|  |
| --- |
|  |
| Capstone Experience  IST 894 |
| Lab 2 – Password Auditing and Buffer Overflows |

Table of Contents

[1. General Context 3](#_Toc82902350)

[2. Technical Context 4](#_Toc82902351)

[3. Solution 6](#_Toc82902352)

[Task 1 6](#_Toc82902353)

[Task 2 8](#_Toc82902354)

[References 11](#_Toc82902355)

# General Context

In this lab we take a look at two very distinctly different security concerns. The first is password security and the importance of how it’s stored, and also how important setting and enforcing minimum password complexity requirements. Secondly, we take a look at buffer overflows, and how they can easily exploit vulnerable or flawed code to execute arbitrary code, gain access and escalate privilege, and even exfiltrate data.

Passwords are, at the same time, incredibly important and yet almost obsolete. When best practices are not enforced when passwords are created they can easily be cracked, even when they are stored as hashes and not in plain text. There are many open-source programs (11 Password Cracker Tools (Password Hacking Software 2021), n.d.) that are freely available, incredible easy to use, and quite effective when used properly. We are going to take a look at ‘John the Ripper’ which is one of the most popular open-source password cracking programs used, and show that it doesn’t matter if password are stored as MD5 or in SHA-512. If the passwords are commonly used words, we can crack a SHA-512 hash in 6 seconds.

A low-level language is a programming language that provides little or no abstraction of programming concepts and is very close to writing actual machine instructions (What Is a Low-Level Language?, n.d.). C is considered a middle-level language, but when it’s compiled it produces assembly code which interacts directly with the computer’s execution stack. There are a few well known vulnerable functions that are trivial to exploit with a buffer overflow. In a computer’s execution stack C can allocate a space in memory where the code can store data, and if the bounds aren’t checked you can input a larger value than is allocated and it will overwrite whatever is next in line in the stack. You can carefully place code at the end of that input and have the computer execute it instead of the original intended code. When this is exploited, the sky is the limit to how bad it can be.

# Technical Context

There are two main types of password attacks: online and offline (Miller, 2018). Online password attacks are much more limited and easily mitigated than offline attacks. Online attacks are attacks against running infrastructure like websites, email, SSH, and other services where someone logs in. Online attacks are limited by the speed of the network, the number of connections that a service allows, and they can be extremely noisy and very difficult to hide. These traditionally were often just brute force attempts, however in recent years a process called ‘credential stuffing’ has exploited end users tendencies to reuse password across multiple sites and services (Credential Stuffing Software Attack | OWASP Foundation, n.d.). Credential Stuffing takes advantage of username/password combinations that were exposed in previous data breaches, and there are trillions of combinations out in the wild at this point. Malicious attackers utilize botnets and attack multiple services simultaneously to help not overwhelm a single service and keep their actions hidden. They take these username/password combinations and just try to log in to various common websites and services to find these combinations that were reused.

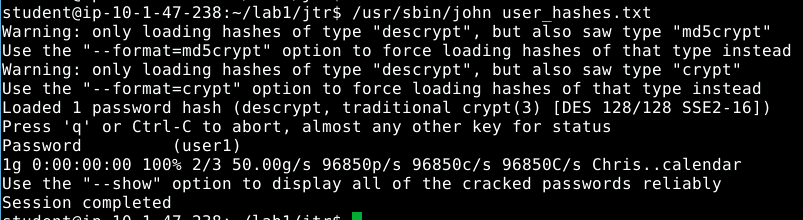
Offline attacks are potentially much more damaging, first in order for an attacker to have password hashes to try and crack they had to have compromised a server or service first, but once they have a list of hashes they can do whatever they want and there is no way know what they are doing or if they are successful. We use John the Ripper to crack four passwords that are stored in various hash types. If these passwords were of sufficient length and complex, meaning not using common words, at least two of them would have been incredibly difficult to crack. However, we identify that all four users use the password of ‘Password’ and we can crack a MD5 hash in .27 seconds, but we can crack a SHA-512 hash in just over 6 seconds. This lab shows that it’s equally as important to enforce password complexity as it is to store the passwords securely.

