Will Gusackova 20107603 Computer Forensics and Security BSc (Hons) year 2
Secure Programing

Secure Programing – Overflows

Introduction:

This report contains information about overflows in C coding language. The following overflows are mentioned: Stack-based buffer overflows, heap-based buffer overflows and format string attack. Examples will follow by ways to avoid buffer overflow threats, to protect your code against malicious attackers or accidental errors that a user can make, which can affect the program.

All codes used are uploaded to:
https://github.com/WillGusackov/SecureProgramingr/tree/main/CA2

Overflows in C:

Overflows in C are caused by the coder who didn't consider memory usage, exploitation possibilities and unsecure c coding methods by the usages of strcpy() instead of strpcpy() etc.

The attacker modifies the code in order to gain access or manipulate for malicious intent, they manipulate coding errors to overwrite certain data, to the attackers' needs or goals, to compromise, destruct, steal or sell data.

Buffer overflows generally a big vulnerability in the website / program, as memory storage capacity gets overflowed which overwrites memory data.

Stack-based buffer overflows CWE-121:

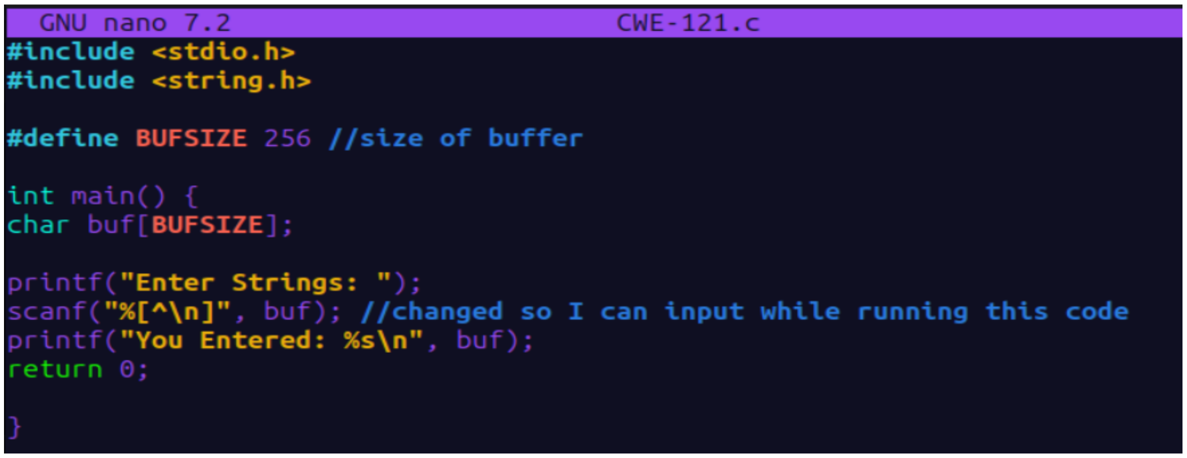
Stack based buffer overflow is traditionally the most common way of c exploitation, as the process of overwriting memory is simple, all you need is a poorly written code that doesn't disclose the memory space of char[], if this is not defined, the attacker can write a lengthy line of code that will overwrite the memory space, retreat back to the start of the string and treat the input as instructions instead.

This is known as CWE-121 (Common Weakness Enumeration 121 for Stack-based buffer overflows)

Impacts:

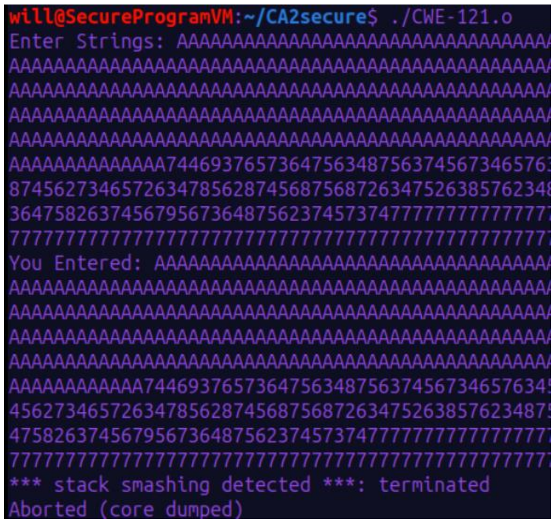
This vulnerability could lead to: Crashes, memory corruption, Recourse Consumption (CPU / Memory), and memory exploitation and bypassing / access control by execution of unauthorized code.

