**Assignment 1**

**Of**

**CS6898W– Embedded Systems Security**

**On**

**Buffer Overflow**

**Prepared by**

**Dipendu Ghosh**

**CS23M509**

17th October, 2024

Contents

[Initial analysis of the program file lab1.c 3](#_Toc179890261)

[An explanation of the Makefile 3](#_Toc179890262)

[Summary 4](#_Toc179890263)

[Disassembling Techniques for C Binaries 5](#_Toc179890264)

[Using objdump 5](#_Toc179890265)

[Using gdb (GNU Debugger) 5](#_Toc179890266)

[Here I have used objdump to disassemble the code to view the opcodes as well 5](#_Toc179890267)

[Details of the exploit API 7](#_Toc179890268)

[High-Level Overview 9](#_Toc179890269)

[Manually creating the payload 9](#_Toc179890270)

[Executing manually from the command line 10](#_Toc179890271)

[Executing using gdb 10](#_Toc179890272)

[Normal execution with input “Random” 10](#_Toc179890273)

[Exploit execution with input “DipenduGhoshHELOCS23M509\x49\x58\x49\x41\x7c\x88\x04\x08” 10](#_Toc179890274)

[Payload generation 12](#_Toc179890275)

[Exploit generation and execution steps: 12](#_Toc179890276)

[Precautions and Counter Measures 12](#_Toc179890277)

[Contents of the zip 12](#_Toc179890278)

Initial analysis of the program file lab1.c

In the source file lab1.c, a command-line argument (string) is passed to the **welcome**() function. This string is copied into a local character array using **strcpy**() without checking the input first. This is a common sign of a potential security issue known as stack smashing.

The function **strcpy**() uses a special character called a NUL (represented as \0) to know when to stop copying. If this character is missing, either accidentally or on purpose, it can cause a buffer overflow. This happens when the input string is larger than the space allocated for the destination array. Attackers can take advantage of this vulnerability to overwrite nearby memory, corrupt data, or run harmful code, which can compromise the security of the application.

The following sections describe how this vulnerability can be exploited, with explanations for each method. The exploit is shown by calling the exploit() function, even though it can't be reached directly from the main() function.

An explanation of the Makefile

1. FLAGS = -m32 -g -fno-stack-protector -O0 -no-pie -static

* **FLAGS**: This variable holds the compilation options that will be passed to the GCC (GNU Compiler Collection) command.
* -**m32**: This flag tells the compiler to generate 32-bit code. It is useful if you're compiling on a 64-bit system but need to execute the code in a 32-bit environment.
* -**g**: This option includes debugging information in the compiled executable, making it easier to debug the program with tools like gdb.
* -**fno-stack-protector**: This disables the stack protection feature, which adds checks to prevent buffer overflows. Disabling it can help demonstrate vulnerabilities in educational contexts.
* -**O0**: This flag disables optimization. When optimization is turned off, the compiler does not attempt to improve the performance of the code, which makes debugging easier and ensures the original code structure is preserved.
* -**no-pie**: This option tells the compiler not to produce a Position Independent Executable (PIE). This can simplify certain types of analysis, as the address space will be fixed rather than random.
* -**static**: This flag instructs the compiler to create a statically linked executable. This means that all necessary libraries will be included in the executable file, rather than relying on shared libraries at runtime.

2. EXES = lab\_1

* **EXES**: This variable holds the name of the executable that will be generated. In this case, the output will be an executable named lab\_1.

3. all: clean $(EXES)

* **all**: This is the default target of the Makefile. When you run make, this is the target that gets executed.
* **clean**: This is a prerequisite, meaning that it will run the clean target before building the executables. This ensures that any previous build artifacts are removed before a new build starts.
* **$(EXES)**: This expands to lab\_1, which is the target that will be built after cleaning.

4. $(EXES):

This defines the rule for building the executable defined in the EXES variable. When make attempts to build lab\_1, it will execute the commands specified below this line.

* **gcc** $(FLAGS): This calls the GCC compiler with the flags specified in the FLAGS variable.
* **$@.c**: $@ is a special variable that refers to the target name (in this case, lab\_1). So this expands to lab\_1.c, meaning it will compile the source file lab\_1.c.
* -**o $@**: This tells the compiler to name the output file as the target name (i.e., lab\_1).

5. clean:

This target is used to clean up the build environment by removing the executable created by the build process.

