University of Zurich[UZH]

MASTER PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Analysis of the NimPlant Command-and-Control

Anton Crazzolara, Aleksandar Ristic,
Karim Khamaisi, Samuel Brügger
Zurich, Switzerland
Student ID: 17-729-831 ,19-745-017,
19-740-067, 18-919-498

Supervisor: Dr. Bruno Rodrigues, Jan von der Assen,
Prof. Dr. Burkhard Stiller
Date of Submission: February 1, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

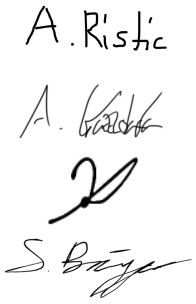
ifi

Master Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

We hereby declare that we have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). We are aware that we take full responsibility for the scientific character of the submitted text ourselves, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zurich, 1. Februar 2024



Abstract

With the growth of the Internet and a perpetual increase in electronic devices, botnets have become increasingly threatening to private and public networks. Using Command-and-Control (C2) frameworks, malicious actors can perform large-scale attacks on single devices or entire networks. One such framework is NimPlant, which went open-source in February 2023. Once a device has been infected and becomes a bot, it can be difficult to identify that bot in a network. Modern network security systems heavily rely on present information about known malware, making them poor at detecting novel threats, such as NimPlant. This project studied the NimPlant framework to guide the detection of NimPlant bots and other bots with similar communication patterns by monitoring network traffic. We deployed a NimPlant server in a testing environment with an infected device and developed strategies to both detect the bot using intrusion detection systems and to avoid said systems by improving the NimPlant framework. We then equipped a Reinforcement Learning algorithm with these evasion strategies to find how AI can improve C2 systems. Our findings show that AI systems can indeed improve the evasion capabilities of C2 systems. On the other hand, a proper setup of an intrusion detection system has a large impact on the performance of such AI systems and the detection rate of bots through network traffic. Based on all results, we discuss some recommendations for better detection of bot infections.

Abstract German

Im Zeitalter des Internets und mit einem konstanten Zuwachs an elektronischen Geräten, sind Botnets über die vergangenen Jahrzehnte zu einer immer grösseren Gefahr für private und öffentliche Netzwerke geworden. Mithilfe von Command-and-Control (C2) Frameworks können sogenannte Botmaster schwere Angriffe auf einzelne Geräte oder ganze Netzwerke ausüben. Ein solches Framework ist NimPlant, eine Software deren Code im Februar 2023 auf GitHub veröffentlicht wurde. Sobald ein Gerät mit einer solchen Software infiziert ist, kann es schwierig sein, das infizierte Gerät in einem Netzwerk zu entdecken. Moderne Netzwerksicherheitssysteme verlassen sich zu grossen Teilen auf spezifische Erkennungsmerkmale von bereits bekannter Schadsoftware. Eine Schwäche dieser Systeme ist somit unerforschte Bedrohungen wie NimPlant zu entdecken. In diesem Projekt haben wir uns der Untersuchung von NimPlant gewidmet, um Empfehlungen für das Entdecken von NimPlant und ähnlichen Bots durch die Analyse von Netzwerkaktivitäten zu formulieren. Zu diesem Zweck haben wir in einer gesicherten Umgebung einen NimPlant-Server eingerichtet und ein weiteres Gerät mit dem Bot infiziert. Als Nächstes haben wir Strategien entwickelt, um zum einen die Kommunikation mit dem Server auf dem infizierten Client zu entdecken, und zum anderen Strategien, um vom Server aus genau dies zu verhindern. Anschliessend haben wir zudem einen Reinforcement-Learning-Algorithmus mit diesen Ausweichstrategien ausgerüstet, um herauszufinden, wie KI C2-Systeme verbessern kann. Unsere Resultate zeigen, dass solche KI-Systeme tatsächlich die Entdeckbarkeit eines C2-Systems erschweren können. Zudem wird aus unseren Resultaten ebenfalls klar, dass wohldurchdachte Sicherheitssysteme einen grossen Einfluss haben auf die Leistung eines solchen KI-Systems und die Erkennungsrate von Bots durch den Netzwerkverkehr. Unsere Resultate nutzen wir zudem, um einige Empfehlungen für das Bekämpfen von Botinfektionen zu diskutieren.

Acknowledgments

We would like to express our deepest appreciation to Dr. Bruno Rodrigues for supervising this project, providing guidance, support, and invaluable advice. Our weekly meetings were always helpful and motivating, and a major contributor to the smooth progression of this project. We are also grateful to Jan von der Assen for his supervision and advice on AI models in particular. Special thanks go to Prof. Dr. Burkhard Stiller for providing the opportunity to undertake this project as part of the Communication Systems Group and allowing us to present our results in front of the research team. Lastly, we thank the Department of Informatics IfI for providing us with the needed infrastructure for this project.

Contents

Chapter 1

Introduction

This chapter will present the motivation for this project, the thesis goals, the methodology used and close with the thesis outline.

1.1 Motivation

Due to their continuous operation and significant vulnerabilities, the Internet is teeming with machines that create an ideal habitat for malicious actors to create and propagate botnets in the contemporary digital landscape [3, 14]. These botnets pose a significant threat to cybersecurity because they comprise a network of compromised computers, colloquially known as "zombies," which attackers can remotely exploit to carry out various damaging activities [14]. These include Distributed Denial of Service (DDoS) attacks, spamming, data theft, and other fraudulent actions.

Not only are these criminal activities widespread, but they also have grave consequences for both businesses and individuals. The infrastructure established by these botnets is primarily responsible for the widespread nature of DDoS attacks, email spamming, illegal acquisition of confidential information, and cyber fraud. Moreover, profit-driven cybercriminals frequently employ botnets, leading to the emergence of an entire underground economy based on them. This illegal activity causes significant economic harm, affecting individuals, businesses, and even entire nations.

An attacker can use a system known as a Command-and-Control (C2) framework to orchestrate and manage a botnet effectively. This system gives them a centralized platform to manage the botnet's infected computers efficiently. This includes sending commands to compromised machines, collecting data, and updating the malware to infect additional computers. Due to the severity of these botnets' threats, network traffic monitoring is essential for mitigating potential losses. Detecting C2 activity within a network is essential to initiate a prompt and effective response. Security teams can vigilantly monitor network traffic to identify atypical communication patterns between internal systems and external endpoints. In addition, they can detect traffic patterns that match known C2 traffic signatures.

These suspicious activities may involve encrypted traffic, non-standard ports or protocols, or any other atypical network interaction that deviates from the norm. Identifying these early indicators of C2 activity permits security teams to contain the threat quickly. They can inhibit the possibility of data exfiltration and remove malware from compromised systems, thereby reducing the scope of the attack and mitigating the risk of further compromise. Thus, security teams can protect their digital infrastructure and its sensitive data and information.

NimPlant is a C2 targeted for initial infection before deploying elaborate malware to the targets [56, 57]. As such, NimPlant provides a lightweight framework aiming to strike a balance between the ability to evade detection systems and still provide C2 functionality. Therefore, NimPlant, by design, only permits functionality deemed benign and could be applied by legitimate (remote access) tools that restrict the use of shellcode executions but permit basic filesystem operations to evade detection systems. Critical to this master's project is the investigation of automated malware evasion, mainly through Artificial Intelligence (AI) within Command-and-Control (C2) systems. Malware based on artificial intelligence can modify its behavior to blend in with its surroundings, making it much more difficult to detect.

One prominent application of AI in this domain is within C2 systems, which can optimize communication patterns and camouflage with regular network traffic, decreasing the likelihood of detection and increasing the potential damage [4, 9].

There are also other studies similar to this project where AI was used for malware evasion purposes such as embedding malware inside video conference software [33, 34], using reinforcement learning to manipulate portable executable files [5], using generative adversarial networks to hide malware communication by simulating legitimate network traffic [42] or using evolutionary packers to hide malware binaries [20].

In the initial phase of this project, a comprehensive examination of NimPlant and other cutting-edge botnets will be conducted to delve deeply into this rapidly evolving field. Therefore, understanding these advanced threats' tactics, techniques, and procedures will provide security researchers with invaluable insight. This information is crucial for developing effective countermeasures and fortifying defense systems against future AI-enhanced threats. Adopting a proactive approach to studying AI-based evasion techniques makes it easier to anticipate these threats and develop early detection algorithms.

1.2 Thesis Goals

From the described motivation result the following goals for the project:

- The project should include an overview and comparison of related work in the field of C2 frameworks, comparing major characteristics observed in NimPlant with similar state-of-the-art technologies.

- We create a testing environment by setting up a client and server environment, decoupled from real network devices. We also install adequate monitoring tools on the client that allow us to monitor its network traffic. In this environment, we deploy NimPlant by configuring a Nimplant server on the server machine, and injecting and executing an implant on the client, simulating a real application of the framework.
- Utilizing the installed monitoring tools, we collect and analyze metrics on anomalous patterns, ports, protocols, and spikes in traffic to unknown destinations and usage of encrypted traffic. The results should facilitate identification of major characteristics that enable the detection of NimPlant C2 network traffic patterns.
- We use NimPlant to deploy further, different purpose malware on the infected device, capitalizing on NimPlant's first-stage infection identity.
- With the acquired information on NimPlant network traffic patterns, we manually configure different modes for the NimPlant server to run in. These modes adhere to different evasion strategies. We further evaluate the testing scenario with respect to the individual evasion modes.
- Based on the designed evasion modes, we develop an AI system to implement adaptive evasion strategies on the C2 server. We also evaluate the AI system and the results it yields.
- From all of our acquired results, we deduce recommendations for improving Intrusion Detection Systems (IDS), in particular with respect to C2 botnets.

These goals were critical to the procedure of the project, the required infrastructure, and the evaluation of the conducted activities. They do not necessarily correspond to explicit sections of this work.

1.3 Methodology

To achieve the goals described in the previous subsection, the project was divided into different phases. The first phase consisted of a review of existing literature covering topics that are essential to this project. In particular, this included a survey of similar studies, books, or reports. While we were mostly focusing on white literature, a couple of technologies, such as NimPlant and some monitoring tools, were not sufficiently covered by previous works and required the inclusion of official documentations and blog posts, or a manual review. Some of our findings were used to support the planning of the project, provide input to the choices of methodologies applied in this project, to validate the motivation and necessity of our own work, and to ultimately compare our results to the results of similar previous works. Additionally, this phase contained an investigation of the fundamentals of botnets, offensive and defensive strategies, NimPlant itself, and AI in cyber security.

In the second phase, the group installed the environment that would be used for the rest of the project. This included the installation and configuration of several hardware

and software components, such as a client and server machine, the NimPlant server, and monitoring and intrusion detection tools. This phase also contained the deployment and initial operation of the entire system and first tests of the capabilities of the C2 system and the defensive tools, including but not limited to the injection of additional malware through NimPlant. While continuously testing the system, incremental improvements were made to the defensive aspects and several evasion strategies were developed on the offensive side of the system. A notable product of this phase was a collection of major characteristics that allow the identification of network traffic originated by NimPlant.

The third phase involved the selection, design, and implementation of an adequate AI system with the purpose of developing adaptive evasion strategies on the offensive side. This technology was then faced with and tested on different layers of defenses on the infected client. For the reinforcement learning algorithm, Q-learning was used. The learning environment, the reward function, which is based on the Snort alerts and the sever were adapted to the NimPlant application. The agent was able to enable and disable strategies, got rewards and improved the information it had about the environment. All the data from the learning process and the results were then analyzed and discussed with the additional goal of developing recommendations for the defense.

1.4 Thesis Outline

The rest of this project report is organized as follows. Section 2 covers the fundamentals of several topics relevant to the understanding of this work and to the classification of several activities performed during this work. It also contains a survey of existing literature and work related to or covering topics that are substantial to the goal and contents of this work. Additionally, Section 2 closes with a brief discussion of NimPlant, its relevance and the state of modern cyber security. In Section 3 we present and discuss the design aspects of all stages of this project. Section 4 describes all implementation parts of the stages of the project. Section 5 constitutes the evaluation of our systems at the individual stages of the project and covers our intermediate findings. Section 6 finishes with a summary and conclusion of our work and discusses some opportunities and insights for future work.

Chapter 2

Fundamentals

The fundamentals which include the background and the related work are presented in this chapter to provide fundamental information helping the reader understand the concepts addressed in this work. Additionally, a discussion of the fundamentals including NimPlant is added.

2.1 Background

This section describes some major fundamental concepts related to this project. It covers information about Command-and-Control frameworks such as NimPlant. Further the detection, defense, and evasion of C2 malware and botnets is presented. The section is finally concluded with a subsection about reinforcement learning.

2.1.1 Command-and-Control

Botnets are a type of malicious software (malware) and are considered to be a major threat to public and private services [12, 24, 48]. A botnet is a network of infected machines called bots, a short form of "robot" [31]. The sizes of these networks are often unknown, though individual networks can reach sizes of up to a million bots. Experts in 2012 estimated that 16 to 25 percent of computers with access to the internet are part of at least one botnet [48]. Botnets are used for various malicious activities, including Distributed Denial of Service (DDoS) attacks, forms of spam, phishing, data theft, identity theft, creation of backdoors, and more [31, 48]. Most of the malware botnets use is designed to target the MS Windows operating system, making it the main target for infections [48]. Despite being commonly known for their malicious purposes, botnets can also be used for legal activities, such as to defend against DDoS attacks or to be used as organizational tools [31, 54].

The main characteristic that differentiates botnets from other types of malware is that they function based on a Command-and-Control (C2) architecture [12, 24, 48]. Each botnet

is controlled by a so-called botmaster [48]. The C2 architecture allows a botmaster to connect to the bots over relatively stable communication channels to send commands to all bots in real time [24, 35]. The connection to the botmaster is not only decisive for the robustness, stability, and reaction time of the system, but it is also the most vulnerable part [26, 48]. Originally, C2 botnets were centralized systems in which a single botmaster server would communicate with the bots over Internet Relay Chat (IRC) channels [24, 31]. This made systems traceable and vulnerable, and botnets could be effectively dissolved by shutting down the botmaster [31, 48]. For this reason, in the early 2000s, decentralized C2 architectures and alternative, safer communication channels became more popular [31, 48]. Instead of IRC communication channels, modern centralized C2 systems increasingly use HTTP/HTTPS channels and decentralized systems mostly rely on Peer-to-Peer (P2P) communication [31].

Bots can spread through different means, such as websites, peer-to-peer (P2P), email attachments, file-sharing, or previously installed backdoors [48]. The installation of bots usually happens in three phases [31]: In the first phase, the initial injection, an attacker infects a potential bot using various injection techniques, such as spam emails or phishing. In the second phase, the secondary injection, the infected machine downloads and installs relevant malware binaries. In the last phase, the bot connects to the C2 server to receive commands. Different approaches exist to establish this connection [26]. Possibilities for establishing this connection include hard-coded IP addresses, using alternative types of internet infrastructures, and using third-party services. The connection to the commander can be permanent or reestablished whenever needed [24]. Commands can be delivered to bots in a ”push” or ”pull” fashion. The former sends commands immediately to the bots. The latter lets bots request their commands from the commander regularly [24]. Bots are typically initialized every time the victim device is started [48].

Internet of Things (IoT) devices are modern everyday life devices that can communicate over the internet to improve quality of life. In smart cities, the number of IoT devices is very high and still growing, and despite that, these devices are often weak in security [35, 48]. This makes them popular targets for botnets. Online social networks have become an important and large part of network traffic. This makes them another popular target for botnets, offering a completely new way for C2 networks to communicate with their bots [26, 31, 48]. These networks are called Social Botnets [31]. Cloud computing, another technology growing in popularity, opens further possibilities for the creation of botnets, resulting in so-called Dark Clouds [31].

The distributed nature of botnets makes it difficult to track them down and hinders law enforcement, especially across countries with inconsistent laws [48]. The lack of certification for applications available for download, especially on mobile devices, presents a significant security weakness. Research in this area is handicapped by the limited access to large numbers of devices and realistic network protocols. The latter are often considered business secrets or contain other sensitive information. This makes it difficult to study real botnets and evaluate the effectiveness of security systems [48].

2.1.2 NimPlant

NimPlant is a C2 framework partially written in the programming language Nim [57]. C2 allows remote communication with malware implants, typically in a client-server architecture. The functionalities of C2 frameworks can differ substantially between different frameworks, though, most are designed to be flexible and extensible to allow personal adaptations. Even though NimPlant had already been developed back in 2021, its creator made it open-source in February 2023 by releasing the project on GitHub [56, 57]. It is therefore a recent and up-to-date technology. At the current date, Nimplant implants only target x64 Windows machines.

On its GitHub page, Nimplant is described as a lightweight and configurable implant. It encrypts and compresses all traffic by default [57]. In addition, NimPlant is written in Nim, compiles directly to C, C++, Objective-C, or Javascript, and uses XOR-encoding for static strings, concealing the binary, and making it inherently harder to trace [56, 58]. It also supports multiple implant types, including self-deleting executables. Once the server is running and an implant is placed, users have access to a wide range of functionalities with a focus on early stage operations. The implants can be operated from the server's browser interface, seen in Figure 2.1, showing a list of all implants and allowing users to enter commands directly into a built-in console. This also constitutes a decent logging system [57]. Some further notable useful features of the web interface are a kill server switch, which shuts down the server and all active implants, and a download section, containing the data retrieved from the agents. With so many accessibility features, NimPlant is beginner-friendly and effortless to use.

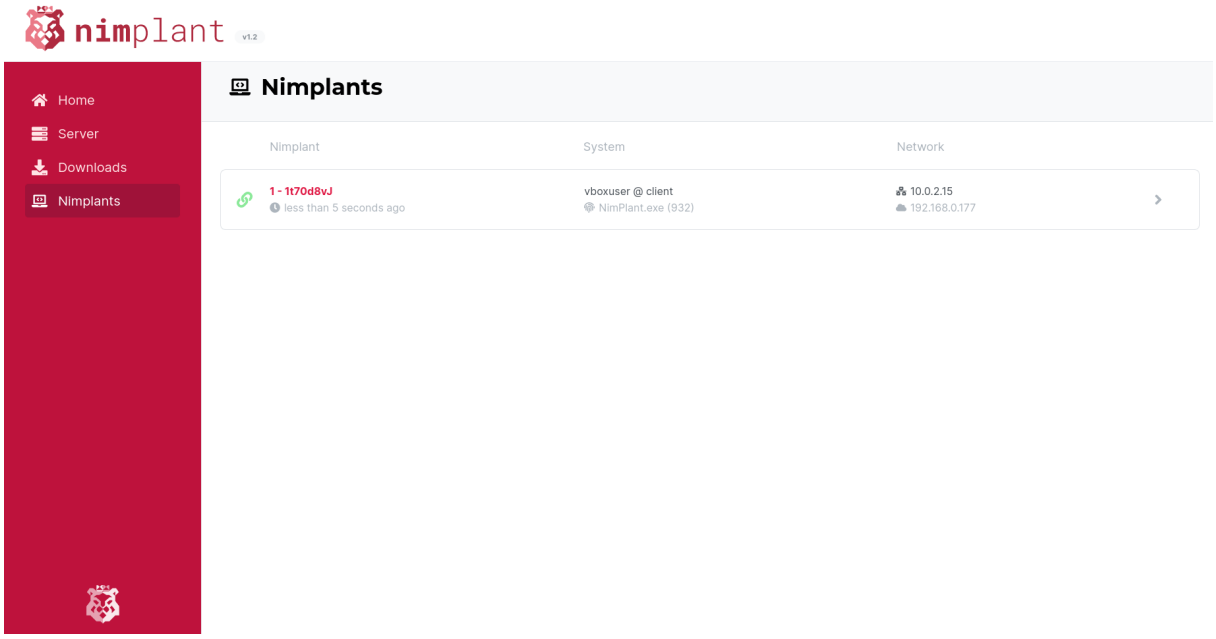


Figure 2.1: The overview of connected bots in the server's browser interface.

NimPlant's intended use is for the initial infection of systems (first stage) before injecting more elaborate malware as a follow-up. Its main requirements are, therefore, to be

lightweight, evasive, and functional. Since this initial implant would provide an entry for further malware, these requirements would be relieved from the other malware [56].

Per the developer's advice, detecting NimPlant on an infected device should not be done by focusing on specific tools. Instead, one should look out for generic suspicious activities, such as untrusted binaries or unusual processes communicating through the internet. The reason is that NimPlant does nothing that other malware has not done before. Additionally, getting hold of any relevant binaries or retrieving memory dumps of running NimPlant processes (and subsequently finding the encryption key) should suffice for identifying unambiguously malicious activities [56].

2.1.3 Detection

Command-and-Control traffic is similar to normal, non-malicious traffic. The traffic volume is low, and there may be very few bots in the monitored network. Additionally, the traffic may also contain encrypted communication. Such characteristics make detecting Command-and-Control traffic challenging [24]. But one weak point of the Command-and-Control traffic that can be used to detect botnets is that bots of a botnet can have spatial-temporal correlations and similarities. This results from their pre-programmed response activities to control commands. It means that at a similar time, the bots within a botnet will execute the same command and report to the Command-and-Control server the progress or result of the task, with the reports being similar in structure and content. Normal network activities are not likely to have a much synchronized or correlated behavior [24]. The most common approaches to botnet Command-and-Control detection are packet header analysis and deep packet analysis [29].

