

Solve Nixu challenges

Felix Goding Jacob Almrot
Email: {felgo673, jacal517}@student.liu.se
 Supervisor: Andrei Gurtov, {andrei.gurtov@liu.se}
 Project Report for Information Security Course
 Linköpings universitetet, Sweden

Abstract – In this report we present some problems taken from the Nixu Challenges from 2020 and 2021. The main focuses of the report are reverse engineering and steganography as well as scripts. The report introduces each problem and discusses the interesting topics that the problems revolve around.

Reverse engineering is a possible attack for almost every program and works by analyzing a program to understand how it operates by for instance reading the code. Therefore, it is important to make it hard for the attackers to attempt such an attack. One way to accomplish this is to obfuscate the code to make it difficult to read.

Steganography is a difficult attack to prevent as it is hard to know when it is present. An example of a steganography attack is hiding malicious code in images where the image looks like an ordinary image. For that reason, it is important to not forget that these attacks are possible and be mindful of what to download from the Internet.

I. INTRODUCTION

Our project goal is to solve the Nixu challenges for year 2020 and 2021. The Nixu challenges are made by the company Nixu and the challenges revolve around cybersecurity and ranges from simple to very complex problems. The challenges are of a Capture the Flag (CTF) nature which means to solve the problem you need to find a flag (token) hidden in the problem files. The problems vary in different areas of cybersecurity, from cryptography to memory analysis.

In this report we will go through what challenges we solved and how we solved them. We aim to solve as many challenges as we can, but we prioritize problems of different nature. We believe this will give us a broader understanding of different concepts in cybersecurity and be more interesting to read about.

II. BACKGROUND

In this section we will go through some important terms to know about and the theoretical areas that the problems revolve around.

A. Terms

In this subsection we will describe terms necessary to understand the report without disrupting the flow of the text, such as SHA or port.

Port – Is an identifier on a computer network used when communicating with a specific process on the network [1].

Netcat – A program for reading and writing to network connections [2].

IDA Freeware – A program used for reverse engineering. It disassembles a binary into assembly and then shows relations between parts of the code. It also analyzes the programs memory for data like strings [3].

Base64 – An encoding used to represent binary data to text [4].

SHA – A cryptographic hash algorithm [5].

Salt – Random data added to the information that will be hashed to make the result more secure [6].

Stack-smashing attack – When an attacker overwrites stored data on the stack [7].

B. Theory

Here we will explain the theory behind the problem types, such as what steganography is or how reverse engineering typically works.

Reverse engineering – Analyze a program to understand how it works. For example, turn a binary file into a series of assembly instructions. This makes it possible to see the

execution of the program during runtime. It is also possible to view strings written in the program and analyze the flow of the program [8].

Steganography – This is a technique where you hide some confidential information in message in a way that only the receiver of the message can retrieve this information. The most common ways this information is hidden are in an image, a file, or a video. The advantage of this is that you can exchange messages that will seem normal when they in fact hide some confidential information [9].

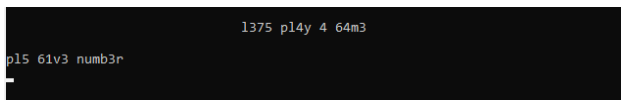
Cryptographic hash functions – These functions are really useful when you want to transform data of an arbitrary size into data of a fixed length. These functions has multiple different properties, but the main ones are that the same message will always map to the same hash and it’s almost impossible to find two different messages that has equal hash value. It is also very difficult to find the original message if the hash value is known, which makes cryptographic hash functions a great tool for storing passwords where if the hashed passwords are leaked it would take a long time to get the original password [5].

III. RESULTS

Here we will present the problems and then go through how we solved them, but without any reflections. We will aim to present each problem in a way so that the reader can follow along each step of the solution.

A. Numb3rs

The exercise gave the instruction to connect to the server address “numb3rs.thenixuchallenge.com” on port 1337. Fig. 1 show the initial response from the server if we connect using the program netcat. We can interact with the program running on the server using the terminal window. The exercise instructions told us that to get the flag we needed to enter one specific combination of numbers to get the flag.



