Software Security 04

Sven Schäge

Learning Goals

- Understand the abstraction behind the concept of format string vulnerabilities.
- Be able to use that concept in other attack scenarios (of Kingdom 1) as well.
- Understand how printf – and variadic functions – work.
- Be able to visualize the stack state when variadic functions are called.
- Be able to connect attacks to improve attack success: code injection and buffer overflows (to read out the stack).

Format String Vulnerabilities

- Requirement: program relies on user-supplied input as part of a format string for certain functions without proper validation or sanitization.
- Can lead to unauthorized access, information disclosure, or even remote code execution.
- In C and C++, functions like printf and sprint are often used to format and print strings.
- Functions like scanf are used to read in user input in some formatted way.
- These functions allow the use of **format specifiers** (such as %s for strings, %d for integers, etc.) to control the output format.
- When user input is directly incorporated into the format string without proper validation, an attacker can exploit this vulnerability.

Format Strong Vulnerability

- Surprised the security community
- Applicable to old software that was written at a time when the vulnerable mechnism were not considered problematic

Printf – Variadic Functions

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char user_input[100];
    printf("Please enter your name: \n");

    gets(user_input); // unsafe function, use fgets instead

    printf("Thanks. Your name is: %s",user_input);
}
```

```
Alice

Program returned: 0
Program stdout
Please enter your name:
Thanks. Your name is: Alice
```

format string

format specifier

list of input arguments -- comma separated, variable length

Variadic Functions

- Have a variable number of inputs that are expected to be on the stack
- The format strings decides how many inputs are read from the stack
  - “%s” – one argument
  - “%s%s” – two arguments
- Examples
  - printf
  - fprintf
  - sprintf
  - vfprintf
  - snprintf
  - vsprintf
  - vsnprintf

Format Specifiers in C

SPECIFIER	USED FOR
%c	a single character
%s	a string **%s** indicates that the data should be a pointer to a string!
%hi	short (signed)
%hu	short (unsigned)
%Lf	long double
%n	prints nothing
%d	a decimal integer (assumes base 10)
%i	a decimal integer (detects the base automatically)
%o	an octal (base 8) integer
%x	a hexadecimal (base 16) integer
%p	an address (or pointer)
%f	a floating point number for floats
%u	int unsigned decimal
%e	a floating point number in scientific notation
%E	a floating point number in scientific notation
%%	the % symbol

Digits can be used as modifiers: %4x outputs a field
printf("%10$x");
-> will print the tenth element next on the stack

```c
#include <stdio.h>

int main() {
  char user_ch;
  scanf("%c", &user_ch); // user inputs Y
  printf("%c\n", user_ch);
  return 0;
}
```

```
Y

Program returned: 0
Program stdout
Y
```

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("%4d\n", 1);
    printf("%4d\n", 12);
    printf("%4d\n", 123);
    printf("%4d\n", 1234);
}
```

```
Program stdout
   1
  12
 123
1234
```

```c
#include <stdio.h>

int main() {
  char user_str[20];
  scanf("%s", user_str); // user inputs fCC
  printf("%s\n", user_str);
  return 0;
}
```

```
fCC

Program returned: 0
Program stdout
fCC
```

```c
#include <stdio.h>

int main() {
  int found = 2015, curr = 2020;
  printf("%d\n", found);
  printf("%i\n", curr);
  return 0;
}
```

```
Program stdout
2015
2020
```

Format String Attack: Basic Idea

- Code

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char user_input[100];
    printf("Please enter your name: \n");
    gets(user_input); // unsafe function, use fgets instead
    printf("Thanks. Your name is: ");
    printf(user_input); // format string vulnerability
    return 0;
}
```

- Normal execution

Execution arguments...

SoftwareSecurityStudent

```
Program returned: 0
Program stdout
Please enter your name:
Thanks. Your name is: SoftwareSecurityStudent
```

- Reading the stack

Execution arguments...

%x%x%x%x%x%x%x%x%x%x

