WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

BACHELOR THESIS

SUPERVISOR:

lect. dr. Cristian Cira

GRADUATE:

Valase Paul Mihai

TIMIŞOARA
2025

WEST UNIVERSITY OF TIMIOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

Development of a Command-and-Control (C2) Framework with Go-Based Implants and React Web Interface for Threat Emulation

SUPERVISOR: lect. dr. Cristian Cira

GRADUATE: Valase Paul Mihai

TIMIOARA
2025

Abstract

This bachelor's thesis outlines the design, implementation, and evaluation of a **Command-and-Control (C2) system** developed for cybersecurity research and threat emulation. The system features a robust backend server developed in **Go** utilizing the **Gin Gonic framework**, a contemporary web-based operator interface constructed with **React**, and **cross-platform implants** compatible with both **Windows** and **Linux** operating systems. **Go-based implants** are lightweight entities deployed on target machines that establish enduring communication linkages to the command and control server. This is the operational mechanism of the framework. This enables operators to **execute commands remotely**, **interact with the filesystem**, and **stream the live desktop** of a compromised server. The primary objective of this project is to develop a **modular and extendable platform** capable of simulating the **Tactics, Techniques, and Procedures (TTPs)** of actual adversaries in controlled environments. The implants employ several methods to circumvent conventional security monitoring. These characteristics encompass **daemonization**, enabling the software to operate as concealed background processes, **altering process names** to resemble legitimate software, and **self-deletion**, which eradicates forensic evidence from the disk. The thesis examines the implementation of these methodologies to attain **stealth and persistence** on a target system. Experimental validation substantiated the framework's fundamental functionalities, showcasing **remote command execution**, **filesystem enumeration**, **snapshot capture**, and the efficacy of its principal **evasion mechanisms** against conventional security configurations. This project provides a functional, **command and control system** that security professionals can utilize to evaluate various defense strategies, researchers can employ to analyze adversarial tactics, and students can access for educational purposes. This report emphatically underscores the significance of ethics and the appropriate use of the framework for defense research.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The modern cybersecurity environment is marked by the increasing complexity of threat actors. These adversaries frequently utilize Advanced Persistent Threats (APTs)[1]sustained and focused cyberattacks intended to preserve enduring, covert accessto penetrate and dominate essential infrastructure[2] [SH12]. At the core of these operations are Command and Control (C2) frameworks, the infrastructure used to oversee compromised systems and sustain operational persistence[3], conduct internal reconnaissance, and exfiltrate sensitive data. The strategies utilized by these frameworks are extensively recorded in industry standards such as the MITRE ATT&CK® framework, which classifies C2 as a crucial phase of the attack lifecycle [The25a].

This necessitates the establishment of realistic and controlled settings for cybersecurity professionals and students to simulate, analyze, and devise defenses against advanced threats. The current ecosystem of C2 tools poses considerable obstacles for academic and research applications. Commercial platforms that adhere to industry standards, such as Fortra's `Cobalt Strike` [For25] and Rapid7's `Metasploit Pro` [Rap25], possess considerable capabilities but impose substantial obstacles. Their exorbitant costs render them inaccessible to several academic institutions and individual researchers. Moreover, their proprietary and closed-source characteristics inhibit the comprehensive architectural study required for educa-

[1]Advanced Persistent Threats (APTs) denote extended and focused cyberattacks wherein an intruder acquires unauthorized access to a network and remains undetected for a protracted duration. They are frequently ascribed to state-sponsored entities.

[2]Critical infrastructure encompasses the assets, systems, and networks, both physical and virtual, deemed essential such that their impairment or destruction would significantly undermine security, the national economy, or public health and safety.

[3]Operational persistence denotes strategies employed by an adversary to sustain prolonged access to a compromised system, despite reboots, credential alterations, or other system modifications.

tional inquiry.

Although the open-source community has generated various alternatives, including `Sliver` [Bis25] and `Havoc` [Hav25], many are plagued by inconsistent documentation, convoluted deployment processes, or architectural constraints that impede their effectiveness as educational instruments. This scenario presents both a practical need for an accessible educational tool and a significant research opportunity: to explore the architectural and implementation principles necessary for constructing a contemporary, resilient, and ethical C2 framework from the ground up. Comprehending the design trade-offs, security implications, and software engineering issues inherent in developing such a system constitutes a significant scholarly pursuit.

This thesis tackles this difficulty by detailing the design and implementation of an academically-focused C2 framework. The project utilizes contemporary software engineering principles, incorporating the Go programming language for a high-performance, concurrent backend [DK15], a responsive web interface developed with React [Met25], and a modular design to guarantee usability and extensibility. This project seeks to create and evaluate a system that serves as a practical instrument for threat emulation[4] and a definitive, reproducible case study on the architecture of contemporary command and control (C2) frameworks.

1.2 Primary Objectives

The principal objective of this research is to devise, execute, and assess a Command and Control structure specifically designed for cybersecurity teaching and experimentation. This purpose is further delineated into the subsequent technical and academic aims.

1.2.1 Primary Technical Objectives

The technological objectives of this thesis concentrate on developing a comprehensive and operational C2 system. This commences with the **Architecture Design and Implementation**, which entails constructing a scalable and modular backend utilizing the Go programming language and the Gin web framework, proficient in handling several concurrent implant[5] connections without jeopardizing stability or performance. A fundamental requirement is **Cross-Platform Implant Development**, particularly the creation of Go-based implants that are interoperable

[4]Threat emulation entails replicating the tactics, methods, and procedures (TTPs) of actual threat actors within a controlled setting to evaluate an organization's security posture.

[5]An implant is a type of malicious software (malware) installed on a hacked system that interacts with a command and control (C2) server to receive directives and exfiltrate data. Also known as an agent or beacon.

with both Windows and Linux environments and provide important C2 functionalities such as command execution and filesystem interface. To guarantee usability, a primary goal is **User Interface Development**, which involves creating a responsive React-based online interface for straightforward implant maintenance, safeguarded by contemporary authentication techniques such as JSON Web Tokens (JWTs)[6]. Central to the entire project is a comprehensive **Security Implementation**, implementing stringent measures such as encrypted communications, secure authentication systems, and Role-Based Access Control (RBAC)[7]. The framework is augmented with **Advanced Feature Integration**, incorporating realistic attacker capabilities such as real-time desktop streaming, comprehensive filesystem traversal[8], and screenshot capture to replicate real-world scenarios.

1.2.2 Academic and Research Objectives

This initiative aims to generate a significant resource for the cybersecurity community through its academic and scientific contributions. This entails the creation of an instructional framework, encompassing the production of tutorials, detailed code documentation, and supplementary pedagogical resources to aid in classroom and laboratory implementation. A comprehensive performance evaluation will be executed using empirical testing to measure the frameworks latency, scalability, and throughput across different load situations. A comprehensive security assessment, including both static and dynamic analysis[9] of the framework, will be utilized to identify potential vulnerabilities and inform best practices for secure usage. The project will ultimately be distributed as an open-source contribution under a permissive license to foster transparency, facilitate peer review, and promote community-driven improvements.

1.3 Statement of the Problem

The proficient analysis and disassembly of adversary tradecraft are essential components of contemporary cybersecurity education. Command and Control (C2) structures are fundamental to this discipline, functioning as the operational foun-

[6]JSON Web Tokens (JWTs) are a concise, URL-safe method for conveying claims between entities. They are frequently utilized in online applications for secure authentication and session management. See: https://jwt.io/

[7]Role-Based Access Control (RBAC) is a security framework that limits system access to authorized individuals according to their organizational positions.

[8]Filesystem traversal is an attack technique that permits an adversary to access or manipulate files and directories located outside the designated application directory.

[9]Methods for defect analysis in software. Static analysis evaluates the code without execution, whereas dynamic analysis entails executing the software to observe its behavior and detect issues such as vulnerabilities or memory leaks.

dation for practically all advanced cyber-attacks. Nonetheless, a substantial disparity exists between the instruments employed by enemies and those accessible for legitimate research and instruction. This discrepancy engenders numerous impediments that impede the advancement of proficient defenders. The fundamental problems with the existing ecosystem of C2 frameworks in academic and research environments are as follows:

Accessibility Limitations: Premium commercial frameworks that accurately simulate real-world risks are excessively costly and frequently unattainable for students, educators, and independent research organizations.

Transparency Deficiencies: Proprietary, closed-source tools do not provide visibility into their source code. The "black box" characteristic hinders comprehensive academic investigation, code examination, and a fundamental comprehension of their implementation, which is essential for developing successful detections.

Technical Constraints: Numerous free or open-source alternatives, although readily available, exhibit inadequate scalability, restricted cross-platform compatibility, unsafe design vulnerabilities, or an absence of contemporary evasion capabilities, rendering them ineffective substitutes for modern threats.

Documentation Deficiencies: Effective instructional utilization necessitates thorough documentation elucidating design decisions, operating protocols, and foundational ideas. Existing open-source solutions sometimes neglect education in favor of functionality.

Ethical Ambiguity: Certain public frameworks are exclusively offensive in nature and lack definitive ethical rules or pedagogical protections, hence restricting their appropriateness for responsible application within a formal academic program.

The cumulative effect of these obstacles significantly impedes experiential education and empirical research. This training deficiency leaves aspiring security professionals insufficiently equipped to understand the speed, nuance, and ramifications of a modern C2-driven attack. An assailant can employ a C2 framework to transition from initial access to total system control within minutes. Once contact is established through an implant, the attacker gains the immediate capability for comprehensive monitoring, sensitive data exfiltration, lateral movement inside the network, and the dissemination of malicious payloads such as ransomware. This thesis addresses the lack of a C2 framework designed to bridge this gap: a framework that is transparent, accessible, and technically robust enough to serve as an effective platform for teaching the principles of modern cyber-attacks and for examining defenses against them.

1.4 Ethical Considerations

Considering the possible dual-use characteristics[10] of Command and Control frameworks, this project implements intentional measures to guarantee ethical and responsible utilization. These issues are informed by concepts of responsible growth and a dedication to legal and regulatory adherence.

1.4.1 Principles of Responsible Development

Design Focused on Education: The system is designed exclusively for educational and research reasons, not for illicit access or malicious activities. Its attributes and documentation are congruent with educational objectives.

Isolated Testing Environments: All development, testing, and evaluation were performed within isolated virtual environments[11] to avert any unwanted influence on live networks or systems.

Clear Documentation of Risks: The system's restrictions and potential misuse scenarios are thoroughly defined to ensure users comprehend its intended purpose and ethical constraints.

Clear and Accessible Procedure: The development process is conducted transparently to promote community feedback, independent security evaluation, and ongoing ethical discourse.

1.5 Thesis Structure

The subsequent sections of this thesis are structured as follows:

Chapter 1 Introduction: Articulates the rationale for this research, delineates the primary objectives (both technical and intellectual), introduces the issue statement, and examines ethical implications.

[10] Dual-use denotes technology that can serve both lawful and nefarious goals. A C2 framework intended for educational purposes may also be misappropriated for illicit attacks.

[11] A sandboxed environment is an isolated testing area on a system that prohibits programs from touching the host machine or network, hence restricting their activity to avert any unwanted influence on live networks or systems.

Chapter 2 Background and Terminology: Examines fundamental principles pertinent to Command and Control (C2) frameworks, malware architecture[12], and key threat models[13].

Chapter 3 State of the Art: Examines contemporary C2 tools and frameworks, evaluates their structures, and highlights deficiencies in the literature that this thesis solves.

Chapter 4 Implementation Details: Delivers an exhaustive technical delineation of the deployed system, encompassing the C2 server, the implant, and the web client.

Chapter 5 Experimental Setup and Results: Elucidates the testing environment, experimental conditions, and provides comprehensive results and observations.

Chapter 6 Conclusions: Summarizes the research findings, examines broader security implications and system limits, and proposes potential avenues for future work and enhancement.

[12] Malware architecture denotes the structural design of malicious software, encompassing its components, such as droppers, payloads, and command-and-control communication mechanisms.

[13] A threat model is a systematic approach to identifying, assessing, and mitigating potential security threats pertinent to a system or application.

Chapter 2

Background and Terminology

This chapter lays the groundwork for future discussions by introducing the essential concepts and technical vocabulary required to understand Command-and-Control (C2) frameworks, malware implants, and threat emulation. It imparts the fundamental information required to contextualize the system's architecture and function within the broader fields of malware investigation and cybersecurity operations.

2.1 Command-and-Control (C2/C&C)

A Command-and-Control (C2 or C&C) server is essentially a centralized system overseen by a threat actor or, in lawful contexts, by a red team[1] to oversee and coordinate compromised hosts. These hosts, referred to as implants or agents, interact with the C2 server to obtain directives, convey their status, and extract data. This complete C2 infrastructure[2] is an essential element of post-compromise operations and falls under the *Command and Control (TA0011)* tactic within the MITRE ATT&CK® framework [The25a].

2.2 Implant, Agent, and Beacon

Software operating on a hacked endpoint is referred to by several terms: implant, beacon, agent, bot, or backdoor.[3] Its major function is to provide a clandestine communication channel to the C2 server, facilitating remote control of the compro-

[1] Red teams are security experts sanctioned to replicate actual assaults in order to evaluate an organization's detection and response capability.

