

Enhancing Command & Control Capabilities: Integrating Cobalt Strike's Plugin System into a Mythic-based Beacon Developed at cirosec

Leon Schmidt

Zitiervorschlag im APA Stil:

Schmidt, L. (2025). *Enhancing Command & Control Capabilities: Integrating Cobalt Strike's Plugin System into a Mythic-based Beacon Developed at cirosec*. Hochschule Offenburg.

Abstract

Command & Control (C2) frameworks are a popular tool for bad actors to attack and infiltrate infrastructures and systems. They allow long-lasting inroads to be made into the infrastructure, through which attackers can interact with it through covert channels. These frameworks thus also play a crucial role in cybersecurity, enabling red teams and penetration testers to simulate those real-world adversary tactics. Cobalt Strike, a widely used proprietary C2 framework, offers an extensible plugin system through Beacon Object Files (BOFs). Mythic, an open-source alternative, provides a modular architecture but lacks native BOF compatibility.

This thesis explores the feasibility of integrating Cobalt Strike's BOF capabilities into a Mythic-based beacon developed at cirosec. The research begins by analyzing the structural and functional differences between Cobalt Strike and Mythic, focusing on their plugin systems and execution environments. It then examines the technical details of BOF execution, including Dynamic Function Resolution (DFR), memory management, and interactions with the beacon Application Programming Interface (API).

The core contributions of this work are the design and implementation of a generic BOF runtime and the implementation of it within the Mythic-based beacon "ciroStrike" developed by cirosec. By adapting BOF execution mechanisms and ensuring compatibility with Mythic's architecture, this integration enhances the beacon's flexibility while maintaining its compact and evasive nature. Furthermore, an analysis of publicly available BOF implementations evaluates their applicability to this approach.

The results demonstrate that BOFs can be successfully executed within Mythic with minimal modifications, bridging the gap between proprietary and open-source C2 frameworks. This research contributes to the evolution of offensive security tooling by expanding the interoperability of red team frameworks and improving the adaptability of C2 beacons.

Nutzungsbedingungen

Dieses Dokument wird unter diesen Bedingungen zur Verfügung gestellt:
Urheberrechtlich geschützt
Für weitere Informationen siehe:
<https://rightsstatements.org/page/InC/1.0/>

Kontakt

Hochschule Offenburg | Bibliothek
Badstraße 24
77652 Offenburg
Telefon: (0781) 205-240
E-Mail: bibliothek@hs-offenburg.de
www.hs-offenburg.de/bibliothek



Master's thesis for the acquisition of the academic degree "Master of Science (M.Sc.)"

Enhancing Command & Control Capabilities: Integrating Cobalt Strike's Plugin System into a Mythic-based Beacon Developed at cirosec

Leon Schmidt, B.Sc.
University of Applied Sciences Offenburg
Course of studies: Computer Science (Master)
Offenburg, Germany

Supervision

Prof. Dr. rer. nat. Daniel Hammer
Professor for IT Security
University of Applied Sciences Offenburg
Offenburg, Germany

Michael Brügge, Ing. M.Sc.
Managing Security Consultant
cirosec GmbH
Heilbronn, Germany

Preface

First and foremost, I would like to express my sincere gratitude to cirosec, and in particular Michael Brügge and Hagen Molzer, for supporting me as a working student throughout the majority of my studies and for supporting and making this thesis possible. Their guidance and the opportunities they provided have been invaluable to my academic and professional development in the field of cybersecurity.

Furthermore, I would like to thank Offenburg University of Applied Sciences and especially Dr. Daniel Hammer for supervising this thesis, as well as my bachelor's thesis. The university's endorsement of my personal objectives and aspirations and the possibility of the resulting individual course of study is deeply appreciated.

I am also grateful to my parents and grandparents for their financial and emotional support, which has allowed me to focus on my studies without undue worries. Additionally, I would like to thank my fellow students and friends, especially Svenja Kuschmierz, Nicola Jäger, Pirmin Arnold, Martin Christoph and Robert Eikmanns, for their encouragement, motivation, and the countless discussions that have contributed to my knowledge and personal growth. I would like to thank all the other persons who have proofread this work.

Disclaimer

This document is intended for educational and research purposes only. The techniques and tools discussed herein are related to offensive IT security, ethical hacking, and penetration testing, which are strictly to be used in controlled environments with proper authorization.

The author does not condone, support, or encourage any illegal or unethical activity, including but not limited to unauthorized access, data theft, or other forms of cybercrime. All demonstrations, examples, and case studies provided in this work aim to improve defensive measures and raise awareness of potential vulnerabilities in systems and networks.

Readers are advised to adhere to all applicable laws, regulations, and ethical guidelines when applying the knowledge and techniques presented. Any misuse of the information contained in this document is the sole responsibility of the individual or entity involved, and the author disclaims any liability for such misuse.

By reading or utilizing this document, you agree to these terms and affirm your commitment to using this information responsibly.

Transparency Statement

Parts of this thesis were supported by the following AI tools:

- OpenAI's ChatGPT (GPT-4o model) and Google's Gemini (1.5 Flash model) for data analysis and generation of smaller code snippets (always labelled as such)
- DeepL Translate for translations from German to English
- DeepL Write for text enhancements

All content generated or translated has been critically reviewed and adapted to ensure academic integrity and alignment with the research objectives. Furthermore, generated content, excluding running text, is always labelled accordingly, both within this document and in all by-products (especially source code). The processing and interpretation of references, as well as the citation thereof, were carried out manually without exception.

Abstract

Command & Control (C2) frameworks are a popular tool for bad actors to attack and infiltrate infrastructures and systems. They allow long-lasting inroads to be made into the infrastructure, through which attackers can interact with it through covert channels. These frameworks thus also play a crucial role in cybersecurity, enabling red teams and penetration testers to simulate those real-world adversary tactics. Cobalt Strike, a widely used proprietary C2 framework, offers an extensible plugin system through Beacon Object Files (BOFs). Mythic, an open-source alternative, provides a modular architecture but lacks native BOF compatibility.

This thesis explores the feasibility of integrating Cobalt Strike’s BOF capabilities into a Mythic-based beacon developed at cirosec. The research begins by analyzing the structural and functional differences between Cobalt Strike and Mythic, focusing on their plugin systems and execution environments. It then examines the technical details of BOF execution, including Dynamic Function Resolution (DFR), memory management, and interactions with the beacon Application Programming Interface (API).

The core contributions of this work are the design and implementation of a generic BOF runtime and the implementation of it within the Mythic-based beacon “ciroStrike” developed by cirosec. By adapting BOF execution mechanisms and ensuring compatibility with Mythic’s architecture, this integration enhances the beacon’s flexibility while maintaining its compact and evasive nature. Furthermore, an analysis of publicly available BOF implementations evaluates their applicability to this approach.

The results demonstrate that BOFs can be successfully executed within Mythic with minimal modifications, bridging the gap between proprietary and open-source C2 frameworks. This research contributes to the evolution of offensive security tooling by expanding the interoperability of red team frameworks and improving the adaptability of C2 beacons.

Index terms: IT Security, Red Teaming, Malware Development, Command and Control, C2, Cobalt Strike, Mythic, Beacon Object Files, BOF

Table of Contents

Preface	I
Disclaimer	II
Transparency Statement	II
Abstract	III
List of Figures	VI
List of Tables	VII
List of Listings	VIII
List of Abbreviations	IX
1 Introduction	1
1.1 Motivation	1
1.2 Objective	2
1.3 Methodology	2
1.4 Time and organizational conditions	3
2 Basics on C2 Frameworks	5
2.1 Cobalt Strike	8
2.2 Mythic	12
2.3 Comparison and Plugin Systems	15
2.4 Other C2 Frameworks and Beacons	20
3 Examination of Beacon Object Files (BOFs)	23
3.1 Intended Functionality of BOFs	23
3.2 Examination of the Beacon APIs	26
3.3 Compilation Instructions	36
3.4 Understanding the COFF File Format	37
4 Analysis of existing public BOF implementations	47
4.1 Preparing a BOF sample collection	47
4.2 Assessing the Usage of Aggressor Script	54
4.3 Assessing the Usage of Beacon APIs	56
4.4 Assessing the Usage of DFR	57
5 Implementation of a generic C++ BOF runtime	59
5.1 Requirements	59
5.2 Existing or Related BOF runtimes	63
5.3 Runtime Implementation and Public API	67
5.4 Other Considerations	79
5.4.1 Linking BOFs on the Server	79
5.4.2 Resolving DFR Symbols on the Server	80

5.4.3 Using a pre-built Windows Executable as a Runtime	80
5.4.4 Implementing an Exemplary BOF Using all Beacon APIs	81
6 Implementation into ciroStrike	83
6.1 Current State of ciroStrike	83
6.2 Beacon Implementation	85
6.3 Server-side Mythic Implementation	94
6.4 Adding Compatibility to use Command Augmentation with Forge	97
7 Conclusion	105
7.1 Summary	105
7.2 Outlook	106
Bibliography	107
Statutory Declaration	113
A Git commit hashes for tools used in this thesis	114
B Cobalt Strike's beacon.h	116
C Code for Checking Beacon API Usage in GitHub Repository	126
D Code for converting a file to a C/C++ char buffer	129
E Code for the discontinued Use-it-all BOF	130

List of Figures

Figure 1: C2 infrastructure with a simple single controller (own illustration)	7
Figure 2: C2 infrastructure with two redirectors using different protocols (own illustration)	7
Figure 3: Example of a staged Cobalt Strike payload using persistence methods [1]	9
Figure 4: Mythic traffic flow diagram [2]	13
Figure 5: Simplified illustration of a full compilation process on Windows (own illustration)	37
Figure 6: COFF File Structure [3, p. 2]	44
Figure 7: Sample COFF Object File [3, p. 3]	44
Figure 8: Schema on how a BOF interacts with the beacons internal code (own illustration)	85
Figure 9: Mythic’s parameter UI for the <code>execute_bof</code> command	97
Figure 10: Compatibility layer workings within the “Forge” plugin (own illustration)	98
Figure 11: Mythic’s parameter UI for the <code>execute_bof</code> command with typed array parameter	101
Figure 12: Mythic’s parameter UI for the <code>forge_bof_nanodump</code> command	102

List of Tables

Table 1: List of beacon API groups with a short description	26
Table 2: Constants for the <code>&bof_pack</code> format string and the corresponding unpacking functions [4] .	28
Table 3: List of the BOF sample collection with their usage of Aggressor Script	54
Table 4: List of the BOF sample collection with their usage of the Cobalt Strike beacon APIs	56
Table 5: List of BOF with their usage of DFR	57
Table 6: General requirements for the BOF runtime	60
Table 7: Loader requirements for the BOF runtime	61
Table 8: Compatibility layer requirements for the BOF runtime	62
Table 9: Public API requirements for the BOF runtime	63
Table 10: List of BOF runtimes with their respective support for the Cobalt Strike beacon APIs	64
Table 11: Git commit hashes for publicly available C2 frameworks used in this thesis	114
Table 12: Git commit hashes for publicly available Mythic beacons used in this thesis	114
Table 13: Git commit hashes for publicly available BOFs used in this thesis	114
Table 14: Git commit hashes for publicly available BOF/COFF runtimes used in this thesis	115
Table 15: Git commit hashes for other repositories used or referenced in this thesis	115

List of Listings

Listing 1: Malleable C2 Profile to imitate OCSP as implemented in Windows (truncated) [5]	10
Listing 2: Example Sleep script showing the random generation of a string with specified length [6]	16
Listing 3: Example Aggressor Script listening for all beacon events [7]	18
Listing 4: Using DsGetDcNameA and NetApiBufferFree from netapi32.dll via DFR	25
Listing 5: Example BOF that reads three arguments and prints them	27
Listing 6: Example Aggressor Script that adds a command with three arguments to Cobalt Strike	28
Listing 7: Using the Spawn+Inject beacon API to inject shellcode from a BOF	31
Listing 8: Minimal example of a BOF printing Hello World and the raw argument string [8]	36
Listing 9: Type definitions for external functions required in the BOF loader	69
Listing 10: Struct definition of external_functions_t in the BOF runtime	70
Listing 11: Usage of functions within the external_functions_t struct	70
Listing 12: Type and struct definitions for the beacon API functions for the BOF loader	71
Listing 13: Function signature of UnhexlifyArgs from the BOF runtime	72
Listing 14: Function signature of RunBOF from the BOF runtime	73
Listing 15: Simplified loading process of the BOF runtime	74
Listing 16: Simplified implementation of symbol processing in the BOF runtime	78
Listing 17: Reduced Command struct definition in ciroStrike	87
Listing 18: Reduced headers for commands supported by ciroStrike	88
Listing 19: Reduced command type switch for the execute_bof command in ciroStrike	88
Listing 20: Functions to access the Thread Local Storage without using the Windows API	91
Listing 21: Implementation of BeaconFormatAlloc using the custom TLS implementation	91
Listing 22: Reduced concrete implementation of the execute_bof command in ciroStrike	93
Listing 23: Translation container implementation of the execute_bof command	95
Listing 24: Mythic definition of the execute_bof command in ciroStrike	96
Listing 25: Apollo support entry in payload_type_support.json in Forge [9]	99
Listing 26: Mythic argument definition for the argument array required by Forge	100
Listing 27: Example of a typed array in Mythic	100
Listing 28: Translation container implementation of the execute_bof command, adjusted for Forge	102

List of Abbreviations

AD:	Active Directory
API:	Application Programming Interface
APT:	Advanced Persistent Threat
ARP:	Address Resolution Protocol
BOF:	Beacon Object File
BUD:	Beacon User Data
C2:	Command & Control
CLI:	Command Line Interface
CLR:	Common Language Runtime
COFF:	Common Object File Format
CRT:	C Run-Time
CSV:	Comma-Separated Values
DFR:	Dynamic Function Resolution
DLL:	Dynamic Link Library
DNS:	Domain Name System
DSL:	Domain-Specific Language
EDR:	Endpoint Detection and Response
ELF:	Executable and Linking Format
ETW:	Event Tracing for Windows
HTA:	HTML Application
HTTP:	Hypertext Transfer Protocol
IoC:	Indicator of Compromise
JSON:	JavaScript Object Notation
JXA:	JavaScript for Automation
LDAP:	Lightweight Directory Access Protocol
LSASS:	Local Security Authority Subsystem Service
MSVC:	Microsoft Visual C++
NTLM:	New Technology LAN Manager

OCSP:	Online Certificate Status Protocol
OPSEC:	Operational Security
OS:	Operating System
PE:	Portable Executable
PIC:	Position Independent Code
PoC:	Proof of Concept
RAT:	Remote Access Trojan/Toolkit
REPL:	Read-Eval-Print Loop
RPC:	Remote Procedure Call
SDK:	Software Development Kit
SIEM:	Security Information and Event Management
SMB:	Server Message Block
SPN:	Service Principal Name
Syscall:	System Call
TEB:	Thread Environment Block
TGT:	Ticket Granting Ticket
TLS:	Thread Local Storage
TTP:	Tactics, Techniques, and Procedures
UDRL:	User-Defined Reflective Loader
URL:	Uniform Resource Locator
VBA:	Visual Basic for Applications
VPN:	Virtual Private Network

1 Introduction

Command & Control (C2) frameworks play a pivotal role in both offensive and defensive cybersecurity operations. C2 frameworks have evolved into sophisticated tools that allow security professionals to simulate adversarial Tactics, Techniques, and Procedures (TTP). These frameworks provide a versatile environment for managing payloads, executing commands, and controlling compromised systems, often leveraging encrypted communication channels to evade detection.

The increasing complexity of modern cyber threats has driven the development of advanced C2 frameworks, such as Cobalt Strike, and Mythic, which are widely used by red teams, penetration testers, and threat actors alike. These tools have become essential in assessing the resilience of organizational defenses by emulating real-world attacks and refining defensive measures. At the same time, they present a significant challenge for security practitioners tasked with detecting and mitigating unauthorized access and lateral movement within networks.

1.1 Motivation

Cobalt Strike is a proprietary solution which is costly: Licensing is per user per year [10], which quickly leads to high prices for a large, collaborative team. Due to its high profile in the C2 sector, there is also the problem that the beacon implementation is known to many Endpoint Detection and Response (EDR) solutions, such as the Microsoft Defender, where it is detected as malware. The reason for this is the largely constant signature of the beacon. Although Cobalt Strike offers its own collection of obfuscation methods for the beacon in its “Cobalt Strike Arsenal” [11], this offers far less flexibility than a custom implementation. However, Cobalt Strike offers the option of using so-called Beacon Object Files (BOFs) to extend the given possibilities of the beacon: they enable the implementation and transport of additional functions after the beacon has been deployed.

Mythic allows the use of a completely self-developed beacon, which significantly reduces detection by EDR solutions. Mythic is also fully open-source and, due to its 3-Clause BSD license, free to use [12]. cirosec has a young C2 beacon written for the Mythic framework called “ciroStrike”. A fundamental requirement of ciroStrike is its compactness, which makes it easier to bypass EDRs and facilitate its use in packers. Making Cobalt Strike’s BOFs available for ciroStrike is therefore a good way of maintaining this requirement and at the same time increasing its functional versatility. However, due to the differences in architecture and the design decisions that were made, this has not yet been possible. This work is intended to demonstrate the feasibility of such an approach while also making the numerous public BOF implementations, which were originally developed for Cobalt Strike, available to be used with ciroStrike.

There is currently no ready-to-use method for using BOFs in a Mythic beacon. However, there are some runtimes that make the format in which BOFs are transported executable [13], [14], [15], [16]. Some of them state that they are also suitable for executing Cobalt Strike BOFs to a specific degree. As part of this thesis, these implementations will be tested for their suitability and finally transferred into a common runtime, which is to be used in ciroStrike.

The many publicly available BOF implementations will contribute as examples to the development of this runtime. They will later be used for testing the implementation of the BOF runtime.

The research and documentation of BOFs would not only enhance the capabilities of the Mythic framework but also to open-source development in cybersecurity in general. By bridging the gap between proprietary and open-source tools, this research seeks to provide a comprehensive solution that leverages the strengths of both approaches, thereby contributing to the advancement of C2 frameworks in both offensive and defensive contexts. This is especially true due to the fact that there are already many BOF implementations available in public code repositories such as GitHub. Having a good understanding of how BOFs work would allow other frameworks to utilize these existing implementations without rewriting them from scratch.

1.2 Objective

This work is orientated towards implementing a BOF runtime in ciroStrike. However, the scientific added value of the thesis lies in describing the BOF format extensively and explaining it in a framework-agnostic way, so that it can also be used in other environments. An exemplary C++ library for the local execution of BOFs is to be created.

This raises the following research questions, which are to be answered by this thesis:

- 1) How is the BOF format structured, and how is it made executable?
- 2) What are the features apart from BOF that are provided by the Cobalt Strike Team Server and Beacon to enable their execution?
- 3) Is it possible to develop a runtime based on this proprietary technology?
- 4) What are the requirements that also need to be met to make existing BOF implementations run in Mythic and potentially other C2 frameworks?

These questions are addressed methodologically, as described in the following chapter, and the results are documented.

1.3 Methodology

Chapter 2 starts with the basics of C2 frameworks. The general framework-agnostic architecture will be described and the corresponding use cases in the context of offensive security practice will be explained. In this chapter, the interfaces addressed by this thesis are to be worked out. Subsequently, the Cobalt Strike and Mythic frameworks and their functions will be discussed specifically. This preliminary work is done so that ultimately the plug-in systems used in them can be worked out and compared with each other. In particular, side effects and dependencies within the frameworks will be discussed. It will be shown which architectural adaptations to Mythic are necessary in order to be able to add a compatibility layer to plugins that were initially developed for Cobalt Strike.

Chapter 3 examines the BOFs as an independent unit. The advantages of the format will be demonstrated and justified. The development best practices and build instructions specified by Cobalt Strike are then examined in order to explain how the BOF format comes about. In particular, the provided `beacon.h`

header file is consulted for this purpose. Finally, the functions to be implemented are examined for usefulness and implementability, and potential obstacles are identified.

Chapter 4 deals with the analysis of existing, publicly available implementations of various BOFs. The most popular BOFs on GitHub – measured by the number of GitHub stars – and the official Cobalt Strike Community Kit [17] are used for this purpose. In addition, the BOFs frequently used by cirosec will be identified. For each of these implementations, the functions they commonly use in the Cobalt Strike framework will be worked out in order to define the subset of functions that should be implemented in the generic runtime. This chapter therefore partly serves to define the first requirements for the BOF runtime.

In Chapter 5 the generic BOF runtime will be developed and implemented. The landscape of existing implementation approaches is examined and evaluated. The generic BOF runtime is to be developed as a standalone C++ static library. The headers examined in Chapter 3.2 are to be implemented here as far as a generic approach allows. Care should be taken not to publish any functionality native to ciroStrike in the function implementations, primarily with the aim of preventing the generation of malware signatures. The generic runtime is to be published on the GitHub platform under the BSD license.

In Chapter 6, the generic runtime is to be integrated into cirosec’s own C2 beacon ciroStrike. The existing requirements (size of the beacon, obfuscation by the C2 profile, etc.) are to be preserved to maintain the lightweight characteristics of ciroStrike. Special care should be taken to ensure that existing public BOF implementations are compatible with this system. Furthermore, the necessary translation containers are adjusted, and command definitions are implemented to allow seamless execution of BOFs through Mythic’s tasking interface. Finally, the command augmentation plugin “Forge” is addressed. This is a new Mythic extension that is supposed to simplify the handling of BOFs for the operator. In this chapter, compatibility between ciroStrike and Forge is established. The integrated runtime is then extensively tested with the BOFs offered by Forge.

Chapter 7 concludes the thesis by summarizing the technical contributions and reflecting on the feasibility and impact of the integration. It highlights the architectural trade-offs and discusses the potential for future extensions. The outlook provides concrete next steps for runtime extensions, the integration of Aggressor Script compatibility, and ideas for enhancing Operational Security (OPSEC).

1.4 Time and organizational conditions

Work on this thesis started on October 1st 2024 and was supported by a Master’s contract of employment with cirosec GmbH. The submission date set by the examination office at Offenburg University was April 1st 2025. The work was supervised by Michael Brügge, Ing. M.Sc. on cirosec’s side and by Prof. Dr. rer. nat. Daniel Hammer on the university’s side. There were occasional reviews during the course of the thesis until final submission.

2 Basics on C2 Frameworks

In this chapter, the general functionality of C2 frameworks, as well as that of Cobalt Strike and Mythic in particular, will be examined. The differences between the frameworks are to be worked out. Special attention will be paid to the plugin systems of both frameworks in order to find out which architectural challenges have to be met when implementing an custom plugin system.

C2 frameworks are an important element of modern cybersecurity strategies and threats. They serve as a communication infrastructure through which cyber criminals and Advanced Persistent Threats (APTs) can coordinate large-scale attacks to connect to and control compromised systems. They can also be utilized to build and coordinate botnets. Such frameworks are often used in real-life attack scenarios, but also as part of red teaming exercises. The goal of a C2 framework is to maintain persistent and covert communication with compromised systems, execute commands and send results back to the so-called controller. Security experts use them as part of penetration tests and red teaming exercises to assess a company's resistance to targeted attacks. Red teams simulate real attackers in order to uncover weaknesses in the security apparatus and test the effectiveness of the detection and response mechanisms, carried out by the defenders – the “blue team”.

According to the established Cyber Kill Chain developed by Lockheed Martin, adversarial attacks can be divided into several, ordered phases [18]:

- 1) **Reconnaissance:** Reconnaissance is understood as the practice of covertly discovering and collecting information about a system or multiple systems that are to be attacked. [19]
- 2) **Weaponization:** Weaponization couples an exploit with a backdoor into a deliverable payload (e.g., a beacon or stager). This often includes the development and customization of code that exploits vulnerabilities.
- 3) **Delivery:** Delivery names the transportation of the weaponized bundle to the victim via several means like email, web, and others.
- 4) **Exploitation:** Exploitation refers to the targeted misuse of a vulnerability in software, hardware or networks to execute the code on a victim's system.
- 5) **Installation:** Installation describes the phase in which attackers successfully place and execute malware or other malicious components on a victim's system. The aim is to establish permanent access to the system and create the basis for further actions such as data exfiltration or privilege escalation.
- 6) **Command & Control (C2):** C2 establishes a command channel for remote manipulation of a victim, as prepared by the previous step.
- 7) **Actions on Objectives:** Actions on Objectives is understood as the phase in which attackers realize their actual goals after gaining full access to a system. These actions may include data theft, sabotage, espionage or other malicious activities that reflect the original purpose of the attack.

Especially when doing a red team operation, the following phases are additionally featured, some of which are very similar to the steps in the Cyber Kill Chain [18], [20]:

- **Initial access:** Initial access consists of techniques that adversaries may use as entry vectors to gain an initial foothold within a system or environment.
- **Lateral movement:** Lateral movement refers to moving from a compromised system to other systems within the network to compromise them as well. These techniques abuse default credentials, known accounts, and vulnerable services.
- **Privilege escalation:** Privilege escalation consists of techniques that adversaries use to gain higher-level permissions on a system or network. Privilege escalation may be part of lateral movement.
- **Persistence:** Persistence describes the techniques to maintain access to a temporarily compromised system to be able to access it again at a later stage.

Although C2 frameworks reside in the Command & Control step of the Cyber Kill Chain, they can be utilized in any of these steps, as they only serve as a vehicle to carry out the practices mentioned. The MITRE ATT&CK Framework also lists Command & Control as an independent tactic [21].

The C2 principle is implemented using two main components, the “beacon” (also known as the “agent” or “implant”) and the “controller” (also known as the “team server”). The beacon, which can also be categorized as Remote Access Trojan/Toolkit (RAT), is the component that is installed and often hidden on the compromised system using various techniques. This can be done using process injection or a dedicated shellcode loader, for example. Once the beacon is launched, it connects back to the C2 infrastructure. Each new incoming connection from a beacon is usually referred to as a “callback”. The payload data that is transmitted through the callback is usually hidden and obfuscated by two separate components – the “C2 profile” and the “translator”. The C2 profile defines the transport channel through which the payload data is transported (“carrier”). Usually, the Hypertext Transfer Protocol (HTTP) is employed for this, as it is frequently used for legitimate connections. It is rarely recognized as conspicuous in most environments and therefore rarely blocked. In some cases, other common network protocols such as the Domain Name System (DNS) or Server Message Block (SMB) named pipes are misused to hide these messages. The translator, on the other hand, is responsible for translating the payload data itself within the C2 profile. Its use is optional, but it can further obfuscate and thus hide this payload data. This type of obfuscation done by C2 profiles and the translator is called a “covert channel”. The red team can now send commands to the beacon through this channel. The connection is usually opened by the beacon using a polling procedure: it asks for new commands at a fixed or random interval and transmits any results of this command in the next iteration. To make the connections appear even more authentic, an additional jitter is calculated into the interval. Additionally, “chunking” is sometimes used to divide large payloads into several smaller packets in order to avoid large data packet anomalies, which can be seen as an Indicator of Compromise (IoC).

The controller is the second important component. It is the central control instance for all beacons. It must be accessible for every beacon – at least indirectly, when a redirector is in place, which are covered briefly. The controller is provided and administered by the red team. Depending on the C2 framework, the administration is carried out differently, for example via a web interface or a dedicated client. A default C2 infrastructure with a single controller is shown in Figure 1.

A beacon that regularly talks to its controller can also be detected as an anomaly, depending on how much data is being transmitted. Even a well-covert controller can be noticed simply because there is only one. Some frameworks therefore offer additional options for hiding the controller by using redirectors. Redirectors are usually smaller servers that are ideally equipped with their own IPs and hostnames. To hide them more effectively, they can also occur in even smaller units, e.g., within a shared cloud service. Their task is to redirect incoming traffic from the beacon to the controller. The controller can then be secured so that only the redirectors can access it in order to hide it more effectively. Several redirectors can (and should) be configured for each controller or beacon, so that beacons distribute their traffic among them in order to make it less conspicuous or more difficult to correlate. Another important advantage of this approach is that redirectors can be replaced very easily without having to rebuild the controlling infrastructure, which can impact the success of the operation. This is particularly useful if the victim (the blue team) recognizes the connection target of a beacon as malicious and blocks it. A C2 infrastructure setup with redirectors in place can be seen in Figure 2. Some beacons even allow proxying by other beacons in the environment, e.g., via the SMB protocol [22], however, these methods are not part of this thesis.

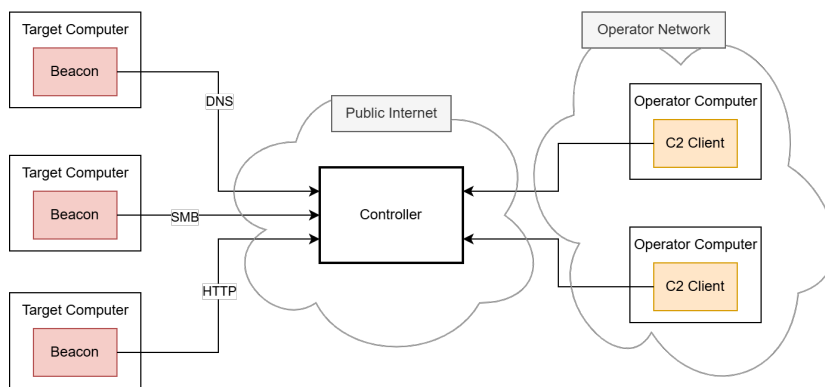


Figure 1: C2 infrastructure with a simple single controller (own illustration)

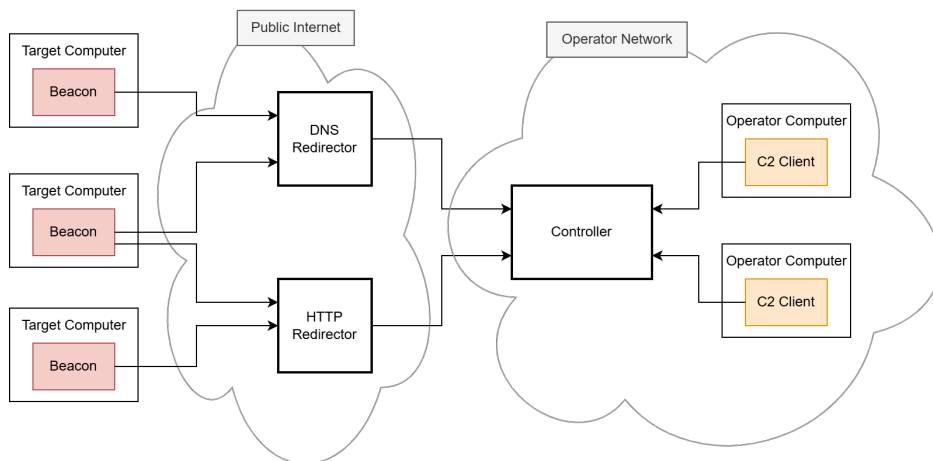


Figure 2: C2 infrastructure with two redirectors using different protocols (own illustration)

It is important to understand that C2 frameworks usually work according to the scheme explained here, but that some approaches differ both in function and, above all, in naming. The following chapters will focus specifically on Cobalt Strike and Mythic – the frameworks on which this thesis is centered.

2.1 Cobalt Strike

Cobalt Strike is a penetration testing software that has recently become a favored tool among malicious actors for the performance of sophisticated cyberattacks on vulnerable targets, due to its advanced evasion capabilities. First created by Raphael Mudge in 2012, Cobalt Strike was one of the first public red team C2 frameworks. Now, it is part of Fortra LLC [23].

As reported by Cisco’s Talon Incident Report from 2020 [24], Cobalt Strike was involved in 66% of the ransomware attacks of this year. According to Cisco’s Talos Year in Review 2022 [25], this hasn’t changed much. However, 2022 has seen an explosion in new offensive frameworks, which may present more challenges for defenders [25, p. 65]. Cobalt Strike is also employed as an APT, as evidenced by the 2020 SolarWinds hack. In this instance, the attackers delivered a bespoke Cobalt Strike payload via malicious software updates to 18,000 customers of the Orion software, including numerous global companies (e.g., Microsoft, Cisco) and various US government agencies. [26]

Package and Administration

Cobalt Strike comes as a batteries included solution. It contains a controller application to be set up on a Linux host, as well as a pre-configured and pre-implemented beacon. The controller – called “Team Server” in Cobalt Strike methodology – is written in Java. Administration takes place using a client GUI application also written in Java, that connects to the team server. Since Cobalt Strike is a collaborative framework, several clients can connect at the same time to work together on the red teaming campaign.

Payload Types

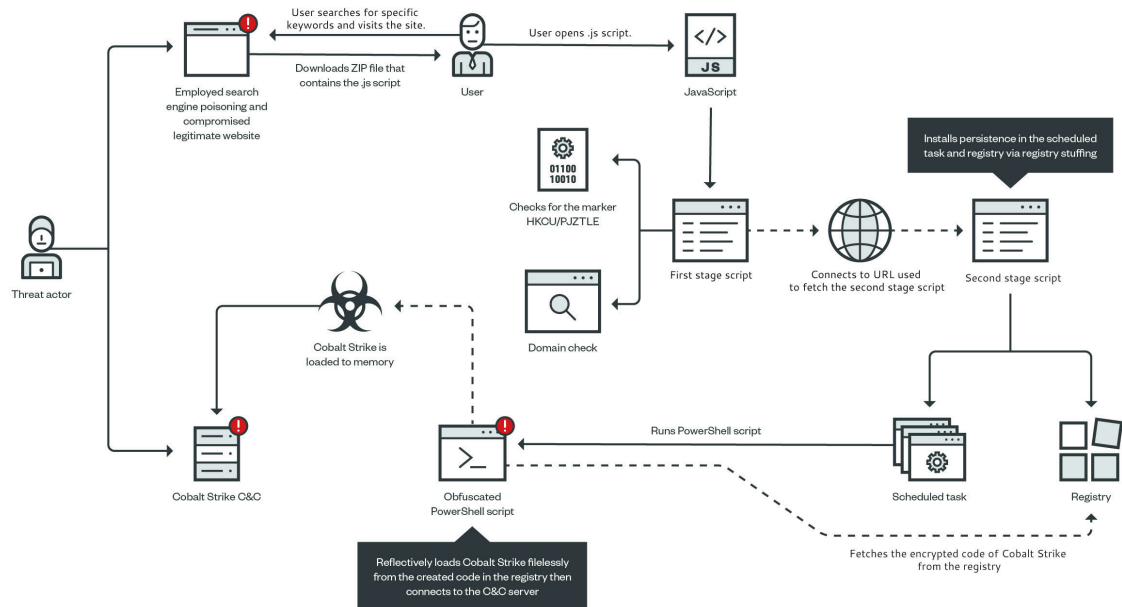
The beacon payload can be generated in different formats. A distinction is made between staged and stageless payloads. The latter is a payload type that basically contains the entire beacon code. Staged payloads, on the other hand, contain a much smaller so-called stager, which downloads the beacon code at execution time – usually from the Cobalt Strike Team server – and executes it. This enables the implementation of better evasion techniques regarding EDRs and often significantly reduces the payload size.

