Secure Coding in C and C++

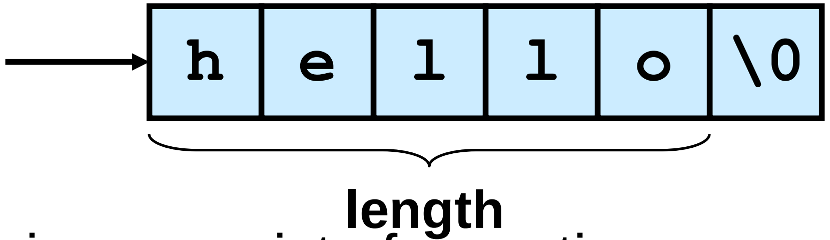
Chapter 2:-*Strings*

Strings

- In secure programming, strings from external sources like command line arguments, environment variables, console inputs, text files, and network connections are of special concern because they can introduce vulnerabilities if not handled properly.
- Comprise most of the data exchanged between an end user and a software system
- Software vulnerabilities and exploits are caused by weaknesses in string
  - representation
  - management
  - manipulation

String Data Type: C-Style Strings

- Strings are a fundamental concept in software engineering, but they are not a built-in type in C or C++.



- C-style strings consist of a contiguous sequence of characters terminated by and including the first null character.
  - A pointer to a string points to its initial character.
  - String length is the number of bytes preceding the null character
  - The string value is the sequence of the values of the contained characters, in order.
  - The number of bytes required to store a string is the number of characters plus one (x the size of each character)

C-Style Strings

- **Because strings are not a built-in type, programmers must manage memory allocation and deallocation manually, which can lead to issues like buffer overflows and memory leaks if not handled carefully.**

- **The C standard library provides a limited set of functions for string manipulation (like strlen(), strcpy(), strcat(), etc.), but these functions do not provide safety features like automatic bounds checking.**

- **The sizeof operator in C provides the size of a data type or a variable in bytes. When used with strings, it behaves differently depending on whether you’re dealing with C-style strings**

Using sizeof with C-style Strings

```c
#include <stdio.h>
#include <string.h>
int main() {
    char myString[] = "Hello, World!";
        // Size of the array (including null terminator)
    size_t arraySize = sizeof(myString);
        // Size of the string length (without null terminator)
    size_t stringLength = strlen(myString);
    printf("Size of the array: %zu\n", arraySize);
    printf("Length of the string: %zu\n", stringLength);
        // Demonstrating a potential pitfall
    char buffer[10];
        // Unsafe copy that can cause buffer overflow
    // Be cautious: `sizeof(buffer)` gives the size of the buffer, not the length of myString
    strncpy(buffer, myString, sizeof(buffer)); // This can still lead to issues if not handled properly
    buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination
    printf("Buffer content: %s\n", buffer);
    return 0;
}
```

UTF-8

- UTF-8 is a variable-width character encoding that can represent every character in the Unicode character set. It uses 1 to 4 bytes per character.

- UTF-8 is widely used for web content and databases because it supports multiple languages and special characters.

```
#include <stdio.h>

int main() {
    // UTF-8 encoded string
    const char* utf8String = "Hello, 世界!"; // "Hello, World!" in Chinese

    // Print the UTF-8 string
    printf("%s\n", utf8String);

    // Get the length of the string
    // Note: strlen counts bytes, not characters
    size_t length = strlen(utf8String);
    printf("Byte length: %zu\n", length);
    return 0;
}
```

Wide Strings

- **Wide strings are used to represent Unicode characters, typically using wchar_t in C and C++. Each wide character usually occupies 2 or 4 bytes, depending on the platform.**
- **Wide strings are helpful when you need to handle characters outside the ASCII range, such as characters from different languages.**
- **To process the characters of a large character set, a program may represent each character as a wide character.**
- **It is used for handling character sets require more than 1-byte space is provided by standard char type**

```
wchar_t wideStr[] = L"Hello, World!";
```

String Literals

- **In C, string literals are sequences of characters enclosed in double quotes. They are used to represent text data in a program.**

Characteristics of String Literals

- **Null-Termination: String literals in C are automatically terminated by a null character ('\0'). This means that the actual size of the string is one character longer than the number of visible characters.**

For example, the string literal "Hello" actually occupies 6 bytes in memory: 5 for the characters and 1 for the null terminator.

- **Type: String literals are treated as arrays of char. When you create a string literal, it is stored as an array of characters in read-only memory.**

The type of a string literal is char[], but when used, it decays to a pointer to its first element (const char*).

- **Example const char* greeting = "Hello, World!";**

C++ Strings

- The standardization of C++ has promoted
  - the standard template class **`std::basic_string`**
  - and its **`char`** instantiation **`std::string`**
  - The **`basic_string`** class is less prone to security vulnerabilities than C-style strings.
- C-style strings are still a common data type in C++ programs
- Impossible to avoid having multiple string types in a C++ program except in rare circumstances
  - there are no string literals
  - no interaction with the existing libraries that accept C-style strings OR only C-style strings are used

C++ Strings

```
#include <iostream>

int main() {
    // Declare a C-style string (character array)
    char name[50];  // Array to hold up to 49 characters + null terminator

    // Ask for the user's name
    std::cout << "Enter your name: ";
    std::cin.getline(name, sizeof(name));  // Read input

    // Output a greeting message
    std::cout << "Hello, " << name << "!" << std::endl;

    return 0;
}
```

Character Strings

- **Char**: Typically 1 byte (8 bits), which can store values between -128 to 127 for signed char and 0 to 255 for unsigned char (this range may vary on some systems).
- **Unsigned Char**: An unsigned version of char. All values are non-negative, which makes it useful when working with raw bytes. Typically 1 byte (8 bits), storing values between 0 and 255.
- **Signed Char**: Stores both positive and negative values.

Typically 1 byte (8 bits), storing values between -128 and 127.

Character Strings

- **wchar_t :** Designed to store wide characters, primarily used for wide-character sets such as Unicode. Varies between platforms. On Windows, it's typically 2 bytes (16 bits, UTF-16 encoding); on Linux and Unix systems, it's usually 4 bytes (32 bits, UTF-32 encoding).

Sizing Strings

- Sizing strings correctly is essential in preventing buffer overflows and other run time errors. Incorrect string sizes can lead to buffer overflow.

```
#include <iostream>
#include <string>
int main() {
 std::string name;  // No need to specify size
std::cout << "Enter your name: ";
std::getline(std::cin, name);  // Read input
    std::cout << "Hello, " << name << "!" << std::endl;
return 0; }
```

Common String Manipulation Errors

- Programming with C-style strings, in C or C++, is error prone.
- Common errors include
    - Improperly bounded string copies
    - Off-by-one Errors
    - Null-termination errors
    - String Truncation

Unbounded String Copies

- Improperly bounded string copies occur when data is copied from a source to a fixed-length character array.

```
char source[] = "This is a long string.";
char destination[10]; // Too small for the source string
// Incorrect: This can cause buffer overflow
std::strcpy(destination, source); // Undefined behavior
```

```
#include <iostream>
#include <cstring>

int main() {
    char source[100];          // Buffer for input (up to 99 characters + null terminator)
    char destination[10];     // Small buffer for copying (only 9 characters + null terminator)

    // Take input from stdin
    std::cout << "Enter a long string (up to 99 characters): ";
    std::cin.getline(source, sizeof(source));  // Read input into source

    // Improperly bounded copy - this can cause buffer overflow
    std::strcpy(destination, source);  // Unsafe copy

    // Output the copied string
    std::cout << "Copied string: " << destination << std::endl;

    return 0;
}
```

Copying and Concatenation

- It is easy to make errors when
  - copying and concatenating strings because
    - standard functions do not know the size of the destination buffer

```
int main(int argc, char *argv[]) {
  char name[2048];
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
    ...
}
```

Simple Solution

- Test the length of the input using **`strlen()`** and dynamically allocate the memory

```
int main(int argc, char *argv[]) {
  char *buff = (char *)malloc(strlen(argv[1])+1);
  if (buff != NULL) {
    strcpy(buff, argv[1]);
    printf("argv[1] = %s.\n", buff);
  }
  else {
    /* Couldn't get the memory - recover */
  }
  return 0;
}
```

C++ Unbounded Copy

- Inputting more than 11 characters into following the C++ program results in an out-of-bounds write:

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
char buf[12];
cout << "Enter a string (up to 11 characters): ";
 cin >> buf;
 cout << "echo: " << buf << endl;
   return 0; }
```

Simple Solution

```

 cin.width(12);
 cin >> buf;
 cout << "echo: " << buf << endl;
}
```

Off-by-One Errors

```
#include <iostream>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int sum = 0;

    // Incorrect loop condition: iterates one time too many
    for (int i = 0; i <= 5; ++i) { // This should be i < 5
        sum += arr[i]; // Accessing arr[5] is out of bounds
    }
    std::cout << "Sum: " << sum << std::endl;
    return 0;
}
```

Null-Termination Errors

Null termination errors refer to issues that arise when dealing with C-style strings (character arrays) in languages like C and C++. These strings must end with a null character ('\0') to signify the end of the string.

