Article

Not peer-reviewed version

C3: Leveraging the Native Messaging API for Covert Command and Control

Efstratios Chatzoglou * and Georgios Kambourakis

Posted Date: 28 February 2025

doi: 10.20944/preprints202502.2263.v1

Keywords: Command and Control; C2; Browser; Post-exploitation; EDR; Lateral Movement; Evasion; Privilege escalation; Network Security

Article

C3: Leveraging the Native Messaging API for Covert Command and Control

Efstratios Chatzoglou [1,2,*] and Georgios Kambourakis [1]

[1] Department of Information and Communication Systems Engineering, University of the Aegean, 83200 Karlovasi, Greece
[2] LRQA, B37 7ES 1 Trinity Park Bickenhill Ln, Birmingham, United Kingdom
* Correspondence: efchatzoglou@aegean.gr (E.C.)

Abstract: Traditional Command and Control (C2) frameworks struggle with evasion, automation, and resilience against modern detection techniques. This paper introduces Covert C2 (C3), a novel C2 framework designed to enhance operational security and minimize detection. C3 employs a decentralized architecture, enabling independent victim communication with the C2 server for covert persistence. Its adaptable design supports diverse post-exploitation and lateral movement techniques for optimized results across various environments. Through optimized performance and the use of the Native Messaging API, C3 agents achieve a demonstrably low detection rate against prevalent Endpoint Detection and Response (EDR) solutions. A proof-of-concept implementation demonstrates C3's effectiveness in real-world adversarial simulations, specifically in direct code execution for privilege escalation and lateral movement. Our findings indicate that integrating novel techniques, such as the Native Messaging API, and a decentralized architecture significantly improves the stealth, efficiency, and reliability of offensive operations. The paper further analyzes C3's post-exploitation behavior, explores relevant defense strategies, and compares it with existing C2 solutions, offering practical insights for enhancing network security.

Keywords: Command and Control; C2; browser; post-exploitation; EDR; lateral movement; evasion; privilege escalation; network security

1. Introduction

Command and Control (C2) servers are essential in network administration, cybersecurity operations, and adversarial engagements [1]. From remote system administration and ethical hacking (Red Team operations) to, also, cybercriminal activities, C2 servers have a varied history. As cyber threats evolve, so must the infrastructure supporting security professionals in defending networks and developing resilient defenses [2].

C2 servers are crucial for remote command execution, data exfiltration, system monitoring, and network penetration testing. Traditional C2 frameworks, however, struggle with scalability, particularly in the face of modern cloud and hybrid infrastructures. Their centralized architecture creates a single point of failure, making them vulnerable to takedowns and forensic analysis. Furthermore, their common implementations and lack of scalability make them easily detectable by Endpoint Detection and Response (EDR) solutions. Finally, most C2 frameworks rely on well-known and often outdated post-exploitation techniques, limiting their adaptability.

This work introduces a new C2 server designed to overcome these limitations. It incorporates modern data exfiltration, persistence, and command execution techniques, along with a decentralized architecture, to improve scalability, stealth, and Operational Security (OpSec). This system, named C3 (Covert C2), provides a robust alternative to legacy C2 frameworks for ethical cybersecurity professionals [3]. Overall, C3 [4], integrates advanced scalability and command execution features, high evasion rates, adaptability, and a decentralized architecture for resilient operation against modern defensive countermeasures.

 doi:10.20944/preprints202502.2263.v1

In more detail, while existing well-known C2 frameworks like Cobalt Strike [5], Mythic [6], and Sliver [7] offer valuable capabilities but often lack stealth, automation, and adaptability, C3 distinguishes itself through:

- Enhanced stealth: C3 employs uncommon techniques for direct, undetected system command execution.
- Decentralized architecture: Peer-to-peer (P2P) communication eliminates single points of failure.
- Optimized performance: C3 minimizes resource consumption while maintaining high-speed command execution and data transfer.
- Increased adaptability and expandability: Using infrequent techniques and direct command execution, C3 adapts to new environments, offers diverse post-exploitation and persistence methods, and enables flexible expansion.
- Out-of-the-box Persistence: Persistence is quite crucial in a red team engagement. As a result, C3 offers out-of-the-box persistence to the post-exploitation attack.

The rest of the paper is structured as follows. Section 2 describes the threat model. Section 3 overviews the proposed scheme. Section 4 details the testbed and scenarios. Section 5 evaluates C3's detectability against popular EDRs. Section 6 analyzes C3's post-exploitation behavior and defenses. Section 7 compares C3 with existing C2 frameworks. Section 8 discusses related work. Finally, the paper concludes with future directions.

2. Threat Model

This work considers an outsider attacker aiming to compromise a target machine within a local network protected by a typical standard firewall (e.g., the default MS Windows firewall, characterized by stateful packet filtering and allowing outbound traffic) and common antivirus software. The attacker's goal is to exfiltrate data and execute commands on the target system, thereby violating the confidentiality and integrity properties of the Confidentiality, Integrity and Availability (CIA) triad. While availability may also be a target through credential or access manipulation, this is not the primary focus of this threat model.

In more detail, the below assumptions are considered:

- Outsider attacker: The attacker operates from outside the target network and does not have prior authorized access.
- Target environment: The target machine resides within a local network protected by a standard firewall and antivirus.
- Initial access: We assume the attacker has already gained initial access to the target network. The specific methods used for network penetration (e.g., phishing, social engineering, exploiting vulnerabilities in network services) are outside the scope of this threat model, which rather focuses on the post-exploitation phase. It is acknowledged that initial access is crucial, and different methods can impact the post-exploitation phase.
- Client-side attack: The attack vector targets the client machine within the network.
- Attack Objective: The attacker aims to exfiltrate data and execute arbitrary commands on the compromised machine, and perform lateral movement.
- Communication channel: The attacker leverages the generally open outbound HTTP/3 (QUIC) [8] traffic from the target machine's web browser as a covert communication channel. This protocol is leveraged due to the challenges associated with inspecting its encrypted traffic.
- Tools and techniques: The attacker utilizes a malicious browser extension and a corresponding native application installed on the target machine. Communication between the attacker's C2 server and the target is facilitated through the browser extension and native application via the Native Messaging API. Reflective DLL loading is used to execute malicious code within the native application's memory space.

 doi:10.20944/preprints202502.2263.v1

Building upon the assumptions outlined, this threat model does not consider: insider threats, attacks directed at network infrastructure (such as firewall compromise or router exploits), the specific methodologies employed for initial network penetration, or Denial-of-Service (DoS) attacks designed to disrupt availability. Specifically, the model addresses the post-exploitation phase, focusing on the techniques used for data exfiltration and command execution after the attacker has gained initial access to the internal network.