Example of CWE-121:



```
  GNU nano 7.2                                CWE-121.c
#include <stdio.h>
#include <string.h>

#define BUFSIZE 256 //size of buffer

int main() {
char buf[BUFSIZE];

printf("Enter Strings: ");
scanf("%[^\n]", buf); //changed so I can input while running this code
printf("You Entered: %s\n", buf);
return 0;

}
```

(CWE-121.c)



```
will@SecureProgramVM:~/CA2secure$ ./CWE-121.o
Enter Strings: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA7446937657364756348756374567346576
874562734657263478562874568756872634752638576234
364758263745679567364875623745737477777777777777
777777777777777777777777777777777777777777777777
You Entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA744693765736475634875637456734657634
456273465726347856287456875687263475263857623487
475826374567956736487562374573747777777777777777
777777777777777777777777777777777777777777777777
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Code modified to accept user input while program is running, inspired by: (Secure Software, Inc., 2005)

To run the code download and run on the Linux terminal:

- gcc -o CWE-121.o CWE-121.c
- ./CWE-121.o

- This code has a buffer size of 256, it asks for the user to input a string, scanf() scans the user input and reads back the input the user has written.

- The code returns true no matter the input (return 0;)

Code Vulnerabilities:

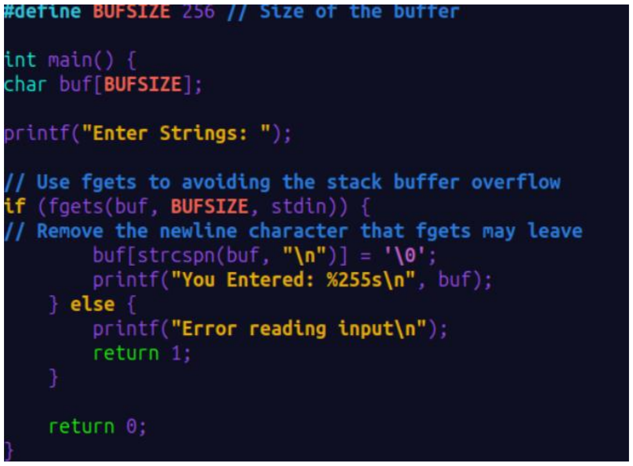
- This code is unsafe to use, as scanf(%[^/n]) reads the data till it reaches the null character, it does not validate the user input and will lead to a stack-based buffer overflow as it can potentially overwrite the memory.

- "buf" Array has no input validation to check if the user input will exceed 256 bytes.

- Attacker can enter address for e.g. the POSIX system() command, manipulating the stack, which calls a return into libc exploit. (Secure Software, Inc., 2005)

Vulnerability protections:

Following changes that where made:



```c
#define BUFSIZE 256 // Size of the buffer

int main() {
char buf[BUFSIZE];

printf("Enter Strings: ");

// Use fgets to avoiding the stack buffer overflow
if (fgets(buf, BUFSIZE, stdin)) {
// Remove the newline character that fgets may leave
        buf[strcspn(buf, "\n")] = '\0';
        printf("You Entered: %255s\n", buf);
    } else {
        printf("Error reading input\n");
        return 1;
    }