[2] The term C2 infrastructure refers to the comprehensive network of servers, redirectors, domains, and communication channels utilized by an operator to oversee and control compromised systems.

[3] These phrases are frequently used interchangeably, however they may possess subtle distinctions in meaning. For example, "bot" is predominantly utilized in botnet contexts, whereas "agent" is often employed in red teaming activities.

mised system [SH12]. The functionality of an implant is determined by numerous essential characteristics. It participates in *beaconing*,[4] where it intermittently communicates with the C2 server at a configurable interval to retrieve new tasks or report outcomes. The identification of this consistent, heartbeat-like activity is a primary concern for network security solutions [Net23]. This channel also enables *task execution*, permitting an operator to execute arbitrary commands, interact with the filesystem, or collect system intelligence. An essential component of contemporary implants is *evasion*, wherein they proactively strive to elude detection by antivirus (AV) and Endpoint Detection and Response (EDR)[5] solutions.

2.3 Payload

In malware or threat emulation, a payload is the component that performs a specific malicious activity.[6] These steps may include creating persistence, collecting data, and encrypting files for ransomware purposes. In numerous C2 architectures, the implant functions as the principal payload, transmitted to the target system during a post-exploitation phase [SH12]. Upon activation, it accomplishes its goal by ensuring continuous C2 communication and providing the operator with a platform to execute additional commands or deploy supplemental payloads.

2.4 Threat Emulation versus Penetration Testing

Despite their similarities, threat emulation and penetration testing are distinct security assessment methodologies with differing objectives. `Penetration Testing` primarily seeks to detect and exploit numerous system vulnerabilities within a certain scope.[7] Its objective prioritizes breadth rather than depth, and it is typically less focused on emulating the particular actions of a recognized foe.

Conversely, `Threat Emulation` is a more precise and focused field. In this context, red teamers diligently emulate the Tactics, Techniques, and Procedures (TTPs)[8] of known threat actors, often informed by frameworks such as MITRE

[4] Beaconing is the procedure by which an implant transmits intermittent network communication to its controller. The frequency and jitter of this communication can be adjusted to avoid detection.

[5] Endpoint Detection and Response (EDR) solutions provide continuous monitoring and automated response to threats by analyzing behavioral patterns, system events, and other telemetry from endpoints [Cro25].

[6] Payloads may be either modular or monolithic. They may encompass secondary phases that execute functions like as privilege escalation or credential extraction.

[7] A standard penetration test output comprises a catalog of vulnerabilities, proof-of-concept attacks, and prioritized remediation suggestions.

[8] Tactics, Techniques, and Procedures (TTPs) delineate the operational methods of adversaries, encompassing overarching strategic objectives (Tactics) to the precise technical methodologies

ATT&CK® [The25d]. The primary objective is not merely to identify vulnerabilities, but to systematically evaluate the efficacy of an organization's security posture in detecting and responding to specific, realistic attacker actions.

2.5 Operational Concepts and Techniques

2.5.1 Common C2 Communication Channels

Stealth, the target environment, and the availability of protocols are factors that affect the selection of communication protocols employed by C2 frameworks to manage implants. The predominant channel is `HTTP/HTTPS`,[9] owing to its extensive utilization and the simplicity with which nefarious traffic can be camouflaged as authentic web browsing.

Adversaries moreover utilize more clandestine channels. The `DNS` protocol is frequently exploited to exfiltrate data or to receive commands concealed within DNS queries or responses.[10] Likewise, `ICMP` packets can be utilized for data transmission by concealing messages within conventional echo request and reply packets.[11] In advanced campaigns, attackers may exploit legal `Cloud Services or Social Media` platformssuch as GitHub, Dropbox, or Twitterto host payloads or transmit commands, thereby further concealing their actions [Net23].[12]

2.5.2 Evasion Techniques and Defensive Counterparts

To evade contemporary security measures, implants utilize a range of evasion strategies, collectively classified under the Defense Evasion tactic (TA0005) in the MITRE ATT&CK® framework [The25c]. A crucial technique is `daemonization and background execution`,[13] assuring its covert operation without a visible window or console. A prevalent strategy is `self-deletion`, in which the implant's binary is eliminated from the disk post-execution to obliterate forensic evidence.[14]

To enhance its concealment, an implant may employ `process name spoofing`,

employed for implementation (Techniques and Procedures).

[9] Due to the essential nature of HTTP/HTTPS traffic for modern commerce, it is typically permitted by firewalls and proxies, rendering it an ideal choice for obfuscation.

[10] Known as DNS tunneling, this technique can be difficult to detect without deep packet inspection or sophisticated anomaly detection.

[11] ICMP is commonly utilized for network diagnostics, such as ‘ping‘, and is regarded as innocuous; hence, its traffic is frequently disregarded by security monitoring.

[12] This method, frequently referred to as "living off the trusted land," exploits services that are already authorized and deemed reliable within the majority of corporate settings.

[13] Daemonization is typically accomplished by forking a child process and terminating the parent, so detaching the implant from the user's terminal and session.

[14] Self-deletion can be executed by batch scripts, PowerShell commands, or direct API calls such as ‘DeleteFileA‘ to initiate eradication once the virus is active in memory.

adopting a designation that resembles a benign system process such as `svchost.exe` or `explorer.exe` to deceive system administrators. Finally, several contemporary implants employ `in-memory execution`, executing malicious code directly in RAM without ever persisting it to disk.[15]

Beyond runtime evasion, implants often incorporate features to thwart reverse engineering, which is the static and dynamic analysis of the binary itself by researchers using tools like IDA Pro, Ghidra, and Binary Ninja. To complicate static analysis, malware authors employ `code obfuscation`, which scrambles the program's logic, and use `packers or crypters`. These tools compress or encrypt the main malicious code, meaning the true payload is only revealed in memory during execution, making on-disk analysis ineffective. To defeat dynamic analysis, implants use `anti-debugging` techniques, such as programmatically checking if a debugger is attached (‘IsDebuggerPresent‘ on Windows), using timing checks to detect the performance overhead of a debugger, or leveraging complex exception handling to crash analysis tools. These defensive layers are designed to significantly raise the cost and complexity of analyzing the implant's functionality [SH12].

Although these techniques may prove efficient against conventional, signature-based antivirus (AV) solutionsfrequently assessed using services such as VirusTotal[16]they generally underperform versus contemporary Endpoint Detection and Response (EDR) systems. EDRs utilize behavioral analysis, frequently employing techniques such as API hooking,[17] to spot malicious activity. In response, advanced C2 frameworks like `Havoc` [Hav25] and `Sliver` [Cyb23] have developed complex countermeasures, including direct syscalls and sleep obfuscation, to circumvent these enhanced defenses.[18]

2.6 Industry Frameworks and Standards

2.6.1 The MITRE ATT&CK Framework

The MITRE ATT&CK® structure is fundamental to contemporary cybersecurity. It is a publicly accessible, community-driven repository that categorizes adversary behaviors into a systematic taxonomy of tactics and approaches [The25d]. It offers a standardized vocabulary for blue teams, red teams, and threat intelligence

[15] Common in-memory execution techniques encompass reflected DLL injection, process hollowing, and direct memory manipulation via APIs such as ‘VirtualAlloc‘ and ‘CreateThread‘.

[16] VirusTotal is a prevalent online service that consolidates numerous antivirus engines to analyze files and URLs, utilized by both defenders and attackers to evaluate detection rates [Vir25].

[17] API hooking entails intercepting function calls between applications and the operating system, usually at user-mode libraries such as ‘ntdll.dll‘, to examine or modify runtime behavior.

[18] Direct system calls circumvent user-mode hooks by directly accessing system services, whereas sleep obfuscation distorts time signals to avoid behavioral detection during implant inactivity.

analysts to synchronize defensive measures and security assessments with actual adversarial activities. The attributes of any C2 framework may be directly correlated with ATT&CK tactics, including *Execution (TA0002)*, *Persistence (TA0003)*, *Defense Evasion (TA0005)*, *Discovery (TA0007)*, *Collection (TA0009)*, and, of course, *Command and Control (TA0011)*.[19]

2.6.2 Cybersecurity Standards and Vulnerability Management

Alongside hostile tradecraft, the cybersecurity domain is governed by regulations and standards that enable uniform communication regarding defense strategies. The National Institute of Standards and Technology (NIST) provides fundamental guidance, exemplified by the NIST Cybersecurity Framework (CSF)[20] which furnishes a policy framework for organizations to evaluate and enhance their capacity to manage cybersecurity risk [Nat25].

Upon the discovery of a specific software vulnerability, it is assigned a Common Vulnerabilities and Exposures (CVE) number,[21] hence establishing a universal reference for a known security defect. The gravity of these vulnerabilities is usually assessed with the Common Vulnerability Scoring System (CVSS),[22] an open framework that delivers an objective, numerical assessment of the flaw's severity.

[19] This mapping assists red teams in establishing operational objectives and enables blue teams to prioritize the formulation of detection rules and threat hunting queries.

[20] The NIST CSF is organized around five core functions: Identify, Protect, Detect, Respond, and Recover.

[21] A CVE ID functions as a unique, standardized reference, enabling security tools, vendors, and researchers to communicate consistently regarding a specific vulnerability [The25b].

[22] CVSS assigns a score ranging from 0.0 (low) to 10.0 (critical) based on variables such as attack complexity, impact, and exploitability, hence assisting companies in prioritizing remediation activities [FIR25].

Chapter 3

State of the Art

3.1 Introduction

The establishment of Command and Control (C2) frameworks is a critical domain within cybersecurity. It encompasses both aggressive security operations and defensive analytical tools. This chapter provides a comprehensive analysis of current developments in C2 framework progress. It examines existing solutions, evaluates their architectural paradigms, and contextualizes their contributions within the broader cybersecurity research community. Essential information in this field, as clarified by Sikorski and Honig in "Practical Malware Analysis," provides the foundation for understanding the fundamental operations of these systems [SH12]. This systematic review establishes the theoretical foundation for this thesis, identifies critical shortcomings in existing C2 solutions, and contextualizes this research within the wider field of cybersecurity studies, often categorized by resources such as the MITRE ATT&CK® framework [The25d].

3.2 Taxonomy of Command and Control Frameworks

To comprehend the present situation, it is essential to systematically categorize Command and Control frameworks. The architectures and functionalities of C2 systems can be classified according to their design, communication protocols, and operational applications [AMCH17]. Contemporary frameworks range from publicly accessible open-source tools to highly proprietary, government-grade surveillance systems. This environment illustrates a sophisticated cybercriminal and nation-state ecosystem characterized by the development, sale, and repurposing of tools, a trend repeatedly noted in threat intelligence reports [Man24]. This section assesses the alignment of several framework categories with established adversarial behaviors by analyzing representative cases from each major category.

3.2.1 Commercial and Government-Grade C2 Solutions

The pinnacle of the market comprises commercial and government-grade C2 frameworks. These platforms include comprehensive post-exploitation functionalities, advanced evasion strategies, and are frequently offered as a complete service solution. `Cobalt Strike`, currently managed by Fortra, serves as the de facto industry standard for red team operations, characterized by its renowned Beacon implant and Malleable C2 profiles that enable traffic to emulate legitimate services [For25]. Although potent, its recognized signs are closely scrutinized by defensive teams.

Beyond authorized red teaming tools exists the clandestine realm of the cyber-arms black market and state-sponsored surveillance technologies, a subject thoroughly examined in Nicole Perlroth's "This Is How They Tell Me the World Ends" [Per21]. An illustrative instance is the Pegasus spyware, created by the Israeli company NSO Group [Amn25]. Pegasus epitomizes the pinnacle of C2 technology, engineered for "zero-click" infections.[1] It accomplishes this by utilizing zero-day exploits[2]software vulnerabilities that remain unidentified by the manufacturer and for which no corrective fix is available. The mechanism employed in the intricate "Operation Triangulation" attack involved a zero-day exploit that targeted a vulnerability in an image rendering library, enabling an attacker to obtain complete control of an iOS device by transmitting a meticulously designed image through iMessage [Kas23]. Upon deployment, Pegasus grants its operators total control over the device, transforming it into an omnipresent surveillance instrument. The presence of such robust frameworks underscores the pinnacle of C2 capabilities and the essential need to comprehend their foundational design concepts.

3.2.2 Open-Source C2 Frameworks

The open-source community has developed a varied selection of C2 frameworks, overcoming the accessibility constraints of commercial alternatives. `Sliver`, created by Bishop Fox, is a contemporary framework designed in Go that emphasizes cross-platform interoperability and operational security [Bis25]. Its utilization of gRPC[3] for communication and its multiplayer architecture render it an effective instrument

[1] A zero-click exploit is a cyberattack that necessitates no engagement from the victim. The attack can be effectively carried out without the user needing to click a link, open a file, or engage in any other actions that necessitate user input.

[2] A zero-day exploit targets a vulnerability in software that is unknown to the vendor and lacks an available patch. The phrase "zero-day" denotes that the developer has had no time to rectify the vulnerability.

