Cracking the Nixu Challenges

Joakim Argillander, Christian Wahl

Abstract—In this report we illustrate a subset of the main topics of this year's NIXU Challenges (2019). Through our work of solving these challenges, we discuss the attack vectors made possible by exploiting the vulnerabilities or intrinsic weaknesses of the challenge topics. Procedures for mitigation of the threats are suggested where applicable and design practices that can aid in detecting ongoing exploits are suggested elsewhere.

Index Terms—NIXU challenge, CTF, Reverse Engineering, Forensics

1 Introduction

The cybersecurity company *NIXU Cybersecurity* is presenting their annual information security challenges, the *NIXU challenges*. Participants are given the task of finding so-called *flags* in a system that are to be submitted on the challenge website.

The challenges are related to a selection of areas within memory forensics, reverse engineering and networking. The purpose of the challenges is to attract potential employees to the company, and to serve as a recruitment platform for the consulting company.

In the following, we show our solutions to a subset of the challenges presented by NIXU in the areas of *Data Exfiltration*, *Memory Forensics* and *Reverse Engineering*.

2 Network Traffic Exfiltration

In order to send data outside of controlled premises it is oftentimes desirable to hide the data traffic. This is useful in cases where attackers' malware sends acquired data out of company networks from within, or if an insider employee steals intellectual property from its employer. Conventional tunnelling of data is often detectable, and it is very clear what is going on if the network traffic is monitored. The data is not readable in such cases, but the amount of outgoing traffic can be sufficient to determine that an exfiltration attack is ongoing.

These attacks are particularly dangerous, as the data is sent via normal protocols, remotely, while hidden in plain sight.

2.1 Data Exfiltration over DNS

A common way of exfiltrating data is to tunnel it over a DNS request. Seemingly normal DNS requests pass through a firewall inconspicuously, even though the requests contain a custom payload. DNS tunnelling is a very stealthy communication way that is often used to silently send data, and fetch commands to be executed by malware [1].

This can also be used for bypassing payment portal for services such as customer internet access at hotels, for example. Those services can be exploited particularly easy if the firewall accepts outgoing DNS requests. If so, it is reasonable to believe that the hotel does not have local DNS servers on site that enforce the limitation but is using a purchased service from a payment portal provider. This enables exploiters to send HTTP data in the payload field of the DNS request. The payload is usually formatted as a domain query where the domain part is the payload encoded as a hexadecimal string. The request is then made to a server the exploiter controls, which then acts as a proxy for relaying the HTTP requests and responses.

Tools such as *Iodine* [2] have been developed to make DNS tunnelling easier, and allows data to be encoded as a base64-string in the requests for A-records. The receiving server is set to respond with the payload within NULL-responses, in which payload of up to 16 bits may be used [3]. This method supposedly can reach up to 1 Mb/s downstream, but with a highly asymmetrical bandwidth thus yielding a much lower upstream bandwidth. [2]

2.2 Data Exfiltration over UDP Port Numbers

One of the challenges demonstrated another form of data exfiltration, but this time over UDP. The data was encoded into the source port numbers of datagrams sent over the network. This data is then retrieved on the server and reassembled into data. In this way, UDP packets can be sent with dummy payload that can contain seemingly inconspicuous data, while the real exfiltration is done by the port number modulation.

2.3 Mitigation

Although some novelty use cases, e.g. pay-wall evasion, for exfiltration exploits exist, these can also manifest in the form of real attacks on corporate's intellectual property. It is of importance that enterprises who wish to uphold confidentiality of internal data need to take measures to mitigate potential attack vectors.

2.3.1 Verifying DNS Host Name

To mitigate data-exfiltration over DNS tunnelling, the requests' host-name may be verified with a trusted third party DNS provider such as, for example *Google* or *Cloudflare*. This is, in practise, done by monitoring each outgoing DNS requests and verifying the host-names before either allowing the DNS request coming from the suspected exploiter to go through, or just returning the response to the verification query. If the query is legitimate, both of the queries should return the same response [4].

2.3.2 Disallow Rapidly Changing Source Port

If an UDP data exchange is followed, and many packets are being sent from within a network to an outside host, and the source port numbers are changing with each datagram sent, it should raise suspicion. It can be said, with some degree of accuracy, that it is an attempt of exfiltration of data. As the data amount that can be sent with each UDP packet is very limited (16 bit), it would require a substantial number of messages being sent in order to exfiltrate any real data. By imposing a limit on the number of source port number changes of datagrams to the same destination host within a certain time interval before packets are dropped, mitigation of UDP port number exfiltration may be achieved.

