

Breaking SU With Parallelism

Mayank Patel
Clemson University
Clemson, South Carolina
mdpatel@g.clemson.edu

Nathan Jones
Clemson University
Clemson, South Carolina
najones@g.clemson.edu

Josh DeWitt
Clemson University
Clemson, South Carolina
jldewit@g.clemson.edu

ABSTRACT

Two well-known cybersecurity principles are the Principle of Least Privilege and the idea of minimizing the attack surface. A system is highly vulnerable to insiders without the former, while the latter allows both internal and external vulnerabilities. In this project, we take advantage of an insider's privilege and a vulnerability of the built-in Linux command *su* [1], by completely bypassing its cooldown timer to conduct a brute-force password attack.

1 INTRODUCTION

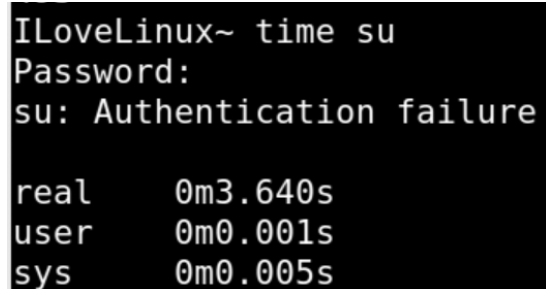
Our project explores a vulnerability in the *su* command in the Linux operating system. *Su* is a command that stands for substitute user, and allows one user to essentially login and run commands as another user, provided they know that other user's password. Suppose a user can use *su* without the necessary password. In that case, they can conduct any amount of malicious activity under another user's name and could potentially even gain access to a privileged or root account.

2 BACKGROUND

Most password interfaces implement simple protection against brute-force attacks known as a cooldown, where the user must wait a few seconds after an incorrect password attempt to try again. This time is usually unremarkable for a human, but for a machine attempting to brute-force a password, that time quickly pushes brute-force out of the feasible space.

This cooldown mechanism is where the *su* vulnerability we exploit lies. The cooldown for a *su* attempt is around 3 seconds but is local to the process that was attempted. In other

words, all a user needs to make x *su*-attempts in 3 seconds is to run x concurrent processes.



```
ILoveLinux~ time su
Password:
su: Authentication failure

real    0m3.640s
user    0m0.001s
sys     0m0.005s
```

Figure 1: *su* cooldown mechanism takes ~3 seconds

Due to the cross-distribution presence of the *su* command, the vulnerability we target is similarly omnipresent. The process cooldown vulnerability is relatively well known in the security community to the point that Kali Linux comes standard with a utility written in C designed to take advantage of it named Sucrack [2]. While looking into different versions of the same attack, the majority of the versions we found were written in either C or Python. Another attack conducted by Polop and Aron [3], brute-forces a user using *su*, and for the password, it tries some passwords related to the user and from the list of thousands of common passwords. The program was written in bash script. In their program, the user can configure timeout between *su* tries and sleep time every two *su* tries.

3 MOTIVATION AND OBJECTIVES

As previously mentioned, *su* is standard across Linux distributions, along with its vulnerabilities. As such, we wanted to build a utility that could assist system administrators and users with building solid passwords that would therefore secure each account even in the

case of exploiting the multi-process vulnerability.

Since the other versions of the utility we found were all in either C or Python, we decided to build ours in C++.

We did this for two reasons:

1. To take advantage of the speed of C compared to python.
2. To take advantage of the more object-oriented aspects of C++ and the native support for specific data types that would have caused a more complex development process had it been written in C.

To meet these objectives, our program has both a password cracking mode and a password analysis mode, described further in the next section.

4 METHODOLOGY

Our project has two main functions. First is a brute-force attack on the *su* user's password. Second is a password analysis that estimates how long a password or list of passwords would take to crack.

The main obstacle to the brute-force attack is *su*'s defensive timeout. Whenever a user inputs an incorrect password, *su* will freeze for around 3 seconds to hamper brute-force attacks. To avoid this mechanism, we wrap our *su* attempt in a bash script called *suprobe.sh*, which we run concurrently in the background. Because the *su* cooldown is not global, the defensive timeout will occur in the background while the rest of our *su* attempts continue. With this approach, we could just run all of our attempts sequentially. As a bonus, when our program launches a bash script in the background, that process becomes owned by the root, allowing it to bypass the process limits of our unprivileged user.

For this reason, we attempted (and failed) to implement a processing cap to avoid an unintentional fork bomb – this will be further discussed in future work. Finally, when one of our *suprobes* is successful, it writes the

password into the file *su.txt*. Our parent process periodically checks to see if *su.txt* has been updated; therefore, it exits once the correct password has been found.

The other major design decision we made for our brute-force approach is the order in which to try passwords. We decided to use a hybrid of dictionary and brute-force methods to quickly defeat easy passwords while still guaranteeing that our program will eventually crack any (alphanumeric) password. Our order works as follows: first, our parent program attempts each password in a provided dictionary of common passwords. We use the list of the 10,000 most common passwords found on Wikipedia and Daniel Miessler's GitHub[4]. If none of them work, it starts with the string "", adds 'a', and dials up through [a-z, A-Z, 0-9] before adding a second character. This method ensures that our program can reach every possible alphanumeric string¹.