```
Program returned: 0
Program stdout
Please enter your name:
Thanks. Your name is: 40201e001212e478161212e4f07825782578257825782578251212e500
```

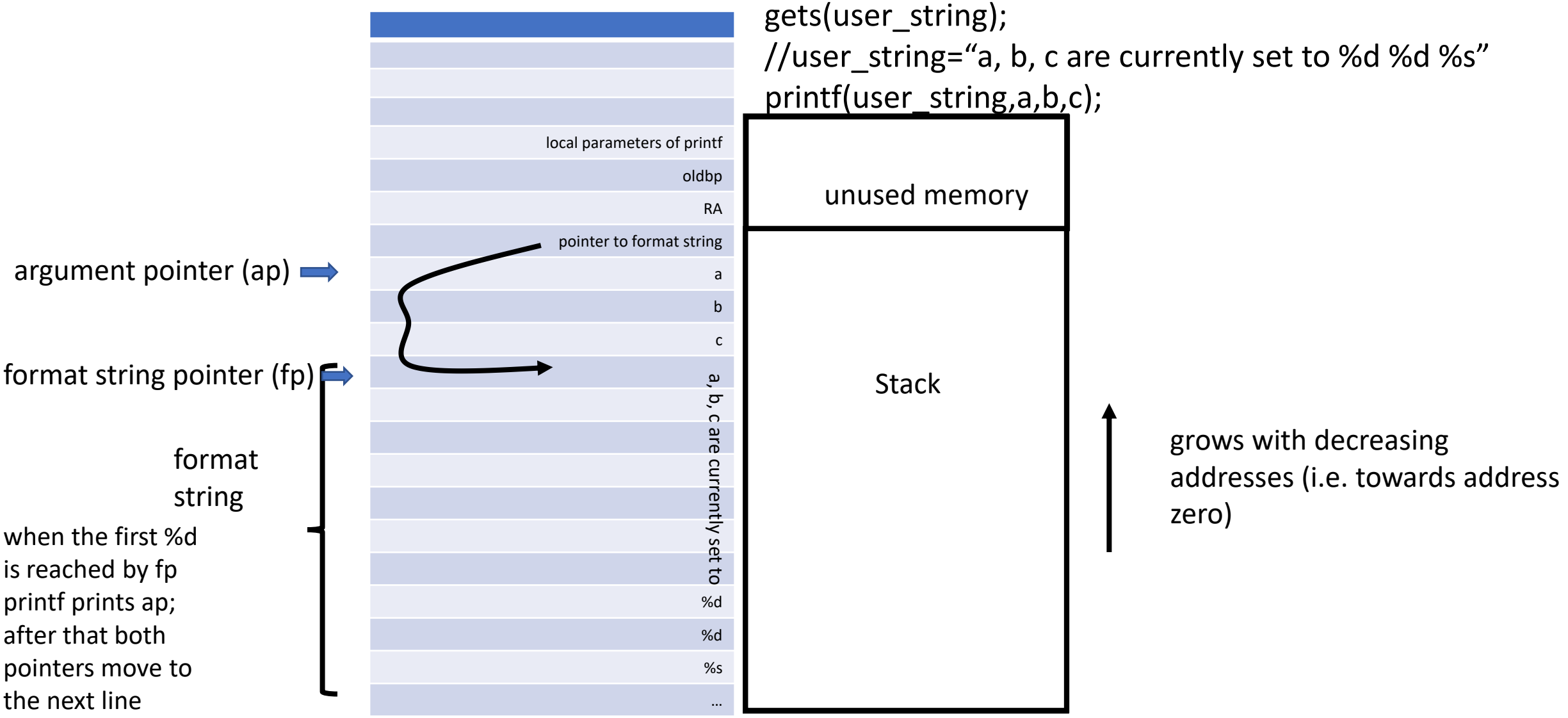
%1$x