Cobalt Strike allows the deployment of the beacon using the following formats:

- **Stager/Stageless Payload:** Beacon shellcode for many languages, including C, C++, C#, PowerShell and more exotic languages like Ruby; can also output the payload format used by the Veil Evasion Framework.
- **Windows Stager Payload:** Stager transported as Windows EXE, DLL or service binary.
- **Windows Stageless Payload:** Beacon transported as Windows EXE, DLL, service binary, PowerShell script or raw binary.

- **HTML application:** Stager that runs the beacon from an HTML Application (HTA) file served via the Cobalt Strike web server (or any other web server).
- **Microsoft Office macro:** Visual Basic for Applications (VBA) stager which can be embedded into a Microsoft Office document.

The official Artifact Kit [11] allows more payload generation possibilities than Cobalt Strike can offer itself. Figure 3 shows an example where a custom stager is injected via JavaScript found in a poisoned search engine (top center) which checks for the correct environment and then downloads its second stage, generated by Artifact Kit (right). The beacon payload is stored in the registry for persistence while the loader is stored as a scheduled task (bottom right). When the user logs in, the scheduled task fires, loads the beacon payload from the registry via an obfuscated PowerShell script and connects back to the Cobalt Strike Team Server (bottom left).



©2022 TREND MICRO

Figure 3: Example of a staged Cobalt Strike payload using persistence methods [1]

Feature: Malleable C2

Cobalt Strike works with so-called malleable C2 profiles to disguise the traffic between the team server and the beacon. This is an abstract program definition that specifies how transmitted and received data should be transformed. The profile also defines the transport protocol and default values for possible transfer intervals and jitter. The memory footprint of the beacon can also be customized to correspond to a certain attack pattern, for example to emulate a specific APT or to make extra noise to test the reaction of the blue team. An example of a malleable C2 profile can be seen in Listing 1. This profile mimics Online Certificate Status Protocol (OCSP) traffic as generated by Windows.

```
1 set sleeptime "20000";
2 set jitter    "20";
3 set useragent "Microsoft-CryptoAPI/6.1";
4
5 http-get { ... } // redacted
6
7 http-post { // defines how HTTP POSTs should look like
8     set uri "/ocsp/a/"; // base uri used by both client and server
9     client {
10         header "Accept" "*/*"; // Requests headers added from the beacon
11         header "Host" "ocsp.verisign.com";
12         id { // appends beacon id with netbios encoding to uri
13             netbios;
14             uri-append;
15         }
16         output { // beacon output without transformation in POST body
17             print;
18         }
19     }
20
21     server {
22         header "Connection" "keep-alive"; // response headers
23         header "Content-Type" "application/ocsp-response";
24         output { // server output without transformation in response body
25             print;
26         }
27     }
28 }
```

Listing 1: Malleable C2 Profile to imitate OCSP as implemented in Windows (truncated) [5]

Feature: Reports

Cobalt Strike also has the feature of generating PDF reports from the collected logs and artifacts. These can provide the customer with an insight into the exploited vulnerabilities, but can also simply serve as a log. The following reports are supported [27]:

- **Activity Report:** The activity report provides a timeline of red team activities. Each post-exploitation activity is documented here.
- **Hosts Report:** The hosts report summarizes information collected by Cobalt Strike on a host-by-host basis. Services, credentials, and sessions are listed here as well.
- **Indicators of Compromise (IoCs) Report:** This report resembles an IoC appendix from a threat intelligence report. Content includes a generated analysis of the Malleable C2 profile, which domain was used, and MD5 hashes for uploaded files.

- **Sessions Report:** This report documents indicators and activity on a session-by-session basis. It includes: the communication path each session used to reach you, MD5 hashes of files put on disk during that session, miscellaneous indicators (e.g., service names), and a timeline of post-exploitation activity.
- **Social Engineering Report:** Since Cobalt Strike can also conduct Spear Phishing, this is the corresponding report listing all clicks by recipient.
- **Tactics, Techniques, and Procedures Report:** This report maps Cobalt Strike actions to tactics within the MITRE ATT&CK matrix.

Feature: Aggressor Script / CNA

As the Cobalt Strike Beacon comes pre-implemented and is only generated when needed, only a fixed set of commands is available by default, unlike in other frameworks. However, these can be extended dynamically using the Script Manager in the Cobalt Strike Team Server. The Script Manager allows so-called CNA files to be loaded, which may contain additional command definitions. These CNA files contain a scripting language specially developed for Cobalt Strike called “Aggressor Script”, which was introduced in version 3.0 of Cobalt Strike. It allows the beacon to be modified and extended after deployment. Aggressor Script often serves as an entry point for the execution of BOFs, as the latter are sometimes too complex to be executed without additional preparation or input validation. The Script Manager is the primary plugin system of Cobalt Strike and will be further investigated in Chapter 2.3.

Feature: Reflective Loaders

Cobalt Strike also supports the usage of a User-Defined Reflective Loader (UDRL) for beacon payloads. A loader like this makes it possible to execute the Cobalt Strike beacon payload in-memory using reflective Dynamic Link Library (DLL) injection and thus hide it more effectively. The staging properties, such as the activation of import table obfuscation, Portable Executable (PE) stomping or the general in-memory behavior, can be defined in the Malleable C2 profile. The Linux package also comes with the utility `pec_loader` which allows the extraction of a ready to use C2 profile block from an existing DLL [28]. Cobalt Strike offers its own UDRL called “UDRL Kit” obtainable from the Arsenal, which also has to be licensed. However, there are also some freely available loaders, that also makes it possible to develop a custom UDRLs based on them, such as the `BokuLoader` [29].

In conclusion, Cobalt Strike remains one of the most powerful and widely adopted tools in both red teaming and malicious cyber operations. Its pre-implemented beacon payloads, and extensive customization options through Malleable C2 profiles, Aggressor Script, and Artifact Kits allow it to evade detection and adapt to various attack scenarios. Although many of the described features of Cobalt Strike are of little relevance in this thesis, it is important not to hinder the respective counterparts in Mythic in their function by adapting `circoStrike`. The relevant features, such as Aggressor Script and the Script Manager, but also some parts of the Malleable C2 profiles, are examined further in the following chapters and compared with Mythic.

2.2 Mythic

The article “Mythic Case Study: Assessing Common Offensive Security Tools” by Team Cymru [30] examined the utilization of the Mythic C2 framework in various cyber activities. As of the first quarter of 2021, approximately 76 internet-facing Mythic servers were identified with the help of the Shodan search engine. This number constitutes about 2% of the C2 framework market share and surpasses that of other frameworks like Sliver but remains significantly lower than Cobalt Strike. A significant majority (90%) of these servers operated with default settings, such as a the web interface running on port 7443 and TLS certificates issued to a subject containing 0=Mythic. This uniformity facilitated the tracking of multiple servers through shared attributes like E-Tag information in HTTP headers. Whilst 76 servers may seem like a low number, this only accounts for internet-facing infrastructure and doesn’t include the Mythic instances being used for red team operations within closed environments (e.g., using redirectors). [30]

Some specific Mythic instances were discovered to be associated with the delivery of the “BazarLoader / BazarBackdoor” malware. Additionally, a campaign targeting Pakistan and Turkey was uncovered in which Mythic was used, among other C2 frameworks. The Uniform Resource Locator (URL) used in the campaign specifically identified the “Pakistani Ministry of Foreign Affairs” as the primary target. Some Mythic instances were also observed having the popular reconnaissance framework “reNgin” deployed on the same infrastructure to scan targets for vulnerabilities, followed by deploying a Mythic beacon for further exploitation. [30]

Architecture

Mythic is a “cross-platform, post-exploit, red teaming framework designed to provide a collaborative and user-friendly interface for operators” [2]. This is an important statement since this means that Mythic only delivers the interface and the platform Application Programming Interfaces (APIs) but no beacon. The project’s main goal is to provide quality of live improvements and robustness for the maintainability of beacons and for the operators in general. Furthermore, it provides a reliable and simple overview of statistics, utilized TTPs from the MITRE ATT&ACK framework and artifacts left during operations. This integration not only aids in maintaining operational security but also enhances the effectiveness of red teaming activities. To ensure OPSEC, Mythic commands can also conduct corresponding checks before and after commands are set [31].

Mythic is designed to facilitate a “plug-and-play architecture” where beacons, channels, obfuscations and other modifications are attached via modules. A clean installation only consists of a React frontend and several Docker containers which form the backend infrastructure. One of those containers contains the main server, that handles most of the web requests with GraphQL APIs and WebSockets. This server was rewritten from Python to Golang in Version 3.0 of Mythic in order to simplify the development of modules. The main server’s purpose is to access a PostgreSQL database and to orchestrate the other Docker containers using a RabbitMQ message broker service and the gRPC protocol. This allows each component to reside on separate physical machines or in different virtual machines, if desired. Furthermore, Mythic provides utility containers for testing GraphQL queries with “Hasura”, e.g., for the development of own modules, a documentation service based on the static site generator “Hugo” and a Jupyter server for testing scripts. A central nginx HTTP reverse proxy provides a single port to connect

to the different backend services and the frontend. The whole architecture, as documented in the official Mythic documentation [2], is shown in Figure 4. The orange harp on the right side of the picture is the logo of one of the more popular Mythic beacons “Apollo” – in this context, it represents a generic Mythic beacon.

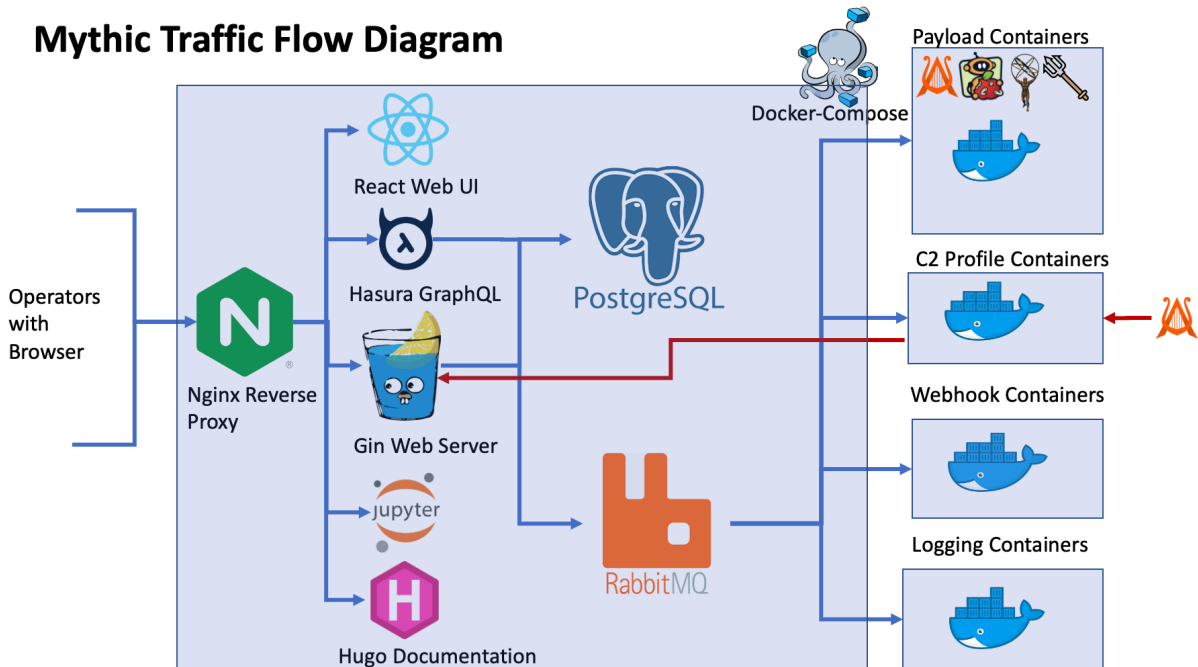


Figure 4: Mythic traffic flow diagram [2]

Components and modules

Since Mythic itself does not host any beacons or C2 profiles, they need to be installed separately, using `mythic-cli` – the Command Line Interface (CLI) tool for Mythic. The Mythic Community Overview can be used to identify publicly available C2 profiles, beacons/agents, wrappers, as well as webhook and logging services [32]. Popular choices for beacons are the Apollo beacons targeting Windows machines via .NET Framework 4.0, Apfell targeting OSX, and the Thanatos cross-platform beacons written in Rust. The most popular C2 profiles however are offered by Mythic itself, with options for HTTP, DNS and several other protocols as transport layers.

C2 Profiles and Translation Containers

As in Cobalt Strike, C2 profiles contain the entire receiving and sending logic for messages. In Mythic, a C2 profile is a Docker container that must also provide the HTTP server for an HTTP-based profile, for example. In contrast to Cobalt Strike, a C2 profile is developed entirely in a high-level programming language (usually in Python or Golang) and is not mapped using a separate format such as “Malleable C2”. Within these profiles, Mythic uses its own JavaScript Object Notation (JSON) schema for communication between the beacon and server, which is transported by the C2 profile (for example in an HTTP request/response header). Since JSON can appear relatively conspicuous under most circumstances, Mythic offers the option of choosing a custom transport format, which the beacon will use. This is made

possible by translation containers. The Mythic server uses them to serialize outgoing payload and to deserialize incoming payload back to Mythic's JSON format. The beacon never talks directly to the translation container, but continues to talk to the mythic server, which instructs the translation container by itself.

Feature: Scripting API

Since almost all actions in Mythic are handled using the GraphQL interface and WebSockets, it is possible to script all these actions in a user-defined way. Thanks to the standardized interfaces, any programming language can be used for this – Mythic provides corresponding Software Development Kits (SDKs) for Python and Golang. The backend infrastructure already comes with a graphical Hasura GraphQL engine, which can be used to explore the capabilities exposed to scripting [33]. The built-in Jupyter server can be used to develop and test scripts, which also comes with some scripting examples. This feature will only be referred to as “Mythic Scripting” in this thesis, as there are other, more specialized scripting methods in Mythic. Mythic Scripting and its possibilities will be further examined in Chapter 2.3.

Feature: Browser Script

In addition to scripting, Mythic offers another variant, the “Browser Scripts”. They can be used to process the command output of the beacon within the Mythic web interface and display it differently. For example, a beacon that supports the `tasklist` command for displaying running processes can provide a browser script that displays this output as a table instead of a simple text output for the Mythic operator. It is also possible to display buttons, for example for downloading screenshots or downloaded files. In addition to being provided by the beacon developer, operators can also add their own scripts or deactivate the provided scripts. As browser scripts are intended exclusively for displaying data in the web interface, they are not to be understood as a plugin system and are therefore no longer relevant for the rest of this thesis. [34]

Feature: Reports

Mythic allows the creation of reports per running or completed operation. However, these are much less customizable than in Cobalt Strike: there is one report type which contains all collected information. Only the MITRE ATT&CK coverages can be manually included or excluded. The reports are therefore less customizable to individual target groups (e.g., blue team executives), but contain very extensive information such as a list of all callbacks and the commands executed within them, compromised user accounts, environment information, and leftover artifacts.

Feature: Delegates / P2P messaging

For more complex setups, some beacons for Mythic allow the use of so-called “delegates”. They are a mechanism that allows tasks to be delegated between different beacons within a C2 network. They provide the ability to coordinate communication and collaboration between beacons, especially in scenarios where a beacon is required to perform tasks that are not included in its original payload type – i.e. that it does not support itself. Delegates allow a beacon to receive a task, forward it to another

beacon and report the result back to the original sender. This concept allows complex operations to be modular and efficient by combining the capabilities of different payloads. It minimizes the need to develop extensive or multifunctional payloads, as existing payloads can be efficiently integrated and used. Delegates thus promote flexibility and modularity in the development and use of beacons within the Mythic C2 framework. [35]

In summary, Mythic's extensibility and collaborative features have established it as a valuable asset in the arsenal of modern red teams, contributing to more efficient and effective security assessments. As noted already, Cobalt Strike remains the most popular C2 framework, but for some red team operators, an upward trend in the usage of Mythic is already occurring – it is also expected that malicious actors have taken note and are either considering or have already adopted the use of Mythic into their TTPs. [30]

2.3 Comparison and Plugin Systems

In summary, it can be stated that Cobalt Strike and Mythic only differ in a few, but very important aspects: the user interaction is largely the same, with the barely significant difference that Cobalt Strike uses a client application and Mythic uses a web interface. In both frameworks, C2 profiles are managed as standalone units and can be configured independently of the rest of the functionality of the framework. Mythic also has the translation containers that Cobalt Strike already covers with their Malleable C2 profiles, but this is also an insignificant difference. The non-technical supplementary features such as reporting and MITRE ATT&CK mapping are also similar in many respects.

The biggest differences are actually in the beacons, as well as how they establish and maintain the connection to their servers. Since there is virtually no coupling between the server and the beacon in Mythic in terms of how the beacon must be designed, there is also no common feature set: the beacon implementation is completely free to decide which Mythic functions are supported and used. This is particularly problematic when implementing plugin systems, which is probably the reason why Mythic doesn't offer one related to beacon functionality. The provision of such a system is left entirely to the beacon. This can of course be an absolute advantage, as it enables more flexible systems that can be adapted more easily to specific situations. The implementation, however, is significantly more complex. Therefore, the first step is to look at how Cobalt Strike establishes a plugin system.

Cobalt Strike is already able to transfer executable files and execute them in the beacon. This includes PowerShell scripts, native PE files and .NET executables. Various injection techniques, such as reflective DLL loading, are also already supported by the Cobalt Strike beacon. Many Mythic beacon implementations offer the same or similar techniques for executing executables, as these techniques are widely used and can therefore usually be easily ported to the respective beacon codebase. cirosec's Mythic beacon also contains these functions. Additionally, the underlying attacks are less sophisticated (except for the Reflective DLL approach), as they usually rely on existing functions of the Windows operating system – in particular the PE loader and the Common Language Runtime (CLR) for .NET executables, as well as the PowerShell runtime. Accordingly, what all these techniques have in common is that they are more easily detected by EDR software. Therefore, these techniques will not be discussed further in this thesis.

Scripting in Cobalt Strike: Sleep / Aggressor Script

As already addresses, Cobalt Strike relies on a scripting approach using Aggressor Script. Also, many of the built-in capabilities of Cobalt Strike are modeled with it by default. The Script Manager in Cobalt Strike represents the component that allows users to customize and modify the behavior of the teamserver and the Beacon by integrating external scripts written in Aggressor Script. This functionality enables operators to tailor the toolset to meet the unique demands of specific operations, environments, or targets from within the teamserver’s dashboard, offering a high level of flexibility and adaptability.

Aggressor Script is a domain-specific scripting language based on another language, created in 2002 by Raphael Mudge, called “Sleep” [36]. It is a procedural scripting language heavily inspired by Perl syntax, designed to add scripting capabilities to Java – that is, writing source code at JVM-runtime that can use Java classes and functions. This approach is similar to a so-called Read-Eval-Print Loop (REPL), which Python uses to execute code, as well. Today, Sleep has many primitives of modern programming languages that are capable of:

- Working with integers, doubles, strings, arrays, hashmaps and structs (called “containers” in Sleep)
- Utilizing all common control structures
- Using complex paradigms, like closures and concurrency
- Handling Exceptions
- Accessing the entire Java API

The standard library also provides built-in functionality for string manipulation, basic input/output operations, Regular Expressions, and more [36]. Listing 2 lists an example of a function definition (using the `sub` keyword) and the corresponding function call that generates a random string of specified length, using only the Sleep standard library.

```
1 sub rand_string {
2   local('$length $charstr $out $i $index');
3   $length = $1;
4   $charstr = 'abcdefghijklmnopqrstuvwxy';
5   $out = '';
6   for($i = 0; $i < $length; $i++) {
7     $index = rand(strlen($charstr));
8     $out = $out . substr($charstr, $index, $index + 1);
9   }
10  return $out;
11 }
12
13 println("Random 20: " . rand_string(20));
```

Listing 2: Example Sleep script showing the random generation of a string with specified length [6]

The language exposes an extensive API that can be used to add functions to the Sleep interpreter, which is usually embedded in a Java application (called the “host application”). This is also how Cobalt Strike utilizes and implements the Sleep language: as of December 2024, Cobalt Strike provides 403 additional types and functions to the Sleep language [4], which culminates in Aggressor Script. These scripts are conventionally stored in files ending in `.cna`, and can be identified as such on GitHub or similar platforms: many scripts are openly shared and distributed, forming a shared knowledge base of enhancements and operational techniques. The functions added to Sleep ultimately allow interaction with Cobalt Strike’s core components. Through this API, operators can manipulate beacons, define custom commands, and respond to events within the framework. Once an Aggressor Script has been loaded and activated using the Script Manager, it can change the behavior of the operation in the following ways:

- **Add/Manipulate commands:** With the `command` and `alias` instructions, new commands can be added to the beacon console.
- **React to events:** One feature of Sleep is its event-driven architecture. With the `on` instruction, the language supports the definition of handlers for specific events, such as beacon check-ins, session terminations, or received messages. This design allows scripts to react dynamically to changes in the operational environment, automating responses like logging specific actions, notifying operators of key events, or executing predefined sequences of commands. Such responsiveness is critical for simulating advanced threat actors and testing an organization’s detection and response capabilities under realistic conditions. An exemplary script, that listens to all events, can be seen in Listing 3.
- **Set/Manipulate the Teamserver’s user interface:** In addition to its event-driven capabilities, Aggressor Script offers support for user interface customization using the `popup` instruction. Operators can use it to add new menus, dialogs, and even entirely new tabs to the Cobalt Strike interface. This functionality enables users to create a more intuitive or operation-specific interface, improving workflow efficiency during engagements. This feature can be compared with Mythic’s Browser Script feature.
- **Add keybindings:** Using the `bind` instruction, Aggressor Script can listen for keyboard events, which can trigger a user-defined routine.
- **Define custom output of events:** Using the `set` instruction, the format of an event, and its presentation to the user can be defined. There are many events one can choose from, allowing the configuration of nearly all important parts of the user interface.
- **Create custom reports:** Aggressor Scripts can also create new report types, which can be generated via the UI.

```
1 on * {
2   local('$handle $event $args');
3
4   $event = shift(@_);
5   $args  = join(" ", @_);
6
7   $handle = openf("events.txt");
8   writeb($handle, "[ $+ $event $+ ] $args");
9   closef($handle);
10 }
```

Listing 3: Example Aggressor Script listening for all beacon events [7]

Most of these functionalities are not relevant for this thesis. However, the addition or manipulation of commands is important: this function can potentially intervene directly in the behavior of the beacon and thus adapt and extend its functions after deployment. In the context of this thesis, this would therefore fit into the definition of a plugin system. To some extent, listening for events can also be considered as such: for example, the beacon can perform periodic tasks on the target, like collecting environmental variables. However, this task only describes a different use case, but does not differ in the implementation compared to self-defined commands. For the sake of simplicity, this function is therefore not considered further until Chapter 6.

Basically, Aggressor Script only runs on the teamserver and not on the beacon. Above all, this can also be used for writing custom modules that interact with other security tools, such as vulnerability scanners or password-cracking utilities. This interoperability creates a cohesive attack framework, allowing red teams to execute complex, multiphase attack chains without switching between disparate tools, thereby maintaining operational continuity. However, Cobalt Strike also provides functions that can change the behavior of the beacon directly. The most important function is `beacon_inline_execute`: it instructs a beacon to execute a so-called Beacon Object File (BOF). This function implements the transmission, execution and cleanup of it. A BOF is a small application that can access beacon functions at runtime using an API – the so-called “Beacon API”. BOFs are developed in C or C++ and are therefore, in theory, infinitely flexible and can add an arbitrary number of features to the beacon.

In conclusion, it can be said that BOFs are the approach that needs to be adopted in Mythic, or the Mythic Beacon, in order to achieve basic compatibility with the Cobalt Strike plugin system. Achieving BOF compatibility is a substantial issue of this thesis and will be further discussed in Chapter 3, 4 and 5. However, the big problem with its instrumentation method Aggressor Script – or rather the underlying Sleep language – is the fact that the Domain-Specific Language (DSL) embedded in it has so far only been implemented for use with Java [36]. Of course, this also makes sense with Cobalt Strike, as the team server is written in Java. For Mythic, however, which relies on Golang and Python, it is likely to be difficult to achieve compatibility with Sleep. Since many BOFs are instrumented with Aggressor Script, this could very well be a requirement to achieve general BOF compatibility. The implementation of such a compatibility level with Aggressor Script is discussed further in Chapter 6.

Scripting in Mythic: Mythic Scripting / Browser Script

As already mentioned, Mythic also has a dedicated scripting engine based on Jupyter, called “Mythic Scripting” [33]. It allows for intervention with the RabbitMQ message broker and the Mythic core to intercept communication between the components or add additional handlers. Either the GraphQL endpoint, which is also used by the Mythic frontend, can be queried, or RPC calls can be made directly to the Mythic core. This way, many server-side actions can be added in the first instance, such as the automated building of beacon payloads or the setting and customization of C2 profiles. The basic installation of Mythic also offers a small number of exemplary scripts that implement the aforementioned scenarios. However, this part of the functionality is not useful for achieving BOF compatibility, as the inner implementation of the beacon remains untouched. With Mythic Scripting it is possible to intervene in the tasking process of Mythic, i.e. to view messages to and from deployed beacons and to manipulate them if necessary. The enforcement of OPSEC is often cited as an example, such as checking transferred files to see whether they contain known virus signatures in order to enforce further obfuscations [37, Timecode 23:30]. The automated creation of subtasks is also mentioned, for example to create regular tasks or prepare payloads [37, Timecode 24:59]. However, these instructions are handled entirely via tasking.

Although this approach could easily be compared to Cobalt Strike’s Script Manager, Mythic Scripting seems to be intended for customizing and hooking existing functionality rather than extending it. The loose coupling between Browser Script (for displaying UI) and Mythic Scripting is also a disadvantage for the endeavor of mimicking a Script Manager. Typically, Mythic payloads define their own capabilities themselves, so that when it comes to achieving BOF compatibility, Mythic Scripting is unlikely to offer any significant added value compared to pure implementation in the beacon itself.

In contrast to this, an adaptation of Aggressor Script – should one want to adopt this in Mythic – could be implemented using Mythic Scripting: Aggressor Script expressions could be modeled and mapped using it, since the execution of server-side actions can also be realized with it. This has already been done manually in the predefined examples in the Mythic Scripting Engine: a relatively complex Aggressor Script for listing and tagging all important running processes on a target [38] was transferred to the Jupyter notebook `TagAllProcesses.ipynp` [39]. The Aggressor script uses the `BEACON_OUTPUT_PS` hook, while the Mythic Script marks processes recognized by the beacon via Mythic’s tagging system. This means that it is also architecturally intended to use Mythic Scripting for such tasks.

In conclusion, it can be said that Mythic Scripting offers no significant advantage in making BOFs available in the beacon. However, it can be used in Chapter 6 to make Aggressor Script usable in Mythic. It is therefore clear that the implementation of a BOF plugin system must mainly take place in the beacon. Adjustments to the transport of the beacon payload may be necessary, but these should only be seen as a technical necessity rather than main parts of the feature set of Mythic. Obfuscation measures must also be implemented mainly in the C2 profile and the translation container or by using Mythic Scripting.

2.4 Other C2 Frameworks and Beacons

In addition to Cobalt Strike and Mythic, several other C2 frameworks are widely used by cybersecurity professionals, red teams, and sometimes malicious actors. Each framework has unique features, strengths, and weaknesses, which make them suitable for different use cases. Below is an overview of some notable C2 frameworks:

- **Havoc:** Havoc is a modern C2 framework designed for red team operations. It is lightweight and highly customizable, with an emphasis on ease of deployment and operational security. Havoc is gaining popularity as an alternative to Cobalt Strike due to its simplicity and free availability [40]. It has many similarities with Mythic, including the operation of the dashboard and the creation of payloads. Havoc also has BOF execution support through a custom loader [15], which will be discussed later in Chapter 5.
- **Sliver:** Sliver is an open-source C2 framework created by Bishop Fox. It is written in Go, offering cross-platform compatibility and modern features. Sliver focuses on ease of use, operational security, and providing a robust alternative to commercial C2 frameworks like Cobalt Strike.
- **Empire:** Empire is a post-exploitation C2 framework designed for stealthy operations. Originally developed by Veris Group, it features a powerful PowerShell-based beacon and supports Python beacons for cross-platform operations. Empire also has BOF execution support through the “Invoke-Bof” PowerShell cmdlet project by Airbus CERT [41], [42].
- **Merlin:** Merlin is another open-source C2 framework, also written in Go, that only supports HTTPS-based communication. It is designed to provide a lightweight and stealthy alternative to commercial solutions. Merlin’s small payload size and cross-platform compatibility make it ideal for low-profile operations.
- **Brute Ratel:** Brute Ratel is a C2 framework following a very similar approach as Cobalt Strike. It has one proprietary beacon called “Badger” with a predefined set of features which offers a plugin system that relies on the same technique as Cobalt Strike, but using different APIs, internally called “Badger Object Files”, making them incompatible with Cobalt Strike. However, there is an implementation called “CS2BR” for using BOFs in Badger by adding a compatibility/translation layer to it [43].

Some available Mythic beacon implementations also have their own plugin systems – and some already contain BOF runtimes. However, due to the limitations mentioned above, they run completely standalone in the beacon and have no connection to Mythic-specific functionality, except for file transfer and the announcement of the callback command to the operator.

A Mythic Beacon that is very similar to cirosec’s ciroStrike is “Hannibal” [44]. It is a beacon written in C, which is designed as Position Independent Code (PIC) so that it can also be used as shellcode. However, Hannibal does not have a BOF runtime. Instead, the specially developed file format HBIN (“Hannibal Binary”) is used. It is in fact a position-independent binary format, which is built using a Makefile-based build system instead of a standard compiler. With the help of small assembly snippets and linker scripts, the format is instrumented so that it can be executed by Hannibal as it is [45]. This reduces the workload

for the beacon, as no complex runtime is required, but on the other hand imposes several restrictions on the development of HBINs: they must be developed completely position-independent, which makes heap allocations impossible, for example. This also makes it more difficult to use most existing libraries.

The “Apfell” beacon implementation takes a different approach. Apfell is a JavaScript for Automation (JXA) payload targeting macOS running Yosemite or later [46]. JXA is a feature with which applications can be controlled using JavaScript. The possibilities range from displaying simple notifications to executing shell scripts [47]. Apfell’s commands are designed in such a way that JavaScript can also be executed after deployment, and NPM packages can be installed later to extend the range of functions even further. Basically, this approach is only possible because of the way JXA works – so it is not transferable to other environments – but requires very few adjustments to the Beacon or Mythic.

Both of these beacons implement approaches that take place entirely in the beacon itself and have no API-averse dependencies on Mythic. This confirms the thesis from Chapter 2.3 that there are no suitable interfaces for the execution of loadable code from Mythic.

3 Examination of Beacon Object Files (BOFs)

This chapter examines the BOF file format in detail. First, the intended functions that BOFs should fulfill are defined. This specifically involves the interactions with Cobalt Strike, how these are implemented and what advantages and disadvantages they have in each case. Particular attention is paid to the differences between the loading of BOFs and the so-called Beacon API as well as the possible restrictions when building a custom implementation on top of it. The official instructions for creating a BOF are then followed in order to understand and explain them. The structure of the resulting Common Object File Format (COFF) format, which is created within the build process, is examined in detail to lay the foundations for the development of the runtime in Chapter 5. The advantages and disadvantages of the format and why it was chosen in Cobalt Strike are discussed.

3.1 Intended Functionality of BOFs

The primary use case of BOFs is easy to summarize: it is about creating programs or executable code to abuse found weaknesses after already having set foot in the target infrastructure. In the cyber kill chain, this step is called “Actions on Objectives” [18], which is often the last stage of an attack. This step is usually only achieved with a great deal of preparatory work and limited resources, as OPSEC also plays an important role: during an attack, it is imperative to maintain stealth in order to avoid detection. Therefore, obfuscated techniques are often used, such as shellcode injection.

In the example of the specific attack via a C2 infrastructure, the delivery step would be the first installation of the C2 beacon on a target device. Now, in the subsequent steps, the goal is to utilize this state, e.g., by ensuring persistence in the system, and to be able to react appropriately to the vulnerabilities already found in it. To do this, it must be possible to subsequently “teach” the beacon to exploit these vulnerabilities.

The Problem with Existing Approaches

C2 beacons, but also other tools used in offensive IT security, already include options for reloading program code. Cobalt Strike, for example, already has some commands to execute PowerShell commands on the target system built in. There are also more complex commands in some Mythic beacons that even allow entire PE files to be passed to the Windows API and executed by the operating system in this way. However, both approaches have a decisive disadvantage: they are conspicuous because they rely on dependencies outside the beacon process, the use of which is easily recognized by EDR software.

The execution of PowerShell commands requires an underlying PowerShell interpreter, which must be provided by the operating system. Even when launching executable programs such as PE files, the operating system must provide a runtime in a separate process. The creation of processes and threads is usually closely monitored and regulated by EDR software: a malicious program that disguises itself as calculator software, for example, will quickly attract attention if it starts PowerShell processes or attempts to run other executables. This attack technique is known as “fork and run” and describes the creation of an “auxiliary process” as a child process (“fork”), in the context of which the program to be loaded

is then executed (“run”). Fork and run has not been a viable solution in well-secured environments for some time now, as EDR software is developing rapidly in this area.

Furthermore, some of these methods have the disadvantage that some attacks rely on the file system – that is, the hard disk – to temporarily store payloads. Windows executables that have dynamic dependencies on system libraries can only be executed by the operating system if they themselves are stored on the hard disk. This is of course dangerous in covert operations, as EDR tools can easily detect malware on the hard disk.

This is exactly where BOFs come into play. They are designed in a way that they are not dependent on the fork and run pattern, but instead can be executed completely within the beacon process. Of course, this also has the advantage that they do not have to be stored on the hard disk at any time. BOFs are developed in C or C++ and are therefore theoretically unlimited in their range of functions. They can be created with the common Microsoft compilers as well as with the Linux cross-compiler toolchain MinGW by specifying some compiler flags (Chapter 3.3 deals with compilation in detail).