    return 0;
}
```

(CWE-121fix.c)

- %255s Truncates the string for user input, it will show the user the truncated version.
- fget() reads specifically 256 bytes only which avoids stack-based overflows.
- If loop implemented in case of failure occurs e.g. if fget() fails it will give the user an error.
- strcspn() cleans input from fget().
- Return 1 indicating a failure.

Heap-based buffer overflows CWE-122:

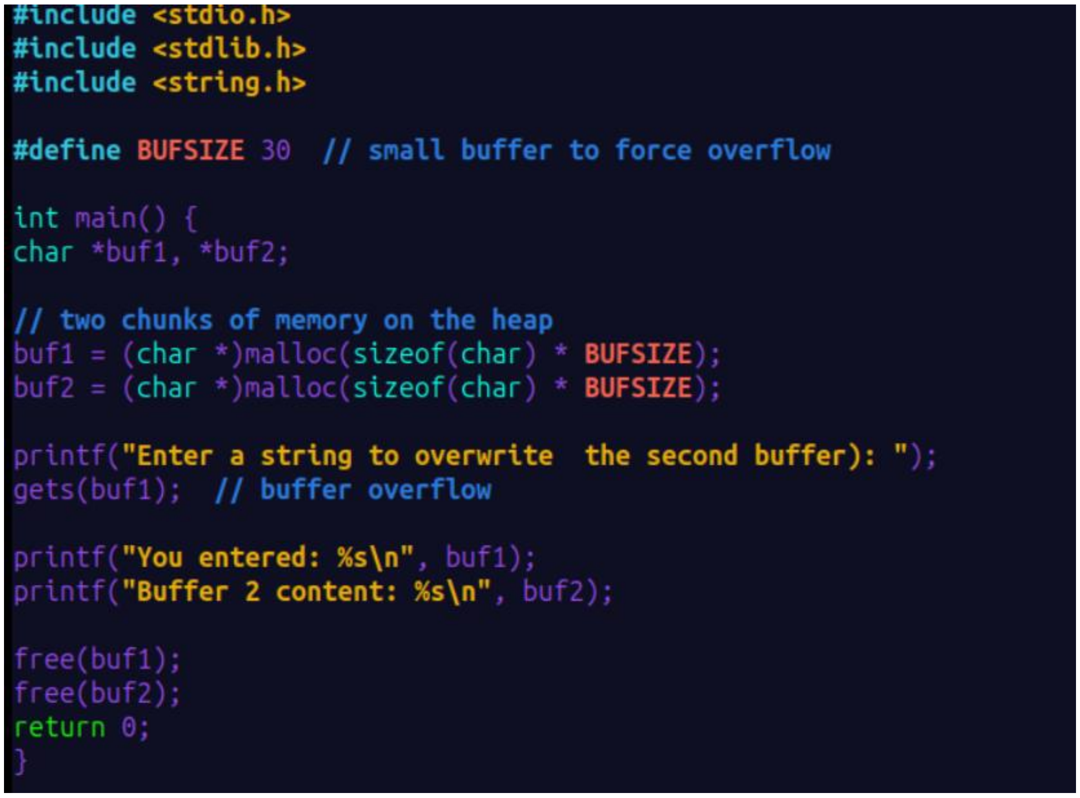
Heap-based buffer overflows are caused by overflow of the memory space that's reserved for program. Heap-based overflows risk other memory spaces to bleed into other ones in the heap. It occurs when data bounds beyond the heap. What makes heap overflows different compared to stack-based overflows is that heaps usually contain complex memory management with pointers.

Usually allocated using a routine like as malloc() (CWE, 2024)

Impacts:

This vulnerability could lead to: Memory bleeding, Crashes, Resource consumption, Data loss/ Corruption.

Example of CWE-122:



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

#define BUFSIZE 30  // small buffer to force overflow

int main() {
char *buf1, *buf2;

// two chunks of memory on the heap
buf1 = (char *)malloc(sizeof(char) * BUFSIZE);
buf2 = (char *)malloc(sizeof(char) * BUFSIZE);

printf("Enter a string to overwrite  the second buffer): ");
gets(buf1);  // buffer overflow

printf("You entered: %s\n", buf1);
printf("Buffer 2 content: %s\n", buf2);

free(buf1);
free(buf2);
return 0;
}
```

(CWE-122.c)

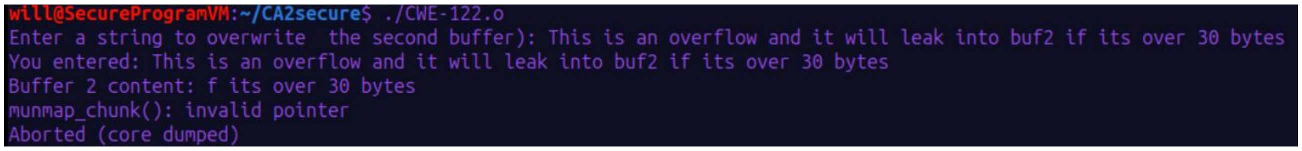
To run the code download and run on the Linux terminal:

- gcc –o CWE-122.o CWE-122.c
- ./CWE-122.o

- Points in both buf1 and buf2, to allocate memory spaces to both buf1 and buf2, with the usage of malloc().