```
Program returned: 0
Program stdout
Please enter your name:
Thanks. Your name is: 40201e
```

Format String Vulnerabilities

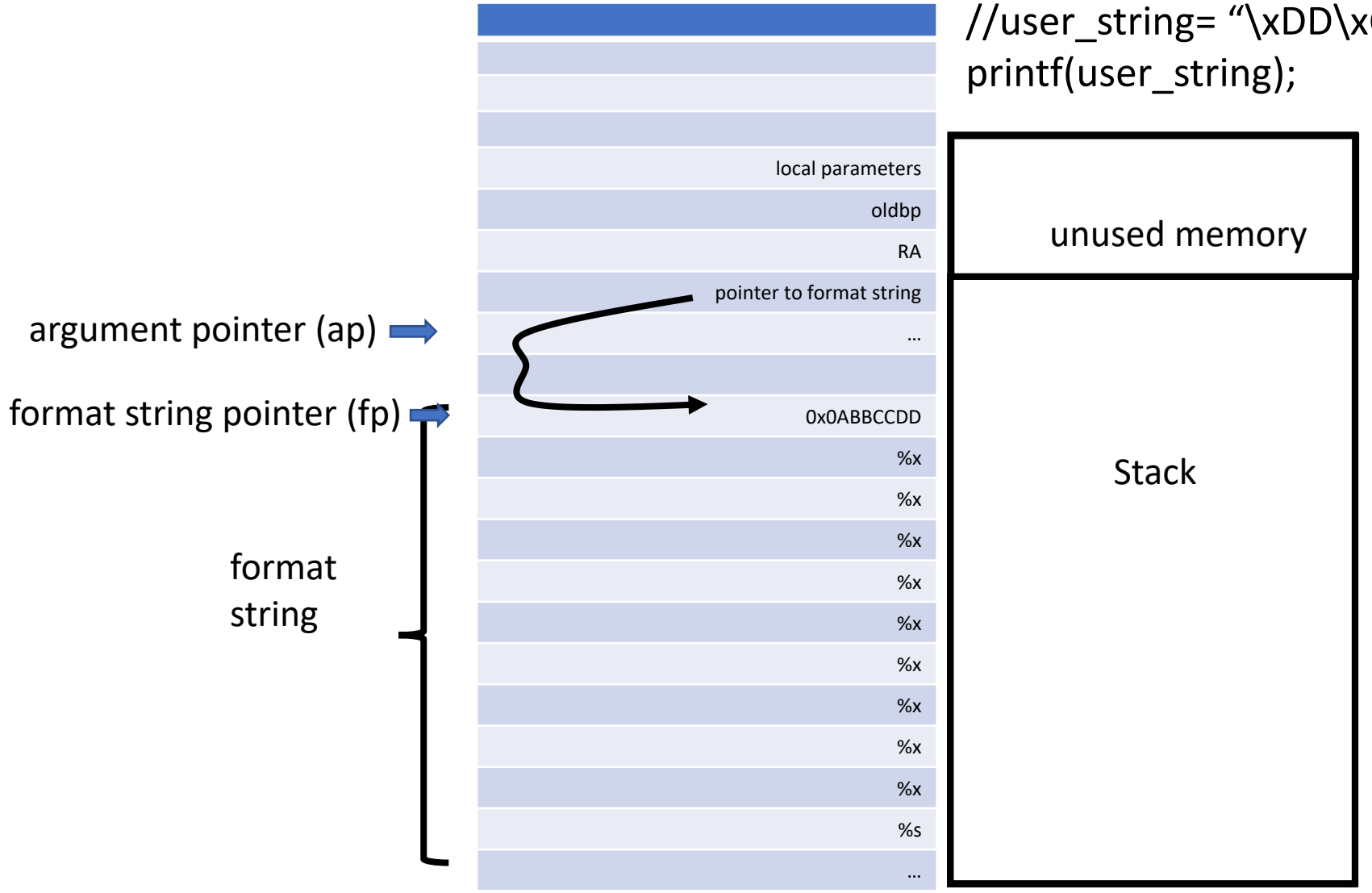
- The vulnerability occurs when there is a mismatch between the number of **format specifiers** in the string and the number of **function arguments** (like A and B from above) provided to fill those places. If an attacker is able to supply more placeholders than there are parameters, she can use the format function to read or write the stack.ontrol over the format string fed into the function:
  - **printf(**“A is the number %d, reading stack data: %x”, A**)**;
  - **printf()** will still attempt to retrieve two values from the stack
- Compilers in general cannot detect this inconsistency because format strings can be generated dynamically at runtime.
- Printf does not implement checks to detect this inconsistency: it just prints out the next value on the stack, once it sees the next format specifier
- When **%s** is used as the format specifier, the function will treat the data on the stack as an address to go fetch a string from. (This is called **pass by reference**.) This means that the attacker can potentially make **%s** read from any address, even if the data is not located on the stack.
  - Store address of format string on the stack, occupying a dynamical number of memory units
  - Move towards the memory unit that holds this address. This can be achieved by successively reading out the next stack value.
  - Read it via %s

Program Memory Layout – Calling printf



Program Memory Layout – Calling printf

```
gets(user_string);
//user_string= “\xDD\xCC\xBB\x0A %x %x %x %x %x %x %x %s”
printf(user_string);
```



Format String Attacks: Writing

- In **printf()**, **%n** is a special case format specifier. Instead of being replaced by a function argument, %n will cause the **number of characters written so far** to be stored into the corresponding function argument.
- For example, the following code will store the integer **5** into the variable **num_char**.