Restrictions

The use and development of BOFs is, however, also associated with some restrictions that must be considered. The following restrictions apply to the use and design of BOFs [48]:

- BOFs are not suitable for long-running tasks. As the beacon process is usually single-threaded and the BOF code is executed within the same process, the BOF would block the beacon’s execution no more commands could be received by the C2 server and processed.
- The C standard library is not available by default. This is due to the fact that the BOF format is an unlinked file format, as will be explained in the course of this chapter. Functions such as `strcmp` or `strcpy` must therefore either be implemented by the BOF developer itself or resolved manually.
- A crash of the BOF destroys the entire beacon. Since BOFs are executed in the same process as the beacon, program errors in the BOF also have a direct effect on the stability of the beacon. The BOF must therefore be able to handle all error cases, otherwise there is a risk of losing access to the beacon, and thus to the compromised target system.

There are also the following implementation-specific restrictions, which have been identified by Cobalt Strike and TrustedSec, among others [49]:

- No non-const global variables may be used during development. This is because the `.bss` section in the BOF format, which is responsible for saving variables of this type, cannot be loaded by the corresponding runtime. Details on the structure and limitations of those sections are explained in Chapter 3.4.
- Complex multiplications and divisions that use the `long long` data type for the operands must not be performed if the BOF was built for the x86 architecture (32 bit Windows).
- Some Win32 APIs cannot be accessed nor called. This is because they sometimes use several process heaps, which the runtime cannot allocate.

- No more than 4 MB of memory can be allocated on the process stack at once. This is a limitation of Windows, not of BOFs, as this limitation is automatically circumvented by the compiler in “normal” programs. This does not apply for the dynamically loaded allocation functions used by BOFs.

These restrictions must always be taken into account when designing and developing a BOF. Carelessly constructed BOFs otherwise increase the risk of destroying and losing the beacon. A resilient beacon infrastructure and safe BOFs are vital for the success of C2 operations.

Quality of Life Features

To simplify the interaction with the beacon and the C2 infrastructure, or to enable it in the first place, Cobalt Strike provides the so-called beacon APIs for BOFs. They are made available to the beacon developer as a header file and can then be called in the BOF like normal C/C++ functions, for example to send output to the C2 server, to persist data in the beacon’s memory or to use predefined functions for process injection. The beacon API is further described and examined in the following Chapter 3.2.

As already mentioned in the restrictions, a BOF does not have access to the C standard library. Instead, the Windows API can be used. It is, however, very limited by default within BOFs for OPSEC reasons – only `GetProcAddress`, `LoadLibraryA`, `GetModuleHandle`, and `FreeLibrary` are available to at least load additional API functions [50]. The runtime of the Cobalt Strike beacon can be instructed to load additional APIs using these four functions. Alternatively, they can be declared in the BOF according to a convention called Dynamic Function Resolution (DFR) [50]. It signals to the beacon that the BOF requires additional Win32 APIs, which are then loaded and dynamically linked before the BOF is executed. By convention, a DFR declaration must consist of the exact same function name as specified by Microsoft and must have the following function signature: `<LibraryNameUppercase>$<FunctionName>`. An example of the use of DFR is documented in Listing 4.

```

1  #include <Windows.h> // only used to import type definitions
2
3  // DFR declaration for "DsGetDcNameA"
4  DWORD WINAPI NETAPI32$DsGetDcNameA(LPVOID, LPVOID, LPVOID,
5  LPVOID, ULONG, LPVOID);
6
7  // DFR declaration for "NetApiBufferFree"
8  DWORD WINAPI NETAPI32$NetApiBufferFree(LPVOID);
9
10 // BOF entry point
11 void go(char *args, int alen) {
12     PDOMAIN_CONTROLLER_INFO pdcInfo;
13     NETAPI32$DsGetDcNameA(NULL, NULL, NULL, NULL, 0, &pdcInfo);
14     // e.g. do something with the domain name (pdcInfo->DomainName)
15     NETAPI32$NetApiBufferFree(pdcInfo);
16 }
```

Listing 4: Using `DsGetDcNameA` and `NetApiBufferFree` from `netapi32.dll` via DFR

Ultimately, it is precisely these properties and intended functionality that need to be reproduced. The following chapters deal with the technical requirements for the features mentioned here and the execution of BOFs itself, explained on the basis of the file format.

3.2 Examination of the Beacon APIs

This section describes the beacon APIs mentioned in quality of life features in detail. The beacon APIs are provided to the beacon developer as a header file – usually as `beacon.h`. In the context of this thesis, the `beacon.h` delivered with Cobalt Strike version 4.10 is consulted [51]. The contents of this file is listed in Appendix B, where the APIs are grouped using code comments for reference. This grouping can only be found in this file and therefore appears to be unofficial or mostly undocumented. The groupings that can be found in the official documentation differ from the categorization made in `beacon.h`, as well as within the documentation pages themselves. In the context of this thesis, the categorization according to the code comments in `beacon.h` is assumed as a reference, as this file is the one used for BOF and runtime development.

This results in the following categorization of the beacon APIs, which are referred to below as beacon API groups. Table 1 lists these groups, along with a brief description of their functionality.

Beacon API group	Short description
Data Parser API	Reads the parameters passed to the BOF at invocation
Format API	Utility functions to help with formatting strings
Output API	Sends output to the C2 server
Token API	Can manipulate the beacon's current thread token
Spawn+Inject API	Leverages some of the beacons process injection capabilities
Utility API	Several utility functions
Key/Value Store API	Gives access to a minimal key/value store within the beacon's memory
Data Store API	Data store with the ability to obfuscate the stored data at runtime
User Data API	Retrieves the Beacon User Data (BUD) buffer when using a UDRL
Syscall API	Macros that call several Syscall functions resolved by the beacon
Beacon Gate API	Enables/Disables Cobalt Strikes BeaconGate feature

Table 1: List of beacon API groups with a short description

The API groups are described in detail in the following and the functions they contain are given [51]. It is determined which of the functions are relevant for this thesis. The relevance is mainly determined based on the dependency on Cobalt Strike-proprietary functions, as some of them are not applicable or not useful when used in a Mythic beacon. For the more complex API groups code examples are also listed.

Data Parser API

The Data Parser API is used to extract arguments given to the BOF at invocation from Cobalt Strike. They are usually serialized (called “packing” in Cobalt Strike) with Aggressor Script’s `&bof_pack` function [4] using a format string. The arguments are passed as a size-prefixed binary blob which must be deserialized first using this API. BOFs that accept arguments must at least use the following parts from the Data Parser API group:

- The `datap` struct definition, representing the parser instance.
- `void BeaconDataParse(datap *parser, char *buffer, int size)` for preparing the parser, with `buffer` being the arguments buffer passed to the BOF entry point function.
- `char *BeaconDataExtract(datap *parser, int *size)` for reading strings and wide strings.
- `int BeaconDataInt(datap *parser)` for reading 4 byte integers.
- `short BeaconDataShort(datap *parser)` for reading 2 byte integers (short).
- `char *BeaconDataPtr(datap *parser, int size)` for reading a pointer from the current parsing position. *(The documentation recommends using the `BeaconDataExtract` function instead, as `BeaconDataPtr` only extracts a direct buffer without size prefix. Usually, all `char*` parameters are passed with a size prefix.)*
- `int BeaconDataLength(datap *parser)` to get the amount of data left to parse.

To better understand the data parser API, Listing 5 shows an example of a BOF that takes two strings and a short. Listing 6 shows the corresponding Aggressor Script that uses the `&bof_pack` function to prepare the arguments for this BOF and adds the command `some_command` to the Cobalt Strike console.

```

1  #include "beacon.h"
2
3  void go(char *args, int alen) {
4      datap parser;
5      char *arg1;
6      char *arg2;
7      short arg3;
8
9      BeaconDataParse(&parser, args, alen); // initialize parser
10     arg1 = BeaconDataExtract(&parser, NULL); // get first arg (string)
11     arg2 = BeaconDataExtract(&parser, NULL); // get second arg (string)
12     arg3 = BeaconDataShort(&parser); // get third arg (short)
13
14     BeaconPrintf(CALLBACK_OUTPUT, "Args: %s, %s, %d", arg1, arg2, arg3);
15 }

```

Listing 5: Example BOF that reads three arguments and prints them

```

1 alias some_command {
2     local('$barch $handle $data $args');
3
4     # read in the BOF file with the correct architecture
5     $barch = barch($1);
6     $handle = openf(script_resource("some_command. $+ $barch $+ .o"));
7     $data = readb($handle, -1);
8     closef($handle);
9
10    # pack arguments ($1 = beacon ID, "zsz" = format string)
11    $args = bof_pack($1, "zsz", "Hello", "World", 13);
12
13    # execute it
14    beacon_inline_execute($1, $data, "go", $args);
15 }

```

Listing 6: Example Aggressor Script that adds a command with three arguments to Cobalt Strike

The following Table 2 lists the constants for the format string of the `&bof_pack` function [4]. This defines the data types that can then be resolved by the Data Parser API. A format string of "zsz", for example, means "two strings (z) and one short (s)".

Constant	Represented type	Data Parser API function
b	binary data	BeaconDataExtract
i	4-byte integer	BeaconDataInt
s	2-byte integer	BeaconDataShort
z	zero-terminated + encoded string	BeaconDataExtract
Z	zero-terminated + wide-char string	(wchar_t*)BeaconDataExtract

Table 2: Constants for the `&bof_pack` format string and the corresponding unpacking functions [4]

The problem with the approach of using this proprietary size-prefixed binary blob is that its entries are platform-dependent. This is abstracted by the `&bof_pack` Aggressor Script function, but not by the `inline-execute` command of Cobalt Strike. Accordingly, it is neither trivial nor recommended executing BOFs directly with `inline-execute`, because the preparation of arguments works differently depending on the architecture. Especially the distinction between UTF-8 and UTF-16 strings can go wrong here and endanger the stability of the beacon. This complicates the testing of directly executable BOFs within Cobalt Strike. The current state of Mythic, on the other side, only allows for direct BOF execution, as there is no Script Manager for loading and parsing Aggressor Scripts. This issue will be an important consideration of the implementation done in Chapter 6.

This API group is essential as it is the only way for BOFs to accept arguments. Most BOFs are parameterized or at least parameterizable, meaning that implementing this API group is vital.

Format API

The format API is used to build large or repeating strings for output. It helps with allocating memory for strings and simplifies formatting, as this is not trivial within BOFs. Syntactically, it works similar to the `printf` from the standard library. As in the Data Parser API, there is a dedicated struct definition `formatp`, which is used to manage memory and to keep the state of the current allocation. The following functions are provided to format the string allocation:

- `void BeaconFormatAlloc(formatp *obj, int maxsz)` for allocating new memory and to initialize the `formatp` instance.
- `void BeaconFormatAppend(formatp *obj, char *data, int len)` to append data to the `formatp` instance.
- `void BeaconFormatFree(formatp *obj)` to free the memory and deinitialize the instance.
- `void BeaconFormatInt(formatp *obj, int val)` to append a 4 byte integer (big endian) to the instance.
- `void BeaconFormatPrintf(formatp *obj, char *fmt, ...)` to add a formatted string to the instance.
- `void BeaconFormatReset(formatp *obj)` to reset the `formatp` instance to its default state.
- `char *BeaconFormatToString(formatp *obj, int *size)` to extract the formatted data into a single string.

By definition, this API group contains helper functions. It is therefore not absolutely necessary for the development of BOFs, as these can also fall back on only minimally more complex alternatives. It is therefore necessary to identify, how commonly this API group is used in order to assess the importance of it.

Output API

The Output API returns output to the C2 server (i.e. Cobalt Strike) through the C2 profile. This is probably the most important API because it is the only way to see any results from BOFs. It allows displaying messages as informational and as errors using the `type` parameter. The following functions are available:

- `void BeaconOutput(int type, char *data, int len)` to send a plain string to the operator.
- `void BeaconPrintf(int type, char *fmt, ...)` to send a formatted string to the operator.

The `type` parameter is defined using the following constants to denote the type or format of message:

- `CALLBACK_OUTPUT`: Generic output, that will get converted to UTF-16 using the target's default character set.
- `CALLBACK_OUTPUT_OEM`: Generic output, that will get converted to UTF-16 using the target's character set. This is usually not required unless output directly from `cmd.exe` is used.

- `CALLBACK_OUTPUT_UTF8`: Generic output, that will get converted to UTF-16 from UTF-8.
- `CALLBACK_ERROR`: Same as `CALLBACK_OUTPUT`, but denotes an error message.

Internally, the `BeaconPrintf` function likely uses the same logic as the one used in the Format API, which is why the latter is treated as a direct dependency of the Output API. This classifies the Format API as important, contrary to the previous statement.

Token API

The Token API is used to set, drop or manipulate the thread token used in the current beacon context. It can also be used to check if the current process runs with administrative privileges using the following functions:

- `BOOL BeaconUseToken(HANDLE token)` is used to apply the specified token handle to the beacons current thread token and returns `TRUE` if the operation was successful.
- `void BeaconRevertToken()` is used to drop the current thread token and cleans up state information about it.
- `BOOL BeaconIsAdmin()` returns `TRUE` if the beacons process has a high-integrity context.

This API group also consists only of helper functions. However, these are comparatively complex functions for which it is practical if they are already offered by the BOF runtime. Nevertheless, the question remains whether this API group is used or if BOF developers usually rely on other known techniques for token manipulation.

Spawn+Inject API

The Spawn+Inject API contains several functions to spawn temporary processes and to inject payloads into remote processes. These functions are part of a feature called “Session Passing”: it allows the beacon to either inject shellcode into existing processes or create a temporary process itself for it. These functions are included in this API group:

- `void BeaconGetSpawnTo(BOOL x86, char *buffer, int length)` populates the passed buffer value to contain the so-called “spawnto value”. This value is set via the `spawnto` command from the Cobalt Strike console. It sets the path that is to be used by several other commands that rely on injections, namely `execute-assembly`, `psinject`, `powerpick`, and others. If the command has not been called yet, the default value from the listener configuration the beacon is configured to is used. The default value for x64 operations is `C:\Windows\System32\rundll32.exe`.
- `void BeaconInjectProcess(HANDLE hProc, int pid, char *payload, int p_len, int p_offset, char *arg, int a_len)` is used to inject a payload into an already existing process on the system, specified by its process ID (`pid`).
- `BOOL BeaconSpawnTemporaryProcess(BOOL x86, BOOL ignoreToken, STARTUPINFO *si, PROCESS_INFORMATION *pInfo)` is used to spawn a temporary process. The `pInfo` variable is populated with the handle of the new process that can be used for injecting shellcode into it.

- `void BeaconInjectTemporaryProcess(PROCESS_INFORMATION *pInfo, char *payload, int p_len, int p_offset, char *arg, int a_len)` is used to inject shellcode into a process created by `BeaconSpawnTemporaryProcess`.
- `void BeaconCleanupProcess(PROCESS_INFORMATION *pInfo)` is used to clean up a temporary process correctly. It closes all process handles associated with the specified process.

To better understand how these functions are used, Listing 7 contains an example that reads and outputs the `spawnto` value first, then creates a temporary process and injects shellcode into it. Finally, the temporary process is cleaned up.

```

1  #include "beacon.h"
2
3  void go(char *args, int alen) {
4      // get and print spawnto value
5      char spawnto[60];
6      BeaconGetSpawnTo(FALSE, &spawnto, 60);
7      BeaconPrintf(CALLBACK_OUTPUT, "spawnto: %s\n", spawnto);
8
9      // create temporary process
10     // (first, initialize STARTUPINFO and PROCESS_INFORMATION structures)
11     STARTUPINFO si;
12     ZeroMemory(&si, sizeof(si));
13     si.cb = sizeof(si);
14
15     PROCESS_INFORMATION pi;
16     ZeroMemory(&pi, sizeof(pi));
17
18     if (!BeaconSpawnTemporaryProcess(FALSE, FALSE, &si, &pi)) return;
19
20     // inject some shellcode into it
21     char payload[255] = /* some payload */;
22     BeaconInjectTemporaryProcess(&pi, payload, 255, 0, NULL, 0);
23
24     // cleanup
25     BeaconCleanupProcess(&pi);
26 }
27

```

Listing 7: Using the Spawn+Inject beacon API to inject shellcode from a BOF

The Spawn+Inject API group offers complex functions for injection, which is a generic red teaming technique. This makes BOF development much easier, but the question remains whether this API group is used by developers.

Utility API

The Utility API only has a single function:

- `BOOL toWideChar(char *src, wchar_t *dst, int max)` is used to copy a string to a UTF16-LE wide-character string buffer.

It is unclear why such a simple function and the corresponding API group even exists, since the functionality can easily be replicated in other ways using plain C. However, since there are no external dependencies on C2 servers or beacon internals, this function can be easily reimplemented.

Key/Value Store API

The Key/Value Store API is a simple way to store generic data in form of memory addresses within the beacons memory. Subsequent BOF invocations can then retrieve these addresses by key to get the data.

- `BOOL BeaconAddValue(const char *key, void *ptr)` adds the pointer contained in `ptr` as key.
- `void *BeaconGetValue(const char *key)` returns the pointer stored in key.
- `BOOL BeaconRemoveValue(const char *key)` deletes the pointer in key. Returns `FALSE` if the key does not exist and `TRUE` if the deletion was successful.

The interface of this API group looks very straightforward at first glance, since it mimics the common dictionary data structure. The greater difficulty, however, is the memory management: the key/value store has a lifetime that is linked to the runtime of the beacon, not to the runtime of the BOF. If a BOF invocation, for example, creates the key called “temp”, each subsequent BOF invocation – regardless of whether it is the same BOF or not – must be able to find it again. The key/value store must therefore be stored in the heap of the beacon. However, such an implementation is definitely possible and should be straightforward, given the generic nature of the interface.

Data Store API

The Data Store API is used to access and store data in the beacon process. Data is accessed by index and can be obfuscated and de-obfuscated. Compared to the Key/Value Store API, the data store contains actual copies of the data instead of just pointers. Furthermore, indices are used instead of keys. The following functions are available to interact with the data store:

- `PDATA_STORE_OBJECT BeaconDataStoreGetItem(size_t index)` retrieves the data at index. Data is returned as a struct, which contains the following information:
 - Type of the item: `DATA_STORE_TYPE_EMPTY` or `DATA_STORE_TYPE_GENERAL_FILE`.
 - Hash of the item.
 - Whether the item is currently masked (obfuscated) or not; if it is masked, `BeaconDataStoreUnprotectItem` must be called first.
 - The actual data buffer and its size.

- `void BeaconDataStoreProtectItem(size_t index)` manually protects (obfuscates) the item at `index` in memory.
- `void BeaconDataStoreUnprotectItem(size_t index)` manually unprotects (deobfuscates) the item at `index` in memory.
- `size_t BeaconDataStoreMaxEntries()` the maximum number of items that can be stored in the data store.

The interface is kept very slim as well. The problem with this API group is the masking feature: it is unclear how exactly the in-memory obfuscation works and how effective it is. However, this is an abstracted implementation detail and irrelevant for the interface, that is used by the BOF. Implementation is thus possible. Together with the Key/Value Store API, this API group is important to enable subsequent BOF invocations to communicate with each other.

User Data API

In case of this API group, “User Data” refers to the BUD feature of Cobalt Strike. This is a 32-byte sized buffer by default, which can be created by a User-Defined Reflective Loader (UDRL) and transferred to the beacon at loading time. UDRLs is a feature that is used to further obfuscate the beacon by placing such a loader in front of it. The user data buffer can be read with the single function contained in this group:

- `char *BeaconGetCustomUserData()` reads the BUD buffer. It always returns a valid pointer, which can, however, be all zeros if not set by the UDRL.

Since it is not intended to support UDRL in this thesis, this API group is therefore no longer considered important.

System Call (Syscall) API

The Syscall API provides access to pre-resolved function addresses and System Call numbers. This information is either resolved by the Beacon or is provided through a UDRL using the `USER_DATA` structure. This information can be useful for implementing custom System Calls within a BOF. The following structs are provided:

- `SYSCALL_API_ENTRY` containing the function address, jump address and Syscall number per Syscall.
- `SYSCALL_API` containing `SYSCALL_API_ENTRY`s for all supported Syscalls.
- `RTL_API` containing additional Run Time Library (RTL) addresses used to support Syscalls. If they are not set, the beacon uses the standard Windows API.
- `BEACON_SYSCALLS` combining the `SYSCALL_API` and `RTL_API`.

There is one function that is used to initialize the Syscall API:

- `BOOL BeaconGetSyscallInformation(PBEACON_SYSCALLS info, BOOL resolveIfNotInitialized)` resolves (or only gets, if `resolveIfNotInitialized` is `FALSE`) all Syscall addresses and stores them in the `info` struct.

All other functions are mappings/shortcuts to Windows API functions. They use the structs and data types defined in the `Windows.h` header file. In Cobalt Strike, they may use different Syscall methods depending on whether BeaconGate is configured and enabled or if the `RTL_API` struct is configured. The function signatures match the ones in `Windows.h`, but are prefixed with `Beacon`:

- `LPVOID BeaconVirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect)`
- `LPVOID BeaconVirtualAllocEx(HANDLE processHandle, LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect)`
- `BOOL BeaconVirtualProtect(LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect)`
- `BOOL BeaconVirtualProtectEx(HANDLE processHandle, LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect)`
- `BOOL BeaconVirtualFree(LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType)`
- `BOOL BeaconGetThreadContext(HANDLE threadHandle, PCONTEXT threadContext)`
- `BOOL BeaconSetThreadContext(HANDLE threadHandle, PCONTEXT threadContext)`
- `DWORD BeaconResumeThread(HANDLE threadHandle)`
- `HANDLE BeaconOpenProcess(DWORD desiredAccess, BOOL inheritHandle, DWORD processId)`
- `HANDLE BeaconOpenThread(DWORD desiredAccess, BOOL inheritHandle, DWORD threadId)`
- `BOOL BeaconCloseHandle(HANDLE object)`
- `BOOL BeaconUnmapViewOfFile(LPCVOID baseAddress)`
- `SIZE_T BeaconVirtualQuery(LPCVOID address, PMEMORY_BASIC_INFORMATION buffer, SIZE_T length)`
- `BOOL BeaconDuplicateHandle(HANDLE hSourceProcessHandle, HANDLE hSourceHandle, HANDLE hTargetProcessHandle, LPHANDLE lpTargetHandle, DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwOptions)`
- `BOOL BeaconReadProcessMemory(HANDLE hProcess, LPCVOID lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize, SIZE_T *lpNumberOfBytesRead)`
- `BOOL BeaconWriteProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPCVOID lpBuffer, SIZE_T nSize, SIZE_T *lpNumberOfBytesWritten)`

The Syscall API group is certainly useful in Cobalt Strike because it can use the various Syscall methods that come through UDRL and BeaconGate [52], meaningfully abstracting them away from the BOF

developer. In Mythic, however, this approach makes little sense, since each Mythic beacon can determine for itself how functions and Syscalls are resolved and called. In most cases, there is only a single method, as is the case with `ciroStrike`. Accordingly, there are no advantages of the Syscall API over DFR, since with the latter these functions can also be resolved and used. How these functions are resolved is determined by the BOF runtime itself in the approach taken in this thesis. The implementation of this API group is therefore of low priority.

Beacon Gate API

`BeaconGate` is a feature introduced in Cobalt Strike 4.10 to make it more difficult to detect anomalous API calls. It allows beacons to execute Windows API calls via the `Sleepmask` BOF, increasing control over these calls. This allows techniques such as call stack spoofing to be used and beacons to be masked during potentially suspicious API calls, making detection by security solutions more difficult [52]. This API group allows BOFs to enable or disable `BeaconGate` using the following functions:

- `VOID BeaconEnableBeaconGate()` to enable `BeaconGate`.
- `VOID BeaconDisableBeaconGate()` to disable `BeaconGate`.

`BeaconGate` is not discussed further in this thesis, as it serves no purpose in Mythic. Mythic beacons don't use a standardized way to obfuscate function calls.

It is difficult to overlook when examining the beacon API that many of the functions offered are only helpers, but some of them encapsulate very complex functionality. The problem here is that this functionality is provided by the closed source implementation of the Cobalt Strike beacon, so it is not clear to the developer what exactly happens when the functions are called. This can sometimes be an issue, especially in very OPSEC-critical scenarios. It is therefore reasonable to assume that BOF developers will not use the offered beacon API in such scenarios and instead use their own techniques and implementations.

In order to correctly determine the importance of these API groups, it is therefore important to establish whether BOF developers use them at all or not. This assessment is important because the implementation of many of these groups can be very complex and therefore time-consuming. This matter is examined in Chapter 4 using a collection of exemplary BOFs.

3.3 Compilation Instructions

This section examines the official instructions for creating (compiling) a BOF in order to understand how they are structured. For this, the minimal example given in the documentation is conducted, which is listed in Listing 8. The example uses the `BeaconPrintf` function from the Beacon Output API to send “Hello World” and the raw argument string to the operator as soon as the BOF is executed.

```
1 #include <windows.h>
2 #include "beacon.h"
3
4 void go(char *args, int alen) {
5     BeaconPrintf(CALLBACK_OUTPUT, "Hello World: %s", args);
6 }
```

Listing 8: Minimal example of a BOF printing Hello World and the raw argument string [8]

The documentation gives three possibilities to build this minimal example (`hello.c`) as a BOF (`hello.o`) [8]:

- With Microsoft’s own compiler Microsoft Visual C++ (MSVC):
`cl.exe /c /GS- hello.c /Fo hello.o`
- With the cross-compiler MinGW for Linux for the x86 architecture:
`i686-w64-mingw32-gcc -c hello.c -o hello.o`
- With the cross-compiler MinGW for Linux for the x64 architecture:
`x86_64-w64-mingw32-gcc -c hello.c -o hello.o`

MSVC is used when the BOF is developed on Windows. MinGW is a collection of cross-compilers that can generate Windows binaries on Linux. The resulting compilation `hello.o` shall then in theory be passed to Cobalt Strike’s `inline-execute` command together with any arguments: (`inline-execute /path/to/hello.o arg1 arg2 ...`) or invoked via the Aggressor Script `&beacon_inline_execute` function [51].

When examining the compiler invocations, there are the following similarities: the input file `hello.c` is passed to the compiler as a positional argument. The output file is specified with the `/Fo` option in MSVC or the `-o` option in MinGW. The `/GS-` flag of MSVC is used to deactivate the buffer overflow protection – a corresponding counterpart is missing in the MinGW calls, presumably because the MSVC does not behave in accordance with the standard here and this protective measure leads to errors when generating BOFs. This leaves the `/c` and `-c` flags: they instruct the compilers to call only the compilation step, but not the linker afterwards.

When the linking step is left out, the compiler produces a so-called object file (e.g., `.o` or `.obj`) from the source code instead of a runnable program. Although this file contains the translated machine code, it does not yet contain a complete execution environment – in particular, there are no references to external libraries and functions. Their pointers are not yet filled with actual addresses – one of the tasks of the

linker is to resolve them. This also has the effect that there can always be exactly one object file (.o or .obj file) per translation unit¹: linking several object files together is also a task of the linker. The linker also provides the entry point for the executable so that the operating system knows where to begin running it. A simplified compilation process, where the linking step is missing in this case, is shown in Figure 5.

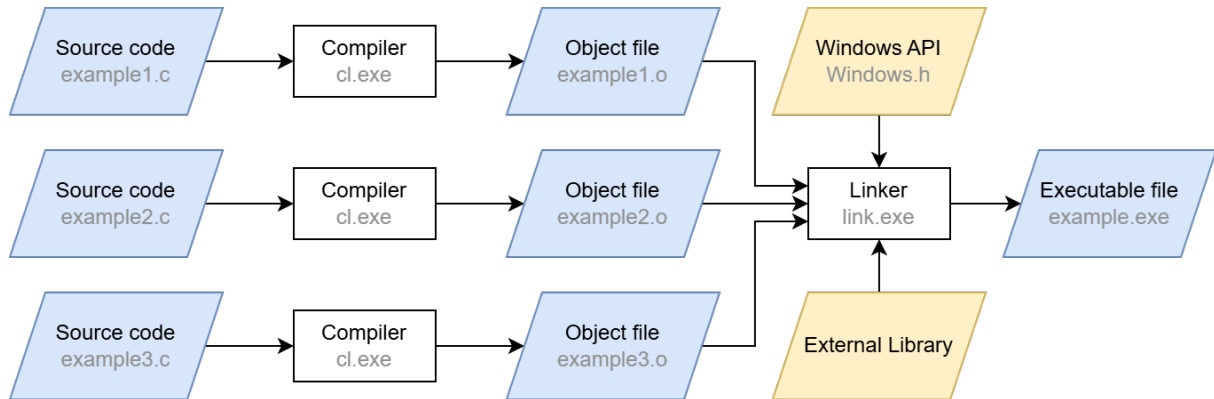


Figure 5: Simplified illustration of a full compilation process on Windows (own illustration)

When targeting Linux, these object files are saved in the Executable and Linking Format (ELF) format just like linked, executable files. For Windows, there is usually a separate format used called Common Object File Format (COFF). Since BOFs are targeting Windows, COFFs are the ones generated by the compilation instructions provided by Cobalt Strike. In the following chapter, this format is examined in detail in order to understand how this non-executable format can be processed in order to be used as an executable BOF.

3.4 Understanding the COFF File Format

The Common Object File Format (COFF) is a file format that is used in Microsoft Windows and sometimes other operating systems as a format for object files. It provides a structured way to represent machine code and data, along with metadata required for linking and execution. As already mentioned in the previous section, BOFs are merely machine code in the COFF format. It is therefore important to understand the format in order to allow its execution. This section delves into the technical aspects of the COFF file format, explaining its structure, purpose, and internal workings.

The COFF file format was originally developed for Unix systems but was later adopted and modified by Microsoft for Windows [53]. It serves as a foundation for PE – the format for executables and DLLs used within Windows. As already mentioned, it is an intermediate unit right before PE where the linker has not yet engaged. Therefore, a COFF – and thus a BOF – is the result of a usual program compilation but without the linking part. As a result, COFF files must hold metadata for the linker, as it is intended that the linker will later process them into an executable. These two types of metadata are of highest relevance for the linker:

¹A translation unit is a single C/C++ source code file after precompilation

- **Symbols:** Symbols are means of identifying variables and functions. In COFFs, they contain the variable or function name and the memory address where they are used. The symbols are stored in a symbol table, which is covered later in detail.
- **Relocations:** Relocations describe how a linker must resolve a symbol in order to link it correctly. This primarily concerns the format of the target address, for example, whether it must be specified relative to the start of the file or to the section.

Due to this metadata, the COFF format is more verbose and contains more debugging information, but still remains smaller than a PE file as most external implementations and Operating System (OS) specifics to run it are not yet included. This usually results in file size savings between 65 and 90 percent, mostly depending on the proportion of external symbols, compared to a linked PE file. These numbers were recorded while testing the BOFs collected in chapter Chapter 4: the object file created from an experimental implementation of beacon APIs was linked with the BOFs itself to create a standalone executable (PE). This way, several size comparisons for several BOF implementations could be made.

A COFF file consists of several sections, each serving a specific purpose. Below is a breakdown of its components in the order that they appear in memory [3]:

File Header (fixed size of 20 bytes)

The file header is the first structure in a COFF file [54, section “COFF File Header”]. It provides general information about the file, such as:

- **Machine Type:** Specifies the target architecture (e.g., x86, x64).
- **Number of Sections:** Indicates how many sections are present in the file.
- **Time Stamp:** The time when the file was created.
- **Pointer to Symbol Table:** A file offset pointing to the symbol table.
- **Number of Symbols:** Number of symbols present in the symbol table.
- **Size of Optional Header:** Indicates the number of bytes of the optional header.
- **Characteristics/Flags:** Flags indicating file attributes (e.g., executable, debug information present).

The file header is always the entry point when parsing a COFF file, because with the help of the pointers, offsets and size specifications contained in it, every location in the file can be accessed from it. Specifically, the “Pointer to Symbol Table” and “Number of Symbols” can be used to address each symbol, since each symbol entry has a fixed size. The size of the file header and the optional header can also be used to access the first section header. With the help of the size of the section given there and the “Number of Sections” field in the file header, each section can also be accessed.

The “Characteristics” field can be used to determine the COFF variant. This field is mainly used to store information for images, but also other important debug information that may be relevant for the linker. If the image flags do not exist, it can be assumed that the file is indeed an object file.

The following flags are intended for images and must not be set for object files [54, section “Characteristics”]:

- `IMAGE_FILE_RELOCS_STRIPPED`: Flag for images only; indicates that the file does not contain base relocations and must therefore be loaded at its preferred base address.
- `IMAGE_FILE_EXECUTABLE_IMAGE`: Flag for images only; indicates that this file is a valid executable.
- `IMAGE_FILE_DLL`: This image is a DLL.

Optional Header (dynamically sized)

For executables and DLLs (so-called “PE images”), an optional header follows the file header. Its content and fields differ depending on the COFF variant used. For executable COFFs or PE files (which use the COFF header format), it contains the following fields:

- **Entry Point Address**: Address of the entry point where execution starts.
- **Image Base**: Preferred address of the first byte of the image in memory.
- **Section Alignment**: Alignment of sections in memory.
- **Subsystem**: Defines the required subsystem (e.g., console or GUI).
- **DLL Characteristics**: Flags specific to DLLs (e.g., dynamic base, NX compatibility).

The optional header is, as the name implies, optional. It is specifically nonexistent in object files, which are the result of a compilation before linking. This means that this header is not relevant in this thesis, as BOFs do not contain it. The “Size of Optional Header” field in the main file header is thus set to zero, which can be an indicator to check for correctness of file input in the BOF runtime, to ensure beacon stability.

Section Header (fixed size of 40 bytes)

Each section header describes a distinct section of the file. Sections may contain code, data, or other raw resources. Key fields of each section header include:

- **Name**: A short identifier for the section (e.g., `.text`, `.data`).
- **Virtual Size**: Size of the section in memory.
- **Virtual Address**: Memory address where the section will be loaded.
- **Size of Raw Data**: Size of the section’s data in the file.
- **Pointer to Raw Data**: File offset (relative to the start of the file) to the section’s data.
- **Pointer to Relocations**: File offset (relative to the start of the file) to the relocation entries of that distinct section.
- **Pointer to Line Numbers**: File offset (relative to the start of the file) to the line number entries. This value is deprecated and should always be set to zero.
- **Number of Relocations**: Number of relocation entries for this section. This is non-zero for non-linked files.

- Number of Line Numbers: Number of line number entries. This value is deprecated and should always be set to zero.
- Characteristics: Flags describing section properties (e.g., executable, readable, writable).