- The size is both 30 bytes, if a user enters more than that, then a heap overflow occurs, and it will leak into the memory of buf2.

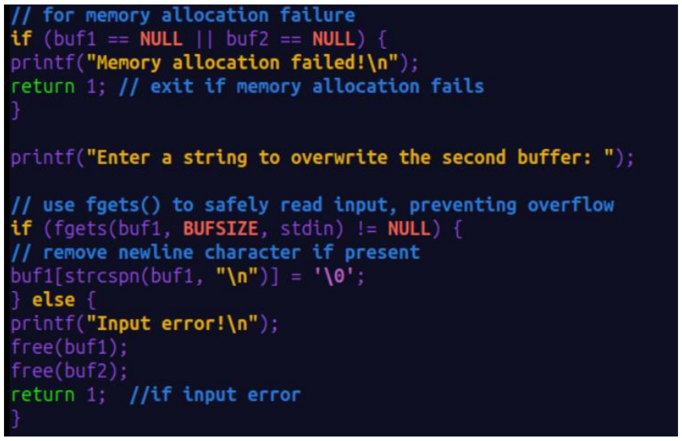
- The gets() function gets the user input in a bad way, it's an example of a buffer overflow.



```
will@SecureProgramVM:~/CA2secure$ ./CWE-122.o
Enter a string to overwrite  the second buffer): This is an overflow and it will leak into buf2 if its over 30 bytes
You entered: This is an overflow and it will leak into buf2 if its over 30 bytes
Buffer 2 content: f its over 30 bytes
munmap_chunk(): invalid pointer
Aborted (core dumped)
```

Vulnerability protections:

Following changes that where made:

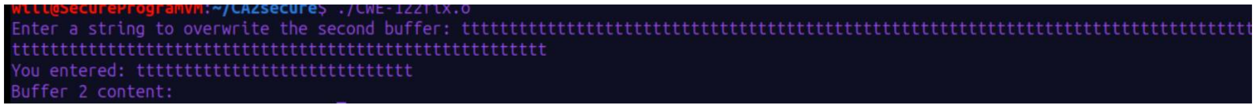


```
// for memory allocation failure
if (buf1 == NULL || buf2 == NULL) {
printf("Memory allocation failed!\n");
return 1; // exit if memory allocation fails
}

printf("Enter a string to overwrite the second buffer: ");

// use fgets() to safely read input, preventing overflow
if (fgets(buf1, BUFSIZE, stdin) != NULL) {
// remove newline character if present
buf1[strcspn(buf1, "\n")] = '\0';
} else {
printf("Input error!\n");
free(buf1);
free(buf2);
return 1;  //if input error
}
```

- get() changed to getf(), which limits the size, so it won't overflow to the other allocated memory space like buf2.
- If statement was added to avoid unexpected errors.
- Else is added to abort if there's an input error as it returns false.

(CWE-122fix.c)

<u>Fixed version output:</u>



```
will@SecureProgramVM:~/CA2secure$ ./CWE-122fix.o
Enter a string to overwrite the second buffer: ttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
tttttttttttttttttttttttttttttttttttttttttttttttttttttttt
You entered: ttttttttttttttttttttttttttttttt
Buffer 2 content:
```

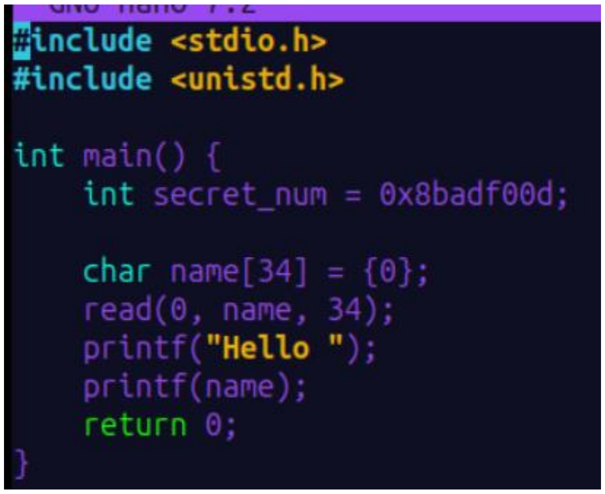
Format String Attack CWE-134:

Formatted String Attacks are the cause of printf() or other functions of the family. printf() prints formatted data to stdout, this can be exploited by a formatted string attack, this usually happens when the code doesn't apply string validation, e.g. inputting that if a user enters %, it doesn't indicate it's a code but instead a string.