```
1   #include <stdio.h>
2
3   int main(int argc, char *argv[]) {
4       int num_char;
5       printf("11111%n", &num_char);
6       printf("%i",num_char);
7       return 0;
8   }
```

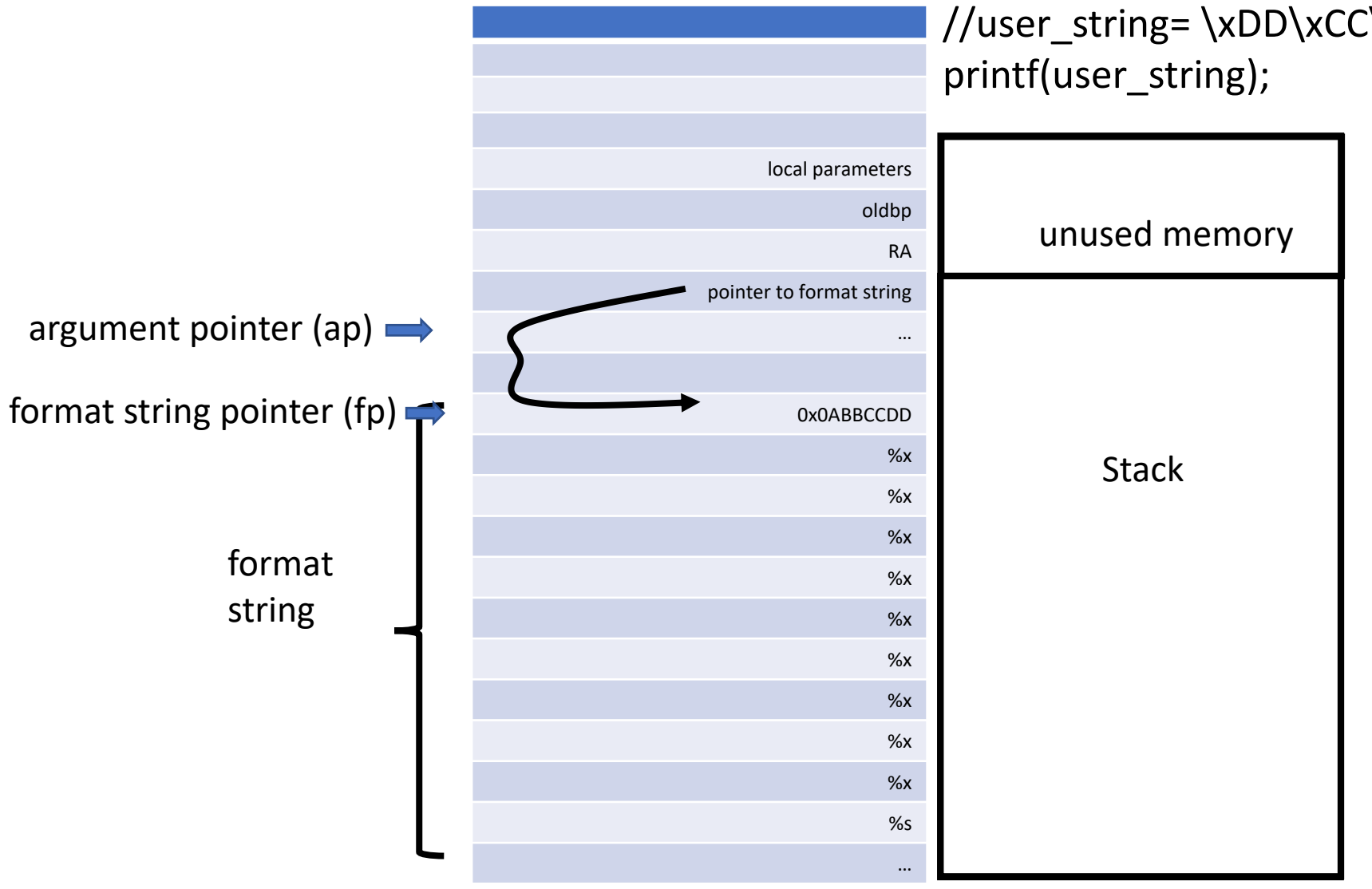
Writing More Concisely

```c
#include <stdio.h>

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


    int num_char;
    printf("Let us write on the stack: \n");
    printf("11111%n", &num_char);
    printf("\n num_char=%i",num_char);

    printf("%10d%n", 0, &num_char); //writes 4 bytes to &num_char
    printf("\n num_char=%i",num_char);

    printf("%100d%hn", 0, &num_char); // writes 2 bytes to &num_char
    printf("\n num_char=%i\n\n",num_char);

}
```

```
Program stdout
Let us write on the stack:
11111
 num_char=5         0
 num_char=10                                                                0
 num_char=100
```

Program Memory Layout – Calling printf

```
gets(user_string);
//user_string= \xDD\xCC\xBB\x0A %x %x %x %x %x %x %n
printf(user_string);
```



Instructive Examples of Security Vulnerabilities Related to Memory Corruption

Prioritizing Widespread Problems

The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Out-of-bounds Write

CWE-787: Out-of-bounds Write

Weakness ID: 787
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The software writes data past the end, or before the beginning, of the intended buffer.

Extended Description

Typically, this can result in corruption of data, a crash, or code execution. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent write operation then produces undefined or unexpected results.

Alternate Terms

Memory Corruption: The generic term "memory corruption" is often used to describe the consequences of writing to memory outside the bounds of a buffer, or to memory that is invalid, when the root cause is something other than a sequential copy of excessive data from a fixed starting location. This may include issues such as incorrect pointer arithmetic, accessing invalid pointers due to incomplete initialization or memory release, etc.

Example 1

The following code attempts to save four different identification numbers into an array.

Example Language: **C** *(bad code)*

```
int id_sequence[3];

/* Populate the id array. */

id_sequence[0] = 123;
id_sequence[1] = 234;
id_sequence[2] = 345;
id_sequence[3] = 456;
```

Example 1

The following code attempts to save four different identification numbers into an array.

Example Language: **C** *(bad code)*

