Nixu Challenges

Adrian Byström adrby928
Martin Ryba marry564

Supervisor: Andrei Gurtov, andrei.gurtov@liu.se
Project report for Information Security course
Linköpings universitet, Sweden
May 24, 2021

Abstract

In the rapidly changing world of information security the need to hone ones skill is always needed. To do this one can partake in Capture-the-flag challenges. These challenges range in topic and are either hosted by a third party or directly by companies. This report is a write-up of a collection of challenges from the company Nixu and the lessons drawn from trying to solve these.

1 Introduction

The world of information security is constantly changing. In the beginning, it was simple worms, made for the amusement of the author and the admiration of said authors peers. To now where information security is becoming more and more important as more of society is taking place online, and more and more devices are being connected to the Internet. However, the need to show one's prowess within the field is still prevalent to this day. This is done through different challenges posted online, called Capture-the-flag challenges (often shorted to CTF challenges). These challenges range wildly in the scope and difficulty and tackle subjects from Cross-site-scripting and SQL injections to reverse engineering of executables and image steganography. The challenges are often posted by professionals within the industry. Not only are these challenges coupled with bragging rights by, it also hones the challenger's skills within the field of Information security, having to act as an "bad actor" or "black hat" instead of being on the side that protects from attacks. Using this type of challenges to learn is a form of gamification and is a known pedagogical tool for teaching.[1]

There are currently many websites hosting CTFs across all the sub-fields of security. One of such websites is Nixu challenges, with a broad array of challenges ranging from reversing to web security. Each year, a new set of tasks is published for the participants to solve. The *creator* of these challenges is the company Nixu[1], a security firm started in Finland but with offices around Europe, with an office in Linköping as well.

The aim of this project is to solve as many of these challenges as possible and explain the solutions in this report. To narrow the scope of the project six (6) challenges were chosen (three (3) from 2021, and three (3) from 2021). These challenges are, in order of difficulty: numb3rs, magic, PEasy, Stegböögö, parasite, and PEybass.

These challenges could then be grouped into different subjects of Information Security. The "numb3rs" challenge is a challenge focused on scripting and sending packets to a server, whilst "magic", "PEasy", and "PEybass" are focused on reverse engineering. "Stegoböögö" is an image steganography challenge and "parasite" is an analysis of network data. Not a lot of information is given about these challenges, often not more than a paragraph of flavor-text describing the task in quite a vague sense, and some sort of zip-archive or a hostname and port to a system. It is up to the solver to figure it how to solve the challenge at hand.

2 Challenges

The following sections will describe each challenge as well as the proposed solutions to them with requisite analysis done for said challenge.

[1]https://www.nixu.com/

2.1 Numb3rs

The "numb3rs" challenge deals with scripting a TCP-connection to the server

```
numb3rs.thenixuchallenge.com
```

at port 1337. The challenge text reads as follows *Hello, friend. You don't know me, but I know you. I want to play a game. There is only one combination of numbers that gives you the answer. Know what to do. You better hurry up. Make your choice..* When connected to the server using for example netcat an ascii-image of a clown character and the text

```
    1375 pl4y 4 64m3

pl5 61v3 numb3r
```

is printed, the text being "Lets play a game" and "pls give number" in leet-speak. Leet-speak (sometimes stylised as 1337-speak or 31337-speak) is a style of writing that emerged in the early days of the internet, where letters where changed to numbers that on a computer-screen looked similar to each other. It was a soft way of obfuscating text and passing through automatic text analysers as well as keeping people not part of said this subculture out. [2] The attacker then has to provide the correct number to proceed. Is the wrong input (wrong number or wrong input entirely) the connection closes, but not before printing the correct number to the screen.

2.1.1 Analysis and solution

The challenge does not require that in-depth analysis to solve. Given that the "correct" input is provided when "failing" the solution is quite straightforward.