3 Memory Forensics

Memory forensics is the act of retrieving data about a particular computer's state at a given time. The techniques of memory forensics often used in law-enforcement purposes where a computer is analysed for proof of criminal activity, but it is also used as an advanced attack vector targeting an entire computer. This is done by means of *memory dumps*, where the current working memory of the computer is recorded and often extracted to another machine for thorough analysis. By recording the volatile working memory of the target computer, the current state of the target computer can be saved and analysed to find traces of actions taken by the users, malwares and other software.

3.1 Method and tools

The memory forensics challenges were almost exclusively solved by using the *Volatility* framework [5]. *Volatility* is a comprehensive open-source framework that can perform analysis such as process listing, process memory extraction, file extraction and other operating system specific operations. Only some knowledge of how the program's data is stored in memory is needed in order to interpret the extracted raw data and to retrieve data from the raw data. For common Windows programs, much information is available on the Internet for how to interpret data.

A memory profile has to be determined for the memory dump such that *Volatility* knows where in memory to look for symbol tables and which algorithms to use for extracting data. After running the `imageinfo` utility, the framework can correctly identify the image as coming from a Windows 7 SP1 x86 - machine.

Worth mentioning is that memory forensics is easier when something known is searched for. In the case of the *NIXU challenges* the keys' format is known (`NIXU{FLAG}`), searching for the flag format can serve as an initial hypothesis.

3.2 Process Memory Extraction

After listing the processes running on the machine, three noteworthy programs are open and running:

- `notepad.exe`
- `paint.exe`
- `lsass.exe`

3.2.1 `notepad.exe`

Notepad is Windows' default text editor, and works by storing the text into a memory buffer which is only saved to a file upon the user's command. No auto-saving of the text buffer is done, and it can safely be assumed that any content that is visible in the text area of the editor exists in the process' memory.

The assumption is proven to be correct as the memory content can be retrieved by dumping the process memory to a file, running `strings` and either manually analysing the content of its output, or piping to `grep` if parts of the content is known. In the case of these challenges where the format of the flag is known, it could easily be `grep`'ed for,

In reality however, in a forensics use case, analysts would have to manually search through the outputted strings that were found by the `strings` tool. Depending on text buffer size and operating system, this can be more or less difficult.

It is also worth noting that as the text is stored into the process' memory only, it does not matter if it has been saved to file or not. As long as the text is visible in the editor, it exists in the memory, and thus can be retrieved if a snapshot is taken with the process running.

3.2.2 `paint.exe`

Much like in the case with `notepad.exe`, the graphics' buffer of Windows' default raster graphics editor, `paint.exe`, stores the current drawings of the canvas in the process memory. The graphics buffer exists in memory entirely even if not written to a file.

By dumping the memory and opening it in a graphics editor as a raw input file, the canvas can be restored. Any metadata or other content of the process can be treated as noise, as the canvas itself occupies the majority of the data.

It is known that *Microsoft Paint* saves the current canvas in a PNG-like format, and the data can therefore be extracted as each pixel row of the image is stored sequentially.

3.2.3 `lsass.exe`

The *Local Security Authority Subsystem Service* is a process running on *Microsoft Windows* that authenticates user accounts, handles verification of user passwords and updates of user passwords. This is an exceptional process to target with *Volatility* by dumping its process memory, and then running it with the *Volatility* plugin `mimikatz`.

According to the `mimikatz` source code, the `lsass.exe` service stores the passwords encrypted with a reversible cipher (AES or DES) [6] in memory. The plugin searches the process memory for the correct memory position in which it can find the decryption key for the passwords. After it found the key to decrypt the passwords, `mimikatz` shows the unencrypted passwords to the attacker.

3.3 File Extraction Using the Microsoft Windows File Cache

Windows caches files in memory for quicker access times. This means that files can be retrieved as part of the memory dump with the tool `filedump` in *Volatility*. The framework iterates over the *Virtual Address Descriptors, VADs* to retrieve files that are known to contain data.

3.4 Microsoft Windows-specific Graphical User Interface Extraction

Volatility has great tools for targeting Windows' graphical user interface.

3.4.1 Clipboard retrieval

By targeting the Windows system's clipboard array and cross-referencing its *user handles*, the contents of the recent clipboard copies or cuts can be easily retrieved. This is particularly easy if the data is text content.

In the scope of this project, the clipboard retrieval was done and contained a clue that implied that the flag was to be found somewhere related to text editing. This led to the analysis of the process dump for *Notepad*.

Copied passwords or other sensitive information is made available to a potential attacker in cleartext. This is particularly dangerous as not many users may be aware that the data of the clipboard is retained after pasting. This could potentially mitigate long, strong passwords entirely, as they are more prone to be copied and pasted than shorter ones that can be memorized.