```
int id_sequence[3];

/* Populate the id array. */

id_sequence[0] = 123;
id_sequence[1] = 234;
id_sequence[2] = 345;
id_sequence[3] = 456;
```

Since the array is only allocated to hold three elements, the valid indices are 0 to 2; so, the assignment to id_sequence[3] is out of bounds.

Example Language: **C** *(bad code)*

```
int returnChunkSize(void *) {

  /* if chunk info is valid, return the size of usable memory,

  * else, return -1 to indicate an error

  */
  ...
}
int main() {
  ...
  memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
  ...
}
```

Example 2

In the following example, it is possible to request that memcpy move a much larger segment of memory than assumed:

Example Language: **C** *(bad code)*

```
int returnChunkSize(void *) {

  /* if chunk info is valid, return the size of usable memory,

  * else, return -1 to indicate an error

  */
  ...
}
int main() {
  ...
  memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
  ...
}
```

If returnChunkSize() happens to encounter an error it will return -1. Notice that the return value is not checked before the memcpy operation (CWE-252), so -1 can be passed as the size argument to memcpy() (CWE-805). Because memcpy() assumes that the value is unsigned, it will be interpreted as MAXINT-1 (CWE-195), and therefore will copy far more memory than is likely available to the destination buffer (CWE-787, CWE-788).

Example 5

In the following C/C++ example, a utility function is used to trim trailing whitespace from a character string. The function copies the input string to a local character string and uses a while statement to remove the trailing whitespace by moving backward through the string and overwriting whitespace with a NUL character.

Example Language: **C** *(bad code)*

```
char* trimTrailingWhitespace(char *strMessage, int length) {
  char *retMessage;
  char *message = malloc(sizeof(char)*(length+1));

  // copy input string to a temporary string
  char message[length+1];
  int index;
  for (index = 0; index < length; index++) {
    message[index] = strMessage[index];
  }
  message[index] = '\0';

  // trim trailing whitespace
  int len = index-1;
  while (isspace(message[len])) {
    message[len] = '\0';
    len--;
  }

  // return string without trailing whitespace
  retMessage = message;
  return retMessage;
}
```

Example 5

In the following C/C++ example, a utility function is used to trim trailing whitespace from a character string. The function copies the input string to a local character string and uses a while statement to remove the trailing whitespace by moving backward through the string and overwriting whitespace with a NUL character.

Example Language: **C** *(bad code)*

```
char* trimTrailingWhitespace(char *strMessage, int length) {
  char *retMessage;
  char *message = malloc(sizeof(char)*(length+1));

  // copy input string to a temporary string
  char message[length+1];
  int index;
  for (index = 0; index < length; index++) {
    message[index] = strMessage[index];
  }
  message[index] = '\0';

  // trim trailing whitespace
  int len = index-1;
  while (isspace(message[len])) {
    message[len] = '\0';
    len--;
  }

  // return string without trailing whitespace
  retMessage = message;
  return retMessage;
}
```

However, this function can cause a buffer underwrite if the input character string contains all whitespace. On some systems the while statement will move backwards past the beginning of a character string and will call the isspace() function on an address outside of the bounds of the local buffer.

Example Language: **C** *(bad code)*

```
int main() {
  ...
  strncpy(destBuf, &srcBuf[find(srcBuf, ch)], 1024);
  ...
}
```

Example 7

The following is an example of code that may result in a buffer underwrite, if find() returns a negative value to indicate that ch is not found in srcBuf:

Example Language: **C** *(bad code)*