By creating a python script that connects to the challenge-server and records every correct number when failing an array of the correct input is gathered. The script will then exit when the flag is returned. Since the flag is formatted as NIXU... it is quite simple to catch. The only issue that occurred when solving this challenge was that the server did not accept solutions being sent too quickly or slowly after each other, so a sleep of 0.5 (point 5) seconds needed to be added to the script. This added a lot to the execution time of the script since there were 80 numbers in the array at the end.

This solution to this challenge was definitely not the most efficient and technologically *clean* solution. The python script written for this challenge was quite unstable and needed to be monitored so that it could be restarted if the connection ended unexpectedly. However, given the fact that the other challenges required more time and effort to solve, the need to harden the script fell behind the need to solve the other challenges.

2.2 Stegoböögö

"Stegoböögö" is a challenge pertaining to image steganography. Steganography (derived from the Greek word *steganos*, which translates to "reticent" or "covered"[3]) is the art of concealing information within other seeming innocuous information. In terms of computer security, steganography is often used to hide messages or files within other files, such as images. The art of hiding messages is not new being used for thousands of years. The fifth (5) century tyrant Histaiacus is said to have hidden a message by tattooing said message on the scalp of a slave and sending the message by having the slaves hair grow out and sending said slave to the intended recipient, who read the message by shaving the head of the slave. In regards to this challenge there are many different steganographic techniques used to hide information within specific images. The different techniques take advantages of different aspects of digital images. Data can be hidden within the bits pertaining to the color values of each pixel (the most notable technique being least-significant-bit steganography) or by using the transform functions found in image processing techniques, hiding it the coefficients of the frequency components when processing the image using, for example, a discrete Fourier transform.[4]

The challenge provides a JPEG image of hamburgers and the challenge text:

```
Just a delicious böögö.
```

The attacker has to figure out how the data has been steganographically stored in the image and retrieve the key from there.

2.2.1 Analysis and solution

Given that the only thing provided by the challenge is the text and image mentioned above the initial problem to solve is to figure out where in the image to look for the hidden data. By opening the image in an xxd (a hexdump program for GNU/Linux) a string of base64-encoded characters could be read. Decoding the characters to ASCII results in the string *aes-256-cbc w/ salt and PASSWORD, sha256 digest.* Using the tool binwalk[2] on the image reveals that a trailing file is hidden in the image that is encrypted using Openssl. To verify that After this, the tool stegovertias (an image steganography analysis tool) extracts this file. The resulting analysis from this tool also indicates that no data has been hidden in the color values of pixels or in frequency components, since when trying to extract that data only results in *junk*. The technique used for hiding the file was simply to append the file to the image, after the JPEG End-Of-Image segment. Returning to the image, in the upper left-side corner one can make out some faint square shapes. By playing with the brightness when viewing the image, a *string* of braille characters can be made out. This string is read as "BURGERSAREGOODWITHBRAILLE". By using this as a passphrase when decoding the trailing file using OpenSSL returns a password-protected 7z-archive containing a wav-file. Trying to unlock the archive using the same passphrase used for the OpenSSL file yields no result. Given that no other hint for a potential password is found the archive needs to be brute-forced.

No further solutions were found for this challenge, and it was therefore not solved.

2.3 Magic

The challenge starts with a website that generates pseudo-qr codes based on the input. However, upon closer inspection, it is revealed that the underlying javascript takes the first 80 bits of input and uses them in 1681 logical formulae that result in 41x41 binary image.

The goal seems to be solved where the logical formulae give 1 in all output bits or 1s and 0s in such a manner to produce a QR code. First, we tested inputting all 1s or all 0s to the formula to see the behavior. Next, we inspected the logical formulae to see if we can figure out some kind of simplification. We split up the formulae for individual bits into smaller parts that had to be true or false based on the desired result for the entire bit. The idea was to calculate the truth tables for these smaller parts (with far fewer variables) and see if we can find some input bits with unchangeable value. After crashing few online solvers (even with the smallest parts of the formulae), we used programs running locally. After calculating several of these smaller parts, we did not find any satisfying results that would help us with the rest of the problem.

