Somaiya Vidyavihar University
K. J. Somaiya College of Engineering
Department of Computer Engineering

Batch: B1 Roll No.: 16010121045

Experiment No. 2

Title: Buffer Overflow attack

Objective: To study the effect of buffer overflow attack on a C program.

Expected Outcome of Experiment: To implement Cryptanalysis Tools .

CO	Outcome

Books/ Journals/ Websites referred:

https://resources.infosecinstitute.com/topics/cryptography/cryptanalysis-tools/

Abstract:-

A buffer overflow attack is a type of security vulnerability that occurs when an attacker inputs more data into a buffer or memory area than it can handle, overwriting adjacent memory locations. This can cause unexpected behavior or crashes and can potentially allow the attacker to execute malicious code. Buffer overflow attacks can occur in software written in any programming language and are typically caused by errors in code such as using insufficient bounds checking or failing to properly validate input data. To prevent buffer overflow attacks, developers should follow best practices such as using secure coding techniques, performing input validation, and using safer programming languages and libraries.

Related Theory: -

Buffer overflow errors are characterized by the overwriting of memory fragments of the process, which should have never been modified intentionally or unintentionally. Overwriting values of the IP (Instruction Pointer), BP (Base Pointer) and other registers causes exceptions, segmentation faults, and other errors to occur. Usually, these errors end execution of the application in an unexpected way. Buffer overflow errors occur when we operate on buffers of char type.

Buffer overflows can consist of overflowing the stack [Stack overflow] or overflowing the heap [Heap overflow]. For example, a buffer for log-in credentials may be designed to expect username and password inputs of 8 bytes, so if a transaction involves an input of 10 bytes (that is, 2 bytes more than expected), the program may write the excess data past the buffer boundary. Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.

Let’s understand with the help of an example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void main()
{
   char *ptr;
   char *dptr;
   ptr = (char *)malloc(10 * sizeof(char));
```

```
    dptr = (char *)malloc(10 * sizeof(char));
    printf("Address of ptr: %d\n", ptr);
    printf("Address of dptr: %d\n", dptr);
    printf("Enter The String:\n");
    gets(ptr);
    system(dptr);
}
```

The code you provided has a potential buffer overflow vulnerability due to the use of the **gets** function. The **gets** function is inherently unsafe because it does not perform any bounds checking on the input, and it can easily lead to buffer overflows.

Hence an attacker can execute commands directly by storing the commands in the dptr as it gets executed as a system command. In the below example I have executed the **ls -a** command which displays all the stored files in the directory.

```
❯ cd "/Users/pargatsinghdhanjal/Documents/College/Sem6/IS/Programs/" && gcc exp2.c -o exp2 && "/Users/pargat
singhdhanjal/Documents/College/Sem6/IS/Programs/"exp2
Address of ptr: 904937936
Address of dptr: 904937952
Enter The String:
warning: this program uses gets(), which is unsafe.
pargatsinghdhanjls -a
.       ..      exp2    exp2.c
```

Types of Buffer Overflow Attacks:

1. **Stack-based Buffer Overflow:**
    - This occurs when a program writes more data to a stack buffer than its allocated space. It often involves overwriting the return address, allowing an attacker to control the program's execution flow.
2. **Heap-based Buffer Overflow:**
    - Unlike stack-based overflow, heap-based overflow exploits vulnerabilities in dynamically allocated memory on the heap. Attackers can manipulate heap structures, leading to unintended consequences such as data corruption or execution of malicious code.
3. **Format String Vulnerabilities:**
    - This attack occurs when user input influences the format string of a function, leading to unexpected behavior. If not handled correctly, format string vulnerabilities can result in information leakage or code execution.

4. **Return-Oriented Programming (ROP):**

    - ROP attacks involve chaining together short sequences of code, known as gadgets, that end with a return instruction. Attackers construct a series of gadgets to achieve their goals without injecting new code. This technique is often used to bypass DEP (Data Execution Prevention) mechanisms.

5. **Integer Overflow:**

    - In this scenario, an arithmetic operation results in an integer value that exceeds the maximum representable value. If not properly checked, integer overflow can lead to unexpected behavior, potentially exploited by attackers.

Implementation Details:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void main()
{
   char *ptr;
   char *dptr;
   // clrscr();
   ptr = (char *)malloc(10 * sizeof(char));
   dptr = (char *)malloc(10 * sizeof(char));
   printf("Address of ptr: %d\n", ptr);
   printf("Address of dptr: %d\n", dptr);
   printf("Enter The String:\n");
   // gets(ptr);
   fgets(ptr, 10, stdin);
   system(dptr);
}
```

To fix this issue, you should use a safer alternative for reading input, such as **fgets**, which allows you to specify the maximum number of characters to read:

