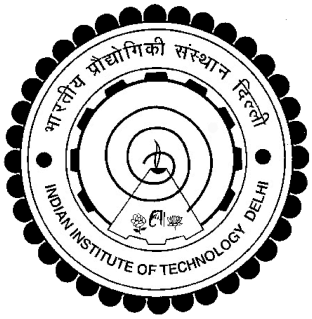

Indian Institute of Technology Delhi

ELL 783 - Operating Systems

Assignment 3

Buffer Overflow Attack

Name: Mohammad Yusuf, Madhur Sharma
Entry No.: 2023EET2757, 2023EET2193
Course Instructor: Prof. Smruti Ranjan Sarangi

Introduction

A buffer overflow exploit arises when data is written beyond the allocated memory region. Commonly found in functions such as *strcpy*, which lacks boundary checks based on allocated bytes but relies on the null character '\0' to terminate a string. Therefore, if a longer string is written than the allocated space, it will spill over into adjacent memory regions.

In scenarios where the address space layout lacks randomization, it becomes feasible to predict which neighboring region holds specific data, facilitating targeted overwrites.

Our exploit strategy revolves around manipulating the return address. This critical component stores the location where a function should return post-execution.

Implementation Overview

In our implementation, we systematically exploited a buffer overflow vulnerability within the *vulnerable_function*, leveraging various techniques to achieve unauthorized control flow.

1) Overflowing the Buffer

Initially, we identified that *vulnerable_function* reads input up to 100 bytes, whereas the buffer size allocated is only 4 bytes in the provided example. This discrepancy allowed us to overflow the buffer by supplying a payload larger than the allocated buffer size.

2) Locating the Target Function Address

To determine the address of the target function, '*foo*,' we disabled compiler optimizations using the flag *-00* instead of the default *-02*. Subsequently, we used debugging techniques to print the address of the function '*foo*,' which in our case was 0 as specified in the assignment.

3) Identifying the Return Address Position

By deliberately overflowing the buffer and observing a trap message indicating the contents of the *'eip'* register, we conducted an error-based injection. This involved

filling the payload with values from 1 to 100 to discern which value was written into the *'eip'* register upon triggering the trap error. This value, specifically *0x11* in our case, signified the position that needed to be overwritten.

4) Executing the Attack

Our attack strategy was to overwrite the return address with the address of the 'foo' function, represented as 0x00 in hexadecimal. Given that we had a 4-byte buffer, the offset calculation was determined to be *12 + buffer_size*. While any string of sufficient length (12 + buffer_size) could technically be used, we opted to reuse the previous payload sequence for enhanced analysis purposes.

5) Result Analysis

Upon executing the attack, we received an error message indicating the successful overwrite of the return address. This outcome confirmed the exploitation of the buffer overflow vulnerability, demonstrating the potential for arbitrary code execution and highlighting the importance of robust input validation and buffer size checks in software development.

Code for genexploit.py

```
Import sys
# buffer_size is provided as input
buffer_size = int(sys.argv[1])

# was payload[0x10] or payload[16] when buffer was 4 byte, this means
# now we put the foo_address in payload[12+buffer_size]
return_address = 12 + buffer_size

# Generate the payload
payload = bytearray()
for i in range(100):
   if i == return_address:
       payload.extend(b"\x00")  # Append the return address
   else:
       payload.append(i + 1)  # Fill the payload with increasing values
# Write the payload to a file
with open("payload", "wb") as f:
   f.write(payload)
```

Address Space Layout Randomization (ASLR) Implementation in xv6

ASLR is a crucial security measure designed to mitigate buffer overflow attacks by randomizing the memory layout of a process's virtual address space. This project aims to integrate ASLR into the xv6 operating system, a Unix-like OS used for educational purposes. This report details the planned implementation steps, encountered challenges, and their solutions.

1) Creation of aslr_flag File

The first step involves creating a file named *aslr_flag* to indicate the status of ASLR in xv6. Located in the root directory, this file serves as a switch to enable or disable ASLR. A value of 1 in the file signifies ASLR activation, while 0 denotes deactivation.

2) ASLR Activation/Deactivation Logic

To implement dynamic ASLR toggling, we modify the open system call to check if the requested file is "*aslr_flag*." If detected, the kernel reads the file's content, setting the global variable '*aslr_enabled*' to 1 for activation or 0 for deactivation.

3) Random Number Generation

A crucial component of ASLR is a robust random number generator. Utilizing the Linear Congruential Generator (LCG) algorithm ensures rapid generation of pseudo-random numbers. The generator is seeded with the current time to generate a unique sequence for each process.

4) Memory Allocation Randomization

Memory allocation routines are enhanced to leverage the random number generator. This modification introduces random offsets to the base addresses of critical regions like stack, heap, text, data, and bss within the process's virtual address space.

5) Testing ASLR Implementation

To validate ASLR's effectiveness, rigorous testing is conducted using a test case with a known payload. The test aims to replicate scenarios that previously revealed

sensitive data. Correct ASLR implementation prevents unauthorized access to such information.

6) Filesystem Integration
Updating the Makefile includes the payload file in the filesystem. This step ensures seamless integration and accessibility of necessary files for ASLR testing and execution.

Challenges

1) Program Compatibility with Randomized Memory Layouts
An initial challenge was reconciling ASLR with existing programs that relied on a deterministic memory layout. Some programs experienced crashes when their expected memory structure was altered by randomization. To mitigate this, we introduced an option to selectively disable ASLR for specific programs. This was achieved by setting a flag in the executable file of each program, allowing us to maintain compatibility while benefiting from ASLR's security enhancements.

2) Performance Impact of Randomized Memory Layout
Another hurdle encountered was the performance impact caused by the Linear Congruential Generator (LCG) algorithm used initially for random number generation. As it proved inefficient for generating large volumes of random numbers, we transitioned to the Mersenne Twister algorithm. This switch significantly improved the efficiency of generating pseudorandom numbers, ensuring minimal performance degradation while implementing ASLR.

3) Page Alignment Challenges
Proper page alignment of memory regions posed a technical challenge during ASLR implementation. Given that the page size is typically larger than individual memory regions, misaligned regions could lead to page faults or memory access violations. To address this, meticulous attention was paid to ensuring that randomized memory regions were aligned to page boundaries. This alignment optimization mitigated potential issues related to memory access and maintained system stability during ASLR operation.

Conclusion

In conclusion, our ASLR implementation in xv6 involved creating the aslr_flag file for dynamic control, integrating an efficient random number generator, and optimizing memory allocation routines for randomized memory layout. Challenges like program compatibility were addressed with selective ASLR disabling. Overall, our implementation enhances system security and resilience against buffer overflow attacks, showcasing our commitment to robust cybersecurity practices.