As the testing of logical parts was taking a significant portion of time, it was decided to try and see if there is some easier way to see the connection between input and output bits. We created several output/input samples and compared various encodings (including image encodings) and unfortunately had to conclude there is no encoding that would explain the data transformations observed that we know of.

After trying these options this challenge needed to be dropped priorities and focus changed to solving other challenges due to time constraints.

2.4 Parasite

The parasite challenge gives,*in regular Nixu fashion*, little to no guidance when from its challenge text. It (the challenge) is tagged as a pcap challenge, and only provides one small paragraph of text, a picture of a strange device the authors of the challenge found in their office2.4, and a zip-file containing the data the authors captured from the device. In the picture, the device is connected through a VGA-splitter to a computer.

The paragraph of text reads as:

```
We found a mysterious device attached
to a computer and it seems to be
transmitting some data. Can you
make any sense from this?
```

[2]https://github.com/ReFirmLabs/binwalk

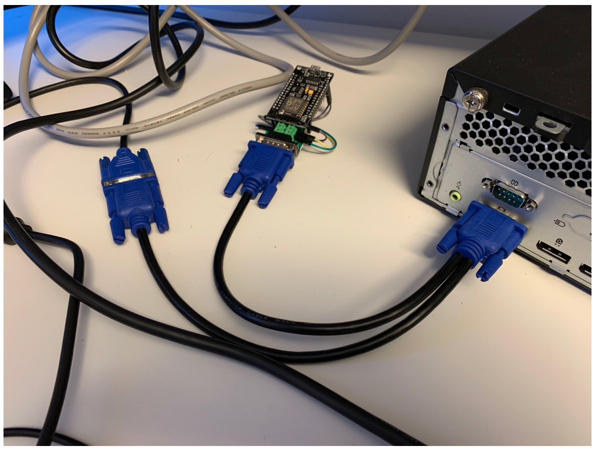


Figure 1: Strange device attached to computer using VGA cable.

The provided captured data given in the zip-file contains a large text-file. This text-file is presented as the network data sent from the device through the authors office network. That data is formatted line-by-line with entries. These entries are in turn comma-separated, with four different datapoints. Here follows an excerpt of said data.

```
1580765524.480193840,,,-0.061
1580765524.480193920,,,-0.061
1580765524.480194000,,,-0.061
1580765524.480194080,,,-0.056
1580765524.480194160,,,-0.061
1580765524.480194240,,,-0.056
1580765524.480194320,,,-0.061
1580765524.480194400,,,-0.061
1580765524.480194480,,,-0.061
1580765524.480194484,0,1,
1580765524.480194492,1,1,
1580765524.480194524,0,1,
1580765524.480194536,1,1,
1580765524.480194544,0,1,
1580765524.480194560,,,-0.061
```

2.4.1 Analysis and solution

Initial analysis of the challenge text and the provided material (picture2.4 and captured data) hints towards the file being the PCAP-file[3] for the data captured from the *mysterious device*. The PCAP file format is a format used when analysing and capturing network traffic and given that the challenge was tagged with pcap the logical next step is to analyse the data using a program designed specifically for that purpose. However, when opening said file in wireshark, a network protocol analyser, no discerning information is found. Wireshark claims the file is in the wrong format, and only garbage data is displayed. One can see that as well on closer inspection of the file, as it does not follow the standard of a PCAP-file.

[3]https://www.tcpdump.org/

Going back to the challenge text, data, and picture gives some more hints on how to actually go forward with the challenge. Firstly the challenge is not only tagged as a pcap challenge but is also tagged with the text *or is it?*. This indicates that what has been captured is not actually the network data from the device, but something else.