Smashing the Stack for Fun and Profit is a commonly referenced tutorial that goes over the basics of how to exploit buffer overflow attacks (Aleph One, n.d.). In this tutorial, they start off with an explanation of how buffer overflows work. They say “On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine” they note that when code does this it’s called smashing the stack, and can cause the computer to point and jump to a random or arbitrary address and if an attacker is able to place malicious shell code in that exact spot that the stack jumps to, anything is possible. You can gain unauthorized access, or retrieve data that you would otherwise be unauthorized to obtain. The primary purpose of this section of the lab is to show the importance of not using known flawed code, and also doing proper bounds checking so that you can’t write past the end of the buffer.

# Solution

## Task 1

We start this lab by opening a terminal window and changing our current working directory to ‘/home/student/lab1/jtr/’ and copying the file ‘user\_hashes.txt’ into it. In Figure 1 below, we do a basic password attack using John the Ripper (John) against the file with `/usr/sbin/john user\_hashes.txt` and we are able to crack user1’s password of ‘Password’ right off the bat. This is because John only loaded hashes of type ‘descrypt’, but you can see that it notes that it also saw type ‘md5crypt’ as well as ‘crypt’ and gives instructions on how to load those to be processed.



Figure

So our next step is to do what John prompted us to do; we run `/usr/sbin/john --format=md5crypt user\_hashes.txt` as shown in Figure 2, and then `/usr/sbin/john --format crypt user\_hashes.txt` as shown in Figure 3. So in a matter of 3 commands, we are able to crack all four passwords and find out that all four accounts used ‘Password’ as their password. So no matter how strong of a hash it’s stored in, it’s trivial to crack.

Text

Description automatically generated

Figure

Text

Description automatically generated

Figure

**Our next step is to create a table that documents the different user hashes and shows what type of hash each password is stored in, what command flags John needed to crack them, and how long it actually took to crack each. In order to time this we split each user:password pair into separate files and use the linux command ‘time’ to show how long a process takes to complete. Below in Figure 4, 5, 6 and 7 we show the timing that each hash requires to complete the cracking of it.**

Text

Description automatically generated

Figure

Text

Description automatically generated

Figure

Text

Description automatically generated

Figure

Text

Description automatically generated

Figure

**Now that we have all of the data that we need, we are able to fill out Table 1 which compares the 4 different hash types, which shows that even when stored in more secure hash types and salted it’s fairly quick to crack a hash when the underlying password is a common word like ‘Password’.**

|  |  |  |  |
| --- | --- | --- | --- |
| Username | Hash method | Command-line flags to John the Ripper to crack pw (if any) | Time to crack |
| User1 | DES |  | .027 seconds |
| User2 | MD5 Crypt | **--format=md5crypt** | .243 seconds |
| User3 | SHA-256 Crypt | **--format=crypt** | 7.207 seconds |
| User4 | SHA-512 Crypt | **--format=crypt** | 8.189 seconds |

Table 1

## Task 2

In our next task we will be looking at buffer overflows. We extract the buffer\_oflow.tar that is in the /home/student/ directory and then change our current directory to /home/student/lab1/buffer\_oflow/. In here we see two files, buffer\_oflow1 and buffer\_oflow2. We run buffer\_oflow1 and enter a guess for a password and as shown in Figure 8 are denied access.



Figure

We then run `objdump -d ./buffer\_oflow1` to disassemble the compiled program, and find the memory address where the ‘accessGranted()’ function is and we see in Figure 9 that it’s found at ‘080484AC’. We then take a look at the ‘main()’ function to see the order of how things are called. We see in Figure 10 that it uses ‘gets()’ to receive input from the end user, and then calls ‘check\_pw()’ to compare the password and see if it’s valid. Finally, Figure 11 shows the order in which ‘check\_pw()’ processes things, and we see that we need to manipulate the stack to execute ‘accessGranted()’ without knowing a valid password.