[3] gRPC is a contemporary, high-performance open-source Remote Procedure Call (RPC) framework capable of operating in any context.

for collaborative interactions. Likewise, `Covenant` [cob25] and `Mythic` [its25] have advanced user experience by implementing web-based interfaces constructed with React [Met25, BP20] and utilizing architectural paradigms such as microservices.[4] `Havoc` [Hav25] exemplifies the implementation of sophisticated evasion strategies, including indirect syscalls[5] and sleep obfuscation.[6]

3.2.3 Academic and Research Prototypes

Academic contributions frequently emphasize the investigation of innovative research inquiries rather than the provision of exhaustive instruments. These prototypes often examine particular facets of C2, such unconventional communication methods or sophisticated avoidance strategies. A considerable amount of scholarly research also emphasizes the defensive aspect, including the machine-learning-based C2 detection techniques examined by Walter et al. [WSS17]. Nevertheless, these efforts frequently lack the sustained support and documentation necessary for extensive acceptance.

3.3 Architectural Analysis and Gap Identification

Modern C2 frameworks utilize various communication architectures, ranging from the conventional client-server paradigm shown by `Cobalt Strike` [For25] and `Sliver` [Bis25] to more robust peer-to-peer (P2P)[7] models. The progression of user interfaces has paralleled significant trends in software development, transitioning from command-line interfaces to sophisticated desktop applications and, more recently, to collaborative web-based platforms such as `Covenant` [cob25] and `Mythic` [its25].

Notwithstanding these gains, a thorough investigation uncovers substantial constraints within the existing ecosystem. A significant deficiency is the operational longevity of public frameworks. When a tool becomes popular, its default indicators are rapidly included into the detection signatures of antivirus and endpoint detection and response programs, making it worthless without considerable customiza-

[4] A microservices architecture organizes an application as a set of loosely linked services, each accountable for a distinct functionality.

[5] Indirect syscalls constitute an evasion strategy whereby a software circumvents direct system calls, frequently scrutinized by security products, by employing an intermediary instruction to redirect to the syscall's memory location.

[6] Sleep obfuscation is a method whereby malware encrypts its code in memory prior to entering a dormant state and subsequently decrypts it upon reactivation, thereby evading memory scanners.

[7] In a P2P C2 architecture, compromised hosts (peers) interact directly to transmit commands and data, eliminating dependence on a singular central server.

tion. Moreover, numerous open-source tools are deficient in advanced evasion techniques required to circumvent contemporary EDRs, especially on Windows, where defenses such as API hooking[8] are widespread. This presents a distinct research opportunity to establish a C2 framework that is both accessible for instructional purposes and functions as a platform for exploring and executing next-generation evasion tactics.

3.4 Synthesis and Research Positioning

This review indicates that although current C2 frameworks are advanced, there exists a notable deficiency in a tool that is both pedagogically accessible and sufficiently technically refined to function as a platform for contemporary evasion research. This thesis seeks to fill this vacuum by creating a framework that is visible, thoroughly documented, and modular, facilitating the seamless integration and evaluation of novel adversarial strategies.

Table 3.1 presents a consolidated comparison of the examined C2 frameworks.

Table 3.1: Comparison of Principal C2 Frameworks

Framework	**License**	**Architecture**	**UI Type**	**Documentation**	**Educational Use**
Cobalt Strike	Commercial	Client-Server	Desktop	Exceptional	Restricted
Pegasus (NSO)	Proprietary	Client-Server	N/A	N/A	None
Sliver	Open Source	Client-Server	CLI/Web	Satisfactory	Moderate
Covenant	Open Source	Web-Based	Web	Good	High
Mythic	Open Source	Microservices	Web	Moderate	Moderate
Havoc	Open Source	Client-Server	Desktop	Limited	Low

This research leverages the strengths of current paradigms while squarely confronting their recognized shortcomings. This chapter's study establishes the essential basis for the architectural decisions and implementation strategies discussed in other chapters, guaranteeing that the resulting framework significantly contributes to cybersecurity teaching and research.

[8]API hooking is a technique used by security software to intercept function calls made by an application to the operating system, allowing it to monitor for malicious behavior.

Chapter 4

Implementation Details

This chapter delves into the specific implementation details of the C2 server, the Go-based implants, and the React web client. Key code structures, algorithms, and challenging aspects are discussed.

4.1 C2 Server Implementation (Go & Gin Gonic)

4.1.1 Project Structure

The Go-based C2 server, `awesomeProject`, is structured to promote modularity and maintainability. The project's entry point is `main_go`, which initializes the Gin Gonic router, establishes database connections through a GORM instance, configures all API routes by calling `routes.SetupRouter()`, and starts the HTTP server. The `config/` directory centralizes application-wide configurations, most notably the database connection setup and JWT secret keys. All API endpoints are defined in the `routes/` directory, where a central `routes_go` file contains a `SetupRouter` function to group public, implant-facing, and operator-facing protected routes.

The core business logic resides in the `controllers/` directory, which houses the Gin handler functions. Key files include `auth_go` for user authentication and `implant_controllers_go`, which contains the comprehensive logic for all implant interactions, such as check-ins (`CheckinImplant`), command fetching (`ImplantClientFetchCommands`), result handling (`HandleCommandResult`), implant generation (`GenerateImplant`), and feature management like livestreaming (`HandleLivestreamFrame`) and screenshots (`GetScreenshotsForImplant`). Custom Gin middleware, including the crucial JWT authentication middleware (`AuthMiddleware()`), is located in the `middleware/` directory. Data persistence is managed through GORM data structures defined in the `models/` directory, which map to database tables for users (`user_go`), implants (`implant_go`), commands (`command_go`), and other entities like screenshots (`screenshot_info_go`) and filesystem entries (`fs_entry_go`). A dedicated data access layer in the `database/` directory abstracts all database operations. Finally, the `binaries/` directory stores pre-compiled base implant executables for Windows (`base_client_windows.exe`) and Linux (`base_client_linux`), which serve as templates for on-demand configuration. This organization effectively separates concerns, making the codebase easier to navigate and extend.

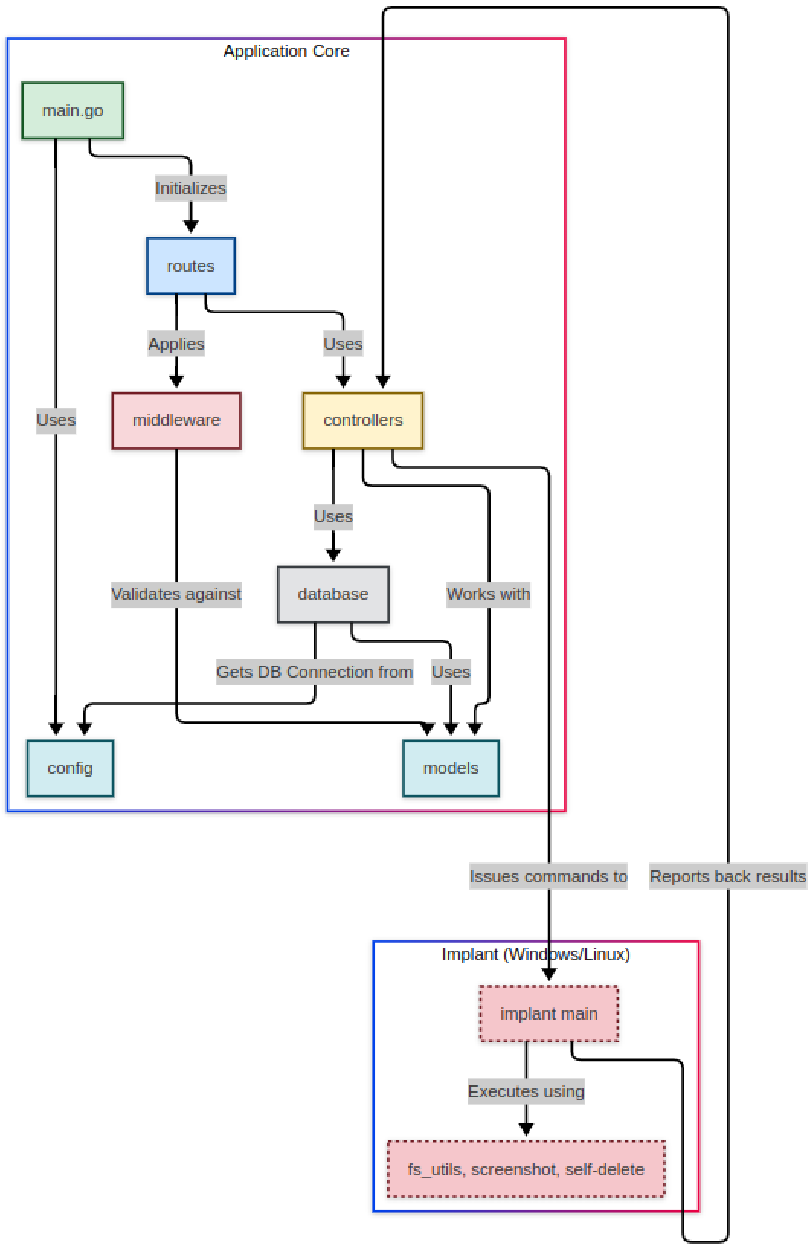


Figure 4.1: Packages Diagram.

4.1.2 API Endpoint Handling

The C2 server exposes several RESTful API endpoints for communication with both the implants and the operator's web client. The Gin Gonic framework is utilized for routing and request handling. Below is a representative sample of the route setup:

```
func SetupRouter() *gin.Engine {
    r := gin.Default()

    r.POST("/checkin", controllers.CheckinImplant)
    r.GET("/implant-client/:unique_token/commands", controllers.ImplantClientFetchCommands)
    r.POST("/command-result", controllers.HandleCommandResult)
    r.POST("/livestream-frame", controllers.HandleLivestreamFrame)

    r.POST("/register", controllers.Register)
    r.POST("/login", controllers.Login)

    protected := r.Group("/api")
    protected.Use(middleware.AuthMiddleware())
    {
        protected.GET("/implants", controllers.GetUserImplants)
        protected.POST("/generate-implant", controllers.GenerateImplant)
        protected.POST("/send-command", controllers.SendCommand)
        protected.GET("/implants/:implant_id/commands", controllers.DashboardGetCommandsForImplant)
        protected.DELETE("/implants/:implant_id", controllers.DeleteImplant)
        protected.POST("/implants/:implant_id/download-configured", controllers.DownloadConfiguredImplant)
        protected.GET("/implants/:implant_id/screenshots", controllers.GetScreenshotsForImplant)
    }
    return r
}

func CheckinImplant(c *gin.Context) {
    var payload struct {
        ImplantID string `json:"implant_id"`
        PWD       string `json:"pwd"`
    }
    if err := c.ShouldBindJSON(&payload); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid payload: " + err.Error()})
        return
    }

    c.JSON(http.StatusOK, gin.H{"status": "checked_in", "message": "Implant check-in successful"})
}
```

Other important controllers include `SendCommand`, which queues a new command for an implant by creating a database record with a "pending" status. The `HandleCommandResult` controller updates the corresponding command record with the received output and sets its status to "executed"; it includes special logic for saving screenshot data to files if the output indicates a screenshot. Implants use the `ImplantClientFetchCommands` controller to retrieve their pending commands. The `DownloadConfiguredImplant` controller is critical for implant delivery, as detailed in the next section.

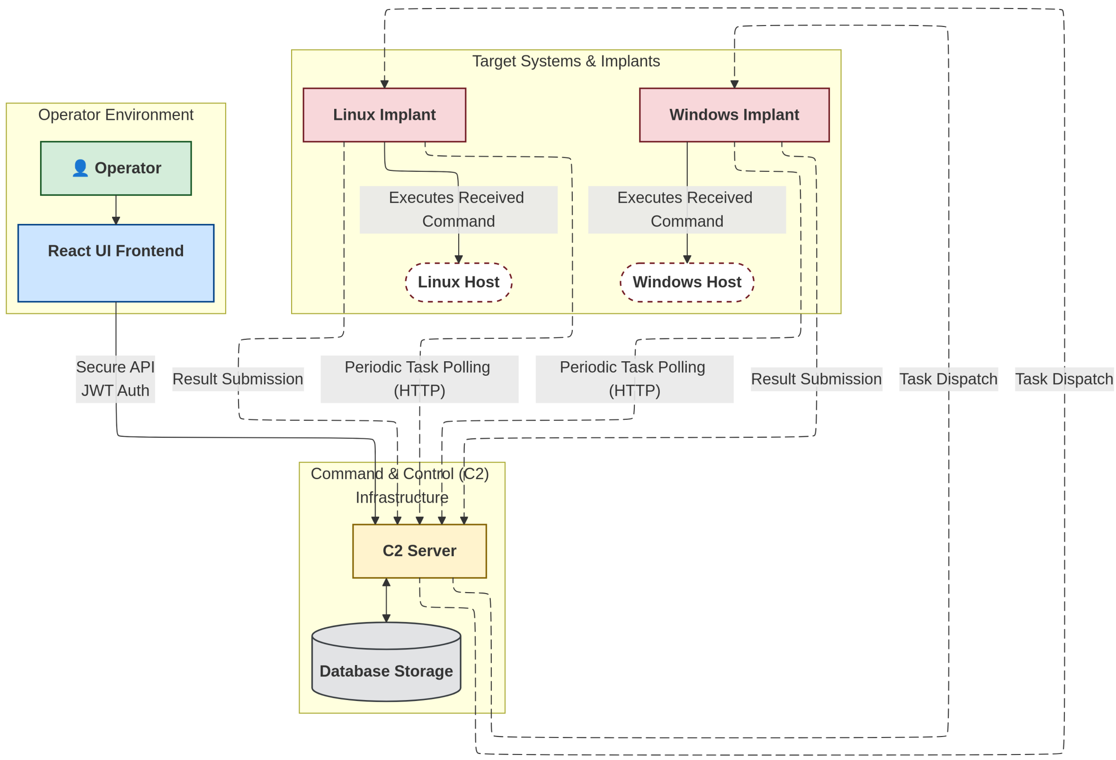


Figure 4.2: Process diagram of the C2 server workflow.