```
#include <iostream>
#include <cstring> // For strlen and strcpy

int main() {
    char buf[10] = {'H', 'e', 'l', 'l', 'o'}; // Missing null terminator
    std::cout << "Length: " << strlen(buf) << std::endl; // Undefined behavior
    return 0;
}
```

String Truncation

- **Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities**
  - `strncpy()` instead of `strcpy()`
  - `fgets()` instead of `gets()`
  - `snprintf()` instead of `sprintf()`
- **Strings that exceed the specified limits are truncated**
- **Truncation results in a loss of data, and in some cases, to software vulnerabilities**

String Truncation

```
#include <iostream>
#include <cstring>

int main() {
    char source[] = "Hello, World!";
    char destination[10]; // Smaller buffer

    strcpy(destination, source); // Unsafe copy, truncation occurs
    std::cout << "Destination: " << destination << std::endl; // Undefined behavior
    return 0;
}
```

String Vulnerabilities and Exploits

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
bool IsPasswordOK(void) {
char Password[12]; // Buffer size of 12, can hold 11 characters + null terminator
    gets(Password); // Unsafe function, can lead to buffer overflow
return 0 == strcmp("Password", "goodpass");}
int main(void) {
 bool PwStatus;
   puts("Enter password");
   PwStatus = IsPasswordOK();
 if (PwStatus == false) {
puts("Access Denied"); exit(-1); } }
```

String Vulnerabilities and Exploits—Security Flaw

Use of gets():

- The gets() function does not perform bounds checking, allowing a user to input more characters than the Password buffer can hold, which can lead to a buffer overflow. This vulnerability can be exploited to overwrite adjacent memory, potentially allowing an attacker to execute arbitrary code.

Incorrect Password Comparison:

- The use of strcmp("Password", "goodpass") incorrectly checks the password against the wrong string. This means the authentication will always fail regardless of the user's input.
- While this flaw might not pose a direct security risk, it leads to poor user experience and undermines the application’s intended functionality.

Buffer Overflows

Buffer Overflows occur when data is written outside of the boundaries of the memory allocated to a particular data structure. C and C++ are susceptible to buffer overflows because these languages

- Define Strings as null terminated arrays of characters
- Do not perform implicit bounds checking
- Provide standard library calls for strings that do not enforce bounds checking
- This can cause unpredictable behavior, including crashes, data corruption, and security vulnerabilities such as arbitrary code execution.
- When a program allocates a fixed amount of memory for a buffer (e.g., an array), it expects that data written to this buffer will not exceed its size. If a program does not properly check the length of the input before writing to the buffer, an attacker can exploit this oversight by providing input larger than the buffer can hold.

Process Memory Organization

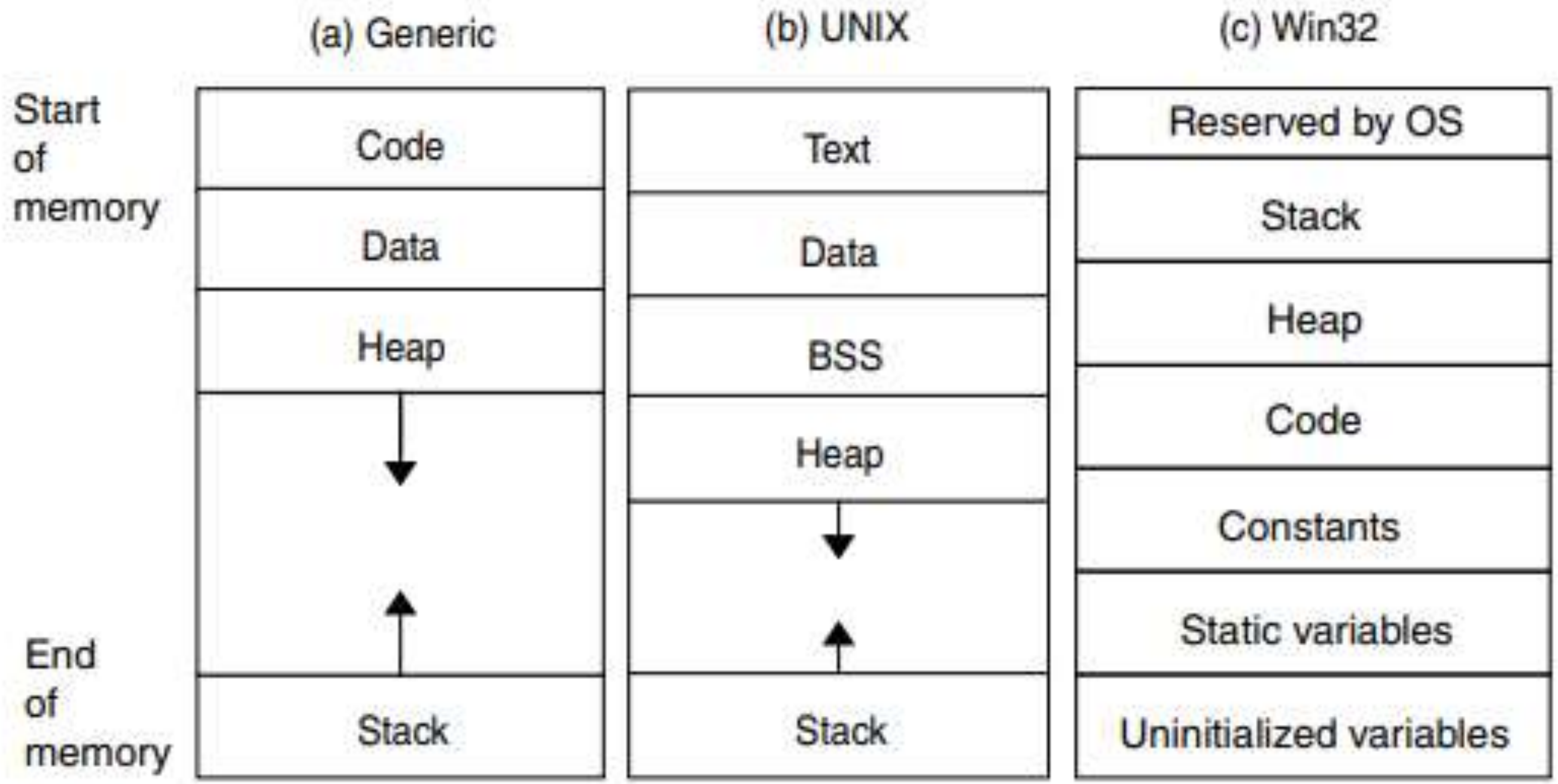


Figure 2.5 Process memory organization

Stack Management

- **The stack supports program execution by maintaining automatic process state data.**

- **A stack is well suited for maintain the sequence of return addresses.**

- **When subroutine returns, the return address is popped from the stack, and program jumps to the specified location.**

- **The stack is used to store the arguments to the routine as well as local variables.**

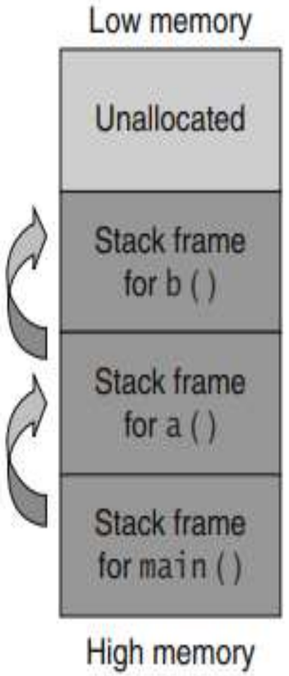


Figure 2.7 Calling a subroutine

Stack Management

The compiler and operating system handle most of the stack management automatically. This includes:

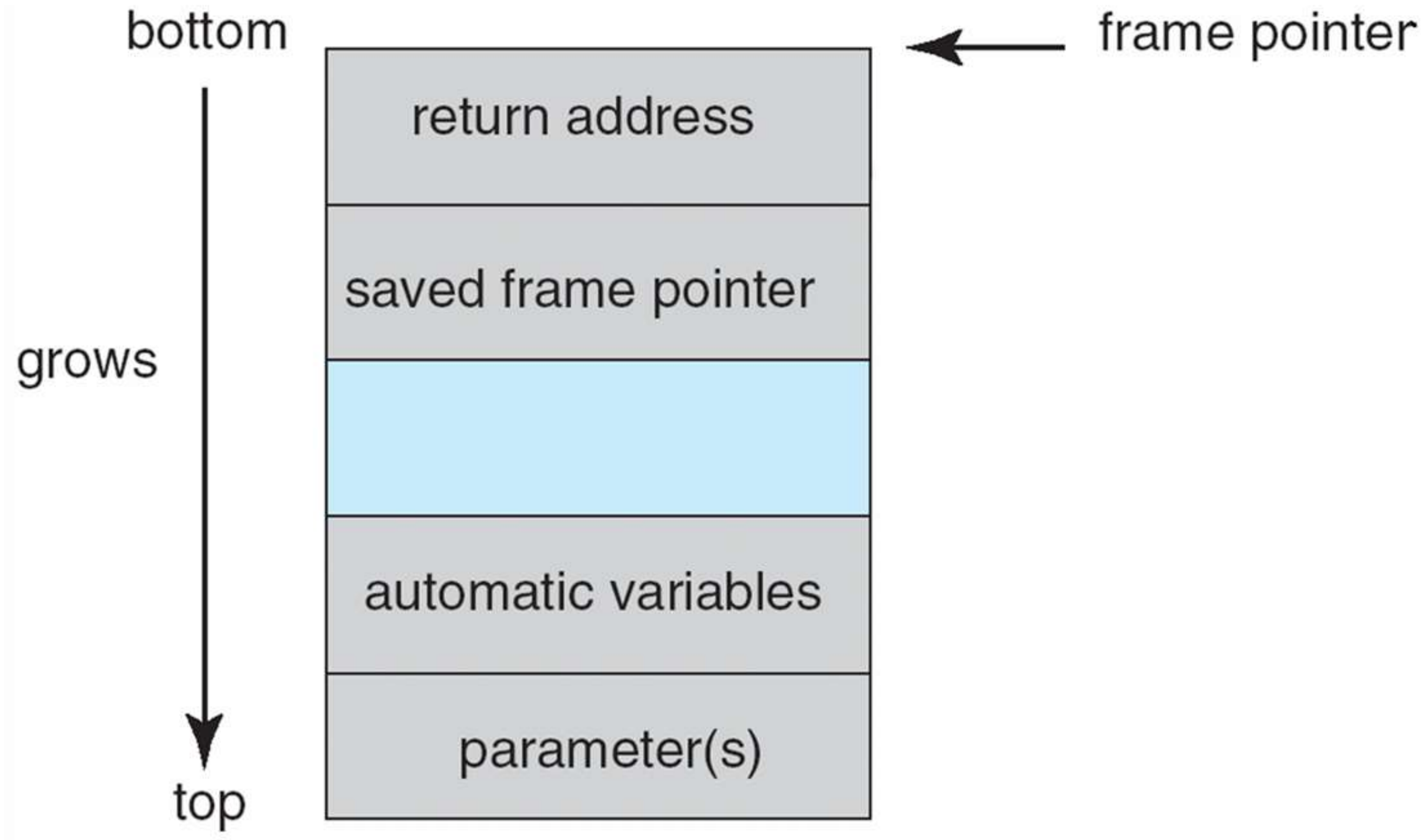
Function Calls: Every time a function is called, a new stack frame is created. The stack frame contains:

Function parameters.
Local variables.
The return address (the address where the control needs to return after the function finishes).
Saved registers (depending on the system's calling conventions).

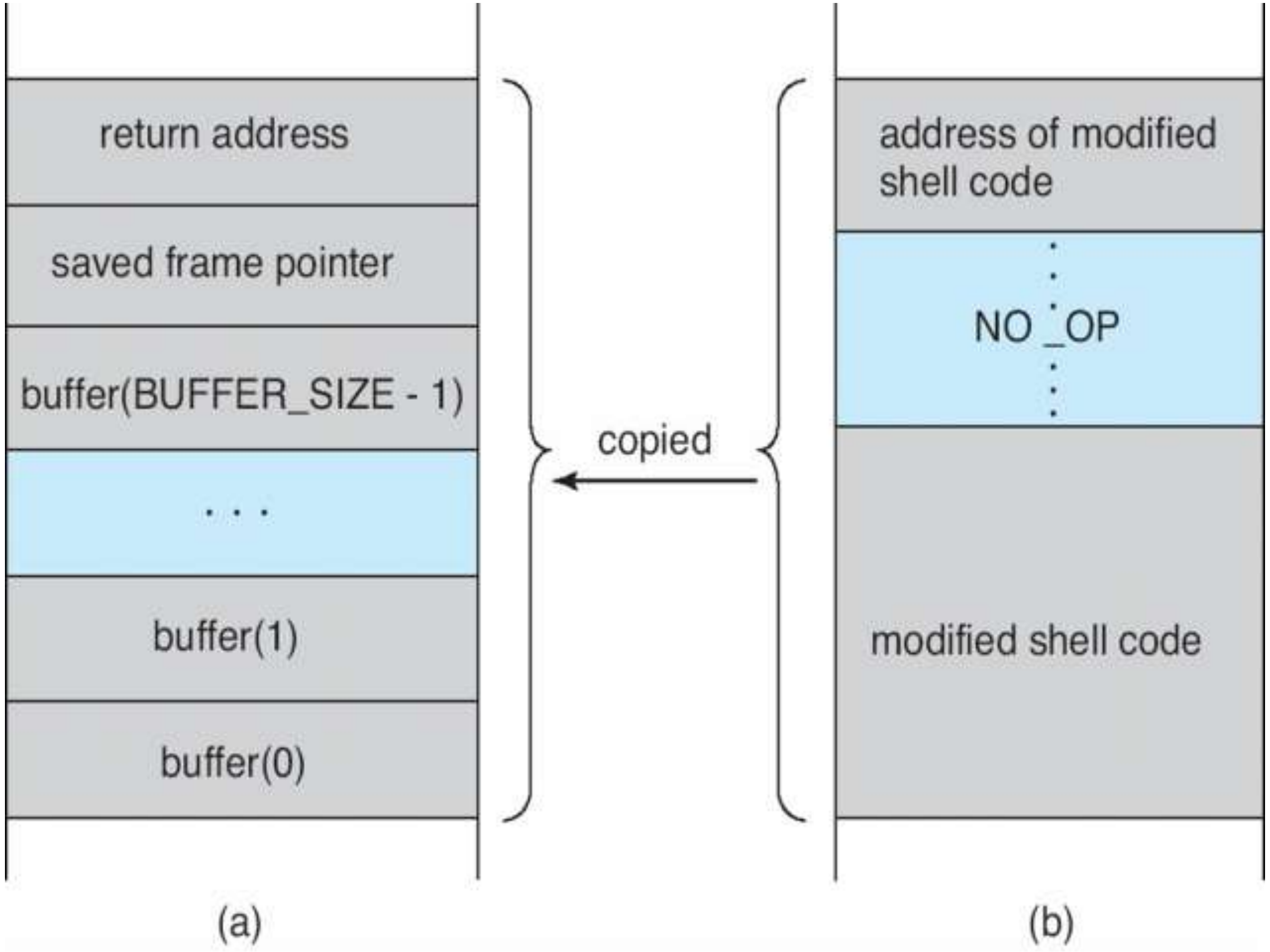
Local Variables: Local (automatic) variables declared in a function are stored on the stack. These variables are automatically deallocated when the function returns.

Stack Frames: A new stack frame is pushed onto the stack when a function is called, and it is popped off when the function returns. This happens in LIFO (Last-In-First-Out) order.

