Cyber Security Project: Multi-Stage Attack on a Linux Server

Jan Büchele
Henrik Lümkemann

June 19, 2025

1 Introduction

Over the course of this Programming project, we looked at two severe Linux vulnerabilities. We demonstrate a multi-stage attack on a Linux server, using these exploitable security issues, and propose a way to manually patch them as well. This report will show and document our work and explain our results and achievements.

2 Problem Statement

Server systems, even when configured with layered defenses, can be compromised by combining seemingly unrelated vulnerabilities. An attacker can exploit a flaw in a single network-facing service to gain initial access and subsequently use a separate kernel-level vulnerability to escalate privileges, leading to a complete system compromise. This project will demonstrate the risk of such multistage attacks by exploiting a libssh authentication bypass to gain remote access to an Ubuntu 16.04 server, followed by the Dirty COW privilege escalation vulnerability to gain root access on said server.

For this project, the application scenario is as follows:
A company has set up a local Ubuntu 16.04 server for its employees to connect to and have their own work space where they can save their data. For employees to be able to connect to the server remotely, an ssh server using libssh has been set up. In general employees can also access company data on this server, however for more private data, they are only able to read it themselves. Some data that is stored on the server can only be read by users with root privileges (higher-ups in the company).

We can face different types of adversaries:

External Adversaries: These are unauthorized individuals or groups operating outside a network. Their goal is typically to compromise as many systems as possible. For them, the libssh vulnerability is a perfect entry point, it is remotely exploitable, and it does not require any prior knowledge of the system or its users.

Threat: Once inside, they could use the Dirty COW exploit to gain root privileges. With full control, they could steal or encrypt sensitive data for ransom, install crypto miners, or simply destroy the system's data.

Malicious Insiders: This could be an employee of the company. Their starting point is different from the external adversaries, since they already have some level of access to the server.

Threat: While the libssh vulnerability might not be necessary for them, the Dirty COW exploit is the real threat here. It allows them to elevate their limited privileges to full root access, bypassing all internal access controls. This would enable them to access confidential company data, sabotage systems, or cover their tracks by altering logs.

2.1 Libssh Exploit

The libssh Authentication Bypass vulnerability (CVE-2018-10933) is a critical vulnerability in the libssh library. SSH functionality, which is provided by this library, enables clients to connect to a system remotely if you know the system's IP address, as well as the username and password for a valid user account on that system. Under normal circumstances, password or public key authentication by incoming client requests is strictly required. However, this is not the case in versions 0.6 to 0.75 of the libssh library, posing a critical security issue.

Normally, a client first sends a **SSH2_MSG_USERAUTH_REQUEST** to the Linux server to begin the authentication process. The server then proceeds with the configured authentication methods, such as asking for a password or a valid public key. However, to avoid authentication and still gain access to the server, the client sends a **SSH2_MSG_USERAUTH_SUCCESS** message as an initial message to the server instead. In vulnerable versions, the libssh library interprets this message as a signal that the client has already been successfully authenticated. The client can gain access to any system using the libssh library without presenting any valid credentials, which is a severe security flaw, possibly enabling attackers to exploit the system even further after gaining access.

The severity of the libssh vulnerability was assigned a ranking of 9.1 out of 10 in the CVSS (Common Vulnerability Scoring System) due to the following factors:

- There is no authentication required. Valid credentials are not needed.
- The vulnerability can be exploited over the network and does not require physical presence of an attacker.
- The vulnerability is relatively easy to exploit, making it more accessible to a wider range of possible attackers.
- There is the possibility that attackers have access to a shell on the compromised system, which could lead to a complete compromise of a system.

2.2 Dirty COW Exploit

Dirty Copy-On-Write (COW) is a vulnerability (CVE-2016-5195) affecting the Linux Kernel (versions 2.6.22 - 4.8.3). DirtyCOW exploits a problem in the way the Linux Kernel handles memory management, through this an attacker can use a race condition to escalate their privilege and gain root access to a system.
As already seen in the name, the fault lies in the implementation of the Copy-On-Write Mechanism. This fault allows an attacker to write to any file they can read, even if it is owned by the root user or marked as read-only. COW is used to efficiently manage memory, when a process needs a copy of a memory page, initially the process only receives a reference to the original page and marks it as read-only. Only when the process attempts to write to the page, then a private copy is created.
This is where Dirty COW exploits the race condition. The exploit works by using two threads:

- Thread 1 (Writer): This thread continuously tries to write to a private memory mapping of a read-only, root owned file. This triggers the COW mechanism again and again.
- Thread 2 (Discarder): This thread tells the kernel to discard that same memory mapping using the `madvise` system call.