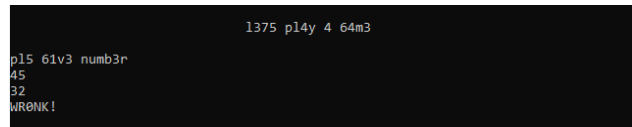
```

1375 p14y 4 64m3
p15 61v3 numb3r

```

Fig. 1: The initial server response.

We tried to enter a random number to see how the program would respond. The response can be seen in Fig. 2 After entering the wrong number, the connection would be terminated. We tried to establish a new connection entering the right number given to us by the server. This prompted a new question for a number which gave us also the correct number if we entered the wrong one.



```

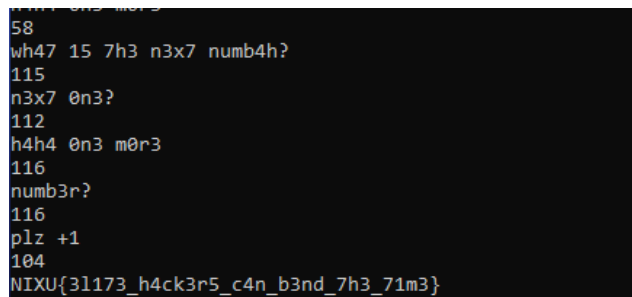
1375 p14y 4 64m3
p15 61v3 numb3r
45
32
WR0NK!

```

Fig. 2: The server response after sending a number.

It would be very inefficient for us to manually connect to the server to write numbers to find out the correct sequence. Instead, we wrote a script that would handle this for us. We tried different approaches for how the script would communicate with the server. Our first idea was to use sockets in python to communicate with the server. Neither of us had experience using sockets and we had problems with sockets blocking and not sending data correctly.

When we got stuck with sockets, we tried to start a child process using python that the program could communicate with. This worked better for us but eventually we got a problem using this method and we then realized that we could instead create a very simple script to achieve our goal. We piped the input into the command “nc numb3rs.thenixuchallenge.com 1337” and then piped the output into a file. Then we wrote a script that used to output file to increase the input sequence and then rerun the process until we find the whole sequence. One problem that we found was that the input could not be too fast or too slow otherwise the server would not give us the next number. We used sleeps in our script to circumvent the problem and after letting the script run for a while, we got the output shown in Fig. 3.



```

58
wh47 15 7h3 n3x7 numb4h?
115
n3x7 0n3?
112
h4h4 0n3 m0r3
116
numb3r?
116
plz +1
104
NIXU{31173_h4ck3r5_c4n_b3nd_7h3_71m3}

```

Fig. 3: The server response with the flag

B. PEazy

The exercise gave us a windows binary and the challenge are to reverse engineer this binary. We also got a .pdb file that contains debugging information about the program. After executing the program, we got the following prompt which can be seen in Fig. 4.

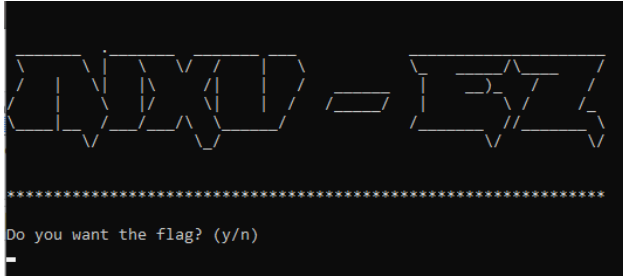


Fig. 4: The prompt when running the PEazy exe.

We tried different inputs that gave us different responses, but never got any new information that could lead to the flag. To disassemble the binary, we needed an application that could interact with the program and let us look at the instructions and stop the program during execution. We tried a few different programs that did not work or gave us too much information that made it difficult to analyze. During a presentation of our progress the examiner gave us some recommendations for tools we could use. The program that was most suitable for us was IDA Freeware. In Fig. 5 you can see the resulting disassembly of the program using IDA.