2.1.3.1 Packet Header Analysis

This method analyzes packet headers to distinguish between malicious and normal traffic. One way would be to use header elements like source and destination IP for the distinction [29]. Packets sharing common attributes such as source and destination IP, source and destination port, and protocol type can be collected, and unique flows can be identified based on these attributes. Those unique flows represent the fingerprint of a packet, and the attributes can then be used to determine whether the packet is distinctive or shares common attributes with other packets. This information can then be used to identify network anomalies, which can help classify traffic into normal or malicious ones [11]. Another possible way to analyze packet headers would be to perform a n-gram analysis. A n-gram analysis can be used against botnets that use Domain Generative Algorithms (DGA). When botnet malware is using DGA to decrease its probability of detection, then a n-gram score smaller than normal can be observed. This can help to identify whether normal or malicious traffic from a DGA botnet is present [53].

2.1.3.2 Deep Packet Analysis

Deep packet analysis deals with analyzing packet payloads, searching for specific malware signatures, or analyzing specific protocol traffic, such as Domain Name System (DNS) traffic, to detect anomalies [29]. Botnets use multiple domain names to connect to remote control servers, send spam mail, drive users to infection servers, and find victims. As accessing a server over the Internet with a domain name requires DNS, one can obtain evidence of botnet activities from DNS queries generated by botnets. There are also cases where botnets use hard-coded IP addresses to connect to the target servers, but recent botnets tend to use domain names or both IP addresses and domain names [36].

2.1.4 Defense

There are two main aspects of defense against botnets. One part is the measurements to reduce the possibility of an infected device, which can be grouped as device security measures and user sensitization [59]. The other part concerns active defense, a possible tactic when the device is already part of the botnet. The goal of the defense against botnets can either be local or global, meaning that one can either focus on getting the device and all the devices in the own local network out of the botnet or one can gather information and try methods to disturb or get the whole botnet to shut down [39, 62].

2.1.4.1 Device security measures

Using a firewall can help reduce the risk of getting infected by a botnet as it tries to filter out malicious network requests [51]. Installing an antivirus program can also be useful. Both security methods use many of the detection methods listed above [13]. Using an automatic updater for the used software on the device or keeping it manually updated is also a defensive measure, as it can reduce the risk of getting infected via a known security risk in an earlier version of the program [7].

2.1.4.2 User sensitization

One approach to initially infecting a user's device with a virus is using social engineering methods. Those try to bypass the security measurements by forcing the user to manually download, install, or execute a malicious file while pretending to be benign. Well-known social engineering methods for malicious purposes are phishing emails, spoofed websites, scareware, and similar URLs [43]. A reliable way to protect a user from such attacks is to sensitize the user to the possible hints and the newest attacks. General behaviour changes like not using links in emails, using multi-factor authentication for login, and only using safe sources for downloading files can prevent many possible infections [27].

2.1.4.3 Active Defense

If the goal of the infected user is to get information about the botnet or to get it to shut down, one can use active defense techniques. These are more sophisticated methods and need in comparison to the other defense mechanisms more knowledge of the subject. Some of those methods, like fuzzing, alter the response message that the infected client sends back to the C2 server. This can help to identify vulnerabilities in the C2 framework. Alternatively, there is a method called milking, where the user can try to impersonate an infected device and communicate with the C2 server to get information about commands and templates. With this information, one can try to halt or shut down the botnet, which is the key idea of active defense [62].

2.1.5 Evasion

The following chapters first describe which techniques are used to develop evasive malware based on malware analysis. Then, several mechanisms are described that are employed to ensure that the bot binary (which is executed on the victim's machine once the infection happened), Command-and-Control server(s), Command-and-Control communication, and botmaster are not easily detected [32]. Finally, the last sub-chapter deals with evasion using AI.

2.1.5.1 Malware Analysis

Malware authors have employed several techniques to develop evasive malware [1]. Mainly to stay stealthy, bypass detection mechanisms, and divert from the analysis process [2]. In the literature [2, 6, 41], we observed two prominent ways of analyzing malware:

- **Static analysis:** analyzing the source code without executing the malware. This type of analysis has become challenging since malware authors have been applying all types of code obfustication (like polymorphism, encryption, packers, and so forth).
- **Dynamic analysis:** a complementary process for static analysis, where the malware is executed in a safe environment, and its behavior is observed. To perform this type of analysis, there are two methods:
  - MANUAL: using debuggers (as the primary analysis tool) and other tools like Wireshark.
  - AUTOMATIC: utilizing the sandbox technology to automatically run the malware in a safe environment separated from the host machine. Automating the process makes it scalable and allows it to analyze several malware and threats [60].

For both analysis types, malware writers are constantly engineering new ways to hinder the analysis and thus evade detection [2]. Figure 2.2 shows the evolution in malware concealment techniques.

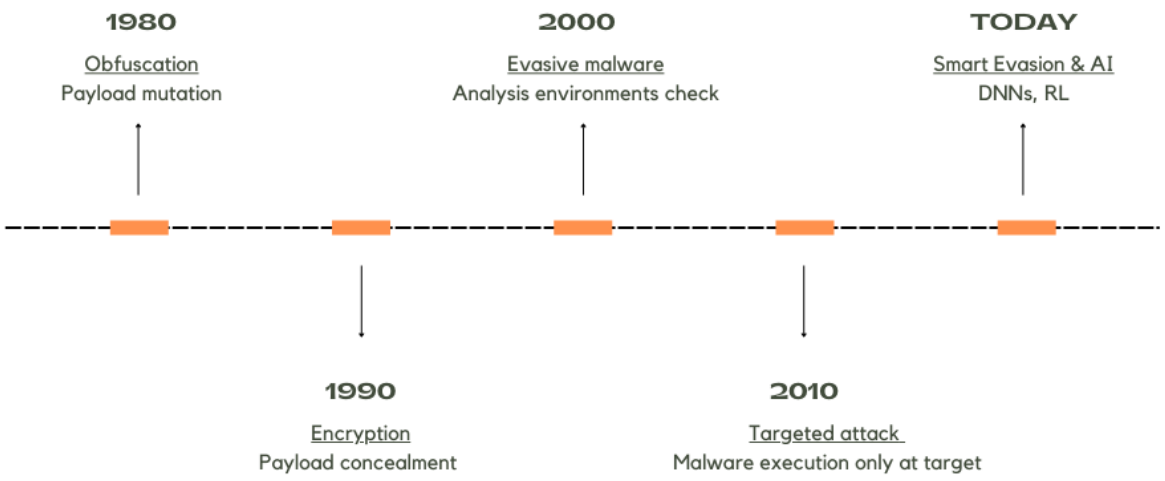


Figure 2.2: Evasive malware techniques evolution based on [33]

Malware that is not straightforward to detect uses various methods to identify whether analysis tools or environments exist. The most common methods include anti-debugging and anti-VM tactics, which use several techniques to obstruct the process of manual and automatic dynamic analysis [2, 10]. If the malware detects that it is being run on a VM, often used for analysis purposes or in a debugger, it will employ various techniques to disrupt the analysis process. Specific techniques are detection-dependent, where the malware looks for signs of an analysis environment and mounts an evasion action accordingly. We present selected tactics employed in the anti-debugging category based on [2]:

- **Fingerprinting:** The malware looks for clues from which it can infer the existence of a debugger. *E.g.*, reading and analyzing Process Environment Blocks (PEB), *i.e.*, reading the data structure that contains data about a specific process, one of the fields is the *BeingDebugged* field.
- **Debugger traps:** Introducing certain parts in the code that, when the debugger steps on, reveals its presence, *e.g.*, Exception handling.
- **Debugger Specific:** Utilizing exploits and vulnerabilities of specific debuggers.
- **Targeted:** Encrypting the malware payload using a key on the victim machine (*e.g.*, registry keys); thus, it can not be reverse-engineered without being run.
- **AI-Powered Keying:** Where an AI generates the decryption key based on the target-specific attributes.

In detection-independent techniques, the malware behaves the same regardless of the execution environment, and the employed evasive tactics do not vary according to the execution environment. Similarly, we present employed tactics in the anti-VM category based on [2]:

- **Stalling:** In an automated analysis, the time allocated for each sample is limited, and thus, the malware hides its malicious behavior for the post-analysis phase. *E.g.*, using sleeping techniques, injecting time-consuming code like a code that writes and reads from memory, or encrypting the payload using an easy encryption key and starting brute-forcing it.
- **Trigger-based:** The malware executes the malicious payload once it receives data about a pre-determined variable. It can be a system date, opening a specific window, or receiving data from the network.
- **Fileless malware:** The analysis environment has no executables to start analyzing. The malware exploits vulnerabilities in the target system and compromises a browser or a browser plug-in to inject malicious code into the memory. Malicious actors employ Windows PowerShell to carry out this task.

2.1.5.2 Evasion tactics at Bots

To evade host-based detection, several mechanisms are employed such that the bots remain available to the botmaster for an extended period of time [32].

Binary Obfuscation: The bot family expands by exploiting vulnerabilities on machines infected by the bot-binary. The bot-binary contains mechanisms to coordinate with the botmaster to receive commands. Several evasion techniques can be used to avoid being detected by host-based security applications and to hide the bot-binary. For example, using polymorphism can help against pattern-based detection. Polymorphism describes the ability of the bot-binary to exist in several forms. One way would be to use encryption, or another way would be by packing the bot-binary (file condensation). Packing can help hide the malicious code, and some packers can even produce new binaries whenever the original malicious executable is packed [32].

Security Suppression: A botnet can proceed and disable existing security software on the victim's machine once a successful infection happens. Other competing malware may also be wiped out if the host is already infected with them [32]. For example, Conficker can disable several security-related Windows services and registry keys after being installed [47].

2.1.5.3 Evasion tactics at Command-and-Control Servers

DNS-fluxing / IP-fluxing: DNS-fluxing, also called IP-fluxing, deals with frequently changing the IP address associated with the Command-and-Control server's domain name. This evasion tactic is efficient against IP-based blacklisting and blocking by detection and

defense systems. Botnets can use Dynamic DNS (DDNS) to keep the mapping of the Command-and-Control server domain name to IP address up to date in real-time [32].

Domain Generating Algorithms: A Domain Generating Algorithm (DGA) dynamically generates a large number of random domain names and then selects a small subset of these domains for Command-and-Control communication. The domains are computed based on a given seed, consisting of numeric constants, current date/time, or Twitter trends. The purpose of the seed is to serve as a shared secret between botmasters and the bots to compute shared gathering points. Generating and changing the used domains constantly makes detection based on static domain blacklists ineffective. DGA has several advantages for attackers. By dynamically generating domain names, the attackers do not have to include hard-coded domain names in their malware binaries, making extracting this information by the defenders more complicated. Additionally, suppose the generated domains depend on time. In that case, the value of domains extracted by the defenders from dynamic malware analysis systems is reduced because different domains will be observed at different time points. Also, using short-lived domains registered shortly before they become valid evades domain reputation services [40].

2.1.5.4 Hiding Command-and-Control Communication

Encryption: Command-and-Control communication can be encrypted to evade detection, which makes content-based analysis inefficient. This forces the defenders to rely on other traffic characteristics such as packet arrival times or packet length [32].

Traffic Manipulation: Botnets can purposely create low-volume Command-and-Control traffic spread over relatively large periods such that statistical and volume-based detection techniques become less effective [32].

Novel Communication Technique: Novel communication techniques such as social networking websites like Twitter can be used by botnets for communication. For example, the information-stealing botnet Brazen used Twitter to spread links that contained commands or executables to download. Bots subscribed to the malicious Twitter account to get status updates [32].

2.1.5.5 Evasion by the Botmaster

Stepping-Stones: Botmasters can hide their identity by setting up a number of intermediate hosts called stepping-stones. They are then placed between the Command-and-Control server and the botmasters. Examples can be network redirection services like proxies such as HTTP. The stepping-stones are then hosts compromised by the botmaster. Also, botmasters can use an anonymization network as a stepping-stone which offers the additional benefit of hiding the botmaster's IP address, making it hard to trace back the botmaster [32].

2.1.5.6 Evasion with AI

The use of AI to improve malware's capabilities is a relatively recent development. There are multiple use cases for AI in malware, such as using AI to hide malware code from detection or using AI for network traffic detection evasion [18]. A single representative method is presented in the following subchapters for each case.

Evolutionary Packers

Standalone software that encodes and compresses an executable program is called a packer. The encoding done by a packer can happen, for example, with common techniques like the Caesar chiper. Packers were initially used to mitigate reverse engineering and protect intellectual property [20]. In Microsoft Windows operating systems, packers are used for the Portable Executable (PE). The PE is the native file format for every executable in Windows operating systems, for example, for .NET executables. Since most shareware comes packed to reduce the size and to provide an added layer of protection, the packer is used to insert code to unpack the file in memory when the execution happens [23]. If a computer has anti-virus software, then the anti-virus software analyzes portions of code inside a PE. This can be done statically and dynamically. In the static approach, file sections are checked against a database of signatures of known malicious software. In the dynamical analysis, the program operations are tracked while a heuristic mechanism tries to recognize behavioral patterns of typical malware [20].

Packers can also be misused for malicious purposes to transform the executable binary of malware into another form such that it is smaller and/or has a different appearance than before. The goal in such a case is to evade signature-based detection by anti-virus scanners [25]. The idea behind obfuscation through evolutionary packers is to use malware that can evolve its packer such that a new encoding routine is created in each infection. The approach is based on evolutionary computation and embeds an evolutionary core directly in the malware. The evolutionary core can have the ability to learn and to be trained. Packers are generated through genetic operators, and the encoding routine is a Turing-complete evolutionary algorithm able to generate completely new algorithms. The encoding and the decoding functions are a randomly generated, variable-length sequence of x86 assembler instructions [20].

Those instructions perform operations understood and executed by x86 microprocessors widely used in personal computers and servers [21]. The instructions are directly handled as binary opcodes, so no compilation or linking phases are needed. The generation of new packers requires finding reversible assembly instructions (*e.g.,* INC or ROR) and small blocks of code that have a complementary one [20]. Reversible assembly instructions can be undone or reversed by applying the same instruction in the reverse direction. Executing an instruction and then executing its reverse leads to the original state of the data.

Since anti-virus software can even mark a few bytes as a signature, it is also necessary to partially shuffle the instructions. While the encoding and the decoding functions are created in parallel, only the decoding functions are included in the generated malware. To measure whether a packer is useful, the encoding and decoding routines are subsequently applied to the randomly generated sequence of bytes. If the final result differs from the

original sequence, the packer is disposed of. Otherwise, if the packer turns out to be useful, then the packer is used to obfuscate the malware. The Jaccard Similarity is also evaluated to assess the packer's fitness values to achieve maximal dissimilarity from the original code. Having the decoding routine embedded in the PE, once the new malware is executed, it will restore each part of the program in memory so the malware is ready for execution. The same code generation engine is used at run-time to mitigate heuristic-based recognition of behavioral patterns of typical malware [20].

Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a class of generative models that can be used to evade detection of malware [42]. GANs focus on generating new samples based on a distribution learned from a training set. They have two neural networks: a generator (G) and a discriminator (D). The generator is provided with a random vector z as input and is trained to generate fake data indistinguishable from the real data distribution. Conversely, the discriminator tries to tell whether a given sample came from the real or the generator data distribution. This is done by taking in a data sample and outputting a single scalar, which describes the probability that the data sample x is real. Each data sample x receives a probability $D(x)$ if x has a high probability. The discriminator believes that x is likely to be real. Otherwise, if x receives a low probability, the discriminator believes that x is more likely to be fake [22]. The training of the discriminator is done using two decks of data: the training data x that results from the real data probability distribution p_{data} and the data generated by the generator $G(z)$ [42].

GANs are specific Artificial Neural Networks (ANNs) [28], and loss functions play a significant role in ANNs, because they represent the error measure. They are based on the difference between the generated and the true values of the outcome. This means that they describe how closely the generated output of an ANN matches the true values. To improve the matching of the generated and the true values, the goal of ANNs is to reduce the loss function but also to prevent overfitting (which would mean that the model has memorized the training data rather than learned to generate new and diverse samples) [8]. The discriminator and the generator have their loss functions in GANs [42]. This means that the discriminator is trying to reduce its discriminator loss function and the generator's loss function. The discriminator loss can be interpreted as how well the discriminator performs distinguishing between the generated and the real data. Typically, the discriminator loss $J^{(D)}$ is mathematically described as following [42]:

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}} log(D(x)) - \frac{1}{2}\mathbb{E}_z log(1 - D(G(z)))$$

On the other side, the generator loss can be interpreted as how well the generator generates realistic data to deceive the discriminator. Usually, the following formula describes the generator loss [42]:

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_z log(D(G(z)))$$

2.1.6 Reinforcement Learning

This subsection provides an overview of reinforcement learning and explains the concepts of the Q-Learning approach which is applied in this project.

2.1.6.1 Overview of Reinforcement Learning

Reinforcement Learning (RL) is a category of machine learning. It is classified as neither supervised nor unsupervised learning [37, 50]. Akin to other techniques of machine learning, reinforcement learning is used for making predictions and helping with decision making. It is generally based on probability theory and optimization [37].

In reinforcement learning, there exists an agent that faces a problem in an oftentimes dynamic environment. The purpose of the agent is to approach this problem by trial and error and to develop strategies that optimize the outcome and possibly achieve a given goal. Initially, the agent is usually not given any strategy and instead must discover them by trying randomly [30, 50]. To achieve these things, the agent must be aware of the environment and must be able to evaluate the outcome.

A reinforcement learning model traditionally can be divided into four subelements [50]:

- **Policy:** A policy is usually the core of the system. It defines the behaviour of the agent and determines its actions. A policy can be simple rule-based decisions or complex computations.
- **Reward:** The reward signal allows the agent to evaluate the outcome of its actions. Underlying the reward function is a goal that the agent is supposed to achieve. The reward signal forms the basis of altering the agent's behaviour with every step and should lead to iterative improvements in the strategy.
- **Value function:** A potential value function is similar to a reward function. However, reward functions tend to favour short-term optimization. Opposite to the reward function, the value function is responsible for evaluating long-term effects and the potential of strategies. This is usually much harder to develop than the reward signal.
- **Model:** The model of the environment mimics the behaviour of realistic scenarios if the system is tested outside of a real application. It reacts to the agent's behaviour and acts as an input to the reward and value functions.

With respect to these components, reinforcement learning agents learn how to optimize the reward signal. By doing so, the agent relies on the concept of state, cause, and effect, and faces significant uncertainty [50]. This makes reinforcement learning the closest to how humans and animals learn when compared to other forms of machine learning. To model the interaction between states, actions, and probabilities, reinforcement learning uses the Markov decision process framework, which is intended to represent the fundamental features of artificial intelligence problems in a simple manner. A problem that

reinforcement learning systems tend to face, is the trade-off between exploration and exploitation. An agent may try to optimize a successful strategy and stop investigating new strategies. The value function may prevent this from happening.

2.1.6.2 Q-Learning

One of the possible options for learning an optimal action-selection policy is Q-Learning. It is a model-free learning method as it does not try to mimic the environment with a model of it but only relies on the reward of an action in the current state. Q is a function that gives a value to each state-action pair, which indicates the expected reward. The method uses a Q-table for this with the states as the indices for the rows and the actions for the indices of the columns and each cell holding the expected reward value. A learning process starts with the Q-table being initialised with the value zero everywhere, as no information was yet collected. The actions selection balances exploration, which means taking a random possible action of the current state, and exploitation, which chooses the best possible action of the current state according to the Q-table. After taking the action and receiving the reward, the Q-table gets updated according to the following update rule [50]:

$$Q(s,a) \leftarrow (1-\alpha) * Q(s,a) + \alpha * (R + \gamma * max_{a'} Q(s',a') - Q(s,a))$$

- s is the current state
- a is the current action
- *Q(s,a)* is the value of the Q-table for the current state and action
- α is the learning rate, that balances the impact of old and new information
- R is the reward for taking action a in state s
- γ is the discount factor, that weights future rewards to a value in the present

The Q-table gets updated for each action in only one place. Thus, to gather enough information, the updates continue until one of the termination conditions is reached. These are not unique to the Q-learning method. Common termination conditions can be the reaching of a maximum number of episodes, only absolute or relatively small changes in the Q-table updates occurring, or the optimal policy, which is derived from the Q-table has not changed for a longer period of time.

2.2 Related Work

The related work section describes scientific work that includes botnet infrastructure, detection and defense of malware, evasion of malware with AI and two C2 tools as a comparison to NimPlant.

2.2.1 Botnet Infrastructure

Previous works have investigated different aspects of C2 infrastructures to understand botnets better and create systems that can detect and defend against them more efficiently. In 2014, Gardiner *et al.* wrote a comprehensive report on the state of C2 channels. The main focus of their report lies in the mechanics and techniques used to establish different C2 channels [19]. They also provide introductory discussions of other aspects of C2 malware and present some case studies of C2-based attacks. An extensive analysis of all aspects of botnet communication was provided by Vormayr *et al.* in 2017 [61]. Their work examines botnet topologies and protocols and presents a taxonomy for botnet communication patterns.

In 2019, Jovanovic and Vuletic analyzed the dynamic behavior of the Mirai and Gafgyt (also known as BASHLITE) malware which are both powerful botnets used for DDoS attacks [29]. To observe the C2 systems from the perspective of typical network defense systems, they installed four Raspberry Pi devices, connected them to the internet, and infected them with the previously mentioned malware. Their work presents insights into communication, obfuscation techniques, and the Mirai and Gafgyt malware lifecycle. Compared to this study, their analysis is more observant and general and considers different C2 frameworks. A different study by Marzano *et al.* also investigates Mirai and Bashlite but focuses on the evolution of these botnets instead [38]. For this purpose, they studied 47 *honeypots* (mock-up bot hosts) over 11 months.

2.2.2 Detection

Network-based defense and detection systems are often insufficient to protect individual members of the network and offer too little help to fight malware on an infected machine. For this reason, in 2012, Etemad and Vahdani developed a host-based approach that identifies an infection on a host by analyzing the inbound and outbound network traffic [12]. This subsequently allows the filtering out of all malicious traffic to suppress all communication with the C2 network. Their work is relevant to this study since we are also interested in detecting a bot by analyzing an infected machine instead of a complete network.

