NETWORK SECURITY LAB

Testing Integer Overflow and Buffer Overflow Using GDB

SAMEER'S LAB

CYBER SECURITY

Introduction

This report focuses on identifying and exploiting software vulnerabilities, specifically integer overflow and buffer overflow, using the GNU Debugger (GDB). These vulnerabilities, if left unchecked, can lead to software crashes, data corruption, and even security breaches. This lab demonstrates how to exploit these issues and provides insight into mitigating them to promote secure coding practices.

Key Points

- **Tools Used:** "***GNU Debugger (GDB)"*** and "***GCC (GNU Compiler Collection)".***
- **Concepts Explored:** Integer overflow, buffer overflow, debugging, and mitigation techniques.
- **Steps Performed:** Writing vulnerable code, testing for vulnerabilities, exploiting the weaknesses, and implementing fixes.

Purpose of This Lab

The primary purpose of this lab is to:

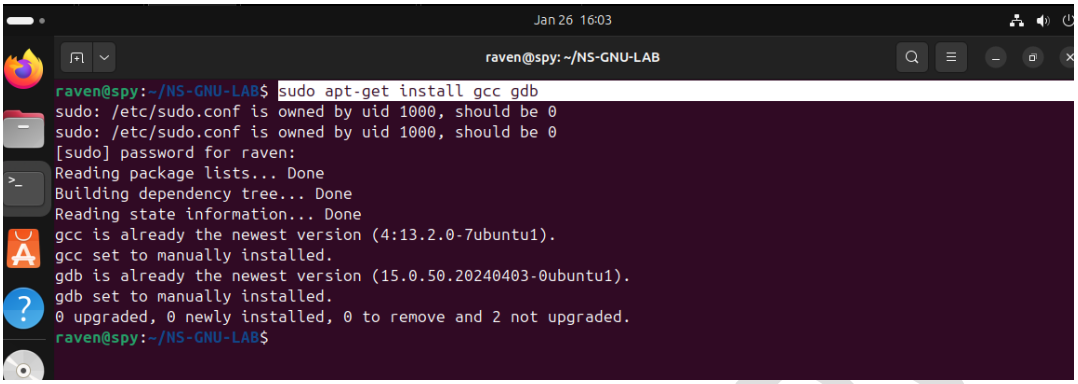
- Gain a deep understanding of integer overflow and buffer overflow vulnerabilities.
- Learn how to exploit these issues using GDB to observe their impact.
- Practice secure coding techniques to prevent such vulnerabilities.

Prerequisites

Ensure the following are installed:

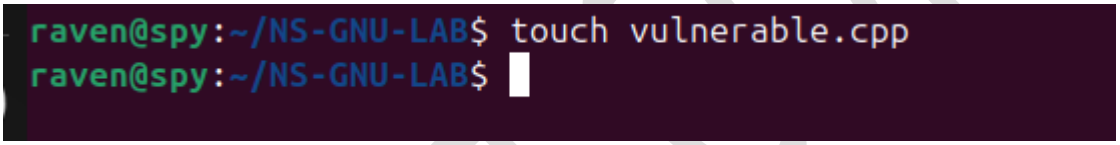
1. Linux system (Ubuntu).
2. **GCC (GNU Compiler Collection):** "***sudo apt-get install gcc"***.
3. **GDB (GNU Debugger):** "***sudo apt-get install gdb***".

1. Installing *"GCC & GDB".*

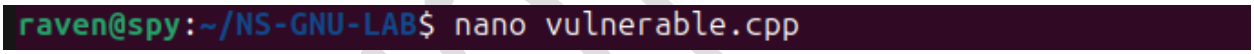


2. Create the Vulnerable Program

- **Open a text editor and create a file named *"vulnerable.cpp"* using command "*touch*".**



- **Writing code into the file "*vulnerable.cpp".***



```
    } else {
        printf("Number is safe: %d\n", number);
    }

    // Call the vulnerable function
    printf("Enter a string: ");
    scanf("%s", input);
    vulnerable_function(input);

    return 0;
}
```