```
int main() {
  ...
  strncpy(destBuf, &srcBuf[find(srcBuf, ch)], 1024);
  ...
}
```

If the index to srcBuf is somehow under user control, this is an arbitrary write-what-where condition.

Improper Input Validation

CWE-20: Improper Input Validation

Weakness ID: 20
Abstraction: Class
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.

Extended Description

Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components. When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

Input validation is not the only technique for processing input, however. Other techniques attempt to transform potentially-dangerous input into something safe, such as filtering (CWE-790) - which attempts to remove dangerous inputs - or encoding/escaping (CWE-116), which attempts to ensure that the input is not misinterpreted when it is included in output to another component. Other techniques exist as well (see CWE-138 for more examples.)

Input validation can be applied to:

- raw data - strings, numbers, parameters, file contents, etc.
- metadata - information about the raw data, such as headers or size

Data can be simple or structured. Structured data can be composed of many nested layers, composed of combinations of metadata and raw data, with other simple or structured data.

Many properties of raw data or metadata may need to be validated upon entry into the code, such as:

- specified quantities such as size, length, frequency, price, rate, number of operations, time, etc.
- implied or derived quantities, such as the actual size of a file instead of a specified size
- indexes, offsets, or positions into more complex data structures
- symbolic keys or other elements into hash tables, associative arrays, etc.
- well-formedness, i.e. syntactic correctness - compliance with expected syntax
- lexical token correctness - compliance with rules for what is treated as a token
- specified or derived type - the actual type of the input (or what the input appears to be)
- consistency - between individual data elements, between raw data and metadata, between references, etc.
- conformance to domain-specific rules, e.g. business logic
- equivalence - ensuring that equivalent inputs are treated the same
- authenticity, ownership, or other attestations about the input, e.g. a cryptographic signature to prove the source of the data

Implied or derived properties of data must often be calculated or inferred by the code itself. Errors in deriving properties may be considered a contributing factor to improper input validation.

Note that "input validation" has very different meanings to different people, or within different classification schemes. Caution must be used when referencing this CWE entry or mapping to it. For example, some weaknesses might involve inadvertently giving control to an attacker over an input when they should not be able to provide an input at all, but sometimes this is referred to as input validation.

Finally, it is important to emphasize that the distinctions between input validation and output escaping are often blurred, and developers must be careful to understand the difference, including how input validation is not always sufficient to prevent vulnerabilities, especially when less stringent data types must be supported, such as free-form text. Consider a SQL injection scenario in which a person's last name is inserted into a query. The name "O'Reilly" would likely pass the validation step since it is a common last name in the English language. However, this valid name cannot be directly inserted into the database because it contains the "'" apostrophe character, which would need to be escaped or otherwise transformed. In this case, removing the apostrophe might reduce the risk of SQL injection, but it would produce incorrect behavior because the wrong name would be recorded.

Example 2

This example asks the user for a height and width of an m X n game board with a maximum dimension of 100 squares.

Example Language: **C** *(bad code)*

```
...
#define MAX_DIM 100
...
/* board dimensions */

int m,n, error;
board_square_t *board;
printf("Please specify the board height: \n");
error = scanf("%d", &m);
if ( EOF == error ){
  die("No integer passed: Die evil hacker!\n");
}
printf("Please specify the board width: \n");
error = scanf("%d", &n);
if ( EOF == error ){
  die("No integer passed: Die evil hacker!\n");
}
if ( m > MAX_DIM || n > MAX_DIM ) {
  die("Value too large: Die evil hacker!\n");
}
board = (board_square_t*) malloc( m * n * sizeof(board_square_t));
...
```

Example 2

This example asks the user for a height and width of an m X n game board with a maximum dimension of 100 squares.

Example Language: **C** *(bad code)*

```
...
#define MAX_DIM 100
...
/* board dimensions */

int m,n, error;
board_square_t *board;
printf("Please specify the board height: \n");
error = scanf("%d", &m);
if ( EOF == error ){
  die("No integer passed: Die evil hacker!\n");
}
printf("Please specify the board width: \n");
error = scanf("%d", &n);
if ( EOF == error ){
  die("No integer passed: Die evil hacker!\n");
}
if ( m > MAX_DIM || n > MAX_DIM ) {
  die("Value too large: Die evil hacker!\n");
}
board = (board_square_t*) malloc( m * n * sizeof(board_square_t));
...
```

While this code checks to make sure the user cannot specify large, positive integers and consume too much memory, it does not check for negative values supplied by the user. As a result, an attacker can perform a resource consumption (CWE-400) attack against this program by specifying two, large negative values that will not overflow, resulting in a very large memory allocation (CWE-789) and possibly a system crash. Alternatively, an attacker can provide very large negative values which will cause an integer overflow (CWE-190) and unexpected behavior will follow depending on how the values are treated in the remainder of the program.

Out-of-bounds Read

CWE-125: Out-of-bounds Read

Weakness ID: 125
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The software reads data past the end, or before the beginning, of the intended buffer.

Extended Description

Typically, this can allow attackers to read sensitive information from other memory locations or cause a crash. A crash can occur when the code reads a variable amount of data and assumes that a sentinel exists to stop the read operation, such as a NUL in a string. The expected sentinel might not be located in the out-of-bounds memory, causing excessive data to be read, leading to a segmentation fault or a buffer overflow. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent read operation then produces undefined or unexpected results.

Example 1

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method

Example Language: **C** *(bad code)*

```
int getValueFromArray(int *array, int len, int index) {

  int value;

  // check that the array index is less than the maximum

  // length of the array
  if (index < len) {

    // get the value at the specified index of the array
    value = array[index];
  }
  // if array index is invalid then output error message

  // and return value indicating error
  else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }

  return value;
}
```

Example 1

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method

Example Language: **C** *(bad code)*

```
int getValueFromArray(int *array, int len, int index) {

  int value;

  // check that the array index is less than the maximum

  // length of the array
  if (index < len) {

    // get the value at the specified index of the array
    value = array[index];
  }
  // if array index is invalid then output error message

  // and return value indicating error
  else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }

  return value;
}
```

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

Example Language: **C** *(good code)*

```
...