The opposite approach is shown in a paper by Gu *et al.* from 2008 [24]. Instead of observing a single user's network traffic, they propose a detection system that analyzes traffic in a local area network to detect C2 channels based on network traffic anomalies. They argue that even without prior knowledge about a network, it should be possible to detect patterns in network traffic, given that all bots in a botnet work similarly. Their experiments showed positive results. However, this technique is less relevant to our study since we aim to detect an infection on an individual machine instead of finding a complete botnet.

There exist other techniques to expose botnets. Since many defense systems work by blacklisting known malicious domains, major botnet malware uses Domain Generation Algorithms (DGA) to avoid being blacklisted. In 2016, Tong and Nguyen presented a

DGA botnet detection scheme that applies DNS traffic analysis to identify botnets using randomly generated domains [53]. They tested their method with more than 300.000 domain names, including domains generated by several DGA botnets, and achieved a detection rate of 99 percent.

Network traffic monitoring is a common approach when detecting an infection on a host machine. However, for this reason, advanced malware applies several techniques to behave like benign traffic, making it difficult to detect it. Another approach is to monitor terminal processes on a machine to search for processes that might be executed by malicious software. In their study, Tobiyama *et al.* proposed a detection method based on such process behavior using feature classification in two different types of neural networks [52].

2.2.3 Defense

In 2006, Bayer *et al.* developed a tool that analyzes Windows executables by executing them in an emulated operating system environment [6]. This enables a secure analysis of unknown executables and helps understand unexplored malware. The results can, therefore, help detect malicious executables in the first place and help develop appropriate defensive mechanisms. However, the efficiency of such a tool depends on the number of executables to be analyzed. It is, therefore, most useful if another virus scanner has already found unknown executables or if detected malware needs to be further analyzed.

2.2.4 Evasion with AI

- **DeepLocker: How AI Can Power a Stealthy New Breed of Malware:** In 2018, IBM Research presented the DeepLocker tool as a new family of stealthy malware [33, 34]. In a proof of concept, they showcased an evasive ransomware attack embedded inside innocuous video conference software. They illustrated how AI could aid targeted attacks to evade detection. Besides other concealing techniques often employed by malware authors to increase the detection evasion rate, DeepLocker only reveals its malicious payload once it ensures it is being run on the target's machine. To this end, the tool leverages AI to identify the target. They demonstrated that indicators and characteristics like the victim's facial expression, geographic location, or voice can be used as the malware's trigger condition to disclose its malicious behavior. Especially worth mentioning is that the malware can be embedded inside benign applications, similar to their proposed Video conference application, that can spread and be downloaded by thousands of users without the malicious payload being detected or executed on any of these users' devices. Utilizing the complicated nature of Deep Neural Network (DNN) AI models to generate the decryption key of the adversarial payload makes it nearly impossible to recover the individual attack characteristics,*e.g.*, determining the individuals for whom the DNN will generate a valid key, such as in the case of facial recognition, remains an elusive task. Using the DeepLocker technique, a range of malware families can be concealed and encrypted inside benign software, and other AI models can be used to trigger the condition effect.

- **Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning:** In another work from 2018, researchers introduced a reinforcement learning agent endowed with a sequence of operations enabling the manipulation of portable executable (PE) files associated with malware [5]. These modifications included adjusting header checksums, appending unused sections, and renaming them. Crucially, these modifications were carefully designed to preserve the PE file format while introducing alterations that did not compromise the malware's fundamental behavior. Nevertheless, they discovered that the behavior of some mutations changed in a subsequent test. In rounds, they trained the Agent on various types of malware. Each training round allowed the Agent to perform up to ten modifications with a specified budget of 50,000 mutations. The Agent was rewarded with either 10 or 0 points depending on the effect of its action. They targeted a static anti-malware engine employing machine learning techniques. This engine provided binary verdicts- benign or malicious-based on extracted features, command sequences, and other traits from PE files. While static AV does not offer a 100% guarantee, they are a primary tool for pre-inspection. In their work, the authors used a black-box attack approach to simulate a realistic scenario. They claim this is one of the reasons why the success rate (decrease in median detection rate over the validation set) was moderate compared to other works (*e.g.,* using white-box, grey-box attacks), which partially achieved more than 90% evasion. In the context of their study, the researchers observed significant results concerning 200 ransomware samples from the validation set. Initially, these samples had a median detection rate of 52.5 out of 65 on VirusTotal. However, using the trained Agent, the detection rate notably decreased to 16.5% out of 65%. Interestingly, using a random policy to mutate the PEs from another 200 ransomware validation set samples decreased the median detection rate from 44.5 out of 65 to 9.5 out of 65.

- **Bringing a GAN to a Knife-fight: Adapting Malware Communication to Avoid Detection:** Maria Rigaki and Sebastian Garcia demonstrated in 2018 the use of GAN for generating network traffic to mimic other types of traffic [42]. They modified the network behavior of malware to mimic the traffic of a legitimate application to avoid detection. The malware they used was the open-source Remote Access Trojan (RAT) called Flu. Flu was modified first to receive the input from the GAN generator and then to adapt its network behavior to mimic Facebook messaging traffic such as Facebook chat. They used the Intrusion Prevention System (IPS) Stratosphere Linux IPS to evaluate the quality of the generated samples from the GAN generator. The Stratosphere Linux IPS system was chosen because it could model behaviors in the network and uses machine learning algorithms to detect those behaviors in the network. The IPS blocked all the traffic that did not look like Facebook chat, and the GAN generator was trained until the imitation of Facebook chats was sufficient to pass the IPS. The malware monitored whether it was being blocked by the IPS and used this information as a feedback signal to improve the GAN models. The experiment results showed that after enough training epochs, the researchers managed to reduce the number of blocking actions to zero, even with a relatively small dataset, which means that the malware can keep behaving like Facebook forever and not be blocked. One constraint that the researchers highlighted in the experiment was that the communication between malware and C2

server had to continue unimpeded, which means that the C2 channel of the malware had to be kept operative.

- **Malware Obfuscation through Evolutionary Packers:** In 2015, a group of researchers applied an obfuscation mechanism based on evolutionary algorithms [20]. The idea was to embed an evolutionary core in the malware to generate a different, optimized hiding strategy for every infection. They performed experiments on a Windows-based Operating System with an Intel x86 architecture and used two malware scanners. The malware had a high initial detection rate such that the malware scanner would detect it without the appliance of the evasion method. Applying the evolutionary algorithm led to the creation of a malware variant that had three different stages of the evolution of the packing mechanism. The results showed that the detection rate was high without the packing mechanism. However, as soon as the packing mechanism was applied, the detection rate decreased drastically. With every further evolutionary step, the detection rate decreased further.

	DeepLocker	**Evade Static PE Machine Learning Malware Models**
AI Model	DNN	Reinforcement Learning
Goal	Executes malware only on the target machine.	Modify certain instructions in the malware PE such that it cannot be classified as malware by ML anti-malware engines.
Results	Functioning proof-of-concept, in which Malware is embedded in a video app and encrypts payload.	Detection rate varies by malware family. For Ransomware, median detection dropped from 52.5 to 16.5 out of 65.
Method	Malware payload encrypted and embedded in benign app. DNN generates valid decryption key only when malware runs on victim's machine.	Iterates through multiple rounds with ten modifications each. RL model rewards based on anti-malware ML engine's classification.
Attack Type	Black-box attack	Black-box attack
Evasion Dimension	Malware Binary	Malware Binary

Table 2.1: Evasion with AI part 1

	Bringing a GAN to a Knife-fight	**Evolutionary Packers**
AI Model	GAN	Turing-complete evolutionary algorithm
Goal	Mimics legitimate network traffic,	Hides malware binary.
Results	With sufficient training, full evasion is possible, meaning a detection rate of 0.	With increased evolutionary steps, the detection rate decreased. The last evolution step led to a detection rate of 1/57.
Method	Adapts the network behavior based on GAN generator input.	Adapts packers to encrypt the malware in different variations.
Attack Type	White-box attack	Black-box
Evasion Dimension	Network	Malware Binary

Table 2.2: Evasion with AI part 2

2.2.5 C2 Tools

- **Empire**: is an open-source tool employed within red team engagements to simulate the actions of actual adversaries. Initially introduced in 2015, the original framework was eventually archived, leading to its further development and maintenance being taken over by various forks. The most used and known fork is by *bc-security* [45]. Empire boasts an array of impressive attributes, including encrypted communication channels, support for Graphical User Interface (GUI) and Command-Line Interface (CLI) clients, execution of in-memory .NET assemblies, customizable bypass techniques, and more [45]. It supports profile creation to customize the behavior of each beacon. Of significant relevance is the observation that both Cobalt Strike and Empire leverage a common technique of *"malleable"* Command and Control (C2) listeners [46]. One can define and set profiles using the CLI. For example, it is possible to utilize the Dropbox profile, which emulates the behavior of legitimate communication with the Dropbox API [44], thereby further camouflaging the communication's purpose and origin:

```
1     (Empire) > uselistener dbx
```

Listing 2.1: Dropbox profile - Empire

A simple script profile defines the listener's instruction for data interpretation, extraction, and storage. Both Cobalt Strike and Empire can launch one single profile per Empire instance [46].

- **Cobalt Strike**: is another post-exploitation toolset engineered to replicate the actions of sophisticated threat actors, thus mirroring the behaviors of adversarial entities [16]. Characterized by its commercial nature, Cobalt Strike (CS) is primarily tailored to serve red teaming endeavors. The cobalt strike toolset costs $3,540 per user for one year license, aligning with its comprehensive and specialized capabilities [16]. At its core, CS can operationalize various formats to carry its post-exploitation payload, known as the *"beacon"* (to be installed on the victim machine) [17]. A notable facet pertains to its integrated phishing mechanism, serving for the delivery task of the beacon [17]. A distinctive aspect of CS's methodology is adopting a deliberate "low and slow" communication strategy [15]. This methodology simulates the behavior of advanced malicious software, thereby rendering the communication inconspicuous and evading detection. It allows custom profiles to camouflage communication between the beacon and the C2 server as much as possible [55].

Table 2.3 presents a concise comparative analysis encompassing Nimplant, Cobalt Strike, and Empire.

	Nimplant	**Cobalt Strike**	**Empire**
Language	Nim/Python	Java	Python/PowerShell2.0
Release Date	Initial public release 02.2023	Initial public release 02.2012	Initial public release 10.2015
Configuration	Uses TOML file, which is flexible and extensible to allow personal adaptations.	Uses a flexible configuration, that allows for various modifications.	The configuration is modular to allow operator flexibility. There are large amounts of configurable parameters.
Interaction	Web GUI & CLI	Web GUI & CLI	Web GUI & CLI
Platform Support	Windows	Windows, MacOS X, and Linux	Windows, MacOS X, and Linux
Main Purpose	Information gathering, first-stage infection	Commercial tool for Adversary Simulations and Red Team Operations	Post-exploitation and adversary emulation framework
Evasion Support	Encryption, compression, obfuscates static strings	Malleable C2 with changing network indicators to look like different malware each time	Adaptable communication profiles

Table 2.3: Nimplant and other C2-frameworks

2.3 Discussion

NimPlant was recently published; this master project analyzes and applies an up-to-date C2 framework. Although there is literature about using AI to evade malware detection, as shown in Chapter 2.2.4, no prior scientific paper has been released using NimPlant. This project aims to set up NimPlant in a safe testing environment, use NimPlant to deploy malware, and apply evasion strategies on NimPlant using AI. Evasion can be applied on several dimensions and with several strategies, such as using evolutionary packers on the binary dimension or GANs on the network dimension. AI can reduce the detection rate of C2 frameworks significantly with sufficient training, leading to powerful C2 frameworks. This is dangerous since it may incentivize hackers to empower their C2 frameworks with AI to increase their success when executing malicious work. Therefore, it is important to be one step ahead and to improve detection systems beforehand. By applying AI on NimPlant and analyzing the findings to propose detection strategies for enhancing detection systems, this paper can help prepare detection systems for future C2 frameworks that use AI for evasion.

Chapter 3

Design

The design chapter presents the scenario including the assumptions for this project, summarizes the snort rules and describes the design of the different stages of NimPlant such as using NimPlant with command-based evasion or with AI.

3.1 Scenario and Assumptions

For this project, the following scenario was chosen: An IT security employee hired by a company uses an intrusion detection system and defines monitoring rules based on his/her IT security knowledge. The reason for selecting this scenario is that it can be regarded as realistic as there might be cases in companies where companies hire IT security employees for such purposes. Related to the scenario, two assumptions regarding defensive models were made, resulting in two different defensive models:

- One naive IT security employee
- One expert IT security employee

Having a more naive and a more expert IT security employee is a reasonable assumption, as the level of experience of real IT security employees can vary as well. Additionally, the following assumptions had to be defined such that strategies could be implemented that worked on our testing setup:

- The system is always online.
- The intrusion prevention system analyses all incoming traffic on a given interface (i.e., internet connection) and alerts based on packet contents and headers. The client does not blacklist IP addresses.
- The network packets between different clients and the server are similar.
- The client device already runs the NimPlant executable.

The first additional assumption of having the system always online is a reasonable assumption since this is the usual goal that providers of client-server systems try to achieve to provide continuous services. The second additional assumption is necessary because of our limitations regarding the number of servers. Having only one server means that if the detection system detects the malicious network communication and then alerts or blocks every packet from that same IP address, NimPlant would have no chance to continue working on the affected client. Therefore, the intrusion detection system analyses packets and alerts based on packet content without alerting based on IP addresses. If there were more possible servers, then in such a case, the implant may change the server it is communicating with and continue working on the infected client. But on one side, we are missing the resources to use multiple servers, and on the other side, changing IP addresses of the same server would lead to an overextended complexity for this project. We further assume that the network packets between different clients and the server are similar since multiple clients would only differ in the IP addresses at the HTTP network layer. Finally, we assume the client device already runs the NimPlant executable. This is a reasonable assumption, as one can argue that the file could be transferred and executed on the victim's machine through methods like social engineering.

3.2 Snort Rules

The intrusion detection and prevention system Snort [1] was installed on the client during the setup of the testing environment. Snort was chosen as a detection system because it provides flexibility regarding the use of monitoring actions, allows the creation of custom rules, and comes with a large number of predefined community rules which can be applied to detect potentially malicious network communication. For this project, Snort was configured to alert based on the rules presented in Table 3.1. The design of these rules is based on the scenario, the assumptions, and the indicators of compromise described in the NimPlant network analysis in subsection 5.1.3. The rules are divided into naive rules and expert rules, following the assumptions of Section 3.1. The complete Snort setup, including all applied rules, can be found in the following GitHub branch: *configuration-stage-2*[2].

[1] https://www.snort.org/

[2] https://github.com/MAP-Cyber-Security-AI/Snort-Setup/tree/configuration-stage-2

Defensive Model	Nr.	Rule	Reason for Implementation
Naive	1	Alert packets that contain the keyword "nimplant".	Such keywords may indicate malicious network behavior, and therefore it is plausible that detection systems may send alerts when detecting them.
	2	Alert packets that contain the keyword "register".	
	3	Alert packets that contain the keyword "task".	
	4	Alert packets that contain the keyword "result".	
	5	Alert specific ports	Monitoring multiple ports generates more network traffic and presumably more unrelated alerts. Fewer ports are easier to manage, and some ports are at higher risk of attacks than others.
Expert	1	Alert all ports	Monitoring all ports negates the risk of overlooking attacks on unforeseen ports.
	2	Alert host header	Host headers are commonly replaced with domain names. IPv4 addresses in the header are against common practice and indicate that the packet might not originate from a trustworthy source.
	3	Alert packet frequency and packet size	Constant packet frequency may indicate keep-alive mechanisms and high frequency may indicate DDoS attacks, or other harmful behaviour. Receiving streams of packets of similar size may indicate procedural communication, such as keep-alive mechanisms.

Table 3.1: Snort Rules

The idea of having two defensive models is rooted in our assumptions, potential infection scenarios, and how security employees with various knowledge and experience would react upon discovering the infection.

- **Naive defensive model**: To simulate small-size enterprises without a dedicated IT security department and generally low budget for IT. In this defensive model, the Snort rules creator is assumed to look for apparent indicators, like the NimPlant keyword, known ports, and potentially suspicious keywords.
- **Expert defensive model**: This model simulates a more robust detection by alerting all indicators we discovered during our network and communication analysis phase

between the master and the bot (cf. Subsection 5.1.3). It also covers all evasion strategies we implemented in the NimPlant framework (both server and client-side). Thus, it simulates a more knowledgeable security analyst to better catch and detect NimPlant C2 traffic. It goes beyond simple static detection to include other dynamics found in C2 traffic, like different ports (to cover port hopping), frequency, and size of the packets related to the C2 communication.

3.3 NimPlant with command-based evasion

After the configuration of the rules, the initial NimPlant version could be detected and alerted by Snort. Thus, the NimPlant code was adapted at this stage to change strategies if commanded proactively by the botmaster. The following table 3.2 describes the strategies that were implemented and also could be enabled on runtime when giving commands to the server using the console:

Nr.	**Strategy**	**Console Command**	**Description**
1	Server name changing	Strategy One	When commanded, the server randomly changes its server name after each 100 client requests such that the server name is no longer ”NimPlant C2 Server”. The server uses server names like ”Apache”, ”GWS” or ”Domino” combined with numbers to simulate versioning.
2	User agents changing	Strategy Two	When commanded, the client changes its user agents after every 50 requests such that the user agent is no longer ”NimPlant C2 Client”. The client uses user agent names like ”Mozilla/5.0 (Windows NT 10.0; Win64; x64)”.
3	Port Hopping	Strategy Three	When commanded, the ports for the communication between client and server change every 10 minutes.
4	Endpoint changing	Strategy Four	When commanded, the endpoints of the client and server change such that the initial endpoints are no longer observable as long as the strategy is enabled.
5	Host header changing	Strategy Five	When commanded, the host header is changed to no longer show the server’s IP address but to show a randomly defined web page.
6	Command request frequency changing	Strategy Six	When commanded, the client changes the frequency for sending the command requests. Instead of the initial constant frequency, a varying frequency is initiated.
7	Packet size changing	Strategy Seven	When commanded, then the client and server requests are artificially increased to show a variety of packet sizes and avoid packets having a constant packet size.

Table 3.2: Strategies

The possibility of enabling or disabling strategies leads to the availability of a control mechanism that creates the base to allow NimPlant to test out strategies and even combine them.

3.4 NimPlant with AI

This section describes the process of the AI model selection, the assumptions for the training of the AI, the design of the Q-Learning, the chosen learning parameters and additional information that is relevant for the design of this stage.

3.4.1 AI model selection

First, we had several AI models in mind that could be used and also were used for evasion purposes, as we described in the related work in 2.2.4. Using evolutionary packers and DeepLocker was an interesting approach but is not well suited for this project as we focus on evasion at the network level. Therefore, we could work with two models: generative adversarial networks and reinforcement learning. We liked using GANs but encountered problems generating sufficient and diverse data. The requirements regarding data are much higher, as, for example, we would have to collect a sufficient amount of legitimate network data (for example, WhatsApp communications) and malicious network data (of NimPlant communication). Both data sets posed difficulties during the project as we had only one client and one server. Therefore, we decided to use reinforcement learning because the agent's training is less challenging as not two large data sets are required compared to the GAN. Reinforcement learning is also better fitted for our purpose as we could automate the activation and deactivation of strategies with machine learning. GANs would help to hide the general NimPlant communication by simulating legitimate network communication. Still, GANs would not help us in enabling disabling strategies and measuring the consequences of it. Having an agent enabling/disabling strategies and getting rewarded based on the outcome of its actions was more intuitive and had lower data requirements.

3.4.2 Assumptions related to Training of AI

We used a lab scenario with one infected client and the NimPlant server, where the strategies are controlled by the RL agent. A main goal was to test if the chosen AI method can be used to learn patterns for improving evasion. Two different agents were trained, one for the naive rules and one for the expert rules. Following assumptions were established to reduce noise and improve the learning results.

- The model has access to the snort alerts.
- The connection between server and client is reliable and NimPlant is not blocked.

- All strategy changes are done by the RL agent.

Getting access to snort alerts in a real scenario could be possible as a NimPlant connection to a client would enable the attacker to install additional malware, which could get access to the snort alerts and transfer them to the server. Because Snort does alert but does not block any traffic, it is realistic that a connection between client and server could continue even though several alerts would indicate a NimPlant infection. The case, where disconnections and blocking of traffic are considered is part of future work.

3.4.3 Q-Learning

In this project Q-Learning was used as the reinforcement learning algorithm. Each state has seven boolean attributes indicating if the strategies are enabled or disabled. This gives us two to the power of seven different states, thus the observation space has 128 distinct states. The agent has eight possible actions, which include actions to switch each strategy from enabled to disabled and vice versa and an additional action to do nothing. Thus, the Q-table has 128 rows and 8 columns. For the reward function we use the set of triggered alerts in the current state and weight each alert by a factor indicating how clearly this alert may indicate a NimPlant communication. For example if an alert is triggered, that indicates that the keyword ”NimPlant” is detected in the packet header, the alert is weighted with factor 5, because it can be seen as a clear indicator for NimPlant communication. In contrast if an alert was triggered, that indicates the detection of the keyword ”register”, then this alert is weighted with factor 1 as ”register” can also be a non-malicious network activity coming from legitimate systems.