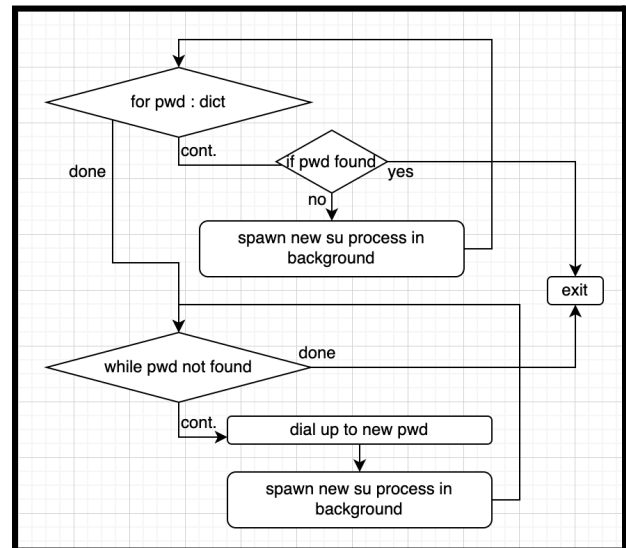


Figure 2: Design

The next important function of our program is the password analysis function.²

¹ Mayank's Graduate Functionality Extension for this project was to create the hash table, store each tried password in the hash table, and print those passwords in a separate file called *Tried_Passwords.txt*. Also, count the tries and print it in *su.txt*.

² Nathan's Graduate Functionality Extension for this project was allowing the program to handle as many passwords to analyze as the user decides to supply as command line arguments.

During initialization, the parent program stores the provided dictionary in a hash table for an optimized lookup time of $O(1)$. If a password to test is in the dictionary, and then its index tells us exactly how many *su* attempts would precede it during the cracking process. If not, we can quickly calculate how many attempts our dial-up method will require to reach it. Once we have the number of attempts, we use $TIME = 3 * NUM_ATTEMPS / PROC_CAP$ to find an estimate for how long that password would take to crack.

5 RESULTS

We found that we could easily crack passwords inside the password dictionary in just a few seconds. On the other hand, passwords that are not in the dictionary take a surprisingly long time to crack; this is due to the fact that our method attempts every possible alphanumeric permutation instead of just the common ones. The time to crack the password “1234”, and “ff” which are in the dictionary and dialed up, respectively, are shown in **Figures 3 & 4**. The password analysis feature is shown in **Figure 5**. The time to reach “g00dPwD” is so high partially due to the self-imposed process cap to ensure the Ubuntu virtual machine used for testing would not be accidentally fork-bombed due to its limited resources.

```
ILoveLinux~ time ./a.out passwords.txt
Password found. Aborting search

real    0m1.446s
user    0m0.338s
sys     0m0.214s
ILoveLinux~ cat su.txt
Password is 1234
The number of passwords tried are 7
ILoveLinux~
```

Figure 3: Output of cracking “1234”, which is in the dictionary passwords.txt

```
ILoveLinux~ time ./a.out passwords.txt
Password found. Aborting search

real    4m34.627s
user    0m43.360s
sys     0m43.025s
ILoveLinux~ cat su.txt
Password is ff
The number of passwords tried are 11324
```

Figure 4: Output of time it takes to crack password “ff”, which is not in the dictionary.

```
ILoveLinux~ time ./a.out passwords.txt apple banana g00dPwD
Analysis Mode:
It would take 2 seconds to crack the password: apple.
It would take 1 seconds to crack the password: banana.
It would take 5758718017 seconds to crack the password: g00dPwD.

real    0m0.014s
user    0m0.010s
sys     0m0.001s
ILoveLinux~ vim suprobe.sh
```

Figure 5: Output of password analysis for “apple”, “banana”, and “g00dPwD”.

6 CONCLUSIONS AND FUTURE WORK

In the future, a variety of things would be good extensions of the program. One such is the previously mentioned implementation of a processing cap. As it currently is, our program has no successful implementation that helps with the possibility of an accidental fork bomb. In the future, this could be addressed by doing some calculations based on the system to know how many possible processes can be run specific to the machine on which it is run. Similarly, it would preferably have a version of the program that would handle the spawned processes not being handled by root.

Another essential modification is to define the user we want to crack on the command line. As it currently stands, the user we are cracking is effectively a constant string in the program that requires re-compiling each time we want to change users.

7 REFERENCES

- [1] Michael Kerrisk. *The Linux Programming Interface*. <https://man7.org/linux/man-pages/man1/su.1.html>. (2021)
- [2] Kali Linux Tools. *sucrack*. <https://www.kali.org/tools/sucrack/>. Accessed: 2022-11-29
- [3] Carlos Polop, and Aron. “su-bruteforce.” Available: <https://github.com/carlospolop/su-bruteforce>. (2020)
- [4] Daniel Miessler. *Top 10000 Passwords*. GitHub. <https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt>. (2019)