4.1.3 Implant Generation and Configuration

The C2 server facilitates the generation of customized implant binaries by patching pre-compiled base executables, a process that avoids the need for a Go compiler on the server. The implant's Go source code is compiled with embedded placeholder files using the `go:embed` directive. These placeholders, such as `c2_address.txt` and `placeholder_txt`, contain predefined, padded strings like `C2_IP_PLACEHOLDER_-STRING...` and a placeholder UUID, respectively. When an operator requests a new implant, the `GenerateImplant` controller creates a database record with a new unique ID. When the operator chooses to download this implant, they provide the C2 server's accessible IP address. The server then reads the appropriate base binary into memory and performs an in-memory patching operation. It uses `bytes.LastIndex` to locate the byte sequences for the placeholders and overwrites them with the new unique ID and the provided C2 address. The modified binary is then streamed to the operator with a filename such as `implant_[unique_token]_-windows.exe`. This on-demand patching ensures each implant is uniquely configured without requiring on-the-fly compilation.

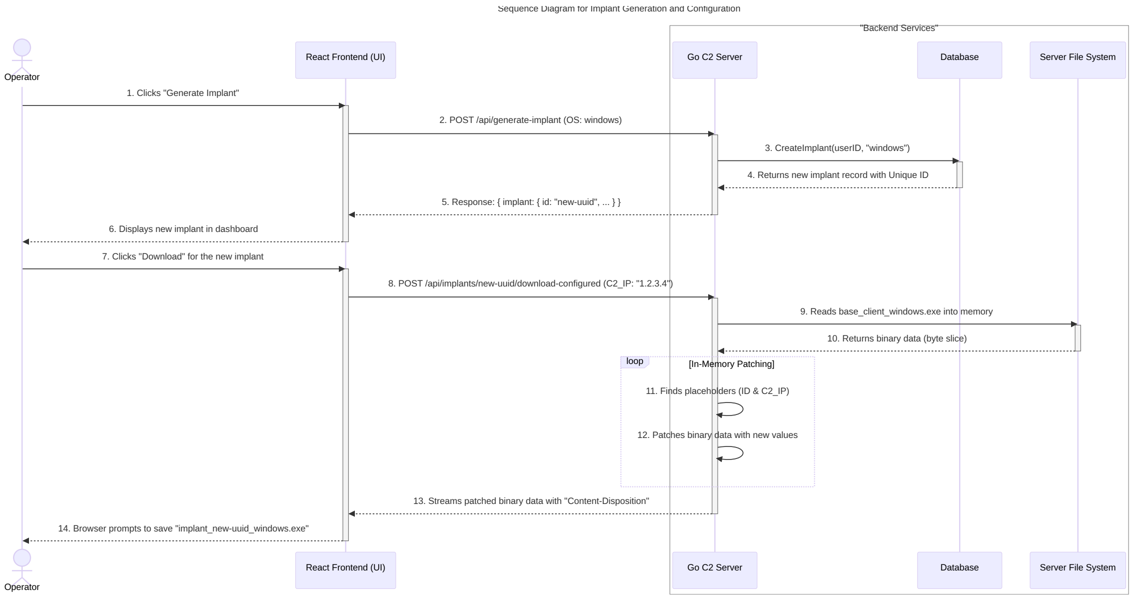


Figure 4.3: Generate Workflow.

4.1.4 Managing Implant State and Commands

The C2 server relies on a relational database, managed via GORM, to persist all implant-related data. Information for each implant is stored in a table mapped to the `models.Implant` struct, including fields such as a `unique_token` for identification, `user_id` to associate it with an operator, ‘status‘ to track its operational state (e.g., "new", "online", "offline"), `target_os`, `last_seen` timestamp, `ip_address`, and a ‘deployed‘ flag. The implant’s status is updated dynamically during check-ins, command fetches, result submissions, or by a periodic server-side task.

Commands and their results are managed in a separate table mapped to the `models.Command` struct. This table stores the `implant_id`, the ‘command‘ string itself, a ‘status‘ field indicating its lifecycle state (e.g., "pending", "executed"), and an ‘output‘ field for the result. When an operator issues a command, a new record is inserted with a "pending" status. Implants poll the `/implant-client/:unique_token/commands` endpoint to retrieve these commands. After execution, the implant sends the results to the `/command-result` endpoint, and the server updates the command’s record to "executed". Livestream frames sent to the `/livestream-frame` endpoint are saved directly to the C2 server’s filesystem in a structured path like `c2_screenshots/[implant_token]/livestream_frame_-[timestamp].png`. The Gin framework inherently handles concurrent requests from multiple operators and implants in separate goroutines, while the database connection pool configured with GORM efficiently manages concurrent database operations.

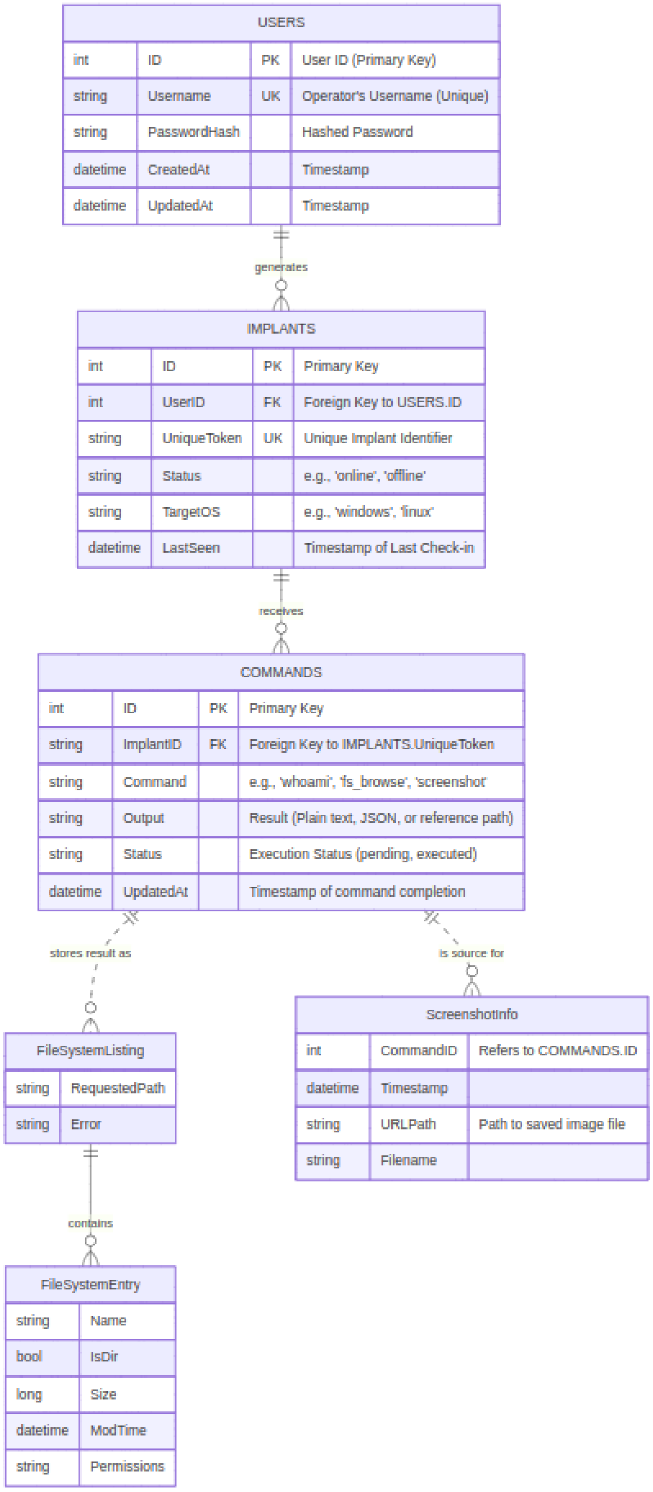


Figure 4.4: Structure of the data.

4.1.5 Daemonization Techniques

To operate stealthily, implants employ platform-specific techniques to run as background processes. On Windows, the `relaunchDaemonWindows` function in `relaunch_windows_go` handles this process. It first identifies the path of the currently running launcher, then copies this executable to a new, innocuously named file within the user's temporary directory, such as `audiosrvhost_[timestamp]_[random].exe`. An `exec.Command` is then prepared to run this new copy, setting environment variables like `IMPLANT_IS_BACKGROUND_XYZ123=1` to mark it as the daemonized instance and `IMPLANT_ORIG_LAUNCHER_PATH_XYZ789` to pass the original launcher's path. The command's `SysProcAttr` is configured with flags like `CREATE_NO_WINDOW` and `DETACHED_PROCESS` to ensure it runs without a console and is detached from the parent. The parent launcher then starts the new process, detaches from it, and exits, leaving the backgrounded copy to continue C2 operations.

On Linux, daemonization is achieved through a similar but distinct method in the `linuxRelaunchAsDaemon` function defined in `platform_linux_go`. This routine copies the launcher to the `/tmp` directory with a random name. When constructing the `exec.Command` to run this copy, `argv[0]` is overridden to spoof the process name to resemble a legitimate system process like `kthreadd`. Similar environment variables are set to mark the process and store the original path. Critically, the `SysProcAttr` is configured with `Setsid: true`, which creates a new session and fully detaches the child process from the terminal. Once the new process starts, the temporary binary in `/tmp` is immediately removed, causing the implant to run from an unlinked file descriptor, which enhances its "fileless" runtime footprint. To further obscure its presence, the implant also invokes the `prctl(PR_SET_NAME, ...)` system call to modify its kernel-level process name.

4.1.6 Self-Deletion Mechanisms

To minimize forensic artifacts, implants are designed to delete both the initial launcher and their own running executable upon receiving a `self_destruct` command. The self-deletion process on Windows, orchestrated by the `doSelfDeleteWindows` function in `exec_attrs_windows_go`, uses two methods. To delete the original launcher, it generates and executes a temporary, hidden batch script containing commands to forcefully delete the specified path. To delete the currently running implant, it uses the Windows API `SetFileInformationByHandle` to mark its own executable for deletion by the OS as soon as the implant process exits.

On Linux, the `linuxScheduleSelfDeleteGrandchild` function in `platform_linux_go` handles deletion. It constructs and executes detached shell commands, such as `sleep 1 && rm -f "[path]"`, for both the original launcher and the current executable. These commands are run in new sessions to ensure they outlive the parent implant process. Upon receiving a `self_destruct` command, the implant exits, allowing these scheduled `rm` commands to complete their task.

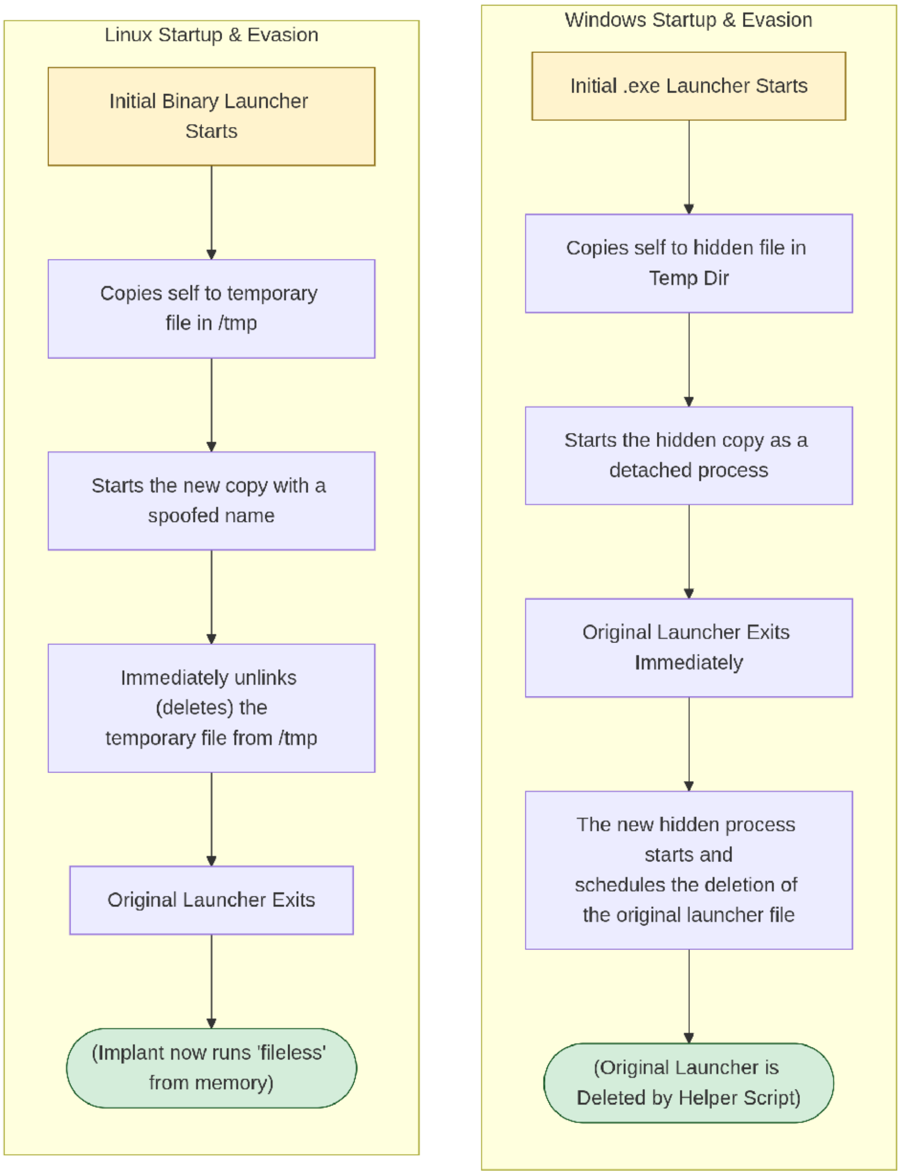


Figure 4.5: Deletion Flow.