The section header, like the file header, is very important in BOFs. In addition to the usual information such as the size of the corresponding section, it also contains the so-called relocations. These are particularly important for the development of a COFF loader, since they mark the memory positions within the section where unresolved symbols are located. Symbols are used to abstractly denote variables, functions, but also cross-referencing data such as string constants. Since the linker has not yet been applied to the file, these symbols have not been set correctly, and it these particular locations that are marked by the relocations. They are usually only resolved once the final memory layout is known. Relocations are handled more in detail in the “Relocation Entries” section of this chapter and further during development of the runtime in Chapter 5.

The line numbers are not discussed further here. They are intended for debugging purposes only and are deprecated and therefore usually set to zero.

Within this header, the “Characteristics” field indicates the authorizations and properties of the section. For example, sections containing code must include `IMAGE_SCN_CNT_CODE` flag to mark the contained code as executable [54, section “Section Flags”]. Anomalies of these flags are very frequently and reliably used by EDR as an IoC.

Section Content (dynamically sized)

This is the actual raw content of a section. It varies depending on the purpose it. These are some common sections and their uses:

- `.text` Section: Contains executable code.
- `.data` Section: Holds initialized global and static variables.
- `.bss` Section: Contains uninitialized data, typically zeroed at runtime.
- `.rdata` Section: Stores read-only data like string literals and constants.
- `.debug` Section: Holds debugging information.

Executable code is usually stored in the `.text` section. One might therefore assume that processing the relocations of this section would suffice to run the code. However, this is not correct. Each of the sections has a section header with a fixed size, which means that each section can also have a pointer to and a quantity of relocations. In fact, not only the `.text` section has relocations, but potentially every other section as well: if, for example, a string constant is used in any of the sections – regardless of whether it is used as a local variable (`.text`) or as a global, mutable variable (`.bss`) – this string is probably stored in the `.data` or `.rdata` section. The sections in which this string is used must therefore have at least one relocation to one of the data sections. It is therefore important to edit the relocations of all sections and not just those of the `.text` section.

Symbol Table (dynamically sized, entries have 18 bytes)

The symbol table provides metadata for symbols used in the file. For example, if the function `int add(int a, int b)` is defined in this file, it is represented as the symbol `add` in this table. The table itself can have any number of entries and therefore has an indefinite size. However, the entries themselves are always 18 bytes in size and contain the following information:

- **Symbol Name:** The name of the symbol (e.g., function names, variable names).
- **Symbol Value:** Address or offset of the symbol.
- **Section Number:** Indicates which section the symbol belongs to.
- **Type:** Specifies the symbol type (e.g., function, object).
- **Storage Class:** Defines the scope and visibility of the symbol (e.g., external, static).
- **Auxiliary Entries:** Additional information, such as line numbers for debugging.

When developing a loader, symbols are not accessed directly by iterating this table, but rather via the “Symbol Index” field in an relocation entry. Symbols are of two types: internal and external. Internal symbols reference a symbol within the file. The “Section Number” field then contains the corresponding section in which the symbol is defined. Depending on the value of the “Storage Class” field, the “Symbol Value” field then contains the target address of that symbol. The Storage Class defines what kind of definition a symbol represents. Microsoft commonly uses the following values: `IMAGE_SYM_CLASS_EXTERNAL`, `IMAGE_SYM_CLASS_STATIC`, `IMAGE_SYM_CLASS_FUNCTION` and `IMAGE_SYM_CLASS_FILE`. The “Type” field additionally indicates, whether the symbol is a function or not. The “Auxiliary Entries” field is mostly irrelevant for building a loader.

If the “Section Number” field is set to zero, it indicates that the symbol is not defined within this file. This is called an external symbol. External symbols usually reference functions from a DLL or a Syscall function. The Windows loader injects some common libraries into processes it starts, so that the program can access them. This is not the case when building a custom loader, since it bypasses the Windows loader. External symbols must thus be resolved manually. There are several methods to do this, however, the most common one is to just manually import the `LoadLibraryA` and `GetProcAddress` functions from Windows’ `kernel32.dll` and use them to resolve additional libraries and symbols.

Another important attribute of the symbol entries is the “Symbol Name” field. It is implemented as a C++ union which can take two data types at the same time. The first possible value is a `char[8]` and is defined to contain the name of the symbol – it can therefore only be 8 bytes long. If the symbol is larger, it is stored in the string table instead. To recognize this, the first byte of the union is set to zero. The rest of the union contains a memory offset relative to the beginning of the string table, defined as `uint32_t[2]`. The symbol can be retrieved at this position. External symbol names also follow a convention in which they are prefixed with a constant that is specific to the platform – if marked as such by using the `DECLSPEC_IMPORT` attribute. These prefixes are:

- `__imp_` for the x64 platform
- `__imp__` for the x86 platform

The external `printf` function, for example, would then have the symbol name `__imp_printf` on the x64 platform. This is important as it makes it possible to identify an external symbol by its name prefix only. On Linux, the symbols of a COFF file can be listed manually using the `nm` tool: `nm -C <file>`.

Relocation Entries (dynamically sized, entries have 10 bytes)

A relocation in the context of object files refers to an adjustment applied to machine code or data to correct memory addresses that cannot be determined at compile time. Specifically, relocations mark locations within a section where symbol addresses must be inserted once the final memory layout is known during linking (or in this case during loading). Relocation entries are very small in size, as they only contain these three fields:

- Virtual Address: The address to be relocated.
- Symbol Index: Index in the symbol table for the relocation target.
- Type: Specifies the relocation type (e.g., absolute, relative).

The main target of a COFF loader is to resolve relocations. This is important to be able to work with functions or variables at all. They are structurally quite simple: the “Virtual Address” field contains the address that is to be relocated, relative to the beginning of the section. The “Symbol Index” field contains the zero-based index number within the symbol table under which the corresponding symbol entry can be found. The name of the symbol and the destination address of the relocation can be found there.

The “Type” field is more complex. It indicates the kind of relocation that should be performed. For example, they determine the size of the target address, or whether it must be specified relative to the relocation or absolute as a “virtual address”. This mostly depends on the type of symbol to be relocated: internal symbols are often relocated with relative addresses, because relative jumps are more efficient when used in a closely-spaced memory area. Since the sections of a COFF/PE file are loaded into memory at the same time, they are often close together. Furthermore, relative jumps are not dependent on the absolute position in memory (“position-independent”) and are therefore easier for the operating system to follow. External symbols are generally relocated with absolute addresses, especially if they belong to DLLs that are loaded into the process by the Windows loader. Valid relocation types also depend on machine type [54, section “COFF Relocations”].

The following 7 relative relocation types are used for internal relocations:

- `IMAGE_REL_AMD64_REL32`: 32-bit relative address from the byte following the relocation.
- `IMAGE_REL_AMD64_REL32_1`: 32-bit address relative to byte distance 1 from the relocation.
- `IMAGE_REL_AMD64_REL32_2`: 32-bit address relative to byte distance 2 from the relocation.
- `IMAGE_REL_AMD64_REL32_3`: 32-bit address relative to byte distance 3 from the relocation.
- `IMAGE_REL_AMD64_REL32_4`: 32-bit address relative to byte distance 4 from the relocation.

- `IMAGE_REL_AMD64_REL32_5`: 32-bit address relative to byte distance 5 from the relocation.
- `IMAGE_REL_I386_REL32`: 32-bit relative address from the byte following the relocation (with support for the x86 relative branch and call instructions).

The following 3 absolute relocation types are used for external relocations:

- `IMAGE_REL_AMD64_ADDR64`: 64-bit virtual address of the relocation target.
- `IMAGE_REL_AMD64_ADDR32NB`: 32-bit virtual address without relative virtual address (“image base”).
- `IMAGE_REL_I386_DIR32`: 32-bit virtual address of the relocation target.

It is, always a good idea to support both relative and absolute relocation types for both kinds of symbols. Some more exotic internal symbol types, for example, may also require absolute relocations. There are a few more relocation types, but these are not used in COFF object files by the recommended compilers or are deprecated [54, section “COFF Relocations”], which is why they will not be discussed further in this thesis.

Relocations can also be used to attach local function implementations to symbols, which is necessary for making the beacon APIs available to the BOF, for example.

String Table (dynamically sized)

The string table begins with an integer that specifies its size. This is followed by all contained constant, null-terminated strings. Reading such a string entry requires knowing its starting point and continuing until the termination character is found. The start positions are normally referenced in the symbol table, which is mostly where its importance comes from in this scenario: the symbol table only stores symbol names up to 8 characters within itself. If the name exceeds this number (which it usually does, especially for external symbols), it references an offset within the string table. The symbol names must be retrieved from there.

Summary

To better visualize the somewhat complex structure of a COFF file, Figure 6 shows the generic COFF file structure, while Figure 7 shows a concrete example with common section names.

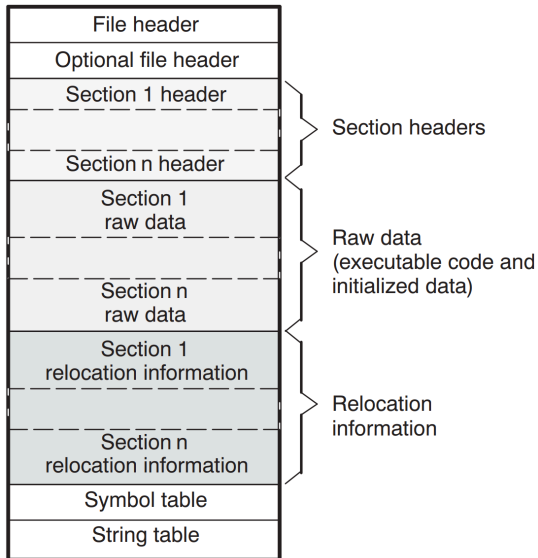


Figure 6: COFF File Structure [3, p. 2]

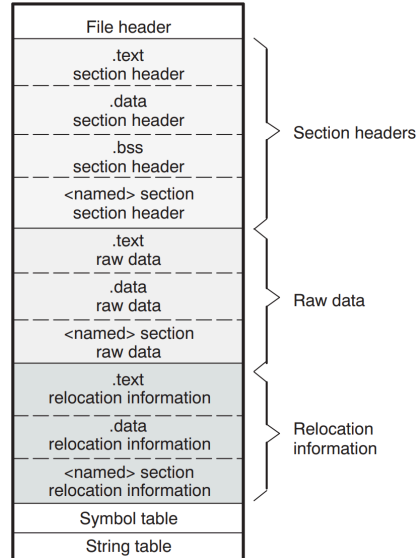


Figure 7: Sample COFF Object File [3, p. 3]

To summarize this chapter, the following simplified procedure is relevant for resolving relocations at runtime and thus executing a COFF file:

- 1) Jump from the file header to the first section header.
- 2) From there, iterate over all section headers.
- 3) For each section header, iterate over all relocation entries for this section.
- 4) For each relocation entry, load the corresponding symbol entry.
- 5) For each symbol entry, check whether its name is stored directly within it.
 - If yes: use this name.
 - If no: get name from the string table.
- 6) Check whether the symbol is an external symbol.
 - If yes: load external symbol and then resolve it manually.
 - If not: resolve symbol manually.

The last step (resolving external symbols) is where both the flexibility and complexity of the loader will lie, since the beacon API functions explained in Chapter 3.2 will also appear as external symbols. Functions that are already available through the C2 beacon runtime can also be made available to the COFF program – and thus to the BOF – by resolving those symbols. Additionally, their linking will be relevant for the implementation of DFR. Both symbol types will be covered in Chapter 5.

As hopefully apparent by now, this process is about replicating the linking process from Windows' own linker `link.exe`, but not "ahead of execution", but rather dynamically at runtime. The Windows linker can assume the memory layout before runtime because the Windows PE loader standardizes it, which

allows the linker to resolve relocations ahead of time. This is not the case with a custom BOF loader, which is why it is necessary to do relocations after copying the BOF to memory. Furthermore, in-memory linking is advantageous because otherwise, linking would have to take place on the file system, which could be quickly classified as suspicious by EDR software.

Now that the goal and the approach is clear, the next step is to determine how to prioritize the implementation of the beacon APIs. These must be implemented by the runtime in order to enable linking them into the BOF by the COFF loader. To do this, a collection of publicly available BOFs will be prepared in the following chapter to determine how they use the beacon APIs. In Chapter 5, this information can then be used to develop the runtime and the loader.

4 Analysis of existing public BOF implementations

This chapter serves to narrow down the necessary requirements for the development of a BOF runtime, which are defined in the following chapter. Methodologically, the first step is to prepare a collection of 15 publicly available BOF implementations. This collection consists mainly of popular repositories on GitHub – measured by the number of GitHub stars – as well as suggestions of commonly used BOFs at cirosec GmbH. The BOFs from this collection are then analyzed individually. It is determined whether they are dependent on the use of the Cobalt Strike Script Manager, i.e. whether they are designed in such a way that they can only be instrumented with the Aggressor Script supplied. This is to determine the importance of implementing Aggressor Script.

Next, the BOF repositories are analyzed with regard to their use of the beacon APIs as specified in `beacon.h` (see Appendix B). These beacon APIs are divided into groups depending on the functionality they serve. This analysis focuses only on these groups, as each of them should be implemented comprehensively. This analysis is used to prioritize which API groups should be implemented first, or at all, as part of this Proof of Concept (PoC) and thesis.

As this chapter is intended to define the requirements for the development of a BOF runtime, the first step is to determine which features are actually used in common BOFs. The following aspects are examined within this chapter:

- 1) Does the BOF's functionality depend on Aggressor Script?
- 2) Which beacon APIs are commonly used by BOFs?
- 3) Is the use of Dynamic Function Resolution (DFR) common?

4.1 Preparing a BOF sample collection

To start with, the 15 BOFs for the sample collection must be developed and defined, on the basis of which the aspects mentioned above are to be evaluated. To accomplish this, the red team at cirosec was asked which BOFs had already been used successfully in past customer projects. The following BOFs were named:

- `trustedsec/CS-Situational-Awareness-BOF`
- `trustedsec/CS-Remote-OPs-BOF`
- `zyn3rgy/smbtakeover`
- `wavvs/nanorobeus`
- `WKL-Sec/HiddenDesktop`
- `fortalice/bofhound`

`fortalice/bofhound` – a BloodHound ingestor built as a BOF – was not included in the further analysis due to officially being discontinued. `WKL-Sec/HiddenDesktop` – a BOF utilizing Hidden Virtual Network Computing (hVNC) to interact with remote desktop sessions – is also excluded, as this project involves considerably more functionality than just a BOF and is based on many steps that are too complex for

the scope of this thesis. Instead, CodeXTF2/WindowSpy was used to capture a similar alternative to this attack vector, as recommended by cirosec.

In addition to the BOFs mentioned by cirosec, the GitHub Search Engine as well as publicly available BOF collections were consulted: some authors, including manufacturers of security solutions and other consulting firms, maintain collections or indices of useful BOFs for this purpose. These collections were consulted:

- rvrsh3ll/BOF_Collection
- outflanknl/C2-Tool-Collection
- HavocFramework/Modules
- RiccardoAncarani/BOFs
- wsummerhill/CobaltStrike_BOF_Collections
- ajpc500/BOFs
- REDMED-X/OperatorsKit
- N7WEra/BofAllTheThings

In the context of this thesis, a restriction was set for the selection of BOFs to simplify the examination: repositories that authoritatively contain the source code for multiple BOFs, that exploit different attack vectors, are not included as collections upfront, but instead split into the single BOFs contained within them. This means, that the individual BOFs contained are included in the sample collection as independent entries. Repositories with multiple BOFs, that all pursue the same goal (e.g., BOFs that assist during the reconnaissance phase), are included as-is.

The public BOF collections and the suggestions from cirosec form the sample selection of 15 BOFs that are to be examined in this chapter. Appendix A provides the commit hashes to document the status or version of the BOFs mentioned here at the time this thesis was written. The selection is shown as follows, each with the number of stars on GitHub and a high-level description of how they work.

fortra/nanodump	GitHub Stars: 1794
NanoDump is a powerful tool designed to create minidumps of the Local Security Authority Subsystem Service (LSASS) with flexibility to adapt to various operational scenarios. It provides multiple methods to handle the dumping process, offering both direct and indirect techniques to obtain LSASS handles securely and covertly. Operators can choose to write the dump to a specified file path or create a valid signature for the dump to avoid detection. The tool supports advanced methods such as duplicating or elevating existing LSASS handles, leveraging the Seclogon service to leak or duplicate handles, and using spoofed call stacks to evade security mechanisms. Additionally, NanoDump enables indirect dumping through external processes like <code>WerFault.exe</code> , which can be triggered using features such as <code>SilentProcessExit</code> or the <code>Shtinking</code> technique.	

The supplied aggressor script is comparatively large, but contains almost exclusively argument parsing, as NanoDump contains numerous flags, some of which are incompatible with each other or must occur in certain group constellations. In addition, Aggressor Script also checks whether, for example, sufficiently high authorizations are available before a command is executed. This greatly increases the OPSEC in particular, which would not be the case without these checks. The BOF can still be executed manually, albeit only with a very precise manual check of the environment and very complex parameter packing.

trustedsec/CS-Situational-Awareness-BOF	GitHub Stars: 1263
<p>Contrary to its name, CS-Situational-Awareness-BOF is not a single BOF, but a collection of smaller BOFs for situational awareness, created by TrustedSec. There are BOFs for enumerating certificates, querying the local ARP table, sending Lightweight Directory Access Protocol (LDAP) queries to the local Active Directory (AD), displaying the visible windows in the current user session and much more. With many of the functions, individual commands of a Windows CMD can be retrofitted in the form of BOFs. As this collection covers the situational awareness area quite comprehensively, this project is probably one of the most important in terms of BOFs.</p> <p>The repository has a single, very large Aggressor Script, which spans all BOFs. However, only basic argument parsing and the invocation of the corresponding BOF is performed for each individual BOF. The use of the Aggressor Script is therefore completely optional.</p>	

trustedsec/CS-Remote-OPs-BOF	GitHub Stars: 828
<p>CS-Remote-OPs-BOF again is a collection of BOFs developed by TrustedSec, complementing their earlier Situational Awareness BOF collection by introducing tools that modify system states, enabling a broader range of offensive security tasks. The BOFs included in this collection cover fundamental Windows operations, such as managing services, registry keys, scheduled tasks, and user accounts. Additionally, the repository offers BOFs for process management, including dumping process memory and handling process states. Recognizing the importance of stealth and evasion, TrustedSec has also included injection BOFs used in EDR testing. While these are provided without support, they serve as valuable resources for understanding and implementing code injection techniques. This collection is probably as important as CS-Situational-Awareness-BOF for red team operations.</p> <p>This repository is divided into two groups: “Injection” and “Remote”. Each of these groups contains its own large Aggressor Script, which covers the BOFs the group contains. Most commands only contain the argument parsing and BOF invocation – but sometimes more is done, such as requesting certain files, like shellcode binaries, via an UI prompt. Some of these can simply be passed to the BOF manually, but this requires a few minor extra steps.</p>	

anthemtothego/InlineExecute-Assembly	GitHub Stars: 610
<p>InlineExecute-Assembly is a PoC BOF developed to facilitate in-process execution of .NET assemblies. This approach serves as an alternative to Cobalt Strike's traditional execute-assembly module, which typically employs a fork-and-run technique². By executing .NET assemblies directly within the current beacon process, InlineExecute-Assembly eliminates the need to spawn sacrificial processes, thereby reducing the operational footprint and enhancing stealth during engagements. The tool is designed to handle assemblies with entry points defined as <code>Main(string[] args)</code> or <code>Main()</code>, allowing for the execution of most existing .NET tools without requiring modifications. It does this by automatically determining and loading the appropriate CLR version before execution.</p> <p>In addition to parsing and executing the BOF, the Aggressor Script handles the server-side loading and serialization of .NET assemblies and passes them as arguments to the BOF invocation call. It is most likely difficult to reproduce this procedure as it is expected by the BOF's argument parser. Furthermore, the remaining parts of the parameter parsing is overly complex, but a translation should be possible with a high amount of effort.</p>	

GhostPack/Koh	GitHub Stars: 487
<p>Koh is a token stealing tool implemented using a server/client architecture. The server, written in C#, is injected into a high-privileged process, such as one running with SYSTEM permissions, where it can continuously monitor and capture user tokens and logon sessions. By operating independently of the C2 infrastructure, the server persists in the target environment, enabling long-term operation without relying on constant communication with the attacker's framework. The client, on the other hand, is implemented as a BOF. It is designed to allow users to send commands to the server, retrieve and use captured tokens for impersonation, and configure its behavior as needed. This server/client architecture avoids the limitations of BOFs, which are inherently ephemeral and tied to the lifecycle of the C2 beacon, meaning that they should not be used for long-running tasks.</p> <p>The supplied Aggressor Script is only used to create the command in Cobalt Strike and for parameter parsing. The various commands such as <code>list</code> and <code>impersonate</code> are mapped in the beacon as an enumeration of integers instead of as a string. This mapping is realized by Aggressor Script. It is therefore possible to use the Koh Client without Aggressor Script, but the mapping of the commands must be translated manually.</p>	

²Technique used in code injections; a new subprocess is spawned ("forked") and executed ("run"), into which the code is injected. Usually more stable, but not OPSEC friendly

mertdas/PrivKit	GitHub Stars: 365
<p>PrivKit is a set of BOFs designed to identify privilege escalation vulnerabilities resulting from misconfigurations in Windows operating systems, thus supporting the work during the reconnaissance phase. The following misconfigurations types can be detected:</p> <ul style="list-style-type: none"> • Unquoted service paths • Autologon registry key set • “Always Install Elevated” registry key set • Modifiable autorun folders • Existence of known hijackable paths • Possible enumeration of credentials from credential manager • Misconfigured token privileges <p>Although the description in the repo says that PrivKit is a single BOF, it actually consists of seven individual smaller BOFs that are bundled into one command with Aggressor Script. None of the BOFs take arguments. They are therefore very trivial to execute standalone.</p>	

CodeXTF2/ScreenshotBOF	GitHub Stars: 346
<p>ScreenshotBOF is a utility to capture screenshots from within a Cobalt Strike beacon using non-malicious Windows APIs. The screenshots can be saved on disk on the targets computer or kept in memory for transmission over the C2 channel.</p> <p>The aggressor script in this repository is comparatively complex, but most of it consists of popup instructions to display the screenshot taken by the beacon in the Cobalt Strike client. The actual invocation of the BOFs is trivial and can be done manually. If wanted, the screenshot displaying capability could be reimplemented in Mythic using Browser Script or Mythic Scripting.</p>	

wavvs/nanorobeus	GitHub Stars: 280
<p>Nanorobeus is a post-exploitation BOF to facilitate privilege escalation, credential dumping, and lateral movement within a compromised Windows environment. While doing virtually the same as the popular Tool “Rubeus”, but as a BOF, it automates the extraction of information, such as credentials, tokens, and service accounts, by utilizing Windows API calls and manipulating native OS processes. Additionally, it supports common attack techniques like Kerberoasting, Pass-the-Hash, and Pass-the-Ticket to bypass authentication mechanisms and move laterally between machines.</p> <p>Aggressor Script usage within nanorobeus is minimal and only performs argument parsing and BOF invocation, which can easily be done manually.</p>	

zyn3rgy/smbtakeover	GitHub Stars: 269
<p>The smbtakeover repository provides techniques to unbind and rebind TCP port 445 on Windows systems without the need to load drivers, inject modules into the LSASS, or reboot the target machine. This approach facilitates SMB-based NTLM relay attacks during C2 operations. The repository includes PoC implementations in both Python and as BOF, utilizing Remote Procedure Call (RPC) over TCP for remote machine targeting. It is also mentioned in the project description that the repositories' layout is inspired by CS-Remote-OPs-BOF by TrustedSec.</p> <p>The use of Aggressor Script is limited here purely to parsing and passing arguments. The invocation of the BOF can also be done manually.</p>	

CodeXTF2/WindowSpy	GitHub Stars: 264
<p>WindowSpy is a BOF designed for targeted user surveillance. Its primary objective is to activate surveillance capabilities only for specific scenarios, such as browser login pages, sensitive documents, or Virtual Private Network (VPN) login screens. This approach enhances stealth by reducing the risk of detection associated with repeated surveillance activities, like taking frequent screenshots. Additionally, it streamlines operations for red teams by minimizing the volume of surveillance data, saving time that would otherwise be spent analyzing extensive logs generated by constant keylogging or screen monitoring.</p> <p>WindowSpy is not actively invoked, but is designed to react to specific, configurable events, such as the creation of a file with a specific file name. This event handling is implemented in Aggressor Script so that the BOF is automatically invoked. Manual execution of the BOF is possible, but misses the purpose of this tool. Implementation of such continuous monitoring within the BOF itself is also not recommended nor feasible, as they are not designed for long-running tasks and would block the runtime of the beacon.</p>	

rsmudge/unhook-bof	GitHub Stars: 262
<p>Unhook-BOF is a simple BOF that removes API hooks on the beacon process. API hooking is often used by EDR software to monitor running processes. This allows certain malicious function calls or memory accesses to be detected and prevented at runtime. With Unhook-BOF, these externally set API hooks can be removed to make the process more stealthy.</p> <p>Aggressor Script usage within unhook-bof is limited to argument parsing and BOF invocation, which can easily be done manually.</p>	

EncodeGroup/BOF-RegSave	GitHub Stars: 186
<p>BOF-RegSave is designed to facilitate privilege escalation and registry key extraction. It enables the beacon to acquire the necessary system privileges and retrieve the SAM, SYSTEM, and SECURITY keys from the Windows registry. These keys can then be analyzed offline to extract password hashes and other sensitive data, aiding in post-exploitation activities. By targeting these critical registry keys, the BOF provides a streamlined and efficient method for gathering credentials and escalating access during red team operations. The results are stored on disk and must be manually extracted afterwards.</p> <p>The description recommends the usage of the included Aggressor Script, however, it only conducts arguments parsing and BOF invocation, which can also be done manually without disadvantages.</p>	

boku7/whereami	GitHub Stars: 160
<p>Whereami is a BOF that extracts information about the running beacon in an OPSEC way. It does this by using handwritten shellcode to return the process environment strings without accessing any DLLs. The shellcode extracts the same information returned from <code>whoami.exe</code> (along with other environment values) from the beacon processes memory. There exists a similar BOF within the CS-Situational-Awareness-BOF collection that can be used to acquire the same information.</p> <p>The provided Aggressor Script only does argument parsing and BOF invocation, which can also be easily done manually.</p>	

connormcgarr/tgtdelegation	GitHub Stars: 154
<p>Tgtdelegation is a BOF to obtain a usable Kerberos Ticket Granting Ticket (TGT) for the current user using the well-known “TGT delegation trick”. A Service Principal Name (SPN) can also be specified if the default SPN is not configured for unconstrained delegation. The process extracts the TGT from Windows API calls and prepares it for the specified target, which must support unconstrained delegation. This approach simplifies obtaining and leveraging Kerberos tickets for Red Team operations.</p> <p>The command definition in the Aggressor Script <code>alias tgtdelegation</code> only deals with the argument parsing and the BOF invocation. This can also be performed manually. However, both the Aggressor Script and the repository contain additional parsing scripts that are executed on the server as soon as the BOF has transmitted its results to the C2 server. These parsing scripts have the task of displaying the transmitted binary blobs as <code>.ccache</code> and <code>.kirbi</code> files. This can also be done manually, but requires higher effort.</p>	

ASkyeye/Cobalt-Clip	GitHub Stars: 1
<p>Cobalt-Clip is a BOF that enables interaction with a target’s clipboard during post-exploitation activities. It allows for dumping and setting the current contents of it, while also offering an option to monitor the clipboard for changes, providing details such as the updated content, the active window at the time of change, and the timestamp, using the <code>clipmon</code> command. This command operates as a reflective DLL instead of within a BOF – correctly adhering to the intended design of BOFs not being used for long-running tasks – and is initiated as a job using the <code>bdllspawn</code> function within the Aggressor Script.</p> <p>For one-shot accesses to the clipboard, the supplied Aggressor Script only registers the respective commands, which are passed on to the BOF as arguments as they are. The BOF can therefore be executed independently with the commands <code>dumpclip</code> and <code>set-clipboard-data</code>. The <code>clipmon</code> command to monitor the clipboard continuously, however, does not fit into the scope of this thesis due to not using a BOF. Accordingly, the assessment that there is no dependency on Aggressor Script is nevertheless made here.</p>	

4.2 Assessing the Usage of Aggressor Script

The first step is to assess the use of Aggressor Script within the BOF repositories from the sample collection. Table 3 shows whether the corresponding repositories contain an Aggressor Script (“AS supplied”) and how strongly the functionality of the respective BOF depends on it (“AS required”). The dependencies are represented using the following values:

- **yes** : The BOF cannot be used without the Aggressor Script.
- **mostly** : Most features of the BOF cannot be used without using the Aggressor Script or require time-consuming extra steps to reproduce the instructions.
- **partially** : Most features of the BOF can be used without using the Aggressor Script or at most require smaller extra steps to reproduce the instructions.
- **no** : The BOF can be executed and utilized with no drawbacks without using the Aggressor Script.

The investigation of the use of Aggressor Script is intended to answer the first research question of this chapter.

BOF Name	AS supplied	AS required
fortra/nanodump	yes	mostly
trustedsec/CS-Situational-Awareness-BOF	yes	no
trustedsec/CS-Remote-OPs-BOF	yes	partially
antheuntohego/InlineExecute-Assembly	yes	mostly
GhostPack/Koh	yes	partially

BOF Name	AS supplied	AS required
mertdas/PrivKit	yes	no
CodeXTF2/ScreenshotBOF	yes	no
wavvs/nanorobeus	yes	no
zyn3rgy/smbtakeover	yes	no
CodeXTF2/WindowSpy	yes	yes
rsmudge/unhook-bof	yes	no
EncodeGroup/BOF-RegSave	yes	no
boku7/whereami	yes	no
connormcgarr/tgtdelegation	yes	partially
ASkyeye/Cobalt-Clip	yes	no

Table 3: List of the BOF sample collection with their usage of Aggressor Script

As shown in Table 3, it is common to provide an Aggressor Script together with the BOF: all BOFs examined have an obviously associated Aggressor Script. In the provided instructions for using the tools, it is almost always recommended importing the Aggressor Script instead of using the BOF directly. Nevertheless, nine of these tools do not require this (marked with “AS required: no”). The Aggressor Script only passes the entered parameters to the BOF using the `bof_pack` function and at most serves to display a help page. Chapter 6 goes into further detail on how the `bof_pack` Aggressor Script function works to prepare the arguments to the BOF and how these function calls can be examined so that this process can be carried out by hand. In addition, the `barch` (“beacon architecture”) function is used in almost every Aggressor Script. It allows to determine the processor architecture of the running beacon process in order to then find the appropriate BOF binary matching this architecture. This is a simple but useful feature, but by no means a necessity for the use of BOFs.

There are also many counterexamples where the Aggressor Script contributes extensively to the actual functionality of the tool: the BOF in `CodeXTF2/WindowSpy`, for example, cannot be executed proactively at all, but relies on the event handling capabilities of Aggressor Script. `ASkyeye/Cobalt-Clip` implements its monitoring feature completely in Aggressor Script. `anthemtotheego/InlineExecute-Assembly` uses Aggressor Script for serializing input files and `connormcgarr/tgtdelegation` uses it for deserializing and storing Kerberos TGTs.

It is probably very time-consuming, but in some cases definitely possible, to perform these tasks either manually or, in the case of Mythic, via Browser Script or Mythic Scripting. Nevertheless, there are certainly BOFs that are simply impossible to execute outside of Cobalt Strike – i.e. without Aggressor Script. An implementation of this plugin system should therefore not only provide a BOF runtime, but also an Aggressor Script interpreter if possible.

4.3 Assessing the Usage of Beacon APIs

After the Aggressor Script usage has been reviewed, the usage of the beacon APIs can now be examined. It is checked which of the beacon API groups already described in Chapter 3.2 are called by the BOF implementation. It is sufficient for the examined BOF to call a single function of an API group for this to count as use of the API group (marked with ✓ in Table 4). This examination serves to determine whether it is even necessary to implement all API groups in the compatibility layer of the runtime.

To determine which functions are called, a Python script is used, which uses the GitHub Search API to perform a full-text search in the source code of the BOF repository. It searches for all occurrences of the functions defined in `beacon.h` (see Appendix B). The Python script used was created using OpenAI's ChatGPT and is listed in Appendix C.

BOF Name	Data Parser API	Format API	Output API	Token API	Spawn+Inject API	Utility API	Key/Value Store API	Data Store API	User Data API	Syscall API	Beacon Gate API
fortra/nanodump	✓		✓								
trustedsec/CS-Situational-Awareness-BOF	✓		✓			✓					
trustedsec/CS-Remote-OPs-BOF	✓		✓		✓	✓					
antheettoheego/InlineExecute-Assembly	✓		✓								
GhostPack/Koh	✓		✓	✓							
mertdas/PrivKit			✓								
CodeXTF2/ScreenshotBOF	✓		✓								
wavvs/nanorobeus	✓										
zyn3rgy/smbtakeover	✓	✓	✓								
CodeXTF2/WindowSpy	✓		✓				✓				
rsmudge/unhook-bof	✓	✓	✓								
EncodeGroup/BOF-RegSave	✓		✓	✓							
boku7/whereami		✓	✓								
connormcgarr/tgtdelegation	✓		✓								
ASkyeye/Cobalt-Clip	✓		✓								

Table 4: List of the BOF sample collection with their usage of the Cobalt Strike beacon APIs

Table 4 clearly shows the beacon APIs that are commonly used and those that are completely unused by the BOF test selection. The Data Parser API for processing arguments is used by almost every BOF, except those that do not take any arguments. The Format API is only used very rarely, but the Output API is used almost everywhere. The function `BeaconPrintf` from the Output API can accept format strings, which is why it can be assumed that it accesses the internal implementation of the Format API. Accordingly, these API groups can also be classified as very useful and must be implemented.

The Token API is rarely used, sometimes for setting a token handle obtained from another function, sometimes only to determine whether a current token has elevated privileges. The Spawn+Inject API is only used by `trustedsec/CS-Remote-OPs-BOF` in the “shspawnas” BOF to enable code injections into a new process created for this purpose. Both API groups provide comparatively complex functions, which above all probably require good obfuscation. Nevertheless, they are in use. Implementation of these API groups is therefore recommended, but not mandatory.

The Key/Value Store API and the Utility API are only used sparsely. However, these should be relatively straightforward to implement as a baseline variant, which is why these API groups are classified as recommended to implement.