3.4.4 Learning Parameters

The learning parameters were chosen based on current literature, documentation and testing from our side. All the parameters have values in their usual range for a Q-learning scenario. For the learning rate α, the value of 0.2 was chosen, which has the effect that the values are more stable and new information only update the values moderately. The Discount factor γ has a value of 0.95, which gives a high weight for future rewards. For the value of ϵ 0.2 was chosen, thus only in 20% of the cases a random action was chosen. The number of episodes was set to 100. This was long enough for the Q-table to converge and still not too long for the setup to run into problems. An episode starts in the initial state, where no strategies are enabled and runs until the done condition is met, which was in this case no Snort alerts and not more than five strategies enabled. The length of episodes can vary widely. In similar settings, episodes at the beginning took significantly longer than later ones.

3.4.5 Alert Reading

The required input for the running RL algorithm were the alerts generated by the intrusion detection system. For this purpose, Snort was run in a mode that writes alerts in a log

file at real time. Additionally, we equipped both the client and the server each with a Python script, which established a connection using Python sockets. The client's script read new log entries the moment they came in, applied some filters, and sent the relevant entries to the server, which then fed them to the RL algorithm.

3.4.6 Additional Adaptions

For the purpose of data collection, after each step all learning data like action, number of alerts, state and reward were saved. Also a fail save was built in where after each episode the current number of the episode and the most recent Q-table were saved. The fail save ensured that if the learning would have stopped then a restart would be able to start right after the last successfully completed episode.

Chapter 4

Implementation

This chapter shows the implementation of NimPlant with command-based evasion and with AI.

4.1 NimPlant with Command-based Evasion

The following section contains subsections which describe the implementation of the different strategies for evading the detection of NimPlant. The adaptions of NimPlant related to this stage were committed in the following GitHub branch *evasion-stage-2*[1].

4.1.1 Strategy One - Server Name Changing

To implement Strategy One, the *listener.py* in the server part, which is responsible for the communication with the client, was adapted to have a random server name generator. Another relevant file is the *server.py*, where a console input had to be added such that the botmaster could enable or disable the strategy using the console. Figure 4.1 shows the architecture of NimPlant with the most relevant files for this project. The marked files are the ones adapted for this strategy.

Since the *nimplant.py* is responsible for keeping track of the server state, three variables were added to this file. One variable was added to track whether Strategy One was enabled or not. The second variable is to reverse the server name back to "*NimPlant C2 Server*," if the strategy was enabled and then set back to be disabled. Finally, the third variable was added to save the number of client requests. Saving the number of client requests allowed the *listener.py* to check whether the desired number of requests has been reached to change the server name.

[1] https://github.com/MAP-Cyber-Security-AI/Nimplant/tree/evasion-stage-2

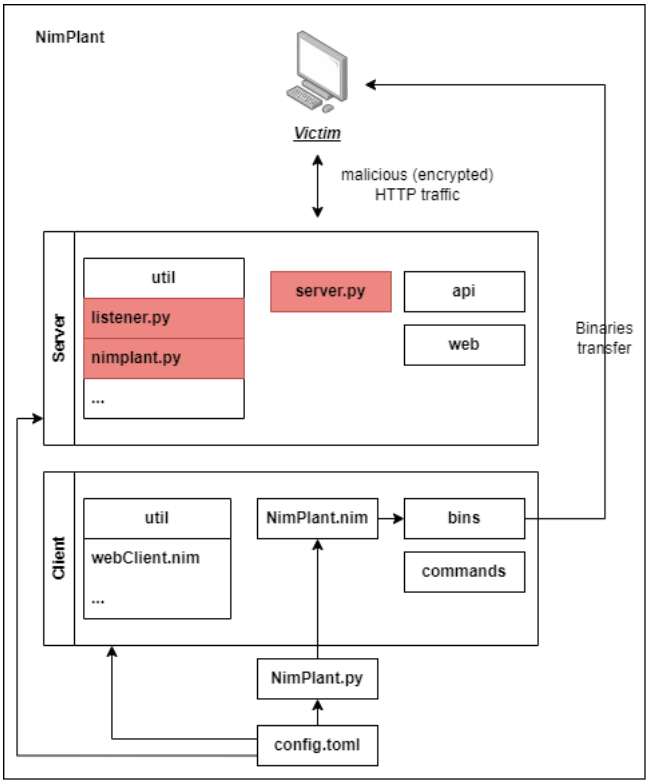


Figure 4.1: Strategy One: Server Name Changing

4.1.2 Strategy Two - User Agents Changing

The server and the client side must be adapted to enable the random user agent changing. Looking first at the server side, the files had to be changed as in Strategy One. The *server.py* and the *nimplant.py* were adapted the same as in Strategy One by adding console input in *server.py* and a variable to keep track of the strategy state in the *nimplant.py*. Figure 4.2 shows the affected files for implementing this strategy.

The third affected server file is the *listener.py* that needed two major adaptions. On one side, the *listener.py* has to communicate the state of whether the strategy is enabled or not to the client. This is why the file was adapted to encrypt the state and send it to the client after each client request such that the client gets informed, whether it has to execute the user agents changing or not. On the other side, the file had to be adapted to not only allow the initial user agent "*Nimplant C2 Client*", but to accept a list of possible legit-looking user agents like "*Mozilla/5.0 (Windows NT 10.0; Win64; x64)*". On the client side, the file *webClient.nim* had to be adapted to generate legit-looking user agents randomly if it detects on the server responses that the strategy is enabled.

Disabling the strategy would trigger the client to generate the initial "Nimplant C2 Client" user agent. To keep track of the strategy state, the *webClient.nim* was adapted to have a state variable updated after each server response. Finally, a counter was needed on the same file to ensure that after the desired number of requests have been reached, the legit user agent would be changed to another legit user agent if the strategy was enabled.

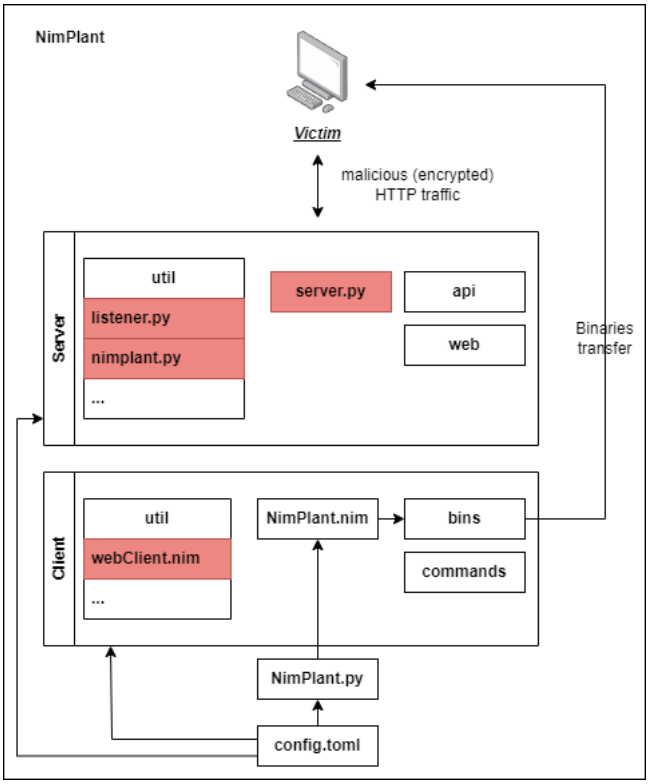


Figure 4.2: Strategy Two: User Agents Changing

4.1.3 Strategy Three - Port Hopping

The same files as in Strategy Two, shown in figure 4.2, had to be adapted. To enable port hopping, the *server.py* was changed, so the botmaster can use a console input to stop the current listener thread, change the port in the configuration for the new listener, and start a new listener thread. Ports are changed in 10 minutes and chosen randomly from a list of possible ports unless Strategy One is enabled, which changes the server name. In this case, the port will change when the server name changes, according to a dictionary shown in 4.1, where each possible server name has a port assigned. In the file *listener.py*, the `GET` endpoint for tasks was also adapted to return information about the next port.

```
1 ...
2 possiblePorts = [8080, 8081, 8082, 8083, 8084, 8085, 8086, 8087]
3
4 serverNamePortDict = {"Apache": 8080, "IIS": 8081, "Nginx": 8082, "
    Lighttpd": 8083, "NetWare": 8084, "GWS": 8085, "Domino": 8086, "
    NimPlant C2 Server": 80}
5 ...
```

Listing 4.1: Port numbers and associated services - *server.py*

On the client side, the *webClient.nim* was changed to receive the information about the next port and change to the target port. To register a new listener, the server will always keep the port from the configuration open. This port is defined at compile time as shown in 4.2.

```
...
# Configure listener port, mandatory even if hostname is specified
port = 80
...
```

Listing 4.2: Default Port number - *config.toml*

After registration and if Strategy Three is enabled, the target port is changed accordingly without having to re-register to the listener; thus, commands given over one port can be returned over another if the port is changed on the server side before asking for a task. If Strategy Three gets disabled, the server and the client will change back to the port set in the configuration file.

4.1.4 Strategy Four - Endpoint Changing

To further improve the evasion, the *config.toml* was changed such that the endpoint for the registration of the client is no longer `"/register"` but instead `"/r"` as shown in figure 4.3 and listing 4.3.

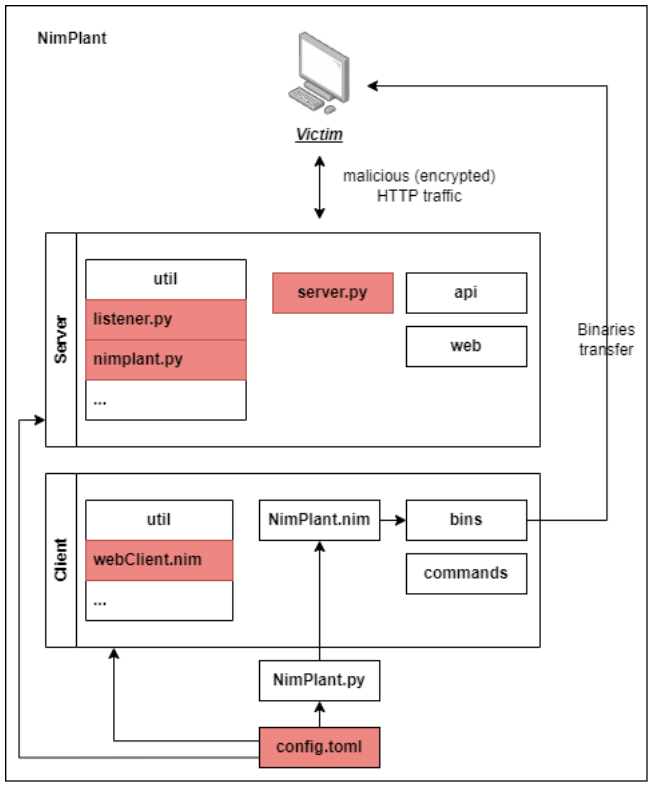


Figure 4.3: Strategy Four: Endpoint Changing

```
...
# Configure the URI paths used for C2 communications
registerPath = "/r" # previoulsy "/register"
taskPath = "/task"
resultPath = "/result"
...
```

Listing 4.3: Compile-Time Config: Endpoints - *config.toml*

To handle the input of the bot master, the *server.py* was adapted on the server side to enable or disable Strategy Four and the *nimplant.py* to save the state of the strategy. The *listener.py* had to be adapted on one side to communicate the Strategy Four state coming from the *nimplant.py* to the client, and on the other side, endpoints were added. The endpoints responsible for answering the client's `getTask()` requests, uploading files, downloading files, and getting results from the client were copied and adjusted with random endpoint names. Traditionally, endpoints like `'/task'` and `'/result'` are clear indicators of automated activity. To counteract this, we implemented a dynamic endpoint renaming mechanism, as illustrated in our proof of concept (PoC), where we replaced `'/task'` with `'/zero'` and `'/result'` with `'/zone.'` This modification was facilitated by the `changeEndPointsStrategy()` function on the client side, as detailed in Listing 4.4.

```
proc changeEndPointsStrategy(li :var Listener): void =
    li.taskPath = "/zero"
    li.resultPath = "/zone"

proc doRequest(li: var Listener, ...) : Response =
...
if li.changeEndPoints:
    changeEndPointsStrategy(li)
...
```

Listing 4.4: Overwriting Communicaiton Endpoints - *webClient.nim*

This function alters the endpoints for task listening and result submission based on the configurations of Strategy Four. The strategy can be extended to other endpoints from a predefined set of endpoint names or generate random endpoint paths. Those endpoints are only used if Strategy Four is enabled; otherwise, they are ignored.

Crucially, for the efficacy of this strategy, synchronization between the client and server is paramount. When an endpoint is changed on the client, the server must be concurrently updated to recognize and respond to requests on the new endpoint. This synchronization ensures seamless communication despite the dynamic nature of the endpoint paths. By implementing such a strategy, we demonstrate the potential to significantly reduce the detectability of bot traffic, as standard, easily identifiable endpoints are replaced with unconventional, varying ones, complicating the task for defensive network monitoring systems.

4.1.5 Strategy Five - Host Header Changing

Based on the inspection of indicators of the compromise phase, we extensively utilized community-based Snort rules. One such rule notably flagged instances where the Host header contained an IP address [49]. To address this, we employed Strategy Five, focusing primarily on the client-side aspect (mainly modifying *webClient.nim*), as the client consistently issued `GET` requests with its IP address in the header. We modified the Host header to validate this in our PoC, setting it to `"www.good-website.com,"` as shown in Listing 4.5.

```
1 proc doRequest(li: var Listener, ...) : Response =
2 ...
3 if li.changeHost:
4     headers.add(Header(key: "Host", value: "www.good-website.com"))
5 ...
```

Listing 4.5: Overwriting Host Header - *webClient.nim*

The underlying premise for such an alert could be substituting a domain name with an IP address, potentially enabling DNS bypass, diverging from standard web traffic patterns, and indicating possible command and control operations.

The server side needed the basic server adaptions on the *server.py*, *nimplant.py*, and *listener.py*, which were done the same as in the previous strategies to allow the input for the bot master, to keep track of the strategy state and communicate the strategy state to the client. Therefore, the affected files are the same as shown in figure 4.2.

4.1.6 Strategy Six - Command Request Frequency Changing

In Strategy Six, we aimed to disrupt the predictability of communication timing between a bot and its master server. Originally, the bot's ping intervals to the server for task retrieval were consistent, revealing a discernible pattern due to the sleep time being fixed at compile time and thus unchangeable during runtime, as illustrated in Listing 4.6 extracted from *config.toml*.

```
1 ...
2 # Configure the default sleep time in seconds
3 sleepTime = 10
4 # Configure the default sleep jitter in %
5 sleepJitter = 0
6 ...
```

Listing 4.6: Compile-Time Config: Sleep and Jitter Parameters - *config.toml*

As demonstrated in Listing 4.7, derived from *webClient.nim*, Strategy Six incorporated a `'jitter'` mechanism into the bot's request protocol to mitigate this.

```
proc doRequest(li: var Listener, ...) : Response =
...
if li.changeSleepTime:
    li.sleepJitter = 0.6
else:
    li.sleepJitter = 0
...
```

Listing 4.7: Activating Jitter in Strategy Six - *webClient.nim*

A jitter (variation of latency) of for example 0.6 means that the actual sleep time between requests varies randomly, with a maximum deviation of 60% from the set sleep interval. For example, if the default sleep time is 10 seconds, with a jitter of 0.6, the actual sleep time could vary between 4 seconds (40% of 10 seconds) and 16 seconds (160% of 10 seconds). This variability is introduced to make the bot's communication pattern less predictable and more akin to human-like traffic.

The importance of adding jitter lies in its ability to mask automated behaviors, making detection by network monitoring tools, which often look for regular, machine-like patterns, more challenging. By implementing this jitter, the bot's traffic blends more seamlessly with regular user traffic, reducing the likelihood of detection and enhancing the bot's effectiveness in an offensive security context.

As described in the previous strategy, the usual adaptions had to be made on the server side.

4.1.7 Strategy Seven - Packet Size Changing

Strategy Seven of this master project addressed the uniformity of packet sizes in bot communications, specifically in `GET` task requests. As demonstrated in the Wireshark capture 4.4, packet sizes remained constant before enforcing Strategy Seven. Upon activating this strategy, packet sizes began to fluctuate randomly, reverting to their original fixed size once the strategy was deactivated.

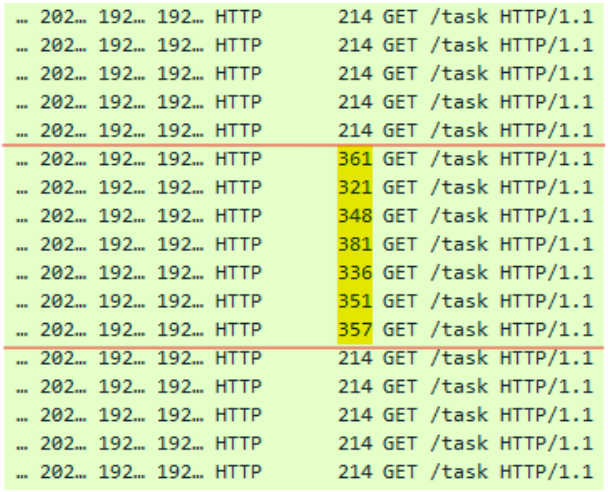


Figure 4.4: Packet Size Variation with Activated Strategy Seven

Varying the packet size is crucial in this context for evading detection by network monitoring systems. Fixed packet sizes, especially with consistent time intervals, indicate automated, non-human traffic. By introducing variability in packet size, the traffic mimics the less predictable patterns of human-generated network activity more closely, thereby reducing the likelihood of detection.

To achieve this variability, and for the sake of our PoC, we utilized the `'X-Request-ID'` header. Typically used to identify `HTTP` requests uniquely, this header was repurposed to add random-length strings ranging between 50 and 200 characters, as illustrated in Listing 4.8, adapted from *webClient.nim*.

```
proc doRequest(li: var Listener, ...) : Response =
...
# check if packet size shall change
if li.changePacketSize:
    # add a header with the random generated string
    headers.add(Header(key: "X-Request-ID", value: randomString(50, 200))
   ))
...
```

Listing 4.8: Random-Size Header Addition - *webClient.nim*

This randomization directly impacted the packet size, creating the desired variability. The choice of the header for this addition was necessitated by the nature of `GET` requests, which do not have body content to modify. Hence, the header was the only viable section to introduce this randomness. This approach underscores the innovative methods employed in offensive security to subtly alter traffic characteristics and evade detection systems, demonstrating the nuances and depth of strategic techniques that can be used.

To adapt the server for the artificial creation of content, a function was added in the *listener.py* that generated random content called `contentGenerator`, shown in Listing 4.9.

```
def contentGenerator():
    np = np_server.getNimplantByGuid(flask.request.headers
    .get("X-Identifier"))

    # Choose one random number
    randomNumber = random.randint(200, 700)

    # Ensure that the last two random numbers are not the same
    while abs(randomNumber - np_server.lastRandomNumber) < 50:
            randomNumber = random.randint(200, 700)

    # Update the last random number for future comparisons
    np_server.lastRandomNumber = randomNumber

    # Multiply random number with a character to generate the content
    createdContent = 'x' * randomNumber

    # Encrypt
    createdContentEncrypted = encryptData(str(createdContent), np.
   cryptKey)
    return createdContentEncrypted
```

Listing 4.9: contentGenerator - *listener.py*

The function uses a range between 200 and 700 to generate a random number. The random number is then multiplied with the character 'x' to create the artificial content, encrypted, and finally added on one side to the header and on the other side to the body of the server request. The range between 200 and 700 resulted from tests that assured a sufficient variety in the resulting packet sizes. Increasing the range on the lower part would increase the risk of still generating alerts, as the resulting content may be in the range of the alert. Increasing the upper range would increase the risk that the server may create a third TCP segment when sending the requests, which would increase the risk that the third segment would have a content size in the alert range as Snort checks each TCP segment of a request. Therefore, a compromise for the range was found to reduce both risks. To further increase the variety, the function ensures that the last two randomly generated numbers must have a difference equal to or larger than 50. Other server adaptions that impacted the *listener.py*, *server.py*, and *nimplant.py* were the same ones as in all previous strategies.

4.2 NimPlant with AI

This section aims to give detailed information about the implementation and integration of the AI model into the NimPlant environment. The code snippets are in some cases cleaned up and unnecessary code, for example for data collection or for the fail save, was removed. The adaptions of NimPlant related to this stage were committed in the following GitHub branch *evasion-stage-3*[2].

