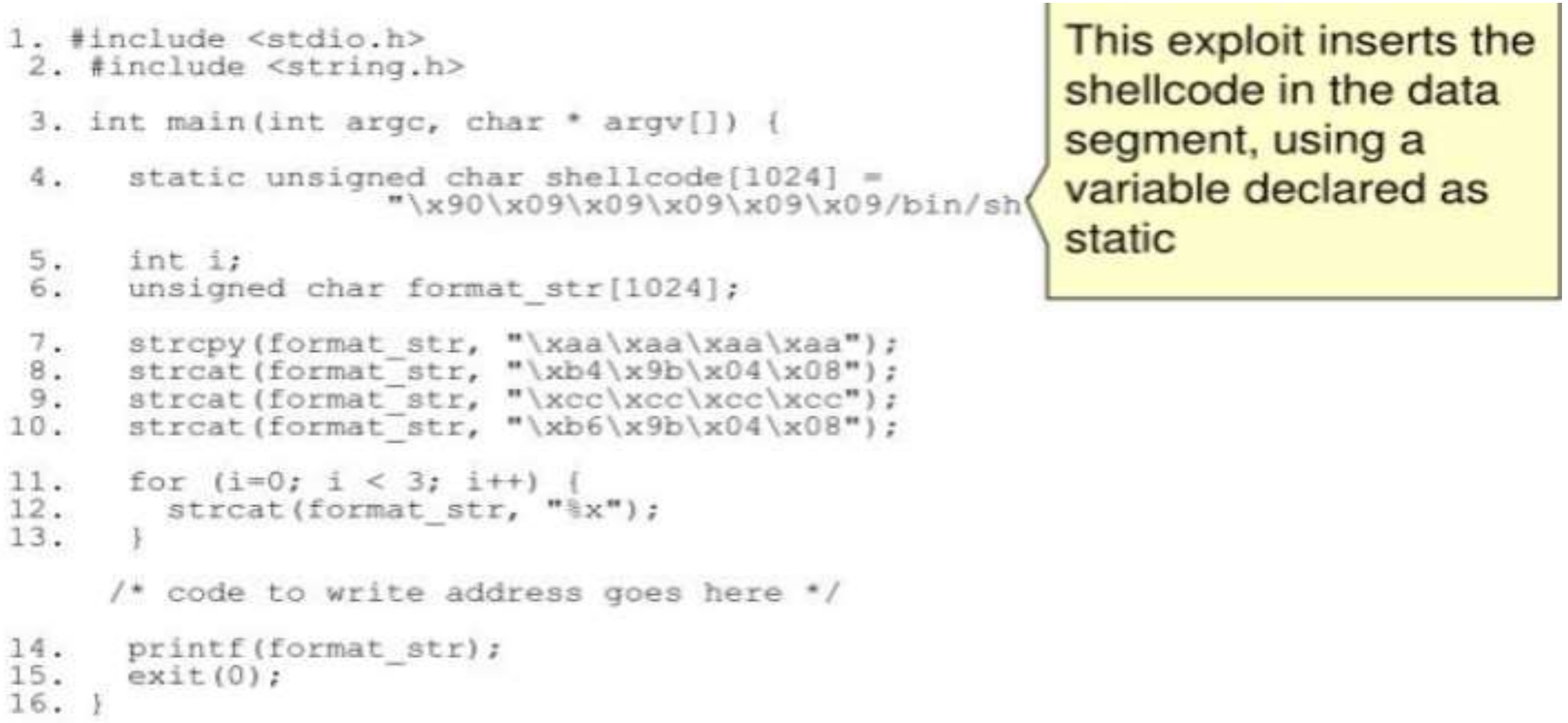
It is easy to calculate the distance from the argument pointer to the start of the format string and insert the required number of %x format conversions.


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.

Writing address in two words

Linux Exploit Variant-1:

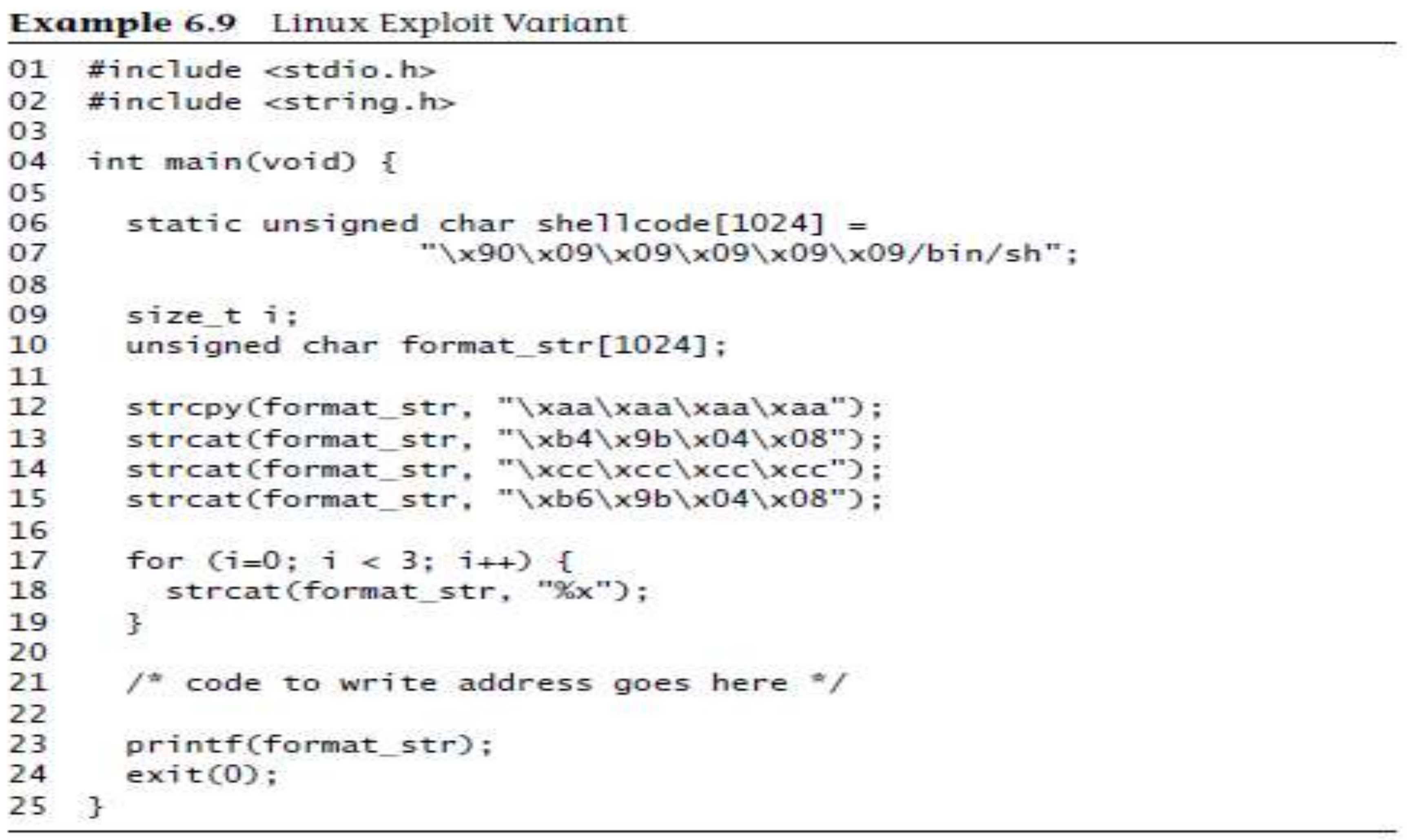


```
#include <stdio.h>
#include <string.h>

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

  static unsigned char shellcode[1024] =
                "\x90\x09\x09\x09\x09\x09/bin/sh

  int i;
  unsigned char format_str[1024];

  strcpy(format_str, "\xaa\xaa\xaa\xaa");
  strcat(format_str, "\xb4\x9b\x04\x08");
  strcat(format_str, "\xcc\xcc\xcc\xcc");
  strcat(format_str, "\xb6\x9b\x04\x08");

  for (i=0; i < 3; i++) {
    strcat(format_str, "%x");
  }

 /* code to write address goes here */

  printf(format_str);
  exit(0);
}
```