3. Compile the Program with Debugging Symbols

- ***"-g"*** **enables debugging information for GDB.**

```
raven@spy:~/NS-GNU-LAB$ gcc -g vulnerable.cpp -o vulnerable
```

4. Launching GDB

```
raven@spy:~/NS-GNU-LAB$ gdb ./vulnerable
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vulnerable...
```

5. Test Integer Overflow

- **Set a breakpoint at the main function** ***"break main"*****.**

```
(gdb) break main
Breakpoint 1 at 0x555555555219: file vulnerable.cpp, line 11.
(gdb)
```

- **Run the program**:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/raven/NS-GNU-LAB/vulnerable hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at vulnerable.cpp:11
11      int main() {
(gdb)
```

- **"step" to see the next step of code. Helps to understand next step.**

```
(gdb) step
16          printf("Enter a number: ");
(gdb)
```

- **Command: *"(gdb) c"***
  - You issued the continue (or c) command in GDB.
  - This resumes the execution of your program from where it had paused (at a breakpoint or the start of main()).

```
(gdb) c
Continuing.
Enter a number:
```

- **Enter a large number (e.g., 2147483647) when prompted.**
  - Observe if an integer overflow message appears.
  - **Since *"2147483647" does not cause an overflow*** in this specific check, the program correctly outputs "***Number is safe***" along with the value.

```
Enter a number: 2147483647
Number is safe: 2147483647
Enter a string:
```

- **Program Output:**
  - The program prompts you to enter a string. You entered a string of "***19 A"*** characters.
  - The input string is passed to the ***"vulnerable_function"***, which uses ***strcpy()*** to copy the input into a smaller buffer (***char buffer[10];),*** creating a buffer overflow.

```
Enter a string: AAAAAAAAAAAAAAAAAAA

Breakpoint 2, vulnerable_function (input=0x7fffffffdcb0 'A' <repeats 19 times>) at vulnerable.cpp:5
5       void vulnerable_function(char *input) {
(gdb)
```

- **Breakpoint:**

The program hits the breakpoint you set in the ***"vulnerable_function."***

Details of the breakpoint:

- ***input=0x7fffffffdcb0:*** This is the memory address of the input variable, which points to your input string.
- ***'A' <repeats 19 times>:*** The input string contains 19 A characters as you entered.

- ***"5 Line of Execution:"***
  - GDB is currently paused at the start of the ***"vulnerable_function"*** (line 5 in vulnerable.cpp).
  - The function will attempt to copy the string input into a buffer (***buffer[10]),*** which is too small to hold the **19 characters** you entered.

➢ **What's Happening in Memory?**

Buffer Overflow:

- The ***strcpy()*** function copies all **19 characters** of input into the buffer, which has space for only **10 characters**.
- This causes a buffer overflow, where the extra characters (**AAAAAAAAA**) overwrite adjacent memory, potentially leading to undefined behavior.

❖ **Next Steps in GDB:**

You can explore the state of the program to understand the overflow better:

- **Inspect the Variables: *"print buffer".* This will show the contents of buffer.**

```
(gdb) print buffer
$1 = "\000\000\000\b\000\000\000\000\000"
(gdb)
```

- ***"Print input"* This shows the full string stored in input.**

```
(gdb) print input
$2 = 0x7fffffffdcb0 'A' <repeats 19 times>
(gdb)
```

- **Inspect Memory: *"x/20x &buffer"* This shows the memory content starting from buffer, so you can see how it was overwritten.**

```
(gdb) x/20x &buffer
0x7fffffffdc7e: 0x08000000      0x00000000      0x00000000      0x00000060
0x7fffffffdc8e: 0xdd200000      0x7fffffff      0x52ab0000      0x55555555
0x7fffffffdc9e: 0x00000000      0x00000080      0x00000000      0xffff0014
0x7fffffffdcae: 0x41417fff      0x41414141      0x41414141      0x41414141
0x7fffffffdcbe: 0x41414141      0x008c0041      0x00000000      0x00000000
(gdb)
```

- **Step Through the Code:** ***"next as n"*** **This moves to the next line in the** ***"vulnerable_function"*** **and** ***"executes strcpy()".***