4.2.1 Q-Learning

The Q-Learning training function is implemented in such a way, that it takes as input the learning environment, the alpha value, which is the learning rate, that balances the impact of old information and new information, the gamma value, which is the discount factor, that revalues future rewards to the current time, the epsilon value, which is between 0 and 1 and balances exploration and exploitation, and the number of episodes the learning algorithm should run. The function call is shown in Listing 4.10.

```
1    env = NimPlantEnv()
2    env.reset()
3    nimplantPrint("Waiting 30 seconds for client to connect . . .")
4    time.sleep(30)
5    nimplantPrint("Started Q_learning")
6
7    Q_learn_pol, Q_table = Q_learning_train(env, 0.2, 0.95, 0.2, 100)
8    # env, alpha, gamma, epsilon, episodes
```

Listing 4.10: Call of Q-Learning - *server.py*

[2]https://github.com/MAP-Cyber-Security-AI/Nimplant/tree/evasion-stage-3

The Q-learning training starts with the initialization of the Q-table with small random values, as it is detailed in Listing 4.11. This is a practical approach for the beginning as no information is seen yet thus we want to explore more in the beginning and only later we want to exploit the gathered information more. For every episode the environment gets reset and the inner loop will run until a high enough reward was achieved. During these iterations the action gets decided either randomly, which happens in our case in 20 percent of the time or by selecting the action with the best Q-values, which happens in the other 80 percent of the time. After the decision of the action is sent to the environment, it gets executed and the reward, the next state and the information if the reward threshold is reached to stop this episode, get returned. With the reward information the Q-table gets updated according to the update rule.

```
1  def Q_learning_train(env, alpha, gamma, epsilon, episodes):
2
3      #Initialize Q table of 128 x 8 size (128 states and 8 actions) with
       small random values
4      q_table = np.random.rand(env.observation_space.n, env.action_space.n
       ) * 0.01
5
6      for i in range(1, episodes+1):
7          state = env.reset()
8          reward = 0
9          done = False
10
11         while not done:
12             if random.uniform(0, 1) < epsilon:
13                 # Explore action space randomly
14                 action = env.action_space.sample()
15             else:
16                 # Exploit learned values by choosing optimal values
17                 action = np.argmax(q_table[state, :])%8
18
19             next_state, reward, done, info = env.step(action)
20
21             old_value = q_table[state, action]
22             next_max = np.max(q_table[next_state, :]) if q_table[
       next_state].size > 0 else 0
23
24             new_value = (1 - alpha) * old_value + alpha * (reward +
       gamma * next_max)
25             q_table[state, action] = new_value
26
27             state = next_state
28
29     policy = derive_policy(env, q_table)
30     return policy, q_table
```

Listing 4.11: Q-learning - *Qlearning.py*

After the training has completed the final Q-table and the optimal policy are returned. The optimal policy can be derived from the Q-table by simply selecting the action with the highest Q-value in each state, as showed in Listing 4.12.

```python
def derive_policy(env, q_table):
    # Start with a random policy
    policy = np.ones([env.observation_space.n, env.action_space.n]) /
   env.action_space.n

    for state in range(env.observation_space.n):  # for each states
        best_act = np.argmax(q_table[state])%8 # find best action
        policy[state] = np.eye(env.action_space.n)[best_act]  # update

    return policy
```

Listing 4.12: Policy derivation - *Qlearning.py*

4.2.2 Learning Environment

The python library gymnasium[3] was used to create the reinforcement learning environment as shown in Listing 4.13. For the action space eight discrete values from 0 to 7 were used, because they can be mapped to their according strategy by the index. As each state has seven boolean values indicating if each strategy is enabled or disabled, the possible observation space has 2 to the power of 7 possible combinations, which results in 128 states. The initial state will be an array of 7 values, which are all zeros, indicating that no strategy is enabled. A key component of the environment is the step function, which first checks whether the action selected in Q-learning is possible in the current state. In this case this is not necessary as in every state every action is possible. After the strategy which is assigned to the selected action has been triggered, the function waits 15 seconds to accumulate the alerts and then count them. Afterwards the state information gets updated, the reward is calculated, the function checks if the done condition is met and then returns all the information.

```python
class NimPlantEnv(gym.Env):

    def __init__(self, natural=False):
        self.action_space = spaces.Discrete(8, start=0)
        self.observation_space = spaces.Discrete(128)
        self.state = np.zeros(7, dtype=bool)

    def step(self, action):

        assert self.action_space.contains(action)
        done = False
        strategy = self.actionDict[action]
        self.trigger_strategy(strategy)
        self.action_time = datetime.now()
        print("Sleep for 15 seconds to count alerts ...")
        time.sleep(15)

        # Read and filter Snort alerts based on the time interval
        alerts = self.read_snort_alerts()
        if(action != 0):
            self.state[action-1] = not self.state[action-1]
```

[3] https://gymnasium.farama.org/index.html

```
22
23          reward = self.reward(alerts)
24
25          if(reward >= min_reward_to_done):
26              done = True
27          return self.state_to_index(self.state), reward, done, {}
```

Listing 4.13: Environment - *NimPlantEnv.py*

4.2.3 Reward Function

In Listing 4.14 the alert weight dictionary and the reward function are shown. An alert that is a good indicator of a NimPlant infection has a high weight, and an alert that could also be triggered by other programs has a lower weight. The range of the weights is from 1 to 5. For the calculation of the reward the alerts and the number of strategies enabled were relevant. If there are no alerts then the max reward of value 10 subtracted with the number of enabled strategies is the final reward. In the case where alerts would be triggered, the function would sum up the weights of the triggered alerts and the sum would then be multiplied with a factor of -4, which is the punishment per weight. Again the number of enabled strategies is subtracted from the reward.

```
1     self.alert_weights = {
2         1000301: 5, # nimplant keyword
3         1000302: 1, # register ->
4         1000303: 2, # task     ->
5         1000304: 2, # result   ->
6         1000305: 5, # nimplant keyword
7         1000306: 1, # register <-
8         1000307: 1, # task     <-
9         1000308: 1, # result   <-
10        1000309: 3, # thresholds 2 in 0.5 minute size 200<>300
11        1000310: 3, # thresholds 2 in 0.5 minute size 150<>200
12        1000311: 3, # thresholds 2 in 0.5 minute size 100<>150
13        1000312: 1  # IP_IN_HOST_HEADER
14    }
15
16    def reward(self, alerts):
17        number_of_alerts = len(alerts)
18        punishment_per_weight = -4
19        max_reward = 10
20
21        if number_of_alerts == 0:
22            reward = max_reward - self.state.sum()
23        else:
24            sum_of_weights = 0
25            for sid in alerts:
26                sum_of_weights+= self.alert_weights[sid]
27            reward = punishment_per_weight * sum_of_weights - self.state
    .sum()
28        return reward
```

Listing 4.14: Reward Function - *NimPlantEnv.py*

Chapter 5

Evaluation

The evaluation chapter describes the findings related to the initial NimPlant configuration, NimPlant with other malware, and NimPlant with AI and concludes with a discussion.

5.1 Initial NimPlant Configuration

The setup, pre-findings, and network analysis of the initial NimPlant configuration are presented in this section.

5.1.1 Setup

For the client, we used a desktop computer with an Intel Core i5-3550 CPU and 8GB RAM, formatted the hard drive, and installed the newest version of Windows 10, as shown in Figure 5.1. To monitor the client's network traffic, we installed Wireshark[1] on the client. Wireshark is a popular network protocol analyzer. It logs individual packets and allows saving logs of given time frames. We use these to determine whether we can detect NimPlant based on the client's network activities. Since bots usually communicate over a network to receive commands, release information, perform attacks, or redistribute, monitoring network connections and traffic is a potentially effective detection technique.

We used an Ubuntu VM with Ubuntu 23 as the operating system for the server. To be able to use the Web-based GUI of NimPlant, a lightweight desktop GUI called Ubuntu MATE[2] was installed, which runs well on the limited resources in the setting of this experiment with 4GB RAM and with 50GB disk space available. To access the same remote desktop from different devices, TightVNC[3] was used. The necessary modules and dependencies for NimPlant were installed according to the instructions on the GitHub repository[4]. After

[1] https://www.wireshark.org/
[2] https://ubuntu-mate.org
[3] https://www.tightvnc.com
[4] https://github.com/chvancooten/NimPlant

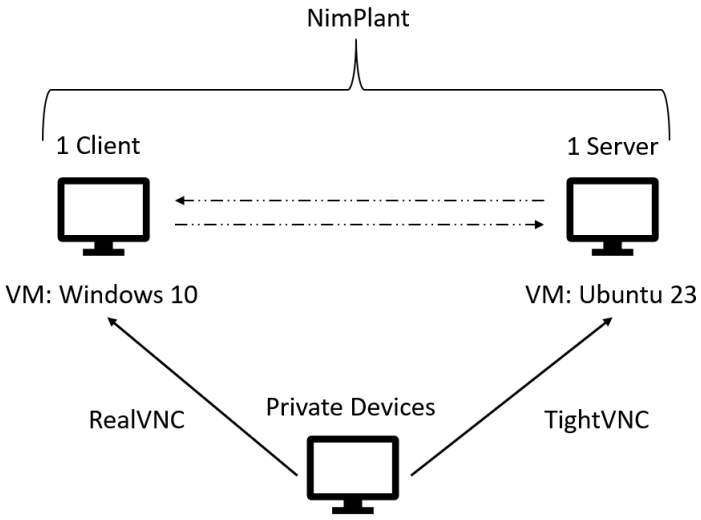


Figure 5.1: Testing Setup

starting the NimPlant server, the web-based GUI can be opened with the preinstalled browser, which was Mozilla Firefox[5].

5.1.2 Pre-Findings

NimPlant, with its initial settings, was tested on the client after completing the setup in stage 1 to determine whether the Windows Defender detects malicious network traffic. Data was collected using Wireshark before and after the infection with NimPlant. It was discovered that the Windows Defender would detect and block the NimPlant binaries. Hence, no execution was possible without explicitly permitting Windows Defender to allow NimPlant to be downloaded and executed. Therefore, a second infection was executed, but in that case, disabling all the features of the Windows Defender except the firewall and network protection. The second infection showed that the firewall and network protection could not detect the malicious NimPlant communication. The NimPlant server could give commands to the client, and the client responded and executed the commands without any interference from the Windows Defender. Thus, NimPlant is initially already able to evade the detection of malicious network traffic by the Windows Defender; the decision was made to use the intrusion detection system Snort and to apply the evasion on Snort.

5.1.3 NimPlant Network Analysis

The underlying network communication of NimPlant follows the same principles as every client-server system, where the communication is based on REST APIs like POST and GET requests. But unlike usual client-server systems, NimPlant uses the REST APIs for malicious activities. Once the victim's machine has been infected and the NimPlant executable is executed, the infected machine sends GET requests to the server, as shown

[5]https://www.mozilla.org

in Figure 5.2. As soon as the server notices the requests from the client, the server creates an ID and sends the ID to the client. Since now the client has the affirmation that the server is online, the client initiates a POST request to complete the registration by sending encrypted data about the victim.

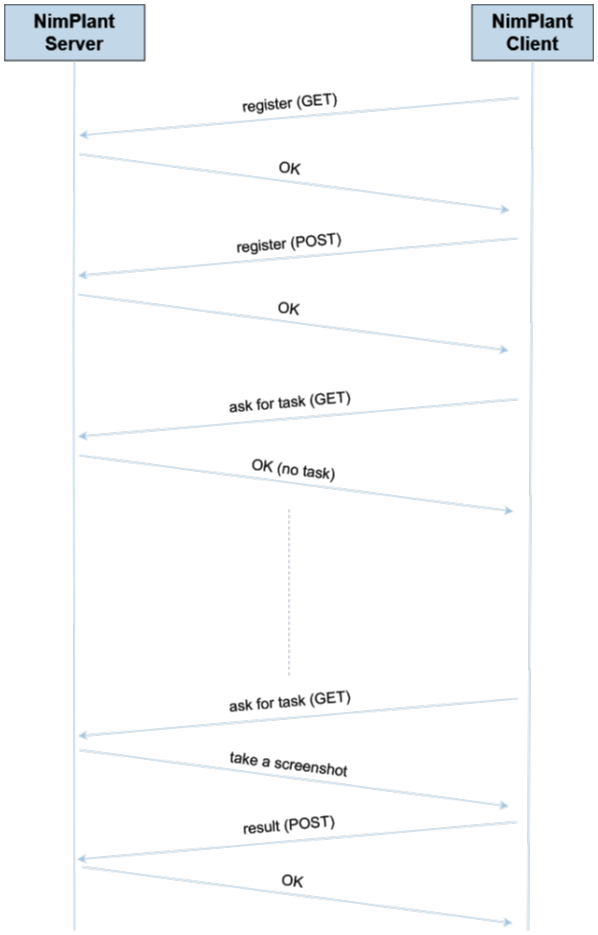


Figure 5.2: NimPlant HTTP Communication Process

After the registration has been completed, the client continuously asks the server for commands using GET requests. As soon as the server sends a command, for example, to take a screenshot, the client executes the command, sends the result with a POST request, and continues asking for further commands.

Analyzing the communication process with Wireshark enabled the display of the HTTP data packets sent between the client and the server. Analyzing the HTTP packets allows for uncovering indicators of compromise. Some of them are marked green in Figure 5.3. The indicators of compromise can be detected on the initial NimPlant configuration without any code adaptions:

- One of the indicators of compromise can be the naming of the client and the server. Naming the client ”User-Agent: NimPlant C2 Client” and the server ”Server: NimPlant C2 Server” openly indicates that a C2 communication is happening.
- A second indicator of compromise could be keywords in the URL like ”register”, ”task”, and ”result”, which may indicate that a malicious bot asks for commands and returns results.
- NimPlant initially uses port 80 to communicate between the client and server. Thus, this port can be seen as a potential indicator of compromise when being aware that NimPlant is using it.
- Revealing the IP address of the host server in each client request header may also indicate a NimPlant-specific indicator of compromise, as it is not usual to reveal the host’s IP address in the header.
- The frequency of the packets from the client asking for commands is constantly 10 seconds. This may indicate a malicious communication as it may indicate a keep-alive mechanism that C2 frameworks use.
- Observing the network traffic also reveals some constant packet sizes in the client and server requests as demonstrated for the client in 4.4. Constant packet sizes that are specific for NimPlant can indicate NimPlant activities.
- Using HTTP instead of HTTPS in the network communication leads to having intrusion prevention systems alerting that the communication is not secure. Thus HTTP can be an indicator of compromise.

A command is given encrypted, and as soon as the client executes it and sends back the results, the results are encrypted, too.

Figure 5.3: Indicators of Compromise in the HTTP Network Communication

5.2 NimPlant with other Malware

Since the Windows Defender can detect the *NimPlant.exe*, we had to deactivate the Windows Defender to be able to run the executable such that the network could be analyzed during the use of NimPlant with additional malware. For the additional malware, we chose to use ransomware. We decided to use ransomware because it is plausible that in a real-world scenario, hackers could combine C2 frameworks like NimPlant as a first infection malware to deploy, then more harmful malware like ransomware to encrypt the data of the victim and then blackmail the victim for money. For our analysis, we chose the ransomware called Crypter [6]. Crypter is an open-source ransomware developed by researchers for experimental and educational purposes. Crypter has the benefit that the key for the decryption is generated on the same device, such that after the encryption happens, the user can use the key provided to them to decrypt the device.

First, we executed the *NimPlant.exe*. As soon as it started running, we executed several commands using the web user interface of NimPlant as shown in Figure 5.4

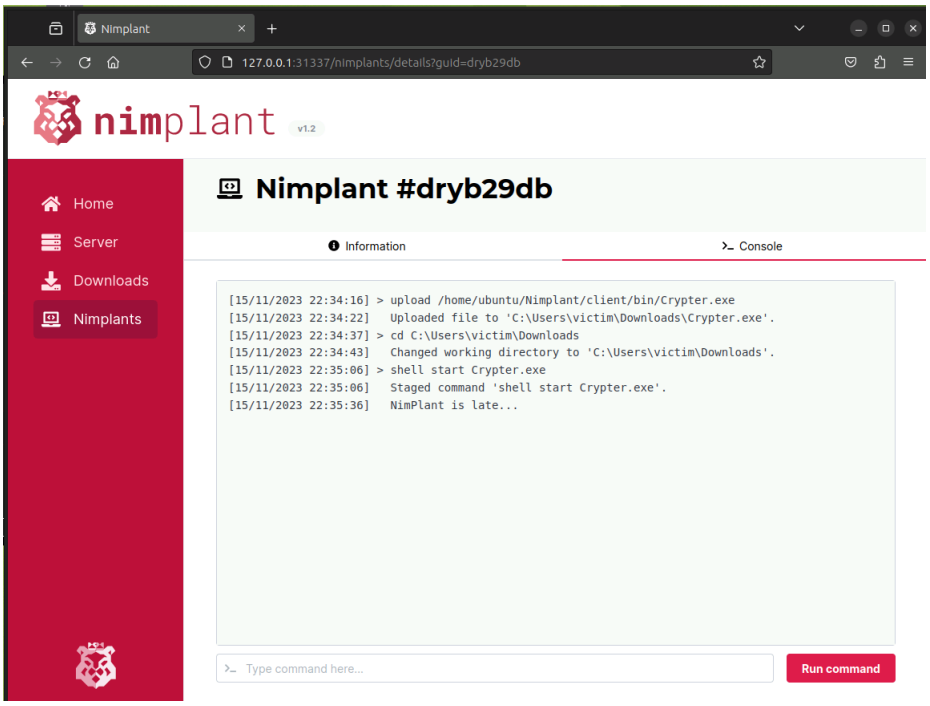


Figure 5.4: Given commands using NimPlant with Crypter

The first command was used to upload Crypter on the victim's machine. Then, the directory was changed to access Crypter, and finally, a shell command was done to execute Crypter. As NimPlant is initially designed to return the location of the upload, the hacker would get the information they need to move to the correct path using the command `cd` and run the Crypter executable with the command `shell start`.

The same process can be analyzed at the network communication level with Wireshark. The upload of Crypter using NimPlant is shown in Figure 5.5, where the requests related

[6] https://github.com/sithis993/Crypter

to the upload command are marked within the red square. It can be spotted that once a command like the upload command is initiated, the constant communication with fixed packet lengths (i.e., client-side 212 bytes and server-side 318 bytes) is interrupted by packets that break this pattern with varying packet lengths.

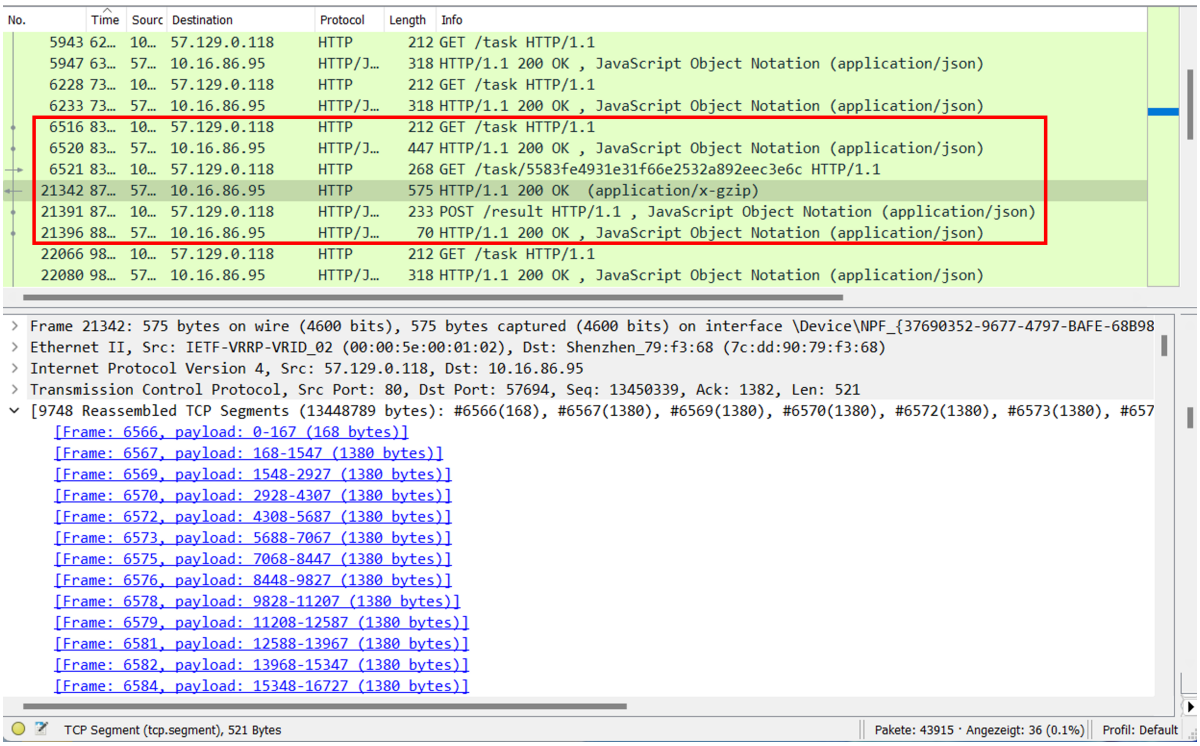
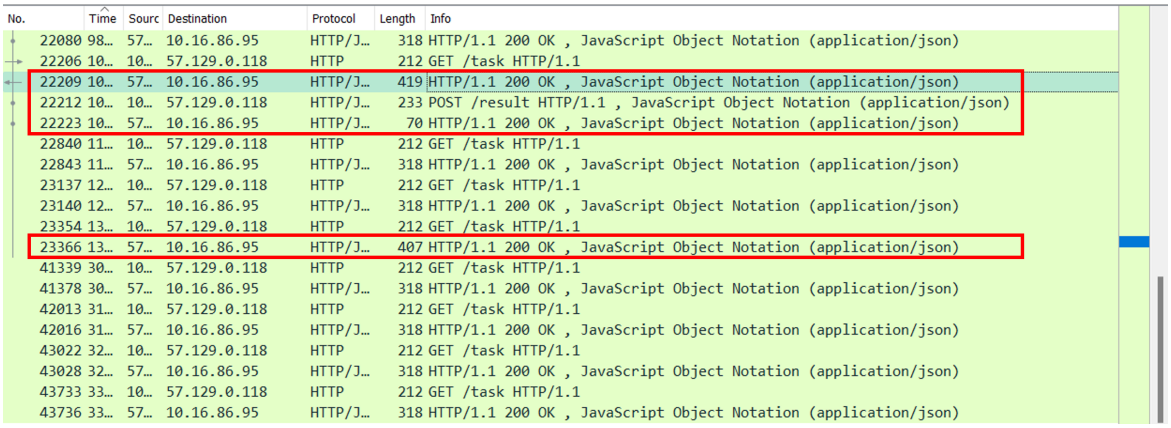


Figure 5.5: Upload of Crypter using NimPlant

It can first be observed that the server informs the client that an upload will take place in the HTTP request of 447 bytes and then confirmed by the client with the HTTP size of 268 bytes. Then the upload starts, and in such a case where a large file like the *Crypter.exe* with a size of around 13.5 million bytes compressed in a gzip has to be uploaded, NimPlant splits the segments of the file into multiple TCP segments. Those are linked with each other and to the main HTTP request. Wireshark captures this segmentation and reconstructs it by linking the TCP segments to the corresponding HTTP request of the size 575 bytes. Clicking on this HTTP request opens the subordinate reassembled TCP segments as shown in the figure. After the file has been uploaded, the client informs the server about the upload location in the request of the size 233 bytes, which confirms this with the HTTP request of the size 70 bytes. Next, the `cd` command with the size of 419 bytes and the `shell start` command with the size of 407 bytes can be observed in the Figure 5.6 which again, like the previous command, disrupt the constant flow of client and server requests of fixed sizes.