The remaining API groups are not utilized in any way. This is most likely due to the fact that they have only been available since newer versions of Cobalt Strike. Some of the groups also make use of functions in Cobalt Strike that are not available in the context of this thesis and are not relevant in Mythic Beacons. The User Data API is only useful if the beacon is used together with a User-Defined Reflective Loader (UDRL). A function like this does not exist in Mythic by design. The same applies to the Beacon Gate API, which makes use of Cobalt Strike’s Sleepmask feature that can be activated or deactivated through it. The Syscall API is probably not used, as the underlying functions can be directly imported using DFR. This leaves only the Data Store API, which can be used to cache arbitrary files in the beacon’s memory. This feature would certainly be useful in specific cases, but its implementation and the other API groups mentioned in this paragraph will be omitted due to the lack of use by BOFs.

4.4 Assessing the Usage of DFR

Finally, it must be determined which of the BOFs rely on DFR to resolve and use Windows APIs. Chapter 3 has already covered DFR. To permit DFR usage in the BOF runtime, there must be a corresponding implementation for resolving the functions, the necessity of which is checked by this examination. Table 5 lists, which of these BOFs rely on DFR. If more than seven DFR functions are defined within the repository and at least one of them is used in more than one place, it is considered an “extensive” usage of DFR. If DFR is only used for very trivial tasks, like using the `GetLastError()` C function, then the usage is considered “trivial”.

BOF Name	Uses DFR
fortra/nanodump	✓ (extensively)
trustedsec/CS-Situational-Awareness-BOF	✓

BOF Name	Uses DFR
trustedsec/CS-Remote-OPs-BOF	✓
anthemtotheego/InlineExecute-Assembly	✓ (extensively)
GhostPack/Koh	✓ (extensively)
mertdas/PrivKit	✓ (extensively)
CodeXTF2/ScreenshotBOF	✓
wavvs/nanorobeus	✓ (extensively)
zyn3rgy/smbtakeover	✓ (trivial)
CodeXTF2/WindowSpy	✗
rsmudge/unhook-bof	✓
EncodeGroup/BOF-RegSave	✓
boku7/whereami	✗
connormcgarr/tgtdelegation	✓ (extensively)
ASkyeye/Cobalt-Clip	✓ (extensively)

Table 5: List of BOF with their usage of DFR

As shown in Table 5, almost all the BOFs examined here use DFR – most of them so extensively that the BOF’s intended functions are realized almost exclusively through DFR functions. For example, ASkyeye/Cobalt-Clip uses the functions `USER32$OpenClipboard`, `USER32$GetClipboardData` and `USER32$SetClipboardData` to interact with the clipboard, which are resolved via DFR. Overall, most BOFs, that use DFR, use functions intended for error handling, most notably `GetLastError` from `kernel32.dll`. Many BOFs also access functions for string manipulation or memory management, although implementations are already available with the `Format` and `Syscall` beacon APIs. This is probably due to the fact that DFR was introduced earlier than the beacon APIs mentioned.

While examining the DFR usage, it was noticed that some of the repositories have syntactically very similar definitions of the DFR functions, but the majority of these functions are not called at all. This is due to the fact that TrustedSec has stored a header file `bofdefs.h` in its CS-Situational-Awareness-BOF collection [55], in which all common Windows APIs are defined using the DFR notation. This header file is part of the BOF template, which is also explicitly advertised so that it can be used by other BOF authors. This facilitates the use of DFR functions for other authors and leads to the `bofdefs.h` also being found in other repositories. Later on, during the development of the BOF runtime, the `bofdefs.h` is also used to extensively test the loading of the defined functions via DFR. This is to ensure that repositories that use the `bofdefs.h` are correctly supported.

The conclusions drawn in this chapter are revisited in the following chapter to define the requirements for developing the runtime.

5 Implementation of a generic C++ BOF runtime

In this chapter, the knowledge gained from the basics of the COFF format in Chapter 3 and the evaluated beacon API usages from Chapter 4 will be used to develop a generic BOF runtime. Firstly, certain principles and requirements will be defined, which should be adhered to in order to keep the runtime as portable and OPSEC-safe as possible. This section also serves as a design document for the implementation. The aim is to make the runtime usable in any project that requires BOF compatibility.

After defining the requirements, a minimal analysis of existing standalone COFF runtimes and loaders is to be conducted. A superficial analysis is to be carried out to determine how they work and which features can potentially be adopted.

Subsequently, the public interface of the runtime is defined. The focus here is on how a user interacts with the runtime and how many options the user has for manipulating the behavior of it. The signatures of the publicly accessible functions are delineated as well.

The next section deals with the implementation of the beacon APIs. The methodology and technique of the implementation are discussed in detail, rather than the source code itself. The implementation will be finalized in this section. It serves as the basis for the following chapter for the final implementation into `ciroStrike`.

Finally, the various approaches and considerations that were not adopted in the runtime are documented and the reasons for not adopting them are explained.

5.1 Requirements

This section defines the requirements that must be met by the BOF runtime. They are supposed to represent the result of the COFF file format basics and the analysis of beacon API and Aggressor Script usage. Some of the requirements are not based on research results within this thesis, but have been set to the best of knowledge and common practices of malware development. They serve as a guideline for the implementation.

The aim of these requirements should be that the runtime can be used meaningfully not only in `ciroStrike`, or even in the context of C2 frameworks in general, but also in other areas. Loaders for certain file formats, such as COFF in this case, are useful in other fields where OPSEC plays a crucial role. The usual restrictions on the use of such tools in offensive security and especially in red teaming are taken into account as much as possible.

The BOF runtime should consist of three components:

- **COFF loader:** The largest component of the runtime, which is responsible for dynamically linking and executing the COFF files.

- **Compatibility layer:** The component of the runtime that establishes the compatibility of the COFF loader with Cobalt Strike’s API. This mainly includes the headers of the beacon APIs, but also features like DFR.
- **Public API:** Public C++ functions exposed to the user of the library to invoke the BOF runtime.

The methodical approach is as follows: the requirements are summarized in tabular form and enumerated with the syntax R[<C>]<N>, where <C> is the component and <N> is the identifier of the requirement across all components. The following constants are used for the components:

- L for the COFF loader.
- C for the compatibility layer.
- A for the public API.
- No indication of <C> if the requirement applies to all components, for example, if they are necessary restrictions to ensure compatibility with shellcode projects.

Table 6, Table 7, Table 8 and Table 9 list the recorded requirements per component. The user of the BOF runtime is referred to as “Caller” below, the runtime code as “Callee”. In the “Source” column, the chapter within this thesis is optionally documented in which the requirement was identified as being necessary. All requirements for which no source was specified were set according to best practice or the best of knowledge.

General requirements		
No.	Source	Description
R1	-	For easier implementation of the loader in other projects, it should not be built as an executable, but as a library. There should be no entry point, but a public API.
R2	-	Only local non-static variables are allowed to be used. The use of global and static-local variables prevents the use of the runtime in some scenarios, due to no <code>.bss</code> section being present.
R3	-	Where possible, the process heap should not be used.
R4	-	The use of exceptions and the <code>try catch</code> control structure must be avoided – likewise, the generation of exceptions must be prevented in all cases. Their use requires the existence of a <code>.pdata</code> section [54, section “The <code>.pdata</code> Section”], which cannot be guaranteed in all areas of application.
R5	-	The functions of the C standard library may not be used. Simpler functions, e.g., for string manipulation or copying of memory, can be implemented by the user as required.

Table 6: General requirements for the BOF runtime

Loader requirements		
No.	Source	Description
RL6	-	All operations of the loader must take place in memory. Access to the file system is not allowed, nor may the content of input buffers be changed.
RL7	Chap. 3.4	The loader must perform the relocations of internal symbols in the BOF. All common relocation types, as documented in the source chapter, must be supported.
RL8	Chap. 3.4	The loader must perform the relocation of external symbols in the BOF. External symbols must either refer to the functions passed to the BOF by default (see RC15) or represent a DFR signature via which means it is to be loaded and resolved.
RL9	-	After processing all relocations, the loader must search for the specified symbol for the entry point function and call it according to the parameters prescribed by Cobalt Strike.
RL10	-	The use of external functions in the loader itself should be minimal.
RL11	-	External functions that are required by the loader and that come from the Windows API must not be resolved by the loader itself. They must be passed to the runtime by the caller as function pointers. This allows the caller to resolve the functions using an appropriate method of choice (e.g., PEB-Walk ³).
RL12	Chap. 4.4	The loader must support DFR with the naming convention specified by Cobalt Strike.
RL13	-	The caller must provide a function that resolves to a function pointer based on library name and function symbol. It will be used by the DFR implementation. This way, the caller can implement the resolution of external functions itself. The function must have the following signature and return a pointer to the resolved function (or NULL if the function could not be resolved): <code>void *Resolve(const char *library, const char *func);</code>
RL14	Chap. 4.2	The effects of Aggressor Script must be ignored within the loader.

Table 7: Loader requirements for the BOF runtime

³Technique to locate loaded modules to use their functions by traversing the Process Environment Block (PEB); often used to evade API-based detection

Compatibility layer requirements		
No.	Source	Description
RC15	Chap. 3.1	The following functions must always be made available to the BOF by default: <code>GetProcAddress</code> , <code>LoadLibraryA</code> , <code>GetModuleHandle</code> and <code>FreeLibrary</code> . This is a requirement of Cobalt Strike [50] and allows the BOF to resolve functions independently and without the use of DFR.
RC16	-	The functions of the beacon API are implemented by the caller, not by the runtime itself. The implementations must be passed to the runtime via a suitable struct of function pointers. This is particularly important to make it easier for the caller to implement the functions in the mythic beacon code (e.g., so that <code>BeaconPrintf</code> can route its output through the beacon's C2 profile).
RC17	Chap. 4.3	The struct of function pointers described in RC16 must allow the passing of all functions of the Data Parser API.
RC18	Chap. 4.3	The struct of function pointers described in RC16 must allow the passing of all functions of the Format API.
RC19	Chap. 4.3	The struct of function pointers described in RC16 must allow the passing of all functions of the Output API.
RC20	Chap. 4.3	The struct of function pointers described in RC16 must allow the passing of all functions of the Token API.
RC21	Chap. 4.3	The struct of function pointers described in RC16 must allow the passing of all functions of the Utility API.

Table 8: Compatibility layer requirements for the BOF runtime

Public API requirements		
No.	Source	Description
RA22	-	It must be possible to pass the BOF to the runtime as a <code>const unsigned char*</code> buffer via the public API.
RA23	Chap. 3.2	The API must accept the parameters for the BOF as a size-prefixed binary blob, as specified by Cobalt Strike. This reduces the effort required to serialize the parameters immediately before the BOF invocation.
RA24	-	The API must provide a function to convert the parameters from a null-terminated string to a size-prefixed binary blob. This makes it easier for the caller to transfer the arguments (e.g., through a C2 profile).
RA25	-	The API must provide a parameter that can be used to define the name of the symbol of the entry point function of the BOF.
RA26	-	The API must provide parameters for passing the structs of function pointers. These structs are described in RL11 , RC16 and RL13 . This simplifies the fulfillment of R2 , since the structs are stack-allocated.

Table 9: Public API requirements for the BOF runtime

These requirements serve as a guideline for the implementation of the BOF runtime in the following sections.

5.2 Existing or Related BOF runtimes

Before delving into the development of the BOF loader, it is essential to explore existing projects in this domain. By analyzing different projects, their strengths, limitations, and contributions to the field can be identified, laying a foundation for the approach done in this chapter. Similar to the samples gathered in Chapter 4.1, the BOF runtimes found here are sorted by popularity measured by the GitHub stars on the repositories.

There are many BOF loaders that are suitable for different use-cases and the exploitation of different attack vectors, such as PowerShell loaders or .NET-based loaders. As this thesis intends to implement the loader into ciroStrike, only implementations developed in C and C++ are considered in the following. The following collection in Table 10 is, to best knowledge, considered complete, as there were no other projects found developed in that language.

The projects are evaluated against the requirements defined in the previous chapter, in order to incorporate similar and possibly field-tested approaches into the development of an own runtime. Special attention should be paid to the dependencies of a BOF loader implementation, because requirement **R5** prohibits the use of the standard library. The projects will also be examined regarding their initial support for the beacon APIs as mentioned by requirement **RC16** and the BOFs collected in Chapter 4.

Table 10 lists the investigated BOF runtimes with their respective beacon API support. Full support per group is denoted as ✓, while partial support is shown with !.

Runtime/Loader Name	GH Stars ⁴	Data API	Format API	Output API	Token API	Spawn+Inject API	Utility API	Key/Value Store API	Data Store API	User Data API	Syscall API	Beacon Gate API
trustedsec/COFFLoader	471	✓	✓	✓	!	!	✓					
Cracked5pider/CoffeeLdr	273	✓	✓	✓	!	!	✓					
Yaxser/COFFLoader2	203	✓	✓	✓	!	!	✓					
OtterHacker/CoffLoader	48	✓	✓	✓	!	!	✓					
sliverarmory/COFFLoader	39	✓	✓	✓	!	!	✓					
cloudflare/cloudflare-blog (2021-03-obj-file/3)	unknown ⁵											

Table 10: List of BOF runtimes with their respective support for the Cobalt Strike beacon APIs

Beacon API Support

As can easily be seen from the table, all projects share the same level of beacon API support, except for Cloudflare’s loader (subproject of cloudflare/cloudflare-blog). This is because, unlike the other projects, the Cloudflare loader does not explicitly focus on loading Cobalt Strike BOFs, but rather handles the loading of Linux object files in a more general way. These are very similar to COFFs, which is why this project was included here. Relocations are not patched to self-implemented functions by this loader and DFR is not supported either, since this is a feature introduced by Cobalt Strike. Internal relocations and external relocations on imports are still carried out. This project will only be consulted for the purpose of detailed documentation [56], [57], [58] in the further course of the thesis, but won’t be mentioned in this section again.

Looking at the other projects, it is noticeable that many parts of the source code are similar. Cracked5pider/CoffeeLdr, Yaxser/COFFLoader2 and OtterHacker/COFFLoader contain references to trustedsec/COFFLoader in the form of code comments or in the corresponding project description, but they all have their own loader implementations. Only the implementations of the compatibility layer and the beacon API overlap. This explains why the beacon API support of all these projects is very similar. The support of the API groups also shows once again that the more specialized groups are hardly relevant for the loaders due to their low usage. sliverarmory/COFFLoader, on the other hand, is a

⁴As of November 18th 2024

⁵Repository contains other independent projects; star count is 1176

fork with extensions and improvements over trustedsec/COFFLoader (built as a DLL, more permissive permissions on the memory regions) and has a direct fork relationship to it on GitHub.

Since the projects share the implementations of the beacon API, the problem also arises in all of them that the Token and Spawn+Inject API are only partially implemented. This means that the function headers are present, but they are left unimplemented. API groups that are not implemented at all do not contain the function header either, so calling these functions from a BOF will result in a memory access violation. However, the empty implementations also increase the risk of the runtime crashing when executing a BOF: the return behavior may differ massively from that of previously executed tests during development, which can cause crashes in later parts of the code. Likewise, by implementing the beacon APIs in the loader itself, all projects violate requirement **RC16** and thus make integration into custom C2 infrastructure setups more difficult.

DFR Support

All loaders mentioned have DFR support that is implemented similarly. To resolve the functions, GetLibraryA and GetProcAddress are used and the DFR signature is parsed by splitting at the \$ sign, without any further validation. This means that all projects fulfill **RL12**. However, the mechanism for resolving the functions is hard-coded in all projects, which means that **RL13** cannot be fulfilled.

C Standard Library and Windows API Dependencies

What all projects also have in common is that they are built as a proof of concept executable. In particular, trustedsec/COFFLoader is intended to serve as a drop-in replacement for a complete Cobalt Strike environment. This is very valuable for developing BOFs and allows for easier and more efficient testing, especially since no Cobalt Strike license is required to do so. The provider Zero-Point Security even uses this loader for its BOF development course [59, section “FAQ”]. The problem with this is that the “convenience features”, such as loading the BOF from disk, have deep dependencies on the Windows API and the C standard library. The trustedsec/COFFLoader project includes precompiler flags that allow suspending compilation of the entry point, effectively turning it into a library. However, the function for loading the BOF from disk, as well as the corresponding standard library functions, remain, and must thus be removed manually.

All projects also use parts of the standard library in the actual loading process. This means that even when removing the standalone components from the projects, there will still be dependencies to the Windows API and the C standard library. All projects thus violate the requirements **R1**, **RL10** and **R5**. Functions of this kind must be removed.

BOF Parameter Passing

The standalone version of Cracked5pider/CoffeeLdr does not support passing arguments to the BOF. The loader itself is potentially capable of doing so, but the arguments must be specified directly in the size-prefixed binary format specified by Cobalt Strike. A function for preparing this binary blob does not exist. Yaxser/COFFLoader2 can accept arguments for the BOF as a hex string. This hex string must

exactly represent the content of the size-prefixed binary blob. A function for preparing such a hex string is not provided, but can be created manually with knowledge of the Data Parser API. For OtterHacker/CoffLoader, it was not possible to determine the format in which it wants to receive arguments due to missing documentation. There is, however, a function to pack arguments returning the correct size-prefixed binary blob from a struct definition, that actually holds all data that the binary blob carries. It should therefore be possible to pass arguments using this struct, however, this was not tested in this thesis.

The trustedsec/COFFLoader project is the only one that documents how argument passing works [13]. The loader implementation accepts the size-prefixed binary blob, but there is also a utility function that can be used to generate it from a hex string. The hex string can in turn be generated using the companion CLI tool `beacon_generate.py`, written in Python, which is also included in the repository. It can be used to interactively specify the data type and the value for each argument, which ultimately yields a hex string that can be passed to the loader. Thus, trustedsec/COFFLoader is the only project that fulfills **RA24**, while all other projects only fulfil requirement **RA23** – except Cracked5pider/CoffeeLdr, which does not fulfill any of the requirements. Since sliverarmory/COFFLoader adopts many of the source code of trustedsec/COFFLoader, the argument passing works exactly the same way here: the `beacon_generate.py` tool can even be used to prepare the arguments for it.

Aggressor Script Support

None of the projects take the effects of Aggressor Script into account and thus fulfill **RL14**. No adjustments are necessary in this regard.

COFF Loading Process

All projects implement a structurally uniform loading process. The process corresponds the findings in Chapter 3.4. During testing of all loader projects, simpler BOFs such as implementations of the Windows commands `whoami` and `dir` could be executed without errors. Only the order of the loading steps performed differs: in some projects, the entry point is searched for first, in some the relocations are performed first. Even the resolving of external functions used by the loader itself sometimes happens ahead of execution and sometimes “as needed”. However, this does not change the actual functionality of the loader and the result of BOF execution. Thus, all projects fulfill the requirements **RL7**, **RL8** and **RL9**.

Similarly, all projects make the standard functions `GetProcAddress`, `LoadLibraryA`, `GetModuleHandle` and `FreeLibrary` available to the BOF and thus also fulfill **RC15**. All projects also avoid the use of exceptions, which fulfills **R4**, as well.

Summary

As can be seen, many runtime implementations make use of the research results presented in trustedsec/COFFLoader. The projects merely implement the loading process differently, but structurally with the same approach. All projects have DFR support and support the same set of beacon APIs. However, they all have the problem that they depend on the C standard library and/or on certain Windows API functions.

In the following sections of this thesis, trustedsec/COFFLoader is therefore mainly used as a reference for the implementation of a custom BOF runtime, since most of the given requirements have already been met here compared to the other projects. In summary, the following requirements still need to be fulfilled:

- **R1, RA26:** The runtime must be built as a library and the public API must be adjusted accordingly.
- **R2:** The beacon API implementation currently uses global variables.
- **R3:** The loader uses different heap segments as well as virtual pages. This inconsistent memory usage must be limited.
- **R5, RL10:** Both loader, standalone utilities and beacon API implementations use the C standard library and Windows API functions.
- **RL11:** External functions must be passed by the caller, not resolved by the loader itself.
- **RL13:** The function for DFR resolution must be passed by the caller.
- **RC16:** Beacon API implementations must be passed by the caller. The following requirements must also be met for the handover: **RC17, RC18, RC19, RC20, RC21.**

5.3 Runtime Implementation and Public API

This section describes the implementation of the runtime. The most important code sections are briefly explained. Details that, for example, resemble the results of Chapter 3 or come from official documentation are omitted. These details are explained in the source code with code comments. The source code and documentation of the BOF runtime are available at <https://github.com/cirosec/bof-loader> at the publication date of the thesis.

As mentioned in the previous chapter, trustedsec/COFFLoader is used as the basis for the implementation. The Microsoft Visual Studio Build Tools are used for this implementation, since ciroStrike also relies on them. However, no MSVC-specific features are to be used in the source code to make it easier to port to CMake or other build tools.

The Visual Studio solution contains the BOF runtime as a static library project, as well as an executable project, called “TestMain”. The BOF runtime project is configured so that it is not linked against the Windows API and the C standard library. A successful compilation of the project thus implicitly fulfills the requirements **R5** and **RL10**. The TestMain project, on the other hand, is a Windows executable project with standard configuration. The executable is meant to be used to test the runtime only. It loads a BOF from disk, serializes it and then passes it to the runtime. The preparation of the arguments is also carried out in this project. When using the runtime as a library, the TestMain project is not included. The details of the implementation of TestMain are not discussed again until Chapter 6 – this section only deals with the BOF runtime.

Removal of Standard Library Dependencies

First, all utility functions must be removed from the project. These include the `getContents` function that loads the BOF from disk. The function is moved to the `TestMain` project. In addition, the debug print macros that use the `stdio.h` header are removed.

The next step is to identify and replace some string and memory manipulation functions from the standard library. The following functions are currently used in the project:

- `memcpy`: Function for copying buffer contents.
- `memset`: Function for setting buffer contents to a fixed value.
- `strlen`: Function for determining the length of a zero-terminated string.
- `strcmp`: Function for comparing strings.
- `strncmp`: Function for comparing strings for a certain number of characters.
- `strncpy`: Function for copying a string with a certain number of characters into another buffer.
- `strtol`: Function for converting a string to a `long int`.
- `strtok`: String tokenizer, usually used to split strings at a specified separator.

Calls to the `strcmp` function were replaced with the more versatile variant `strncmp`, which is already used in other places within the project, to reduce the amount of functions. Furthermore, the tokenizer function `strtok` was replaced with its thread-safe variant `strtok_r`, because `strtok` relies on the use of static local variables to store the state of the tokenization. Since static local variables, just like global variables, are stored in the `.bss` section, their use is not possible as defined by **R2**. `strtok_r`, on the other hand, uses a buffer as a cache in an additional function argument, thus avoiding this problem.

The remaining functions were implemented with the help of ChatGPT, so that neither heap nor global variables are used by them. The function headers are provided to the loader via `Memory.h` and `StringManipulation.h`. This fulfills requirement **R5**.

Reviewing Heap Usage

The use of the heap is very fragmented throughout the project. A mixture of the following functions for memory management is used:

- `VirtualAlloc/VirtualFree`
- `malloc/free`
- `calloc/free`

Since `malloc`, `calloc` and `free` are the only remaining memory management functions that do not natively come from the Windows API, it was decided to remove them. Instead, `HeapAlloc` and `HeapFree` from the Windows API are used – these can be used to behave identically to `malloc`, `calloc` and `free`. `VirtualAlloc` could not be used as a replacement, because it does not request plain heap memory, but instead allocates so-called pages – an abstraction above the heap memory to prevent fragmentation [60].

This means that they may have a different memory layout than plain heap memory, which is particularly disadvantageous when the contents of entire COFF files sections have to be mapped in memory. In addition to `HeapAlloc`, the function `GetProcessHeap` must be added to specify the target heap for the `HeapAlloc` call.

These adjustments streamlined heap usage, thereby fulfilling requirement **R3**.

Passing External Functions

To fulfill requirement **RA26** in the context of the public API later, a struct must first be defined, via which the previously recorded functions, required by the runtime, can be transferred. These functions are called “External Functions”. To ensure the type safety of the passed functions, a separate type definition is created for each, instead of using the generic function pointer type `void*`. These definitions should match the original definitions.

In addition to the external functions, the `Resolve` function for reloading additional external functions via DFR is also specified in this struct. This thus also fulfills requirement **RL13**.

This results in the definitions as per Listing 9 for the external functions in the `ExternalFuncs.h` header, which is provided to both the loader and the caller.

```

1 #include "wintypes.h"
2
3 typedef LPVOID(__stdcall* VirtualAlloc_t)(LPVOID lpAddress,
4     SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);
5 typedef BOOL(__stdcall* VirtualFree_t)(LPVOID lpAddress,
6     SIZE_T dwSize, DWORD dwFreeType);
7 typedef LPVOID(__stdcall* HeapAlloc_t)(HANDLE hHeap,
8     DWORD wFlags, SIZE_T dwBytes);
9 typedef BOOL(__stdcall* HeapFree_t)(HANDLE hHeap,
10    DWORD dwFlags, LPVOID lpMem);
11 typedef HANDLE(__stdcall* GetProcessHeap_t)();
12
13 typedef HMODULE(*LoadLibraryA_t)(LPCSTR lpLibFilename);
14 typedef HMODULE(*GetModuleHandleA_t)(LPCSTR lpModuleName);
15 typedef FARPROC(*GetProcAddress_t)(HMODULE hModule, LPCSTR lpProcName);
16 typedef BOOL(*FreeLibrary_t)(HMODULE hLibModule);
17
18 typedef void(*Resolve_t)(const char *library, const char *func);

```

Listing 9: Type definitions for external functions required in the BOF loader

Since the Windows API usually uses the data type definitions from itself for parameters and return values, these must also be redefined, since including `Windows.h` would cause a linker error. Although most of these types are defined in the separate headers `BaseTsd.h`, `WinDef.h` and `WinNT.h`, they are not suitable for inclusion in projects that do not already include `Windows.h` [61]. Referencing these headers in the

loader led to various errors that could not be resolved. As a result, the required definitions, such as HANDLE, LPVOID, DWORD and others, were redefined by hand in `wintypes.h` in such a way that they are compatible with the Windows type definitions. They were implemented in a platform-agnostic way using precompiler directives, which is especially important regarding the size of pointers. Collisions with the eponymous definitions from the original `Windows.h` were also avoided using precompiler directives. `wintypes.h` is used in several places within the runtime, but mainly in the definitions of the external functions.

The struct that contains the function pointers just defined is shown in Listing 10.

```
1 typedef struct external_functions {
2     VirtualAlloc_t VirtualAlloc;
3     VirtualFree_t VirtualFree;
4
5     HeapAlloc_t HeapAlloc;
6     HeapFree_t HeapFree;
7     GetProcessHeap_t GetProcessHeap;
8
9     LoadLibraryA_t LoadLibraryA;
10    GetModuleHandleA_t GetModuleHandleA;
11    GetProcAddress_t GetProcAddress;
12    FreeLibrary_t FreeLibrary;
13
14    Resolve_t Resolve;
15 } external_functions_t, *external_functions_ptr_t;
```

Listing 10: Struct definition of `external_functions_t` in the BOF runtime

Using the example of `HeapAlloc_t` with `GetProcessHeap_t`, a function, that creates a 10-character-long string with space for the null terminator, would be called like shown in Listing 11:

```
1 external_functions_ptr_t external_functions = {...};
2 char *some_string = (char*)external_functions->HeapAlloc(
3     external_functions->GetProcessHeap(), 0, sizeof(char) * 10
4 );
```

Listing 11: Usage of functions within the `external_functions_t` struct

Passing Beacon API Implementations

Just as with passing external functions, it must also be possible to pass the beacon API functions implemented by the caller according to **RC16**. These are generally called “Cobalt Strike Compatibility Functions” (or “CS Compat Functions” for short). For this, function type definitions and a struct for passing are created. In addition to these, the structs `datap_t` for holding the state of the Data Parser API and `formatp_t` for holding the state of the Format API must also be defined, as they are used as parameter types within the corresponding API groups. These structs have already been explained in

detail in Chapter 3.2 and will therefore only be presented in a simplified way. The contents of the header `BeaconApiFuncs.h`, which is provided for the loader and the caller, is shown in Listing 12:

```

1 #include "wintypes.h"
2
3 typedef struct { /*...*/} datap_t;
4 typedef struct { /*...*/} formatp_t;
5
6 // Data Parser API
7 typedef void (*BeaconDataParse_t)(datap_t* parser, char* buffer, int size);
8 typedef int (*BeaconDataInt_t)(datap_t* parser);
9 typedef short (*BeaconDataShort_t)(datap_t* parser);
10 typedef int (*BeaconDataLength_t)(datap_t* parser);
11 typedef char* (*BeaconDataExtract_t)(datap_t* parser, int* size);
12 // Format API
13 typedef void (*BeaconFormatAlloc_t)(formatp_t* format, int maxsz);
14 typedef void (*BeaconFormatReset_t)(formatp_t* format);
15 typedef void (*BeaconFormatFree_t)(formatp_t* format);
16 typedef void (*BeaconFormatAppend_t)(formatp_t* format, char* txt, int len);
17 typedef void (*BeaconFormatPrintf_t)(formatp_t* format, char* fmt, ...);
18 typedef char* (*BeaconFormatToString_t)(formatp_t* format, int* size);
19 typedef void (*BeaconFormatInt_t)(formatp_t* format, int value);
20 // Output API
21 typedef void (*BeaconPrintf_t)(int type, char* fmt, ...);
22 typedef void (*BeaconOutput_t)(int type, char* data, int len);
23 // Token API
24 typedef BOOL (*BeaconUseToken_t)(HANDLE token);
25 typedef void (*BeaconRevertToken_t)(void);
26 typedef BOOL (*BeaconIsAdmin_t)(void);
27 // Utility API
28 typedef BOOL (*toWideChar_t)(char* src, wchar_t* dst, int max);
29
30 typedef struct cs_compat_functions {
31     // Data Parser API
32     BeaconDataParse_t BeaconDataParse;
33     BeaconDataInt_t BeaconDataInt;
34     BeaconDataShort_t BeaconDataShort;
35     BeaconDataLength_t BeaconDataLength;
36     BeaconDataExtract_t BeaconDataExtract;
37
38     // all other beacon functions...
39 } cs_compat_functions_t, *cs_compat_functions_ptr_t;

```

Listing 12: Type and struct definitions for the beacon API functions for the BOF loader

The pointers to the beacon APIs, as well as `GetProcAddress`, `LoadLibraryA`, `GetModuleHandle` and `FreeLibrary`, are mapped from the caller struct into a stack-allocated `void*` array right after the runtime is invoked, to hide the identities of the functions – especially their name and type – from EDRs.

Implementing the Public API

According to the defined requirements, the public API must consist of at least two functions: one for invoking the BOF itself and one for preparing the size-prefixed binary blob from a string (**RA24**). For the string format, the `trustedsec/COFFLoader` project is consulted once more: the string is generated using the script `beacon_generate.py`, which uses the “struct” Python library for this purpose. The following argument types specified by Cobalt Strike (see Chapter 3.2, Table 2) correspond to these function calls in Python, with `raw_val` being the actual user-provided argument:

- `int16`: Little-endian short (2 bytes)
`struct.pack("<h", int(raw_val))`
- `int32`: Little-endian int (4 bytes)
`struct.pack("<i", int(raw_val))`
- `string`: Little-endian unsigned long as size prefix, with UTF-8 encoded `char[]` of this size (string size + 4 bytes)
`s = raw_val.encode("utf-8")`
`struct.pack(f"<L{len(s)}s", len(s)+1, s)`
- `wchar`: Little-endian unsigned long as size prefix, with UTF-16_le encoded `char[]` of this size (string size + 4 bytes)
`s = raw_val.encode("utf-16_le")`
`struct.pack(f"<L{len(s)}s", len(s)+1, s)`

The encoded arguments are appended one by one onto a buffer. At the end, this buffer is prefixed once again with a little-endian unsigned long, which represents the size of all arguments. This buffer is output as a hex string. The size-prefixed binary blob for the runtime can now be obtained from this format. The function listed in Listing 13 is provided as a public API for this purpose:

```
1 unsigned char* /*binary blob*/ UnhexlifyArgs(  
2     external_functions_ptr_t external_functions,  
3     unsigned char* value,           // hex string  
4     int* outlen                     // [out] length of binary blob  
5 );
```

Listing 13: Function signature of `UnhexlifyArgs` from the BOF runtime

Since the function has to allocate the return value on the heap itself, it must also be supplied with the `external_functions_ptr_t` struct, which contains the required functions `HeapAlloc` and `GetProcessHeap`. Finally, this API function fulfills the requirements **RA23** and **RA24**.

Next, the function for transferring and executing the BOF needs to be added. Since no global variables may be used, it makes little sense to provide separate functions for initializing and invoking the runtime. Instead, a single function having all required arguments is provided, as shown in Listing 14.

```

1 int /*exit code*/ RunBOF(
2     external_functions_ptr_t external_functions,
3     cs_compat_functions_ptr_t compat_functions,
4     char* functionname,                // e.g. "go"
5     unsigned char* coff_data,          // raw BOF buffer
6     uint32_t filesize,                 // size of coff_data
7     unsigned char* argument_data,     // return of UnhexlifyArgs
8     int argument_size                  // outlen of UnhexlifyArgs
9 );

```

Listing 14: Function signature of RunBOF from the BOF runtime

Ultimately, the following parameters fulfill the following requirements:

- external_functions and compat_functions: **RA26**
- functionname: **RA25**
- coff_data: **RA22**
- argument_data: **RA23**

Implementation of the Loading Process

The only thing still missing is the implementation of the loader component. The function definition of RunBOF already provides it with all the necessary parameters for the loading process.

The mapping of the BOF into memory is carried out section by section. To do this, an index array is created in which the raw bytes of each section are stored. The relocations are carried out within these section copies and finally the entry point function is searched for and executed within it.

The COFF data structures used to process the raw BOF bytes in a simpler and more stable way were obtained partly from the official Microsoft documentation on the file format [54] and partly from trustedsec/COFFLoader.

The following is a high-level description of the steps required to implement the loading process from loading the BOF to executing the entry points. This approach follows the results from Chapter 3.4:

1. Cast the raw BOF bytes into a COFF data structure.
2. Create the index array to access all sections later on.
3. Iterate over each section header of the BOF using the NumberOfSections field in the file header.
 1. Get the section header data via an offset that is calculated as follows:
 $\text{BOF base pointer} + \text{sizeof(file header)} + (\text{sizeof(section header)} \cdot \text{section counter})$
 2. Copy raw data of the section into the index array.