Figure

Text

Description automatically generated with low confidence

Figure

Text

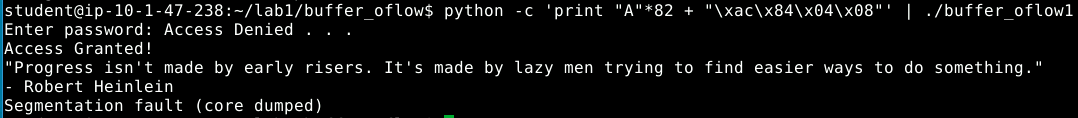
Description automatically generated

Figure

The next goal is to overflow the input buffer to the point where we can input a memory address and have the execution stack go to the address that we enter instead of the address that was originally in the stack. We use python to enter a variety of lengths of input until we get a ‘Segmentation fault’ which means that the stack tried to access the return pointer and we overwrote it with ‘A’s, the next goal is to reduce the number until we cat an ‘Illegal Instruction’ error. This is seen in Figure 12, and we find that after 82 characters is the exact spot where we want to add in the memory address of the ‘accessGranted()’ function, which we do in Figure 13 and achieve our goal. You’ll also notice that in Figure 13 we have to input the memory address, but since the processor is Intel based it’s little endian, which means the most significant value in the sequence is last and the least significant is store first. This means 080484ac needs to be input as ‘\xac\x84\x04\x08’ where the \x signifies that it’s hex encoded, but when input correctly, we achieve Access Granted!



Figure



Figure

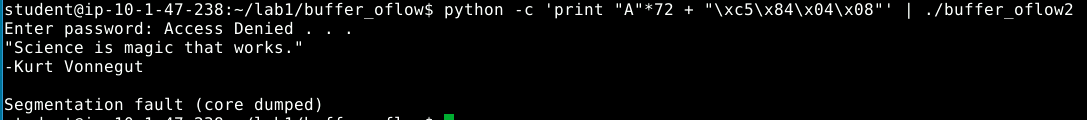
**Next, our goal is to perform these same steps on the buffer\_oflow2 file. So we once again use objdump to disassemble the compiled program. In Figure 14, we see that ‘accessGranted’ is at memory address 080484c5 in this program. So we then do our fuzzing to find the proper number of input characters, and Figure 15 shows that the correct number of characters to overflow buffer\_oflow2 is 72. We then take the memory address os 080484c5 and convert it to little endian hex and get ‘\xc5\x84\x04\x08’ and when that’s added to the 72 character input, we get a mixed result. First, we are told ‘Access Denied’, but then we see a quote from Kurt Vonnegut which is in the ‘accessGranted’ function, and that’s followed by a segmentation fault. This is because we are trying to execute code in the stack which is marked as non-executable. If when the program was compiled it included the flag ‘-z execstack’ it would essentially make the entire stack executable and allow this overflow to take full effect** (Protection against Buffer Overflows, n.d.).



Figure



Figure



Figure

# ****References****

*11 Password Cracker Tools (Password Hacking Software 2021)*. (n.d.). Retrieved September 18, 2021, from <https://www.softwaretestinghelp.com/password-cracker-tools/>

Aleph One. (n.d.). *Smashing the Stack for Fun and Profit by Aleph One*. Retrieved September 18, 2021, from <https://insecure.org/stf/smashstack.html>

*Credential stuffing Software Attack | OWASP Foundation*. (n.d.). Retrieved September 18, 2021, from <https://owasp.org/www-community/attacks/Credential_stuffing>

Miller, M. (2018, October 19). What’s the Difference Between Offline and Online Password Attacks? *Triaxiom Security*. <https://www.triaxiomsecurity.com/whats-the-difference-between-offline-and-online-password-attacks/>

*Protection against buffer overflows*. (n.d.). Retrieved September 18, 2021, from <https://www.win.tue.nl/~aeb/linux/hh/protection.html>

*Stack\_smashing.pdf*. (n.d.). Retrieved September 18, 2021, from <https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf>

*What is a Low-Level Language?* (n.d.). Retrieved September 18, 2021, from <https://www.computerhope.com/jargon/l/lowlangu.htm>