Even if, in a real-world scenario, more adaption of the *NimPlant.exe* and *Crypter.exe* binaries would be needed to evade the detection by the Windows Defender (like, for example, using binary obfuscation with polymorphism as described in chapter 2.1.5.2, in a further way to even use AI-based evolutionary packers like described in 2.1.5.6), this demonstration shows that NimPlant has powerful functionalities to execute attacks



No.	Time	Sourc	Destination	Protocol	Length	Info
22080	98…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
22206	10…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
22209	10…	57…	10.16.86.95	HTTP/J…	419	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
22212	10…	10…	57.129.0.118	HTTP/J…	233	POST /result HTTP/1.1 , JavaScript Object Notation (application/json)
22223	10…	57…	10.16.86.95	HTTP/J…	70	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
22840	11…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
22843	11…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
23137	12…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
23140	12…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
23354	13…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
23366	13…	57…	10.16.86.95	HTTP/J…	407	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
41339	30…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
41378	30…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
42013	31…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
42016	31…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
43022	32…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
43028	32…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
43733	33…	10…	57.129.0.118	HTTP	212	GET /task HTTP/1.1
43736	33…	57…	10.16.86.95	HTTP/J…	318	HTTP/1.1 200 OK , JavaScript Object Notation (application/json)

Figure 5.6: HTTP communication of the ”cd” and ”shell start” commands

which could be potentially used by hackers remotely. In such a case, the hackers would use ransomware that sends the decryption key back to their servers and then start blackmailing the victim.

5.3 NimPlant with AI

This section will compare the results gathered from the RL agent's training. This will provide insights into the agent's performance under varying conditions and rule sets, mainly focusing on its effectiveness under naive rules on the target system compared to its counterpart trained with advanced, expert-driven rules. This comparative analysis aims to unveil the divergences in training outcomes under distinct rule sets and reveal the effectiveness of the implemented strategies in enhancing the agent's proficiency during the training phase.

In our comprehensive results analysis, we organize our presentation of results into three distinct components, each shedding light on various facets of our RL models' performance:

1. **Training Time**: Unveiling the temporal dynamics of our models' training, we scrutinize the time investment required for proficiency under diverse rule sets. This part provides insights into the duration disparity between models subjected to naive and expert-driven rules, offering a nuanced understanding of the training phase.

2. **Actions and Rewards**: Delving into the behavioral aspects of our models, we explore how the number of actions and rewards evolves across episodes for both models. This segment aims to clarify the agents' learning trajectories, showcasing the improvement patterns in their decision-making processes and the corresponding rewards obtained during training.

3. **Strategies**: Embarking on a detailed examination of the learned strategies, this part employs comprehensive heatmap comparisons to reveal the significance and distribution of strategies employed by both models throughout training. By unraveling the intricacies of strategy utilization, we aim to highlight the strategic adaptations that contribute to the models' evolving competence in evading Snort rules.

5.3.1 Q-Learning Results - Training Time

As illustrated in Figure 5.7, a notable divergence in training times emerges. The model subjected to naive rules exhibits a relatively rapid convergence, completing 100 training episodes in just over an hour. In contrast, the model trained against expert rules requires approximately 6 hours to accomplish the same training. This discrepancy can be directly attributed to the termination conditions and the reward function. The more intricate evasion rules in the expert scenario demand extended training periods for the model to navigate the augmented complexity, emphasizing the relationship between the rule set and the time required for model proficiency.

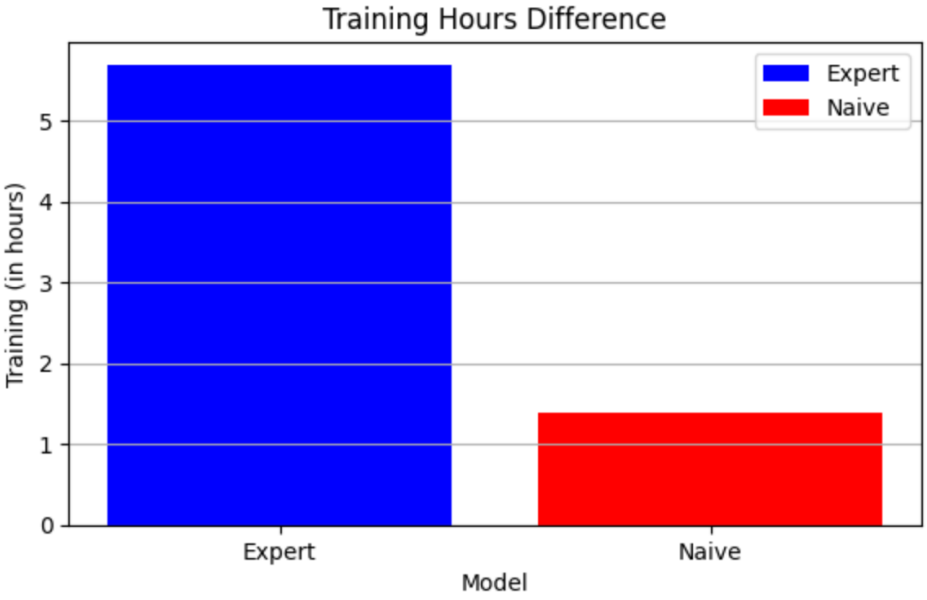


Figure 5.7: Models Training Time

Figure 5.8 provides a nuanced temporal perspective, analyzing the training time per 20 episodes, grouped into five intervals (Episode Groups). The expert model exhibits a compelling trend, demonstrating a reduction in training time as it gains insights into the environment.

As the model accumulates knowledge, it strategically exploits its learned strategies to enhance evasion, gradually reducing alerts. In contrast, the naive model displays slightly higher initial training times but achieves stability in the later episode groups. This stability arises from the model's discovery of an optimal set of strategies that yield minimal alerts, resulting in higher rewards. Consequently, the naive model's overall training time diminishes, reflecting the efficacy of the learned strategies in mitigating alerts and optimizing performance.

These findings emphasize the relationships between rule complexity, training time, and the adaptive learning process of reinforcement learning agents in evading detection within the context of Snort rules. The expert model's extended training duration highlights the necessity of comprehensive training to tackle intricate rule sets. At the same time,

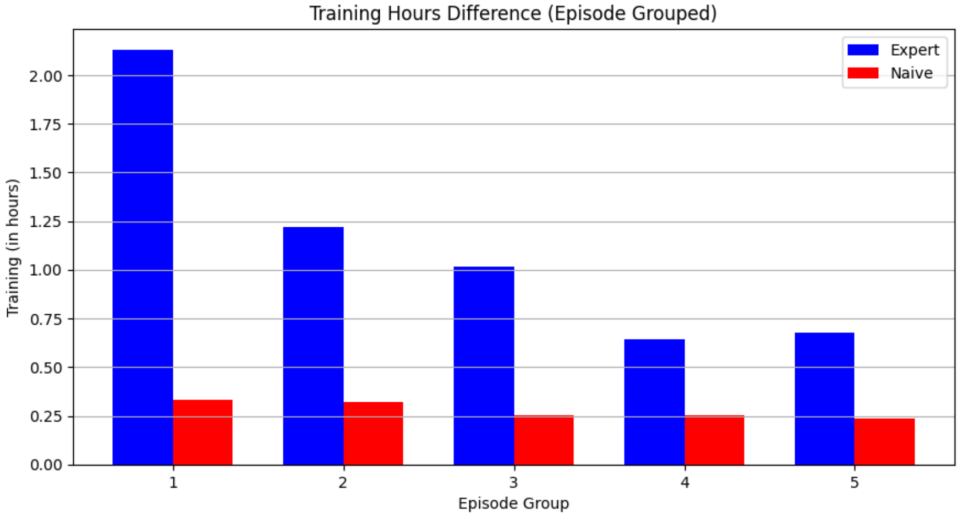


Figure 5.8: Models Training Time - Grouped Episodes

the temporal dynamics depicted in Figure 5.8 emphasize the iterative refinement and exploitation of learned strategies in the pursuit of efficient evasion strategies.

5.3.2 Q-Learning Results - Actions and Rewards

In Figure 5.9, we present a detailed examination of the number of actions the RL agent takes per episode, contrasting its behavior under naive and expert Snort rule scenarios. Initially, due to the stochastic nature of Q-Learning, both models exhibit an increased number of actions. As the episodes progress, a discernible decrease is observed, signifying the models' learning and adaptive capabilities. Notably, the model exposed to the expert rules displays fluctuations, which can be attributed to the nuanced exploration and exploitation phases, reflecting the intricate nature of actions taken based on the environment setup, as expounded in the implementation chapter. Conversely, the model exposed to naive rules stabilizes quickly, indicating the efficiency of a limited set of strategies in achieving low alerts and high rewards.

Figure 5.10 extends our analysis to the rewards obtained per episode, mirroring the fluctuation pattern observed in the actions figure. The expert model, in particular, showcases notable negative spikes beyond the initial episodes, elucidating the impact of randomly enabling/disabling strategies. However, from episode 60 onward, a distinct reward increase becomes evident as the model progressively exploits its acquired knowledge. It is crucial to note that the "Done" condition may be achieved despite an overall negative reward, emphasizing the importance of reaching low alert counts during the iterative strategy enabling/disabling process. Additionally, our implementation incorporates a reward counter, currently set to trigger the "Done" condition after five positive rewards or a single positive reward exceeding 4, contributing to the iterative progression through episodes. Incorporating a reward counter in our implementation introduces a strategic element, ensuring the agent progresses through episodes based on predefined positive reward

thresholds. This counter adds an extra layer of control and adaptability to the learning process.

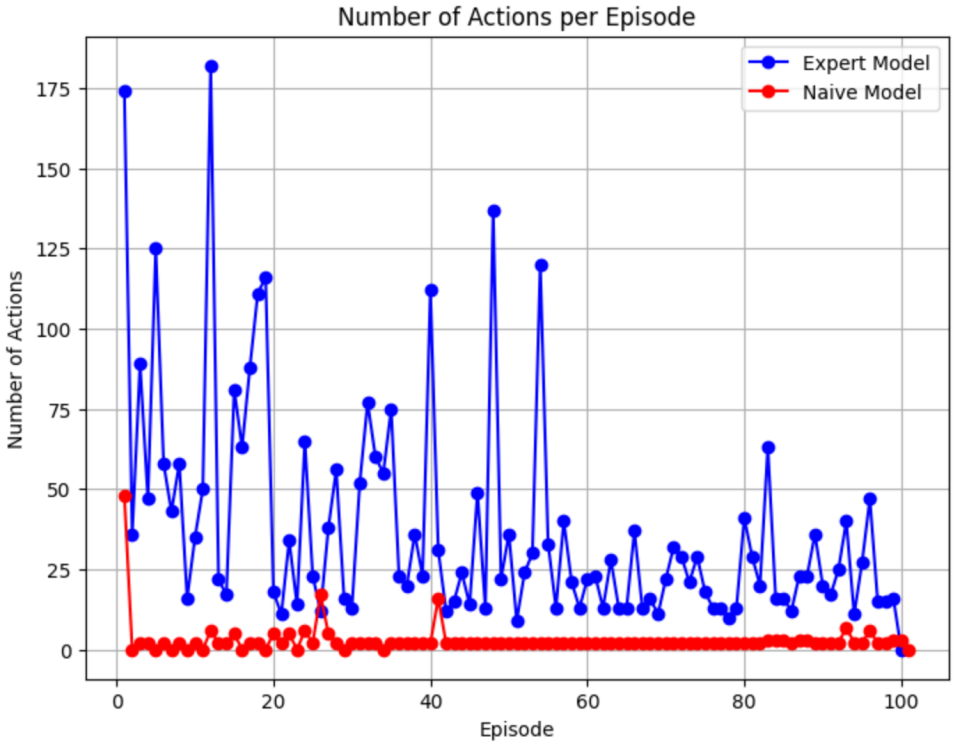


Figure 5.9: Number of Actions per Episode

Figure 5.10: Amount of Reward per Episode

These insights detail our understanding of the models' actions and rewards and lay a robust foundation for interpreting their evolving decision-making dynamics, which is crucial in cybersecurity detection evasion.

5.3.3 Q-Learning Results - Strategies

Figure 5.11 presents a heatmap illustrating strategy counts for the model operating against expert Snort rules across 100 episodes. Initially, the heatmap reveals dense, dark areas, indicating heightened strategy enablement in the early training phases. This suggests the agent's exploration as it assesses optimal strategies.

Figure 5.12 replicates this heatmap for the model under naive rules. In contrast to expert rules, fewer dense dark areas indicate that the model sufficiently explores the set of optimal strategies without requiring extensive exploration.

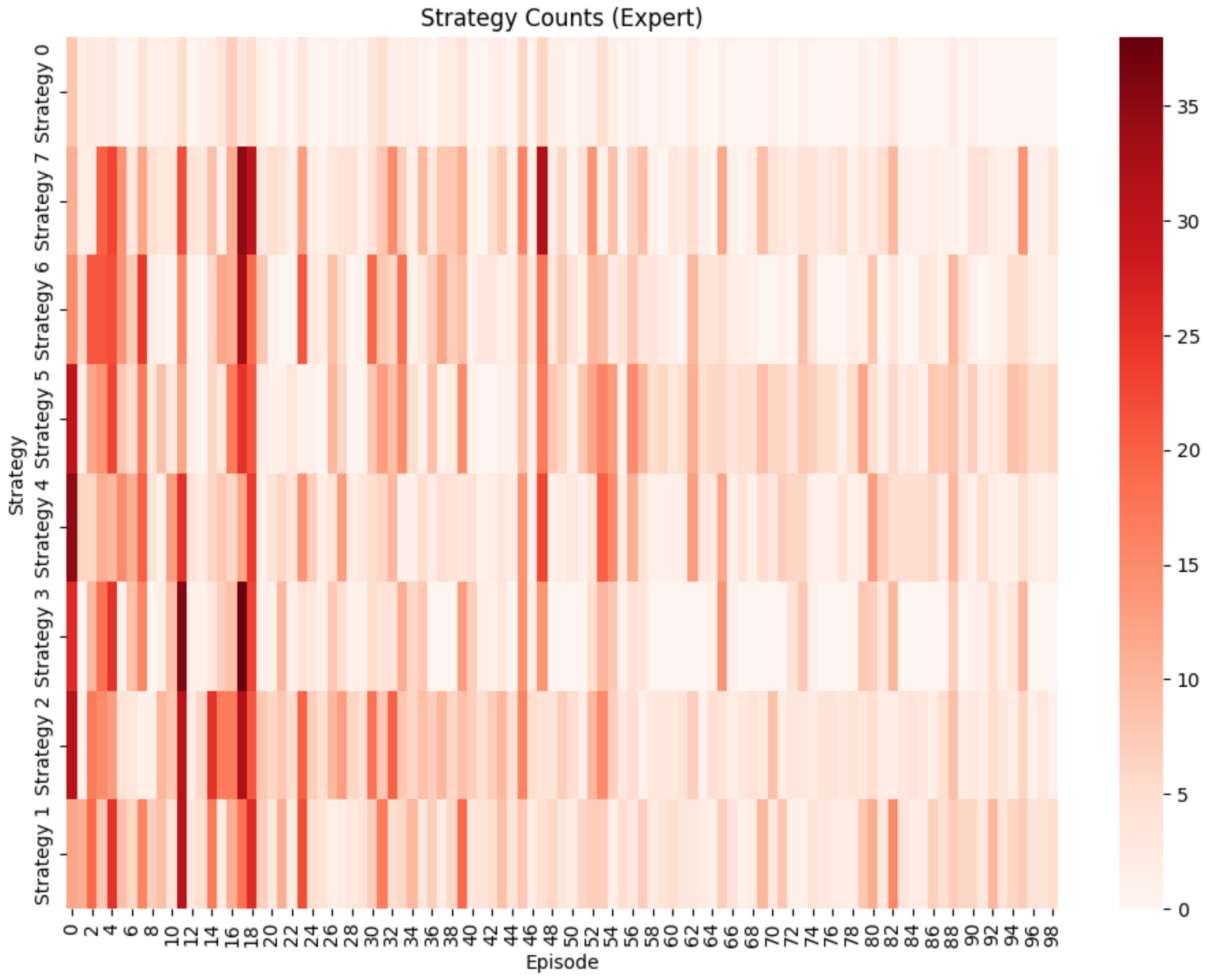


Figure 5.11: Strategy Counts per Episode (Expert)

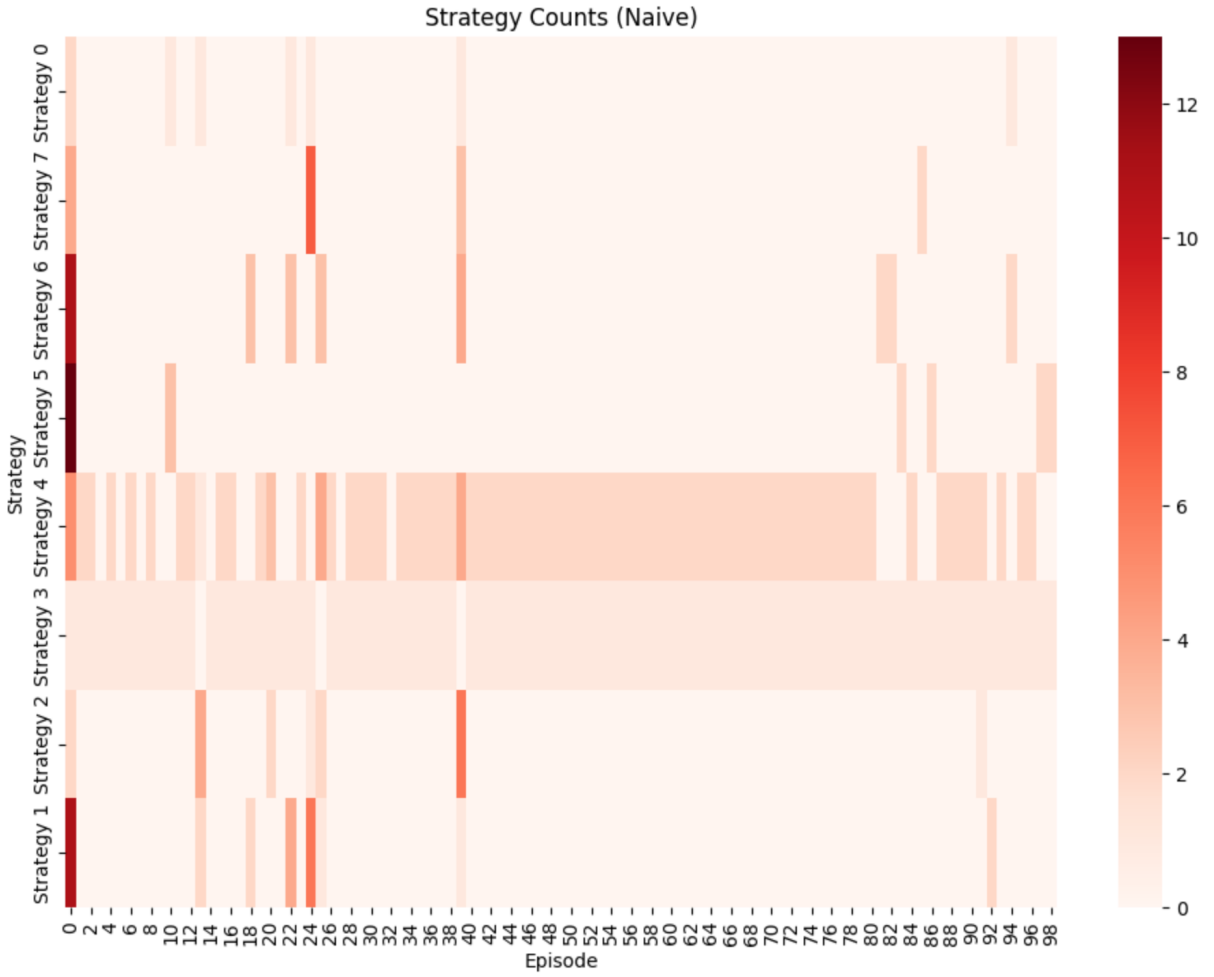


Figure 5.12: Strategy Counts per Episode (Naive)

Figures 5.13 and 5.14 introduce a crucial analytical step by presenting normalized and grouped counts of enabled strategies, contrasting with the absolute counts depicted in the prior heatmaps. This deliberate approach significantly enhances the interpretability and comparative analysis of our RL model's evolving strategies.

Normalization is employed to scale the strategy counts relative to the total actions taken within each episode group. This normalization eliminates potential biases arising from variations in the total number of actions across different episode groups. By doing so, we obtain a proportionate representation of strategy prevalence, enabling a more accurate evaluation of their relative importance.