4. Iterate over each section again to perform the relocations.
 1. Get the section header (as done in step 3.1).
 2. Get the pointer to the relocations via the offset:
BOF Base Pointer + PointerToRelocations in current section header
 3. Iterate over all relocations using `NumberOfRelocations` in the current section header.
 1. Get the corresponding symbol from the symbol table using `SymbolTableIndex` of the current relocation entry.
 2. Check whether the symbol is internal or external.
 3. *If symbol is external*: Find out the target address using the `process_symbol` auxiliary function (see next paragraph).
 4. Perform the relocation within the index array entry based on the relocation type.
 5. Set the iterator to the next relocation entry.
 4. Set the iterator to the next section.
5. Iterate again over all symbols and search for the entry point function.
6. Call the entry point function with the parameters passed to the BOF.

The following, rather simplified source code in Listing 15 describes this algorithm. The actual fields of the COFF data structures are used.

```
1 // Allocate an array to keep track of the sections
2 sectionMapping = (char**)external_functions->HeapAlloc(
3     external_functions->GetProcessHeap(), 0,
4     sizeof(char*) * (coff_header_ptr->NumberOfSections + 1)
5 );
6
7 // Iterate over section headers
8 for (counter = 0; counter < coff_header_ptr->NumberOfSections; counter++){
9     // Get the current section header
10    coff_sect_ptr = (coff_sect_t*)(
11        coff_data + sizeof(coff_file_header_t) +
12        (sizeof(coff_sect_t) * counter)
13    );
14    relocationCount += coff_sect_ptr->NumberOfRelocations;
15
16    // Allocate memory for this section in sectionMapping
17    sectionMapping[counter] = (char*)external_functions->VirtualAlloc(
18        NULL, coff_sect_ptr->SizeOfRawData,
19        MEM_COMMIT | MEM_RESERVE | MEM_TOP_DOWN, PAGE_EXECUTE_READWRITE
20    );
21
22    // Copy raw data to sectionMapping[counter]
23    memcpy(
24        sectionMapping[counter],
25        coff_data + coff_sect_ptr->PointerToRawData,
```

```

26     coff_sect_ptr->SizeOfRawData
27 );
28 } // end of section header loop
29
30 // Iterate over sections again to do the relocations
31 for (counter = 0; counter < coff_header_ptr->NumberOfSections; counter++){
32     // Get section and relocations
33     coff_sect_ptr = (coff_sect_t*)(
34         coff_data + sizeof(coff_file_header_t) +
35         (sizeof(coff_sect_t) * counter)
36     );
37     coff_reloc_ptr = (coff_reloc_t*)(
38         coff_data + coff_sect_ptr->PointerToRelocations
39     );
40
41     // Iterate over the relocations in this section
42     for (
43         reloccount = 0;
44         reloccount < coff_sect_ptr->NumberOfRelocations;
45         reloccount++
46     ){
47         // Check if symbol entry refers to a section in this COFF file
48         // --> internal symbol
49         if (
50             coff_sym_ptr[coff_reloc_ptr->SymbolTableIndex].SectionNumber != 0
51         ) {
52             // Get pointer to symbol
53             symptr =
54                 coff_sym_ptr[coff_reloc_ptr->SymbolTableIndex].first.value[1];
55
56             // Perform the relocation based on the relocation type
57             if (coff_reloc_ptr->Type == IMAGE_REL_AMD64_ADDR64) { // example
58                 uint64_t offval = 0; // offset value
59                 coff_sym_t sym = coff_sym_ptr[coff_reloc_ptr->SymbolTableIndex];
60
61                 memcpy( // copy VirtualAddress to offval
62                     &offval,
63                     sectionMapping[counter] + coff_reloc_ptr->VirtualAddress,
64                     sizeof(uint64_t)
65                 );
66                 offval = (uint64_t)( // add section number
67                     sectionMapping[sym.SectionNumber - 1] +
68                     (uint64_t)offval
69                 );
70                 offval += sym.Value; // add relocation value

```

```

71     memcpy( // copy offval back to sectionMapping --> done
72         sectionMapping[counter] + coff_reloc_ptr->VirtualAddress,
73         &offval,
74         sizeof(uint64_t)
75     );
76 }
77 else if (coff_reloc_ptr->Type == IMAGE_REL_AMD64_ADDR32NB) {...}
78 else if (coff_reloc_ptr->Type == IMAGE_REL_AMD64_REL32) {...}
79 // ...
80 }
81 else { // --> external symbol
82     // Get pointer to symbol
83     symptr =
84         coff_sym_ptr[coff_reloc_ptr->SymbolTableIndex].first.value[1];
85
86     // Get pointer to external function
87     // (the process_symbol function is where DFR happens!)
88     funcptrlocation = process_symbol(
89         external_functions, InternalFunctions,
90         ((char*)coff_sym_ptr + coff_header_ptr->NumberOfSymbols) + symptr
91     );
92
93     // Do relocation with funcptrlocation
94     if (coff_reloc_ptr->Type == IMAGE_REL_AMD64_ADDR64) {...}
95     else if (coff_reloc_ptr->Type == IMAGE_REL_AMD64_REL32) {...}
96     // ...
97 } // end of relocations (if-else for symbol entry name)
98
99 // Set iterator to the next section
100 coff_reloc_ptr = (coff_reloc_t*)(
101     ((char*)coff_reloc_ptr) + sizeof(coff_reloc_t)
102 );
103 } // end of relocation loop
104 } // end of section loop
105
106 // All relocations done!
107 // Search for entry function, by iterating over all symbols again
108 for (temp = 0; temp < coff_header_ptr->NumberOfSymbols; temp++) {
109     if (strncmp(
110         coff_sym_ptr[temp].first.Name,
111         entryfuncname,
112         sizeof(coff_sym_ptr[temp].first.Name) + 1
113     ) == 0) { // symbol name == entryfuncname
114         // define go function signature
115         go = (void(__cdecl*)(char*, unsigned long /* entry point args */))(

```

```

116         sectionMapping[coff_sym_ptr[temp].SectionNumber - 1] +
117         coff_sym_ptr[temp].Value
118     );
119     // call go --> this will execute the BOF
120     go((char*)argument_data, argument_size);
121 }
122 }

```

Listing 15: Simplified loading process of the BOF runtime

In step 4.3.3 of the algorithm and in lines 88-91 of Listing 15, it is described that the external helper function `process_symbol` is used to resolve external symbols to a function pointer. This function also requires the `external_funtions_ptr_t` struct, since `process_symbol` is responsible for resolving DFR signatures by calling the caller-implemented `Resolve` function, which is stored within this struct.

First, the function checks whether the passed symbol in the parameter `char *symbolstring` references a function already known to the loader. “Known functions” refers to all beacon API functions, as well as `GetProcAddress`, `LoadLibraryA`, `GetModuleHandleA` and `FreeLibrary`. These were already provisioned to the loader via the public API before BOF invocation. If the symbol references a known function, its pointer is looked up in the stack-allocated `void*` array `InternalFunctions` and returned from there. While doing so, `process_symbol` always ensures that external functions have the prefix `__imp_` or `__imp__`, as already described in Chapter 3.4. This prefix is represented by the precompiler macro `PREPENDSYMBOLVALUE` in the source code.

If the function is not known to the loader, it is assumed that the symbol is a DFR signature, since BOFs do not support other types of external symbols. If this is not a DFR signature, the `process_symbol` function will fail. To process DFR signatures, the string tokenizer function `strtok_r` is used to first split the symbol at the “\$” sign. The part before it represents the name of the library. The part after it is further separated at the “@” character. This character sometimes appears within the symbol because the MSVC compiler – depending on compiler optimization settings and whether C or C++ is used – encodes additional information for name mangling⁶ behind the “@” character [62]. They are not relevant for the loader and are thus discarded. The remaining string between “\$” and “@” ultimately represents the function name within the specified library. Both library and function names are passed to the `Resolve` function. The caller implementation of this function is now responsible for resolving these names and returning a pointer to the corresponding function.

The return value of the `process_symbol` function is thus either a function pointer from the `InternalFunctions` array or an external function pointer resolved by DFR. This pointer is relocated by the loader to the places in the BOF where the respective function calls occur. This fulfills requirement **RL12**. All DFR functions defined in `TrustedSec’s bofdefs.h` can be loaded using this function.

The simplified implementation of the `process_symbol` function is listed below in Listing 16.

⁶Method to realize function overloading

```

1 static void* process_symbol(
2     external_functions_ptr_t external_functions,
3     InternalFunctions_t InternalFunctions, // array with name -> void* mapping
4     char *symbolstring                    // name of the symbol with prefix
5 ) {
6     void* functionaddress = NULL;
7     // Check if symbolstring is a known -> internal function
8     if (
9         starts_with(symbolstring, PREPENDSYMBOLVALUE"Beacon") ||
10        starts_with(symbolstring, PREPENDSYMBOLVALUE"toWideChar") ||
11        starts_with(symbolstring, PREPENDSYMBOLVALUE"GetProcAddress") ||
12        starts_with(symbolstring, PREPENDSYMBOLVALUE"LoadLibraryA") ||
13        starts_with(symbolstring, PREPENDSYMBOLVALUE"GetModuleHandleA") ||
14        starts_with(symbolstring, PREPENDSYMBOLVALUE"FreeLibrary")
15    ) {
16        // strip __imp_ from symbol
17        localfunc = symbolstring + strlen(PREPENDSYMBOLVALUE);
18        // search for this function in InternalFunctions
19        for (int tempcounter = 0; tempcounter < 30; tempcounter++) {
20            if (InternalFunctions[tempcounter][0] != NULL) { // name exists
21                if (starts_with(
22                    localfunc,
23                    (char*)(InternalFunctions[tempcounter][0])
24                )) { // function matches localfunc name
25                    // take address of function
26                    functionaddress = (void*)InternalFunctions[tempcounter][1];
27                    return functionaddress;
28                }
29            }
30        }
31    }
32    // symbolstring is any other external symbol -> use DFR
33    else if (starts_with(symbolstring, PREPENDSYMBOLVALUE)) {
34        // Strip __imp_, then split DFR-definition at "$" and "@"
35        char *locallib = symbolstring + strlen(PREPENDSYMBOLVALUE);
36        char *saveptr1 = NULL, *saveptr2 = NULL;
37        locallib = strtok_r(locallib, "$", &saveptr1);
38        char *localfunc = strtok_r(NULL, "$", &saveptr1);
39        localfunc = strtok_r(localfunc, "@", &saveptr2);
40
41        // call external Resolve to get function pointer
42        functionaddress = external_functions->Resolve(locallib, localfunc);
43    }
44    return functionaddress; // might still be NULL if not found
45 }

```

Listing 16: Simplified implementation of symbol processing in the BOF runtime

Summary

Now that the loader component is able to perform relocations of all common types, the requirements **RL7** and **RL8** are finally fulfilled. By performing the relocation solely in memory, there are no side effects, for instance, on the file system. Accordingly, requirement **RL6** is also fulfilled.

Ultimately, apart from optimization issues, there are no unfulfilled requirements. For example, the use of the heap in the sense of **R3** can be further improved, for example, by not mapping sections without relocations in memory at all. These optimization issues are addressed again in the outlook of this thesis in Chapter 7.

The implementation of the BOF runtime is therefore considered complete.

5.4 Other Considerations

This section describes the considerations for the BOF runtime that were not adopted in the implementation. Reasons for this may be simple design decisions, but also faulty or incomplete implementations. This section documents other implementation approaches that could have led to different solutions under certain circumstances.

5.4.1 Linking BOFs on the Server

The most difficult task of the BOF runtime is the dynamic linking of the incoming BOF, i.e. to replicate what a classic linker actually does. However, the runtime is subject to many restrictions, since it is part of the beacon code, and therefore must adhere to OPSEC rules in order to avoid detection by EDR tools.

This increases the complexity of the runtime implementation. One consideration for avoiding this problem was to link the BOF before transport to the client – that is, on the C2 server side. In this case, a classic linker could be used to link the BOF against the beacon API implementation and, for example, the Windows API. This would have many advantages: for one thing, the linking process would not have to be emulated. As a result the BOF developer would face fewer restrictions. The implementation of the beacon APIs would not have to reside in the beacon itself, since it does not need to be linked there. This would slightly reduce the size of the beacon itself, which simplifies transport as shellcode. The same applies to a simpler loader implementation, which would further reduce the beacon size.

However, the disadvantages of this approach outweigh the advantages: as already mentioned in Chapter 3, linking increases the size of the payload considerably. It must, however, still be transferred through the C2 profile to remain inconspicuous. Larger payloads would therefore have to be chunked into many smaller parts, and might thus be even more conspicuous than unlinked BOFs due to the high traffic volume. Furthermore, using a ready-made executable in the beacon means losing all the advantages of BOFs and having to fall back on the fork and run pattern. For this reason, server-side linking was avoided.

This approach is still documented here, as it only has some minor drawbacks related to OPSEC. Employing a dynamic linking process, like the one used in this runtime, is considerably more complex.

In scenarios in which OPSEC is not as important, e.g., when no EDR is used for defense, the approach to pre-link the BOF server-side is still worth evaluating.

5.4.2 Resolving DFR Symbols on the Server

The limitation of not having access to standard library functions creates many hurdles, especially when using DFR. To find DFR function signatures, the runtime has to examine all symbols in the BOF and check whether their name contains a \$ according to the convention. The string before the \$ then corresponds to the library name, which has to be loaded with the `LoadLibraryA` function. After the \$, the name of the function from this library is specified, which must be loaded with `GetProcAddress`. Parsing this signature requires string processing functions, which are usually available in the standard library.

This led to the idea of processing the DFR symbols before the BOF is transported to the beacon. Then, the C2 server would send the BOF together with the DFR analysis results in a structured manner, e.g., by using the (de)serialization functions present within the Mythic translation container. The functions can then be resolved more easily by the beacon.

In fact, the effort to parse DFR symbols directly in the loader proved to be straightforward. The functions for string manipulation are often needed in other places in the loader, so it is not an additional effort to use them in the DFR implementation, since these are implemented manually by now.

5.4.3 Using a pre-built Windows Executable as a Runtime

This approach is less fundamentally based on the development of the BOF runtime itself, but includes the existing capabilities of `ciroStrike`. It is already able to run executables such as .NET assemblies.

The idea of this approach is to outsource the complexity of a possible BOF loader implementation to such a .NET assembly. This would have many advantages: since the .NET assembly is usually compiled as it is intended by Microsoft, none of the limitations that apply to an implementation directly in the beacon would apply here. This means that exception handling would be possible, memory management would be handled by the .NET runtime and therefore the use of global variables would not be restricted. In addition, it would be possible to use existing external libraries to parse the COFF file if the assembly were compiled as self-contained (external dependencies are statically linked and embedded in the assembly).

This approach would then work in such a way that the BOF runtime is not included in the beacon by default. If the operator now wants to execute a BOF, they would send it together with the .NET assembly to the beacon and then execute it via an `execute-assembly` command. This would first invoke the .NET loader and then the .NET loader would execute the BOF.

Although this approach would simplify many aspects of the implementation, it misses the goal of a BOF runtime: it would again use the Fork and Run pattern which is to be avoided and would have a higher risk of being detected due to the significantly larger memory footprint. The memory footprint is increased by the comparatively enormous size of a .NET assembly as opposed to shellcode or BOFs.

Furthermore, this would miss the research goal of this thesis: the result would not be a versatile BOF runtime usable in most environments, but would again have all the disadvantages associated with .NET – albeit with higher stability. Therefore, this approach was discarded.

5.4.4 Implementing an Exemplary BOF Using all Beacon APIs

The initial idea at the beginning of the thesis was to create a separate BOF to test the runtime and the implementation of the beacon APIs. This BOF, which would have been called “Use-it-all BOF”, should accordingly call all functions of the beacon API at least once. Furthermore, the quality of life features mentioned in Chapter 3.1 such as DFR should also be used. The BOF should serve as a stress test for the runtime.

The following features should be utilized and tested by the Use-it-all BOF, most of them being common C++ runtime features:

- Declaration of a DFR function, residing in `kernel32.dll`.
- Calling a DFR-resolved function and using its return value.
- Local function linking (using self-defined functions outside the go function).
- Using global non-constant values.
- Allocating and freeing memory on the processes default heap.
- Working with pointers to arrays.
- Calling all beacon API functions, except for the following API groups:
 - User Data API, since UDRL is not required, nor supported by Mythic.
 - Beacon Gate API, since it is also not supported by Mythic and there is no comparable feature.

As the thesis progressed, it became evident that implementing this idea is highly complex: many of the beacon APIs – including the more complex ones – are designed to support sophisticated attacks: the Token API can be used to override the process token and the Spawn+Inject API can be used to inject shellcode into foreign processes. Attacks of this kind often require extensive preliminary work: creating or stealing another process’s privileged token or creating effective, functioning shellcode is not trivial and, furthermore, very much deviates from the topic of this thesis. It is also difficult to find suitable examples for Syscalls that would be useful in an exemplary BOF – especially since Syscalls are often implemented manually by BOF developers instead of using the beacon Syscall API.

Along with the limited use of certain APIs, as recorded in Chapter 4.3, the benefit of such a BOF has been greatly reduced. Since the 15 collected BOFs for Chapter 4.1 already cover a large part of the intended use cases that the runtime should fulfill, the development of the Use-it-all BOF was discontinued.

For documentation purposes and potential further development, the latest version of the Use-it-all BOF is shown in Appendix E. In this version, the Token API and Data Store API are only partially implemented, while the Key/Value Store API implementation is not functional. Parts of the examples were taken from Cobalt Strike’s BOF template repository [63].

6 Implementation into ciroStrike

Now that the BOF runtime is fully developed and fulfills all defined requirements, it needs to be implemented into cirosec's own C2 beacon ciroStrike. This chapter covers all the necessary adjustments to the ciroStrike source code, as well as the adjustments needed for integration into Mythic.

First, the current state of ciroStrike is described, highlighting the restrictions that are relevant for further development and operation. This should make it clear why there were such a significant number of restrictions when developing the BOF runtime and why there are so many of them in the ciroStrike environment. It should also justify design decisions, such as why the implementation of the DFR resolution function and the beacon APIs have to be carried out by the caller (in this case, ciroStrike itself).

After that, the actual implementation of the BOF runtime into ciroStrike is documented. The implementation of the beacon APIs and the `Resolve` function for DFR will also be explained here. In the transition to the next section, the interaction between the beacon and the Mythic server will be discussed and how it is influenced by the beacon APIs.

The last sections cover the adjustments to the beacon's Mythic code and the translation container. This is the interface between the C2 operator and the beacon and is therefore of great importance. The missing integration of the Aggressor Script within this thesis is also addressed again here. It is described how this problem could be avoided with a new tool called "Forge", which just appeared during the course of this thesis. The integration of Forge into the ciroStrike infrastructure is carried out and explained here.

6.1 Current State of ciroStrike

ciroStrike is a Mythic C2 beacon developed by a small team of red teamers at cirosec. It is built using the `clang-cl` compiler to achieve a higher level of compliance with the C/C++ standards than with using MSVC. It comes with custom C2 profiles and a matching custom translation container. Development is carried out closed source in order to stay invisible to EDR software.

The main argument for ciroStrike over other beacons is that it is built natively as shellcode. Although other beacons also offer conversion to shellcode, they accomplish this exclusively by using a wrapper: the beacon itself is always built as a PE image⁷, then converted to shellcode and wrapped by a minimal PE loader. This approach has two major drawbacks. First, the resulting shellcode is very large in size because it effectively contains a full PE image. This makes some operations more difficult or at least slower to perform. One example is the abuse of Microsoft's internal code protection and obfuscation framework "Warbird" as a shellcode loader, which has been documented by cirosec in a blog post [64]. This approach requires a maximum shellcode size of about 64 KB, which is impossible to achieve in most cases with the wrapper approach. The second disadvantage is the detectability of the PE image in memory. The resulting shellcode, along with the loader and PE image, is eventually stored in the memory of a process. The image would be detected by memory scans because the structure and parts of

⁷"PE image" is a phrase which includes files that rely on the Portable Executable specification, like `.exe` or `.dll`.

the contents of PE files are constant by specification. It is therefore essential that the image is obfuscated within the shellcode. However, obfuscation increases entropy, which some EDRs might also detect.

ciroStrike takes a different approach. Restrictions associated with shellcode generation are accepted during development. These include, for example, the often-mentioned avoidance of the use of global variables or the C standard library. After the project has been compiled with default compiler configuration, the generated code is extracted directly from the `.text` and `.rdata` sections using a complex post-build pipeline. All other sections are omitted. By adhering to the shellcode restrictions, this extract is intrinsically position-independent and can be used as shellcode. In an additional test pipeline, this shellcode is embedded in a default configured PE executable and executed from there to ensure the function of the shellcode. With this approach, a shellcode size just under the limiting 64 KB is currently achieved. To maintain this state, the restrictions established in this thesis are adhered to.

Additionally, *ciroStrike*'s build system is designed so that individual command implementations can also be disabled to further reduce the size. Although *Mythic* provides a dedicated build system for creating beacon payloads directly from the web interface, it is not used in this instance. Since *ciroStrike* currently uses the Microsoft Visual Studio build system, it is difficult to integrate it into *Mythic*'s build system. Instead, *Mythic* generates a configuration file that can be passed to Microsoft's build system. In a pre-build step, it translates the configuration file into precompiler variables to adjust the source code accordingly. This is especially beneficial, as the implementations of the invocation commands for reflective DLL loading and shellcode execution cause the largest increases in shellcode size. The same applies to the comparatively large implementation of the added BOF runtime, which exceeds the shellcode size limit by about 18 KB. In fact, several BOFs are designed to exactly replicate some commands already built into *ciroStrike*. This way, these commands – including smaller utility commands like `whoami` but also larger commands like `execute_dll` and `execute_shellcode` – can be disabled in *ciroStrike* without losing functionality. This results in an even smaller shellcode size than before. Furthermore, new additional functionality for *ciroStrike* can now be implemented as BOF without increasing the shellcode size. Additionally, this creates a certain flexibility for the operator, as they can now decide whether they prefer to work with BOFs or with the built-in commands.

ciroStrike also implements further protective measures against EDR software, such as string and symbol obfuscation. The most relevant aspect, however, is the reloading of Windows APIs: the required functions can be specified in a Comma-Separated Values (CSV) file and are embedded in the source code by a pre-build script. When the beacon is started, the functions are resolved using a technique called “PEB-Walk”. This is an OPSEC-safe method to use Windows APIs even though they are not available in the project by default. This technique is the reason why the external functions for the BOF runtime were designed in such a way that they have to be passed by the caller (*ciroStrike*): the BOF runtime implicitly adheres to *ciroStrike*'s convention. The same applies to the resolver function used for DFR – in this context, *ciroStrike* should be able to define how external functions are to be resolved.

With regard to the implementations of the beacon APIs, the goal is to enable communication with *Mythic*. In the Output API, the goal is to send logs to the *Mythic* operator. *ciroStrike* implements its own namespace called “*Mythic*”, in which all functions for communicating with the C2 server are implemented.

These must be accessible from the beacon API implementations. The namespace is exposed through the `Mythic.h` header.

6.2 Beacon Implementation

Before starting the implementation, the desired architecture must be defined. It is shown in Figure 8: the Mythic server sends a BOF invocation command (called a “task” in Mythic) to ciroStrike, which downloads the BOF into memory and links it against the `beacon.h`. The functions it contains – known as compatibility layer – are implemented in ciroStrike itself (“internal beacon code”) and forwarded to the runtime. This enables the BOF to interact with the infrastructure by using the beacon-implemented beacon APIs.

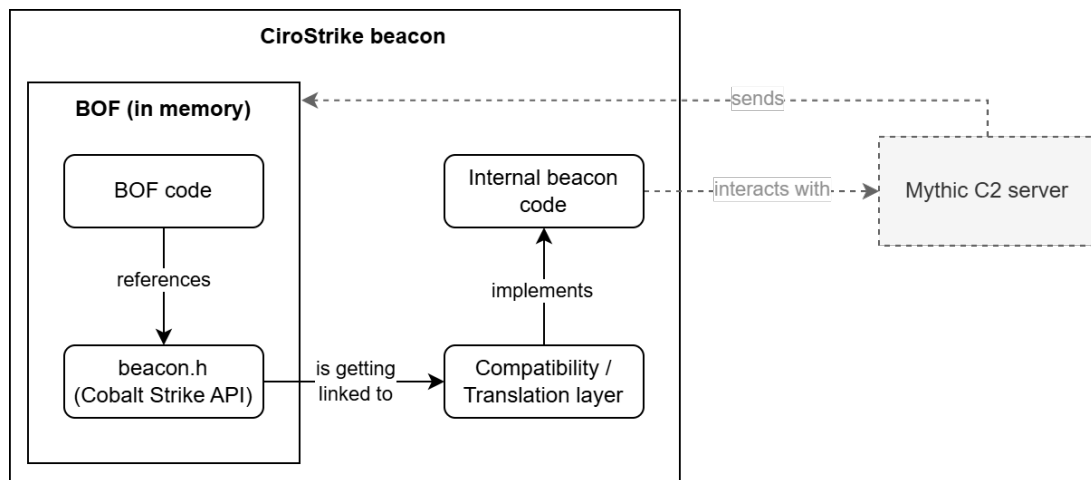


Figure 8: Schema on how a BOF interacts with the beacons internal code (own illustration)

Runtime Integration and Command Apposition

The first step is to integrate the BOF runtime into the ciroStrike codebase. To do this, the BOF runtime project is first added to the solution. However, a few minor adjustments were made to avoid code duplication: the `Memory.h` was removed because ciroStrike already provides its own implementations for `memcpy` and `memset`. The same applies to the recreated definitions of Windows types (`HANDLE`, `LPVOID`, etc.) in `wintypes.h`, which are also already available in ciroStrike in a different header. Otherwise, no further adjustments had to be made to the runtime.

Next, the logic for the new command has to be added to ciroStrike. The current codebase already implements an abstraction layer over commands sent from the Mythic server, so only a few structures and enumerations need to be customized: first, the `execute_bof` union field has to be added to the command struct, where the arguments for it are also defined:

- `bof_file_id` (`Mythic::UUID`): A file ID for the uploaded BOF given by the Mythic server. This ID is used by the beacon to invoke the download of the file at any time.
- `args_hexlified` (`Array<wchar_t>`): Contains the hex string with BOF arguments that will be passed to the `UnhexlifyArgs` function of the runtime’s public API.

- `function_name` (`Array<wchar_t>`): Name of the entry point function symbol within the BOF.
- `chunk_size` (`uint32_t`): Number in bytes indicating the maximum size of a chunk of the BOF file that will be downloaded from the beacon. Smaller chunk sizes reduce the size of a single transaction and thereby reduce detectability at the expense of transfer time.

Listing 17 shows the partial definition of the `Command` struct in `ciroStrike` with the newly added `execute_bof` command. The implementation of the `Serializer` type is used to read/write the command type and command arguments sent from/to the Mythic server, e.g., the translator. Its concrete implementation is not relevant for this thesis.

```

1 // This struct represents a command sent from the Mythic server.
2 // It contains the command type and the command definitions, along
3 // with their arguments and a corresponding (de)serializer for them.
4 struct Command {
5     CommandType type;
6     union { // actual command defined by `type`
7         struct { uint32_t code; } exit;
8         struct { } pwd;
9         struct { Array<wchar_t> path; } ls;
10        // ... other commands
11        struct {
12            Mythic::UUID bof_file_id;
13            Array<wchar_t> args_hexlified;
14            Array<wchar_t> function_name;
15            uint32_t chunk_size;
16        } execute_bof;
17    } args;
18
19    void Serialize(Serializer s) const {
20        s << static_cast<uint32_t>(type);
21        switch (type) {
22            // ... other commands
23            case CommandType::EXECUTE_BOF:
24                s << args.execute_bof.bof_file_id
25                << args.execute_bof.args_hexlified
26                << args.execute_bof.function_name
27                << args.execute_bof.chunk_size;
28            break;
29        }
30
31    void Deserialize(Serializer s) {
32        uint32_t raw_type;
33        s >> raw_type;
34        type = CommandType(raw_type);
35        switch (type) {
36            // ... other commands
37            case CommandType::EXECUTE_BOF:
38                s >> args.execute_bof.bof_file_id
39                >> args.execute_bof.args_hexlified
40                >> args.execute_bof.function_name
41                >> args.execute_bof.chunk_size;
42            break;
43        }
44    }; // struct end

```

Listing 17: Reduced Command struct definition in ciroStrike

After that, the command EXECUTE_BOF must be added to the CommandType enum, so that the type `Commands::CommandType::EXECUTE_BOF` can be switched when processing incoming commands. The command implementation is called within this case. Listing 18 shows the header for the `command_execute_bof` function and Listing 19 shows the switch instruction that calls the function. The `run_task` function described there is called for each incoming task from the Mythic server and first determines the command type.

```

1 // Header definitions for all commands
2 namespace Commands {
3     namespace Impls {
4         // ... other commands
5         Mythic::Request::TaskResult command_execute_bof(
6             WINAPI_LIBRARIES* pWinApi, // function pointers to Windows APIs
7             Mythic::Response::Task&& task, // task information
8             SleepDuration sleep, // other Mythic related parameters
9             Mythic::UUID callback_id, // "
10            C2PROTOCOL_t proto // "
11        );
12    }
13 }

```

Listing 18: Reduced headers for commands supported by ciroStrike

```

1 // run_task is invoked for every incoming task from the Mythic server
2 Mythic::Request::TaskResult run_task(Mythic::Response::Task&& task) {
3     switch (task.cmd.type) {
4         // ... other commands
5         case Commands::CommandType::EXECUTE_BOF:
6             return Commands::Impls::command_execute_bof( // command from header above
7                 m_pWinApi, // function pointers to Windows APIs
8                 static_cast<Mythic::Response::Task&&>(task), // task information
9                 m_sleep, m_id, m_proto // other Mythic related parameters
10            );
11    }

```

Listing 19: Reduced command type switch for the execute_bof command in ciroStrike

Defining the External Functions

The function now has all the necessary information to download the BOF from the Mythic server and prepare the arguments for it using `UnhexlifyArgs`. Before invoking the runtime, the external functions and the beacon APIs must now be defined. For the definition of the external functions, only the `Resolve` function needs to be implemented manually. It currently only uses `LoadLibraryA` for the library name parameter and `GetProcAddress` for the library and function name parameter, for the sake of simplicity.

For all other external functions, the pointers of the functions already resolved by `ciroStrike` can simply be passed to the corresponding struct. The required functions only need to be added to the CSV file explained in the section above for `ciroStrike`'s pre-build script to generate their function definitions. However, an exception to this is `HeapAlloc`, where the pointer cannot simply be passed on: since this is a memory-allocating function, it is provided with the `DECLSPEC_ALLOCATOR` attribute by Windows, in contrast to `VirtualAlloc`, where this attribute is not applied to [65]. This has the effect that function calls to it are traceable by Event Tracing for Windows (ETW) [66]. EDR software can view and analyze these ETW traces and thus detect malicious memory allocations. Fortunately, `HeapAlloc` is underlain by another function with the same parameter signature: `RtlAllocateHeap` from the NT Sys API. This function is not exported by the `Windows.h` header, but it can be loaded manually via `GetProcAddress`. Since it contains almost the entire implementation of `HeapAlloc` and is parameterized in the same way, it can be used as a drop-in replacement for the latter. These adjustments are not necessary for `HeapFree` and `GetProcessHeap`, which is why the "original" functions are still used in the struct. The definition of the external function struct is fully set out in Listing 22 in lines 15-25.

Implementing the Beacon APIs

Following this, the beacon API functions must be implemented. As already specified in Chapter 5.1 in the requirements for BOF runtime and as shown in Figure 8, the implementation of the beacon API functions is the responsibility of the caller, i.e. the beacon itself.

The implementation of the Data Parser API is reasonably trivial. The functions merely read and modify members of the passed `datap_t` struct. For example, the `BeaconDataParse` function only initializes the empty struct. The other functions have no dependencies, except for the `memcpy` function, which has already been implemented in `ciroStrike`.

A major problem became apparent during the implementation of the Format API: the functions have no access to external functions. The reason for this is the fact that due to the shellcode restrictions, these are not globally available in `ciroStrike` – they have to be passed as parameters to be available on the function stack. However, the signatures of the functions cannot be customized because they have to exactly match the specifications of Cobalt Strike's `beacon.h`. BOFs use the functions with this signature and have no way of passing external functions to the beacon APIs on their own. Patching the parameters is also not possible because the runtime only relocates the symbols in the BOF with pointers – the runtime knows nothing of the abstract concept of a function.

To address this problem, attempts were first made, without success, to make the struct available using inline assembly without adapting the function's calling convention. The idea was to store a pointer to the external functions struct in a general-purpose, callee-saved register before the BOF runtime was invoked, such as in registers `r14` or `r15`. However, this approach was unsuccessful – the registers always contained other, invalid addresses when they were read, probably because there were some other function calls between BOF invocation and the reading of the registers. In addition, the use of lambda captures⁸ to keep the external functions in scope was unsuccessful: lambda captures are implicitly converted into minimal

⁸Mechanism of anonymous functions ("lambdas") to access its enclosing scope; available since the C11 standard

classes with implemented `operator()` function by the compiler. These classes cannot be treated like function pointers [67, section “Lambda capture”] and therefore cannot be used as relocation targets by the BOF runtime. This led to the realization that the implementation of the beacon API must be based on native functions.

As a last resort, using Windows’ so-called Thread Local Storage (TLS) was considered. This is memory space provided by the operating system that is globally available per thread. In the case of *ciroStrike*, this behaviour is applicable because the BOF runtime generally runs in the same thread as the BOF itself. If the runtime were to store data there, the BOF could access it. The TLS is usually accessed with these Windows API functions:

- `TlsAlloc` for allocating the so-called slots in the TLS.
- `TlsSetValue` for setting a value in a slot.
- `TlsGetValue` for reading a value from a slot.

However, these are not available in the beacon API functions in this case, which is exactly the problem that needs to be solved. As a workaround, thread-local data can be stored manually by directly accessing the Thread Environment Block (TEB). On 64-bit Windows systems, the TEB is accessible via the `gs` segment register at offset `0x30`. The pointer at `gs:[0x30] + 0x58` references the base address of the thread’s TLS slots. By directly reading and writing to this structure, custom thread-local storage can be implemented without relying on the Windows API.

This technique involves checking if the TLS array is already allocated and, if not, allocating it manually with a fixed size (e.g., 64 slots). An index can then be used to store arbitrary thread-local pointers. This approach ensures compatibility with single-threaded or controlled-thread environments like those used in BOF execution contexts. While undocumented in official API documentation, the structure and offsets of the TEB are well-known in the Windows internals community and are consistent across versions [68].