Looking closer at the data, the first data-point is of a different format then the rest. Running one of those data points through a translator for UNIX-timestamps produces a timestamp down to the nanosecond on the third of February 2020 (2020/02/22). From this one can deduce that the device gathers on each of these timestamps and sends them through the network. A further deduction of the data ends in the following

- First field seems to be a timestamp
- For lines with second and third field empty (type A), the timestamp seems to increment by 80
- Type B lines (with fourth field empty) seem to correspond to some event and do not affect timing of type A

To further deduce what data is being captured one needs to examine the picture2.4. Looking closer at the picture one can see 3 cables going into the device. One blue, one cyan, and one yellow. Given that the device is connected using VGA and consulting a VGA pinout schematic the cables are the blue channel, horizontal sync, and vertical sync. By looking at the data the three last data points can be deduced as such. Given that the last data point is filed with a seemingly analog value that corresponds to the value of the blue channel in a VGA-cable when active. The other values would consequently be the sync signals, given that they are a simple binary on or off. From this the nature of the *mysterious* device is deduced. The device is a homemade frame-grabber, a device used to covertly grab what is being displayed on the screen and either saving it or, in this case, send it over the network to the attacker. The solution to the challenge becomes quite obvious after this. To get the flag for this challenge, one only needs to replay the captured data back into frames.

2.5 PEasy

"PEasy" is a reverse engineering challenge. The challenge provides a 7zipped archive containing a Windows executable and a .pdb file. Using these the attacker needs to disassemble the executable to obtain the flag. When running the executable the attacker is asked whether or not he wants the flag. Regardless of the answer the executable deems it not to be enough and exits the program. The challenge also provides this text:

```
We love Windows binaries, do you?
```

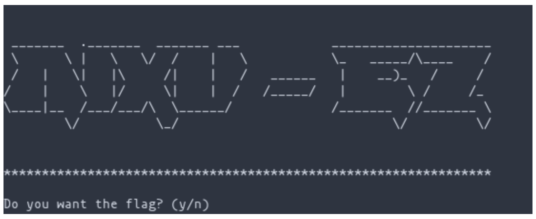
In classical programming, source code is converted to machine instructions through a process known as compiling. Compiling consists of several steps, the last of which is assembling machine instruction from assembly code.

Without going into too much detail, many of the advanced (originally) diagnostic tools can be used by nefarious actors to partially reverse this process by recreating the assembly code. This often provides the attacker with sufficient information about the structure of a given binary to be able to change its behavior (such as to remove digital watermark or bypass password checks) [5].

As can be expected, much effort is spent trying to hamper reverse engineering by other parties through various obfuscation techniques [6].

To complete this challenge, we used a disassembly tool called Cutter[4]. Cutter allows the user to view the processor instructions of the given binary file, view them in graph view (to better visualize the jump instructions), find used strings and edit the used instructions.

[4]https://github.com/rizinorg/cutter



```
 ______  ._______  _______ ___          __________________
 \      \ |   \   \/  /    |   \        \_   _____/\_____   /
 /   |   \|   |\     /|    |   /  ______ |    __)_  /      /
/    |    \   | /     \|    |  /  /_____/ |        \/      /_
\____|__  /___/___/\  \______/           /_______  //_______ \
        \/          \_/                          \/         \/


****************************************************************

Do you want the flag? (y/n)
```

Figure 2: PEasy prompt

After running the given binary, the user is prompted by:

```
Do you want the flag? (y/n)
```

Answering y or n results in a closing message and the program exits. Answering other characters results in a reminder to answer either y or n. Therefore, the initial idea was to inspect the strings contained in the binary and see if any of them is the 'password'. After inspecting the string sections (can be done by either the cutter or GNU strings utility), what was found was one long string that after decoding from base64 and deciphering through Caesar's cipher (n=13), yielded extraction of Wikipedia article about Capture the Flag (CTF)[5]. Various other strings were also tested as the passphrase with no success.