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

UNIT-4—Formatted Output

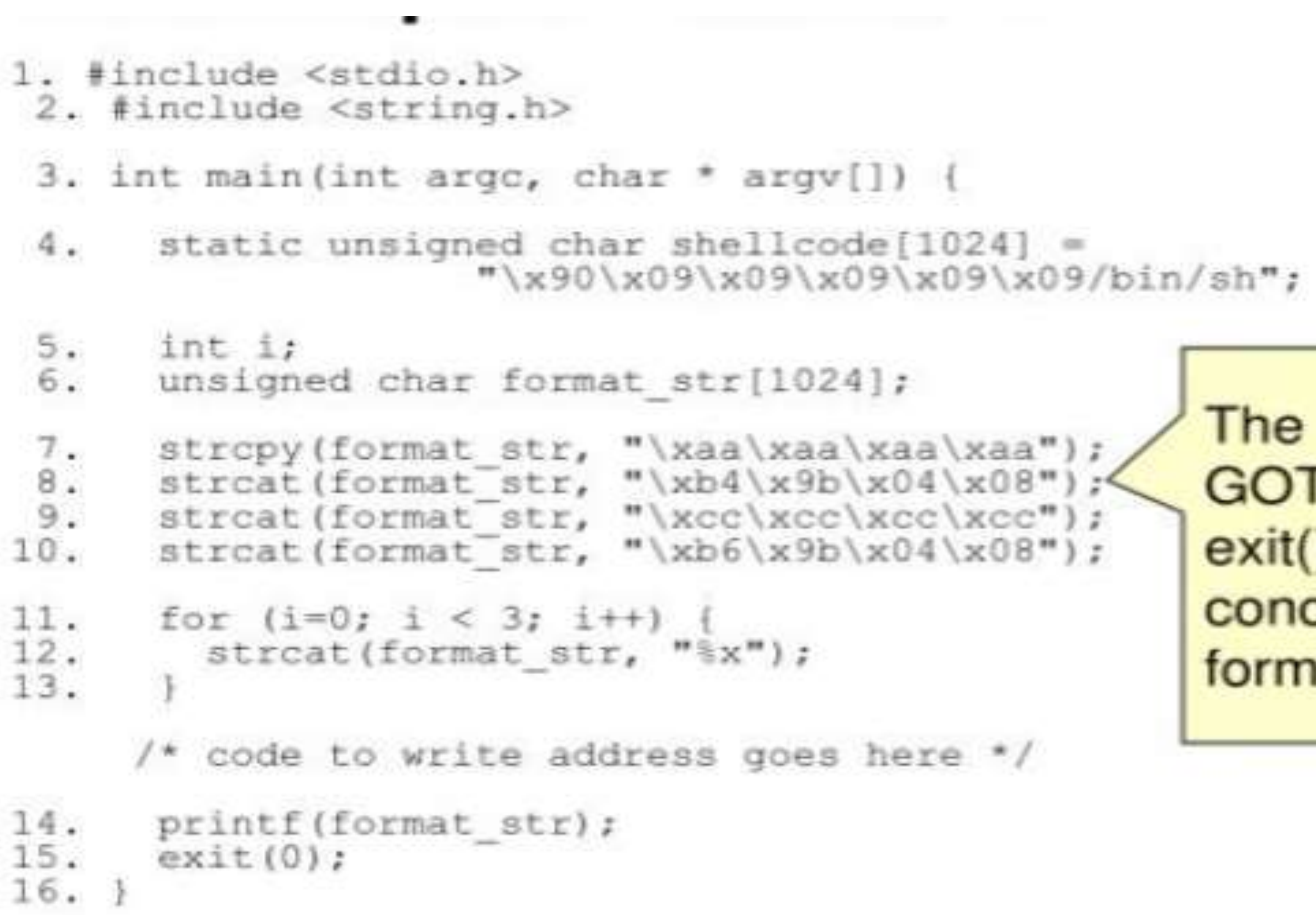
Linux Exploit Variant-1:



Example 6.9 Linux Exploit Variant

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

int main(void) {

  static unsigned char shellcode[1024] =
                "\x90\x09\x09\x09\x09\x09/bin/sh";

  size_t i;
  unsigned char format_str[1024];

  strcpy(format_str, "\xaa\xaa\xaa\xaa");
  strcat(format_str, "\xb4\x9b\x04\x08");
  strcat(format_str, "\xcc\xcc\xcc\xcc");
  strcat(format_str, "\xb6\x9b\x04\x08");

  for (i=0; i < 3; i++) {
    strcat(format_str, "%x");
  }

  /* code to write address goes here */

  printf(format_str);
  exit(0);
}
```

Linux Exploit Variant-1:

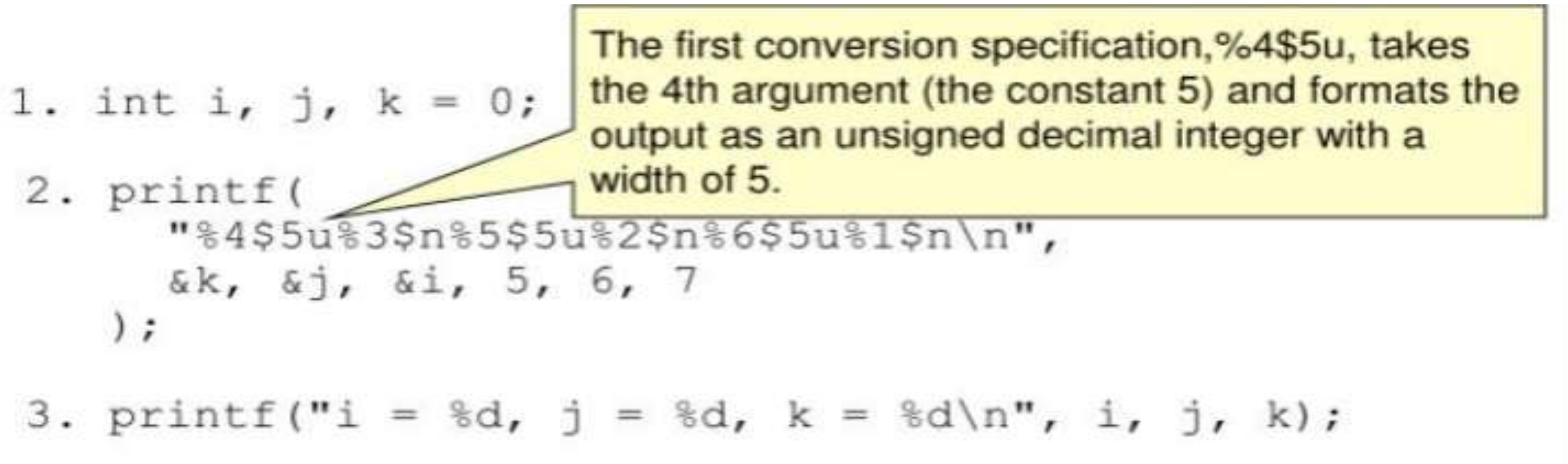


```c
#include <stdio.h>
#include <string.h>

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

  static unsigned char shellcode[1024] =
              "\x90\x09\x09\x09\x09\x09/bin/sh";

  int i;
  unsigned char format_str[1024];

  strcpy(format_str, "\xaa\xaa\xaa\xaa");
  strcat(format_str, "\xb4\x9b\x04\x08");
  strcat(format_str, "\xcc\xcc\xcc\xcc");
  strcat(format_str, "\xb6\x9b\x04\x08");

  for (i=0; i < 3; i++) {
    strcat(format_str, "%x");
  }

 /* code to write address goes here */

  printf(format_str);
  exit(0);
}
```

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

UNIT-4—Formatted Output

Direct Parameter Access:

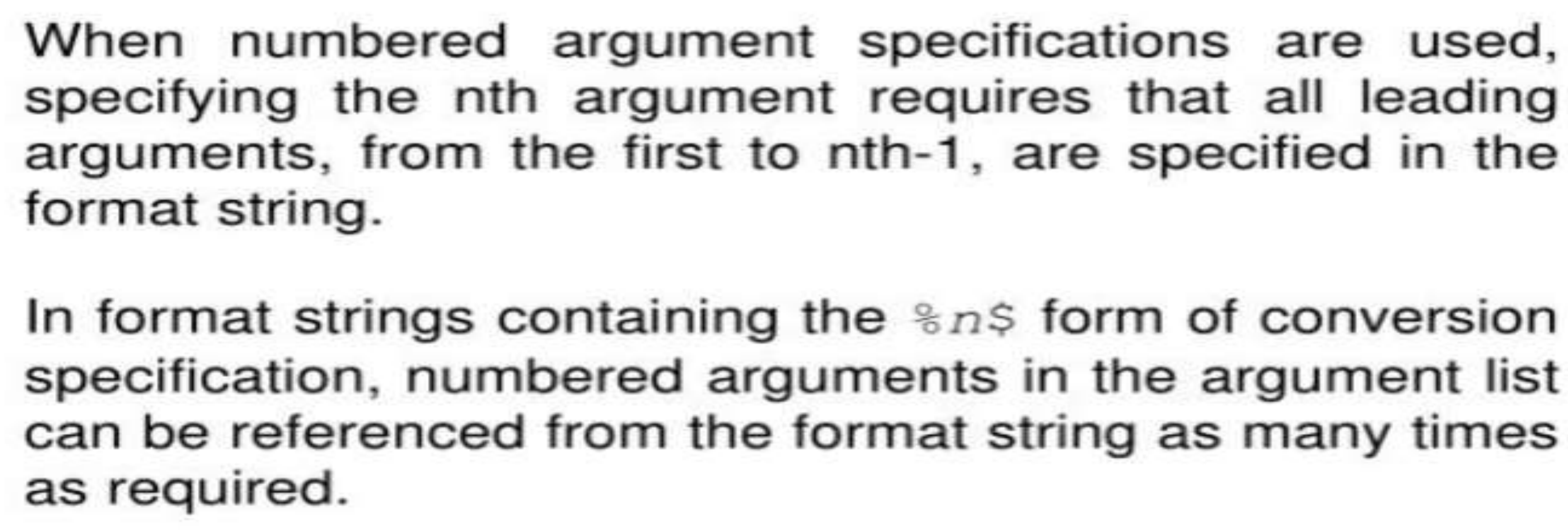