Using this knowledge, some utility functions can be implemented to help with this approach, as shown in Listing 20. They only use compiler intrinsics to access the TEB via the `gs` segment register and only the setter function requires access to some external functions. The functions are made available to *ciroStrike* through the `CustomTls.h` header.

Before BOF invocation, a pointer to the external functions can now be stored in slot 0 of the TLS. With this self-implemented TLS, however, more data than just the external functions can be accessed from anywhere in the code like they were stored globally. For example, the Output API needs the callback ID of the beacon, so that transmitted output can be associated with the correct callback on the server again. The callback ID is stored in slot 1. Both slots are overwritten with null pointers after the BOF invocation using the same setter function.

```

1 inline void tls_set(external_functions_ptr_t ex_f, size_t index, void* ptr) {
2     uintptr_t teb = __readgsqword(0x30); // get TEB from gs:[0x30]
3     void*** tlsArrayPtr = (void***)(teb + 0x58); // get TLS array
4     // check if it is already allocated, self-allocate if not
5     if (*tlsArrayPtr == NULL) {
6         *tlsArrayPtr = (void**)ex_f->HeapAlloc(
7             ex_f->GetProcessHeap(), 0, 64 * sizeof(void*) // space for 64 pointers
8         );
9     }
10    (*tlsArrayPtr)[index] = ptr; // set data
11 }
12
13 inline void *tls_get(size_t index) {
14     uintptr_t teb = __readgsqword(0x30);
15     void*** tlsArrayPtr = (void***)(teb + 0x58);
16     if (*tlsArrayPtr == NULL) return NULL;
17     return (*tlsArrayPtr)[index]; // get data
18 }

```

Listing 20: Functions to access the Thread Local Storage without using the Windows API

The getter `get_tls` can now be used to retrieve the slots within the beacon API implementations. Listing 21 shows how `BeaconFormatAlloc` can access the `HeapAlloc` function by reading slot 0 from TLS without having to modify the function signature.

```

1 #include "CustomTls.h"
2
3 // Output API implementation ...
4
5 void BeaconFormatAlloc(formatp_t* format, int maxsz) {
6     // get external functions from TLS at index 0
7     external_functions_ptr_t ex_f =
8         reinterpret_cast<external_functions_ptr_t>(tls_get(0));
9
10    if (format == NULL) return;
11    // HeapAlloc can now be used here!
12    format->original = ex_f->HeapAlloc(ex_f->GetProcessHeap(), 0, maxsz);
13    format->buffer = format->original;
14    format->length = 0;
15    format->size = maxsz;
16    return;
17 }
18
19 // ...

```

Listing 21: Implementation of BeaconFormatAlloc using the custom TLS implementation

Another problem that arose during the implementation of the Format API was related to `BeaconFormatPrintf`: the function uses the usual string formatting syntax, just like the standard library function `sprintf`. It takes a format string with placeholders and so-called variadic arguments (function arguments of unknown quantity; in the case of `sprintf`, these are the arguments right after the format string). These arguments must be embedded in the format string. Since `sprintf` cannot be used here, the processing of them must be implemented manually. The macros `va_start`, `va_end`, and the type `va_args`, which are commonly used to accomplish this, can be used here because they can be obtained not only from the standard library but also from the static implementation in `<cstdarg>` [69]. However, this does not apply to the function `vsnprintf`, which actually performs the embedding of `va_args`: it comes from the standard library. Instead, the Windows counterpart `wvsprintfA` [70] can be used, which is also passed to `BeaconFormatPrintf` via TLS. The implementation of the Format API is thus also complete.

Since the beacon's callback ID has already been stored in the TLS, implementing the Output API is also trivial. `BeaconOutput` transmits the passed string together with the callback ID to the Mythic server. `BeaconPrintf` simply uses the implementation of the Format API and that of `BeaconOutput` internally.

The Token and Utility API can also be implemented trivially. The functions only represent small macros for functions that are already provided by the Windows API. The implementation consists of the following mappings between beacon API function and Windows API function:

- `BeaconUseToken(HANDLE token) → SetThreadToken(token)`
- `BeaconRevertToken() → RevertToSelf()`
- `BeaconIsAdmin() → Check TOKEN_ELEVATION_TYPE from GetTokenInformation(...)` output
- `toWideChar(...)` → `MultiByteToWideChar(...)`

The functions from the Windows API only have to be resolved by `ciroStrike` and can then be passed via TLS. This completes the implementation of all beacon APIs.

The function pointers to the beacon API functions can now be passed to the `cs_compat_functions_t` struct (Listing 22, lines 27-45) and can then be handed over to the BOF runtime at invocation (Listing 22, line 58).

Summary

Listing 22 shows a simplified version of the final implementation of the `execute_bof` command. In summary, the BOF is first downloaded from the Mythic server, taking the chunk size into account. Then the external functions and the compatibility functions, that have just been implemented, are specified in their corresponding structs and the BOF arguments are prepared. Finally, the BOF runtime is invoked, and after it finished, the return code is processed and reported back to Mythic.

```

1 // some includes hidden
2 #include "BOFLoader.h" // this is the BOF runtime
3 #include "execute_bof.h" // this is the Mythic command header
4
5 #ifdef CONFIG_COMMAND_execute_bof // variable to (de)activate this command
6 Mythic::Request::TaskResult Commands::Impls::command_execute_bof(
7     WINAPI_LIBRARIES* pWinApi, Mythic::Response::Task&& task,
8     SleepDuration sleep, Mythic::UUID callback_id, C2PROTOCOL_t proto
9 ) {
10 // Request BOF download from Mythic by file ID, respecting chunk_size
11 // (download procedure hidden for simplicity, BOF is in `file_buffer`)
12 uint64_t bof_size = file_buffer.size();
13 uint8_t* bof_buffer = file_buffer.data();
14
15 external_functions_t external_functions = { // External functions
16     (VirtualAlloc_t)pWinApi->kernel32.VirtualAlloc,
17     (VirtualFree_t)pWinApi->kernel32.VirtualFree,
18     (HeapAlloc_t)pWinApi->ntdll.RtlAllocateHeap, // uses "RtlAllocateHeap"!
19     (HeapFree_t)pWinApi->kernel32.HeapFree,
20     (GetProcessHeap_t)pWinApi->kernel32.GetProcessHeap,
21     (LoadLibraryA_t)pWinApi->kernel32.LoadLibraryA,
22     (GetModuleHandleA_t)pWinApi->kernel32.GetModuleHandleA,
23     (GetProcAddress_t)pWinApi->kernel32.GetProcAddress,
24     (FreeLibrary_t)pWinApi->kernel32.FreeLibrary,
25     (Resolve_t)Resolve };
26
27 cs_compat_functions_t compat_functions = { // CS Compat functions
28     (BeaconDataParse_t)BeaconDataParse,
29     (BeaconDataInt_t)BeaconDataInt,
30     (BeaconDataShort_t)BeaconDataShort,
31     (BeaconDataLength_t)BeaconDataLength,
32     (BeaconDataExtract_t)BeaconDataExtract,
33     (BeaconFormatAlloc_t)BeaconFormatAlloc,
34     (BeaconFormatReset_t)BeaconFormatReset,
35     (BeaconFormatFree_t)BeaconFormatFree,
36     (BeaconFormatAppend_t)BeaconFormatAppend,
37     (BeaconFormatPrintf_t)BeaconFormatPrintf,
38     (BeaconFormatToString_t)BeaconFormatToString,
39     (BeaconFormatInt_t)BeaconFormatInt,
40     (BeaconPrintf_t)BeaconPrintf,
41     (BeaconOutput_t)BeaconOutput,
42     (BeaconUseToken_t)BeaconUseToken,
43     (BeaconRevertToken_t)BeaconRevertToken,
44     (BeaconIsAdmin_t)BeaconIsAdmin,
45     (toWideChar_t)toWideChar };

```

```

46
47 // Prepare arguments
48 int arg_size = 0;
49 unsigned char *prepared_args = UnhexlifyArgs(
50     &external_functions,
51     (unsigned char*)task.cmd.args.execute_bof.args_hexlified.get(),
52     &arg_size
53 );
54
55 // Invoke BOF runtime
56 int retcode = RunBOF(
57     &external_functions,
58     &compat_functions,
59     (char*)task.cmd.args.execute_bof.function_name.get(),
60     (unsigned char*)bof_buffer,
61     bof_size,
62     prepared_args,
63     arg_size
64 );
65
66 // Check `retcode` and report back to Mythic
67 // (hidden for simplicity)
68 }
69 #endif

```

Listing 22: Reduced concrete implementation of the execute_bof command in ciroStrike

In the following section, the Mythic-side definition of the command is implemented, which is responsible for providing the task details to the beacon.

6.3 Server-side Mythic Implementation

On the Mythic side, some adjustments need to be made, in order to make the operator dashboard aware of the command to execute a BOF. There are two relevant components when implementing the command: the translation container and the payload container. The latter contains various metadata and the command definitions for ciroStrike that are visible to the Mythic operator. The translation container contains the logic for processing the data sent from the operator to the beacon and vice versa. It defines the format expected by the beacon, which was already shown in Listing 17 in the previous section. To add a new command, the translation container must first be adjusted.

The adaptation is straight forward and only three things need to be modified. First, as already done on the beacon side, a new command type in the form of an enum value must be added (see Listing 23, lines 3-8). Then the serialization step (from Mythic to beacon) and the deserialization step (from beacon to Mythic) must be implemented. The latter step is not necessary in this case, since the data generated by the execute_bof command is only textual and the deserialization of strings is already implemented in the

translator. So only the serialization of the arguments passed by the operator remains to be implemented. The following arguments are to be processed:

- file (Mythic file UUID)
- args_hexlified (Wide String)
- function_name (Wide String)
- chunk_size (Unsigned Integer)

The translation container offers utility functions for serializing more complex data types. In this case `ser_uuid` and `ser_dynamically_sized_wstring` are used, which correspond to the `Mythic::UUID` and `Array<wchar_t>` C++ types within the beacon, respectively. The Python struct library is used for the unsigned integer data type used for the chunk size. The arguments are then packed into a binary blob (bytes type in Python) in the order that is expected by the beacon. All adjustments to the translator are described in Listing 23.

```

1 from . import serialization # functions for serializing common data types
2
3 class CommandType(Enum):
4     Exit = 0
5     Pwd = 1
6     Ls = 2
7     # ...
8     Execute_Bof = 19
9
10 # ...
11
12 @command("execute_bof", CommandType.Execute_Bof)
13 def execute_bof(args: dict) -> bytes:
14     return (
15         serialization.ser_uuid(args["file"])
16         + serialization.ser_dynamic_sized_wstring(args["args_hexlified"])
17         + serialization.ser_dynamic_sized_wstring(args["function_name"])
18         + struct.pack(">I", args.get("chunk_size", 2 << 13))
19     )

```

Listing 23: Translation container implementation of the execute_bof command

The command definition has yet to be added. The way commands are defined is prescribed by Mythic [71]: an argument class must be created that inherits from `TaskArguments`, containing all arguments and optional argument processing functions (Listing 24, lines 1-23). For each argument (`CommandParameter`), its name, data type and default value, among others, can be specified. The argument class can then be passed to a command definition class, which must also be a separate class that inherits from `CommandBase`. Additional metadata, such as an attack mapping or required permissions on the system, can also be specified here.

Listing 24 shows the simplified implementation of the `execute_bof` command and its arguments.

```

1 class ExecuteBOFArguments(TaskArguments):
2     def __init__(self, command_line, **kwargs):
3         super().__init__(command_line, **kwargs)
4         self.args = [
5             CommandParameter(
6                 name="file", type=ParameterType.File,
7                 parameter_group_info=[ParameterGroupInfo(required=True)]
8             ),
9             CommandParameter(
10                name="args_hexlified", type=ParameterType.String,
11                parameter_group_info=[ParameterGroupInfo(required=True)]
12            ),
13            CommandParameter(
14                name="function_name", type=ParameterType.String,
15                default_value="go",
16                parameter_group_info=[ParameterGroupInfo(required=False)]
17            ),
18            CommandParameter(
19                name="chunk_size", type=ParameterType.Number,
20                default_value=2 << 13, # 16 KB
21                parameter_group_info=[ParameterGroupInfo(required=False)]
22            ),
23        ]
24
25 class ExecuteBOFCommand(CommandBase):
26     cmd = "execute_bof"
27     author = "Leon Schmidt"
28     version = 1
29     argument_class = ExecuteBOFArguments # this is the argument class
30     attributes = CommandAttributes(
31         builtin=False,
32         supported_os=[SupportedOS.Windows],
33     )
34     # other configuration ...
35     # other functions ...

```

Listing 24: Mythic definition of the `execute_bof` command in ciroStrike

After the customized containers have been installed in Mythic, the operator can now access the parameters UI by simply entering `execute_bof` in the callback console (see Figure 9). This lists all possible parameters – including, most importantly, the file parameter “BOF to use”. The file transfer in Mythic works in such a way that the file is initially only uploaded to the Mythic server. The beacon then only receives a unique ID of this file (Mythic file UUID) when the command is invoked and must request the download of this file in a second step. The translation container converts this UUID and the entered

parameters into the format expected by the beacon. The beacon recognizes the command type and the arguments, requests the download of the BOF and executes it. The beacon output API, which is linked into the BOF, transmits the output of the BOF to Mythic. After complete execution, the success or error code of the execution is reported.

This section hereby completes the basic implementation of BOF support in ciroStrike.

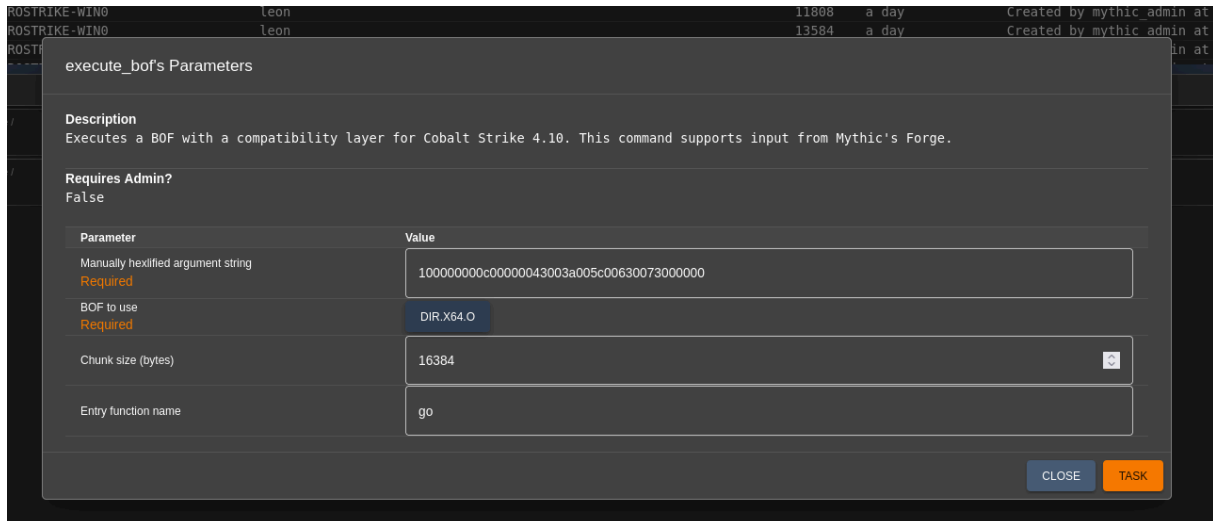


Figure 9: Mythic’s parameter UI for the `execute_bof` command

6.4 Adding Compatibility to use Command Augmentation with Forge

Now that ciroStrike and the Mythic infrastructure are able to handle BOFs, the remaining low-priority components of the Cobalt Strike approach will be examined in this chapter. This includes, in particular, Aggressor Script, which is not yet supported in ciroStrike or any other Mythic beacon and therefore prevents the use of certain BOFs.

On February 5, 2025, Mythic framework developer Cody Thomas (a.k.a. “its_a_feature_”) announced the release of a new Mythic plugin called “Forge” on the social networking service X. It is supposed to be “a way to standardize BOF/.NET execution within Mythic Agents” [72]. On closer inspection, however, it is by no means a standardization of BOF execution, but rather a tool that is supposed to fulfill two main tasks: on the one hand, the tool provides a user interface for the operator for executing BOFs and .NET executables. The actual execution of them is not implemented by Forge, but only “translated” to the invocation commands (i.e. `execute_bof`) of the Forge-supported beacons by means of an “adapter”. This means that beacon implementations must still provide their own BOF runtime – which must also be compatible with Forge’s calling convention. Mythic calls this principle “Command Augmentation”, which was added to Mythic in version 3.3 [73]. Forge is the first tool to take advantage of this feature. Currently, only the beacons “Apollo” and “Athena” are supported by Forge, but custom beacons can be added.

The second task of Forge is the management and provision of a library of BOFs and .NET assemblies. For this, Forge uses the already existing libraries “Sliver Armory” for BOFs [74], and “SharpCollection”

for .NET executables [75]. Thus, in the context of this thesis, only the Sliver Armory is of relevance here. A lot of preparatory work has been done there to adapt BOFs that rely on Aggressor Script, for example, in such a way that they can be used without it. An index of this Sliver Armory is maintained by Forge and provided to the operator in the form of beacon commands that only exist on the server side. Figure 10 illustrates this principle schematically, using the commands for the “whoami”, “regquery” and “nanodump” BOFs as examples.

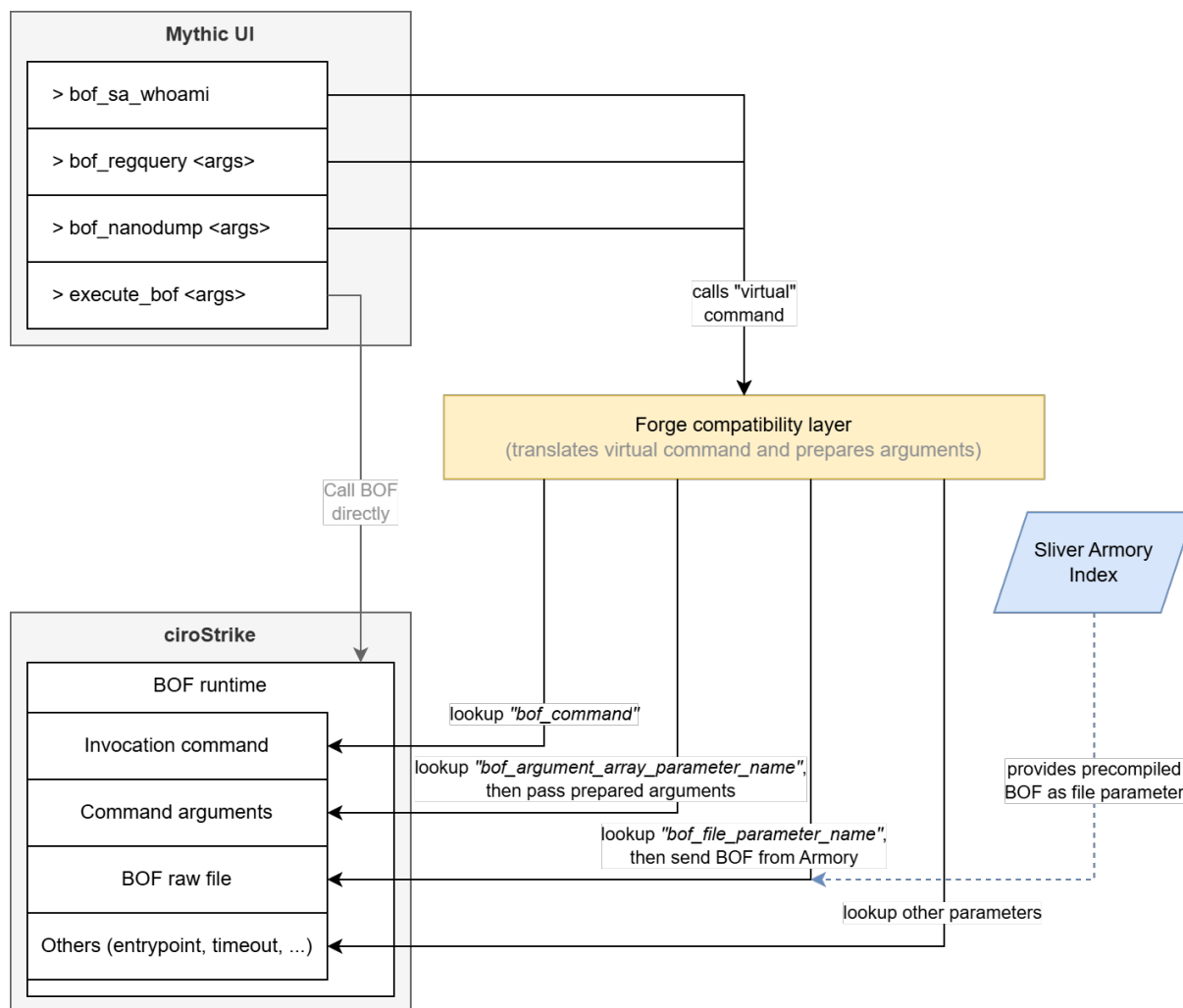


Figure 10: Compatibility layer workings within the “Forge” plugin (own illustration)

Forge greatly simplifies the use of BOFs for the operator by completely eliminating the following steps:

- Finding BOFs for the intended use case from external sources.
- Verifying the functionality of the found BOFs.
- Adjusting the BOFs to avoid the dependency on Aggressor Script (potentially very time-consuming or even impossible, as explained in detail in Chapter 4).
- Manual compilation of the BOFs.
- Manual encoding of parameters due to not being able to use the `&bof_pack` function from Aggressor Script.

The goal of this section is to make Forge available with ciroStrike in order to utilize these advantages for it as well. This will also effectively circumvent the requirement of an Aggressor Script parser in Mythic.

Requirements

To achieve compatibility with Forge, some requirements must first be met in ciroStrike. The compatibility layer in Forge consists of a single configuration file `payload_type_support.json`. It contains an array of payload types (Mythic language for a concrete beacon), in which the commands for BOF/.NET execution along with their argument names are specified. The entry for the already supported beacon Apollo in this configuration file is shown in Listing 25.

```

1 [
2   {
3     "agent": "apollo",
4     "bof_command": "execute_coff",
5     "bof_file_parameter_name": "bof_file",
6     "bof_argument_array_parameter_name": "coff_arguments",
7     "bof_entrypoint_parameter_name": "function_name",
8     // .NET values omitted
9   },
10  // other supported payload types
11 ]

```

Listing 25: Apollo support entry in `payload_type_support.json` in Forge [9]

Forge checks for each callback on the server side whether its payload type is contained in this configuration file. Should this be the case, the built-in Forge commands are made available for the callback. If a BOF is then executed through Forge, e.g., from the Sliver Armory, it searches for the following values in the array element corresponding to the payload type, as shown in Figure 10:

- `"bof_command"`: The main command to invoke the BOF runtime. In ciroStrike, this is `execute_bof`, which was implemented earlier in this chapter.
- `"bof_file_parameter_name"`: Name of the parameter for the main command that specifies the BOF file. When executing a Forge command, this is where the BOF from the Sliver Armory is passed to in the background.
- `"bof_argument_array_parameter_name"`: Name of the parameter for the main command where the parsed arguments are passed to.
- `"bof_entrypoint_parameter_name"`: Name of the parameter for the main command that specifies the name of the entry point function.

There are additional similar values within this configuration file, that specify the commands and parameters for .NET execution. They can be left blank to indicate, that the beacon is not able to execute .NET assemblies. Since this thesis is about BOF execution, these values are not considered here.

This means that the BOF invocation command must accept the parameters mentioned above. The `execute_bof` command in `ciroStrike` already accepts all these parameters, but it is not documented how the arguments are passed to the parameter specified in `"bof_argument_array_parameter_name"`. `ciroStrike` currently only accepts parameters as hex string, but the name of the configuration value suggests that an argument array is assumed to be used instead. The exact format must be determined.

Adjustments to the Mythic Command Definition

To find out precisely how the arguments are passed by Forge, the two supported Mythic beacons “Athena” and “Apollo” will be examined. Both beacons have similar definitions of this parameter [76], [77], as shown in Listing 26:

```

1 # ...
2 CommandParameter(
3     name="argument_array",
4     type=ParameterType.TypedArray,
5     choices=["int16", "int32", "string", "wchar", "base64"],
6     description="""Arguments to pass to the BOF in the following way:
7     -s:123 or int16:123
8     -i:123 or int32:123
9     -z:hello or string:hello
10    -Z:hello or wchar:hello
11    -b:SGVsbG9Xb3JsZA== or base64:SGVsbG9Xb3JsZA==""",
12    typedarray_parse_function=self.get_arguments,
13    default_value=[],
14    # ...
15 ),
16 # ...

```

Listing 26: Mythic argument definition for the argument array required by Forge

This argument uses Mythic’s typed arrays. For each entry in the array, a data type must be chosen from the `choices` array. This results in an `n`-sized array containing arrays of length 2 with the first entry containing the data type and the second entry the value. An exemplary typed array is shown in Listing 27. By using a `typedarray_parse_function`, a raw CLI input can be parsed into a typed array, in this case as explained in the description. Examples of the implementation of this function can be found in Athena and Apollo.

```

1 [
2     ["int16", 123],
3     ["int16", 456],
4     ["int32", 123456],
5     ["string", "Hello, World"]
6 ]

```

Listing 27: Example of a typed array in Mythic

In addition to the currently existing parameter `args_hexlified`, ciroStrike will consequently have an argument `args_array` added, which is of the type of a typed array. With the help of Mythic’s so-called “parameter groups”, the command parameter UI, as visible in Figure 9, can optionally be customized by adjusting the `parameter_group_info` array, so that the operator can decide which of the parameters they wish to use [71, section “parameter_group_info”]. The new parameter UI for the `execute_bof` command now looks like shown in Figure 11.

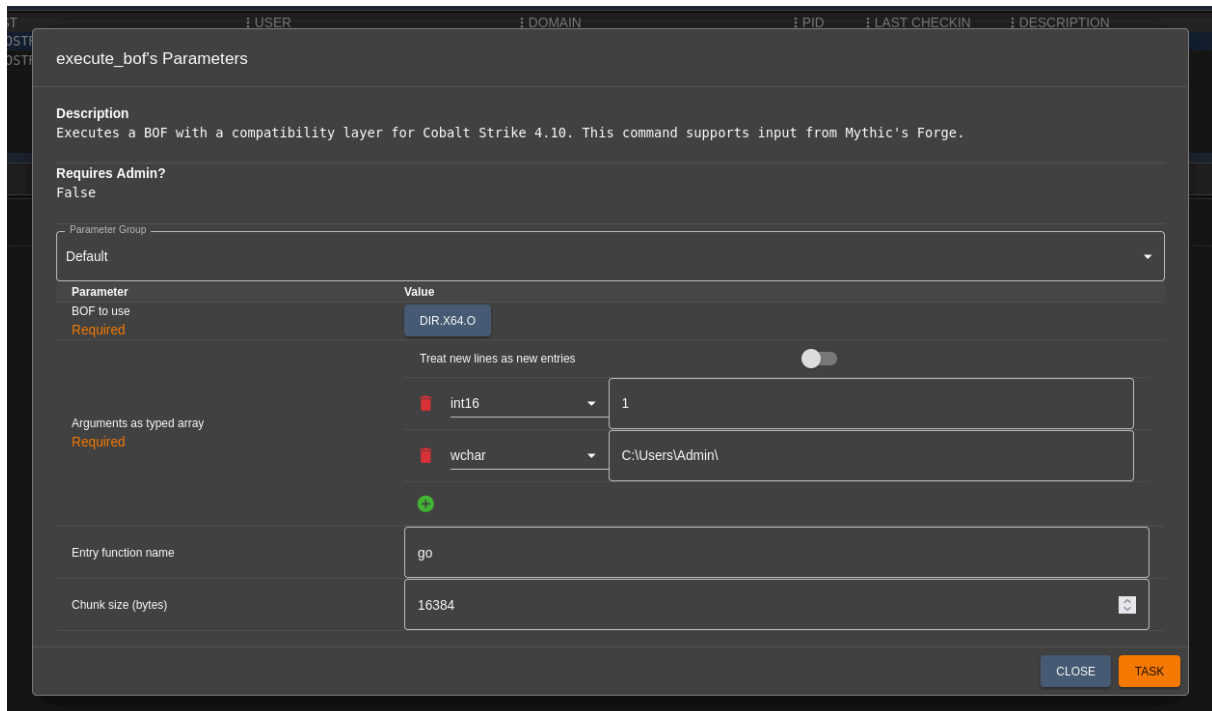


Figure 11: Mythic’s parameter UI for the `execute_bof` command with typed array parameter

From the outside, compatibility with Forge is thus established and the BOFs from the Sliver Armory can be invoked within ciroStrike. Yet, the translator still needs to be adjusted: the beacon and the BOF runtime still expect the arguments as a hex string, not as a typed array. The translator must therefore be able to serialize the typed array back into a hex string. To do this, TrustedSec’s script `beacon_generate.py` [13] is consulted again: it already implements this conversion. This serialization procedure is provided within the translator in the utility function `ser_typed_array`. The command definition must now be modified so that it generates an identical binary output for the beacon, regardless of which parameter is specified. It does this by checking, which of the parameters were actually set and by applying the correct serialization to it. The complete implementation of the command in the translator is shown in Listing 28.

```

1 @command("execute_bof", CommandType.Execute_Bof)
2 def execute_bof(args: dict) -> bytes:
3     if "args_hexlified" in args: # contains hexlified args --> pass raw
4         return (
5             serialization.ser_uuid(args["file"])
6             + serialization.ser_dynamic_sized_wstring(args["args_hexlified"])
7             + serialization.ser_dynamic_sized_wstring(args["function_name"])
8             + struct.pack(">I", args.get("chunk_size", 2 << 13))
9         )
10    else: # assume typed array args --> use ser_typed_array on "args_array"
11        return (
12            serialization.ser_uuid(args["file"])
13            + serialization.ser_typed_array(args.get("args_array", [])) # !
14            + serialization.ser_dynamic_sized_wstring(args["function_name"])
15            + struct.pack(">I", args.get("chunk_size", 2 << 13))
16        )

```

Listing 28: Translation container implementation of the `execute_bof` command, adjusted for Forge

As a result, this renders ciroStrike compatible with Forge and allows it to access and use all tailored BOFs from the entire Sliver Armory. The full BOF collection of Forge can now be queried with the command `forge_collections -collectionName SliverArmory`. For example, to use the comparatively complex BOF `nanodump`, it must first be registered in the current callback using the command `forge_register -collectionName SliverArmory -commandName "Nano Dump"`. After that, it can be used via `forge_bof_nanodump`, which displays the parameter UI specifically for this BOF, as shown in Figure 12.



Figure 12: Mythic's parameter UI for the `forge_bof_nanodump` command

Forge now serializes this command into the typed array and passes it to ciroStrike's BOF runtime, where it is successfully executed. Many of the BOFs collected in Chapter 4.1 are also available in the Sliver

Armory and, like some additional BOFs, could be successfully executed by using Forge. However, some of them were crashing due to wrong beacon API usage or Forge passing parameters erroneously. No errors were observed that were caused by the runtime. This shows that it is imperative to test all BOFs that are to be used in a safe environment prior to production usage when using Forge.

7 Conclusion

7.1 Summary

This thesis explored the integration of Cobalt Strike's Beacon Object File (BOF) system into the Mythic-based beacon "ciroStrike" developed by cirosec, with the goal of enhancing the beacon's extensibility while maintaining its compact, evasive nature. The research process began with a foundational analysis of Command & Control (C2) frameworks, emphasizing architectural differences and plugin systems of both Cobalt Strike and Mythic. The aim was to understand what makes BOF integration viable and how plugin systems can influence the operational flexibility of these red teaming tools.

A comprehensive technical investigation of the BOF format followed, including the role of the beacon APIs, the compilation into Common Object File Format (COFF) object files, and the principles of Dynamic Function Resolution (DFR). These insights were used to establish a firm understanding of the technical dependencies and requirements needed to make BOFs executable outside their native environment.

Building upon this understanding, a public sample collection of BOFs was put together, which informed the prioritization of beacon API group support and the prevalence of Aggressor Script dependencies. These findings were crucial for developing a generic BOF runtime implemented as a standalone C++ library. This runtime was built with strict constraints: no dependencies on the C standard library, no global variables, and complete heap allocation control – requirements necessary to support integration into evasive payloads like ciroStrike.

The runtime supports the core beacon APIs (Data Parser, Output, Format, Token and Utility) and implements full DFR support. Subsequently, the runtime was integrated into ciroStrike, with necessary adaptations to the beacon's execution context and transport mechanism. Compatibility with the Mythic server and the Forge plugin was also ensured, allowing the invocation of BOFs via Mythic's web interface and the usage of the comprehensive Sliver Armory BOF library.

At last, the integrated BOF runtime was tested against the representative subset of both the BOFs collected in Chapter 4.1 and those supplied by Forge. Most of the BOFs, which covered a wide range of use cases and API coverage, executed correctly, with some of them crashing due to wrong beacon API usage or Forge passing parameters erroneously. However, the results confirm the feasibility of integrating Cobalt Strike BOFs into a Mythic-based beacon and demonstrate that significant interoperability between proprietary and open-source red teaming frameworks can be achieved.

7.2 Outlook

The research presented in this thesis contributes a flexible and reusable solution to a recurring problem in red teaming operations: the gap between open-source C2 frameworks and mature proprietary plugin ecosystems. Several directions for future work emerge from this integration effort:

First, the coverage of beacon API groups can be expanded, in order to achieve compatibility with Cobalt Strike's `beacon.h` in version 4.10. Certain API groups such as `Spawn+Inject` or the `Data Store API` were omitted entirely due to low usage prevalence in current BOFs, as identified in Chapter 4. As these API groups gain traction, runtime support can be incrementally extended. The runtime code was designed to allow for simple extension.

Second, implementing an Aggressor Script interpreter for Mythic remains a long-term goal. Although this thesis excluded Aggressor Script support in favor of Forge, due to architectural incompatibilities and language differences (Java-based Cobalt Strike vs. Go/Python-based Mythic), many BOFs still rely on Aggressor Script for complex input handling or post-processing. Developing a compatibility layer, possibly using Mythic Scripting or Jupyter-based input serialization, would enable broader compatibility without modifying existing BOFs.

Third, from an OPSEC perspective, the runtime could benefit from additional features such as built-in call stack spoofing, memory unhooking, or in-memory encryption of BOF payloads. Keeping track of the BOF's heap usage is currently unsupported and could be a way to ensure a low memory footprint. Additionally, the beacon API function names could be obfuscated more – they may become apparent upon close examination of the payload. These features would further reduce detection surfaces when adopted.