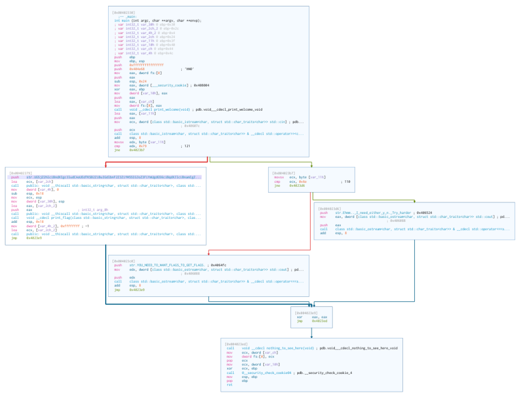


Figure 3: Graph view of the disassembled code

The next step was to inspect the disassembled code - the aforementioned graph view was especially useful in visualising the logical flow of the binary. Simply following the conditions, one can see

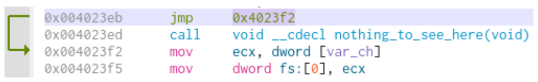
[5]https://en.wikipedia.org/wiki/Capture_the_flag

that the program first displays the prompt and then compares the input with other characters. Written in pseudocode, the program flow was similar to:

```
if input == 'y'
    print "try harder"
else
    if input == 'n'
        print "you must want the flag"
    else
        print "you need y/n"
```

Sch code doesn't check against anything other than y or n so the previous attempts at guessing some secret input were useless.

As this was the extent of logic branches of the binary, the next step was to inspect the disassembled code line-by-line for hidden data or instructions that did not show on the graph view (due to being skipped by jump statements). This probing has been more successful than previous attempts as a skipped function call was quickly found:



```
0x004023eb    jmp     0x4023f2
0x004023ed    call    void __cdecl nothing_to_see_here(void)
0x004023f2    mov     ecx, dword [var_ch]
0x004023f5    mov     dword fs:[0], ecx
```

Figure 4: Function call that is skipped over

Simply changing the jump instruction to point to the omitted instruction caused the program to display additional output before exiting:

```
78 73 88 85 123 104 97 104 97 95 112
108 122 95 116 104 105 115 95 119 97
115 95 115 111 95 101 122 95 120 68 125
```

After translating through ASCII, we get the valid flag for this challenge:

```
NIXU{haha_plz_this_was_so_ez_xD}
```

2.6 PEybass

"PEybass" is, like the challenge mentioned above, a reverse engineering challenge. As with "PEasy" it provides a 7zip archive, this time only with the Windows executable. The attacker then needs to disassemble the executable to obtain the flag. Instead of asking whether or not the user wants the flag, the program asks for the flag directly, to verify that it is correct. The challenge text is to no help in this:

```
We love debugging. <3
```

2.6.1 Analysis and conclusion

This challenge was similar to the PEazy challenge. However, no .pdb file was given. So to solve this challenge one needed to look at the code more directly. Opening the code in a disassembler yielded a flow graph that was quite convoluted. Modules are looping around each other and a lot of arbitrary jumps are done between said modules. Looking closer on the instructions one can see that the code has been obfuscated. Arbitrary comparisons of values, instructions placed in precarious places that have no overall impact of the functionality of the program. To solve this challenge, one therefore need to obfuscate the code and *flatten* so that the logical flow of the program is found. However, due to time constraints, this challenge also had to be dropped.

3 Conclusions

As mentioned in the introduction to this paper, solving challenges are a good way to hone one's skills with information security. Not only are existing skills improved, but knowledge of new techniques are often the result of this. In this conclusion, those lessons will be put forth.

3.1 Scripting

The numb3rs problem was a good introduction to the general concept of these challenges. It was a good lesson about smaller CLI tools such as curl and how to combine these smaller programs together in a bash script in order to create one functional unit tailor-made for the problem at hand. As bash scripts are often an invaluable tool for programmers and power-users alike, having a relatively simple yet non-trivial problem is crucial for developing their capabilities. Solving this challenge also honed one's skill in socket-programming and the TCP protocol using Python. Techniques for sending TCP packets and keeping a socket connection opened over a (relatively) long time, and catching the errors that can occur when working with sockets. However, as this challenge was not the most intricate the knowledge gained from solving this was minuscule.