3. Scheme Overview

After gaining initial access to the target machine, data exfiltration and command execution are paramount. We exploit the generally unrestricted nature of outbound HTTP(S) traffic by leveraging a web browser as a covert channel. Our novel approach utilizes a browser extension with native messaging API capabilities, enabling the attacker's C2 server to communicate through the extension to a native application proxy. This proxy downloads and executes a DLL from a URL provided by the C2 server. This DLL handles command execution and data exfiltration, gaining access to the Operating System (OS) and establishing persistence.

We exploit the popularity of Chromium-based browsers (Chrome and MSEdge) for our data exfiltration and command execution strategy. As already mentioned, our approach relies on the Native Messaging API, which facilitates communication between browser extensions and native applications. A browser extension and associated native application are installed on the target system. The attacker's C2 server communicates with the extension over HTTP/3 (QUIC), relaying messages to the native application. The native application downloads DLLs hosted on the C2 server and uses reflective DLL loading to execute them in memory. This provides a mechanism for dynamic command execution.

Figure 1 illustrates the communication flow between the attacker and the victim. A short description of each step shown in the figure follows.

1. The attacker's C2 server waits for the browser extension on the victim's machine to initiate a connection.
2. Upon connection, the C2 server sends a URL to the extension.
3. The extension forwards the URL to the native application.
4. The native application downloads the DLL from the URL via HTTP/3, loads it into memory, and executes it.
5. The native application sends the execution result to the extension.
6. The extension relays the result back to the C2 server.
7. The attacker views the execution result.

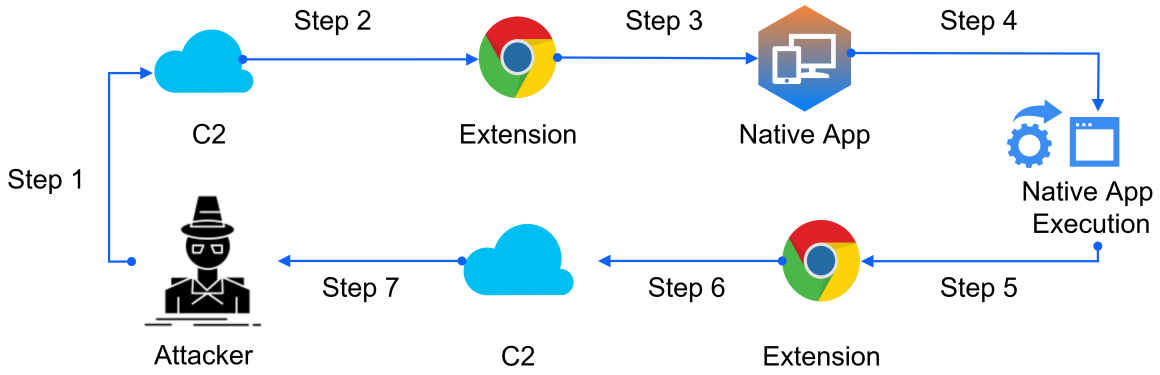

Figure 1. Attacker-victim communication flow.

4. Testbed and Test Scenarios

Our testbed consisted of a clean virtual machine (VM) running MS Windows 10 Enterprise (build 19045.5371), last updated in February 2025. The VM was allocated 8GB of RAM and a quad-core CPU.

 doi:10.20944/preprints202502.2263.v1

MSEdge (version 132.0.2957.140) and Chrome (version 132.0.6834.160) were installed on the VM. The native application and DLLs were implemented in C++, while the browser extension was developed using JavaScript. The C2 server, hosted on an Ubuntu Server 22.04 LTS Gen2 VM (kernel 6.8.0-1020-azure) within Azure cloud infrastructure, utilized PHP and Caddy v2.9.1. A free *azurewebsites* domain was used for C3 communication, with Caddy configured to enable HTTP/3 for secure communication with the target workstation.

The installation required several key files: *native_app.json* (for configuring communication between the native application, *native_messaging.exe*, and the extension), *manifest.json* (specifying extension permissions), *background.js* (containing the extension's code), and *libcurl.dll* (providing HTTP/3 support for the native application). The installation procedure then proceeded as follows:

1. Native application and library placement: Copy *native_messaging.exe*, *native_app.json*, and *libcurl.dll* to a user-accessible directory. The path to this directory must be specified in *native_app.json*.
2. Registry entry creation: Create a new registry entry named *com.example.native_messaging* under *HKEY_CURRENT_USER\SOFTWARE\Google\Chrome\NativeMessagingHosts\com.example.native_messaging*. Set the data of this entry to the path of *native_app.json*.
3. Extension installation (Chrome): Replace a pre-installed Chrome extension (e.g., ID *nmmhkkegccagdldgiimedpiccmgmieda*) with the files of the malicious extension (*background.js* and *manifest.json*). This step is necessary in Chrome to prevent the browser from deleting the extension files. Note that this step is not required for MSEdge, as it does not delete extension files.
4. Extension ID retrieval: Obtain the extension ID from the Secure Preferences file of either MSEdge or Chrome and add it to *native_app.json*.
5. Browser shortcut modification: Modify or create new shortcuts for Chrome or MSEdge to include the *–load-extension* flag, pointing to the malicious extension's directory. This allows the extension to be loaded when the browser is launched and ensures that Chrome will keep this extension, i.e., there is a small possibility for Chrome to delete the replaced extension if the *–load-extension* flag is not used. Basically, the optimal way to ensure the surviveness of the malicious extension is to use both the *–load-extension* flag and the replacement of an already installed extension. For enhanced persistence, this can be combined with the *–headless* and *–user-data-dir* flags and scheduled via Task Scheduler. Note that Chrome requires two initial executions with *–user-data-dir* for connection stability in the headless mode. MSEdge displays a "Turn off extensions in developer mode" popup. Therefore, headless mode is recommended for MSEdge, unless the extension is installed from the Microsoft Edge Add-ons store.
6. Browser launch and C2 communication: Launch the browser (either normally or in headless mode using the modified shortcut) to establish communication with the C2 server.

To validate the effectiveness of the proposed C2 communication method, six test cases were executed on the target host via the C2. These scenarios demonstrated both command execution and lateral movement [9].