3.4.2 Window Environment Extraction

Volatility is also able to generate wire-frame diagrams of the window locations in the Windows graphical interface. The extracted diagrams show the x, y and z position of each window that is open. In the context of these challenges, it was used to see what programs were running in windowed mode.

These extraction tools do not necessarily give complete data from a running process but can be used as little clues as to what the user is currently doing with the machine, from a user's perspective, rather than just what processes are running and their data.

This can give an attacker a good view of what the user was looking at, at the time of memory extraction. This does not reveal any content of the windows, but can give an attacker a good view of what the user was doing, and use this to infer for example, usage patterns and corporate workflows.

From an information security point of view, the safekeeping of intellectual property is as important as securing the data on the system. This implies that, if important intellectual property can be attributed to work processes or windowed tools, it is still worth protecting.

However, this feature can still be argued to be more useful for memory forensics in search for evidence of crimes as it can give good evidence of what the user was doing at the time of extraction.

3.5 Mitigation of Malicious Memory Analysis

Common for all these examples is that the content of a running process can relatively easily be dumped and retrieved with more or less *point-and-shoot* tools such as *Volatility*. As there is no reasonable way of preventing a memory dump from occurring, focus should lie on obfuscating the memory content instead.

3.5.1 Process Memory Encryption

At initial glance, encryption of the memory content appears appealing. The only issue with nave encryption of the memory is that the decryption key must also reside in the memory in order for the encryption to work, thus rendering the encryption useless if an attacker is able to obtain a complete memory dump, as it would, along with the process data, contain the decryption key. For an experienced attacker, this would pose no real challenge, while it may serve as deterrence for simpler attackers if considered a practise of *security by obfuscation*.

As memory forensics is considered advanced it can be assumed that attackers employing such techniques are more akin to *advanced persistent threats*, thus increasing the need for other real protection mechanisms.

3.5.2 Microsoft Windows Graphical User Interface Attack Mitigation

Mitigation of user interface attacks starts at the user. By employing practises that sensitive data cannot be copied nor pasted, the risk of having, for example, passwords leaked through extraction of the clipboard is reduced.

Window location information is guarded, if deemed necessary, by employing practices that windows are to be minimized upon leaving the computer unattended.

However, while these security practises are theoretical measures that mitigate the risk of an attack, they are not realistic to implement in reality. It might not be easy to implement procedures that require that much security awareness by employees in reality, unless the data on the computers is of extremely high security class. If so, then there should already be other measures in place that more efficiently guard against such threats, see Section 3.5.3.

3.5.3 Limiting Physical Access

If the memory content cannot be secured in itself, another way of securing the data is to limit physical access to the machine from which a memory dump is wanted. The two most common ways of acquiring memory dumps is either to dump the memory on site, or to extract the memory in a laboratory environment. As the working memory is volatile, the computer must not be powered off, or else the data is lost. Simply making sure that computers are powered down when left unattended will mitigate the risk of having any data maliciously extracted.

Worth noting is that memory dumps can be acquired over a network connection, and open source solutions exist for this purpose. This can be mitigated through usual network security, and practises for that should already be in place for organizations with rigorous limitations of physical access.

4 Reverse Engineering

We solved two different kind of *reverse engineering challenges*: The first one (*Lisby*) is a previously unknown architecture that needs to be disassembled or implemented in order to receive the flag. This challenge will be described in Section 4.1. The second challenge focused on exploiting buffer overflows together with the usage of string manipulation functions that use a length limit.

4.1 Reverse Engineering of an Unknown Processor Architecture

In this section we focus on illustrating and solving a challenge that dealt with an unknown processor architecture. This means, that there are no publicly available tools that help during this process. Although, the challenge introduction provided us with an implementation specification outlining all possible instructions and with an overview over the architecture [7].

The background story for the challenge tells us that this machine was developed "*decades ago*" and used to run these *Lisby* programs directly. In order to solve this challenge, we followed the hint from the challenge introduction to start with 'disassembling' the binary [8]. Consequently, we implemented a framework to disassemble *Lisby* binaries. However, disassembling the first challenge revealed a large number of assembler instructions that we did not intend to evaluate by hand. Thus, we developed a *Lisby* implementation in *Python* that helped us to solve the challenge by evaluating the *Lisby* binary directly. With this tool we were able to solve these challenges.

If we apply the background story of this challenge now to the world of legacy systems, we see that the intention of the *Lisby* challenges might be to raise awareness for old systems. As we saw in the challenge, old systems might either still contain algorithms that are necessary for businesses and might need to be reverse-engineered or contain information that a company wants to reconstruct or aids an attacker to exfiltrate data.