4.1.7 Screenshot and Livestreaming

Implants can capture screenshots of the host's desktop and stream them to the C2 server. This capability is implemented using a platform-specific function pointer, `takeScreenshot`. On Windows, it utilizes the `github.com/kbinani/screenshot` package to capture the display, which is then encoded as a PNG and Base64 encoded. On Linux, the `linuxTakeScreenshot` function in `screenshot_linux_go` serially attempts to use external utilities like `grim` or `maim`, piping the output or reading from a temporary file before Base64 encoding. The livestreaming feature is initiated by a `livestream_start` command, which launches a goroutine that captures a screenshot at a regular interval (e.g., 1 FPS) and sends each frame to the C2's `/livestream-frame` endpoint until a `livestream_stop` command is received.

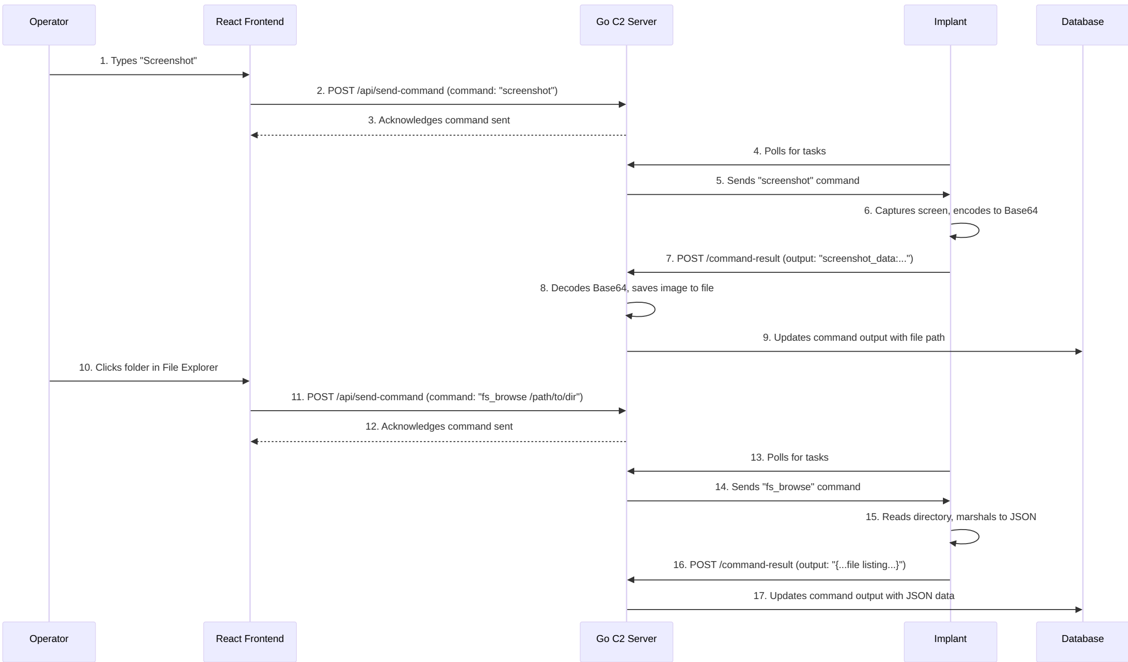


Figure 4.6: Screen View Workflow.

4.1.8 File System Interaction

The implant provides capabilities to browse the remote file system and download files, primarily handled by functions in `fs_utils_go`. The `listDirectory` function reads a directory's contents, gathers metadata for each item, and sends a JSON-marshaled listing to the C2. A `listRoots` function provides a starting point by enumerating drives on Windows or listing the root directory on Linux. File downloads are triggered by an `fs_download` command; the implant reads the specified file, Base64 encodes its content with a prefix of `file_data_b64:`, and sends it to the C2 as the command's output. The implant also supports the ‘cd‘ command by using ‘os.Chdir‘ to change its working directory.

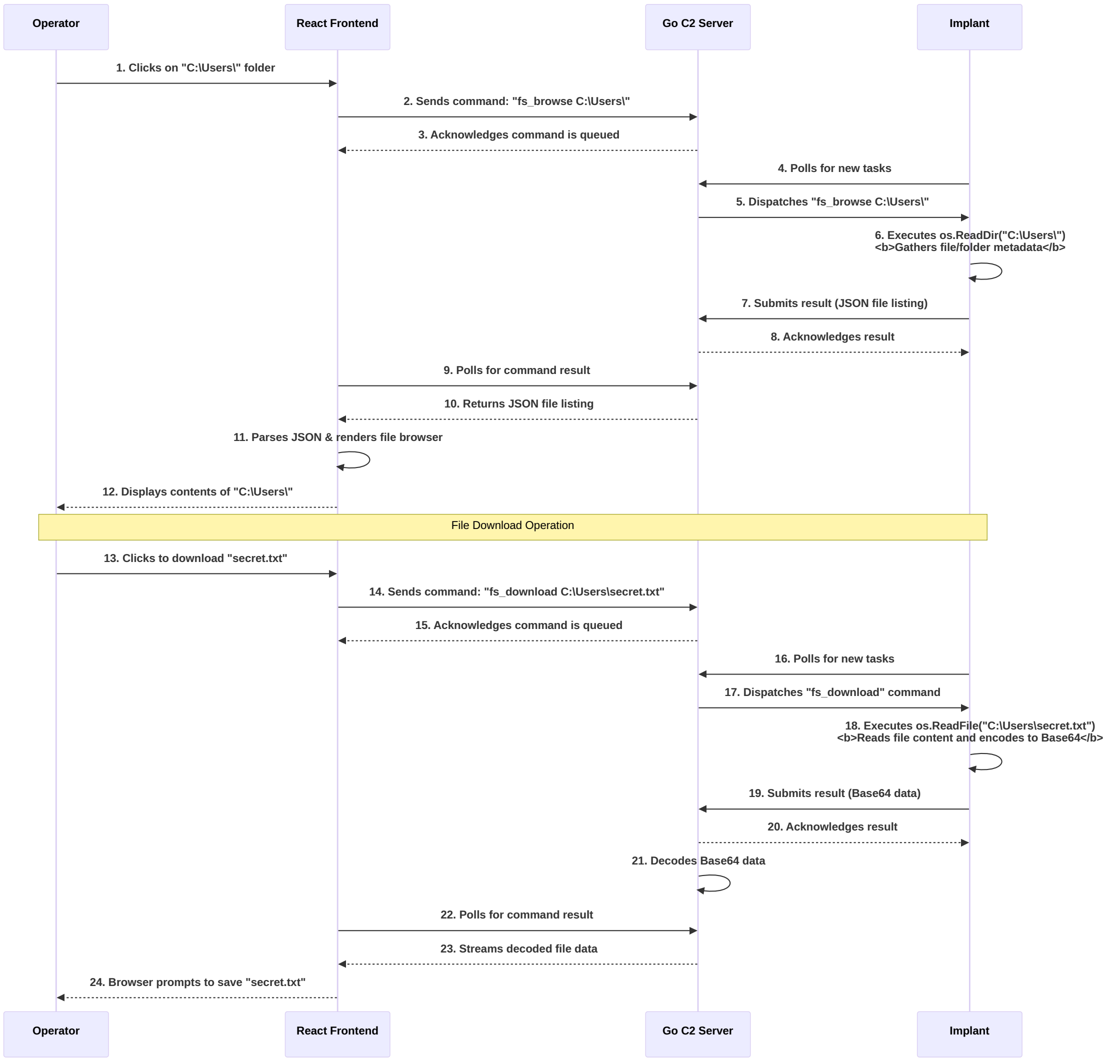


Figure 4.7: Browse Workflow.

4.2 Web Client Implementation (React)

The web client is a Single Page Application (SPA) built with React, providing the operator with an interface to interact with the C2 server and manage implants.

4.2.1 Project Structure

The React application's `src` directory is organized to separate concerns. The main entry point, `index.js`, renders the root `App.js` component, which is responsible for setting up client-side routing and the top-level layout. A central `components/` directory houses a collection of reusable UI components and page-level views. Key components include `AuthForm.js` for user authentication, the main `Dashboard.js` operational view, an interactive `Terminal.js`, a `FileSystemExplorer.js`, and a `ScreenshotViewer.js`. Various modal dialogs for specific tasks, such as

GenerateImplantOSModal.js and DownloadOptionsModal.js, are also defined as components. API calls to the C2 backend are made using the browser's 'fetch' API directly within these components. The application's state is managed primarily through local component state using React hooks like useState and useRef, with data persistence across sessions handled by localStorage for the JWT token. Styling is managed through standard CSS files, with class names suggesting the adoption of a utility-first CSS framework like Tailwind CSS.

4.2.2 Key Components

The React frontend is composed of several key components that enable operator interaction. The central hub for operators is Dashboard.js, which fetches and displays a list of implants from the C2 server, showing details like status and IP address. From this dashboard, operators can initiate the generation of new implants, download configured binaries, delete implants, and launch interactive tools for a selected target. The Terminal.js component provides an interactive pseudo-terminal interface, allowing an operator to send commands to an implant and view the historical output. It periodically fetches new command results from the server to keep the display updated. Similarly, the FileSystemExplorer.js component renders a file and directory listing received from an implant after an fs_browse command, enabling directory navigation and file download requests. For visual data, ScreenshotViewer.js is a modal component responsible for displaying images, either as individual captures in a gallery or as a real-time desktop stream. Authentication is handled by AuthForm.js, which manages user registration and login by making requests to the C2 server and storing the received JWT in localStorage. Finally, a series of specialized modal components, such as GenerateImplantOSModal.js and DownloadOptionsModal.js, provide focused user interfaces for specific actions like selecting a target OS or specifying the C2 server address for a download.

```
//Get Path for the screenshots
const extractScreenshotPath = (output) => {
    if (typeof output !== 'string') return null;

    const c2PathMatch = output.match(/Screenshot saved to C2 server at:
    ↪  (c2_screenshots\/[a-zA-Z0-9_-]+\/[a-zA-Z0-9_.-]+\.png)/);
    if (c2PathMatch && c2PathMatch[1]) {
        return c2PathMatch[1];
    }
    return null;
};

//Terminal exported
export default function Terminal({ implantID, onClose, openScreenshotViewer }) {
```

```
  const [logs, setLogs] = useState([]);
  const [input, setInput] = useState("");
  const [loading, setLoading] = useState(false);
  const polling = useRef(true);
  const containerRef = useRef(null);

  useEffect(() => {
    if (containerRef.current) {
      containerRef.current.scrollTop = containerRef.current.scrollHeight;
    }
  }, [logs]);

  useEffect(() => {
    polling.current = true;

    async function fetchLogs() {
      if (!polling.current) return;
      try {
        const token = localStorage.getItem("token");
        const res = await fetch(
          `${API_BASE}/implants/${implantID}/commands`,
          { headers: { Authorization: `Bearer ${token}` } }
        );
        if (!res.ok) {
          console.error(`Error fetching logs: ${res.status} ${res.statusText}`);
          if (res.status === 404) polling.current = false; // Stop if implant gone
          return;
        }
        const { commands } = await res.json();

        setLogs(prevLogs => {
            const serverCommands = commands.map(cmd => ({
                id: cmd.id,
                command: cmd.command,
                output: cmd.output || (cmd.status === 'pending' ? "waiting for
                ↪  output" : "<no output yet>"),
                status: cmd.status,
                isScreenshot: cmd.command === 'screenshot' &&
                ↪  extractScreenshotPath(cmd.output) !== null,
                screenshotPath: cmd.command === 'screenshot' ?
                ↪  extractScreenshotPath(cmd.output) : null,
            }));

            const persistentOptimisticErrors = prevLogs.filter(log =>
                typeof log.id === 'string' && log.id.startsWith('optimistic-') &&
                ↪  log.status === 'error' &&
                !serverCommands.some(sc => sc.command === log.command)
            );

           const combinedLogs = [...serverCommands, ...persistentOptimisticErrors];

            return combinedLogs.sort((a, b) => {
                const aIsNum = typeof a.id === 'number';
                const bIsNum = typeof b.id === 'number';
                if (aIsNum && bIsNum) return a.id - b.id;
                if (aIsNum) return -1;
                if (bIsNum) return 1;
```

```
              return String(a.id).localeCompare(String(b.id));
          });
        });

      } catch (err) {
        console.error("Terminal fetch error:", err);
      }
    }
    fetchLogs();
   const intervalId = setInterval(fetchLogs, 2500); // Poll slightly more frequently
    return () => {
      polling.current = false;
      clearInterval(intervalId);
    };
  }, [implantID]);

  const sendCommand = async () => {
    if (!input.trim()) return;
    setLoading(true);
    const optimisticId = `optimistic-${Date.now()}`;
    const commandToSend = input;

    setLogs(prev => [
      ...prev,
      { id: optimisticId, command: commandToSend, output: "sending command", status: "pending", isScreenshot: commandToSend.toLowerCase() === 'screenshot' },
    ]);
    setInput("");

    try {
      const token = localStorage.getItem("token");
      const res = await fetch(`${API_BASE}/send-command`, {
        method: "POST",
        headers: { "Content-Type": "application/json", Authorization: `Bearer ${token}` },
        body: JSON.stringify({ implant_id: implantID, command: commandToSend }),
      });
      if (!res.ok) {
        const errorData = await res.json().catch(() => ({ error: "Failed to send command" }));
        const errorMessage = `Error: ${errorData.error || res.statusText}`;
        setLogs(prevLogs => prevLogs.map(log =>
            log.id === optimisticId ? {...log, output: errorMessage, status: "error"} : log
        ));
      }
      // Poller will update with actual result
    } catch (err) {
      const errorMessage = `Error: ${err.message || "Network error"}`;
      setLogs(prevLogs => prevLogs.map(log =>
          log.id === optimisticId ? {...log, output: errorMessage, status: "error"} : log
      ));
    } finally {
      setLoading(false);
    }
  };
```

```
  const handleClose = () => {
    polling.current = false;
    onClose();
  };

  const displayLogs = logs.map(log => {
    let outputDisplay = log.output;
    if (log.isScreenshot && log.screenshotPath) {
        outputDisplay = (
            <button
                onClick={() => openScreenshotViewer(implantID, log.screenshotPath)}
                className="text-blue-400 hover:text-blue-300 underline
                ↪  hover:no-underline transition-all"
            >
                View Screenshot: {log.screenshotPath.split('/').pop()}
  snipet ...
```