1. Basic command execution: The whoami command was executed via CMD.
2. Network connectivity test: The *ping 8.8.8.8* command was executed via CMD.
3. Data Exfiltration: A file in the *Downloads* folder was converted to base64 and uploaded to the C2 server.
4. Scheduled task execution as another user: The *schtasks* command was used to schedule and execute a task as a different user. Due to the limitations of direct command execution without an interactive shell, the output of the task was redirected to a file, which was then retrieved. While this file-based approach impacts OpSec, it was necessary given the constraints of Covert C2's direct command execution and lack of shellcode support. Alternative solutions for avoiding file system interaction include blind command execution or using a C2 framework with shellcode capabilities.

 doi:10.20944/preprints202502.2263.v1

5. Lateral movement: The *net use* command was employed to establish a connection to another host via Server Message Block (SMB). Similar to the *schtasks* scenario, file redirection was used to capture the output. It is noted that enabling WinRM on target hosts would simplify lateral movement and remote command execution with output capture.
6. Random: The last scenario was employed to illustrate the effectiveness of the evasion with random command execution. This means that this scenario used all the previous scenarios in random order.

Overall, these scenarios were selected to illustrate the post-exploitation and lateral movement capabilities of the proposed communication method.

5. Evaluation

As previously detailed in Section 4, to evaluate the detectability of C3, we conducted tests across six specific scenarios –command execution (using *whoami*, *ping 8.8.8.8*, and a scheduled task via *schtasks*), file upload, and lateral movement (using *net use*) – as well as a random scenario, against four anonymized, widely-used EDR products. These EDR products were chosen due to the availability of free trial versions.

The evaluation aimed to determine whether and to what extent these attack techniques could be detected by commonly available EDR solutions. The specific product names of the EDRs are omitted, as the focus of this work is on evaluating the general detection capabilities of a range of EDR products against this technique, rather than the specific performance of individual products. However, the names of the EDR products tested can be made available upon request to the authors.

To evaluate the detectability of C3 by each EDR, we first installed the necessary components: the C3 framework (including the web server and workstation components) and then the EDR software on the target workstation. The evaluation involved initiating a connection from the Chrome browser to the C2 server every 10 sec. If a command was available, the communication proceeded as described in Figure 1. Otherwise, the C2 server responded with "No". Each of the six test scenarios was executed for five minutes with randomized command selection, i.e., either a specific scenario URL or a "No" response from the C2 server, for a total of 15 min per EDR. A final 10-min mixed scenario was then conducted, where commands were randomly selected from any of the five test cases (command execution or file upload).

Remarkably, the proposed scheme evaded detection by all tested EDR solutions across all six scenarios. This finding, based on the specific conditions of this test and the current EDR capabilities, suggests a significant vulnerability in a range of EDR products to this post-exploitation technique. The fact that zero detections were recorded, despite the absence of code obfuscation (excluding the necessary Base64 encoding for file uploads), further underscores the potential impact of this approach. The code and detailed instructions for this Proof-of-Concept (PoC) are publicly available on GitHub [4].

6. Analysis of Post-Exploitation Behavior and Defense Mechanisms

Given the results presented in Section 6, a thorough understanding of the C3's post-exploitation behavior is crucial. Therefore, we proceed with an analysis and offer suggestions for detecting this activity.

Several potential detection points emerge. First, the *–load-extension* flag is present in the browser's process execution, providing a readily identifiable characteristic. Second, communication between the extension and the native application consistently involves a command-prompt (cmd) process initiated as a subprocess of the Chrome browser. Third, this communication occurs in cleartext, enabling EDR solutions to inspect the exchanged messages. Fourth, command executions, such as ping 8.8.8.8, spawn additional child processes of the native application. Specifically, the process flow is: chrome→cmd→native_messaging→cmd→PING. This distinct process hierarchy offers a clear indicator of potentially malicious subprocesses originating from the Chrome browser.

 doi:10.20944/preprints202502.2263.v1

Consequently, to detect this post-exploitation activity, the YARA rule in Listing 1 can serve as a PoC. It is important to note that this rule is tailored to the specific implementation described herein and functions as a signature-based detection method. The rule searches for Chrome or MSEdge process executions that include a cmd process, an *.exe* process, and the *–load-extension* string. To utilize this YARA rule, the appropriate executable can be downloaded from the official VirusTotal YARA repository [10].

Listing 1. Detection of suspicious CMD execution initiated by a browser

```
function DETECT_BROWSER_CMD_EXECUTION(input_data)
    Define suspicious_strings:
        browser_chrome ← "chrome.exe" (nocase)
        browser_edge ← "msedge.exe" (nocase)
        cmd ← "cmd.exe" (nocase)
        exe ← ".exe" (nocase)
        suspicious_string ← "–load-extension" (nocase)
    if browser_chrome or browser_edge in input_data then
        if cmd in input_data then
            if exe in input_data then
                if suspicious_string in input_data then
                    return "Suspicious Execution Detected"
                end if
            end if
        end if
    end if
    return "No Suspicious Activity Detected"
end function
```

An additional defense strategy against this post-exploitation technique in Microsoft Windows-based enterprise environments involves leveraging Group Policy Objects (GPOs) within Active Directory. A specifically configured GPO can mitigate this attack by preventing the installation and execution of browser extensions. This involves first downloading the Windows-specific ADMX template for each target browser and adding it to the Group Policy Management Editor. Subsequently, the extension installation blocklist should be enabled and configured with the asterisk (*) wildcard value. This configuration blocks the execution of all extensions not explicitly included in the allowlist. Finally, the allowlist must be enabled and populated with the extension IDs of permitted extensions.

However, GPO policies are susceptible to circumvention under certain conditions. Misconfigurations, such as applying the policy at the User level instead of the machine level, can enable users to bypass restrictions by switching user profiles or employing alternative methods, including portable browsers or unmanaged software. Furthermore, an attacker could circumvent the policy by utilizing a portable browser that does not adhere to system-wide settings or by switching to a different installed browser not subject to the policy. For example, if the GPO is enforced for Chrome but not MSEdge, an attacker could simply switch browsers to evade the restriction.

Concerning shortcut files, enterprise-managed Windows workstations typically store shortcuts created after application installation in the C:\Users\Public\Desktop directory. By default, all authenticated users possess "Modify" (M) permissions within this location. Furthermore, shortcuts placed in C:\Users\%USERNAME%\Desktop grant the respective user full control.