It can print stack addresses, which shows the memory of the program, this vulnerability is a memory leak concern, attacker can use this to their advantage by inputting *"%x %x"* to show two stack values in hexadecimal. (*CTF101, 2024*)

Impacts:

This vulnerability could lead to: Memory Exposer, Execution of unauthorized commands, Data leak and potentially memory modification.



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

int main() {
    int secret_num = 0x8badf00d;

    char name[34] = {0};
    read(0, name, 34);
    printf("Hello ");
    printf(name);
    return 0;
}
```

To run the code, download and run on the Linux terminal:

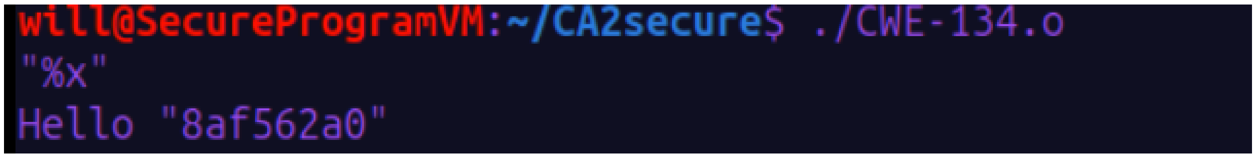
- gcc –o CWE-122.o CWE-122.c
- ./CWE-122.o

- read() reads user input, once input is placed it will print user input as printf() is called.

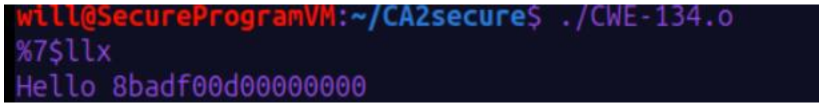
- The program prints "Hello "and then the user's name

Code Inspiration by: (*CTF101, 2024*)

String Attack example:



```
will@SecureProgramVM:~/CA2secure$ ./CWE-134.o
"%x"
Hello "8af562a0"
```



```
will@SecureProgramVM:~/CA2secure$ ./CWE-134.o
%7$llx
Hello 8badf00d00000000
```

example from: (*CTF101, 2024*)

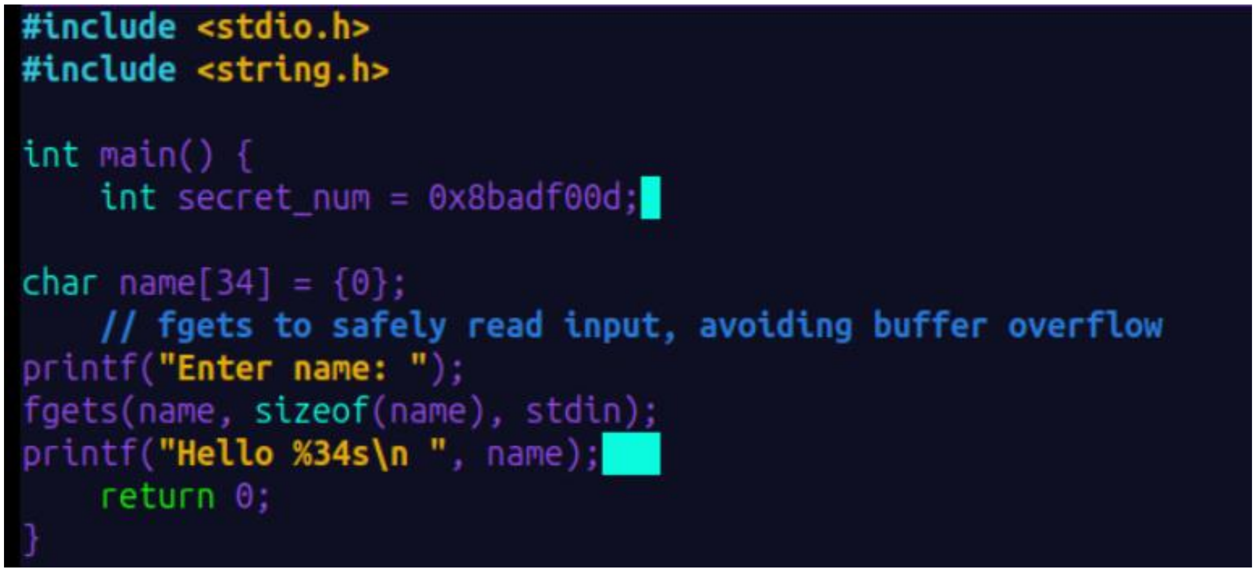
Code Vulnerabilities:

- printf(name); is unsafe as there is no validation to this code.

- Buffer overflow can occur in name as its not formatted in a way to protect itself if the user input is more than 34 bytes.

Vulnerability protections:

Following changes that where made:

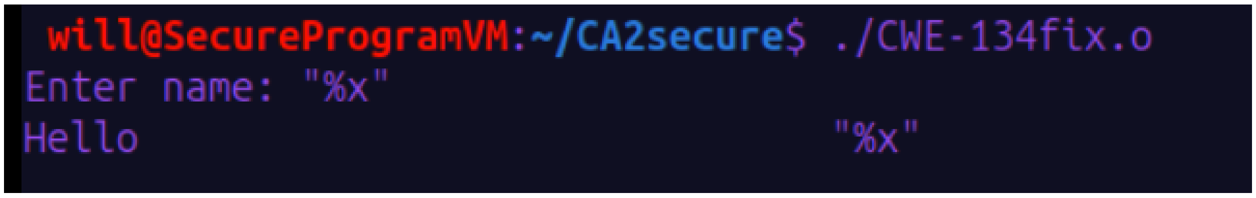