```
int i, j, k = 0;

printf(
   "%4$5u%3$n%5$5u%2$n%6$5u%1$n\n",
   &k, &j, &i, 5, 6, 7
  );

printf("i = %d, j = %d, k = %d\n", i, j, k);
```

The first conversion specification,%4$5u, takes the 4th argument (the constant 5) and formats the output as an unsigned decimal integer with a width of 5.

UNIT-4—Formatted Output

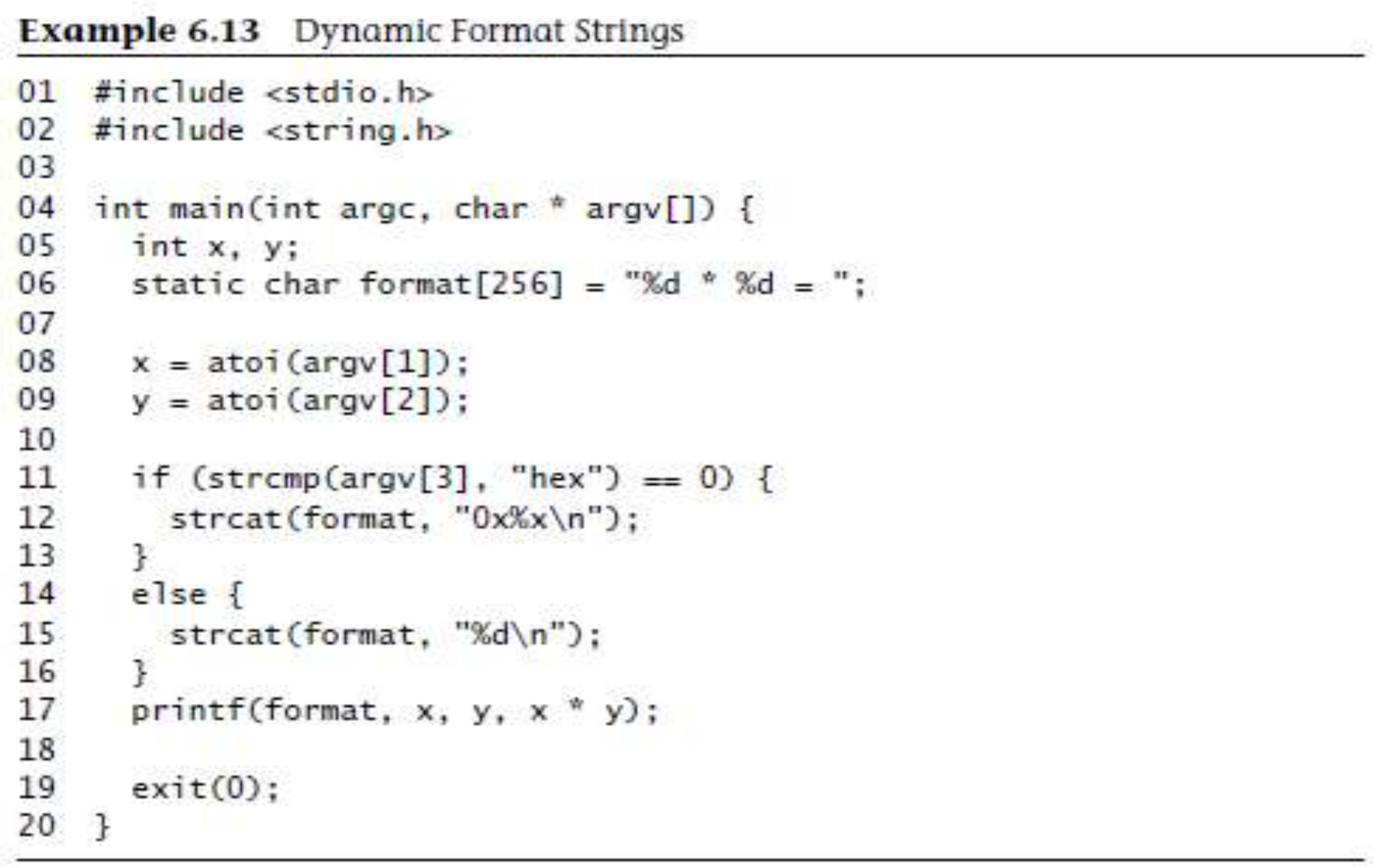
Direct Parameter Access:



When numbered argument specifications are used, specifying the nth argument requires that all leading arguments, from the first to nth-1, are specified in the format string.

In format strings containing the `%n$` form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

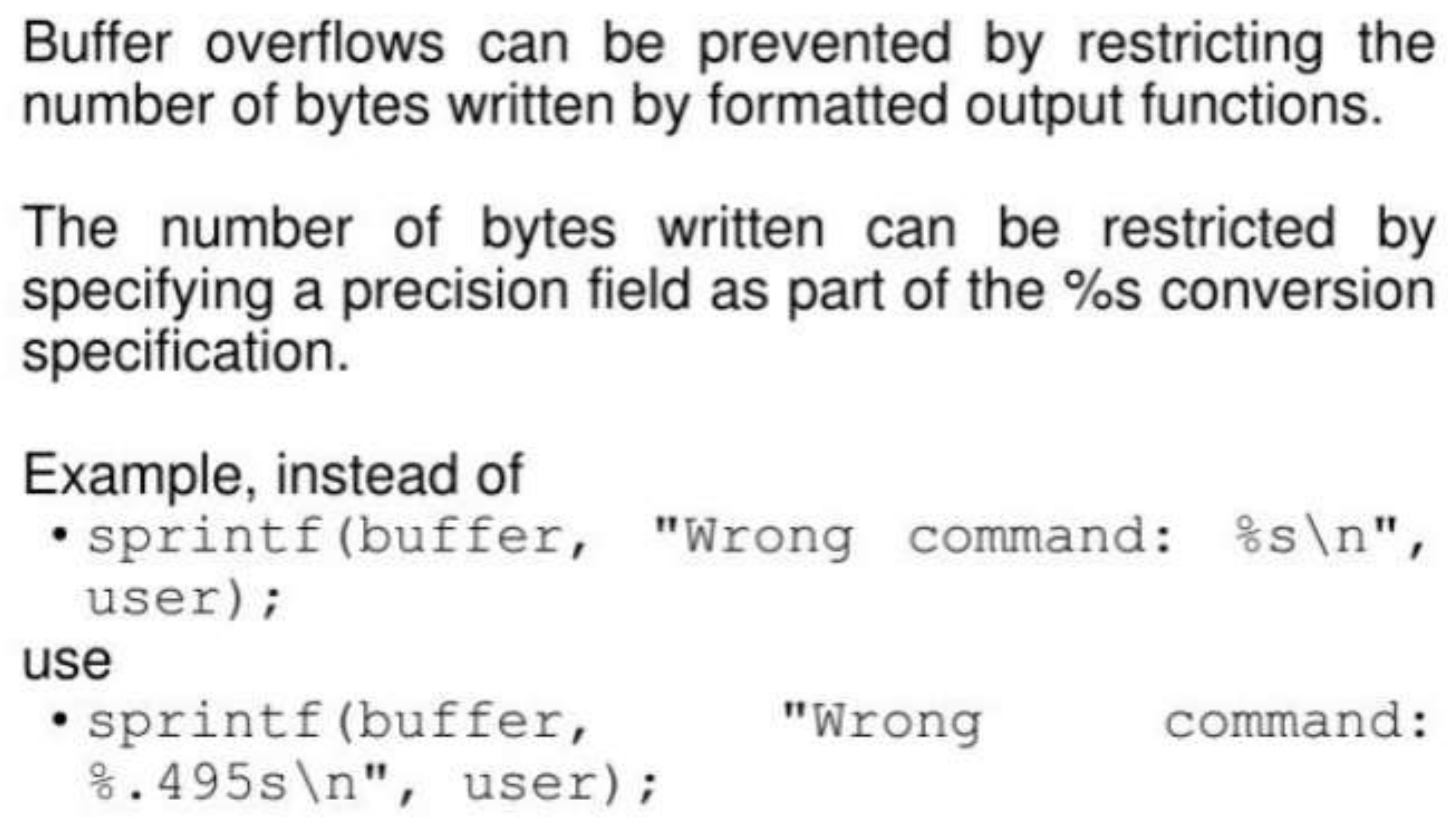
Mitigation Strategies: Dynamic Format Strings



Example 6.13 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);
}
```

