Securing Coding Lab Experiment - 6

String Handling and Buffer overflow vulnerabilities

Objective

Understand string handling in C/C++ programming, identify common errors associated with C-style strings, and explore buffer overflow vulnerabilities and their mitigations.

1. Unbounded String Copies

```c
#include <stdio.h>

int main(void) {
    char Password[8];
    printf("Enter 8-character password: ");
    gets(Password);  // Vulnerable: unbounded input
    printf("Password entered: %s\n", Password);
    return 0;
}
```

Output:

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab6]
└─$ ./unbound
Enter 8-character password: 12345678
Password entered: 12345678

┌──(karthikeyan㉿kali)-[~/Desktop/lab6]
└─$ ./unbound
Enter 8-character password: 123456789
Password entered: 123456789
```

Explanantion :

The gets () function reads input from the user until a newline or EOF is encountered, but it does not perform any bounds checking. This means that if the user enters more than 8 characters (the size of the Password array), the additional characters will overflow the allocated memory, leading to a buffer overflow.

2. String Copy and Concatenation

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

int main(int argc, char *argv[]) {
    char name[2048];
    strcpy(name, argv[1]);  // Vulnerable: no size check
    strcat(name, " = ");
    strcat(name, argv[2]);
    printf("%s\n", name);
    return 0;
}
```

Output:

```
┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# gcc -o copycat copycat.c

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# ./copycat
zsh: segmentation fault (core dumped)  ./copycat
```

Explanation:

The code is vulnerable to buffer overflow because strcpy() and strcat() do not check the size of the input. If the combined size of argv[1], " = ", and argv[2] exceeds the 2048-byte buffer, it can lead to memory corruption or exploitation. Use safer functions like strncpy() and strncat() to prevent this issue.

3. NULL Termination errors

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

int main(int argc, char *argv[]) {
char a[16];
char b[16];
char c[32];

strncpy(a, "0123456789abcdef", sizeof(a));
strncpy(b, "0123456789abcdef", sizeof(b));
strncpy(c, a, sizeof(c));

printf("String a: %s\n", a);
printf("String b: %s\n", b);
printf("String c: %s\n", c);
return 0;
}
```

Output:

```
┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# gcc -o null null.c

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# ./null
String a: 0123456789abcdef
String b: 0123456789abcdef0123456789abcdef
String c: 0123456789abcdef
```

Explanation:

- String **a**: 0123456789abcdef – This string is 16 characters long. It may have been stored in a fixed-size buffer, which fits the exact size.
- String **b**: 0123456789abcdef0123456789abcdef – This string is 32 characters long. It appears that this string was stored in a larger buffer.
- String **c**: 0123456789abcdef – This string is identical to **String a**, indicating that it may have been stored in a buffer similar in size to **String a**.

4. C++ Unbounded errors

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

int main()
 {
 char buf[12];
 cin >> buf;
 cout<< "echo: " << buf << endl;
 return 0;
 }
```

Output:

```
┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# g++ -o uber uber.cpp -fstack-protector

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# ./uber
123456789012345
echo: 123456789012345
*** stack smashing detected ***: terminated
zsh: IOT instruction (core dumped)  ./uber
```

5. Fix Off by One Error

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

int main(void)
{
char s1[] = "012345678";
char s2[] = "0123456789";
char *dest;
int i;
strcpy_s(s1,sizeof(s2),s2);
dest = (char *)malloc(strlen(s1));
for (i=1; i <= 11; i++) {
dest[i] = s1[i];
}
dest[i] = '\0';
printf("dest = %s\n", dest);
/* ... */;
}
```

Explanation:

In the for loop, where the condition i <= 11 causes the loop to access memory beyond the bounds of the s1 array (which only has 9 characters). This can lead to buffer overflow and undefined behavior. Additionally, the destination buffer dest is allocated with strlen(s1) but does not account for the null terminator .

Corrected Code:

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

int main(void) {
    // Adjust the size of s1 to accommodate the string including the null terminator
    char s1[11];  // 10 characters + 1 for '\0'
    char s2[] = "0123456789"; // 10 characters
    char *dest;
    int i;

    // Safely copy s2 into s1, ensuring we do not exceed the buffer size of s1
    strncpy(s1, s2, sizeof(s1) - 1); // Leave space for the null terminator
    s1[sizeof(s1) - 1] = '\0'; // Ensure s1 is null-terminated

    // Allocate enough memory for dest to hold the string s1 including the null terminator
    dest = (char *)malloc(strlen(s1) + 1); // +1 for null terminator
    if (dest == NULL) { // Check if malloc succeeded
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }
```

```c
    // Copy the content of s1 to dest safely
    for (i = 0; i < strlen(s1); i++) {
        dest[i] = s1[i];
    }
    dest[i] = '\0'; // Null-terminate the destination string

    printf("dest = %s\n", dest); // Print the content of dest

    free(dest); // Free the allocated memory
    return 0;
}
```

Output:

```
┌──(root㉿kali)-[/home/karthikeyan]
└─# gcc fix.c -o fix

┌──(root㉿kali)-[/home/karthikeyan]
└─# ./fix
dest = 0123456789
```

6. Buffer overflow

A buffer overflow occurs when data exceeds the allocated buffer size in memory. This excess data can overwrite adjacent memory, potentially corrupting data, leading to crashes, or even allowing an attacker to execute arbitrary code.

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

int main(int argc, char**argv){
    int authentication = 0;
    char cUsername[10], cPassword[10];
    strcpy(cUsername, argv[1]);
    strcpy(cPassword, argv[2]);
    if (strcmp(cUsername, "admin") == 0 && strcmp(cPassword, "adminpass") == 0)
    {

        authentication = 1;
    }

    if(authentication)
    {
        printf("Access Granted \n");
    }
    else
    {
        printf("Wrong Username and Password \n");
    }


}
```

Output:

```
┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# gcc -o buff -fno-stack-protector -z execstack -g buff.c

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# ./buff 123456789012345 12345678
Wrong Username and Password
```

7. Protection Against Buffer Overflows

Corrected Code (Using strncpy and Input Validation):

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

int main(int argc, char** argv) {
    int authentication = 0;
    char cUsername[10], cPassword[10];

    // Ensure at least 2 arguments are passed (username and password)
    if (argc < 3) {
        printf("Usage: %s <username> <password>\n", argv[0]);
        return 1;
    }

    // Safely copy input using strncpy, ensuring no buffer overflow
    strncpy(cUsername, argv[1], sizeof(cUsername) - 1);
    cUsername[sizeof(cUsername) - 1] = '\0';  // Ensure null termination

    strncpy(cPassword, argv[2], sizeof(cPassword) - 1);
    cPassword[sizeof(cPassword) - 1] = '\0';  // Ensure null termination
```

```c
    // Compare username and password safely
    if (strcmp(cUsername, "admin") == 0 && strcmp(cPassword, "adminpass") == 0) {
        authentication = 1;
    }

    // Output authentication result
    if (authentication) {
        printf("Access Granted\n");
    } else {
        printf("Wrong Username or Password\n");
    }

    return 0;
}
```

```
┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# gcc -o buffer buffer.c

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab6]
└─# ./buffer admin 123456789587654356
```

 Since the buffer for cPassword is only 10 characters long, and you're passing a string much longer than that, this could cause undefined behavior or a security vulnerability.
 This scenario should ideally be handled by proper input validation and bounds checking to prevent such overflow.