* **rm -f** $(EXES): This command removes the executable file (in this case, lab\_1) if it exists. The -f flag forces the removal without prompting for confirmation, even if the file does not exist.

Summary

The flags mentioned earlier show that a stack smashing attack is not only possible but also relatively easy to do.

To carry out this type of attack, the simplest method is to change the return address of a function on the stack so that it points to the **exploit**() function instead. This can be done in functions like **welcome**(), where a buffer overflow can occur.

You can trigger this overflow by providing a command-line input (called a payload) that is longer than 12 bytes. This will cause the words array to overflow. To successfully exploit the vulnerability, you need to carefully design this payload.

This Makefile sets up the compilation of the C program (lab\_1) with specific flags for debugging and stack protection disabled. It defines how to build the executable and provides a clean target to remove previous builds before compiling again. This structure is used for demonstrating the security vulnerabilities.

When creating the payload to change the return address, it's important to avoid changing other values on the stack. This is crucial to ensure that the program's flow of execution remains the same. If there are protections in place, like a canary value to prevent buffer overflows, you need to keep these values unchanged. Looking at the disassembled code of the program can help you design the payload more effectively.

# Disassembling Techniques for C Binaries

Using objdump

* objdump is a powerful command-line tool that can display various information about object files, including disassembled code.
* Disassemble the binary using: objdump -d lab\_1.
* Displays addresses, opcodes (hex), and corresponding assembly instructions, making it easy to analyze how the binary works.

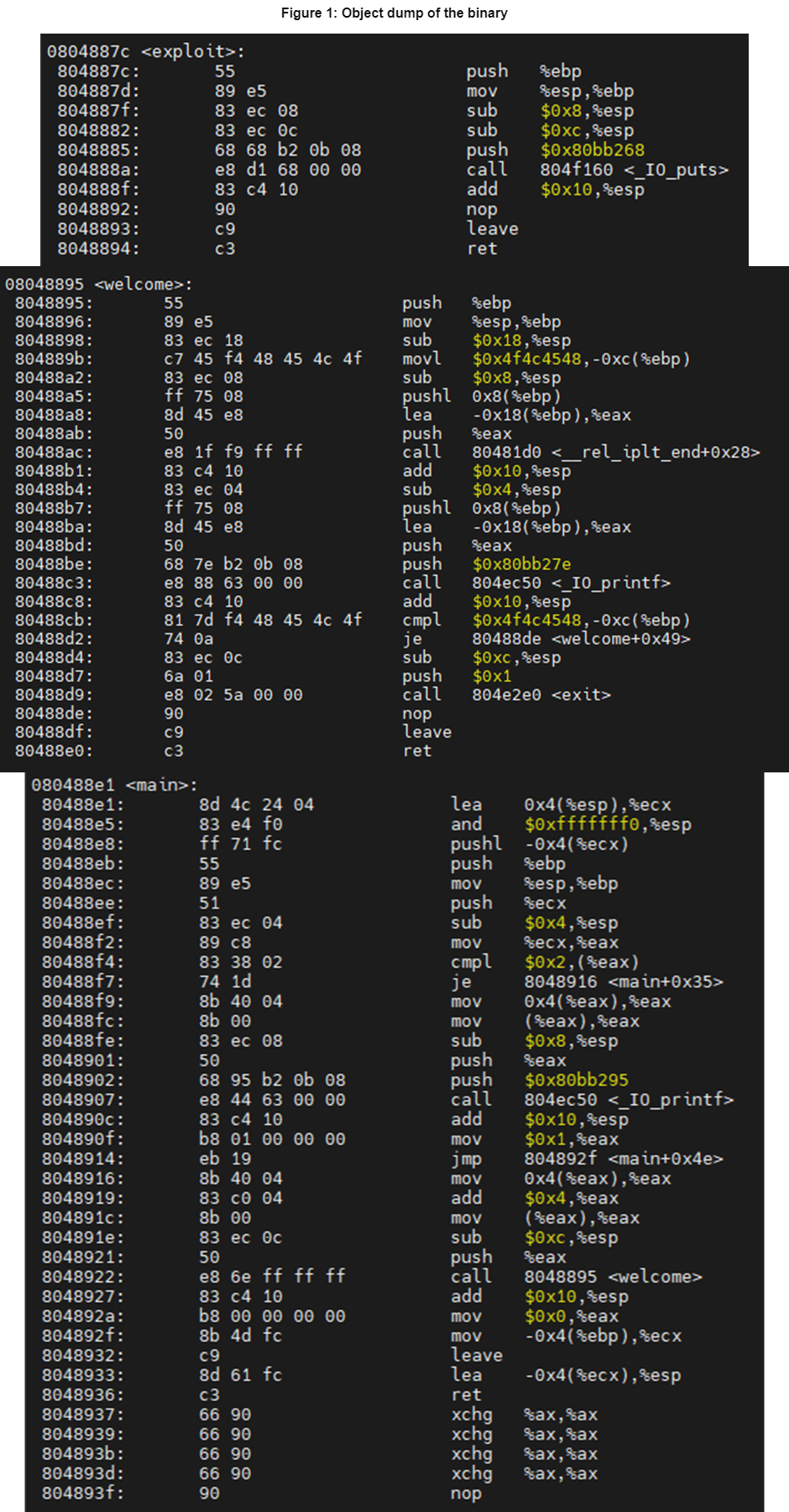
Using gdb (GNU Debugger)