// check that the array index is within the correct

// range of values for the array
if (index >= 0 && index < len) {

...
```

Use After Read

CWE-416: Use After Free

Weakness ID: 416
Abstraction: Variant
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.

Extended Description

The use of previously-freed memory can have any number of adverse consequences, ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the flaw. The simplest way data corruption may occur involves the system's reuse of the freed memory. Use-after-free errors have two common and sometimes overlapping causes:

- Error conditions and other exceptional circumstances.
- Confusion over which part of the program is responsible for freeing the memory.

In this scenario, the memory in question is allocated to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed, it corrupts the validly used memory; this induces undefined behavior in the process.

If the newly allocated data happens to hold a class, in C++ for example, various function pointers may be scattered within the heap data. If one of these function pointers is overwritten with an address to valid shellcode, execution of arbitrary code can be achieved.

Example Language: **C** *(bad code)*

```
char* ptr = (char*)malloc (SIZE);
if (err) {
  abrt = 1;
  free(ptr);
}
...
if (abrt) {
  logError("operation aborted before commit", ptr);
}
```

Example 2

The following code illustrates a use after free error:

Example Language: **C** *(bad code)*

```
char* ptr = (char*)malloc (SIZE);
if (err) {
   abrt = 1;
   free(ptr);
}
...
if (abrt) {
   logError("operation aborted before commit", ptr);
}
```

When an error occurs, the pointer is immediately freed. However, this pointer is later incorrectly used in the logError function.

NULL Pointer Dereference

CWE-476: NULL Pointer Dereference

Weakness ID: 476
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

Extended Description

NULL pointer dereference issues can occur through a number of flaws, including race conditions, and simple programming omissions.

Example 2

This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.

Example Language: **C** *(bad code)*

```
void host_lookup(char *user_supplied_addr){
  struct hostent *hp;
  in_addr_t *addr;
  char hostname[64];
  in_addr_t inet_addr(const char *cp);

  /*routine that ensures user_supplied_addr is in the right format for conversion */

  validate_addr_form(user_supplied_addr);
  addr = inet_addr(user_supplied_addr);
  hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
  strcpy(hostname, hp->h_name);
}
```

Example 2

This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.

Example Language: **C** *(bad code)*

```
void host_lookup(char *user_supplied_addr){
  struct hostent *hp;
  in_addr_t *addr;
  char hostname[64];
  in_addr_t inet_addr(const char *cp);

  /*routine that ensures user_supplied_addr is in the right format for conversion */

  validate_addr_form(user_supplied_addr);
  addr = inet_addr(user_supplied_addr);
  hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
  strcpy(hostname, hp->h_name);
}
```

If an attacker provides an address that appears to be well-formed, but the address does not resolve to a hostname, then the call to gethostbyaddr() will return NULL. Since the code does not check the return value from gethostbyaddr (CWE-252), a NULL pointer dereference (CWE-476) would then occur in the call to strcpy().

Note that this code is also vulnerable to a buffer overflow (CWE-119).

Integer Overflow or Wraparound

CWE-190: Integer Overflow or Wraparound

Weakness ID: 190
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

Extended Description

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

Example 1

The following image processing code allocates a table for images.

Example Language: **C** *(bad code)*

```
img_t table_ptr; /*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

Example 1

The following image processing code allocates a table for images.

Example Language: **C** *(bad code)*

```
img_t table_ptr; /*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

This code intends to allocate a table of size num_imgs, however as num_imgs grows large, the calculation determining the size of the list will eventually overflow (CWE-190). This will result in a very small list to be allocated instead. If the subsequent code operates on the list as if it were num_imgs long, it may result in many types of out-of-bounds problems (CWE-119).

Example Language: **C** *(bad code)*

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

Example 2

The following code excerpt from OpenSSH 3.3 demonstrates a classic case of integer overflow:

Example Language: **C** *(bad code)*

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

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

Example Language: **C** *(bad code)*

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
  bytesRec += getFromInput(buf+bytesRec);
}
```

Example 3

Integer overflows can be complicated and difficult to detect. The following example is an attempt to show how an integer overflow may lead to undefined looping behavior:

Example Language: **C** *(bad code)*

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
  bytesRec += getFromInput(buf+bytesRec);
}
```

In the above case, it is entirely possible that bytesRec may overflow, continuously creating a lower number than MAXGET and also overwriting the first MAXGET-1 bytes of buf.

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

Weakness ID: 119
Abstraction: Class
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

Extended Description

Certain languages allow direct addressing of memory locations and do not automatically ensure that these locations are valid for the memory buffer that is being referenced. This can cause read or write operations to be performed on memory locations that may be associated with other variables, data structures, or internal program data.

As a result, an attacker may be able to execute arbitrary code, alter the intended control flow, read sensitive information, or cause the system to crash.

Example 3

The following example asks a user for an offset into an array to select an item.

Example Language: **C** *(bad code)*