Grouping the counts over 20 episodes offers a condensed overview, smoothing out potential episode-specific anomalies and highlighting overarching trends. This strategic grouping aids in identifying consistent patterns and discerning the emergence of effective strategies throughout training. It provides a more precise narrative of the model's strategic evolution, especially in distinguishing recurrent and impactful strategies from sporadic or context-specific occurrences.

Figure 5.13 introduces normalized values for strategy counts grouped over 20 episodes, offering a nuanced perspective. The dark areas, excluding strategy 0, imply substantial counts across various strategies, emphasizing the absence of a dominant strategy to evade expert rules. Strategy 1 and 2 (related to the NimPlant keyword) and Strategy 5 (host header changing) exhibit consistent dark areas, underlining their significance.

Figure 5.14 extends this analysis to the model under naive rules, showcasing two prominent strategies (3 and 4) across episode groups 3 and 4. Strategy 3, involving port hopping, and Strategy 4, changing communication endpoints, emerge as effective tactics. Strategy 3 aligns with our assumption that network security analysts might block traffic on known default ports, making port hopping an effective evasion tactic.

In both scenarios, the do-nothing strategy sees minimal triggering, signifying the model's recognition that enabling this strategy does not lead to a decrease in alerts.

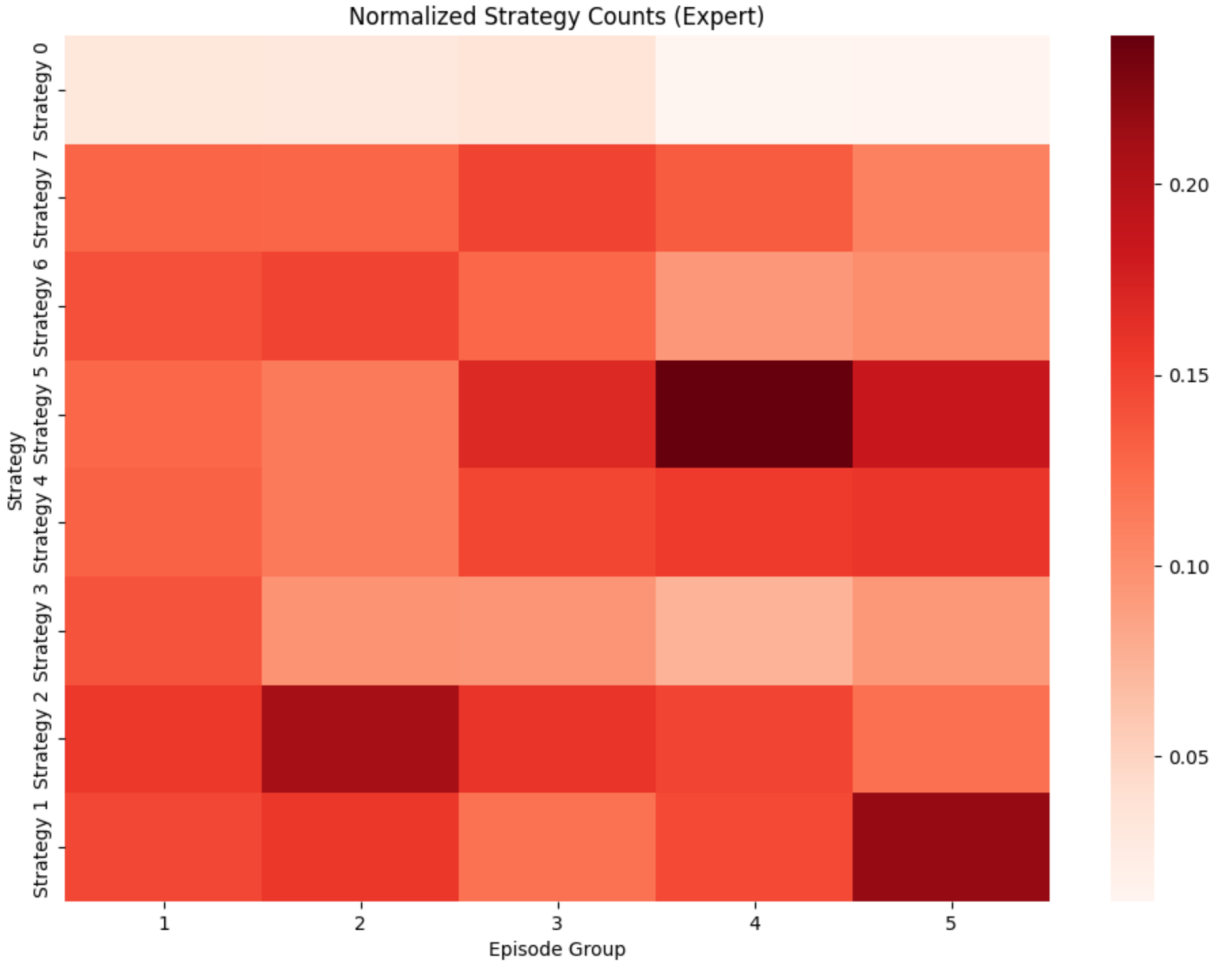


Figure 5.13: Normalized and Grouped Strategy Counts (Expert)

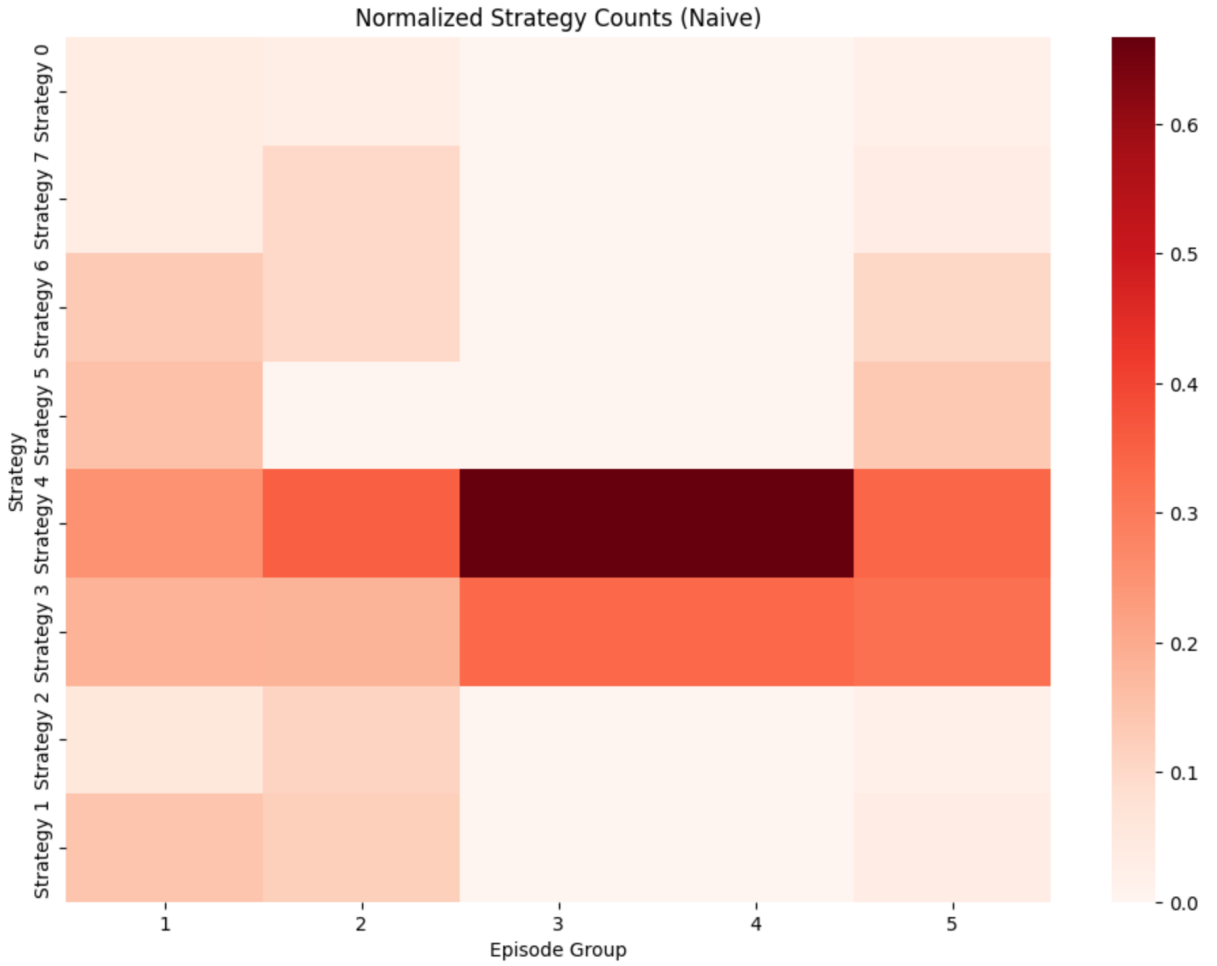


Figure 5.14: Normalized and Grouped Strategy Counts (Naive)

5.4 Discussion

The discussion covers topics like reaction time, scalability, automation level and ends with recommendations for the defense against C2 malware like NimPlant.

5.4.1 Reaction time

One important aspect of C2 malware and defense against C2 malware is the reaction time, which we consider the time that the client of the C2 malware needs to adapt strategies enabled or disabled by the server. This is a relevant concept in cases where C2 malware like NimPlant is used because having a low reaction time enables C2 malware to be more flexible regarding its dynamic behavior, as the server can quickly communicate behavior changes. The clients adapt themselves according to the server's commands (as some strategies need to be communicated to the clients to function, for example, the endpoint changing). For NimPlant, the reaction time is based on the frequency on which the

client sends the `getTask()` requests to the server that can be modified in NimPlant with the variable `sleepTime`. To positively influence the flexibility of NimPlant, one could, therefore, argue that it would be beneficial to maximize the frequency as much as possible (by setting a sleep time as low as possible). However, having a very high frequency comes with downsides, as very high frequencies can be marked by malware defense systems as malicious, such as being marked as DDoS attacks. The challenge from the attacker's perspective lies in achieving sufficient flexibility without creating suspicious behaviors.

Another important aspect in this context is that the benefits of a faster reaction time can only be leveraged if the attacker can decide which strategies to enable or disable and give the commands for enabling or disabling strategies at the same speed as the reaction time (or faster). Consider the following scenario as an example to demonstrate the thought. Let us say we have NimPlant running, and the `getTask()` is arriving all 5 seconds to the server, and the attacker needs 10 seconds to decide which command for enabling or disabling strategies should be given. This means that the adaptation of NimPlant's dynamic behavior always takes 10 seconds as the input from the attacker is needed to change the dynamic behavior, and the benefit of the relatively fast reaction time of 5 seconds is not leveraged. Conversely, if the attacker needs, for example, 4 seconds to decide and give the commands for enabling and disabling strategies, the reaction time is 5 seconds. After 5 seconds, the dynamic behavior of the C2 malware will change. Those examples demonstrate that if the pre-step (deciding which strategies to enable or disable) takes longer than the after-step (communicating the enabling or disabling of the strategies to the clients), then the benefits of the after-step are reduced.

5.4.2 Scalability

If the thought from the previous sub-chapter is expanded further, then one could argue that the more variations of strategies and potential dynamic behaviors the attacker as an option has, the higher the probability that the attacker would need more time to decide which strategies to enable and which to disable. Therefore, scaling the number of strategies and dynamic behaviors for a real-life scenario could lead to a point where human-driven attacks may face limits, as the speed of a human would not be sufficient to leverage the benefits of a high reaction time. Thus, one could argue that automating this decision process of choosing strategies by having an algorithm managing the behavior of the C2 malware at a higher speed than a human could solve this problem. Incorporating AI in the algorithm could increase its cleverness and make faster decisions due to the automation and make smarter ones than humans, as humans tend to make wrong decisions under time pressure.

Since our testing setup only considered one client, scaling the scenario to multiple clients is also worth considering. An attacker may not only infect one device of a company but try to increase its chance of success by infecting multiple devices and hoping that one of them will have a vulnerability to exploit. In such a case, the options for potential strategies would increase further as one could also vary between strategies that have the same effect on all the clients and strategies that only affect specific clients to avoid a synchronized behavior of all clients that could indicate C2 traffic. However, having multiple clients can

also increase the risk for the attacker that the detection systems may observe multiple clients showing similar behaviors (for example, the keep-alive mechanism), which may lead to having those clients marked as part of a C2 botnet.

5.4.3 Automation level

At this stage, it should be evident that the management of a C2 server can demand a lot of actions from the botmaster during any potential training and a real application. As highlighted in the previous paragraphs, this is potentially extended by the additional dimension of timing and reaction. Depending on the use case, any such system piloted by a human may suffer serious performance restrictions. For this reason, automating certain aspects can lead to significantly better results. One such aspect has already been discussed in Section 5.4.2: automated strategy switching. Switching strategies may be necessary during training sessions to develop effective evasion strategies, or in a real scenario as a reaction to a change in the environment or preemptively as part of an evasion strategy. In the former two cases, reaction time is crucial to prevent a bot from being identified. In a training scenario, this also drives the amount of training that can be done in a given time frame. Manual strategy switching is, therefore, simply unreasonable, even more so in a system that may be online the whole day. We have presented a method of automated strategy switching in this project, but there are other ways that one might choose to use for the task. While our method was to learn from all experiences during the system's lifetime, another system might prefer to apply strategies based on the most recent environmental changes.

One aspect that could be expanded is using commands through the implants' command line interface. The necessity of automating this part depends on the context and intended use of the system. While the keep-alive mechanism already warrants an automated strategy switching, the number of commands being sent can vary from a small handful to a single bot to hundreds or thousands to a whole botnet. In the latter case, one might want to automate the execution of some commands. Again, if the submission of commands is bound to certain events, automating these commands might also be practical. Given that NimPlant commands are entered in a command line interface, writing a script that automates certain commands should be perfectly feasible. In this project, using commands was not part of the training process either. Still, in a testing environment with commands, automating the submission of these would be reasonable for the same reasons already discussed.

In theory, the RL model could also be amplified by additional features. The RL program in this project ran for up to six hours uninterrupted. For an AI training model, this is still a very manageable period. If a botmaster were to develop a larger set of strategies, choose a different AI model, train with longer intervals between strategy switches, or train the model on a system with more servers or bots, the time required to train the model may increase drastically. In this case, it may be favorable to give the model options to adjust the training process on runtime based on already gathered information. Last but not least, if the AI model is not only used in a training environment to develop evasion strategies but is instead implemented as an operational application, it could potentially

be trained to improve additional factors, and further training automation would probably be needed.

5.4.4 Recommendations for the defense

This subsection presents several recommendations that result from experiences and knowledge gained throughout the project.

5.4.4.1 Watch out for Indicators of Compromise

On one side, there are indicators of compromise that are specific for individual malware like static characteristics that are, for example, keywords referencing a specific malware or dynamic characteristics that expose themselves during the runtime of the malware like NimPlant had the constant size of the client and server requests. Therefore, when designing detection systems, one should set the basis for the rules by considering static and dynamic indicators of compromise. Attackers can still change those characteristics of the malware, but focusing on them still provides a good starting point for establishing rules as there may exist the chance that an attacker did not successfully cover all indicators such that the defender may be able to detect the malware. The challenge is, therefore, to be more familiar with malware than the attackers are to set rules that may trigger indicators that the attackers didn't think about. On the other hand, there are also indicators of compromise that are not specific for individual malware but are related to a category of malware, like the keep-alive mechanism that is not just NimPlant-specific but a general indicator of compromise for C2 frameworks. Such indicators should also be considered when defining rules as they may increase the general protection against malware.

5.4.4.2 Think like an Attacker

Developing strategies to evade Snort's detection made us think like an attackers. As an attacker, we preferred rules for which less effort was required to develop strategies over those for which more effort had to be invested for evasion. For example, evading a signature-based detection which alerts on keywords like "NimPlant" was developed in a shorter timeframe as keywords could just be exchanged in comparison to changing the content sizes of the packets or changing ports on runtime, which would influence the dynamic behavior of NimPlant. This could imply for the defense that detecting malware like NimPlant based on signatures could be less effective than detecting malware based on dynamic behavior. The reason is that signatures like keywords may need less effort for the adaption by attackers to enable evasion than changing the dynamic behavior of malware. Encrypting, for example, network communication by using HTTPS instead of HTTP would change the malware's signature entirely and make the rules based on keywords useless. Therefore, it is important for detection systems to consider not only rules that analyze the static characteristics of malware but also rules that alert based on dynamic behaviors such as request frequency or content size of requests.

5.4.4.3 Think one step ahead of the Attacker

Thinking as a defender about potential evasion strategies that may not exist yet allows the defender to prepare for future scenarios. Consider how current and future technologies can impact the evasion of malware detection, for example, how GANs were used for evasion, as discussed in the related work, and how the defense can prepare itself against such attacks. In this context, it is essential to stay up-to-date with technology development and watch out for threats and opportunities from evolving technologies such as AI. We found in our literature studies in 2.1.5.6 that using AI to improve malware's capabilities is a relatively recent development. Nevertheless, as AI is continuously evolving, the opportunities for attackers to use it increase, so defenders should increasingly think about how to deal with AI-powered malware attacks.

5.4.4.4 Keep your rules secretly

The defense mechanisms become worthless if attackers find a way to get insights into the rules used. This would allow the attackers to develop specific strategies tailored for the evasion of those rules. The attackers could try to discover the rules through social engineering or by having an insider who could access the companies' intrusion detection rules. To reduce this risk, organizational measures could be taken by the company, like limiting the number of people having access to the company's rules to as few as possible and as much as necessary. People who have gained the trust of a company should be placed in positions to access the rules and the access should be continuously monitored and logged such that if a breach happens, one could first check if suspicious activities might have been discovered in the logs. The monitoring and logging may also have a preventive effect and scare off people with malicious intentions.

5.4.4.5 Use short time spans for threshold rules

Using threshold rules in intrusion detection systems like Snort can contribute to detecting the dynamic behavior of C2 malware. For example, if it is observable that C2 malware contains specific content sizes or specific frequencies during its execution, threshold rules could be defined to search for those specific measures and to alert based on them. But the threshold rules also come with the challenge of how to set the period. Too short periods can increase the probability of false positives as the intrusion detection system would alert traffic that may not be malicious. Otherwise, too long periods could give the attacker sufficient time to react and adapt strategies, as the intrusion detection system would only be able to react after the period ended. This could lead to missing out on the detection of malicious network behavior. So, even if short periods may increase the number of false positives, one could argue that the potential negative consequences of false positives are less problematic than missing out on the detection, which could enable the attacker to perform malicious behavior. Therefore, having better too-short than too-long time spans for the threshold rules should be more secure.

5.4.4.6 School your employees

The reinforcement learning demonstrated in a proof of concept that it was far easier to train an algorithm against naive rules compared to the expert rules. The time in which the optimal order of strategies was achieved was significantly smaller for the naive one compared to the expert. This demonstrates that the harder the rules are, the more time is needed to achieve full evasion. An implication of this can mean that having experts creating rules makes it harder for attackers to create evasion strategies than when having less knowledgeable personnel doing it. Therefore, a company can improve its chances of having more expert rules by investing in schooling to improve the knowledge of its employees. This could lead to more knowledgeable employees, increasing the challenges for the attackers. Since cyber security is constantly evolving, schooling should also happen regularly, for example, yearly schooling that keeps the employees up-to-date.

Chapter 6

Final Considerations

The final considerations chapter concludes the report with a summary of the work done, the conclusions derived from the work, the difficulties encountered, and suggestions for future work.

6.1 Summary

In the foundational exploration of C2 tools, detection and evasion strategies, and related work (cf. Chapter 2), our project investigated the NimPlant framework, an open-source C2 tool written in Nim for clients (victims' bots) and Python for the server. Grounded in a prototypical testing environment, we opted for a Windows 10 machine as a victim simulation, aligning with the OS's prevalence and the NimPlant client's Windows exclusivity. This strategic selection based on our assumptions is substantiated in Section 3.1.

A major pillar in our methodology was the systematic detection using Snort, detailed in Section 3.2, outlining rules and alerts derived from NimPlant's client-server communication. To enhance detection robustness (cf. Section 3.2), we introduced two defensive models - one leveraging naive rules for users with moderate security knowledge and the other incorporating expert-based rules for experienced security analysts.

In our practical experiments, as discussed in Section 5.1, we simulated infecting the Windows 10 device and analyzed the traffic using tools like Wireshark scrutinizing packets for indicators of compromise. Analyzing indicators from the traffic provided insights for detecting C2 activity, forming the basis for constructing the two detection models mentioned above.

Moving on to our interventions (adoptions of the NimPlant framework), we implemented seven command-based evasion strategies, elaborated in Section 4.1 and clarified in Section 3.3. These strategies, activated through the CLI upon a specific connection from the bot, ranged from tweaking the user agent to adjusting the frequency and size of packets between the client and the server. The strategies required user intervention for proactive enabling, hence the labeling *"command-based."*

Aligned with our project's goal to explore the potential of using offensive AI for enhanced evasion, we investigated offensive AI applications. Guided by our literature review and considering our assumptions and timing constraints, we explored various AI models (such as RL, GAN, DNN) to determine their applicability to our scenario. Given these considerations, we decided on RL. Unlike proactive enabling, RL involves the AI model identifying optimal strategies and enabling them dynamically based on the environment (*e.g.*, number of triggered alerts by Snort). Specifically, our implementation of the RL agent to enable evasion strategies, which is trained against Snort, is detailed in Section 4.2. We integrated Q-Learning from the Gymnasium library and trained two RL models - one with naive rules and the other with expert rules - over 100 episodes.