3.2 Stegnography

Steganography is a field that is more akin to actual espionage more than the other CTF challenges found on the internet. Given this, to solve this challenge much research had to be done into different techniques for steganography. Steganography being an old technique for hiding information has a rich history starting in ancient Greece with the scalp being tattooed with the message, their hair regrowing over said message, and said hair being shaved by the receiving party for the message to be read. In the current world, steganography is more sophisticated and often uses quirks in media encoding protocols to hide information without altering the transport media. Media files such as audio or images are preferable for their (relative) large size. For images, hiding information in the bits corresponding to each pixel is a technique often used. The most notable one is the least-significant-bit steganography, hiding information in the least significant bit of the color value of each pixel. Since that bit does not have that large of an impact on the overall image, no noticeable changes are seen when looking at the said image. For the Stegoböögö challenge, another steganographic technique was used, called trailing steganography. In this instance, instead of hiding information in the image data information was hidden after a JPEG trailer. Usage of this quirk of the JPEG file format is the hallmark of modern steganography, but if one is not well versed in the techniques it is still something that can slip under one's nose. In trying to solve this challenge, two specific tools were used to analyse and extract the information. Using these tools are on their own a lesson learned in steganography, giving one knowledge in where to look when analysing potential stenographically hidden messages. The tool binwalk, which is used for analysing binary images for hidden information and executable code. This tool proved especially useful in the beginning, for when finding where the data was hidden. Binwalk found the hidden file trailing behind the original JPEG, and provided more information of the file signature for said file. Stegoveritas is a tool used, much like binwalk, for the analysis of binary image files. It also provides extraction of possible hidden information within an image and using this tool provided the actual hidden file. Stegoveritas is also helpful with more in-depth steganographic analysis and will try to extract information hidden using the aforementioned LSB steganography.

3.3 Dissassembly

The disassembly problems, PEasy, PEybass, were a great tool in helping the team understand the theory behind disassemblers together with basic x86 instructions and inner workings of a processor. A variety of analysis tools were used as part of the cutter reverse engineering platform, such as string extraction, flow diagram, classical disassembler and instruction editing. PEybass also gave the team a basic glimpse into working with an obfuscated binary file.

3.4 VGA and timestamps

The analysis done for the parasite challenge required a deep dive into both timestamps as well as the technical specification of the VGA standard. When looking into the data given from the challenge one needed to figure what the data represented. By looking into different log standards each different datapoint needed to be analysed and figured out. When looking at the first data point one needed to research different timestamps and figure out what it meant. Therefore, looking deeper into UNIX timestamps, on what UNIX timestamps actually measure and what it is presented as, as well as looking into how it represents fractions that datapoint was demystified. One learned a lot more about what actually UNIX timestamps are from this challenge. Looking into the other data points and reading schematics and specifications for the VGA standard also provided insight into not only the challenge but also the VGA standard as a whole. By looking at the actual wires in a VGA connector as well as how each signal in said connector represents data one could discern the other data points.

References

[1] L. McDaniel, E. Talvi, and B. Hay, “Capture the flag as cyber security introduction,” in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, 2016, pp. 5479–5486.

[2] K. Blashki and S. Nichol, “Game geek’s goss: linguistic creativity in young males within an online university forum (94/\/\3 933k’5 9055oneone),” *Australian journal of emerging technologies and society*, vol. 3, no. 2, pp. 71–80, 2005.

[3] M.-W. Dictionary, “steganography definition.” [Online]. Available: https://www.merriam-webster.com/dictionary/steganography

[4] I. J. Kadhim, P. Premaratne, P. J. Vial, and B. Halloran, “Comprehensive survey of image steganography: Techniques, evaluations, and trends in future research,” *Neurocomputing*, vol. 335, pp. 299–326, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218312591

[5] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2003, pp. 290–299.

[6] B. Schwarz, S. Debray, and G. Andrews, “Disassembly of executable code revisited,” in *In Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE.* IEEE Computer Society, 2002, pp. 45–54.