Mitigation Strategies: 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);`

Mitigation Strategies: Restricting bytes written

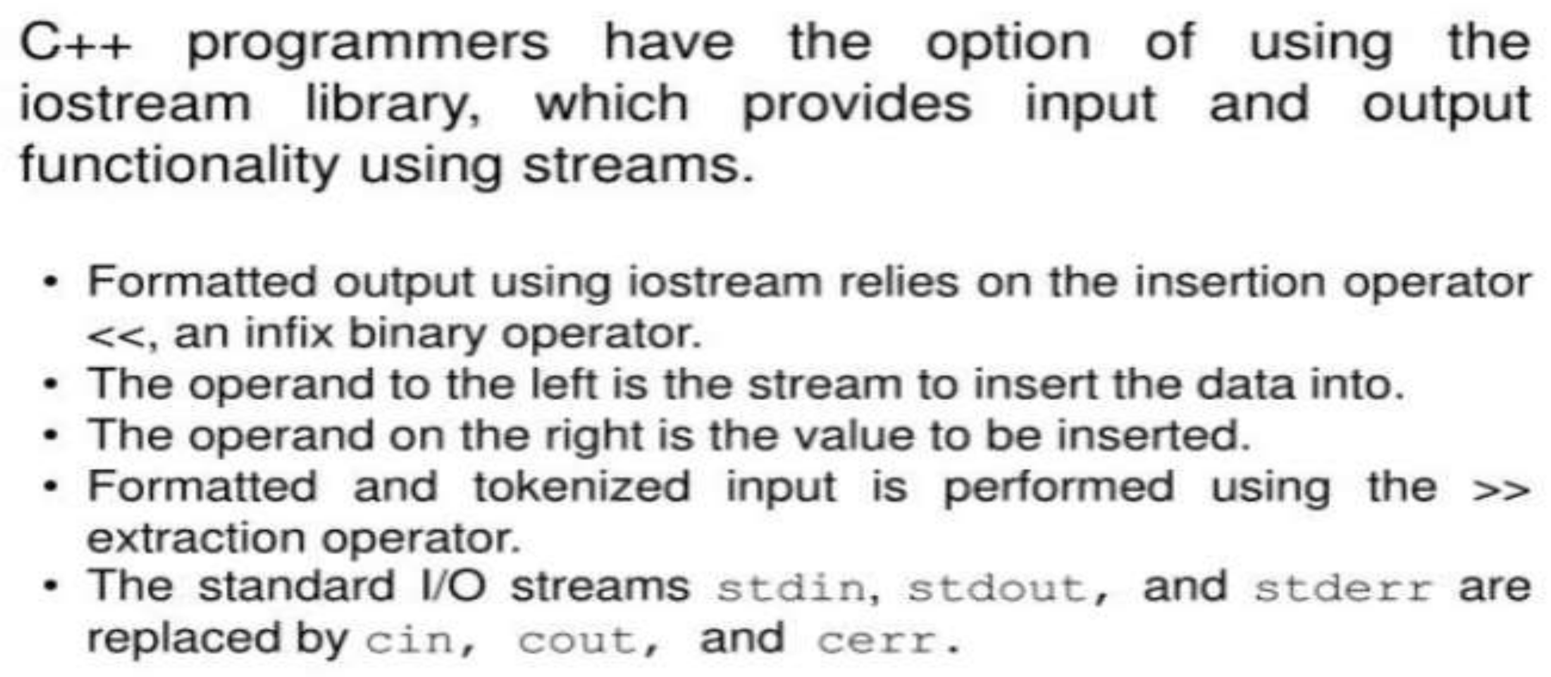
Use more secure versions of formatted output library functions that are less susceptible to buffer overflows.

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.

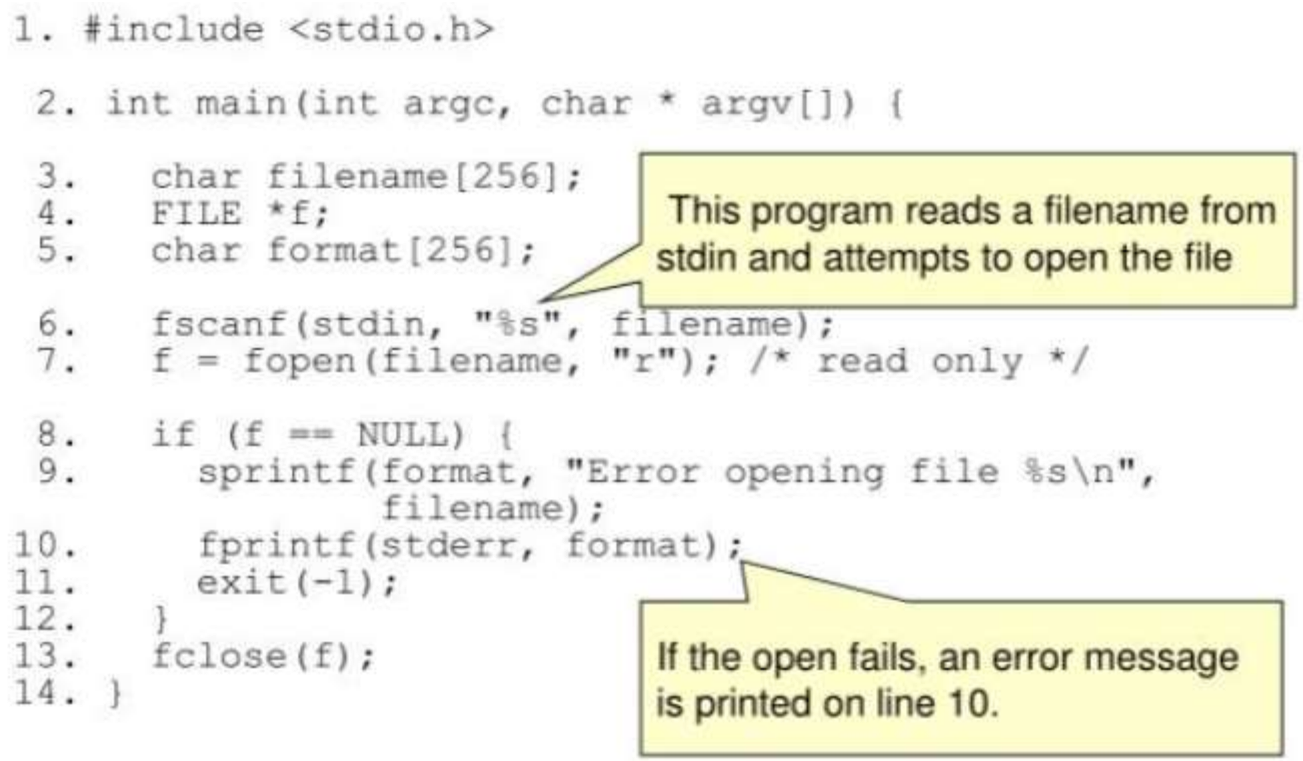
Mitigation Strategies: iostream vs stdio



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`.