Additionally, it's worth noting that the use of **gets** has been deprecated in modern C programming due to its security risks. The recommended alternative is to use functions like **fgets** or **scanf** with proper size limits to avoid buffer overflows.

```
❯ cd "/Users/pargatsinghdhanjal/Documents/College/Sem6/IS/Programs/" && gcc exp2.c -o exp2 && "/Users/pargatsinghdhanjal/Documents/College/Sem6/IS/Programs/"exp2
Address of ptr: 1323327088
Address of dptr: 1323327456
Enter The String:
dfjgklwbrojigeqoghbrjio;efklgrhbhopeifhjbvhjfiruoeqfpjhdvkfsbgoeuijlkdvsnbfskwlhfeuiqjkdvacnbsgdlfwuihojklnvxsmbcdhghuj;kqcndlbvhfjqknjdmvblhefkls;cdnvmablehfkldcnvmabekfjhqlnvdmb.knlefhjjkwscdnmvbfeliqhjedknvlbhelhjdkbvahlguewqifjkdnvbls
```

Conclusion:- Learnt to use and implement Buffer overflow concept.

Postlab questions:

1. What is the difference between malicious and non-malicious program flaws? Give suitable examples of each.

A program flaw can be classified as either malicious or non-malicious based on its intent and impact. Malicious program flaws are those that are designed to cause harm or damage to a system or its users. They are intentionally introduced into a program with the goal of exploiting vulnerabilities for malicious purposes such as stealing data, disrupting services, or compromising security.

Examples of malicious program flaws include:

• **Virus**: A self-replicating program that spreads from one computer to another and causes harm to the system or data.
• **Trojan**: A program that appears to be legitimate but actually contains hidden malicious code.
• **Buffer Overflow Attack**: An attack that exploits a buffer overflow vulnerability in a program by sending it more data than it can handle, causing the program to crash or execute arbitrary code.

Non-malicious program flaws, on the other hand, are unintentional mistakes made by developers that can cause a program to behave in unexpected ways or lead to security vulnerabilities. These flaws are not introduced with the intention of causing harm, but they can still have a negative impact on the program or system.

Examples of non-malicious program flaws include:

• **Input Validation:** Failing to validate user input, this can lead to security vulnerabilities such as SQL injection or cross-site scripting.
• **Resource Leak:** Failing to properly manage memory or other system resources, leading to resource exhaustion and potential denial of service attacks.
• **Exception Handling:** Poorly handling exceptions, this can lead to crashes or unpredictable behaviour in the program.

2. Explain different types of buffer overflow attack.

The different types of buffer overflow attacks are as follows:

1. **Stack-based Buffer Overflow:**
    - This occurs when a program writes more data to a stack buffer than its allocated space. It often involves overwriting the return address, allowing an attacker to control the program's execution flow.
2. **Heap-based Buffer Overflow:**
    - Unlike stack-based overflow, heap-based overflow exploits vulnerabilities in dynamically allocated memory on the heap. Attackers can manipulate heap structures, leading to unintended consequences such as data corruption or execution of malicious code.
3. **Format String Vulnerabilities:**
    - This attack occurs when user input influences the format string of a function, leading to unexpected behavior. If not handled correctly, format string vulnerabilities can result in information leakage or code execution.
4. **Return-Oriented Programming (ROP):**
    - ROP attacks involve chaining together short sequences of code, known as gadgets, that end with a return instruction. Attackers construct a series of gadgets to achieve their goals without injecting new code. This technique is often used to bypass DEP (Data Execution Prevention) mechanisms.
5. **Integer Overflow:**
    - In this scenario, an arithmetic operation results in an integer value that exceeds the maximum representable value. If not properly checked, integer overflow can lead to unexpected behavior, potentially exploited by attackers.

3. List the mechanisms to prevent buffer overflow attack.

The mechanisms to Prevent Buffer Overflow Attacks:

1. **Bounds Checking:**
    - Validate input size and ensure that data written into a buffer does not exceed its allocated size. This can be achieved using functions like **fgets**, **strncpy**, and by explicitly checking array indices.
2. **Use Safe Library Functions:**
    - Replace unsafe functions like **gets**, **strcpy**, and **sprintf** with safer alternatives like **fgets**, **strncpy**, and **snprintf**. Safe functions often include parameters specifying the size of the destination buffer.
3. **Address Space Randomization:**
    - Implement Address Space Layout Randomization (ASLR) to randomize the memory addresses of key system components, making it harder for attackers to predict the location of buffers and injected code.
4. **Compiler Security Features:**
    - Utilize modern compilers with security features that can detect and prevent certain types of buffer overflows. Examples include GCC's **-fstack-protector** and Microsoft Visual C++'s **/GS** flag.
5. **Input Validation and Sanitization:**
    - Validate and sanitize all input data to ensure it adheres to expected formats and lengths. Reject or truncate input that exceeds specified limits.