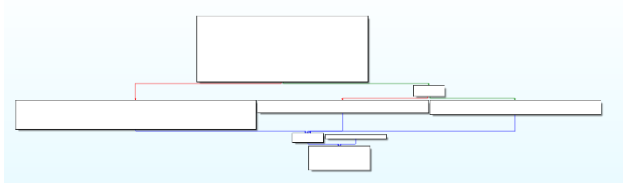


Fig. 5: View of the disassembled program in IDA Freeware.

Fig. 5 shows blocks of assembly code of different execution paths in the program. IDA made it possible to step through the execution showing us each instruction the program runs and how different inputs executes different parts of the program. When we got a good understanding of the flow of the program, we started to analyze the code and different variables in the program. We found a lot of different interesting parts that could help us find the flag, for instance a variable containing a base64 string. Decoding the string gave us a text about how CTF works, but unfortunately nothing about the flag.

Looking at the different blocks of code we found a block with a function called “nothing_to_see_here”, see Fig. 6. The program had no execution path resulting in that this function was called. To make the program execute this part of the code we needed to influence the execution of the program. We checked the code to see if there are any conditional jumps that we could influence by changing flags during execution, but we only found absolute jumps.

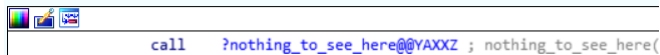


Fig. 6: A code block representing a function in the program.

A way for us to change the execution path of the program is to change the return pointer on the stack. When the program calls a function, it pushes a pointer to the stack indicating where the program should return to after executing the function it called. We can modify this value and make the program execute whatever part of the code we want. When we changed the execution to call the function called “nothing_to_see_here” the program printed the ascii values of the flag to the console. All that was left was to decode the ascii characters to text which resulted in the flag.

C. Stegobööö

In this exercise we were given a .jpg file which was a picture of some hamburgers. The picture can be seen in Fig 7. At first glance the picture looked like an ordinary picture, but the exercise gave us the hint that the problem revolved around steganography. Therefore, we believed there was hidden information encrypted in the file.



Fig 7: The picture of hamburgers we were given.

The first step we took was to open the .jpg file in a text editor to investigate if there was some information appended at the end of the file. Our suspicion was confirmed as there were some characters that looked like Base64 encoding at the end. The characters can be seen in Fig. 8.

```
DEBYWVzLTI1Ni1jYmMgdy8gc2FsdCBhbmQgUEFTU1dPUkQsIHNoYTI1NiBkaWdlc3QK
```

Fig. 8: The characters at the end of the .jpg file.

The result of the decoding can be seen in Fig. 9.

```
DEBbaes-256-cbc w/ salt and PASSWORD, sha256 digest
```

Fig. 9: The Base64 decoded characters.

With this information it was clear how the file was encrypted i.e., with AES-256-CBC with a salted password and then the salted password was hashed with SHA-256. After this we began the search for the password to begin the steganography extraction process. We could not find any hints of the password in the .jpg file, but with some further inspection of the image itself we saw some

suspicious looking characters at the top left corner of the image. After studying the characters, we saw that they were Braille characters. These can be seen in Fig. 10.



Fig. 10: The Braille characters in the image.

After translating these using the Braille alphabet, we got the following text:

BURGERSAREGOODWITHBRAILLE

This seemed to us likely to be the password used to encrypt the file, but with that said we were never able to decrypt the file. We tried several of the known steganography tools available such as Steghide [10], OpenPuff [11] and OpenSSL [12] but with no success.

After some further investigation for tools, we used Binwalk [13] which is a tool to search binary files for information. This gave us another hint on how the image was encrypted:

OpenSSL encryption, salted, salt:0x609AFF3F2C08EB8D

This confirmed the use of OpenSSL to encrypt the file. With this new information we tried decrypting using OpenSSL yet again, but we could still not decrypt the file.

IV. DISCUSSION

In this section we will discuss each problem and the interesting things behind each of them. We will describe what area of cybersecurity it belongs to and how the problem relates to real world problems such as well-known attacks or how this area is necessary for the security.