```
(gdb) next
A syntax error in expression, near the end of `
'.
(gdb) n
7           strcpy(buffer, input);  // Unsafe function usage, can cause buffer overflow
(gdb)
```

- **Investigate Further:** If you want to see how the overflow affects program behavior, you can let the program continue running: **"continue"**

6. Fix the Vulnerabilities

- Replace ***"strcpy"*** with "***strncpy"*** in the code and add bounds checks for integer input.

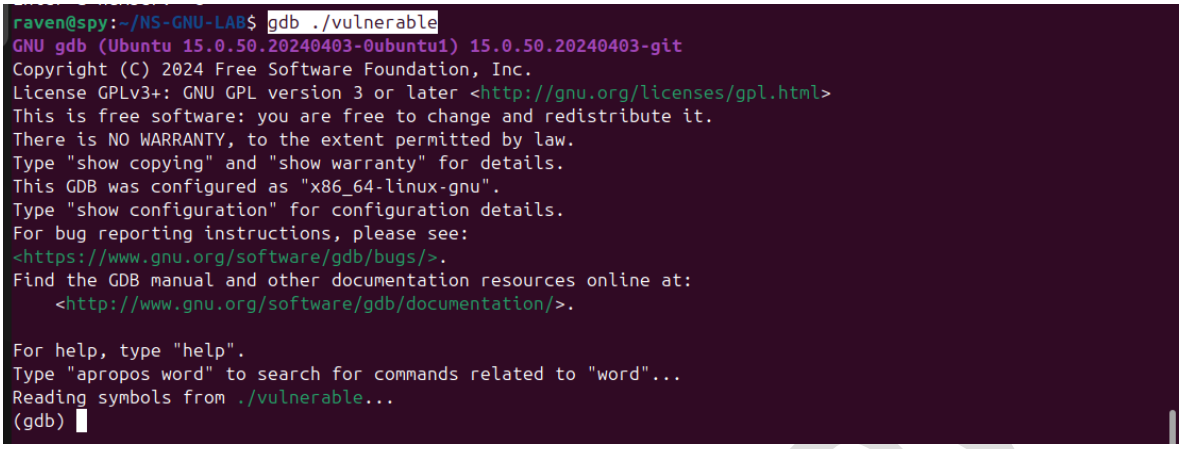
Explanation of the Fix:

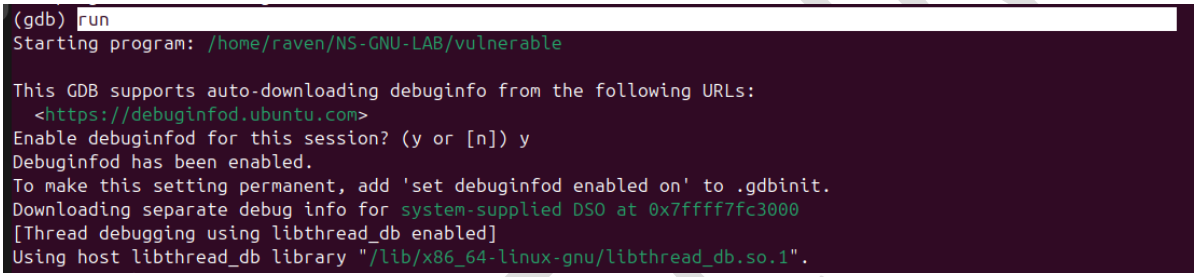
1. **strncpy(buffer, input, sizeof(buffer) - 1):**
   - This copies at most ***sizeof(buffer)*** - 1 characters into buffer, leaving space for the null terminator.
2. **buffer[sizeof(buffer) - 1] = '\0':**
   - Ensures the string in buffer is always **null-terminated**, even if the source string is longer than the buffer.

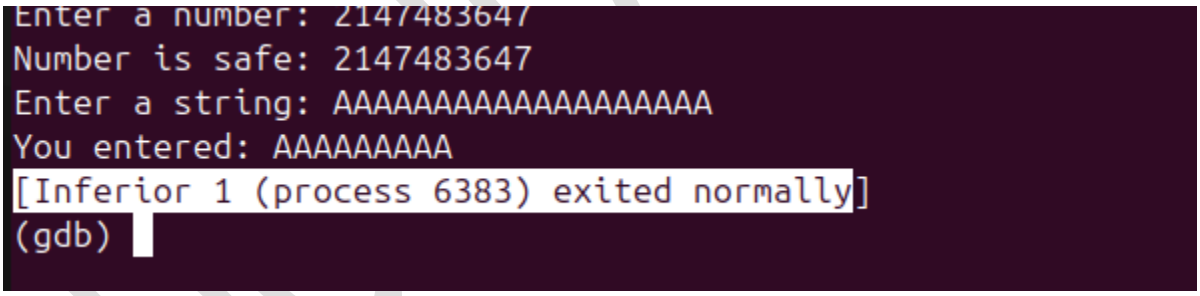
- **Recompile the program *"gcc -g vulnerable.c -o vulnerable"***