While administrators can implement restrictions limiting user access to Read & Execute permissions for these shortcuts, this approach remains susceptible to circumvention. Critically, both tested browsers functioned successfully in headless mode. This capability, coupled with Task Scheduler, enables autonomous communication with a C2 server, eliminating the need for user interaction.

One approach to detecting the execution of malicious shortcut files involves creating an event viewer within Active Directory to monitor Task Scheduler activity in real time. An alternative method,

 doi:10.20944/preprints202502.2263.v1

detailed in the pseudocode presented in Listing 2, outlines a YARA rule designed to identify suspicious Task Scheduler entries that utilize shortcut files to execute Chrome or MSEdge.

Listing 2. Detection of scheduled tasks executing '.lnk' shortcuts that launch Chrome or Edge

```
Input: Task Scheduler XML files, Registry exports
Output: Detection of suspicious scheduled tasks
Initialize task_executes_lnk ← search for schtasks /Create /TN "*.lnk"
Initialize chrome_exec ← search for C:\.*\chrome.exe
Initialize edge_exec ← search for C:\.*\msedge.exe
Initialize lnk_exec ← search for C:\.*lnk
Initialize suspicious_flags ← search for
        -headless, -disable-web-security, -remote-debugging-port
if task_executes_lnk AND (chrome_exec OR edge_exec) then
    Report suspicious scheduled task
else if lnk_exec AND (chrome_exec OR edge_exec) then
    Report suspicious shortcut execution
else if suspicious_flags detected then
    Report potentially malicious browser execution
end if
End
```

To mitigate the risk of portable Chrome or MSEdge execution, the YARA rule in Listing 3 can be implemented for detection. This rule specifically targets Chrome and MSEdge executables located in common portable application paths.

Listing 3. Detect Portable Execution of Chrome or Edge

```
Input: Running processes, Task Scheduler, Registry, Startup Entries
Output: Detection of suspicious portable browser execution
Initialize chrome_portable_paths ← search for:
        C:\Users\.*\Downloads\chrome.exe
        C:\Users\.*\Desktop\chrome.exe
        C:\Users\.*\AppData\Local\Temp\chrome.exe
        C:\Users\.*\PortableApps\ChromePortable\chrome.exe
Initialize edge_portable_paths ← search for:
        C:\Users\.*\Downloads\msedge.exe
        C:\Users\.*\Desktop\msedge.exe
        C:\Users\.*\AppData\Local\Temp\msedge.exe
        C:\Users\.*\PortableApps\EdgePortable\msedge.exe
Initialize suspicious_flags ← search for:
        -no-sandbox, -disable-web-security, -headless, -remote-debugging-port
if chrome_portable_paths OR edge_portable_paths detected then
    Report execution of portable browser
else if suspicious_flags detected then
    Report potentially malicious browser execution
end if
End
```

Naturally, while these YARA rules are not infallible, they offer a degree of mitigation. Specifically, they may necessitate more sophisticated evasion techniques from attackers in certain scenarios.

Furthermore, AppLocker and Windows Defender Application Control (WDAC) can contribute to mitigating this attack. Both are native Windows security features designed to regulate application and script execution. They enable organizations to enforce application control policies, reduce attack surfaces, and mitigate malware. Specifically, AppLocker, available in Windows Enterprise and Education editions, empowers administrators to define allowed and disallowed applications and scripts. It supports application control through various rules and file types. WDAC offers a more advanced and

 doi:10.20944/preprints202502.2263.v1

contemporary approach to application control compared to AppLocker. Its enhanced security posture stems from enforcing strict application control policies at the kernel level, enabling pre-execution policy enforcement.

While both AppLocker and WDAC offer protection against the default implementation of this post-exploitation technique due to the unsigned nature of the native application executable, circumvention remains possible. If specific file types (e.g., .bat, .ps1, .dll, .vbs, .src, .hta, .wsf, .js, .msi, .msp, .inf, .lnk, .cpl, .scf, .xsl, .pptm, .dotm, .xlsm, .xml, .iso, .wim, .diagcab, .py) are not explicitly controlled, or if an attacker can induce a signed executable to launch the unsigned executable, the protections can be bypassed. Specifically, each value within the path parameter of the *native_app.json* file is executed via a start command within cmd due to the extension's use of the *chrome.runtime.connectNative* API. This allows an attacker to execute arbitrary script files (e.g., .bat, .vbs, .py, assuming the relevant interpreter is available) or even a signed executable (to which they may not have direct access) in conjunction with a DLL sideloading attack. In the latter scenario, the attacker would need to place the DLL within the appropriate directory of the signed executable. Consequently, a wide range of attack vectors exists, requiring only the ability to load the extension, replace the relevant extension files, add a registry entry, and save the *native_app.json* file to a publicly accessible location.

Regarding the relevant registry entry, administrators or SIEM tools can monitor the registry for suspicious entries. However, attackers can circumvent this by avoiding registry entries altogether and instead utilizing WebSockets or WebRTC for communication between the extension and the native application. While such communication can be secured, e.g., through a TLS authentication scheme, this approach necessitates prior execution of the native application to establish a listener.

Another vulnerability stems from the use of Task Scheduler for command execution or persistence. Administrators can mitigate this by restricting Task Scheduler access to authorized domain users or by actively logging and monitoring Task Scheduler activity for suspicious tasks.

In summary, the most effective defense against this attack involves using Group Policy Objects (GPOs) to restrict browser extension installation across all managed machines within the network. Implementing AppLocker or WDAC rules can further enhance protection against post-exploitation attacks in general.

7. Comparison with Existing C2 Frameworks

Command and Control frameworks are essential tools for both red team operations and malicious actors, enabling remote access, persistence, and post-exploitation activities. Among the most widely used C2 frameworks are Cobalt Strike, Sliver, Mythic, and Havoc. This section compares C3 with these prominent frameworks, highlighting its distinguishing advantages.

Cobalt Strike [5] is a widely recognized and extensively used commercial C2 framework, frequently observed in both red teaming exercises and real-world threat actor activity. Its comprehensive suite of tools facilitates reconnaissance, privilege escalation, and lateral movement. The Beacon payload, Cobalt Strike's core component, enables operators to execute commands, inject shellcode, perform keylogging, and interact with the file system. It supports diverse command execution methods, including PowerShell, WMI, and native Windows API calls. Cobalt Strike incorporates various obfuscation and evasion techniques, such as in-memory execution, payload encryption, and process injection, aimed at minimizing detection. Seamless integration with Metasploit allows leveraging exploits from a comprehensive offensive security toolkit. For post-exploitation, Cobalt Strike offers capabilities for persistence, credential harvesting, and lateral movement, including built-in tools for Kerberos ticket attacks (Pass-the-Ticket, Golden Ticket), password dumping via Mimikatz, and DCSync attacks. Robust pivoting techniques, including Secure Shell (SSH) tunneling and named pipe relays, contribute to its effectiveness for deep network penetration.