4.2.3 State Management and API Interaction

State management within the React application primarily employs component-level state using React's built-in hooks. The `useState` hook is used extensively for managing local data such as input values, lists of items, and loading indicators. `useEffect` manages side effects, including initial data fetching and the setup of polling intervals for real-time updates. The `useRef` hook is utilized for accessing underlying DOM elements and for storing mutable values that do not trigger re-renders. For sharing state across components, the application relies on prop drilling, where callbacks and state values are passed down from parent to child components, and `localStorage` for persisting the JWT authentication token across browser sessions. Communication with the C2 server's RESTful API is managed using the browser's native 'fetch' API. These API calls are encapsulated within asynchronous functions and are typically triggered by user interactions or `useEffect` hooks. For requests to protected endpoints, the JWT is retrieved from `localStorage` and included in the 'Authorization' header. To achieve real-time updates for features like the terminal and livestream, the frontend employs a polling mechanism implemented with 'setInterval', which is carefully managed to prevent memory leaks and is paused when the browser tab is inactive to conserve resources. Error handling is managed with 'try...catch' blocks and by checking HTTP response statuses, with user feedback provided through a centralized notification system. Loading states are controlled by boolean flags that disable UI elements and display indicators while asynchronous operations are in progress.

4.3 Challenges Encountered and Solutions

Throughout the development of this C2 framework, several notable technical challenges were addressed to ensure functionality, stealth, and cross-platform compatibility. The solutions reflect a series of design trade-offs balancing simplicity against operational security.

A primary architectural challenge was to streamline implant deployment without requiring a Go compiler on the C2 server. The chosen solution of on-demand binary patching, while efficient, introduced a security trade-off. This method involves embedding and overwriting fixed-length placeholders in pre-compiled binaries. Although this simplifies generation, it stores the C2 configuration as plain text, making it susceptible to static analysis[1] if an implant is captured. The implementation required careful management of the patching logic to avoid corrupting the executable.

Ensuring consistent stealthy execution across Windows and Linux necessitated platform-specific implementations for daemonization. On Linux, this was achieved by having the implant re-execute itself from a temporary, unlinked file with a spoofed process name (`kthreadd`), a common "fileless" technique[2]. On Windows, the implant relies on specific Win32 API flags (`CREATE_NO_WINDOW` and `DETACHED_PROCESS`) to run as a hidden, fully detached process.

Achieving reliable self-deletion also required distinct solutions due to OS file-locking mechanisms[3]. On Windows, the implant marks its own executable for deletion upon process termination via a Win32 API call and uses a detached batch script to remove the original launcher. On Linux, the implant spawns a detached shell process that executes a delayed `rm` command, giving the main process time to exit before its file is deleted.

On the frontend, a key challenge was managing real-time UI updates for features like the terminal and livestream without degrading performance. This was solved through careful state management with React hooks and a polling mechanism[4] implemented with `setInterval`, which is paused when the browser tab is inactive to conserve resources. Finally, managing the storage of screenshot and livestream files on the C2 server required careful implementation of file and directory creation with proper permissions and error handling to ensure reliable retrieval.

[1] Static analysis is the process of examining a program's code without executing it, often to find vulnerabilities or, in this case, to extract embedded configuration data like IP addresses.

[2] A "fileless" technique refers to malware that operates primarily in a computer's memory (RAM) rather than writing its main components to the disk, making it harder for traditional antivirus software to detect and analyze.

[3] File locking is an operating system mechanism that prevents a running program's executable file from being modified or deleted while it is in use.

[4] Polling is a technique where a client repeatedly sends requests to a server at regular intervals to check for new information, simulating a real-time connection.

4.4 Deployment

The entire C2 framework is made to be deployed with Docker technology in order to guarantee portability and simplicity of setup. This method uses Docker Compose to orchestrate the database, web client, and C2 server into separate, straightforward services. In addition to offering a dependable way to replicate the operational environment for research and teaching, this approach abstracts away underlying host system dependencies. The backend, frontend, and database are the three main services that make up the deployment architecture. **Database Service:** A PostgreSQL database running in a container based on the `postgres:15-alpine` image forms the basis of the stack. All data persistence is handled by this service. A named Docker volume (`pgdata`) is mounted to the containers data directory (`/var/lib/postgresql/data`) to guarantee that data is preserved when the container is stopped or restarted. **Backend Service:** A multi-stage `Dockerfile` is used to containerize the Go-based C2 server. The application is compiled into a single, statically linked binary by the `builder`, the initial stage, using a `golang:1.23-alpine` image. A minimal and repeatable build environment is guaranteed by this procedure. Starting from a minimal `alpine:latest` base, the second stage copies only the compiled binary from the builder to produce the final, lightweight runtime image. This significantly lowers the attack surface and final image size. Using environment variables, the backend container is set up to connect to the `db` service via the private Docker network and expose port `8080`. Most significantly, the server can perform on-demand patching without having the ability to alter the original template files because the local `./binaries` directory, which contains the base implant templates, is mounted into the container as a read-only volume. **Frontend Service:** Additionally, a multi-stage `Dockerfile` is used to deploy the React web client. The React code is transpiled into a collection of static HTML, CSS, and JavaScript files during the `build` stage, which installs dependencies and runs the `npm run build` command using a `node:18-alpine` image. A lightweight `nginx:stable-alpine` image is used in the last step to serve these static assets. In addition to serving as the clients web server, Nginx is set up as a **reverse proxy**. This is an important architectural decision: Nginx routes all API requests from the web client (for example, to `/api/...`) to the backend service at its internal network address (`http://backend:8080`). This makes configuration easier and fixes cross-origin problems by separating the frontend from the backends precise location and port.

Chapter 5

Experimental Setup and Results

This chapter describes the environment used for testing the C2 framework, the test scenarios executed, and the results obtained. The goal is to validate the functionality and assess the basic characteristics of the system.

5.1 Testing Environment

5.1.1 Hardware and Software

The experiments were conducted using a unified setup where the C2 server and operator's machine were hosted on the same device, a Dell XPS 15 9530 named `paul-XPS-15-9530`. This machine is equipped with a 13th Gen Intel Core i7-13700H processor, 32.0 GiB of memory, a 1.0 TB disk, and NVIDIA/Mesa Intel graphics. The host operating system was Ubuntu 22.04.5 LTS running the Linux 6.8.0-60-generic kernel with an X11 windowing system. The Go compiler version used for the project was 1.21.x, and the React-based user interface was accessed through Google Chrome.

The target systems consisted of two distinct environments. The first was a physical Windows laptop, a `DESKTOP-DISE9AP`, featuring an AMD Ryzen 7 4800H processor and 16.0 GB of RAM. It ran a 64-bit installation of Windows 11 Home, version 23H2 (OS Build 22631.5335), with the default real-time protection of Windows Defender enabled. The second target was a Kali Linux virtual machine (Version 2024.1), configured with 8 vCPUs, 8GB of RAM, and an 80GB disk. This VM ran with default security settings and had no additional antivirus software installed. The virtualization was managed by VMware Workstation 17.2.

All systems, including the C2 server host, the Windows laptop, and the Linux VM, were connected to the same local area network (LAN) through a standard home router. This configuration allowed the implants to communicate directly with the C2 server's local IP address and designated listening port.

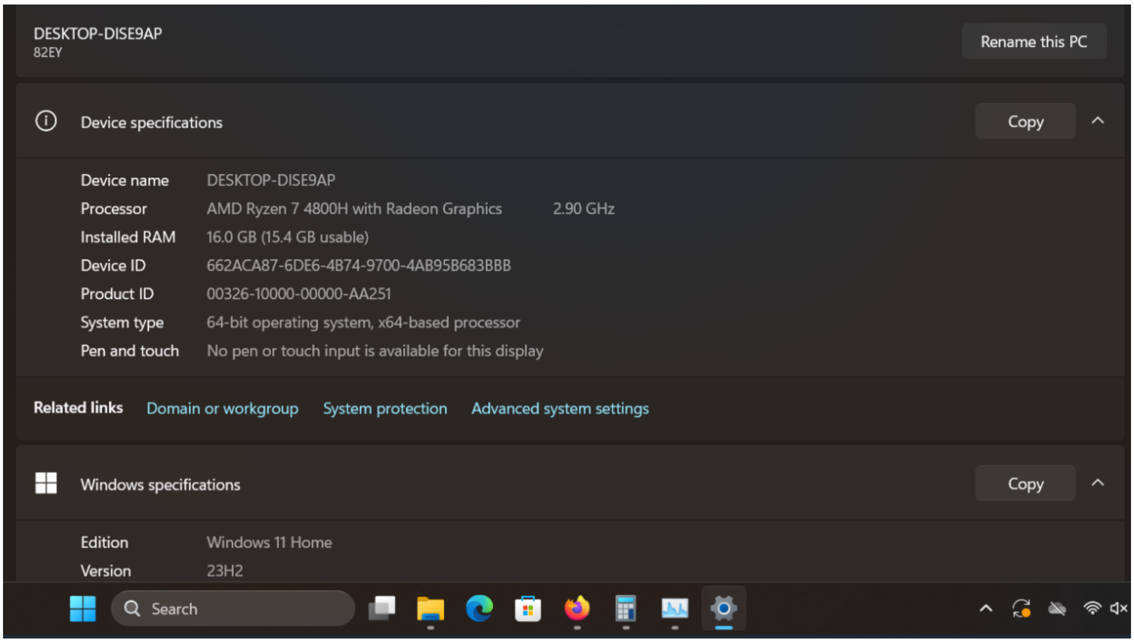


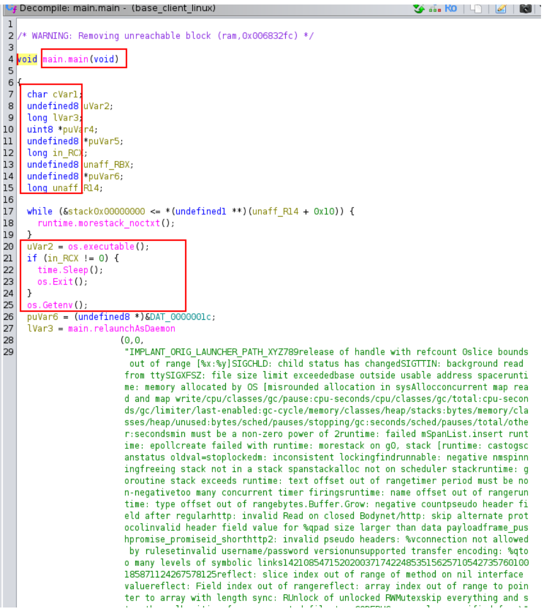
Figure 5.1: Windows laptop environment

5.2 Test Scenarios and Methodology

The framework's fundamental functionalities were assessed through a series of staged scenarios where all directives were transmitted to the implants via the React web interface. A key aspect of the implant generation process involved optimizing the binaries for both size and resistance to analysis. This was achieved by using the Go compiler's `-ldflags="-s -w"` argument, which strips the symbol tables and DWARF debugging information from the final executables.