```
raven@spy:~/NS-GNU-LAB$ gcc -g vulnerable.cpp -o vulnerable
raven@spy:~/NS-GNU-LAB$
```

- **Run the program:*"gdb ./vulnerable"***







❖ Output Explanation

- **[Inferior 1 (process 6383) exited normally]**

After executing the program, the process completes without crashing or errors. This message indicates:

- The program has finished execution successfully.
- No segmentation fault or other runtime errors occurred, thanks to the fixed "***strncpy***" and proper "***null termination.c".***

❖ **Why Is the Output Truncated?**

- The buffer in the "***vulnerable_function***" has a fixed size of ***10 bytes***.
- To prevent overflow, ***"strncpy"*** copies only the first ***9 characters*** into buffer and reserves the last byte for the null terminator (**\0**), ensuring the string fits safely.
- Since the input string is longer (**19 characters**), only the first 9 characters (**AAAAAAAAA**) are displayed, while the rest are ignored.

➢ **Final Output Summary:**

1. **Number Handling:**
   - The program safely processes the integer **2147483647** without detecting overflow.
2. **String Handling:**
   - The input string **AAAAAAAAAAAAAAAAAA** is safely truncated to **AAAAAAAAA** to fit into the **10-byte buffer**, avoiding buffer overflow.
3. **Successful Execution:**
   - The program exits normally, demonstrating that the vulnerabilities (integer overflow and buffer overflow) have been effectively mitigated.

❖ **What I Learned from This Lab**

1. **Integer Overflow:**
   How providing an excessively large integer causes arithmetic operations to behave unpredictably due to exceeding memory limits.
2. **Buffer Overflow**:
   How input data exceeding allocated buffer size can overwrite adjacent memory, leading to crashes or malicious code execution.
3. **Debugging Skills:**
   Gained hands-on experience with **GDB** for setting breakpoints, stepping through code, and observing variable states.
4. **Mitigation Techniques:**
   Implementing safeguards like bounds checking and secure functions (e.g., "**strncpy"** instead of "**strcpy**").

❖ **Exploiting and Mitigating Integer and Buffer Overflow Using GDB**

1. **Exploitation:**

- **Integer Overflow:** Inputting a very large number (e.g**., 2,147,483,647**) revealed how the system failed to handle overflow properly.
- **Buffer Overflow:** Entering a long string (e.g., **AAAAAAAAAAAAAAAAAA**) overwrote the memory buffer, triggering a crash.

2. **Mitigation:**

- For integer overflow, additional conditions were added to validate the input range.
- For buffer overflow, the ***"strncpy"*** function was used to limit input length, ensuring it remained within the buffer size.

❖ **Understanding the Importance of Secure Coding Practices**

This lab highlighted the critical role of secure coding in software development:

- Ensuring proper bounds checking and input validation prevents vulnerabilities.
- Using secure functions, like **"strncpy"**, reduces the risk of overflows.
- Regular testing and debugging with tools like GDB help identify potential risks early in development.

Summary

This lab provided practical experience with debugging tools to identify and address common software vulnerabilities. By intentionally creating a vulnerable program and exploiting it, the significance of secure coding practices became evident. These practices are essential for developing robust and secure applications, ensuring software integrity, and protecting user data.