```
int main (int argc, char **argv) {
   char *items[] = {"boat", "car", "truck", "train"};
   int index = GetUntrustedOffset();
   printf("You selected %s\n", items[index-1]);
}
```

Example 3

The following example asks a user for an offset into an array to select an item.

Example Language: **C** *(bad code)*

```
int main (int argc, char **argv) {
   char *items[] = {"boat", "car", "truck", "train"};
   int index = GetUntrustedOffset();
   printf("You selected %s\n", items[index-1]);
}
```

The programmer allows the user to specify which element in the list to select, however an attacker can provide an out-of-bounds offset, resulting in a buffer over-read (CWE-126).

Uncontrolled Resource Consumption

CWE-400: Uncontrolled Resource Consumption

Weakness ID: 400
Abstraction: Class
Structure: Simple

View customized information: Conceptual | Operational | Mapping-Friendly | **Complete**

Description

The software does not properly control the allocation and maintenance of a limited resource, thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.

Extended Description

Limited resources include memory, file system storage, database connection pool entries, and CPU. If an attacker can trigger the allocation of these limited resources, but the number or size of the resources is not controlled, then the attacker could cause a denial of service that consumes all available resources. This would prevent valid users from accessing the software, and it could potentially have an impact on the surrounding environment. For example, a memory exhaustion attack against an application could slow down the application as well as its host operating system.

There are at least three distinct scenarios which can commonly lead to resource exhaustion:

- Lack of throttling for the number of allocated resources
- Losing all references to a resource before reaching the shutdown stage
- Not closing/returning a resource after processing

Resource exhaustion problems are often result due to an incorrect implementation of the following situations:

- Error conditions and other exceptional circumstances.
- Confusion over which part of the program is responsible for releasing the resource.

Example 2

This code allocates a socket and forks each time it receives a new connection.

Example Language: **C** *(bad code)*

```
sock=socket(AF_INET, SOCK_STREAM, 0);
while (1) {
  newsock=accept(sock, ...);
  printf("A connection has been accepted\n");
  pid = fork();
}
```

Example 2

This code allocates a socket and forks each time it receives a new connection.

Example Language: **C** *(bad code)*

```
sock=socket(AF_INET, SOCK_STREAM, 0);
while (1) {
  newsock=accept(sock, ...);
  printf("A connection has been accepted\n");
  pid = fork();
}
```

The program does not track how many connections have been made, and it does not limit the number of connections. Because forking is a relatively expensive operation, an attacker would be able to cause the system to run out of CPU, processes, or memory by making a large number of connections. Alternatively, an attacker could consume all available connections, preventing others from accessing the system remotely.

Example 4

In the following example, the processMessage method receives a two dimensional character array containing the message to be processed. The two-dimensional character array contains the length of the message in the first character array and the message body in the second character array. The getMessageLength method retrieves the integer value of the length from the first character array. After validating that the message length is greater than zero, the body character array pointer points to the start of the second character array of the two-dimensional character array and memory is allocated for the new body character array.

Example Language: **C** *(bad code)*

```
/* process message accepts a two-dimensional character array of the form [length]
[body] containing the message to be processed */
int processMessage(char **message)
{
  char *body;

  int length = getMessageLength(message[0]);

  if (length > 0) {
    body = &message[1][0];
    processMessageBody(body);
    return(SUCCESS);
  }
  else {
    printf("Unable to process message; invalid message length");
    return(FAIL);
  }
}
```

Example 4

In the following example, the processMessage method receives a two dimensional character array containing the message to be processed. The two-dimensional character array contains the length of the message in the first character array and the message body in the second character array. The getMessageLength method retrieves the integer value of the length from the first character array. After validating that the message length is greater than zero, the body character array pointer points to the start of the second character array of the two-dimensional character array and memory is allocated for the new body character array.

Example Language: **C** *(bad code)*

```
/* process message accepts a two-dimensional character array of the form [length]
[body] containing the message to be processed */
int processMessage(char **message)
{
  char *body;

  int length = getMessageLength(message[0]);

  if (length > 0) {
    body = &message[1][0];
    processMessageBody(body);
    return(SUCCESS);
  }
  else {
    printf("Unable to process message; invalid message length");
    return(FAIL);
  }
}
```

This example creates a situation where the length of the body character array can be very large and will consume excessive memory, exhausting system resources. This can be avoided by restricting the length of the second character array with a maximum length check

Also, consider changing the type from 'int' to 'unsigned int', so that you are always guaranteed that the number is positive. This might not be possible if the protocol specifically requires allowing negative values, or if you cannot control the return value from getMessageLength(), but it could simplify the check to ensure the input is positive, and eliminate other errors such as signed-to-unsigned conversion errors (CWE-195) that may occur elsewhere in the code.

Example Language: **C** *(good code)*

```
unsigned int length = getMessageLength(message[0]);
if ((length > 0) && (length < MAX_LENGTH)) {...}
```