Mitigation Strategies: Extremely insecure stdio implementation



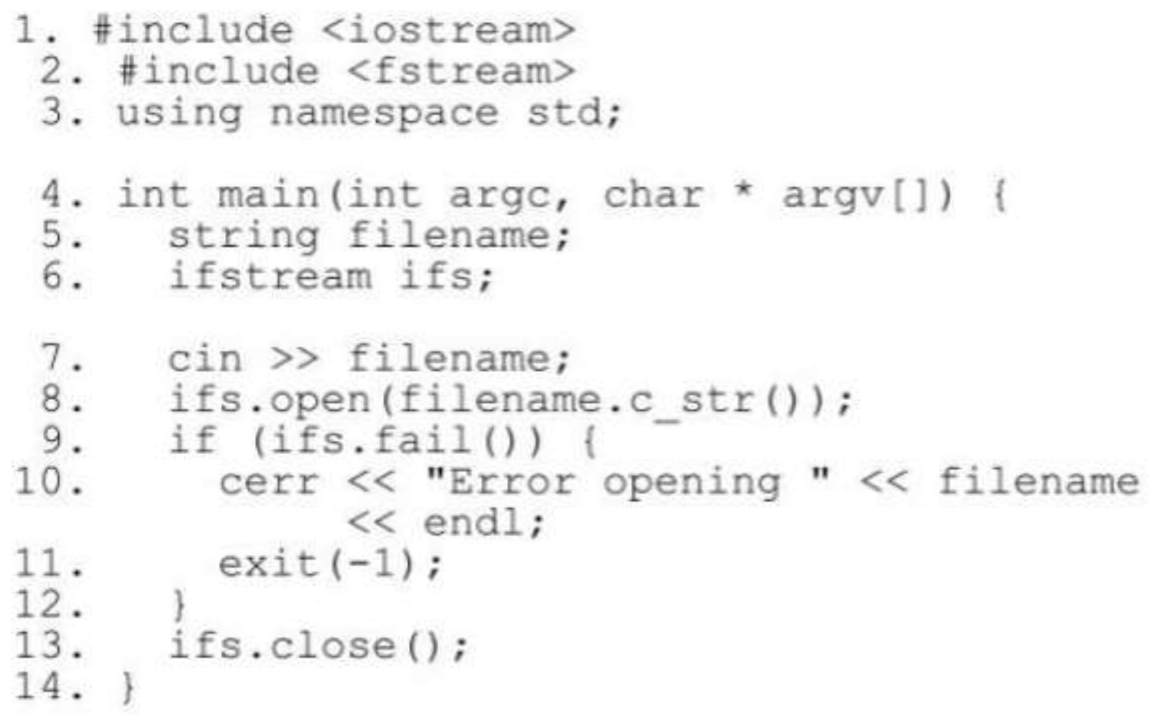
```
1. #include <stdio.h>

 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.

UNIT-4—Formatted Output

Mitigation Strategies: Secure iostream implementation



```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char * argv[]) {
  string filename;
  ifstream ifs;

  cin >> filename;
  ifs.open(filename.c_str());
  if (ifs.fail()) {
    cerr << "Error opening " << filename
         << endl;
    exit(-1);
  }
  ifs.close();
}
```

UNIT-4—Formatted Output

Mitigation Strategies: 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`.

UNIT-4—Formatted Output

Mitigation Strategies: Compiler Checks

-**Wformat**– This flag instructs GCC to check calls to formatted output functions, examine the format string, and verify that the correct number and type of arguments are supplied.

-**Wformat-nonliteral**: This flag performs the same function as -Wformat but adds warnings if the format string is not a string literal and cannot be checked, unless the format function takes its format arguments as a va_list.

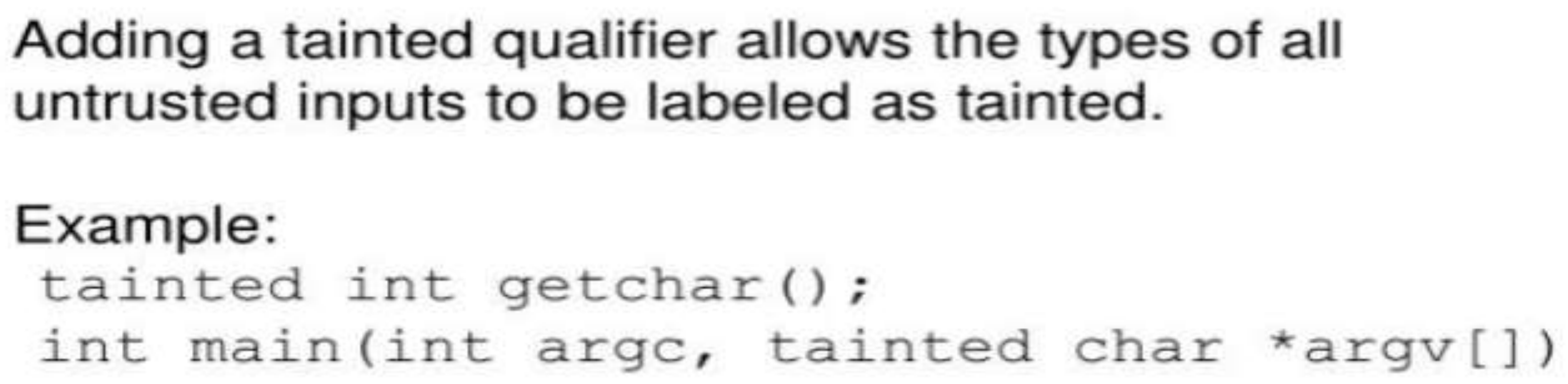
-Wformat-security: This flag performs the same function as -Wformat but adds warnings about formatted output function calls that represent possible security problems. At present this warns about calls to printf() where the format string is not a string literal and there are no format arguments.

Mitigation Strategies: Static taint Analysis

static taint analysis is a technique used to identify potential vulnerabilities in software programs by analyzing the flow of "tainted" (untrusted or user-controlled) data through the code. The goal of static taint analysis is to track how untrusted data is propagated through the program to determine if it eventually reaches a sensitive point where it could cause harm, such as causing a buffer overflow, format string vulnerability

UNIT-4—Formatted Output

Mitigation Strategies: Static taint Analysis



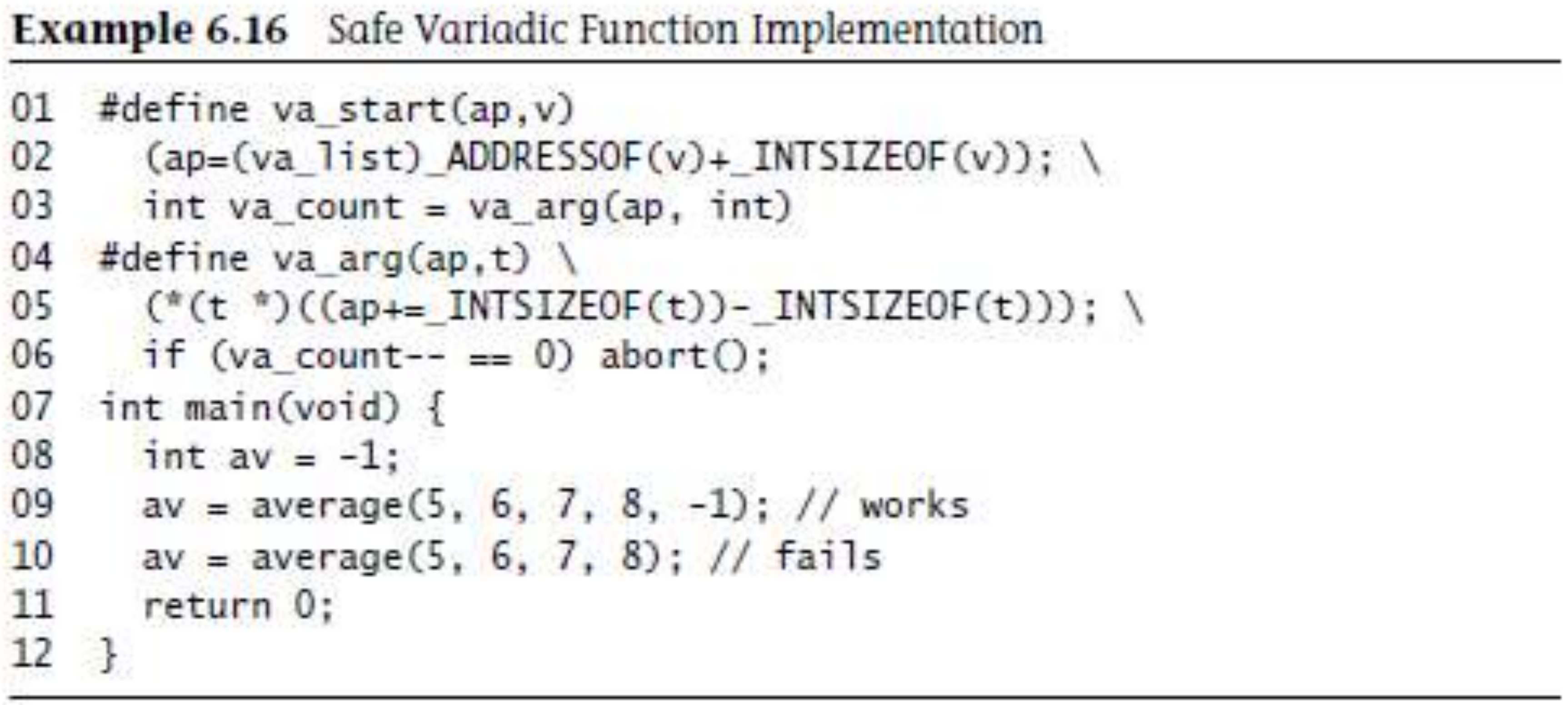
Adding a tainted qualifier allows the types of all untrusted inputs to be labeled as tainted.

Example:

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

- In this example , 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 annonations, 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 tainted type is used as a format string, the user is warned of potential vulnerability.

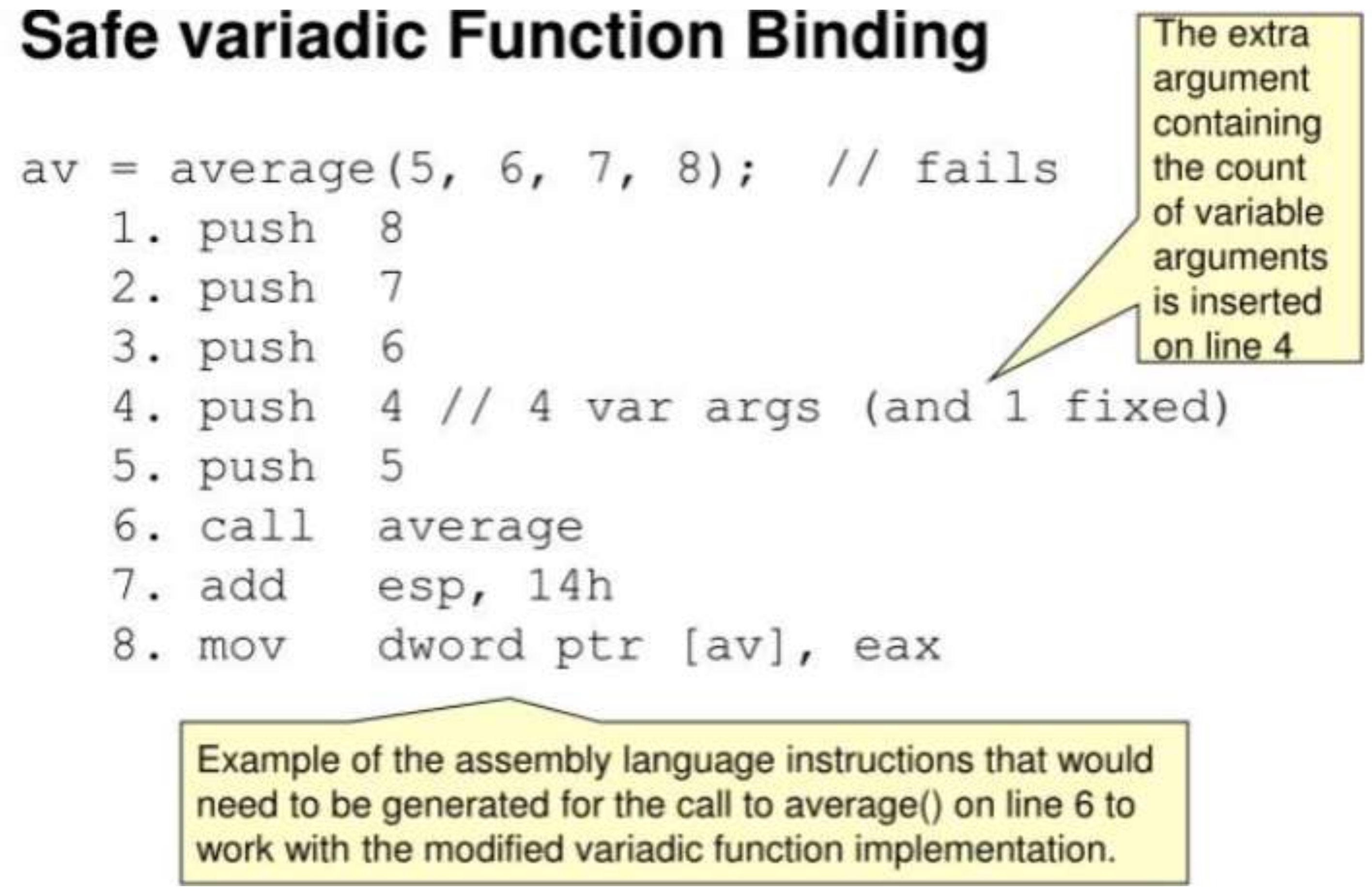
Mitigation Strategies: Modifying the variadic function implementation



Example 6.16 Safe Variadic Function Implementation