A. Numb3rs

We used python to write a script to automate a repetitive task and, in the script, we also needed to manipulate time. When communicating with the server we needed to adjust the communication speed to get the next number. If we sent data too fast or too slow it would send a message warning us about that.

When creating the script, we tried different solutions that gave us different problems. In the end we went with the simplest. Instead of writing code that handled socket communication we used the fact that we can write and read data from programs using the terminal. We can send input data for a program and then read the response. We used netcat to communicate with the server and gave it input to use when communicating with the server. The response was stored in a file and then the script could use the

response to add new data to the input it sent. Then repeat the process until we found the flag.

B. Reverse Engineering

Most programs are vulnerable to reverse engineering. This is because most programming languages use a lot of metadata that greatly helps programmers to debug an application. This could also be used by an attacker to understand how the program works. According to OWASP a program is vulnerable to reverse engineering if any of the following applies:

- Clearly understand the contents of a binary's string table.
- Accurately perform cross-functional analysis.
- Derive a reasonably accurate recreation of the source code from the binary. Although most apps are susceptible to reverse engineering, it is important to examine the potential business impact of reverse engineering when considering whether or not to mitigate this risk. See the examples below for a small sampling of what can be done with reverse engineering on its own [14].

a) Prevent reverse engineering

To prevent reverse engineering, you can use a code obfuscation tool. This makes the program difficult for decompilers to interpret and makes it hard for an attacker to understand the code [14].

b) Possible attacks

One possible attack is that an attacker could analyze the *strings* written in the code. These strings could be credentials for connecting to the backend or find out other information about how the program works.

An attacker could use a decompiler to modify parts of the program to run malicious code designed by the attacker.

According to OWASP the technical and business impacts of reverse engineering are to:

- Reveal information about back end servers;
- Reveal cryptographic constants and ciphers;
- Steal intellectual property;
- Perform attacks against back end systems; or
- Gain intelligence needed to perform subsequent code modification.
- Intellectual Property theft;
- Reputational Damage;
- Identity Theft; or
- Compromise of Backend Systems [14].

c) PEazy

The binary given in the exercise did not contain any code obfuscation which made it possible for us to easily analyze the code i.e., we could clearly see different labels for variables. If you are not used to read assembly it's hard because it's very different from programming languages used today. IDA has a feature that tries to reconstruct the C-code that could be compiled to the given assembly. This helps attackers that are more accustomed to reading and writing in high level languages.

In this exercise we could also examine all content of the binary's string table. In this case we did not find anything specific to solve the challenge, but it helped us understand the program. IDA gave us an overview of the program and the execution path, see Fig. 5. This view helped us identify a function that was never executed, which with the help of a stack-smashing attack enables us to execute the function that was never called. We achieved this by changing the return pointer from a function call. This in turn made the program execute code from the pointer we added to the stack.

There are some different ways to defend a program from a stack-smashing attack. You could use tools when compiling like IBM's ProPolice [15] or Stackguard [16]. Another option is LibSafe [17] which is a dynamic runtime solution. A none-compiler-based solution is placing a *canary* on the stack preventing the attacker from modifying the stack without the program knowing it. The canary is a random value that the program can verify and when an attacker overwrites the stack this value will also get overwritten. Because the value is random the attack does not know what value of the canary is and cannot add in the value. When the program reads from the stack and does not find the canary it knows the stack has been altered [7].

C. Stegoböögö

Stegoböögö was as previously stated a problem related to steganography. Even though we did not manage to solve the problem we learned some things that we can take with us. The most important thing is probably how we perceive data that is sent over the Internet, mainly images. For instance, if a website allows you to upload images it could be exploited with steganography by hiding malicious code inside that will be run by the website's server if they have a poorly designed image loader.

Steganography attacks are according to McAfee "...easy to implement and hard to detect" [18]. For example, on the outside two images can look the same but one of them contains malicious code which will not be noticed since the image is not scanned for tampering. McAfee continues with explaining what organizations should do to prevent

steganography attacks. For instance, they recommend that you have a clear path for the user to download your applications instead of them downloading your application from a third-party site where the content can contain steganographic code.