The background story could also be an example of legacy systems that are still run by banks and rely on programming languages that can only be written by a limited number of people (namely COBOL) [9]. As these programmers began their career during the early days of computers, these people have retired already and might die soon, take their knowledge with them and thus create a huge problem for the industry.

If viewed from a different standpoint, one could also say that an unknown architecture might give a competitive advantage as not everyone is able to reverse engineer or make sense of a binary (an implementation of *security by obscurity*).

4.2 Find Possible Exploits in a Given Source Code

The challenge was called "Device Control Pwnel" and might resemble a program that might be found in an early stage of development. The challenge was written in the *C* programming language and used the `fgets` function in combination with a consecutive `strcpy` function call.

In theory this combination could be secure as `fgets` will terminate the buffer if it gets data and not write over the size of the buffer that was given as a parameter. However, in this challenge the authors provide either a larger intermediate buffer to `fgets` and copy the contents over to the final buffer that is smaller or supply a too large size parameter directly. This leads to buffer overflows that need to be used in order to enable flags that open the possibility to view the flags for the challenges.

In a software project this might not be done intentionally, but a sign of bad documentation or templates or even bad training. If one does not know the real size of the buffer one might as well use the `sizeof` operator to specify the size that the buffer has. In order to mitigate these types of attacks one should either use dynamically allocated buffer with which the length is known at run-time or generate these method calls with templates which include precautions to safeguard against buffer overflows. Although, it might be even better to not use functions that need a length attribute that could be specified wrongly (i.e., employ data-structures that know their length at run-time).

5 Conclusions

The challenges have demonstrated a plethora of exploits and attacks that can be performed to either gain entry, silently exfiltrate data or extract data from an unknown system.

Network data cannot be just assumed to contain whatever data is implied by the standards the packets adhere to. Neither can it be assumed that data is not sent through other fields in the network packets. Even though the non-payload-carrying fields (like lifetime, etc.) are not generally settable by high-level programming languages that do not support raw sockets, data can still be sent in non-essential parameter fields by advanced attackers and exploiters by using custom made tools.

The operating system that was investigated as part of the memory forensics challenges was shown to store much more data in memory about the current state of the machine than initially thought. Intuitively, the process' state must be stored in working memory at *some* point in time, but it is not before a memory dump is analysed with forensic tools that it becomes clear how much is actually retained in memory and essentially retrievable.

Furthermore, an unknown processor architecture might hide information on the first sight but might not hide the information or proprietary knowledge if one has access to an architecture description. Additionally, lesser known systems might have another impact that is not security related: They might include previously available algorithms or information, but these might not be available anymore as there are no people left that are able to process this information.

The challenges have also shown that issues with memory management arise whenever low-level languages are used. Much of this is abstracted in higher level languages, and buffer overrun attacks are very rare in these languages. It serves as a word of caution about how the complexity changes when working closer to the hardware, and how user input cannot be trusted. This is especially true in cases where bounds checking is not implicitly done by the programming language.

References

[1] C. Marrison, "Dns as an attack vector and how businesses can keep it secure," *Network Security*, vol. 2014, no. 6, pp. 17 – 20, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1353485814700613

[2] Iodine. Visited on 2019-05-06. [Online]. Available: https://code.kryo.se/iodine/

[3] L. Nussbaum, P. Neyron, and O. Richard, "On robust covert channels inside dns," in *Emerging Challenges for Security, Privacy and Trust*, D. Gritzalis and J. Lopez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 51–62.
[4] S. Bromberger, "Dns as a covert," *National Electric Sector - Cyber Security Organization, US Government*, 01 2011.
[5] V. Foundation. Volatility memory forensics toolbox. Visited on 2019-05-06. [Online]. Available: https://www.volatilityfoundation.org/
[6] community/mimikatz.py at volatilityfoundation/community. Visited on 2019-05-06. [Online]. Available: https://github.com/gentilkiwi/mimikatz
[7] Nixu challenge 2019. Visited on 2019-05-06. [Online]. Available: https://thenixuchallenge.com/c/lisby-1/static/README
[8] Nixu challenge 2019. Visited on 2019-05-06. [Online]. Available: https://thenixuchallenge.com/c/lisby-1/
[9] A. Irrera. Banks scramble to fix old systems as it 'cowboys' ride into sunset - reuters. Visited on 2019-05-06. [Online]. Available: https://www.reuters.com/article/us-usa-banks-cobol/banks-scramble-to-fix-old-systems-as-it-cowboys-ride-into-sunset-idUSKBN17C0D8