```
#define va_start(ap,v)
  (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v)); \
  int va_count = va_arg(ap, int)
#define va_arg(ap,t) \
  (*(t *)((ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
  if (va_count-- == 0) abort();
int main(void) {
  int av = -1;
  av = average(5, 6, 7, 8, -1); // works
  av = average(5, 6, 7, 8); // fails
  return 0;
}
```

Mitigation Strategies: Modifying the variadic function implementation



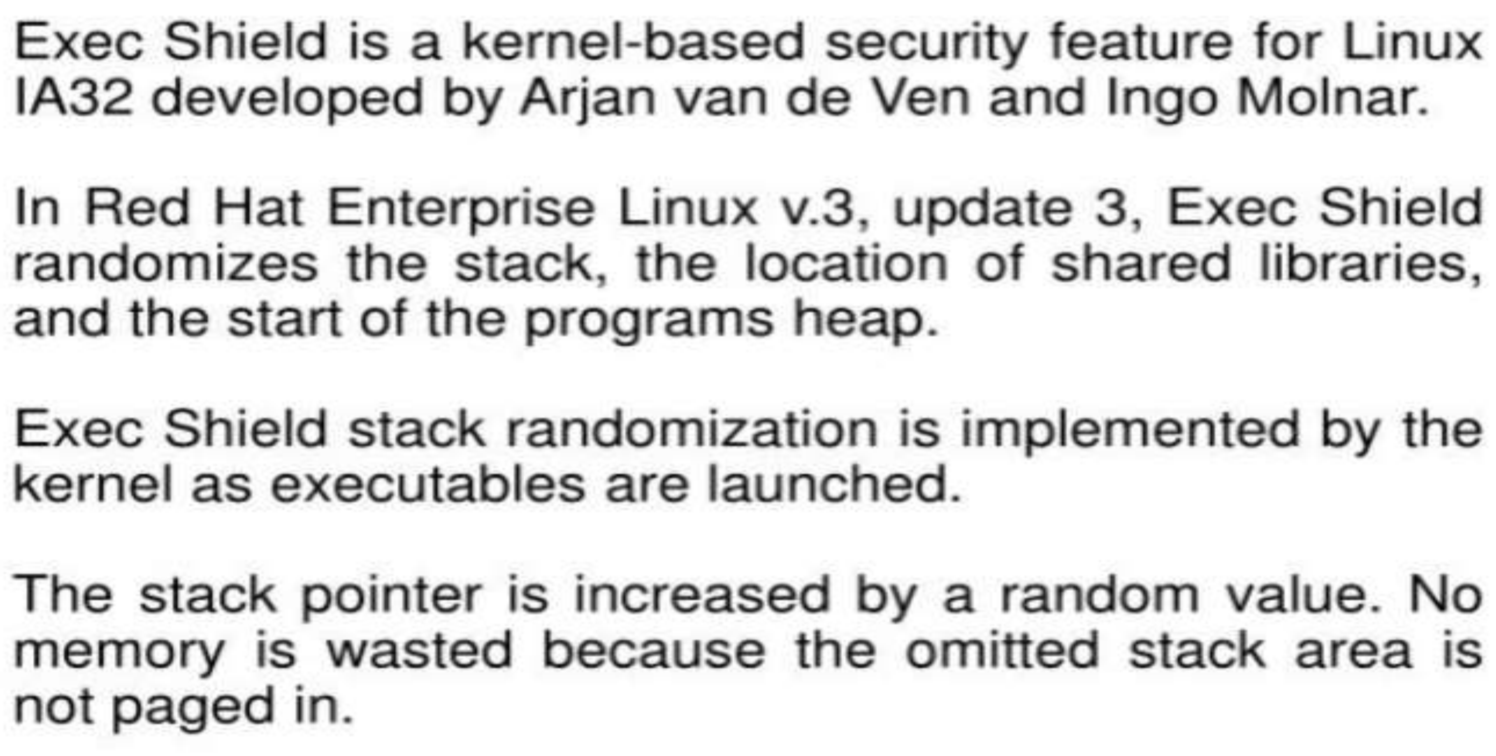
Safe variadic Function Binding