Layout of Typical Stack Frame



Hypothetical Stack Frame



Before attack

After attack

Modified Shell Code

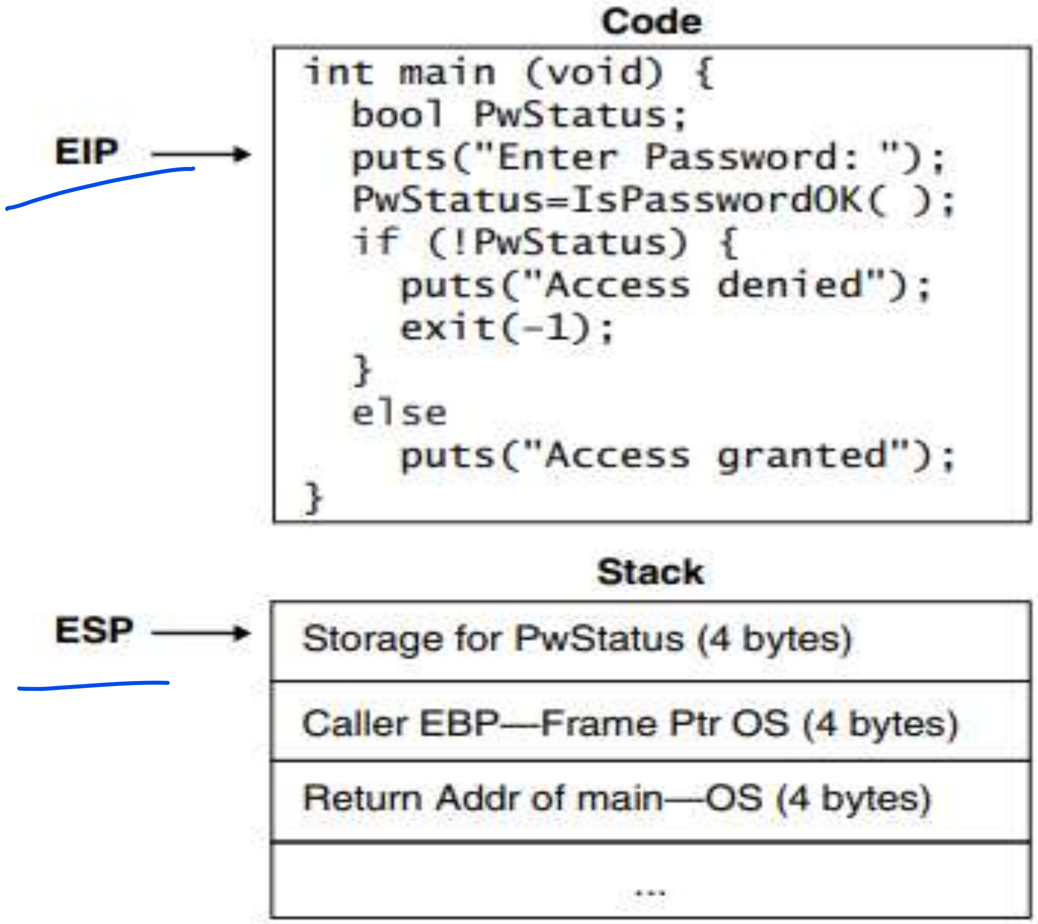
```
#include <stdio.h>
int main(int argc, char *argv[])
{
  execvp(‘‘\bin\sh’’,‘‘\bin \sh’’, NULL);
  return 0;
}
```

Stack Smashing

Stack smashing is a type of buffer overflow attack that occurs when a program writes more data to a buffer on the stack than it is allocated, overwriting adjacent memory locations, including the stack frame (return address, saved registers, etc.). This can lead to arbitrary code execution, crash the program, or cause erratic behavior. It's one of the most common vulnerabilities exploited in C and C++ programs due to their lack of built-in memory safety features.

Overwriting automatic variables result in a loss of data integrity or security breach(for example if a variable is containing a user ID or password.

Stack Smashing



```
int main (void) {
  bool PwStatus;
  puts("Enter Password: ");
  PwStatus=IsPasswordOK( );
  if (!PwStatus) {
    puts("Access denied");
    exit(-1);
  }
  else
    puts("Access granted");
}
```

Storage for PwStatus (4 bytes)

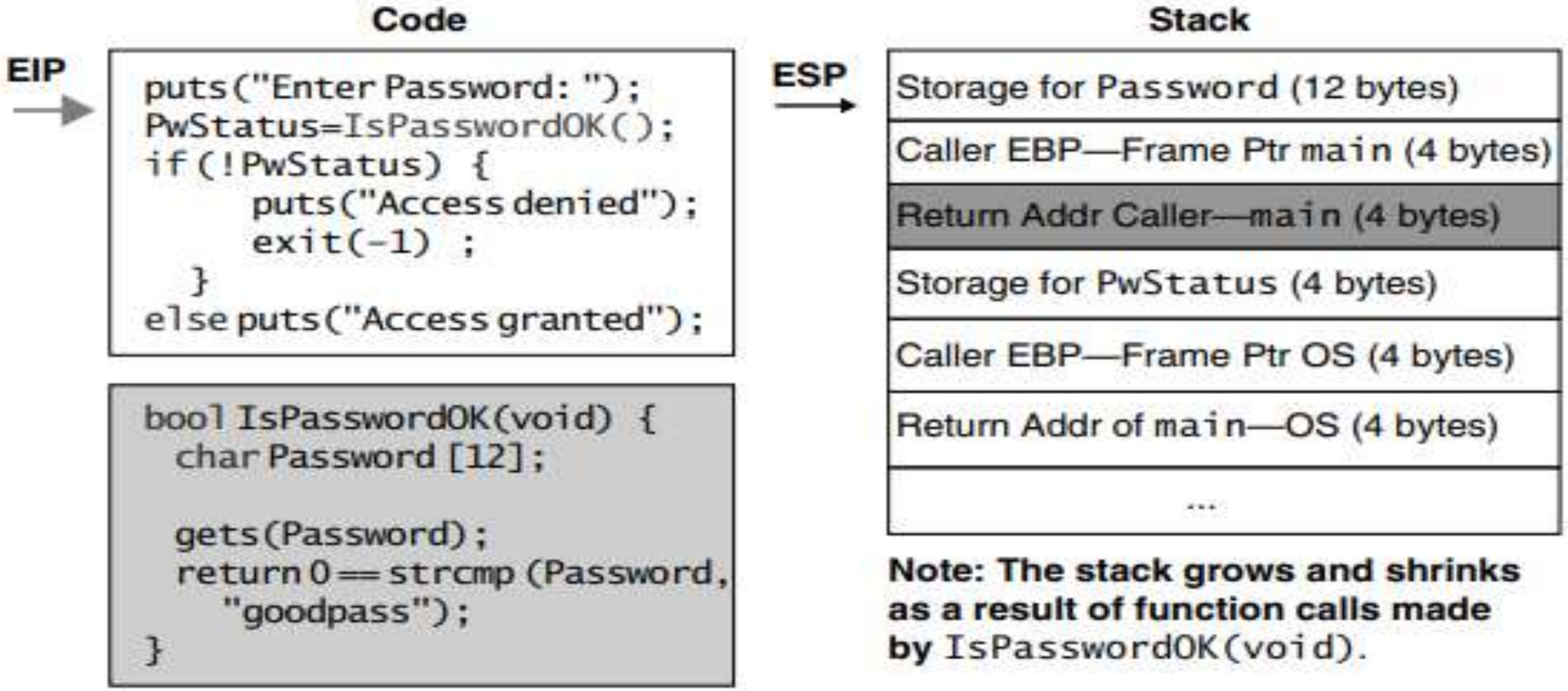
Caller EBP—Frame Ptr OS (4 bytes)

Return Addr of main—OS (4 bytes)

...