The impact of this optimization was significant. Without stripping, the baseline binaries measured 8.1MB for Linux and 9.0MB for Windows. After applying the build flags, the Linux implant (`base_client_linux`) was reduced to 5.6MB, and the Windows implant (`base_client_windows.exe`) was reduced to 6.1MB. This resulted in an average size reduction of over 20%, but more importantly, the elimination of this structural and symbolic metadata significantly complicates reverse engineering efforts. To visually demonstrate this, a comparative analysis of the stripped and unstripped binaries will be presented using Ghidra[1].

[1] A free and open-source software reverse engineering framework developed by the U.S. National Security Agency (NSA). It offers a suite of powerful tools, including a decompiler, for analyzing compiled code in the absence of source code [EN20].



(a) Analysis of the unstripped binary.

(b) Analysis of the stripped counterpart.

Figure 5.2: Ghidra comparison of an unstripped (left) and stripped (right) binary, demonstrating the complete loss of function names and variable type information after stripping.

5.2.1 Scenario 1: Implant Deployment and Basic Check-in

The initial test aimed to verify the successful generation, deployment, daemonization, and initial communication the implants with the Centralised server. The process began with generating a Windows implant executable, `base_client_windows.exe`, from the React UI. This binary was then transferred to the Windows laptop and executed. We observed the process creation using Task Manager, expecting the original launcher process to terminate and a renamed, hidden process (e.g., `audiosrvhost_[timestamp]_[random].exe`) to appear. On the C2 server's web UI, we monitored for the new implant to register in the active list and confirmed that its displayed current working directory (PWD) and source IP address were correct. This entire sequence was repeated for the Linux implant on the Kali VM, where we used `ps aux` to observe the daemonized process, which was expected to masquerade under a name like `[kthreadd]`. The expected outcome was for both implants to run as background processes, establish a connection with the C2 server, and accurately report their initial status in the UI, with the initial launcher processes self-terminating as designed.

5.2.2 Scenario 2: Remote Command Execution and File System Interaction

This scenario tested the framework's ability to execute arbitrary shell commands, manage the remote file system, and exfiltrate files. On both the Windows and Linux targets, commands were issued through the C2 UI's terminal. We started by running commands like `whoami` (or `id` on Linux) to verify the output matched the expected user context. Subsequently, a `cd` command was issued to change the working directory (e.g., `cd C:\Windows\Temp`), and we confirmed the PWD update in the C2 UI. The file system browsing capability was tested by issuing an `fs_browse` command, ensuring the file and folder listings were correctly rendered. To test file download functionality, a test file was created on the implant using a shell command (`echo "test content" > testfile.txt`), followed by an `fs_download` command for that file.It was anticipated that all commands would run successfully, that the user interface would receive accurate output, and that file downloads would be completed with data integrity preserved.

5.2.3 Scenario 3: Screenshot and Livestream Functionality

The objective here was to validate the implant's screen capture and desktop livestreaming capabilities while monitoring its resource consumption. For both Windows and Linux targets, the `screenshot` command was issued, and we verified that the captured image appeared correctly in the C2 web UI. Next, the `livestream_start` command was sent. During the livestream, the implant process's CPU usage on the target machine was monitored. We observed the livestream viewer in the React UI to confirm that frames were updating at approximately 1 frame per second (FPS). To ensure the stream was live, we performed actions on the target's desktop, such as moving the mouse or opening a window, and verified these changes were reflected in the stream. Finally, the `livestream_stop` command was issued to cease the stream. The expected outcome was successful single-shot captures and a functional livestream that started and stopped on command, reflected desktop activity accurately, and imposed minimal CPU overhead.

5.2.4 Scenario 4: Evasion and Self-Destruction

This final scenario was designed to assess the effectiveness of the implant's basic evasion techniques and its self-deletion mechanism. On the Windows target, during the initial deployment, we used Task Manager to confirm the implant ran under a masqueraded name without a visible console window. We also monitored Windows Defender to see if its default settings would trigger any alerts. For the Linux target, we used commands like `ps aux` to verify that the process name was spoofed as intended (e.g., `[kthreadd]`). We also checked the symbolic link at `/proc/[PID]/exe` to confirm it pointed to a deleted path, a common technique for fileless-in-memory execution. The primary test was issuing the `self_destruct` command from the C2 UI to both implants. Afterward, we verified on both target systems that the running implant process had terminated and that its executable filesboth the running instance and the original launcherhad been deleted from the disk. The expected outcome was that the implants would run with their intended stealth characteristics and that the self-destruct function would completely remove their presence from the file system.

5.3 Results and Observations

5.3.1 Results for Scenario 1 Implant Deployment and Check-in

The deployment tests were successful on both systems. The Windows implant is a 6.1MB executable program designated as `base_client_windows.exe`, upon execution on the `DESKTOP-DISE9AP` laptop, copied itself to a path within the user's Temp directory, such as `C:\Users\[UserName]\AppData\Local\Temp\audiosrvhost_-[timestamp]_[random].exe`, and launched as a hidden process. The original launcher terminated as expected. The new implant appeared in the C2 UI within its 5-second check-in interval, correctly displaying the laptop's PWD and local IP address. Similarly, the 5.4MB Linux implant, `base_client_linux`, executed on the Kali VM, copied itself to a path like `/tmp/implant_[randomhexstring]` and successfully masqueraded its process name as `[kthreadd]` in the process list. The corresponding entry at `/proc/[PID]/exe` correctly pointed to a deleted file path. It also checked into the C2 UI within the 5-second interval, providing the correct PWD and IP address.

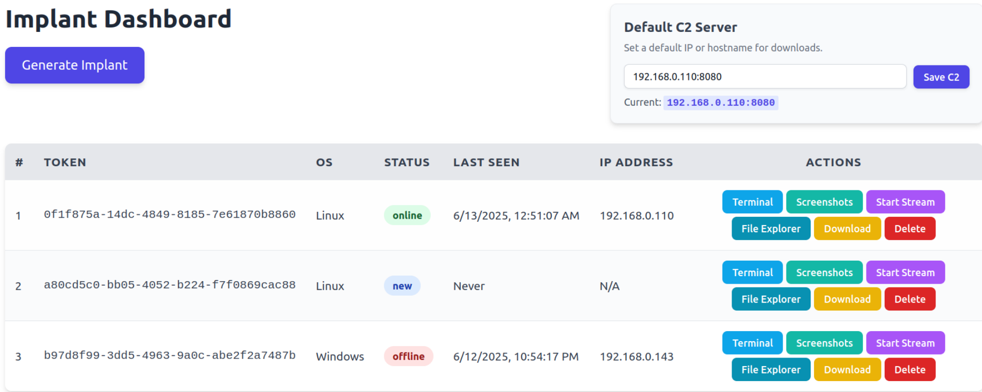


Figure 5.3: C2 Web Interface showing active Windows and Linux implants.

5.3.2 Results for Scenario 2 Remote Command Execution and File System Interaction

All evaluated shell commands, including `whoami`, `id`, `ipconfig`, and `echo`, were executed successfully on both Windows and Linux systems, with their output accurately shown in the C2 UI's terminal interface. The `cd` command consistently altered the implant's working directory, and this modification was promptly evident in ensuing UI state updates. Moreover, the `fs_browse` and `fs_download` commands operated flawlessly, facilitating accurate directory listings and successful file acquisition. The perceived latency for command execution was mostly determined by the implant's 5-second polling period. Upon retrieval of a job by the implant, the execution and result transfer via the local network were noted to be exceedingly swift, generally accomplished in less than 100 milliseconds for straightforward commands.

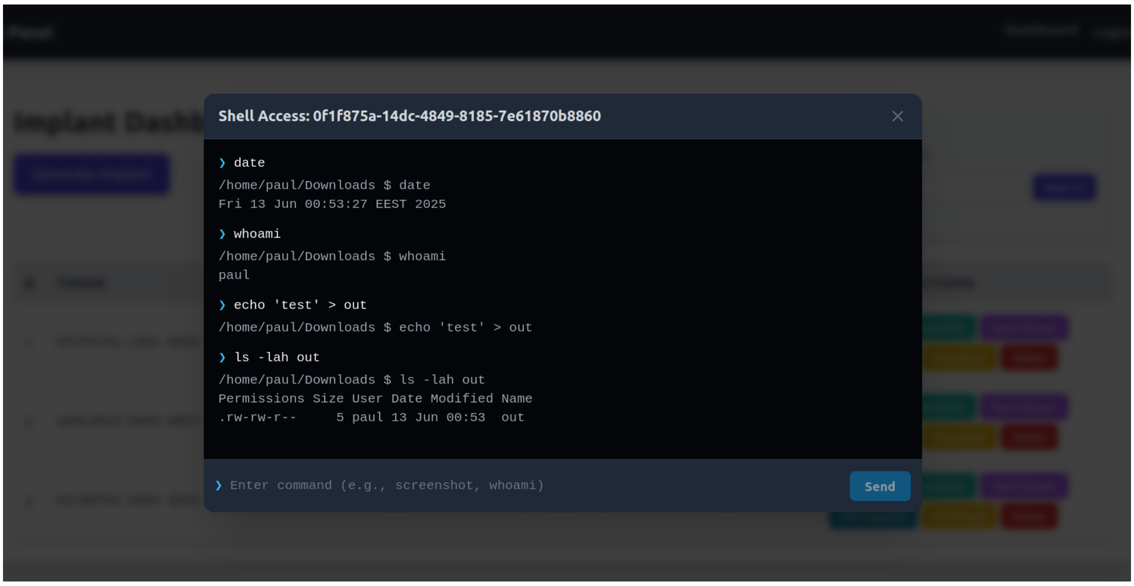


Figure 5.4: Interactive terminal in the C2 Web UI

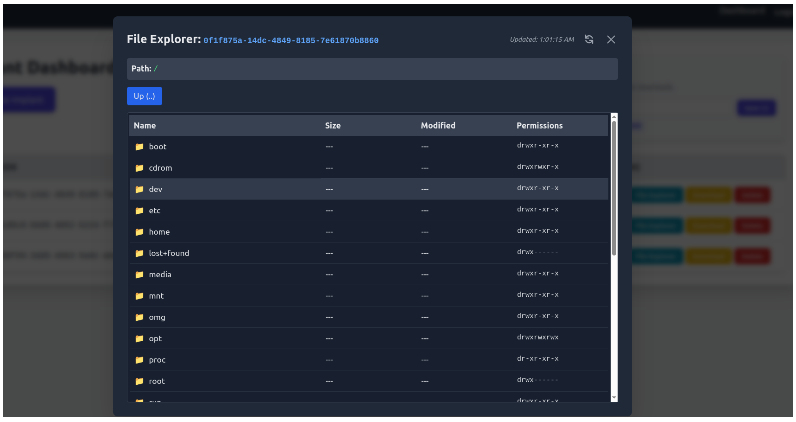


Figure 5.5: Files listed from an infected linux system

5.3.3 Results for Scenario 3 Screenshot and Livestream Functionality

The screenshot and streaming features operated as intended. The `screenshot` command effectively recorded desktop photos on both Windows and Linux, which were accurately shown in the C2 UI. The initiation of a livestream using `livestream_-start` was executed properly. Throughout active streaming, the CPU overhead of the implant procedure on both target PCs constantly maintained below 2%. The feed refreshed in the C2 UI at a consistent rate of roughly 1 FPS. The `livestream_-stop` command successfully halted the stream upon execution.

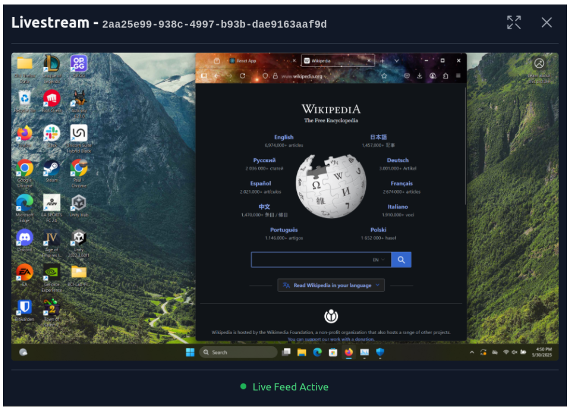


Figure 5.6: Livestream from infected machine

5.3.4 Results for Scenario 4 Evasion and Self-Destruction

The basic evasion and self-destruction mechanisms proved effective in the test environment. On Windows, the implant process ran hidden without a console window under its masqueraded name (`audiosrvhost_...exe`). Notably, Windows Defender, with its default real-time protection enabled on the Windows 11 Home system, did not generate any alerts during the implant's execution or its basic C2 operations. On the Kali Linux VM, the process name was successfully spoofed as `[kthreadd]`, and the on-disk executable was unlinked after the daemonized copy started. The `self_destruct` command worked as intended on both operating systems, resulting in the successful deletion of the associated executable files, including both the running instance and the original launcher binary.