```c
#include <stdio.h>
#include <string.h>

int main() {
    int secret_num = 0x8badf00d;

char name[34] = {0};
    // fgets to safely read input, avoiding buffer overflow
printf("Enter name: ");
fgets(name, sizeof(name), stdin);
printf("Hello %34s\n ", name);
    return 0;
}
```

- Name got truncated.

- fget() to avoid buffer overflows.

- Include name in printf(), this stops string attacks from occurring.



```
will@SecureProgramVM:~/CA2secure$ ./CWE-134fix.o
Enter name: "%x"
Hello                                  "%x"
```

Conclusion:

Securing code against overflows requires proper memory management and input validation. Buffer overflows, such as stack-based, heap-based, and format string attacks, can lead to serious vulnerabilities.

To prevent these, developers should use secure functions like strncpy() and fgets(), validate user input, and implement bounds checking. By following these best practices, developers can reduce the risk of attacks and protect the integrity of their code and data.

General Security Tips:

- **Input Validation:** Always validate and sanitize user input to prevent overflows, especially in functions like scanf(), gets(), or printf() that don't automatically limit input length.

- **Safer Functions:** Replace unsafe functions like strcpy() with their safer counterparts such as strncpy() or snprintf() which allow you to define buffer sizes explicitly.
- **Memory Management:** Properly allocate and manage memory in your programs. For heap-based overflows, be mindful of buffer sizes when using functions like malloc() or calloc().

- **Use of Modern Compilers and Security Features:** Enable security features such as stack protection, buffer overflow checks, and ASLR to make exploiting vulnerabilities much harder for attackers.

- **Regular Security Audits:** Perform regular code reviews and security audits to catch potential vulnerabilities before they can be exploited.

Reference:

CWE, 2024. *CWE-122: Heap-based Buffer Overflow* Available at: https://cwe.mitre.org/data/definitions/122.html [Accessed 2th March 2025]

Secure Software, Inc., 2005. *The CLASP Application Security Process*. [pdf] Available at: https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf [Accessed 2th March 2025]

CWE, 2025. CWE-121: Stack-based buffer overflow. Available at: https://cwe.mitre.org/data/definitions/121.html [Accessed 2nd March 2025].

Rapid7, 2019. *Stack-based buffer overflow attacks: What you need to know*. Available at: https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/ [Accessed 3rd March 2025]

Fortinet, 2025. *Buffer overflow*. Available at: https://www.fortinet.com/resources/cyberglossary/buffer-overflow [Accessed 3rd March 2025]

CTF101, 2024. *What is a format string vulnerability?* Available at: https://ctf101.org/binary-exploitation/what-is-a-format-string-vulnerability/ [Accessed 6th March 2025]