Figure 2.8 The stack before IsPasswordOK() is called

Stack Smashing



```
puts("Enter Password: ");
PwStatus=IsPasswordOK();
if(!PwStatus) {
     puts("Access denied");
     exit(-1) ;
  }
else puts("Access granted");
```

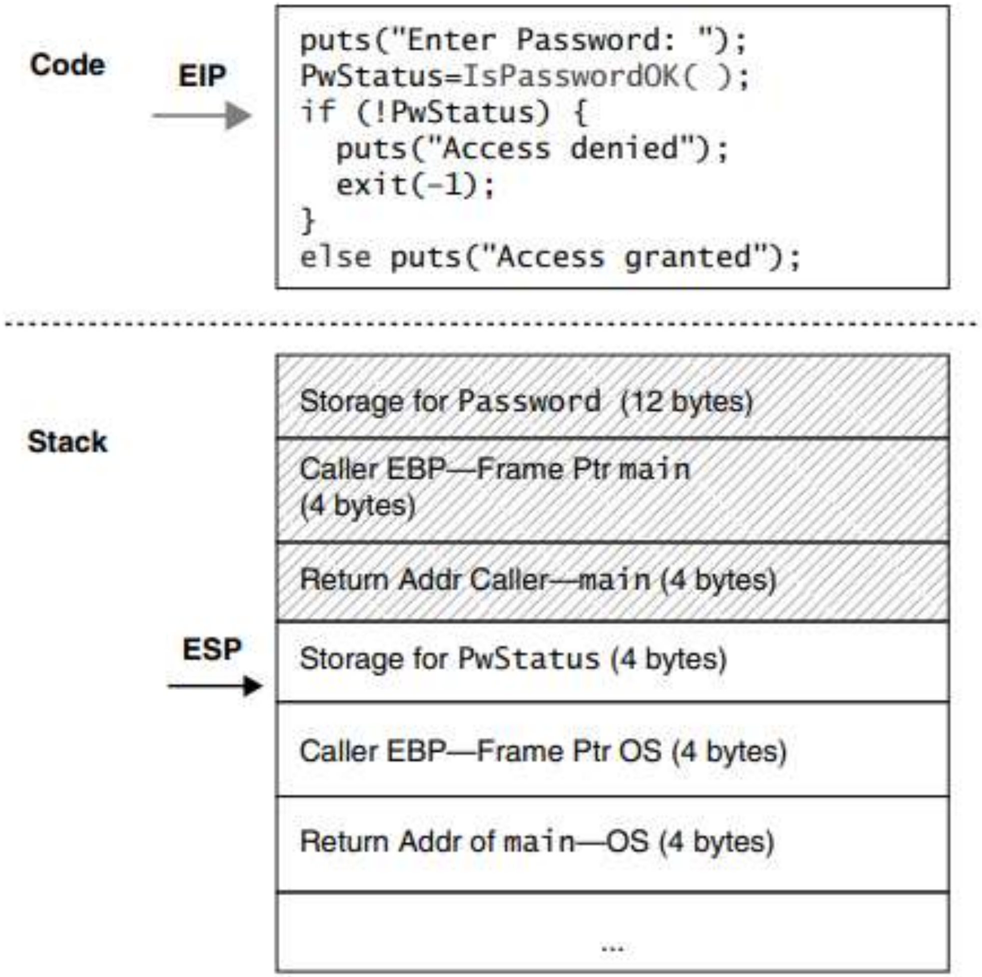
```
bool IsPasswordOK(void) {
  char Password [12];

  gets(Password);
  return 0 == strcmp (Password,
     "goodpass");
}
```

Storage for Password (12 bytes)
Caller EBP—Frame Ptr main (4 bytes)
Return Addr Caller—main (4 bytes)
Storage for PwStatus (4 bytes)
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of main—OS (4 bytes)
...

Note: The stack grows and shrinks as a result of function calls made by IsPasswordOK(void).

Figure 2.9 Information in stack while IsPasswordOK() is executed

Stack Smashing



```
puts("Enter Password: ");
PwStatus=IsPasswordOK( );
if (!PwStatus) {
  puts("Access denied");
  exit(-1);
}
else puts("Access granted");
```

Storage for Password (12 bytes)

Caller EBP—Frame Ptr main (4 bytes)

Return Addr Caller—main (4 bytes)

Storage for PwStatus (4 bytes)

Caller EBP—Frame Ptr OS (4 bytes)

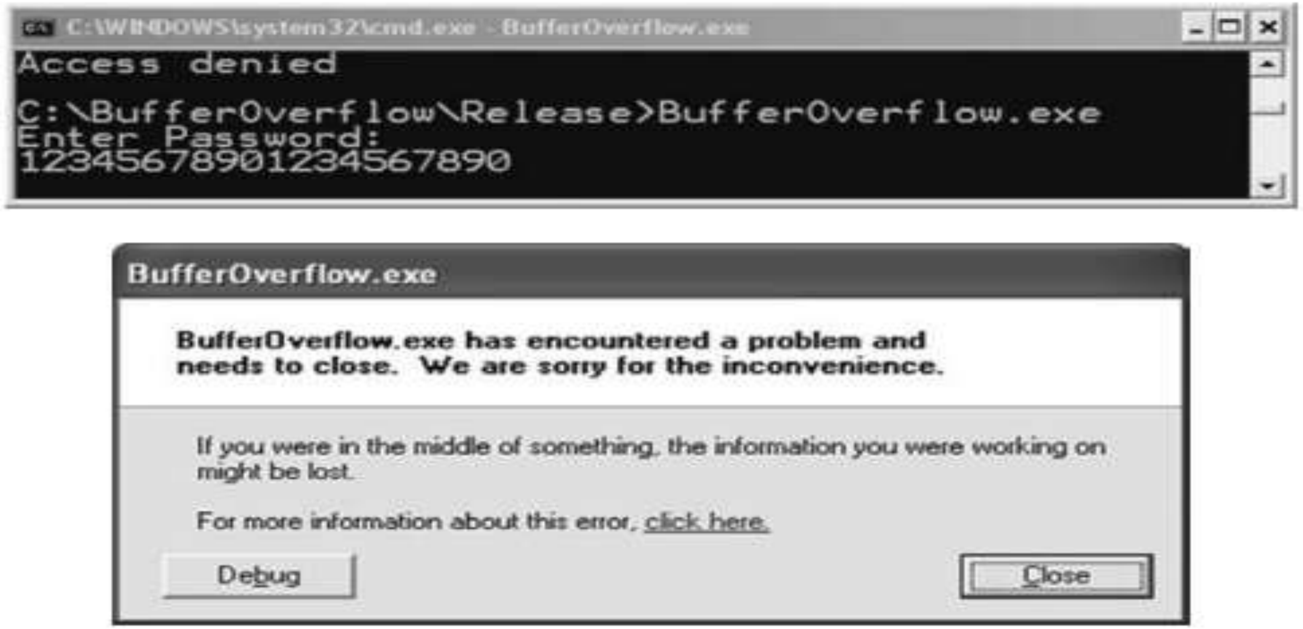
Return Addr of main—OS (4 bytes)

...

Figure 2.10 Stack restored to initial state

Security Flaw: Is PasswordOK

IsPasswordOK program has security flaw because the the Password array is improperly bounded and can hold 11 character password plus trailing null byte.



```
Access denied

C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
12345678901234567890
```

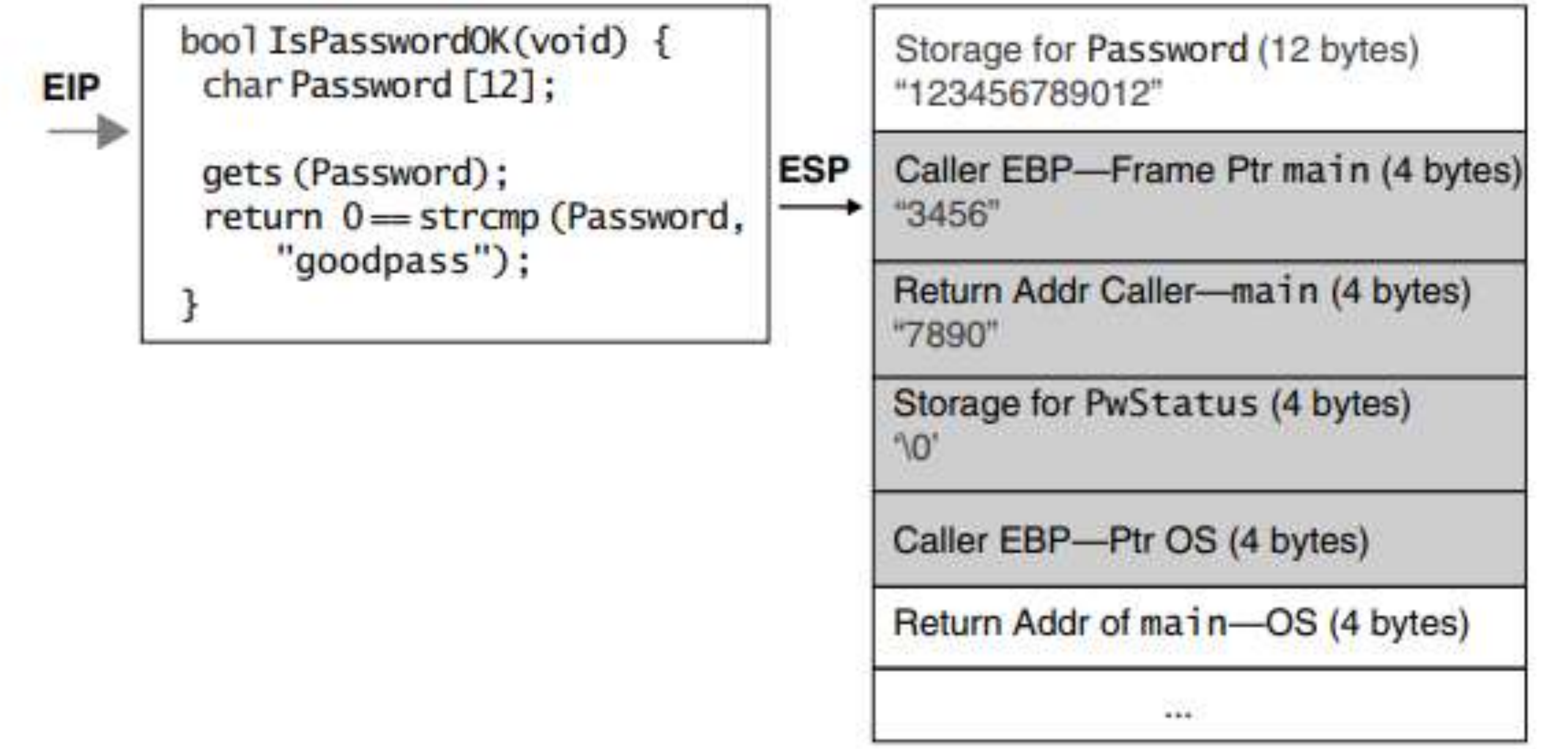
BufferOverflow.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