* gdb allows you to debug programs and inspect their state while they are executing. It also provides functionality for disassembly.
* Start gdb with your binary: gdb lab\_1.
* Set breakpoints, execute the program, and use the disassemble command to view a function (e.g., disassemble main).
* To view opcodes, use x/i command to examine memory (e.g., x/10i main).
* Shows assembly instructions along with their addresses, allowing for interactive debugging and analysis.

Here I have used objdump to disassemble the code to view the opcodes as well

This assembly code is for the **x86 (32-bit)** architecture. Figure 1 below shows the complete opject dump of the lab\_1.c file. Here are the key points that show this:

* **Register names**: It uses 32-bit registers like **%ebp** (base pointer) and **%esp** (stack pointer).
* **Instructions**: Common x86 instructions like **push, mov, sub, call**, and **ret** are present.
* **Memory addressing**: It uses **x86-style** memory addressing, like **-0xc(%ebp)**, to access local variables on the stack.
* **Function setup and teardown**: The code starts with **push %ebp; mov %esp, %ebp** and ends with **leave**; **ret**, which is typical for x86 function calls.
* **Machine code**: The left side shows machine code bytes (like **55 and 89 e5**), which represent x86 32-bit instructions.



Details of the exploit API

This assembly code is a disassembly of a function, named **welcome**(), from a compiled C program. Let's break down the instructions line by line to understand their meaning and the corresponding high-level logic :-

1. Setup the Stack Frame

08048895 <welcome>:

8048895: 55 push %ebp

8048896: 89 e5 mov %esp,%ebp

8048898: 83 ec 18 sub $0x18,%esp

* **push %ebp**: The previous base pointer (ebp) is saved on the stack. This is done at the beginning of a function to save the previous frame’s base pointer.
* **mov %esp, %ebp**: The current stack pointer (esp) is moved to the base pointer (ebp), setting up a new stack frame.
* **sub $0x18, %esp**: Space for local variables is allocated on the stack by subtracting 0x18 (24 bytes) from the stack pointer. Subtract since the stack here is negatively-growing.

2. Setting Up the Canary Value (Security Check)

804889b: c7 45 f4 48 45 4c 4f movl $0x4f4c4548,-0xc(%ebp)

* **movl $0x4f4c4548, -0xc(%ebp)**: The hexadecimal value 0x4f4c4548 (which corresponds to the ASCII string "HELO") is stored at -0xc(%ebp) which is at ebp-12. This is a "canary" value, used to detect if the stack has been overwritten by a buffer overflow later in the function. The ‘l’ in movl means long, making it the 32-bit equivalent instruction for mov.

3. Preparing to Call strcpy

80488a2: 83 ec 08 sub $0x8,%esp

80488a5: ff 75 08 pushl 0x8(%ebp)

80488a8: 8d 45 e8 lea -0x18(%ebp),%eax

80488ab: 50 push %eax

80488ac: e8 1f f9 ff ff call 80481d0 <\_\_rel\_iplt\_end+0x28>

* **sub $0x8, %esp**: Allocate 8 more bytes of space on the stack. Padding the stack for stack alignment. Usually, this indicates a function call in the next few instructions.
* **pushl 0x8(%ebp)**: Push the argument passed to the **welcome**() function onto the stack. This is the value of argv[1], which contains the string passed from the command line. This must correspond to passing the variable name to **strcpy**() which is at ebp+8.
* **lea -0x18(%ebp), %eax**: Load the address of the buffer words (located at -0x18(%ebp)) which is ebp-24 into register eax. It stores the location, and not the value at that location.
* **push %eax**: Push the address of the buffer onto the stack (this is the destination for **strcpy**). It can be inferred that ebp-24 is the address of the local variable words.
* **call 80481d0**: Call the function **strcpy**() at 0x80481d0 as per the source code, which copies the string argument into the buffer words.