```
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
```

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

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.

Mitigation Strategies: Exec Shield



Exec Shield is a kernel-based security feature for Linux IA32 developed by Arjan van de Ven and Ingo Molnar.

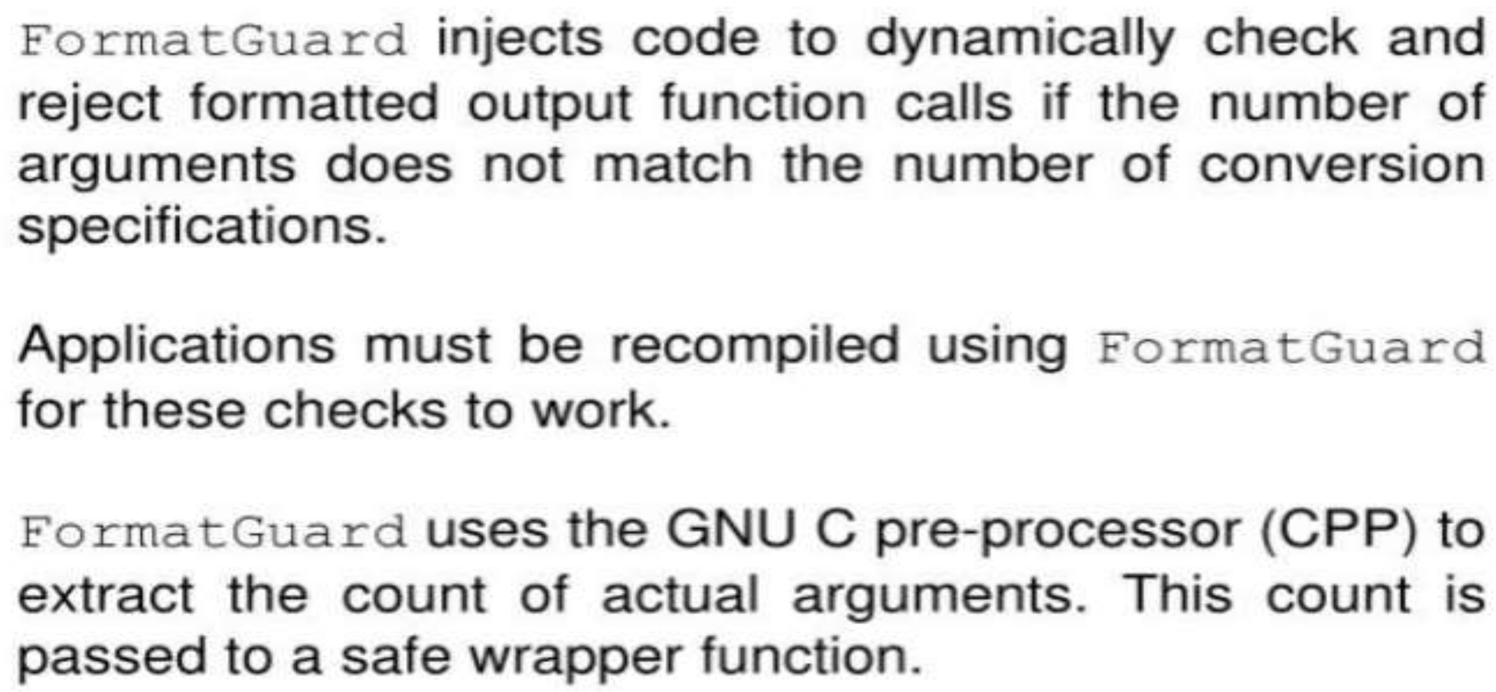
In Red Hat Enterprise Linux v.3, update 3, Exec Shield randomizes the stack, the location of shared libraries, and the start of the programs heap.

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.

UNIT-4—Formatted Output

Mitigation Strategies: Format Guard



`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.

Mitigation Strategies: Format Guard

The wrapper parses the format string to determine how many arguments to expect.

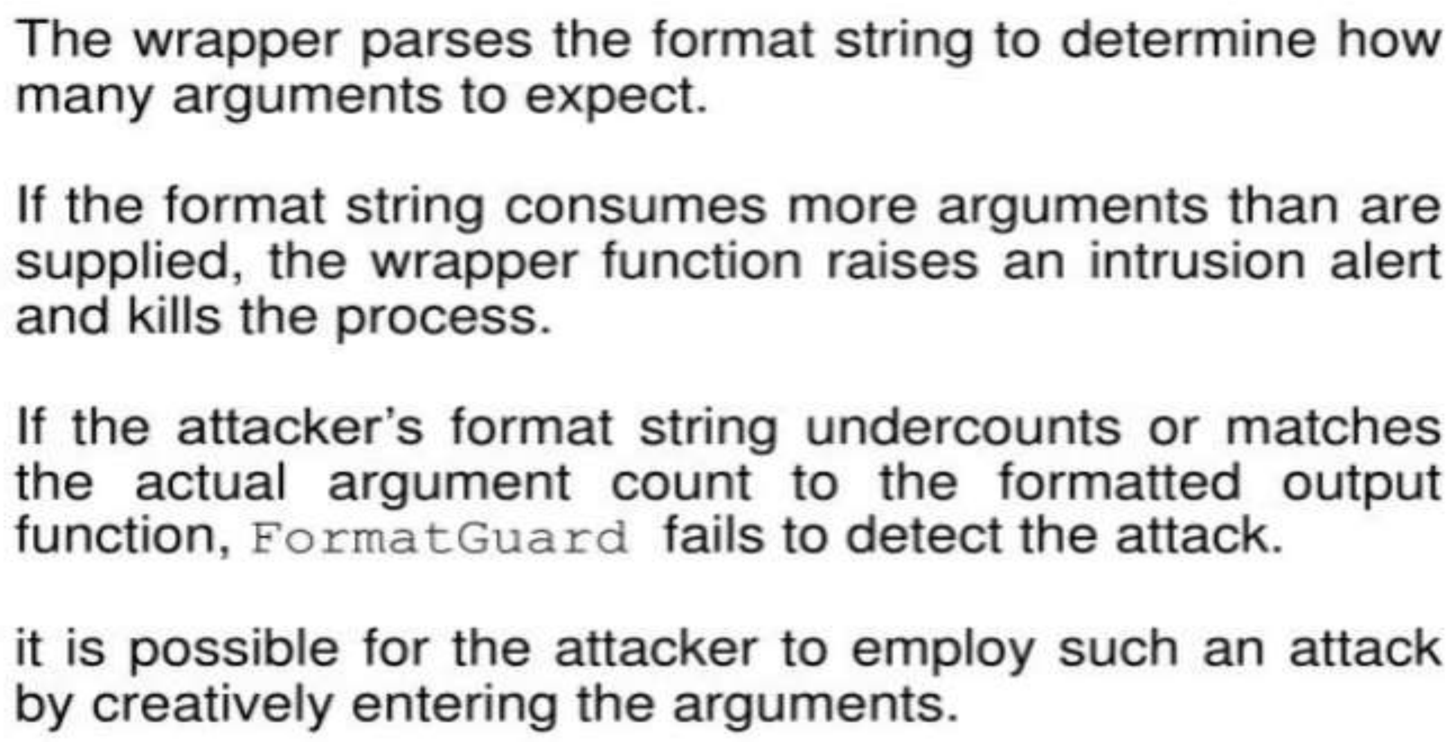
If the format string consumes more arguments than are supplied, the wrapper function raises an intrusion alert and kills the process.

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.

UNIT-4—Formatted Output

Mitigation Strategies: Format Guard



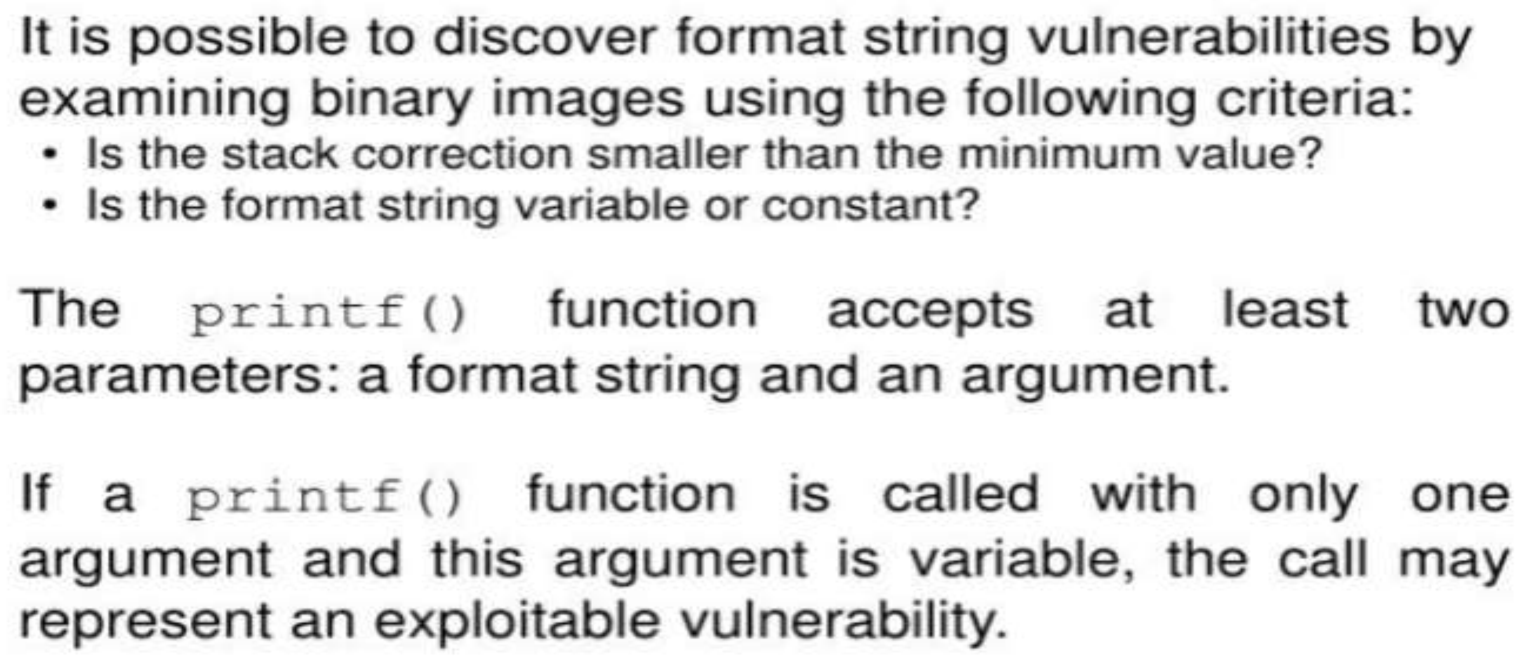
The wrapper parses the format string to determine how many arguments to expect.

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.

Mitigation Strategies: Static Binary Analysis



It is possible to discover format string vulnerabilities by examining binary images using the following criteria:

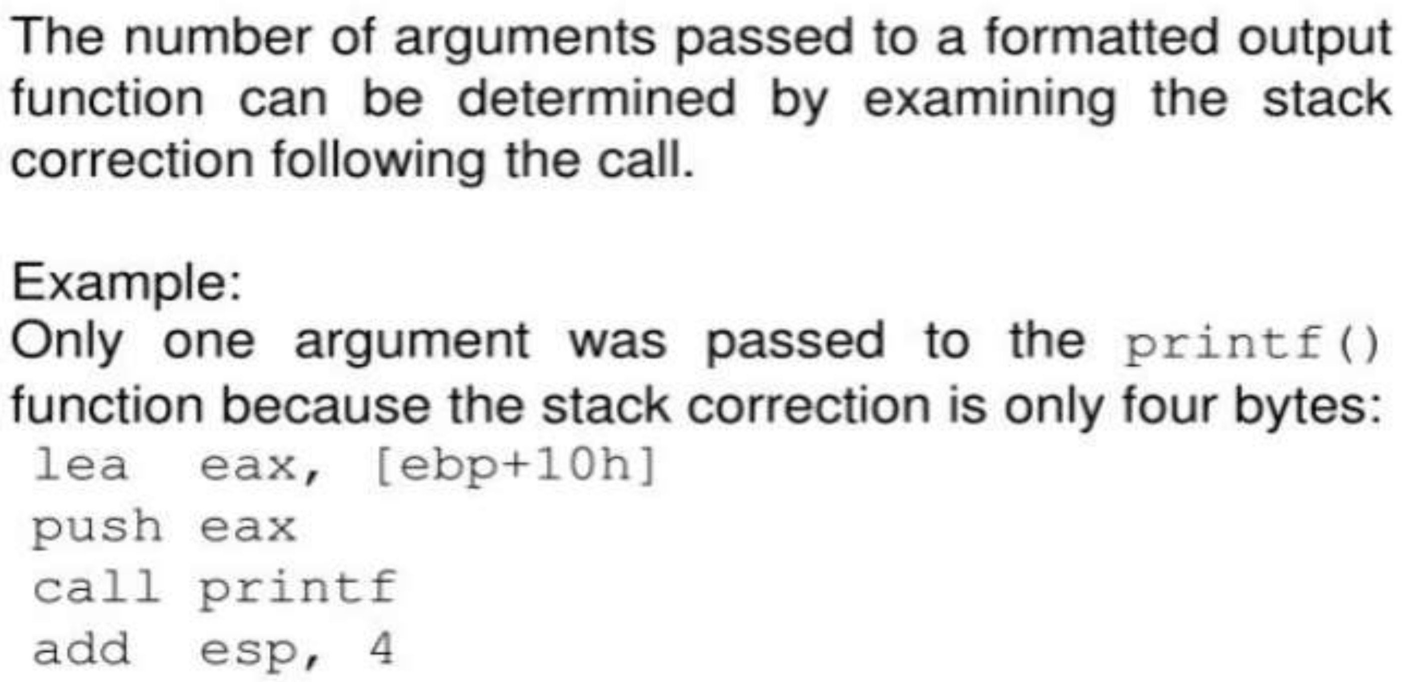
- Is the stack correction smaller than the minimum value?
- Is the format string variable or constant?

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.

UNIT-4—Formatted Output

Mitigation Strategies: 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
```