Sliver [7] is an open-source C2 framework developed as an alternative to Cobalt Strike, offering similar functionalities while emphasizing flexibility and modularity. Unlike Cobalt Strike, which primarily focuses on Windows environments, Sliver is cross-platform and supports Linux and macOS

 doi:10.20944/preprints202502.2263.v1

as well. Its core features include multiple payload types, such as staged and stageless implants, reflective DLL injection, and support for Golang-based payloads, which makes it harder to detect due to its lesser-known execution patterns. Sliver's main capabilities revolve around its use of encrypted communication channels, such as mTLS and WireGuard tunnels, to ensure secure command execution and data exfiltration. It also supports multiple transport mechanisms, including DNS, HTTP, and HTTPS, providing various evasion options. Post-exploitation capabilities in Sliver include standard techniques like process injection, shellcode execution, and credential dumping, but it does not natively support advanced Kerberos-based attacks like Cobalt Strike. However, its open-source nature allows users to extend its functionality and integrate third-party tools such as Mimikatz or BloodHound for deeper post-exploitation techniques.

Mythic [6] is a robust, open-source C2 framework designed for extensibility and modularity. Unlike Cobalt Strike and Sliver, Mythic is entirely open-source and employs an agent-based approach, where each implant (referred to as a "payload type") is a separate, independently developed, and customizable component. Key features include a highly customizable web-based interface, encrypted communications, and support for multiple payloads written in various languages, including Python, C#, and Golang. Mythic's capabilities are notable for their usability, offering a structured API for building custom agents, automating tasks, and integrating with external tools. It supports diverse communication protocols, such as HTTP, TCP, and WebSockets, adapting it to various operational requirements. Mythic provides a range of post-exploitation techniques for persistence, credential dumping, and process injection. While it does not natively support advanced Kerberos-based attacks like Cobalt Strike, its modular architecture allows operators to develop and integrate such functionalities as needed.

Havoc [11] is a relatively recent C2 framework developed as a modern alternative to Cobalt Strike and Sliver. It emphasizes stealth and evasion, employing highly obfuscated payloads designed to minimize detection by modern EDR solutions. Unlike many older frameworks that rely heavily on PowerShell, Havoc is built using modern programming languages and execution techniques, reducing its signature footprint. Its capabilities include process injection, shellcode execution, and in-memory payload execution, similar to other C2 frameworks, but with a focus on improved detection evasion. Havoc's post-exploitation capabilities include standard techniques such as credential dumping, process manipulation, and lateral movement. However, being a newer framework, its ecosystem is less mature than those of Cobalt Strike or Sliver, potentially requiring operators to develop additional modules for more complex post-exploitation scenarios.

Our proposal, C3 [4], provides built-in persistence, lateral movement, and evasion capabilities, along with customization options to facilitate adversary simulations. It can execute shellcode from other C2 frameworks and offers direct code execution on the OS. This enables the execution of arbitrary OS commands, such as *cmd*, without triggering detection alerts. Its web-based interface allows operation from any OS. Lateral movement and credential dumping leverage native OS commands (e.g., *net use*, *schtasks/runas*), minimizing suspicious indicators from EDRs. Each compromised host communicates directly with the C2 server, a unique implementation offering several advantages for red team engagements.

A comparison of these C2 frameworks reveals several key distinctions. Cobalt Strike remains a comprehensive and feature-rich commercial tool, particularly for advanced post-exploitation techniques such as Kerberos attacks and sophisticated lateral movement. However, its commercial nature and associated cost limit accessibility for independent researchers. Sliver and Mythic offer compelling open-source alternatives. Sliver excels in cross-platform support, while Mythic provides a highly customizable and modular approach. Havoc, a more recent framework, introduces novel approaches to stealth and evasion but currently lacks the extensive plugin and module ecosystem of established frameworks like Cobalt Strike and Sliver. C3's streamlined design and unique implementation, leveraging the Native Messaging API and reflective DLL loading, offer a distinct advantage for achieving stealth and resilience in post-exploitation scenarios, as demonstrated by its negligible detection rate

 doi:10.20944/preprints202502.2263.v1

against four popular EDR solutions in our tests. While Cobalt Strike offers a broader range of post-exploitation modules, C3 prioritizes direct command execution and a decentralized architecture, which provides significant operational security benefits. Furthermore, C3's open-source nature makes it readily accessible to all users.

Ultimately, the selection of a C2 framework is contingent upon the specific operational requirements and the target environment. Cobalt Strike remains a strong choice for advanced red team operations demanding extensive post-exploitation capabilities. Sliver and Mythic provide flexible, open-source alternatives, each with distinct strengths. Havoc offers promising stealth capabilities for evading modern defenses. However, C3 presents a compelling alternative, particularly when stealth, resilience, and ease of use are paramount. Its unique approach, leveraging the native messaging API and decentralized architecture, offers significant advantages in minimizing detection and maximizing operational security. While each framework possesses inherent strengths and weaknesses, effective operators often employ multiple C2 solutions in concert to optimize their attack surface and enhance evasion.

An additional crucial factor in evaluating C2 frameworks is their adaptability to modern defensive measures, including EDR solutions, antivirus engines, and network monitoring tools. Each framework employs distinct methods for evasion, persistence, and command execution, significantly influencing its effectiveness in real-world operations. In this context, despite its power and versatility, Cobalt Strike has become a prime target for cybersecurity defenses due to its widespread use by both red teams and threat actors. Consequently, many EDR and AV solutions incorporate signatures to detect its default Beacon payloads, necessitating increased reliance on custom malleable C2 profiles and advanced obfuscation techniques for successful evasion. While features like sleep obfuscation, indirect system calls, and process injection enhancements can improve stealth, the heightened scrutiny of Cobalt Strike traffic and techniques has contributed to the growing adoption of alternative C2 frameworks.