4. Preparing to Print the Welcome Message

80488b1: 83 c4 10 add $0x10,%esp

80488b4: 83 ec 04 sub $0x4,%esp

80488b7: ff 75 08 pushl 0x8(%ebp)

80488ba: 8d 45 e8 lea -0x18(%ebp),%eax

80488bd: 50 push %eax

80488be: 68 7e b2 0b 08 push $0x80bb27e

80488c3: e8 88 63 00 00 call 804ec50 <\_IO\_printf>

* **add $0x10, %esp**: Adjust the stack pointer by adding 16 to it after the **strcpy**() call.
* **sub $0x4, %esp**: Padding the stack for stack alignment.
* **pushl 0x8(%ebp)**: Push the command-line argument (again) which is at ebp+8.
* **lea -0x18(%ebp), %eax**: Load the address of the buffer words into eax this is the source for the printf call which is at ebp-24.
* **push %eax**: Push the buffer onto the stack.
* **push $0x80bb27e**: Push the address of a string likely a format string for **printf**() at 0x80bb27e.
* **call 804ec50**: Call the printf function to print the message which verifies our earlier assumptions and inferences, such as the arguments and their memory locations.

5. Checking the Canary Value (Stack Smashing Detection)

80488c8: 83 c4 10 add $0x10,%esp

80488cb: 81 7d f4 48 45 4c 4f cmpl $0x4f4c4548,-0xc(%ebp)

80488d2: 74 0a je 80488de <welcome+0x49>

* **add $0x10, %esp**: Adjust the stack pointer after the **printf**() call.
* **cmpl $0x4f4c4548, -0xc(%ebp)**: Compare the value in the stack canary with the original value 0x4f4c4548 which is at ebp-12. This checks if the canary has been altered due to a stack overflow.
* **je 80488de**: If the canary value is still intact (i.e., no stack overflow), the function continues normally.

6. Exiting on Canary Mismatch (Overflow Detected)

80488d4: 83 ec 0c sub $0xc,%esp

80488d7: 6a 01 push $0x1

80488d9: e8 02 5a 00 00 call 804e2e0 <exit>

* **sub $0xc, %esp**: Allocate more stack space. Padding the stack for stack alignment.
* **push $0x1**: Push the argument 1 onto the stack (exit code).
* **call 804e2e0**: Call the exit function, which terminates the program if a stack overflow was detected (i.e., if the canary value was altered).

7. Function Epilogue: Clean Up and Return

80488de: 90 nop

80488df: c9 leave

80488e0: c3 ret

* **nop**: No operation (placeholder instruction).
* **leave**: Restores the stack and base pointer to their previous values (mov %ebp, %esp and pop %ebp).
* **ret**: Return from the function, passing control back to the caller(pop %eax; jmp %eax).

# High-Level Overview

* The function starts by setting up a stack frame and storing a "**canary**" value **1330398536** which is equivalent to **0x4f4c4548** and in string it is **HELO**, which is used later to check for stack overflows.
* The **strcpy**() function is called to copy the command-line argument into a local buffer (words), which can potentially cause a buffer overflow.
* The function then prints a message using **printf**().
* After that, it checks if the canary value has changed (due to a buffer overflow). If the canary is altered, it exits the program.
* Finally, the function cleans up the stack and returns.
* This code is vulnerable to stack smashing due to the unsafe use of **strcpy**(), but it tries to mitigate the risk by using a canary value to detect tampering of the stack.

Manually creating the payload

The payload should be designed so that when **strcpy**() writes into words, it overflows the buffer and replaces the return address with the address of exploit(). While it's best to avoid changing other stack values, getting the original ebp value through this type of manual analysis isn't easy. However, the extra padding bytes don't impact the program's execution. To successfully call exploit(), it's important to keep the canary value unchanged.

The payload can be constructed as below :

* **12 bytes** of anything for words.
* **4 bytes** of the canary value which is in little-endian format is 0x48454c4f or HELO in ASCII.
* **8 bytes** of anything for the to replace the argc and argv values.
* **4 bytes** of the old ebp.
* **4 bytes** of address of **exploit**() in which it is again in little-endian format is 0x7c880408, from the objdump.