Now the exploit will run both of these threads at the same time and when the timing is perfect, the kernel will approve of the write operation, but

before the write completes, the discarder thread frees the private copy. The kernel, already committed to the write, will then mistakenly write the attacker's data to the original read-only file.
Usually the file that is written to is `/etc/passwd`, where the attacker can then edit the user ID to get root privileges or remove the root user's password entirely.
The severity of the Dirty COW vulnerability was assigned a rating of 7.8 out of 10 in the CVSS due to the following factors:

- The attacker must have local access to the system to run the exploit code. It cannot be exploited remotely.
- The exploit is very easy to execute. There is well documented proof-of-concept code wide available and it doesn't require any special circumstances to succeed.
- The exploit provides full root access, meaning the attacker can read all data on the entire system.
- With root access, the attacker also has complete control over any file on the system, by being able to modify or delete them.

3 Linux Server Setup

This section gives information about how we designed the server and how it can be installed.

3.1 Instruction on installing the Server

For reproducibility please follow the instructions below.

The first step is the Linux-VM setup, for which we chose version 16.04-desktop `https://old-releases.ubuntu.com/releases/16.04.4/`.

The first step, after installing the VM, is to install some libraries we need to compile libssh later on:

```
sudo apt-get install -y \
    build-essential \
    cmake \
    libssl-dev \
    zlib1g-dev \
    libgcrypt20-dev \
    libkrb5-dev \
    libcurl4-openssl-dev \
    pkg-config
```

After installing these libraries, we need to download libssh, This can be done with:

```
wget https://www.libssh.org/files/0.7/libssh-0.7.4.tar.xz
```

Next we extract the files inside the tar with:

```
tar -xvf libssh-0.7.4.tar.xz
```

Inside the newly created folder we manually create the build directory and set the working directory to this newly created folder. Now we need to update the server file, the CMakeLists.txt file and add the patched server file inside the examples folder. To do this update them with the samplesshd-cb.c, CMakeLists.txt and samplesshd-cb-patched.c files from this git repository `https://github.com/jabue2/Cyber_Security_Project`.

Before we run the server we first need to create some keys, this can be done by running the following commands:

```
sudo ssh-keygen -t dsa -m PEM -b 1024 \
     -f /etc/ssh/ssh_host_dsa_key -N ""
sudo ssh-keygen -t rsa -m PEM -b 2048 \
     -f /etc/ssh/ssh_host_rsa_key -N ""
sudo chmod 600 /etc/ssh/ssh_host_dsa_key /etc/ssh/
   ssh_host_rsa_key
```

Lastly we need to create a new user, that only has privileges in a certain directory, otherwise they only have read privileges. This can be done with:

```
sudo adduser --disabled-password --shell /bin/sh sandbox
sudo mkdir -p /srv/sandbox-work
sudo chown sandbox:sandbox /srv/sandbox-work
sudo chmod 0750 /srv/sandbox-work
```

This is supposed to mimic how an employee would have their own working directory, where only they and the root user have privileges.

Now we can compile the server file, we do this by setting the working directory of the terminal to the build folder we created earlier. Then we first run these three commands in succession to install it:

```
cmake .. \
  -DWITH_SERVER=ON \
  -DBUILD_EXAMPLES=ON \
  -DCMAKE_INSTALL_PREFIX=/usr/local/libssh-0.7.4
make -j$(nproc)
sudo make install
```

When the compilation and installation is done, we can start the server using:

```
sudo ./examples/samplesshd-cb -p 2222
```

Now the server is online and a user can connect to it anytime.