Continuing this line of thought, Sliver, due to its open-source nature and implementation in Go, often benefits from a lower detection rate compared to Cobalt Strike, as Go-based malware is less frequently analyzed by security researchers. It offers robust encryption, transport-layer obfuscation, and modular payloads, enabling operators to blend its traffic with legitimate network activity. The use of mTLS and WireGuard tunnels enhances its resilience to traffic analysis, providing secure communication channels that are difficult to intercept. Furthermore, Sliver's cross-platform compatibility makes it a suitable choice for campaigns targeting Linux and macOS systems, where traditional Windows-centric C2 frameworks are less effective.

By the same token, Mythic's modular design enables operators to create customized implants dissimilar to well-known payloads, thereby increasing the difficulty of detection. In contrast to Cobalt Strike's fixed feature set, which has been extensively studied by defenders, Mythic's dynamic nature allows for continuous modification of attack techniques. By employing diverse payload types and execution methods, Mythic operators can circumvent behavioral-based detections relying on common patterns in established C2 frameworks. However, Mythic's reliance on community-developed plugins means its effectiveness is largely contingent on their maintenance and updates.

As a more recent framework, Havoc benefits from a relative lack of widespread detection signatures, making it a potentially attractive option for red teams seeking to avoid signature-based detection. It incorporates modern evasion techniques, including direct *syscalls*, manual payload mapping, and in-memory execution, all of which contribute to increased resistance against traditional security tools. Havoc's customizable payloads and encrypted communication channels further enhance its stealth capabilities. However, its relative immaturity compared to established frameworks like Cobalt Strike and Sliver means its ecosystem of tools and third-party integrations is less developed, potentially requiring operators to create custom modules for comparable functionality.

As a novel approach, C3 logically exhibits a lower detection rate against EDRs compared to more established C2 frameworks. Its unique implementation and attack capabilities, specifically direct command execution within the OS and a minimal reliance on traditional evasion techniques (beyond

 doi:10.20944/preprints202502.2263.v1

reflective DLL loading), represent a significant achievement. However, this direct command execution approach presents certain limitations. For instance, implementing interactive shell communication is challenging. Consequently, commands requiring interactive shell functionality, such as *runas* or *ssh*, can be difficult to execute directly. Conversely, employing alternative methods, such as utilizing *schtasks* instead of *runas*, can enhance operational security and contribute to lower detection rates.

Another significant differentiating factor among these C2 frameworks is their ease of use and accessibility. Cobalt Strike, despite its power, presents a steep learning curve, demanding substantial experience in red teaming and offensive security for effective utilization. Its strict licensing controls, restricting distribution to vetted organizations, have unfortunately contributed to the proliferation of cracked versions within underground communities. Sliver, being open-source, offers greater accessibility, though advanced customization requires familiarity with Golang. Mythic, while highly extensible, necessitates expertise in API-driven automation and scripting for maximizing its potential. Havoc, as a more recent framework, is still under development, but its modern approach to stealth and evasion shows promise for those willing to invest time in mastering its capabilities. C3 distinguishes itself through its ease of use. Its concise codebase facilitates modification and the addition of payloads and functionalities.

From an OpSec perspective, selecting the appropriate C2 framework is contingent upon the specific adversary simulation objectives. Cobalt Strike remains a strong choice for well-resourced red teams requiring a comprehensive feature set and prepared to invest in extensive obfuscation and custom profiles for effective evasion. Sliver offers a compelling open-source alternative for those seeking cross-platform support and inherent evasion capabilities. Mythic is well-suited to teams prioritizing a modular and highly customizable approach. Havoc presents a promising option for those focused on modern evasion techniques, though further development is necessary to achieve feature parity with more mature frameworks. C3 provides a distinct advantage in operational security, leveraging its unique implementation to achieve a demonstrably low detection rate against prevalent EDR solutions.

As already pointed out, in real-world attack scenarios, adversaries frequently employ multiple C2 frameworks to enhance their operational effectiveness. A common tactic involves utilizing one framework for initial access and reconnaissance, subsequently transitioning to another for post-exploitation and persistence. For instance, an attacker might leverage C3 to establish an initial foothold on a Windows server and then deploy Cobalt Strike for more extensive post-exploitation activities. Alternatively, they could initiate the attack with Mythic for stealthy initial access and subsequently pivot to a different C2 framework for lateral movement. By combining diverse frameworks, attackers can diversify their tactics and mitigate the risk of detection associated with reliance on a single, potentially monitored toolset. Summarizing the above discussion, Table 1 presents a comparative overview of C3 and other prominent C2 frameworks across 13 key criteria.

 doi:10.20944/preprints202502.2263.v1

Table 1. Comparative analysis of established C2 frameworks and the proposed C3 solution

Feature	C2				
	Cobalt	**Sliver**	**Mythic**	**Havoc**	**C3**
License	Commercial	Open-Source	Open-Source	Open-Source	Open-Source
Language	C, Java	Go	Python, Go, C#	C, C++	C++, JS
Support	Windows	Windows, Linux, macOS	Windows, Linux, macOS	Windows	Windows, Linux, macOS
Payloads	Reflective DLL, Staged/ Stageless	Staged/ Stageless	Modular, Agent-based	Reflective DLL, Syscalls	Reflective DLL, Staged/ Stageless
Communication	HTTP/S, DNS, SMB TCP	HTTP/S, DNS, mTLS, WireGuard	HTTP, WebSockets, TCP	HTTP/S, TCP	HTTP/3, IPC
Evasion	Malleable C2, in-memory execution	Encrypted comms	Modular implants	Direct syscalls, manual mapping	In-memory execution
Credential Dumping	Mimikatz, DCSync, Pass-the-hash	Mimikatz	Build-in	Build-in	Build-in
Lateral Movement	Pass-the-ticket, SMB, RDP, SSH	SSH, RDP, SMB, DNS	Agent-to-agent pivoting	Named pipes, Token	Build-in (SMB, schtasks/runas, curl, etc.)
Persistence	Registry, Scheduled Tasks, WMI	Systemd, Cron, Registry	Custom	Registry, Process Injection	Built-in, Scheduled Tasks, .lnk files
Detection	High (widely used)	Moderate	Moderate	Moderate	Low to None
Customization	Partially with Malleable profiles, scriptable	High Difficulty	Moderate	Moderate	Easy
Usability	Steep learning curve	Moderate	Advanced	Moderate	Easy (In general), Moderate (Lateral movement)
Execution Method	Shellcode	Shellcode	Shellcode	Shellcode	Direct Code Execution

8. Related Work

To further evade detection, researchers have explored various obfuscation techniques, including encoding stolen data within images, encrypting payloads, or embedding information within seemingly benign web traffic. Such approaches significantly complicate detection by security tools reliant on behavior-based anomaly detection and browser sandboxing [12]. The effectiveness of these attacks depends on the ability to bypass security measures such as Content Security Policies (CSPs), browser sandboxing, and real-time traffic analysis.