For more information about this error, click here.

Figure 2.11 An improperly bounded Password array crashes the program if its character limit is exceeded.

Corrupted Program Stack



```
bool IsPasswordOK(void) {
 char Password [12];

 gets (Password);
 return 0 == strcmp (Password,
     "goodpass");
}
```

Storage for Password (12 bytes)
"123456789012"

Caller EBP—Frame Ptr main (4 bytes)
"3456"

Return Addr Caller—main (4 bytes)
"7890"

Storage for PwStatus (4 bytes)
'\0'

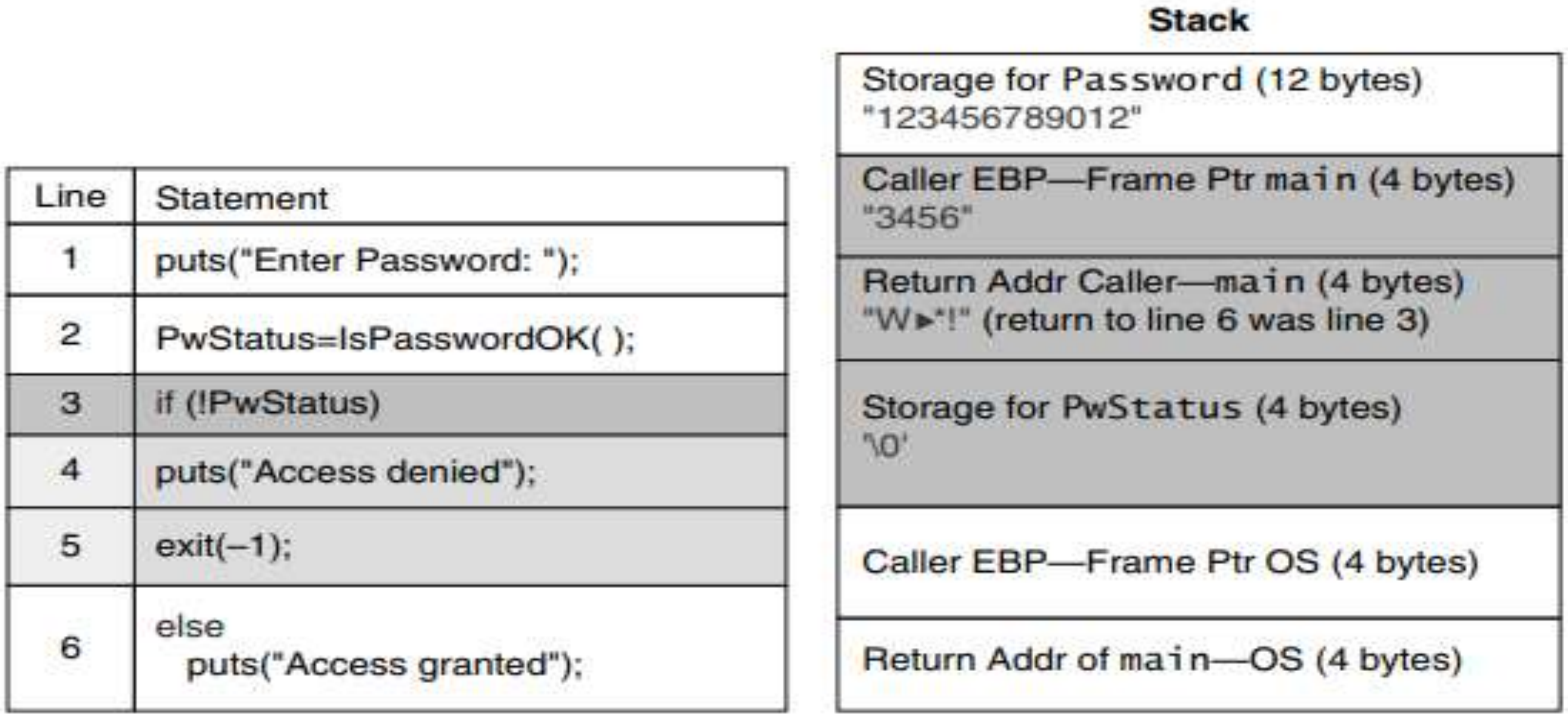
Caller EBP—Ptr OS (4 bytes)

Return Addr of main—OS (4 bytes)

...

Figure 2.12 Corrupted program stack

Corrupted Program Stack



Line	Statement
1	puts("Enter Password: ");
2	PwStatus=IsPasswordOK( );
3	if (!PwStatus)
4	puts("Access denied");
5	exit(–1);

6	else
	puts("Access granted");

Stack
Storage for Password (12 bytes) "123456789012"
Caller EBP—Frame Ptr main (4 bytes) "3456"
Return Addr Caller—main (4 bytes) "W▸*!" (return to line 6 was line 3)
Storage for PwStatus (4 bytes) '\0'
Caller EBP—Frame Ptr OS (4 bytes)
Return Addr of main—OS (4 bytes)

Figure 2.14 Program stack following buffer overflow using crafted input string

Code Injection

Code injection is a type of security vulnerability where an attacker can inject and execute arbitrary code within a program. This often happens when user inputs are not properly validated or sanitized, allowing malicious code to be inserted into the program. Code injection can lead to unauthorized actions, data breaches, or complete system takeover.

In languages like C and C++, which allow for direct memory manipulation, code injection is particularly dangerous. The attacker can often control the flow of execution by inserting code into memory regions like the stack, heap, or even through vulnerable system calls.

Code Injection

- Attacker creates a malicious argument
  - specially crafted string that contains a pointer to malicious code provided by the attacker
- When the function returns control is transferred to the malicious code
  - injected code runs with the permissions of the vulnerable program when the function returns
  - programs running with root or other elevated privileges are normally targeted

Code Injection

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

void vulnerable_function(char *input) {
    char command[256];
    snprintf(command, sizeof(command), "echo %s", input); // Unsafe input handling
    system(command);  // Executes the command
}

int main() {
    char input[100];
    printf("Enter your input: ");
    fgets(input, sizeof(input), stdin);
    vulnerable_function(input);
    return 0;
}
```

Code Injection

Solution:

```
void safe_function(char *input) { // Allow only alphabetic characters to be executed
for (int i = 0; i < strlen(input); ++i)
 {
if (!isalpha(input[i]))
{
printf("Invalid input.\n");
 return;
 }
 }
```

Mitigation Strategies for Strings

- Input validation
- strlcpy() and strlcat()
- stop using cstrings and use strings instead
- Dynamic Allocation Functions

Input Validation

- **Buffer overflows are often the result of unbounded string or memory copies.**
- **Buffer overflows can be prevented by ensuring that input data does not exceed the size of the smallest buffer in which it is stored.**