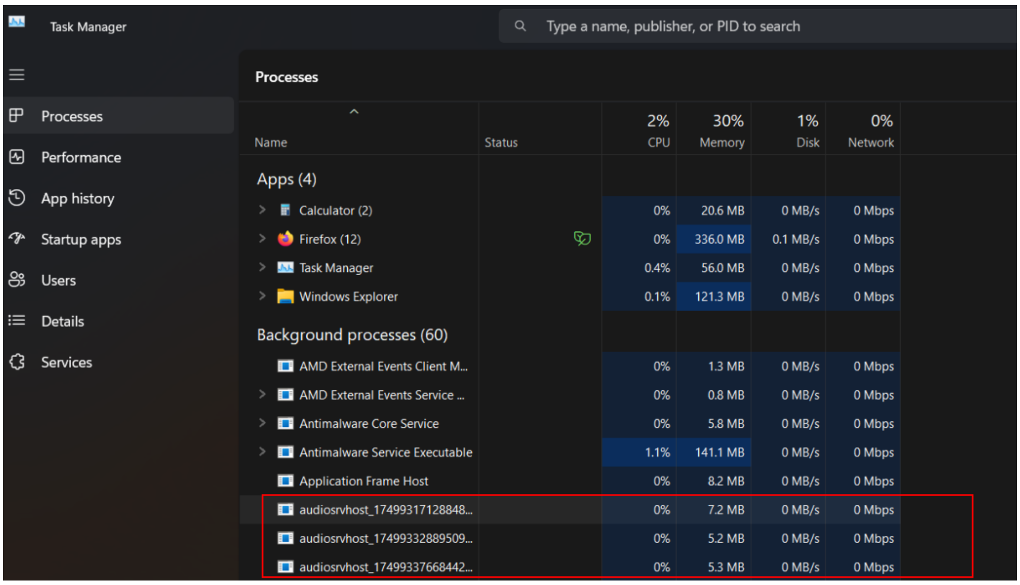


Figure 5.7: Windows Process name spoofed

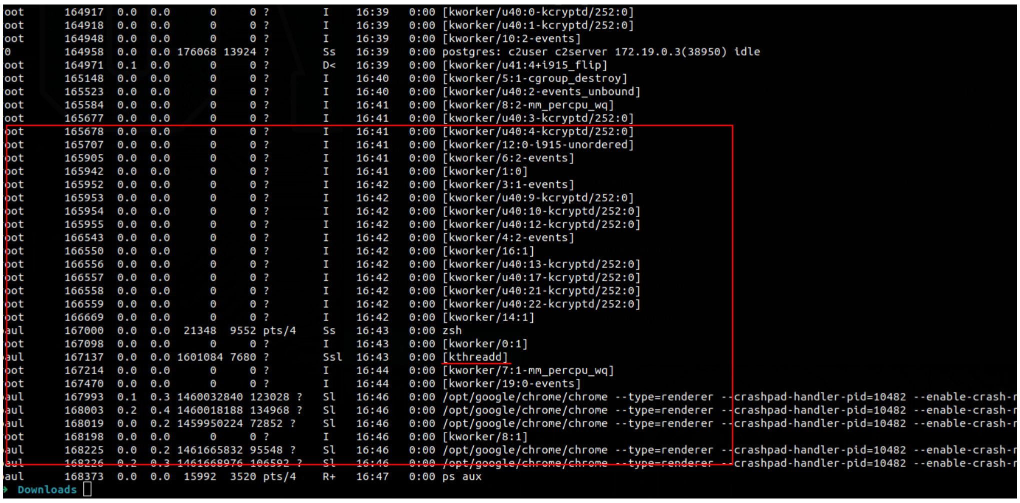


Figure 5.8: Linux Process name spoofed

Chapter 6

Conclusions

6.1 Summary of Findings and Contributions

The Command-and-Control (C2) framework developed in this project successfully demonstrated all of its essential characteristics through a series of structured test scenarios. The primary objectives of this research were accomplished: implants were successfully deployed and daemonized on both Windows and Linux computers, enabling stable communication with the C2 server. The core functions, including remote command execution, file system interaction, screenshot capture, and desktop livestreaming, functioned as expected. Performance benchmarks proved beneficial, with no resource impact on target systems, with CPU overhead remaining under 2% even during processor-intensive tasks like screen streaming. The operator's perception of end-to-end latency for command execution was primarily influenced by the implant's 5-second polling interval; upon task retrieval, local execution and result transmission were consistently rapid, typically under 100 milliseconds.

Moreover, the fundamental evasion techniquesprocess backgrounding, process name masquerade, and self-deletionoperated as designed within the controlled testing environment. Under the precise conditions of this test, the Go-based implant did not elicit quick detection by the default setup of Windows Defender on a fully updated Windows 11 Home PC. This outcome highlights the persistent difficulty for signature-based and fundamental heuristic defenses in detecting custom, statically coded malware. The principal contribution of this thesis is the successful development of a functional, cross-platform C2 framework that provides a beneficial and transparent platform for cybersecurity education and threat emulation research.

6.2 Limitations and Security Implications

Although the framework met its objectives, it is essential to contextualize its capabilities and recognize its limitations. The evasion strategies employed are rudimentary and are likely inadequate to circumvent more advanced Endpoint Detection and Response (EDR) technologies. Contemporary EDRs utilize sophisticated behavioral analysis, memory scanning, and API hooking, which are likely to identify the implant's existing techniques for process injection or command execution. The effective circumvention of default Windows Defender underscores a deficiency in consumer-grade protection but should not be construed as a sign of evasion against enterprise-level security frameworks.

The system possesses operational security shortcomings that must be rectified for practical red team engagements. The technique of embedding the C2 server's address inside the implant binary, albeit effective, retains this configuration in plaintext, rendering it readily accessible via static analysis. The implant's dependence on a fixed-interval HTTP polling technique for command and control communication generates a predictable network signature that may be detected and obstructed by sophisticated network monitoring tools. The binary sizes (5.6MB for Linux, 6.1MB for Windows), although standard for Go applications, may serve as a possible marker for signature-based detection if not additionally obfuscated.

6.3 Future Work and Research Directions

The framework developed in this thesis serves as a robust foundation for numerous avenues of future research and enhancement. The modular design of the implants and the C2 server allows for the integration of more advanced techniques, pushing the boundaries of both offensive capability and defensive analysis.

6.3.1 Advanced Evasion and Defense Bypass on Windows

A primary area for future research is the development of more sophisticated evasion techniques tailored to the Windows operating system, where EDR solutions are most prevalent. This research could focus on:

- **In-Memory Evasion and Sleep Obfuscation:** To mitigate EDRs that intermittently examine process memory, the implant could be augmented with sleep obfuscation. This entails encrypting the implant's code and data segments in memory prior to entering a sleep state and decrypting them upon reactivation. This method would markedly diminish the probability of detection by memory forensics and live analysis instruments.

- **DLL Hijacking:** Future implants may be engineered to leverage vulnerabilities associated with DLL (Dynamic-Link Library) hijacking. This entails locating a valid, signed application that insecurely loads a DLL from an unregulated route, and thereafter positioning the implant, rebranded as that DLL, in the correct spot. This would enable the trusted program to load and run the implant, facilitating a robust means of obfuscation and persistence.

6.3.2 Automated Persistence Mechanisms

The current framework relies on manual execution for persistence. Future work could automate this process by integrating platform-specific persistence modules.

- **For Windows:** The implant could be programmed to automatically establish persistence by creating a scheduled task, modifying common registry keys such as `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`, or by placing a shortcut in the user's startup folder.

- **For Linux:** Persistence could be achieved by automatically adding an entry to the user's or system's crontab for periodic execution. A more advanced method would involve modifying the user's SSH configuration by adding the C2 operator's public key to the `~/.ssh/authorized_keys` file, providing a stealthy and persistent remote access channel. Further research could even explore the creation of malicious PAM (Pluggable Authentication Modules) backdoors to intercept credentials or provide authenticated access.

6.3.3 Enhancing C2 Communications and Infrastructure

The C2 communication protocol could be substantially improved to augment the framework's stealth and durability. Future endeavors may concentrate on the implementation of additional covert channels, including DNS-over-HTTPS (DoH) or domain fronting, to conceal the actual destination of the C2 traffic. Moreover, the creation of entirely malleable C2 profiles, such to those in `Cobalt Strike`, would permit operators to tailor every facet of the implant's network indicators, facilitating its seamless integration with legal traffic from prevalent applications. This thesis has effectively illustrated the design and execution of an educationally-oriented C2 framework. Although useful and successful presently, its true worth is in its potential as a transparent and extendable platform. The proposed future research directions delineate a definitive trajectory, enabling this project to persist as a significant resource for both the academic and cybersecurity sectors in the continuous endeavor to comprehend and counteract sophisticated digital threats.

Bibliography

[AMCH17] Ali Alshamrani, Sashi Myneni, Anil Chowdhary, and Dijiang Huang. A survey on command and control systems for botnets. *IEEE Communications Surveys & Tutorials*, 19(4):2794–2825, 2017.

[Amn25] Amnesty International and Forbidden Stories. The pegasus project. https://www.amnesty.org/en/latest/news/2021/07/the-pegasus-project/, 2025. Accessed: 2025-05-10.

[Bis25] Bishop Fox. Sliver c2 framework. https://github.com/BishopFox/sliver, 2025. Accessed: 2025-02-28.

[BP20] Alex Banks and Eve Porcello. *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, 2nd edition, 2020.

[cob25] cobbr. Covenant - a .net c2 framework for red teaming. https://github.com/cobbr/Covenant, 2025. Accessed: 2025-02-15.

[Cro25] CrowdStrike. What is endpoint detection and response (edr)? https://www.crowdstrike.com/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/, 2025. Accessed: 2025-05-15.

[Cyb23] Cybereason. Sliver c2 leveraged by many threat actors. https://www.cybereason.com/blog/sliver-c2-leveraged-by-many-threat-actors, 2023. Accessed: 2025-05-15.

[DK15] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 2015.

[EN20] Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide*. No Starch Press, 2020.

[FIR25] FIRST.org, Inc. Common vulnerability scoring system version 3.1 calculator. https://www.first.org/cvss/calculator/3.1, 2025. Accessed: 2025-03-20.

[For25] Fortra. Cobalt strike: Adversary simulation and red team operations. https://www.cobaltstrike.com/, 2025. Accessed: 2025-01-22.

[Hav25] HavocFramework. The havoc framework. https://github.com/HavocFramework/Havoc, 2025. Accessed: 2025-03-08.

[its25] its-a-feature. Mythic c2 framework. https://github.com/its-a-feature/Mythic, 2025. Accessed: 2025-01-11.

[Kas23] Kaspersky Global Research & Analysis Team. Operation triangulation: The last hardware mystery. https://securelist.com/operation-triangulation-the-last-hardware-mystery/111669/, 2023. Accessed: 2025-05-12.

[Man24] Mandiant. M-trends 2024. Technical report, Mandiant, Google Cloud, April 2024.

[Met25] Meta and React Community. React documentation. https://react.dev/, 2025. Accessed: 2025-01-30.

[Nat25] National Institute of Standards and Technology. Nist cybersecurity framework (csf) 2.0. https://www.nist.gov/cyberframework, 2025. Accessed: 2025-02-01.

[Net23] Netskope Threat Labs. Effective c2 beaconing detection. https://www.netskope.com/netskope-threat-labs/effective-c2-beaconing-detection, 2023. Accessed: 2025-05-15.

[Per21] Nicole Perlroth. *This Is How They Tell Me the World Ends: The Cyberweapons Arms Race*. Bloomsbury Publishing, 2021.

[Rap25] Rapid7. Metasploit penetration testing software. https://www.metasploit.com/, 2025. Accessed: 2025-04-19.

[SH12] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[The25a] The MITRE Corporation. Command and Control - Enterprise | MITRE ATT&CK®. https://attack.mitre.org/tactics/TA0011/, 2025. Accessed: 2025-04-05.

[The25b] The MITRE Corporation. Common vulnerabilities and exposures (cve) program. https://www.cve.org/, 2025. Accessed: 2025-04-10.

[The25c] The MITRE Corporation. Defense Evasion - Enterprise | MITRE ATT&CK®. https://attack.mitre.org/tactics/TA0005/, 2025. Accessed: 2025-05-14.

[The25d] The MITRE Corporation. MITRE ATT&CK Framework. https://attack.mitre.org/, 2025. Accessed: 2025-03-14.

[Vir25] VirusTotal. Virustotal: Analyze suspicious files, domains, ips and urls to detect malware and other breaches. https://www.virustotal.com/, 2025. Accessed: 2025-05-15.

[WSS17] Johannes Walter, Dimitri Samosseiko, and Siniša Slijepčević. Command and control (c&c) server detection with machine learning. *Procedia Computer Science*, 112:2267–2275, 2017.