This section discusses browser post-exploitation techniques explored in previous studies. While much prior work has focused on the specific capabilities of browser extensions, this study leverages the deeper post-exploitation functionality available in Chromium-based browsers, enabling a wider range of OS-level command execution and data exfiltration.

Web-based data exfiltration techniques have been extensively studied within the context of browser security. Prior research has demonstrated how attackers can leverage JavaScript-based keyloggers to capture and transmit user inputs in real time from a compromised browser [13]. Furthermore, browser APIs such as WebSockets, Fetch, and XMLHttpRequest have been exploited to exfiltrate stolen data while blending with legitimate traffic, making detection more challenging [14].

Session hijacking remains another significant concern, as attackers can intercept authentication tokens or session cookies to maintain persistent, unauthorized access to web services without triggering security alerts [15]. Beyond direct data exfiltration, adversaries have also manipulated browser

 doi:10.20944/preprints202502.2263.v1

components such as DNS and WebRTC requests to tunnel sensitive information outside monitored networks, effectively bypassing traditional network security mechanisms [16].

To further evade detection, researchers have explored various obfuscation techniques, including encoding stolen data within images, encrypting payloads, or embedding information within seemingly benign web traffic. Such approaches significantly complicate detection by security tools reliant on behavior-based anomaly detection and browser sandboxing [12]. The effectiveness of these attacks depends on the ability to bypass security measures such as CSPs, browser sandboxing, and real-time traffic analysis.

To our knowledge, the Earth Kitsune Advanced Persistent Threat (APT) group is the only publicly documented use of the Native Messaging API for post-exploitation. This APT group strategically leveraged the Native Messaging API to facilitate the execution of their *WhiskerSpy* backdoor, enabling seamless payload delivery and persistent access. In this attack, the malicious browser extension used native messaging to communicate with a local native messaging host, which then executed a shellcode-based loader. The extension initiated an inject command, which, rather than taking direct parameters, automatically connected to a hardcoded URL (`http://<delivery server>/help.jpg`) to download and decode the main payload. This payload, disguised as an image file (Help.jpg), contained shellcode that loaded and executed the WhiskerSpy backdoor [17]. Once deployed, WhiskerSpy shellcode established a secure communication channel with its C2 server using Elliptic-Curve Cryptography (ECC) for key establishment, ensuring encrypted command execution. The backdoor shellcode provided extensive remote capabilities, including an interactive shell, file manipulation (download, upload, delete, list), screenshot capturing, shellcode injection into other processes, and the ability to load additional executables and execute their exported functions. By using Native Messaging, Earth Kitsune bypassed browser sandboxing and CSP restrictions, as the malicious extension merely acted as a bridge to the system-level backdoor. This approach enabled persistent access, while remaining largely undetected by traditional browser security mechanisms.

Last but not least, the CursedChrome [18] Github repository contains a PoC malicious browser extension designed to turn Chrome into a covert Remote Access Trojan (RAT). Created for educational and research purposes, it allows attackers to gain control over a victim's browser, enabling session hijacking, keylogging, and data exfiltration. Once installed, the extension connects to an attacker-controlled server, granting full access to the victim's web sessions, including the ability to bypass authentication mechanisms and manipulate content in real time.

In comparison to WhiskerSpy and CursedChrome, which are the most pertinent prior works to the research presented here, C3 distinguishes itself in several key aspects. While both WhiskerSpy and Covert C2 leverage the Native Messaging API for post-exploitation, their approaches and capabilities differ significantly. WhiskerSpy employs a multi-stage approach, downloading a separate, shellcode-based payload disguised as an image file before executing the actual backdoor. C3, conversely, downloads and executes DLLs directly from the C2 server, streamlining the execution process. Furthermore, WhiskerSpy functions as a full-fledged backdoor, providing a wide range of remote administration capabilities, including an interactive shell, file manipulation, and screenshot capturing. C3, in its current implementation, focuses specifically on command execution and data exfiltration, although its functionality can be extended by the DLLs it executes. The security of C3 relies on HTTP/3 over QUIC security services, which are generally accepted to be robust [19,20]. Finally, the native application's role also varies. In WhiskerSpy, it acts as a shellcode loader, while in C3, it directly executes downloaded DLLs to perform post-exploitation tasks. Both techniques, however, effectively bypass browser restrictions and, in the case of C3, evade detection by modern EDR solutions.

CursedChrome, while also a malicious browser extension, takes a different approach. It aims to turn Chrome into a covert RAT by directly granting the attacker control over the victim's browser, enabling actions like session hijacking, keylogging, and data exfiltration. In contrast, both WhiskerSpy and C3 leverage the Native Messaging API to extend their reach beyond the browser sandbox and interact directly with the operating system. Thus, while CursedChrome operates within the confines

 doi:10.20944/preprints202502.2263.v1

of the browser, WhiskerSpy and C3 achieve deeper system-level access, enabling more extensive and persistent post-exploitation capabilities.

9. Conclusions

As cyber threats continue to advance, the development of next-generation C2 servers is crucial for bolstering cybersecurity resilience, enhancing automation, and mitigating risks. The proposed C3 framework introduces a secure, scalable, and adaptable approach to command and control, addressing key limitations of traditional frameworks. By incorporating enhanced stealth, optimized performance, adaptability, expandability, and decentralized communication, this system offers a novel approach to C2 infrastructure in cybersecurity operations. This work contributes to the ongoing discourse in offensive security, penetration testing, and red team operations, providing a foundation for researchers and security professionals to advance the fields of ethical hacking and network security.

For future research, offensive C2 infrastructures must evolve to maintain effectiveness, resilience, and stealth against increasingly sophisticated defenses. A key direction lies in enhanced automation and self-adaptive C2 frameworks, leveraging artificial intelligence and machine learning to optimize command execution, lateral movement, and real-time decision-making. This would minimize manual operator intervention, enabling more autonomous operations. Furthermore, exploring multi-layered and redundant C2 architectures is essential to counter detection and takedown attempts. This includes investigating Peer-to-peer (P2P) C2, domain fronting, and the use of blockchain or decentralized services to ensure uninterrupted command execution, even in contested environments. Integrating serverless and ephemeral infrastructure, such as cloud functions and containerized C2 nodes, can further enhance operational security by minimizing forensic footprints and dynamically shifting C2 endpoints.