```
int myfunc(const char *arg) {
  char buff[100];
  if (strlen(arg) >= sizeof(buff)) {
    abort();
  }
}
```

Input Validation

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

void f(const char *arg) {
    char buff[100];
    if (arg == NULL) {
        fprintf(stderr, "Error: NULL input\n");
        return;
    }
    if (strlen(arg) >= sizeof(buff)) {
        fprintf(stderr, "Error: Input exceeds buffer size\n");
        return;
    }
strcpy(buff, arg);

printf("Buffer contains: %s\n", buff);
}
```

`strlcpy()` and `strlcat()`

- Copy and concatenate strings in a less error-prone manner

```
size_t strlcpy(char *dst,
 const char *src, size_t size);
 size_t strlcat(char *dst,
   const char *src, size_t size);
```

- The **`strlcpy()`** function copies the null-terminated string from **`src`** to **`dst`** (up to **`size`** characters).
- The **`strlcat()`** function appends the null-terminated string **`src`** to the end of **`dst`** (no more than **`size`** characters will be in the destination)

strlcpy() and strlcat()

```
char dest[10];
const char
*src=“hello,world!”;
strlcpy(dest,src,sizeof(dest));
 printf(“Copied string:
%s\n”,dest);
```

```
char dest[10];
const char
*src=“hello,world!”;
strlcpy(dest,src,sizeof(dest
));
 printf(“Copied string:
%s\n”,dest);
```

strcpy() and strncpy()

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

int main() {
    char src[] = "Hello, World!";
    char dest[20];  // Ensure enough space for the copied string

    // Copying src to dest
    strcpy(dest, src);

    printf("Source: %s\n", src);
    printf("Destination: %s\n", dest);

    return 0;
}
```

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

int main() {
    char src[] = "Hello, World!";
    char dest[10];  // Smaller buffer

    // Copying the first 9 characters from src to dest
    strncpy(dest, src, sizeof(dest) - 1);

    // Ensure null termination
    dest[sizeof(dest) - 1] = '\0';  // Manually add null terminator

    printf("Source: %s\n", src);
    printf("Destination: %s\n", dest);
    return 0;
```

Strlen()

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

int main() {
    const char *str = "Hello, World!";

    // Calculate the length of the string
    size_t length = strlen(str);

    // Print the length of the string
    printf("The length of the string \"%s\" is: %zu\n", str, length);

    return 0;
}
```

Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    char *buffer;
    size_t size;
printf("Enter the size of input: ");
    scanf("%zu", &size);  // Get the size from user input
    // Dynamically allocate memory based on user input
    buffer = (char *)malloc(size * sizeof(char));
    if (buffer == NULL) {
        printf("Memory allocation failed!\n");
        return 1;  // Exit if memory allocation fails
    }
    // Now safely receive input without buffer overflow
    printf("Enter a string: ");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
    free(buffer);
    return 0; }
```

C++ Strings

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello";
    str += ", World!";  // Automatically resizes to hold the new content
    std::cout << str << std::endl;
}
```

C++ Strings—Avoids Manual Memory Allocation

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello";
    std::cout << str << std::endl;  // No need to manually free memory
    return 0;
}
```

C11 Annex K Bounds-Checking Interfaces

C11 Annex K defines the strcpy_s(), strcat_s(), and strncat_s() functions as replacements for strcpy(), strcat(), strncpy() and strncat() respectively.

String Handling Functions

- gets()
- stop using cstrings and use strings instead
- memcpy and memmove
- Dynamic Allocation Functions

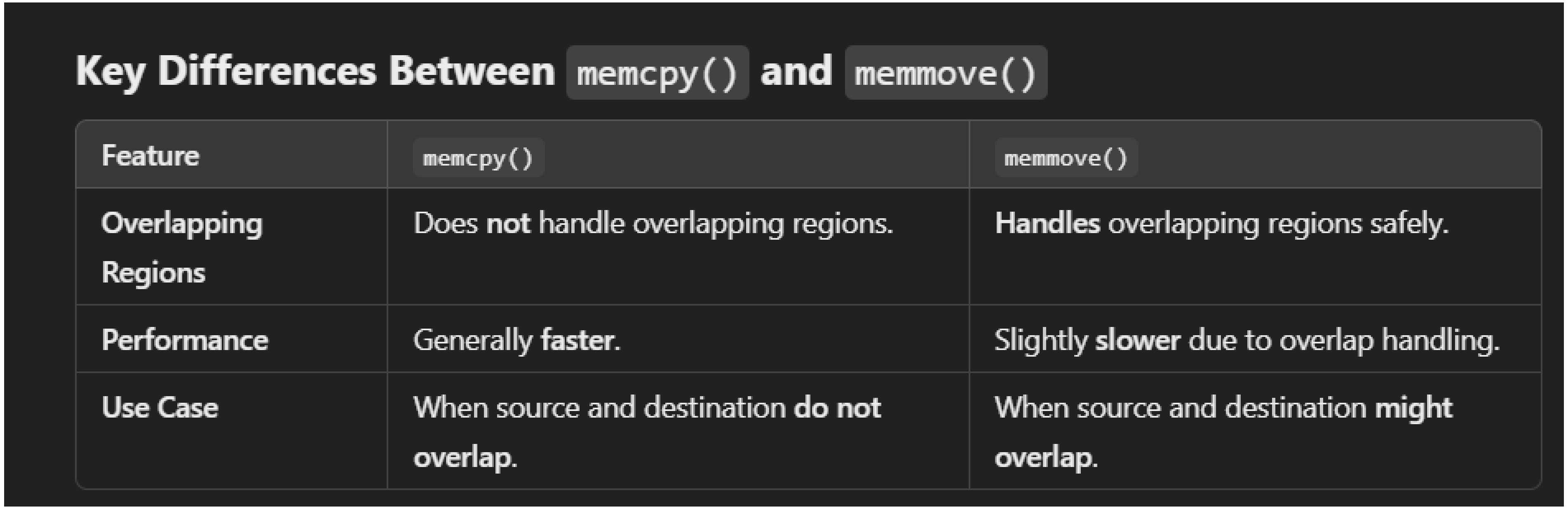
gets() is a function that reads a line of input from stdin (standard input, typically the keyboard) and stores it into a buffer (an array of characters) provided by the user. However, it doesn't check the size of the buffer, which can lead to overflow if the input exceeds the allocated space.

```
#include <stdio.h>
int main() {
char buffer[10];  // Buffer to hold input
 printf("Enter a string: ");
gets(buffer);     // Read input    printf("You entered: %s\n", buffer);
 return 0;
}
```

memcpy() and memmove()

- memcpy() and memmove() are both functions in C (from the <string.h> library) used to copy blocks of memory from one location to another. However, they differ in how they handle overlapping memory regions, and this difference can be crucial when managing memory.

memcpy() and memmove()



Key Differences Between `memcpy()` and `memmove()`

Feature	`memcpy()`	`memmove()`
Overlapping Regions	Does **not** handle overlapping regions.	**Handles** overlapping regions safely.
Performance	Generally **faster**.	Slightly **slower** due to overlap handling.
Use Case	When source and destination **do not overlap**.	When source and destination **might overlap**.

memcpy() and memmove()

```
#include <stdio.h>
#include <string.h>
int main() {
    // Example 1: Using memcpy when there is NO overlap
    char src1[] = "Hello, World!";
    char dest1[20];
    printf("Source (before memcpy): %s\n", src1);
    memcpy(dest1, src1, strlen(src1) + 1);  // Copy string using memcpy
    printf("Destination (after memcpy): %s\n\n", dest1);
    // Example 2: Using memmove when memory regions OVERLAP
    char src2[] = "OverlapExample";
        // Memory overlap: We copy the string starting from index 0 to index 5
    printf("Source (before memmove): %s\n", src2);
    memmove(src2 + 5, src2, 7);  // Move the first 7 characters to index 5
    printf("Source (after memmove): %s\n\n", src2);
    // Example 3: Using memcpy when there is an overlap (this will show incorrect behavior)
    char src3[] = "OverlapIssue";
        printf("Source (before memcpy with overlap): %s\n", src3);
    memcpy(src3 + 5, src3, 7);  // This can cause data corruption due to overlap
    printf("Source (after memcpy with overlap): %s\n\n", src3);    return 0; }
```

Run Time Protection Strategies

❑ Detection and Recovery

- These strategies modify the runtime environment to detect buffer overflows and allow the system or application to recover safely or fail gracefully.

- Detection and recovery are not as reliable as prevention because attackers can still exploit the system once a buffer overflow occurs.

- These strategies serve as a backup in case preventive measures fail, acting as an extra layer of protection.