The results of this training, presented in Section 5.3, highlight the RL agent's successful identification of the optimal combination of enabled strategies. This led to the lowest number of alerts within a relatively short time, approximately 1 hour, for the case of the naive rules. In contrast, the RL agent exposed to the expert rules took 6 hours for the same number of epochs. Importantly, both agents ultimately identified the best strategy combinations independently of the training time.

An ultimate and precise comparison of our obtained results with the state-of-the-art is constrained due to the following factors: first, the utilized tools: In the literature (cf. Chapter 2), the papers we found dealt with evading and attacking ML-based detectors or other detection tools, often using black-box testing (*i.e.*, the trained AI model, *e.g.*, a GAN or RL agent operated with a binary output from the defensive model, either the sample is malicious or not). However, in our case, we used Snort, a non-ML-based IDS, and the trained RL agent's goal was to reduce the number of triggered alerts generated by a set of pre-defined rules. Second, the simplicity and broad assumptions of our approach. In our scenarios, we assumed that the NimPlant infection binaries are already installed and running on the victim's system. Our pre-tests found that AVs like Windows Defender prevent the execution of the NimPlant binaries on the system, flagging them as malicious. Also, the number of trained epochs is relatively low; nevertheless, it was sufficient for the two RL agents to find the optimal set of strategies and converge. Again, this can be attributed to the moderate number of strategies and their complexity in reducing Snort alerts. Last, and directly related to the previous reason, our project, driven by the goal of providing recommendations for the defense, mainly highlights AI's offensive potential. It raises awareness of AI's dual capacity (defensive and offensive) by showcasing the feasibility and extensibility of the trained RL agents, prioritizing this demonstration over building a sophisticated AI model. Overall, the obtained results comply with their trend with the state-of-the-art regarding AI's ability to optimize evasion strategies.

Moreover, in Subsection 5.4.4, our project included broad defense recommendations, emphasizing vigilance for indicators of compromise, dynamic behavior analysis, awareness of emerging threats, rule confidentiality, and employee training.

Additionally, our investigation assessed NimPlant's compatibility with ransomware, revealing implications for network traffic, as discussed in Section 5.2.

This project, rooted in a review of C2 tools, detection, and evasion strategies, and relevant literature, allowed for our subsequent analysis, experimentation, and recommendations for the defense. It offers a holistic view of the ever-evolving cybersecurity threat landscape and

recent offensive techniques, combining foundational insights with a thorough examination of NimPlant's C2 aspects. Moreover, the integration of offensive AI, by our utilization of RL, enhances the sophistication of our approach and contributes to a more subtle understanding of AI's offensive potential.

6.2 Conclusions

In the conclusions of our project, the combination of the NimPlant C2 framework with RL, which refined dynamic strategies for evading Snort alerts, has shown various different insights in the intersection of the fields of cybersecurity and AI. As we wrap up our findings, it becomes evident that navigating these two scientific fields requires a careful approach, which covers the complexities of infrastructure setup, detection mechanisms, and the dynamic interplay between offensive and defensive strategies, along with the potential of AI.

Our exploration shows the importance of establishing a robust infrastructure for activities related to malware testing and malicious operations. Beyond the basic execution of simulated attacks, there is a critical need to emphasize creating a secure and controlled environment which complies with ethical experimentation. This approach is key for evaluating the potential of offensive AI and detecting and isolating C2 network traffic, which is crucial for effectively developing evasive strategies against derived detection models.

Furthermore, our analysis highlights the significance of comprehensive detection of C2 frameworks. Utilizing dedicated tools like Snort and Wireshark, we monitor and inspect the communication between the client and server architecture, which is common in C2 master-bot communication. From rule-based detection models to the thorough examination of packet details (*e.g.*, protocols, ports, addresses, and other packet-specific elements), our project underscores the importance of an observant and adaptive detection framework. This fortifies the defensive posture and provides critical insights for understanding offensive malicious activities and constructing more robust threat models.

Highlighting the offensive capabilities of AI, our project demonstrated its adaptability across diverse detection models, showcasing its potential in optimizing evasion strategies. Noteworthy was the RL agent's ability to determine optimal evasion tactics, significantly reducing the number of Snort alerts. While it is acknowledged that augmenting the model with additional features, such as OS type and extending the various detection mechanisms (specifically related to C2 infection executables and master-bot communication), may extend training times, the demonstrated success implies the feasibility of integrating further elements into the AI model. This highlights the scalability and potential of our approach, opening possibilities for further exploration and refinement in subsequent research projects.

Our project incorporates vital defense recommendations, emphasizing their value in enhancing cybersecurity resilience. Issuing and following these recommendations is crucial for organizations as they provide proactive measures to detect and mitigate threats, fostering adaptability in the face of evolving cyber risks. By implementing these guidelines,

organizations can improve their defenses, stay ahead of potential attackers, and maintain a robust security posture.

Finally, recognizing the important role of AI in cybersecurity goes beyond refining defensive tactics, extending to a crucial awareness of its offensive potential, *e.g.* optimizing evasive strategies. While AI is commonly applied in defensive measures, such as ML-based anomaly detection, it is essential to remain aware of its dual capacity to enhance evasion techniques and attack other AI-based defensive models.

Reflecting upon the outcomes of this project, these insights lay a solid foundation for future advancements in the cybersecurity landscape, contributing to a subtle understanding of the complex interplay between cyber threats and the defensive measures necessary to mitigate them while keeping the offensive potential of AI in mind.

6.3 Challenges

To ensure that each group member had access to the testing environment, we set up a remote connection to the client and server in the testing environment from our private devices. A reoccurring problem that we experienced in this context was that the connection to the client was not consistently stable, which led to repeating disconnections. Once the client had disconnected, we had to contact someone from the university to reestablish the connection. To make sure that those disconnections would not interrupt or cause disturbance in the training process of the AI model, we implemented a fail-safe. This way, in the worst case, if such a disconnection disrupts the training, we could restart and continue the training from where it stopped.

Some other difficulties stemmed from the IDS we chose, Snort. This was our primary monitoring tool for the client's network traffic and served as a simulated adversary for the improved evasion strategies we had either developed manually or that were generated by the RL model. As we developed more offensive and defensive strategies, it became clear that Snort has limited monitoring capabilities, partially due to the client using a Windows operating system. The strongest restriction while writing rules for Snort's packet analysis was that individual TCP segments checked by Snort could only count towards one threshold rule at a time, effectively prohibiting the use of correlating rules. This was mainly an issue for the rules that were monitoring the frequency or size of incoming packages. We were able to compensate for some of the limitations by adjusting some designs, but the final version was weaker than we had hoped for nonetheless.

6.4 Future Work

Given the challenges and limitations that we encountered, we thought about some possible investigations that could be conducted to lower or overcome limitations and could provide more valuable insights related to the topic of this project:

- Future work could focus on testing another IDS like for example Suricata[1]. Suricata may overcome the limitations that we had with Snort and could therefore contribute to the training of a more complex AI model.

- This project only focused on analysing the network traffic of an already infected device (see sub-chapter 3.1). This excluded two major aspects of a complete security system: The initial infection and the monitoring of the infected device on its own. There are many ways in which a device can be infected with malware (*e.g.,* social engineering, backdoors), which also differ vastly in how to best prevent them. These are mostly independent of NimPlant and might include non-technical problem-solving. Higher related to our project would instead be the monitoring of infected devices concerning stored files and running processes. This could lead to an interesting investigation about how NimPlant runs on infected clients. As is, infecting a client with an unmodified NimPlant implant will trigger the Windows Defender. This is not a problem we addressed in this project, and may or may not be difficult to solve (renaming the implant to something other than 'NimPlant.exe' would be an improvement), but it clearly shows that evasion mechanisms for NimPlant go beyond its network traffic.

- In addition, it would be interesting to create a larger physical setup, *i.e.,* a network of computers instead of a single client. This would be much more similar to a real botnet and would open up new possibilities for monitoring. With multiple devices communicating with the server in parallel, one could compare the network traffic of devices in the network. This would allow to identify traffic patterns more deterministically and severely weaken some evasion strategies. Of course, this would require much more sophisticated network traffic analysis algorithms, on top of an IDS on each machine.

- Training AI models in a larger setting could provide even more insights as one could collect more diverse data that could support the development of defence systems.

- Another step that could be done for future analysis would be to use HTTPS communication instead of HTTP for NimPlant which would increase the challenge for detecting and alerting the NimPlant communication. This could lead to more insights about how to deal with C2 malware that uses HTTPS.

[1] https://suricata.io/

Bibliography

[1] Faitouri A Aboaoja et al. "Dynamic Extraction of Initial Behavior for Evasive Malware Detection". In: *Mathematics* 11.2 (2023), p. 416.

[2] Amir Afianian et al. "Malware dynamic analysis evasion techniques: A survey". In: *ACM Computing Surveys (CSUR)* 52.6 (2019), pp. 1–28.

[3] D. S. Alberts and R. E. Hayes. *Understanding command and control*. 2006.

[4] Hyrum S. Anderson et al. "Evading machine learning malware detection". In: (2017).

[5] Hyrum S. Anderson et al. "Learning to evade static pe machine learning malware models via reinforcement learning". In: *arXiv preprint arXiv:1801.08917* (2018).

[6] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. *TTAnalyze: A tool for analyzing malware*. 2006.

[7] Elisa Bertino and Nayeem Islam. "Botnets and Internet of Things Security". In: *Computer* 50.2 (Feb. 2017), pp. 76–79.

[8] Qifang Bi et al. "What is Machine Learning? A Primer for the Epidemiologist". In: *American Journal of Epidemiology* 188.12 (Oct. 2019), pp. 2222–2239.

[9] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. "Adversarial Machine Learning in Malware Detection: Arms Race between Evasion Attack and Defense". In: *2017 European Intelligence and Security Informatics Conference (EISIC)*. 2017, pp. 99–106.

[10] Ping Chen et al. "Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware". In: *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30-June 1, 2016, Proceedings 31*. Springer. 2016, pp. 323–336.

[11] S. Chowdhury, M. Khanzadeh, and R. et al. Akula. "Botnet detection using graph-based feature clustering". In: *Journal of Big Data* 4.14 (2017).

[12] Farhood Farid Etemad and Payam Vahdani. "Real-time Botnet command and control characterization at the host level". In: *6th International Symposium on Telecommunications (IST)*. Nov. 2012, pp. 1005–1009.

[13] Alexander G. Eustis. "The Mirai Botnet and the Importance of IoT Device Security". In: *16th International Conference on Information Technology-New Generations (ITNG 2019)*. Ed. by Shahram Latifi. Vol. 800. 2019, pp. 85–89.

[14] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. "A Survey of Botnet and Botnet Detection". In: *2009 Third International Conference on Emerging Security Information, Systems and Technologies*. 2009, pp. 268–273.

[15] Fortra. *Malleable Command and Control*. 2023. URL: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2_main.htm?cshid=1062 (visited on July 31, 2023).

[16] Fortra. *Software for Adversary Simulations and Red Team Operations*. 2023. URL: https://www.cobaltstrike.com/ (visited on July 31, 2023).

[17] Fortra. *Welcome to Cobalt Strike*. 2023. URL: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/welcome_main.htm#_Toc65482705 (visited on July 31, 2023).

[18] Lothar Fritsch, Aws Jaber, and Anis Yazidi. "An Overview of Artificial Intelligence Used in Malware". In: *Nordic Artificial Intelligence Research and Development*. Ed. by Evi Zouganeli et al. 2022, pp. 41–51.

[19] Joseph Gardiner, Marco Cova, and Shishir Nagaraja. *Command & Control: Understanding, Denying and Detecting - A review of malware C2 techniques, detection and defences*. June 2015. URL: http://arxiv.org/abs/1408.1136 (visited on Aug. 14, 2023).

[20] Marco Gaudesi et al. "Malware Obfuscation through Evolutionary Packers". In: GECCO Companion '15. 2015, pp. 757–758.

[21] Shilpi Goel et al. "Verifying X86 Instruction Implementations". In: CPP 2020. 2020, pp. 47–60.

[22] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. URL: https://doi.org/10.48550/arXiv.1406.2661.

[23] Goppit. "Portable executabler file format - a reverse engineer view". In: *CodeBreakers Magazine* 1.2 (2006).

[24] Guofei Gu, Junjie Zhang, and Wenke Lee. "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic". In: *Computer Science and Engineering Faculty Publications*. 2008.

[25] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. "A Study of the Packer Problem and Its Solutions". In: *Recent Advances in Intrusion Detection*. Ed. by Richard Lippmann, Engin Kirda, and Ari Trachtenberg. 2008, pp. 98–115.

[26] Xiaojun Guo et al. "Progress in Command and Control Server Finding Schemes of Botnet". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. Aug. 2016, pp. 1723–1727.

[27] Ryan Heartfield and George Loukas. "A Taxonomy of Attacks and a Survey of Defence Mechanisms for Semantic Social Engineering Attacks". In: *ACM Computing Surveys* 48.3 (Feb. 2016), pp. 1–39.

[28] Guillermo Iglesias, Edgar Talavera, and Alberto DÃaz-Ãlvarez. "A survey on GANs for computer vision: Recent research, analysis and taxonomy". In: *Computer Science Review* 48 (2023), p. 100553.

[29] Djorđe D. Jovanović and Pavle V. Vuletić. "Analysis and Characterization of IoT Malware Command and Control Communication". In: *2019 27th Telecommunications Forum (TELFOR)*. 2019, pp. 1–4.

[30] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* 4 (May 1996), pp. 237–285.

[31] Navdeep Kaur and Maninder Singh. "Botnet and botnet detection techniques in cyber realm". In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. Vol. 3. Aug. 2016, pp. 1–7.

[32] Sheharbano Khattak et al. "A Taxonomy of Botnet Behavior, Detection, and Defense". In: *IEEE Communications Surveys & Tutorials* 16.2 (2014), pp. 898–924.

[33] Dhilung Kirat, Jiyong Jang, and Marc Stoecklin. *Deeplocker–concealing targeted attacks with ai locksmithing*. 2018. URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Kirat-DeepLocker-Concealing-Targeted-Attacks-with-AI-Locksmithing.pdf (visited on July 24, 2023).

[34] Dhilung Kirat, Jiyong Jang, and Marc Stoecklin. *Deeplocker: How AI can power a stealthy new breed of malware*. 2018. URL: https://securityintelligence.com/deeplocker-how-ai-can-power-a-stealthy-new-breed-of-malware/ (visited on July 29, 2023).

[35] Rajesh Kumar Yadav and Karamveer Karamveer. "A Survey on IOT Botnets and their Detection Approaches". In: *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*. Dec. 2022, pp. 1901–1906.

[36] Jehyun Lee et al. "Tracking multiple C&C botnets by analyzing DNS traffic". In: *2010 6th IEEE Workshop on Secure Network Protocols*. 2010, pp. 67–72.

[37] Yuxi Li. *Deep Reinforcement Learning: An Overview*. Nov. 2018. URL: http://arxiv.org/abs/1701.07274 (visited on Dec. 6, 2023).

[38] Artur Marzano et al. "The Evolution of Bashlite and Mirai IoT Botnets". In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. June 2018, pp. 00813–00818.

[39] Muhammad Mahmoud, Manjinder Nir, and Ashraf Matrawy. "A Survey on Botnet Architectures, Detection and Defences". In: *International Journal of Network Security* 17.3 (May 2015).

[40] Daniel Plohmann et al. "A Comprehensive Measurement Study of Domain Generating Malware". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. 2016, pp. 263–278.

[41] Anusmita Ray and Asoke Nath. "Introduction to Malware and Malware Analysis: A brief overview". In: *International Journal* 4.10 (2016).

[42] Maria Rigaki and Sebastian Garcia. "Bringing a GAN to a Knife-Fight: Adapting Malware Communication to Avoid Detection". In: *2018 IEEE Security and Privacy Workshops (SPW)*. 2018, pp. 70–75.

[43] Fatima Salahdine and Naima Kaabouch. "Social Engineering Attacks: A Survey". In: *Future Internet* 11.4 (Apr. 2019), p. 89.

[44] BC SECURITY. *Dropbox*. 2023. URL: https://bc-security.gitbook.io/empire-wiki/listeners/dropbox (visited on July 29, 2023).

[45] BC SECURITY. *Empire*. 2023. URL: https://bc-security.gitbook.io/empire-wiki/ (visited on July 29, 2023).

[46] BC SECURITY. *Malleable C2*. 2023. URL: https://bc-security.gitbook.io/empire-wiki/listeners/malleable-c2 (visited on July 29, 2023).

[47] Seungwon Shin and Guofei Gu. "Conficker and beyond: A Large-Scale Empirical Study". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. 2010, pp. 151–160.

[48] Sérgio S. C. Silva et al. "Botnets: A survey". In: *Computer Networks*. Botnet Activity: Analysis, Detection and Shutdown 57.2 (Feb. 2013), pp. 378–403.

[49] Snort. *POLICY-OTHER HTTP request by IPv4 address attempt*. 2023. URL: https://www.snort.org/rule_docs/1-50447 (visited on Sept. 24, 2023).

[50] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. Nov. 2018.

[51] Simon Nam Thanh Vu et al. “A Survey on Botnets: Incentives, Evolution, Detection and Current Trends”. In: *Future Internet* 13.8 (July 2021), p. 198.

[52] Shun Tobiyama et al. “Malware Detection with Deep Neural Network Using Process Behavior”. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. June 2016, pp. 577–582.

[53] Van Tong and Giang Nguyen. “A Method for Detecting DGA Botnet Based on Semantic and Cluster Analysis”. In: *Proceedings of the 7th Symposium on Information and Communication Technology*. SoICT ’16. 2016, pp. 272–277.

[54] Enn Tyugu. “Command and control of cyber weapons”. In: *2012 4th International Conference on Cyber Conflict (CYCON 2012)*. June 2012, pp. 1–11.

[55] UnderDefense. *How to Detect CobaltStrike Command & Control Communication*. 2021. URL: https://underdefense.com/guides/how-to-detect-cobaltstrike-command-control-communication/ (visited on July 31, 2023).

[56] C. Van Cooten. *Building a C2 Implant in Nim - Considerations and Lessons Learned*. 2021. URL: https://casvancooten.com/posts/2021/08/building-a-c2-implant-in-nim-considerations-and-lessons-learned/#nim-for-offensive-security.

[57] C. Van Cooten et al. *NimPlant - A light firststage C2 implant written in Nim and Python*. GitHub. URL: https://github.com/chvancooten/NimPlant.

[58] Adversary Village. *NimPlant C2 - Cas Van Cooten | Adversary Guru series #4 | Adversary Village - YouTube*. Mar. 2023. URL: https://www.youtube.com/watch?v=HIEB24IrMls&ab_channel=AdversaryVillage (visited on Dec. 14, 2023).

[59] Ruchi Vishwakarma and Ankit Kumar Jain. “A survey of DDoS attacking techniques and defence mechanisms in the IoT network”. In: *Telecommunication Systems* 73.1 (Jan. 2020), pp. 3–25.

[60] VMRAY. *Malware Sandbox*. 2023. URL: https://www.vmray.com/glossary/malware-sandbox/ (visited on July 29, 2023).

[61] Gernot Vormayr, Tanja Zseby, and Joachim Fabini. “Botnet Communication Patterns”. In: *IEEE Communications Surveys & Tutorials* 19.4 (2017), pp. 2768–2796.

[62] Lanier Watkins et al. “Using inherent command and control vulnerabilities to halt DDoS attacks”. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. Oct. 2015, pp. 3–10.

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Networks
CLI	Command-Line Interface
CS	Cobalt Strike
C2	Command-and-Control
DDNS	Dynamic Domain Name System
DDoS	Distributed Denial of Service
DNN	Deep Neural Network
DNS	Domain Name System
DGA	Domain Generative Algorithms
GAN	Generative Adversarial Network
GUI	Graphical User Interface
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
IRC	Internet Relay Chat
IoT	Internet of Things
PE	Portable Executable
PEB	Process Environment Blocks
P2P	Peer-to-Peer
RAT	Remote Access Trojan
RL	Reinforcement Learning

List of Figures

List of Tables

Listings

Appendix A

Additional Contents

A.1 Repositories

Several repositories were defined in GitHub for the project *MAP-Cyber-Security-AI* under the url `https://github.com/MAP-Cyber-Security-AI`. The following table summarizes information related to those repositories.

Repository	**Content**	**URL**
Nimplant	Contains the code related to all NimPlant stages. The codes for the stages are in the corresponding branches. The *main* branch contains the initial NimPlant code, the *evasion-stage-2* branch contains the stage with the command-based evasion and the *evasion-stage-3* branch contains the code and the data related to the training of the RL agent.	`https://github.com/MAP-Cyber-Security-AI/Nimplant`
Snort-Setup	Contains the description for the setup of Snort. The *main* branch contains the initial Snort setup and the branch *configuration-stage-2* contains the Snort setup with our rules that are relevant for this project.	`https://github.com/MAP-Cyber-Security-AI/Snort-Setup`
Architecture	Contains the draw.io file that was used to create the architecture of NimPlant for the sections 4.1.1, 4.1.2 and 4.1.4.	`https://github.com/MAP-Cyber-Security-AI/Architecture`
Infrastructure-Configuration	Contains information related to the infrastructure used for this project.	`https://github.com/MAP-Cyber-Security-AI/Infrastructure-Configuration`

Table A.1: Repositories

A.2 SharePoint

A SharePoint folder was created where the mid and final presentations, the zipped folder with the Code of all NimPlant stages and a copy of the report were uploaded. The following URL leads to the SharePoint folder: `https://uzh.sharepoint.com/:f:/s/MPCybersecurityAI/EiPeardFc3ZItOm2ABOYrq8B4TUZteTvsl_PJqQo8zicrQ`