Future work should prioritize enhancing autonomous tasking and action execution. Scripted playbooks and intelligent decision trees can reduce operator workload while maximizing operational efficiency. Next-generation C2 frameworks should dynamically adapt their tactics – for example, by implementing interactive shell communication – to ensure persistent access within complex target networks. Additionally, exploring alternative communication channels between the C2 server and compromised hosts, such as WebSockets and WebRTC, can provide further avenues for communication. By incorporating these advancements, offensive C2 capabilities can achieve greater stealth, automation, and resilience, thereby increasing their effectiveness in contested cybersecurity environments.

Author Contributions: Conceptualization, E.C. and G.K.; methodology, E.C.; software, E.C.; validation, E.C. and G.K.; formal analysis, E.C. and G.K.; investigation, E.C.; resources, E.C.; data curation, E.C.; writing—original draft preparation, E.C.; writing—review and editing, E.G. and G.K.; visualization, E.C.; supervision, G.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The relevant implementation of C3 is available in the following public GitHub repository [4].

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

 doi:10.20944/preprints202502.2263.v1

API	Application Programming Interface
APT	Advanced Persistent Threat
C2	Command & Control
C3	Covert C2
CIA	Confidentiality, Integrity and Availability
CMD	Command prompt
CSP	Content Security Policies
CSP	Content Security Policy
DLL	Dynamic-Link library
ECC	Elliptic-curve cryptography
ECC	Elliptic-Curve Cryptography
EDR	Endpoint Detection and Response
GPO	Group Policy Objects
MS	Microsoft
OpSec	Operational Security
OS	Operating System
P2P	Peer-to-Peer
PoC	Proof-of-Concept
RAT	Remote Access Trojan
SSH	Secure Shell
TLS	Transport Layer Security
VM	Virtual Machine
WDAC	Windows Defender Application Control

References

1. Amjad, M.F.; Asif, M.; Ahmad, I.; Alqarni, H. Survey on Botnet Detection Techniques: Classification, Methods, and Evaluation. *Security and Communication Networks* **2021**, *2021*, 1–20. https://doi.org/10.1155/2021/6640499.
2. Owen, H.; Zarrin, J.; Pour, S.M. A Survey on Botnets, Issues, Threats, Methods, Detection and Prevention. *Journal of Cybersecurity and Privacy* **2022**, *2*, 74–88. https://doi.org/10.3390/jcp2010006.
3. Gaonkar, S.; Dessai, N.F.; Costa, J.; Borkar, A.; Aswale, S.; Shetgaonkar, P. A Survey on Botnet Detection Techniques. In Proceedings of the 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE), 2020, pp. 1–6. https://doi.org/10.1109/ic-ETITE47903.2020.Id-70.
4. Chatzoglou, E. Using Native Messaging API for Post-Exploitation. https://github.com/efchatz/Covert-C2. Accessed 14/02/2024.
5. Fortra. Cobalt Strike. https://www.cobaltstrike.com/. Accessed 11/02/2024.
6. Mythic. Mythic - A collaborative, multi-platform, red teaming framework. https://github.com/its-a-feature/Mythic. Accessed 11/02/2024.
7. Sliver. Sliver - Adversary Emulation Framework. https://github.com/BishopFox/sliver. Accessed 11/02/2024.
8. Iyengar, J.; Thomson, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, 2021. https://doi.org/10.17487/RFC9000.
9. Smiliotopoulos, C.; Kambourakis, G.; Kolias, C. Detecting lateral movement: A systematic survey. *Heliyon* **2024**, *10*, e26317. https://doi.org/https://doi.org/10.1016/j.heliyon.2024.e26317.
10. VirusTotal. YARA Releases. https://github.com/VirusTotal/yara/releases. Accessed 11/02/2024.
11. 5pider. Havoc. https://github.com/HavocFramework/Havoc. Accessed 11/02/2024.
12. Acar, G.; Englehardt, S.; Narayanan, A. No boundaries: data exfiltration by third parties embedded on web pages. *Proc. Priv. Enhancing Technol.* **2020**, *2020*, 220–238. https://doi.org/10.2478/POPETS-2020-0070.
13. Oren, Y.; Kemerlis, V.P.; Sethumadhavan, S.; Keromytis, A.D. The Spy in the Sandbox - Practical Cache Attacks in Javascript. *CoRR* **2015**, *abs/1502.07373*, [1502.07373].
14. Papadopoulos, P.; Ilia, P.; Polychronakis, M.; Markatos, E.P.; Ioannidis, S.; Vasiliadis, G. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. *CoRR* **2018**, *abs/1810.00464*, [1810.00464].

 doi:10.20944/preprints202502.2263.v1

15. Snyder, P.; Karami, S.; Edelstein, A.; Livshits, B.; Haddadi, H. Pool-Party: Exploiting Browser Resource Pools for Web Tracking. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, 2023; pp. 7091–7105.
16. Born, K. Browser-Based Covert Data Exfiltration. *CoRR* **2010**, *abs/1004.4357*, [1004.4357].
17. Joseph C Chen, J.H. Earth Kitsune Delivers New WhiskerSpy Backdoor via Watering Hole Attack. https://www.trendmicro.com/en_us/research/23/b/earth-kitsune-delivers-new-whiskerspy-backdoor.html. Accessed 11/02/2024.
18. Matthew Bryant, Trevor Elwell, D.T. CursedChrome - Chrome-extension implant that turns victim Chrome browsers into fully-functional HTTP proxies, allowing you to browse sites as your victims. https://github.com/mandatoryprogrammer/CursedChrome. Accessed 11/02/2024.
19. Chatzoglou, E.; Kouliaridis, V.; Karopoulos, G.; Kambourakis, G. Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study. *Int. J. Inf. Sec.* **2023**, *22*, 347–365. https://doi.org/10.1007/S10207-022-00630-6.
20. Chatzoglou, E.; Kouliaridis, V.; Kambourakis, G.; Karopoulos, G.; Gritzalis, S. A hands-on gaze on HTTP/3 security through the lens of HTTP/2 and a public dataset. *Comput. Secur.* **2023**, *125*, 103051. https://doi.org/10.1016/J.COSE.2022.103051.