V. CONCLUSIONS

The challenges we have attempted have given us a lot of new information and insight to possible cyberattacks and how to prevent them.

It is impossible to make a program completely protected against reverse engineering, but you can make it hard to attempt it. The impacts of reverse engineering could be big and that is why it is important to implement protection accordingly. From PEazy we learned that you can never trust the stack and that a user can circumvent if statements and even function calls. This can have a catastrophic impact on the program depending on its functionality. It could help the attacker to get valuable information about a server or even talk to the server using high level credentials. That is why it is important to secure your application accordingly to the potential impact an attacker can gain from your binary.

Steganography attacks are difficult to notice, but they are easy to implement. A common way for a steganography attack to happen is to include malicious code in images since antimalware software seldom scan for threats in images. Therefore, attackers can utilize a bug in the image loader to make it execute the code hidden in the image. This has broadened our perspective on how we perceive data on the Internet and to take caution by which data we download to our computers.

From the Numb3rs task we learned that sometimes the easy solution is the best. Use programs and solutions other have already made. You do not need to rewrite everything yourself. We also learned how powerful it can be to combine the terminal (bash) with python. Automation is an important role when looking for security risks because you cannot do everything manually. Learning the skill to write your own small scripts to help you out when you find no other option is a valuable skill. That is something we take with us that learning to write small scripts and making different programs interact with each other to get the result is important for security analysis.

REFERENCES

- [1] Cloudflare, "What is a port?," <https://www.cloudflare.com/learning/network-layer/what-is-a-computer-port/>.
- [2] Sourceforge, "Netcat 1.10," <https://nc110.sourceforge.io/>.
- [3] Hex-Rays, "IDA Free," <https://www.hex-rays.com/ida-free/>.
- [4] Mozilla, "Base64," <https://developer.mozilla.org/en-US/docs/Glossary/Base64>, 2021.
- [5] Brilliant.org, "Secure Hash Algorithms," <https://brilliant.org/wiki/secure-hashing-algorithms/>.
- [6] Defuse Security, "Salted Password Hashing - Doing it Right," <https://crackstation.net/hashing-security.htm#salt>.
- [7] O'REILLY, "Prevent Stack-Smashing Attacks," <https://www.oreilly.com/library/view/network-security-hacks/0596006438/ch01s13.html>.
- [8] T. Ciproso and M. Stamp, Software Reverse Engineering, Heidelberg: Springer, 2010.
- [9] M. Semilof and C. Clark, "steganography," <https://searchsecurity.techtarget.com/definition/steganography>, 2018.
- [10] S. Hetzl, "Steghide," <http://steghide.sourceforge.net/>.
- [11] Embedded SW, "OpenPuff - Yet not another steganography SW," https://embeddedsw.net/OpenPuff_Steganography_Home.html.
- [12] OpenSSL Software Foundation, "Welcome to OpenSSL!," <https://www.openssl.org/>.
- [13] refirmlabs, "Binwalk," <https://github.com/ReFirmLabs/binwalk>.
- [14] OWASP, "M9: Reverse Engineering," <https://i.redd.it/yaixm7mbj0x61.jpg>.
- [15] E. Hiroaki and Y. Kunikazu, "ProPolice: Protecting from stack-smashing attack," <https://dominoweb.draco.res.ibm.com/eea6d413c79a5ccc85256b89002dde70.html>, 2002.
- [16] H. Sidhpurwala, "Security Technologies: Stack Smashing Protection," <https://access.redhat.com/blogs/766093/posts/3548631>, 2018.
- [17] SecurityFocus, "Libsafe Multi-threaded Process Race Condition Security Bypass Weakness," <https://www.securityfocus.com/bid/13190/info>, 2009.
- [18] McAfee, "Protecting Against," <https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-quarterly-threats-jun-2017-2.pdf>, 2017.