3.2 Server Code Explanation

We adjusted the existing example server of the libssh library to feature a bit more functionality:

- First, we added the possibility to create a shell. This was crucial, since we needed a shell to perform our chained attack with Dirty-COW. Before the function `pty-request` and `shell-request` were only placeholders, we fully implemented them.
- We used the select() system call to implement an interactive shell, that has two-way communication. That way we are able to see the results of commands sent on the client.
- To mimic an employee of the company, we use a hard coded user named "sandbox" and changes the user ID of the terminal to this user. We also redirect the starting directory of the shell to a specified directory where they have full access to and can read write files, however outside of this the "sandbox" user is low-privilege.
- We hard coded the server to only accept one connection at a time, to make it easier to implement. Also once a connection ends, the server stops as well.

4 Exploit Plan

In order to exploit the two vulnerabilities we have implemented a python script that handles most of it.

4.1 Exploit libssh

The main task of the python script we developed is to establish a connection to the server, by bypassing libssh. After doing that it's only task is to handle the shell.
First we establish a TCP connection to the server and with the library `paramiko` we start the SSH transport layer. Then instead of sending an authentication request, we construct and send a raw `SSH_MSG_USERAUTH_SUCCESS` message. Since the libssh server is vulnerable to the exploit it accepts the message and places the session into an authenticated state.
Once the server is tricked, the script simply proceeds to open a session channel and request an interactive shell, which the server then grants.

4.2 Exploit Dirty COW

For exploiting Dirty COW we used an already existing exploit, it can be found here `https://github.com/thaddeuspearson/Understanding_DirtyCOW`. In order to replicate this, first we calculate the offset of the UID of our user inside the `/etc/passwd` file. We can do this by running:

```
cat /etc/passwd | grep -b sandbox
```

Then we count the number of characters in the response between the beginning of the username and the second colon.

```
2244:sandbox:x:1001:1001:,,,:/home/userx:/bin/bash
```

Here in this case it would be 10 characters, thus our offset is 2254.
Now we edit our exploit script to target `/etc/passwd`, the target content is 0000 (for root) and the target offset is what we've just calculated. We then exit nano and compile the c file with:

```
gcc -o exploit exploit.c -lpthread
```

Lastly we execute the file and wait for a few seconds until we manually interrupt it. If we then run `tail -n 1 /etc/passwd` and see that the sandbox now hass a UID of 0000, then the exploit worked and we just need to use `su sandbox` to gain accesstoot privileges (potentially the client needs to exit the connection and recreconnect again).

5 Patching the Exploits

5.1 Patching libssh bypass

Patching the libssh bypass can be done by adding a few lines of code to the Linux server code. Currently, when a user connects normally to the ssh server, they are asked for a password, this is what we make use of. We create a new global variable `authenticated`, this variable is 0 when no correct or no password at all have been entered, and when the correct password is entered it changes to 1. Then in the main script where we check if the shell on the server has been opened yet, we add an additional check regarding this new variable. We simply check if it is set to 1, if not the program will be stuck forever. While the user theoretically has connected successfully to the server, in the server code the connection is not yet authenticated. The connection is only authenticated via the libssh library, so it is like adding a two layer authentication.

5.2 Patching Dirty COW Exploit

Since Dirty COW is a kernel-level exploit, there is no easy solution like changing a few lines of code. Instead of applying a patch on the Linux

kernel, there is the possibility of hardening the Linux server to try and prevent the exploit.

- First, a possibility is to remove development tools and compilers (e.g., gcc). This way, an attacker is not able to just download and compile the exploit code right on the server, and therefore prevent the exploit. However, this is only a partial mitigation, as an attacker can compile the code on their own machine and then upload it to the server, if possible. If this company server only allows the creation of files on the server and not via upload, then this would essentially mitigate the exploit.

- A more robust mitigation would be to enforce stricter file system policies. By mounting all user-writable directories as non-executable, it would be impossible for an attacker to execute any code, therefore mitigating the exploit.

These hardening techniques are able to mitigate Dirty COW, however, this comes at the cost of a less flexible server, where you could for example not execute any files anymore, or compile any files. Although this works, it is definitely not an optimal solution.

6 Group Work Splits

We strictly worked together on all coding tasks, however we wrote different aspects of the Report.
Jan primarily wrote about the Dirty-COW exploit, and Henrik about the libssh exploit.