# Executing manually from the command line

A black screen with white text

Description automatically generatedOne such payload could be “**DipenduGhoshHELOCS23M509\x49\x58\x49\x41\x7c\x88\x04\x08**”, which can be used as in the screenshot below to trigger the exploit successfully. The snapshot of the manual execution from command line is in Figure 2.

The **printf()** is needed here from the command line directly since “\x08” would be considered as 4 characters and not a single ASCII character as interpreted by printf().

# Executing using gdb

The above analysis was done manually, by checking the stack and finding out the addresses and the bytes required at each step. We can do the same thing with a debugger like gdb which will be much easier to get the required information and execute the exploit. The gdb snapshot with a normal payload and the exploit payload is in Figure 3.

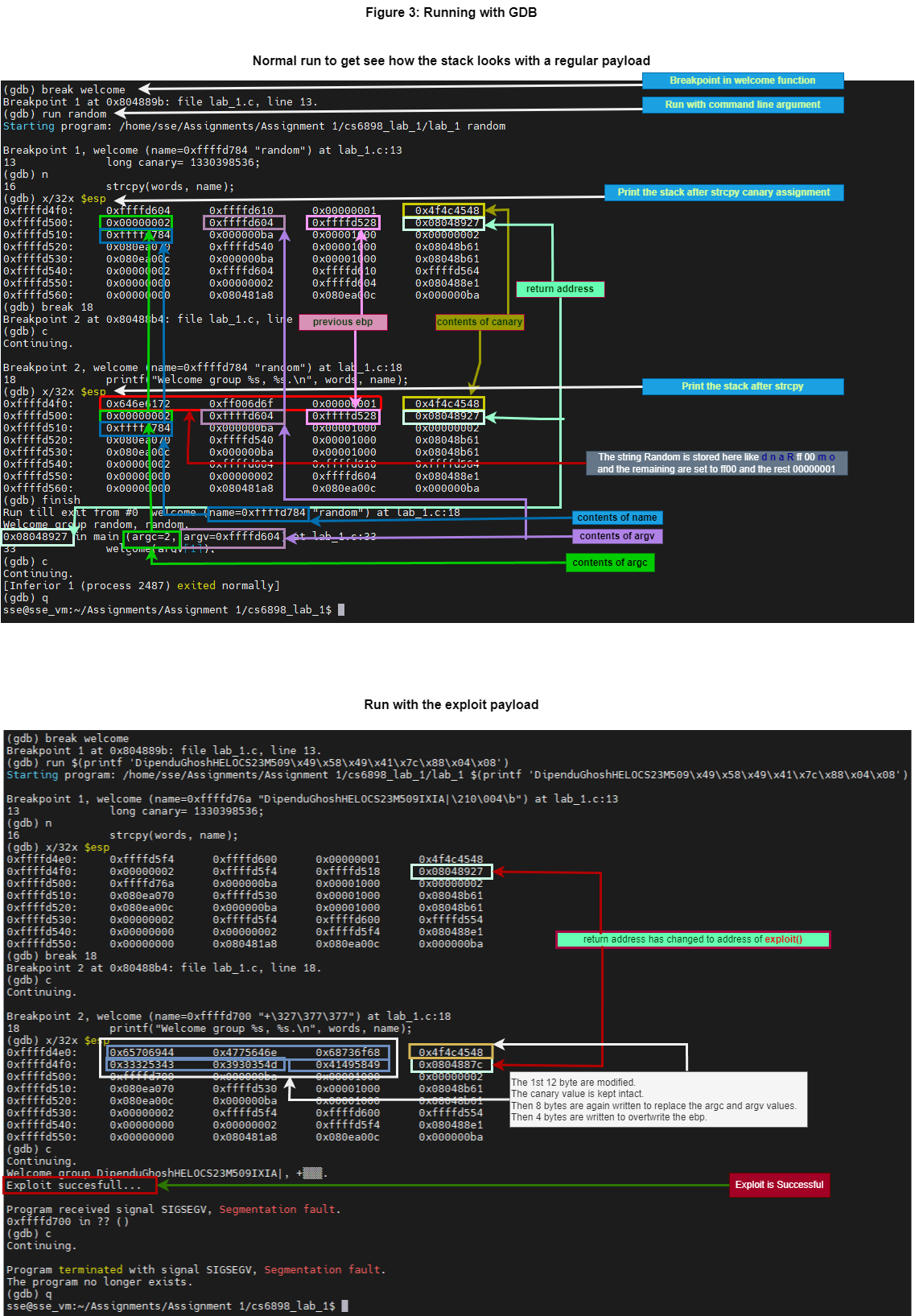
Figure 3 description:

## Normal execution with input “Random”

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Previous Value | Current Value | Comments |
| words[12] | 0xffffd604  0xffffd610  0x00000001 | 0x646e6152  0xff006d6f  0x00000001 | Value changed as per input. |
| canary | 0x4f4c4548 | 0x4f4c4548 | Remains same. |
| argc | 0x00000002 | 0x00000002 | Remains same. |
| argv | 0xffffd604 | 0xffffd604 | Remains same. |
| ebp | 0xffffd528 | 0xffffd528 | Remains same. |
| return address | 0x08048927 | 0x08048927 | Remains same. |

## Exploit execution with input “DipenduGhoshHELOCS23M509\x49\x58\x49\x41\x7c\x88\x04\x08”

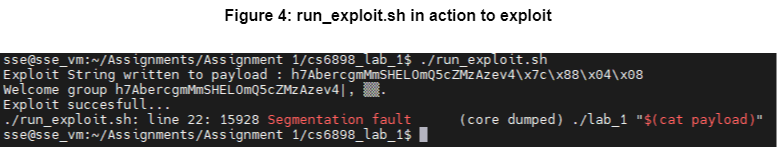
|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Previous Value | Current Value | Comments |
| words[12] | 0xffffd5f4  0xffffd600  0x00000001 | 0x65706944  0x4775646e  0x68736f68 | Value changed as per input 1st 12 bytes. |
| canary | 0x4f4c4548 | 0x4f4c4548 | Remains same else will exit. |
| argc | 0x00000002 | 0x33325343 | Value changed as per input 4 bytes after canary. |
| argv | 0xffffd5f4 | 0x3930354d | Value changed as per input 4 bytes after canary + 4 bytes. |
| ebp | 0xffffd518 | 0x41495849 | Value changed as per input 4 bytes after canary + 8 bytes. |
| return address | 0x08048927 | 0x0804887c | Return address replaced with address of exploit which is the last 4 bytes of the input. |



# Payload generation

The run\_exploit.sh is used to generate the random payload for each execution, saves it in a file named “payload” and sends it as a command line argument to the lab\_1 binary to execute the exploit. The last generated payload can be used to manually execute the binary as well by executing the command “./lab\_1 "$(cat payload)"”. Figure 4 shows the script in action to exploit the vulnerability.

## Exploit generation and execution steps:

1. It generates 3 random strings **str\_12, str\_8** and **str\_4** of **12 bytes**, **8 bytes** and **4 bytes** respectively for the words variable, argc and argv values and ebp.
2. Sets the value of **canary**, canary="**HELO**". If canary value is changed, then the if check will pass and the program with exit.
3. Sets the **exploit**() function address, **exploit\_address**="\x7c\x88\x04\x08".
4. Creates the payload by concatenating str\_12, canary, str\_8, str\_4 and exploit\_address as “**payload="$str\_12$canary$str\_8$str\_4$exploit\_address**" and writes to file named payload.
5. ****Executes the binary, **./lab\_1 "$(** **cat payload)".**

# Precautions and Counter Measures

The binary was vulnerable to exploitation due to input which is not validate and a buffer overflow caused by **strcpy**(). While the canary value check can help prevent accidental overflows, it can still be bypassed, as shown. To protect programs from such vulnerabilities, consider the following:

* Validate user input: Ensure that input strings are properly limited by inserting a NUL character at the maximum allowed length to prevent buffer overflows.
* Use safer functions: Replace **strcpy**() with **strncpy**() or **memcpy**(), which limit the number of copied bytes.
* Enable stack protection: Use the -fstack-protector flag when compiling with gcc to add stack protection and not -fno-stack-protector.
* Enable ASLR: Use Address Space Layout Randomization (ASLR) support.
* Improve canary systems: Implement stronger canary mechanisms than basic ones.

# Contents of the zip

CS23M509\_CS6898W\_Assignment\_1.zip contains the below files:

* report.pdf – The report for this assignment.
* run\_exploit.sh – A script to generate the payload and execute the lab\_1 binary executable.
* lab\_1 – The provided executable.
* payload – the last generated payload.