Object Size Checking

- Verify the size of objects (buffers, arrays, structures) before performing operations that might exceed their capacity.

- Object size checking is a crucial aspect of runtime protection strategies in systems programming, particularly to prevent memory corruption vulnerabilities like buffer overflows, stack smashing, or heap corruption.

- Runtime protection strategies use various techniques to ensure that an object (e.g., an array or a buffer) being manipulated is within its allocated bounds, thereby safeguarding against malicious or accidental memory exploits.

Object Size Checking

In C and C++, programmers can implement their own size-checking mechanisms using standard library functions and custom checks, as these languages do not have built-in runtime size verification for most data structures like arrays and buffers.

sizeof operator: The sizeof operator provides the compile-time size of an object or type.It cannot determine the size of dynamically allocated arrays or pointer-based arrays.
Example:

```
int arr[10];
size_t arr_size = sizeof(arr) / sizeof(arr[0]);  //
Calculate array size at compile-time
```

Object Size Checking

Manual Bound Checking

For dynamically allocated memory (using malloc or new), programmers need to manually track the size of allocated memory and perform bounds checks.

```
int* arr = (int*)malloc(10 * sizeof(int));  // Dynamically allocated array of 10 integers
if (index >= 0 && index < 10) {  // Manual bounds checking
    arr[index] = value;
} else {
    // Handle out-of-bounds access
}
```

Run Time protection tools

Address Sanitizer:

- **ASan is a fast memory error detector that instruments memory accesses at runtime to catch out-of-bounds accesses, use-after-free errors, and buffer overflows.**
- **ASan maintains shadow memory to track the size and boundaries of dynamically allocated objects. Any memory access beyond an object's size triggers an error.**

```
gcc -fsanitize=address -g myprogram.c -o myprogram
```

Operating System Strategies

Address Space layout Randomization

- Address Space Layout Randomization (ASLR) is a security technique used in modern operating systems to prevent exploitation of memory corruption vulnerabilities, such as buffer overflows.
- By randomly arranging the address space of a process, ASLR makes it more difficult for attackers to predict the locations of key data structures, libraries, and executable code.

How ASLR works

1. **Randomization of Memory Addresses:**
   1. **When a program is executed, ASLR randomizes the starting addresses of various regions of a process's memory space, including:**
      1. **The base addresses of executable code and shared libraries (e.g., DLLs).**
      2. **The stack, heap, and memory regions for loaded modules.**
   2. **Each time a program runs, these addresses change, making it difficult for an attacker to predict where to inject malicious code.**
2. **Mapping of Memory Regions:**
   1. **Typical memory regions randomized by ASLR include:**
      1. **Executable code segments: The locations where the program's executable code resides.**
      2. **Shared libraries: Dynamically linked libraries loaded into memory.**
      3. **Heap: The area used for dynamic memory allocation.**
      4. **Stack: The memory area used for function call management and local variables.**
3. **Implementation:**
   1. **ASLR can be implemented at both the kernel level (affecting all processes) and the application level (specific to an application).**
   2. **Most modern operating systems, including Windows, Linux, and macOS, support ASLR.**

Benefits of ASLR

- **Increased Security:**
  - **ASLR significantly complicates the task of exploiting memory corruption vulnerabilities, as attackers can no longer rely on fixed memory addresses for their exploits.**
- **Defense in Depth:**
  - **ASLR is often used alongside other security mechanisms, such as Data Execution Prevention (DEP) and Control Flow Integrity (CFI), to provide a more robust security posture.**
- **Reduction of Exploitability:**
  - **By randomizing memory addresses, ASLR reduces the likelihood of successful exploitation of vulnerabilities that rely on specific memory addresses (e.g., Return-Oriented Programming (ROP) attacks).**

Future Directions

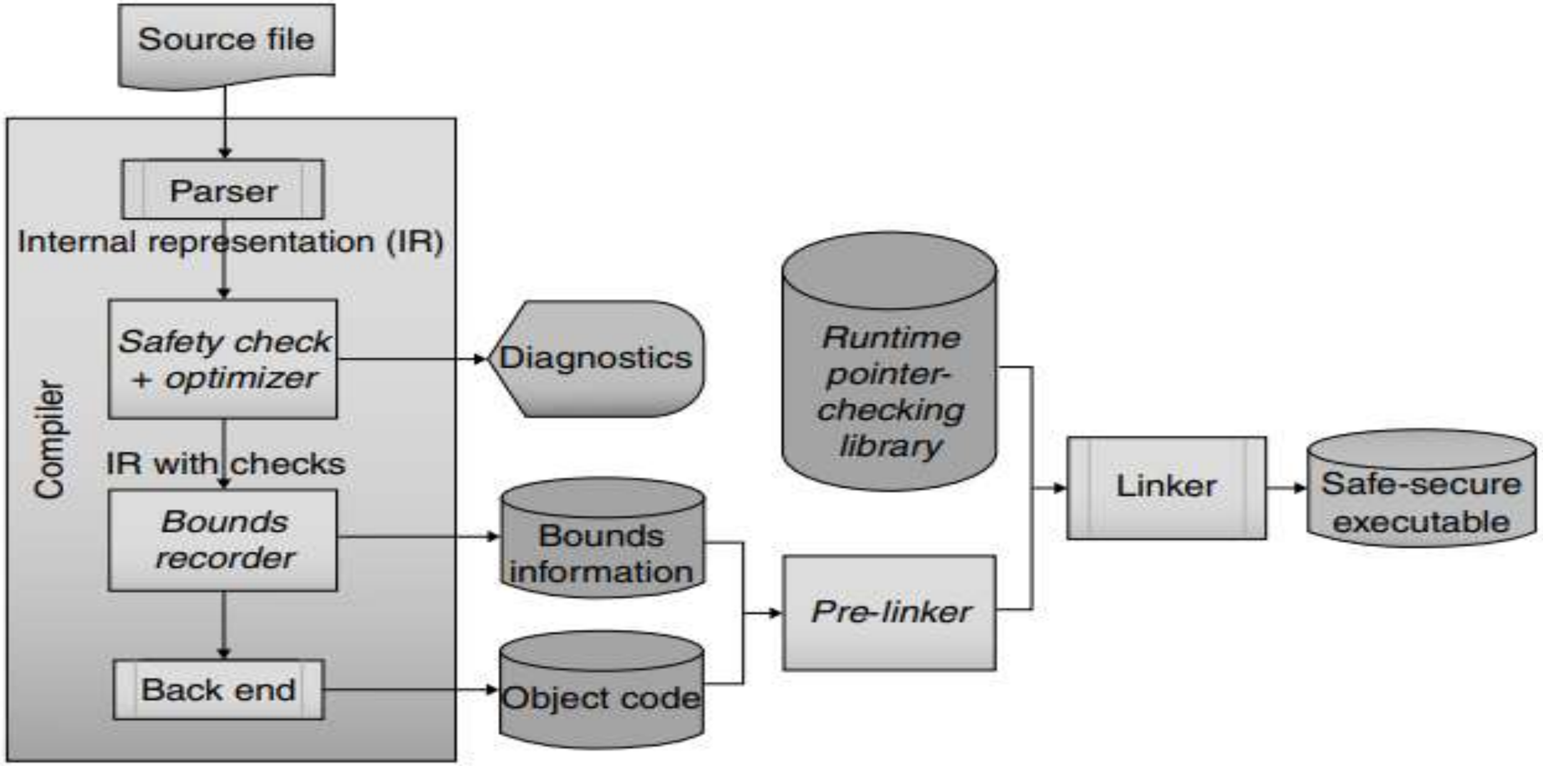


Figure 2.19 A possible Safe-Secure C/C++ (SSCC) implementation

Notable Vulnerabilities

Remote Login:

Remote login refers to the process of accessing a computer or network from a remote location, allowing users to interact with systems and applications as if they were physically present at the device. This functionality is essential for system administration, technical support, and remote work scenarios. Here’s an overview of remote login, common protocols, tools, and security considerations.

Many Unix Systems provide the rlogin program, which establishes a remote login session from its user’s terminal to a remote host computer. The rlogin program passes the user’s current terminal by the TERM environment variable to the remote host computer.

Notable Vulnerabilities

Kerberos:

Kerberos is a network authentication protocol designed to provide secure authentication for users and services over an insecure network. It uses secret-key cryptography to ensure that both users and services can confirm each other's identities without transmitting passwords over the network. Kerberos is widely used in various systems and applications, including Microsoft Windows, UNIX/Linux systems, and various enterprise environments.