UNIT-4—Formatted Output

Notable Vulnerabilities: Washington University FTP Daemon

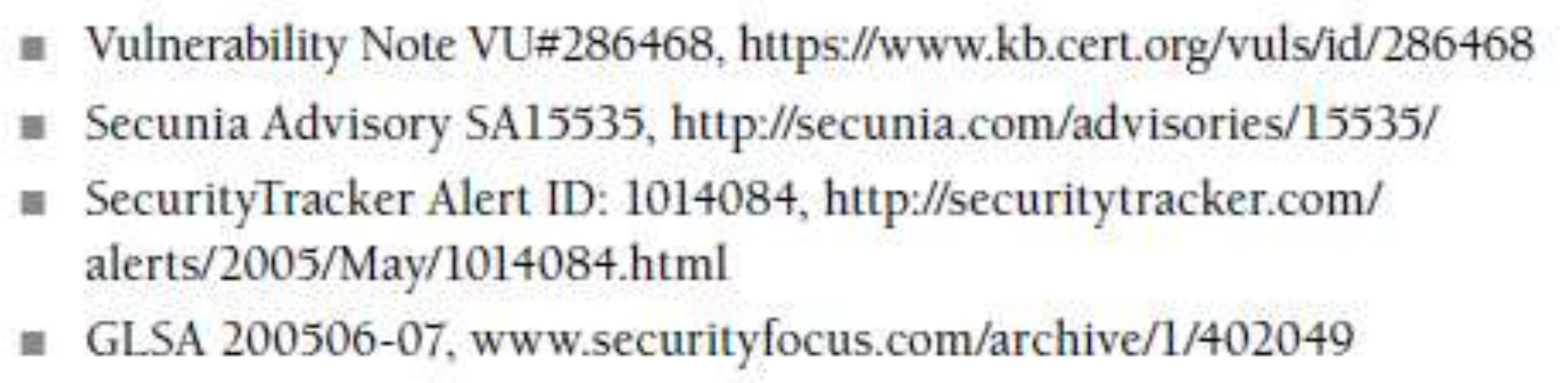
Washington University FTP daemon (wu-ftpd) is a popular UNIX FTP server shipped with many distributions of Linux and other UNIX operating systems.

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.

UNIT-4—Formatted Output

Notable Vulnerabilities: Ettercap version NG-0.72



- Vulnerability Note VU#286468, https://www.kb.cert.org/vuls/id/286468
- Secunia Advisory SA15535, http://secunia.com/advisories/15535/
- SecurityTracker Alert ID: 1014084, http://securitytracker.com/alerts/2005/May/1014084.html
- GLSA 200506-07, www.securityfocus.com/archive/1/402049