Forth, there is a lot of leeway when it comes to ensuring the stability of the BOF runtime. It currently relies on the operator's input being correct and that the execution of BOFs has previously been tested in a safe environment. Errors in the BOF cannot currently be recovered, which will result in the loss of the beacon and, in the worst case, access to the attacked infrastructure. Execution handling had to be omitted due to the shellcode restrictions, but it would be possible to re-add this capability to ensure more stable operation of BOFs – as long as the environment allows it. Additionally, although Mythic has a OPSEC precheck feature to warn the operator of dangerous operations, it cannot be used because Forge does not support those prechecks and therefore cannot call the `execute_bof` command. This is an improvement that needs to be done by Forge.

In conclusion, this work successfully demonstrated how proprietary offensive capabilities can be decoupled from their original tooling and adapted to open-source frameworks. It lays the groundwork for future enhancements in the modularity, reusability, and stealth of C2 beacons and opens new possibilities for cross-framework collaboration and capability sharing.

Bibliography

- [1] J. V. Buddy Tancio, "Gootkit Loader's Updated Tactics and Fileless Delivery of Cobalt Strike." Accessed: Nov. 21, 2024. [Online]. Available: https://www.trendmicro.com/en_us/research/22/g/gootkit-loaders-updated-tactics-and-fileless-delivery-of-cobalt-strike.html
- [2] Mythic Documentation, "Mythic." Accessed: Nov. 29, 2024. [Online]. Available: <https://docs.mythic-c2.net/>
- [3] Texas Instruments, "Application Report SPRAAO8: Common Object File Format." Accessed: Feb. 27, 2025. [Online]. Available: <https://www.ti.com/lit/an/spraa08/spraa08.pdf>
- [4] Cobalt Strike, "Aggressor Script." Accessed: Nov. 18, 2024. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics_aggressor-scripts/as-resources_functions.htm
- [5] Cobalt Strike, "GitHub: Cobalt-Strike/Malleable-C2-Profiles." Accessed: Nov. 20, 2024. [Online]. Available: <https://github.com/Cobalt-Strike/Malleable-C2-Profiles>
- [6] Cobalt Strike, "GitHub: Cobalt-Strike/aggressor_script_examples - Random String example." Accessed: Jan. 14, 2025. [Online]. Available: https://github.com/Cobalt-Strike/aggressor_script_examples/blob/be052b4df74671f525db70e4ac6023e965ac9488/random_string.cna
- [7] Cobalt Strike, "Aggressor Script > Cobalt Strike." Accessed: Jan. 14, 2025. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics_aggressor-scripts/as_cobalt-strike.htm
- [8] Cobalt Strike, "How do I develop a BOF?." Accessed: Nov. 18, 2024. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics_beacon-object-files_how-to-develop.htm
- [9] C. Thomas, "Github: MythicAgents/forged." Accessed: Mar. 23, 2025. [Online]. Available: <https://github.com/MythicAgents/forged/tree/d0cb240dfc6dd403d300201751f101e5dc568600>
- [10] Cobalt Strike, "Cobalt Strike Pricing." Accessed: Oct. 28, 2024. [Online]. Available: <https://www.cobaltstrike.com/product/pricing-plans>
- [11] Cobalt Strike, "Cobalt Strike Arsenal (requires license key)." Accessed: Oct. 28, 2024. [Online]. Available: <https://download.cobaltstrike.com/scripts>
- [12] Ruben Vermeersch, "Mythic License." Accessed: Oct. 28, 2024. [Online]. Available: <https://github.com/its-a-feature/Mythic/blob/74536d4948ba4a1d577b3b12f94dd6390311cc6e/LICENSE>
- [13] TrustedSec, "GitHub: trustedsec/COFFLoader." Accessed: Nov. 11, 2024. [Online]. Available: <https://github.com/trustedsec/COFFLoader/tree/28125d4e6b033280a417cd21e249be8b155a65a1>

- [14] OtterHacker, “GitHub: OtterHacker/CoffLoader.” Accessed: Nov. 11, 2024. [Online]. Available: <https://github.com/OtterHacker/CoffLoader/tree/5f98ec0d0bf94c5867672107874d943208de1634>
- [15] Cracked5pider, “GitHub: Cracked5pider/CoffeeLdr.” Accessed: Nov. 11, 2024. [Online]. Available: <https://github.com/Cracked5pider/CoffeeLdr/tree/8628f278ac8073c7c30d1fc19be10864e26c6a74>
- [16] I. Korchagin, “GitHub: cloudflare/cloudflare-blog - 2021-03-obj-file/3/loader.c.” Accessed: Nov. 18, 2024. [Online]. Available: <https://github.com/cloudflare/cloudflare-blog/blob/1abdb62550b1ad121fb2c5fbb17b0c9a1fdaa527/2021-03-obj-file/3/loader.c>
- [17] Cobalt Strike, “Cobalt Strike Community Kit on github.io.” Accessed: Nov. 13, 2024. [Online]. Available: https://cobalt-strike.github.io/community_kit/
- [18] Lockheed Martin, “The Cyber Kill Chain.” Accessed: Dec. 03, 2024. [Online]. Available: <https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html>
- [19] Blumira, “Blumira Glossary: Reconnaissance.” Accessed: Oct. 29, 2024. [Online]. Available: <https://www.blumira.com/glossary/reconnaissance>
- [20] The MITRE Corporation, “MITRE ATT&CK: ICS tactics.” Accessed: Oct. 29, 2024. [Online]. Available: <https://attack.mitre.org/tactics/ics/>
- [21] The MITRE Corporation, “MITRE ATT&CK: TA0101 - Command and Control.” Accessed: Oct. 29, 2024. [Online]. Available: <https://attack.mitre.org/tactics/TA0101/>
- [22] Cobalt Strike, “SMB Beacon.” Accessed: Jan. 16, 2025. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/listener-infrastructure_beacon-smb.htm
- [23] Cobalt Strike, “Cobalt Strike Homepage.” Accessed: Nov. 13, 2024. [Online]. Available: <https://www.cobaltstrike.com/>
- [24] J. Munshaw, “Quarterly Report: Incident Response trends in Summer 2020,” 2020. Accessed: Nov. 13, 2024. [Online]. Available: <https://blog.talosintelligence.com/ctir-quarterly-trends-q4-2020/>
- [25] Cisco Talos Intelligence, “Year in Review 2022: General Threat Landscape,” 2022. Accessed: Nov. 13, 2024. [Online]. Available: <https://www.talosintelligence.com/resources/586>
- [26] Google Threat Intelligence, “Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor.” Accessed: Nov. 13, 2024. [Online]. Available: <https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor?hl=en>
- [27] Cobalt Strike, “Reports.” Accessed: Dec. 13, 2024. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/reports-logs_reports.htm

-
- [28] Cobalt Strike, “PE and Memory Indicators.” Accessed: Dec. 17, 2024. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2-extend_pe-memory-indicators.htm
- [29] boku7, “GitHub: boku7/BokuLoader.” Accessed: Dec. 17, 2024. [Online]. Available: <https://github.com/boku7/BokuLoader>
- [30] S2 Research Team and Team Cymru, “Mythic Case Study: Assessing Common Offensive Security Tools.” Accessed: Dec. 18, 2024. [Online]. Available: <https://www.team-cymru.com/post/mythic-case-study-assessing-common-offensive-security-tools>
- [31] Mythic Documentation, “9. OPSEC Checking.” Accessed: Dec. 18, 2024. [Online]. Available: <https://docs.mythic-c2.net/customizing/payload-type-development/opsec-checking>
- [32] Mythic, “Mythic Community Overview.” Accessed: Dec. 10, 2024. [Online]. Available: <https://mythicmeta.github.io/overview/>
- [33] Mythic Documentation, “What is Mythic scripting?.” Accessed: Jan. 15, 2025. [Online]. Available: <https://docs.mythic-c2.net/scripting>
- [34] Mythic Documentation, “Browser Scripting.” Accessed: Dec. 18, 2024. [Online]. Available: <https://docs.mythic-c2.net/customizing/payload-type-development/browser-scripting>
- [35] Mythic Documentation, “7. Peer-to-peer messages.” Accessed: Dec. 18, 2024. [Online]. Available: https://docs.mythic-c2.net/customizing/payload-type-development/create_tasking/agent-side-coding/delegates
- [36] Raphael Mudge, “Sleep 2.1 Manual Introduction.” Accessed: Jan. 15, 2025. [Online]. Available: <http://sleep.dashnine.org/manual/index.html>
- [37] Melvin Langvik and Its_a_feature_, “Youtube livestream recoding: Mythic Development Workflow with @its_a_feature_.” Accessed: Jan. 15, 2025. [Online]. Available: https://www.youtube.com/watch?v=LQ9-C_m2h6g
- [38] ars3n11, “GitHub: ars3n11/Aggressor-Scripts - ProcessTree.cna.” Accessed: Jan. 15, 2025. [Online]. Available: <https://github.com/ars3n11/Aggressor-Scripts/tree/461004184fcd4c4cb6fcd719a21eb918e52a2d0c>
- [39] its-a-feature, “GitHub: its-a-feature/Mythic - Example Jupyter Notebooks.” Accessed: Jan. 15, 2025. [Online]. Available: <https://github.com/its-a-feature/Mythic/blob/156bca469269688d2a5fe682909aca69361be5b6/jupyter-docker/jupyter>
- [40] Cracked5pider, “GitHub: HavocFramework/Havoc.” Accessed: Nov. 11, 2024. [Online]. Available: <https://github.com/HavocFramework/Havoc/tree/69ce17cb3686641e92a657cece5989feb9af64ed>

- [41] Airbus CERT, “Github: airbus-cert/Invoke-Bof.” Accessed: Nov. 20, 2024. [Online]. Available: <https://github.com/airbus-cert/Invoke-Bof>
- [42] BC Security, “Empire Wiki.” Accessed: Nov. 20, 2024. [Online]. Available: <https://bc-security.gitbook.io/empire-wiki>
- [43] NVISO Labs and Moritz Thomas, “Introducing CS2BR pt. I - How we enabled Brute Ratel Badgers to run Cobalt Strike BOFs.” Accessed: Jan. 20, 2025. [Online]. Available: <https://blog.nviso.eu/2023/05/15/introducing-cs2br-pt-i-how-we-enabled-brute-ratel-badgers-to-run-cobalt-strike-bofs/>
- [44] SilentWarble, “Github: MythicAgents/Hannibal.” Accessed: Jan. 15, 2025. [Online]. Available: <https://github.com/MythicAgents/Hannibal/tree/97d16fdd1f7bf20c5a45d0ecc1201048fbcede83d>
- [45] SilentWarble, “Making Mini Monsters.” Accessed: Jan. 15, 2025. [Online]. Available: <https://silentwarble.com/blog/making-mini-monsters/>
- [46] its-a-feature, “Github: MythicAgents/Hannibal.” Accessed: Jan. 15, 2025. [Online]. Available: <https://github.com/MythicAgents/apfell/tree/0a8b93afae08d5d7245047c6ae657961235d0387>
- [47] JXA-Cookbook, “Github: JXA-Cookbook/JXA-Cookbook.” Accessed: Jan. 15, 2025. [Online]. Available: <https://github.com/JXA-Cookbook/JXA-Cookbook/tree/e03c8dc6f19f0aed11a0496151450f81892f17d0>
- [48] Cobalt Strike, “Beacon Object Files.” Accessed: Nov. 18, 2024. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/corects/impact/current/userguide/content/topics/appx_bof.htm
- [49] Christopher Paschen, “A Developer's Introduction to Beacon Object Files.” Accessed: Mar. 01, 2025. [Online]. Available: <https://www.trustedsec.com/blog/a-developers-introduction-to-beacon-object-files>
- [50] Cobalt Strike, “Dynamic Function Resolution.” Accessed: Mar. 13, 2025. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/beacon-object-files_dynamic-func-resolution.htm
- [51] Cobalt Strike, “Beacon C API.” Accessed: Nov. 18, 2024. [Online]. Available: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/beacon-object-files_bof-c-api.htm
- [52] Cobalt Strike, “Cobalt Strike 4.10: Through the BeaconGate.” Accessed: Nov. 18, 2024. [Online]. Available: <https://www.cobaltstrike.com/blog/cobalt-strike-410-through-the-beacongate>
- [53] Trend Micro, “Portable executable (PE).” Accessed: Mar. 23, 2025. [Online]. Available: <https://www.trendmicro.com/vinfo/in/security/definition/portable-executable-pe>

-
- [54] Microsoft Documentation, “PE Format.” Accessed: Mar. 04, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/windows/win32/debug/pe-format>
- [55] trustedsec, “GitHub: trustedsec/CS-Situational-Awareness-BOF.” Accessed: Mar. 11, 2025. [Online]. Available: <https://github.com/trustedsec/CS-Situational-Awareness-BOF/tree/849c98a9b97c47de48409411a9f5db5dfaac3185>
- [56] I. Korchagin, “How to execute an object file: Part 1.” Accessed: Nov. 18, 2024. [Online]. Available: <https://blog.cloudflare.com/how-to-execute-an-object-file-part-1/>
- [57] I. Korchagin, “How to execute an object file: Part 2.” Accessed: Nov. 18, 2024. [Online]. Available: <https://blog.cloudflare.com/how-to-execute-an-object-file-part-2/>
- [58] I. Korchagin, “How to execute an object file: Part 3.” Accessed: Nov. 18, 2024. [Online]. Available: <https://blog.cloudflare.com/how-to-execute-an-object-file-part-3/>
- [59] Zero-Point Security, “BOF Dev and Tradecraft course.” Accessed: Dec. 17, 2024. [Online]. Available: <https://training.zeropointsecurity.co.uk/courses/bof-dev-and-tradecraft>
- [60] Microsoft Documentation, “Heap Functions.” Accessed: Mar. 18, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/windows/win32/memory/heap-functions>
- [61] Microsoft Documentation, “Windows Data Types.” Accessed: Mar. 17, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/windows/win32/winprog/windows-data-types>
- [62] Cobalt Strike, “Simplifying BOF Development: Debug, Test, and Save Your B(e)acon.” Accessed: Mar. 18, 2025. [Online]. Available: <https://www.cobaltstrike.com/blog/simplifying-bof-development>
- [63] Cobalt-Strike, “GitHub: Cobalt-Strike/bof_template.” Accessed: Mar. 11, 2025. [Online]. Available: https://github.com/Cobalt-Strike/bof_template/tree/a18692795588fbc4c2bc8ce9630b15f69971c5f5
- [64] J.-L. Gruber and F. Reiter, “Abusing Microsoft Warbird for Shellcode Execution.” Accessed: Mar. 30, 2025. [Online]. Available: <https://cirosec.de/en/news/abusing-microsoft-warbird-for-shellcode-execution/>
- [65] Microsoft Documentation, “HeapAlloc function (heapapi.h).” Accessed: Mar. 21, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapalloc>
- [66] Microsoft Documentation, “allocator.” Accessed: Mar. 21, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/cpp/cpp/allocator?view=msvc-170>
- [67] cppreference.com, “Lambda expressions (since C++11).” Accessed: Mar. 23, 2025. [Online]. Available: <https://en.cppreference.com/w/cpp/language/lambda>

- [68] P. Yosifovich, M. E. Russinovich, A. Ionescu, and D. A. Solomon, *Windows Internals*, 7th ed. 2017.
- [69] cppreference.com, “Standard library header <cstdlib>.” Accessed: Mar. 23, 2025. [Online]. Available: <https://en.cppreference.com/w/cpp/header/cstdlib>
- [70] Microsoft Documentation, “wvsprintfA-Funktion (winuser.h).” Accessed: Mar. 23, 2025. [Online]. Available: <https://learn.microsoft.com/de-de/windows/win32/api/winuser/nf-winuser-wvsprintfa>
- [71] Mythic Documentation, “Payload Type Development - Adding Commands.” Accessed: Mar. 21, 2025. [Online]. Available: <https://docs.mythic-c2.net/customizing/payload-type-development/adding-commands/commands>
- [72] C. Thomas, “X.com: Forging a Better Operator Quality of Life.” Accessed: Mar. 23, 2025. [Online]. Available: https://x.com/its_a_feature_/status/1887156681794085369
- [73] C. Thomas, “Forging a Better Operator Quality of Life.” Accessed: Mar. 23, 2025. [Online]. Available: <https://posts.specterops.io/forging-a-better-operator-quality-of-life-e8bf24fc2aba>
- [74] sliverarmory, “Github: sliverarmory/armory.” Accessed: Mar. 23, 2025. [Online]. Available: <https://github.com/sliverarmory/armory/tree/89d2a4bf8a7f83a667517af5c5afb27c5c17a43f>
- [75] Flangvik, “Github: Flangvik/SharpCollection.” Accessed: Mar. 23, 2025. [Online]. Available: <https://github.com/Flangvik/SharpCollection/tree/515931e71c153d79b2ead0803b6a989b5c97e89f>
- [76] checkymande, “Github: MythicAgents/Athena.” Accessed: Mar. 23, 2025. [Online]. Available: <https://github.com/MythicAgents/Athena/tree/4e822434191bb7abfae042b1f5486cadbb60c6e8>
- [77] its-a-feature, “Github: MythicAgents/Apollo.” Accessed: Mar. 23, 2025. [Online]. Available: <https://github.com/MythicAgents/Apollo/tree/6f57a86b7dfdafa3471c8f7bc3e49cb54853f10c>

Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance.

Leon Schmidt

Offenburg, 01.04.2025

A Git commit hashes for tools used in this thesis

All publicly accessible Git repositories used (GitHub, GitLab, etc.) are listed below together with the last commit hash used in this thesis. The purpose of this list is to compare assertions and assumptions made at the time the thesis was written with the state of the repositories at that time. Repositories that were only mentioned but not utilized in this thesis are not listed here.

List of C2 frameworks	
GitHub slug	Hash
its-a-feature/Mythic	86c6488a890370b94706b774ec6937aa82b83a55
HavocFramework/Havoc	69ce17cb3686641e92a657cece5989feb9af64ed
BishopFox/sliver	d32302a1ab22f42b1c6f144e74a7db3336af6a4c
EmpireProject/Empire	08cbd274bef78243d7a8ed6443b8364acd1fc48b
Ne0nd0g/merlin-agent	ce155028f7cdc07ede24b87751233db009de0eba

Table 11: Git commit hashes for publicly available C2 frameworks used in this thesis

List of Mythic beacons	
GitHub slug	Hash
MythicAgents/Hannibal	97d16fdd1f7bf20c5a45d0ecc1201048fbcd83d
MythicAgents/apfell	0a8b93afae08d5d7245047c6ae657961235d0387
MythicAgents/Apollo	6f57a86b7dfdaf3471c8f7bc3e49cb54853f10c
MythicAgents/Athena	4e822434191bb7abfae042b1f5486cadbb60c6e8

Table 12: Git commit hashes for publicly available Mythic beacons used in this thesis

List of BOFs	
GitHub slug	Hash
fortra/nanodump	450d5b23aeba5e0f8f6e5fc826a08997b2237be9
trustedsec/CS-Situational-Awareness-BOF	849c98a9b97c47de48409411a9f5db5dfaacc3185
trustedsec/CS-Remote-OPs-BOF	4ff580b1909cbbc440cb90aa859a358a6453aff6
antheuntohego/InlineExecute-Assembly	402637229d1c9abab221119601143732eb867e26
GhostPack/Koh	2e60ce274dc4f0bdce265174526f6d439f6a1414
mertdas/PrivKit	9ec2c41f1d00d4b8a74247280fc6a8ce7fbc79cb
CodeXTF2/ScreenshotBOF	58ea97afd210b44ed3e8e8e621942a486e28a741
wavvs/nanorobeus	5bf53021d43759a727a16522cdf24822f59ac4fc

List of BOFs	
GitHub slug	Hash
zyn3rgy/smbtakeover	d6d34201aa8e5e425e23ec7307c967a3efb4b793
CodeXTF2/WindowSpy	3514ef1d1d4c3b673d95addfec1f9fbd6432ae9e
rsmudge/unhook-bof	fa3c8d8a397719c5f2310334e6549bea541b209c
EncodeGroup/BOF-RegSave	07102aedb01c145f40232cac8c91ab05cd08dec8
boku7/whereami	066d8211483d049935adeccac0024c572ab953b0
connormcgarr/tgtdelegation	73a7ef77632cb9bd072ddcf2e2fd293fa1663cec
ASkyeye/Cobalt-Clip	1b6d051936dd178af8a14f5f6793079cd1010886

Table 13: Git commit hashes for publicly available BOFs used in this thesis

List of BOF/COFF runtimes	
GitHub slug	Hash
trustedsec/COFFLoader	28125d4e6b033280a417cd21e249be8b155a65a1
Cracked5pider/CoffeeLdr	8628f278ac8073c7c30d1fc19be10864e26c6a74
Yaxser/COFFLoader2	c9311f87fc178a34f1122ba45e9c215780db1f71
OtterHacker/CoffLoader	5f98ec0d0bf94c5867672107874d943208de1634
sliverarmory/COFFLoader	359529ed9615f769374e18b01b8a44b53769710c
cloudflare/cloudflare-blog	19868f07e397dd7f3e0416b8b2f0b670c31dabd6
airbus-cert/Invoke-Bof	c9f7f9f9e2985aff7d53296d8f289c572ae0d48f

Table 14: Git commit hashes for publicly available BOF/COFF runtimes used in this thesis

List of other repositories	
GitHub slug	Hash
MythicAgents/forge	d0cb240dfc6dd403d300201751f101e5dc568600
sliverarmory/armory	89d2a4bf8a7f83a667517af5c5afb27c5c17a43f
Flangvik/SharpCollection	515931e71c153d79b2ead0803b6a989b5c97e89f
Cobalt-Strike/Malleable-C2-Profiles	07fa1cf6024ae519c6c87a3be544077ee7692214
Cobalt-Strike/aggressor_script_examples	be052b4df74671f525db70e4ac6023e965ac9488
ars3n11/Aggressor-Scripts	461004184fcd4c4cb6fcd719a21eb918e52a2d0c
JXA-Cookbook/JXA-Cookbook	e03c8dc6f19f0aed11a0496151450f81892f17d0

Table 15: Git commit hashes for other repositories used or referenced in this thesis

B Cobalt Strike's beacon.h

```
1  /*
2  * Beacon Object Files (BOF)
3  * -----
4  * A Beacon Object File is a light-weight post exploitation tool that runs
5  * with Beacon's inline-execute command.
6  *
7  * Additional BOF resources are available here:
8  *   - https://github.com/Cobalt-Strike/bof\_template
9  *
10 * Cobalt Strike 4.x
11 * ChangeLog:
12 *   1/25/2022: updated for 4.5
13 *   7/18/2023: Added BeaconInformation API for 4.9
14 *   7/31/2023: Added Key/Value store APIs for 4.9
15 *                   BeaconAddValue, BeaconGetValue, and BeaconRemoveValue
16 *   8/31/2023: Added Data store APIs for 4.9
17 *                   BeaconDataStoreGetItem, BeaconDataStoreProtectItem,
18 *                   BeaconDataStoreUnprotectItem, and BeaconDataStoreMaxEntr
19 *   9/01/2023: Added BeaconGetCustomUserData API for 4.9
20 *   3/21/2024: Updated BeaconInformation API for 4.10 to return a BOOL
21 *                   Updated the BEACON_INFO data structure to add new parameter
22 *   4/19/2024: Added BeaconGetSyscallInformation API for 4.10
23 *   4/25/2024: Added APIs to call Beacon's system call implementation
24 */
25 #ifndef _BEACON_H_
26 #define _BEACON_H_
27 #include <windows.h>
28
29 #ifdef __cplusplus
30 extern "C" {
31 #endif // __cplusplus
32
33 /* data API */
34 typedef struct {
35     char * original; /* the original buffer [so we can free it] */
36     char * buffer;   /* current pointer into our buffer */
37     int   length;   /* remaining length of data */
38     int   size;     /* total size of this buffer */
39 } datap;
40
41 DECLSPEC_IMPORT void BeaconDataParse(datap * parser, char * buffer, int
size);
42 DECLSPEC_IMPORT char * BeaconDataPtr(datap * parser, int size);
```

```

43 DECLSPEC_IMPORT int BeaconDataInt(datap * parser);
44 DECLSPEC_IMPORT short BeaconDataShort(datap * parser);
45 DECLSPEC_IMPORT int BeaconDataLength(datap * parser);
46 DECLSPEC_IMPORT char * BeaconDataExtract(datap * parser, int * size);
47
48 /* format API */
49 typedef struct {
50     char * original; /* the original buffer [so we can free it] */
51     char * buffer; /* current pointer into our buffer */
52     int length; /* remaining length of data */
53     int size; /* total size of this buffer */
54 } formatp;
55
56 DECLSPEC_IMPORT void BeaconFormatAlloc(formatp * format, int maxsz);
57 DECLSPEC_IMPORT void BeaconFormatReset(formatp * format);
58 DECLSPEC_IMPORT void BeaconFormatAppend(formatp * format, const char *
text, int len);
59 DECLSPEC_IMPORT void BeaconFormatPrintf(formatp * format, const char
* fmt, ...);
60 DECLSPEC_IMPORT char * BeaconFormatToString(formatp * format, int * size);
61 DECLSPEC_IMPORT void BeaconFormatFree(formatp * format);
62 DECLSPEC_IMPORT void BeaconFormatInt(formatp * format, int value);
63
64 /* Output Functions */
65 #define CALLBACK_OUTPUT 0x0
66 #define CALLBACK_OUTPUT_OEM 0x1e
67 #define CALLBACK_OUTPUT_UTF8 0x20
68 #define CALLBACK_ERROR 0x0d
69 #define CALLBACK_CUSTOM 0x1000
70 #define CALLBACK_CUSTOM_LAST 0x13ff
71
72
73 DECLSPEC_IMPORT void BeaconOutput(int type, const char * data, int len);
74 DECLSPEC_IMPORT void BeaconPrintf(int type, const char * fmt, ...);
75
76
77 /* Token Functions */
78 DECLSPEC_IMPORT BOOL BeaconUseToken(HANDLE token);
79 DECLSPEC_IMPORT void BeaconRevertToken();
80 DECLSPEC_IMPORT BOOL BeaconIsAdmin();
81
82 /* Spawn+Inject Functions */
83 DECLSPEC_IMPORT void BeaconGetSpawnTo(BOOL x86, char * buffer, int length)
DECLSPEC_IMPORT void BeaconInjectProcess(HANDLE hProc, int pid, char *
84 payload, int p_len, int p_offset, char * arg, int a_len);

```

```

85 DECLSPEC_IMPORT void BeaconInjectTemporaryProcess(PROCESS_INFORMATION *
pInfo, char * payload, int p_len, int p_offset, char * arg, int a_len);
86 DECLSPEC_IMPORT BOOL BeaconSpawnTemporaryProcess(BOOL x86, BOOL
ignoreToken, STARTUPINFO * si, PROCESS_INFORMATION * pInfo);
87 DECLSPEC_IMPORT void BeaconCleanupProcess(PROCESS_INFORMATION * pInfo);
88
89 /* Utility Functions */
90 DECLSPEC_IMPORT BOOL toWideChar(char * src, wchar_t * dst, int max);
91
92 /* Beacon Information */
93 /*
94 * ptr - pointer to the base address of the allocated memory.
95 * size - the number of bytes allocated for the ptr.
96 */
97 typedef struct {
98     char * ptr;
99     size_t size;
100 } HEAP_RECORD;
101 #define MASK_SIZE 13
102
103 /* Information the user can set in the USER_DATA via a UDRL */
104 typedef enum {
105     PURPOSE_EMPTY,
106     PURPOSE_GENERIC_BUFFER,
107     PURPOSE_BEACON_MEMORY,
108     PURPOSE_SLEEPMASK_MEMORY,
109     PURPOSE_BOF_MEMORY,
110     PURPOSE_USER_DEFINED_MEMORY = 1000
111 } ALLOCATED_MEMORY_PURPOSE;
112
113 typedef enum {
114     LABEL_EMPTY,
115     LABEL_BUFFER,
116     LABEL_PEHEADER,
117     LABEL_TEXT,
118     LABEL_RDATA,
119     LABEL_DATA,
120     LABEL_PDATA,
121     LABEL_RELOC,
122     LABEL_USER_DEFINED = 1000
123 } ALLOCATED_MEMORY_LABEL;
124
125 typedef enum {
126     METHOD_UNKNOWN,
127     METHOD_VIRTUALALLOC,

```

```

128  METHOD_HEAPALLOC,
129  METHOD_MODULESTOMP,
130  METHOD_NTMAPVIEW,
131  METHOD_USER_DEFINED = 1000,
132 } ALLOCATED_MEMORY_ALLOCATION_METHOD;
133
134 /**
135 * This structure allows the user to provide additional information
136 * about the allocated heap for cleanup. It is mandatory to provide
137 * the HeapHandle but the DestroyHeap Boolean can be used to indicate
138 * whether the clean up code should destroy the heap or simply free the pages
139 * This is useful in situations where a loader allocates memory in the
140 * processes current heap.
141 */
142 typedef struct _HEAPALLOC_INFO {
143     PVOID HeapHandle;
144     BOOL DestroyHeap;
145 } HEAPALLOC_INFO, *PHEAPALLOC_INFO;
146
147 typedef struct _MODULESTOMP_INFO {
148     HMODULE ModuleHandle;
149 } MODULESTOMP_INFO, *PMODULESTOMP_INFO;
150
151 typedef union _ALLOCATED_MEMORY_ADDITIONAL_CLEANUP_INFORMATION {
152     HEAPALLOC_INFO HeapAllocInfo;
153     MODULESTOMP_INFO ModuleStompInfo;
154     PVOID Custom;
155 } ALLOCATED_MEMORY_ADDITIONAL_CLEANUP_INFORMATION,
    *PALLOCATED_MEMORY_ADDITIONAL_CLEANUP_INFORMATION;
156
157 typedef struct _ALLOCATED_MEMORY_CLEANUP_INFORMATION {
158     BOOL Cleanup;
159     ALLOCATED_MEMORY_ALLOCATION_METHOD AllocationMethod;
160     ALLOCATED_MEMORY_ADDITIONAL_CLEANUP_INFORMATION
    AdditionalCleanupInformation;
161 } ALLOCATED_MEMORY_CLEANUP_INFORMATION,
    *PALLOCATED_MEMORY_CLEANUP_INFORMATION;
162
163 typedef struct _ALLOCATED_MEMORY_SECTION {
164     ALLOCATED_MEMORY_LABEL Label; // A label to simplify Sleepmask development
165     PVOID BaseAddress; // Pointer to virtual address of section
166     SIZE_T VirtualSize; // Virtual size of the section
167     DWORD CurrentProtect; // Current memory protection of the section
168     DWORD PreviousProtect; // The previous memory protection of the
    section (prior to masking/unmasking)

```

```

169     BOOL    MaskSection;           // A boolean to indicate whether the section
    should be masked
170 } ALLOCATED_MEMORY_SECTION, *PALLOCATED_MEMORY_SECTION;
171
172 typedef struct _ALLOCATED_MEMORY_REGION {
173     ALLOCATED_MEMORY_PURPOSE Purpose;    // A label to indicate the purpose
    of the allocated memory
174     PVOID AllocationBase;             // The base address of the allocated
    memory block
175     SIZE_T RegionSize;                // The size of the allocated
    memory block
176     DWORD Type;                       // The type of memory allocated
177     ALLOCATED_MEMORY_SECTION Sections[8]; // An array of section information
    structures
178     ALLOCATED_MEMORY_CLEANUP_INFORMATION CleanupInformation; // Information
    required to cleanup the allocation
179 } ALLOCATED_MEMORY_REGION, *PALLOCATED_MEMORY_REGION;
180
181 typedef struct {
182     ALLOCATED_MEMORY_REGION AllocatedMemoryRegions[6];
183 } ALLOCATED_MEMORY, *PALLOCATED_MEMORY;
184
185 /*
186  * version                - The version of the beacon dll was added for
    release 4.10
187  *                        version format: 0xMMmmPP, where MM = Major, mm =
    Minor, and PP = Patch
188  *                        e.g. 0x040900 -> CS 4.9
189  *                        0x041000 -> CS 4.10
190  *
191  * sleep_mask_ptr         - pointer to the sleep mask base address
192  * sleep_mask_text_size  - the sleep mask text section size
193  * sleep_mask_total_size - the sleep mask total memory size
194  *
195  * beacon_ptr            - pointer to beacon's base address
196  *                        The stage.obfuscate flag affects this value when using CS
    default loader.
197  *                        true: beacon_ptr = allocated_buffer - 0x1000 (Not a
    valid address)
198  *                        false: beacon_ptr = allocated_buffer (A valid address)
199  *                        For a UDRL the beacon_ptr will be set to the 1st argument
    to DllMain
200  *                        when the 2nd argument is set to DLL_PROCESS_ATTACH.
201  * heap_records          - list of memory addresses on the heap beacon wants to mask
202  *                        The list is terminated by the HEAP_RECORD.ptr set to NULL

```

```

203 * mask          - the mask that beacon randomly generated to apply
204 *
205 * Added in version 4.10
206 * allocatedMemory - An ALLOCATED_MEMORY structure that can be set in
the USER_DATA
207 *                  via a UDRL.
208 */
209 typedef struct {
210     unsigned int version;
211     char * sleep_mask_ptr;
212     DWORD  sleep_mask_text_size;
213     DWORD  sleep_mask_total_size;
214
215     char * beacon_ptr;
216     HEAP_RECORD * heap_records;
217     char  mask[MASK_SIZE];
218
219     ALLOCATED_MEMORY allocatedMemory;
220 } BEACON_INFO, *PBEACON_INFO;
221
222 DECLSPEC_IMPORT BOOL BeaconInformation(PBEACON_INFO info);
223
224 /* Key/Value store functions
225 * These functions are used to associate a key to a memory address and sa
226 * that information into beacon. These memory addresses can then be
227 * retrieved in a subsequent execution of a BOF.
228 *
229 * key - the key will be converted to a hash which is used to locate the
230 *      memory address.
231 *
232 * ptr - a memory address to save.
233 *
234 * Considerations:
235 * - The contents at the memory address is not masked by beacon.
236 * - The contents at the memory address is not released by beacon.
237 *
238 */
239 DECLSPEC_IMPORT BOOL BeaconAddValue(const char * key, void * ptr);
240 DECLSPEC_IMPORT void * BeaconGetValue(const char * key);
241 DECLSPEC_IMPORT BOOL BeaconRemoveValue(const char * key);
242
243 /* Beacon Data Store functions
244 * These functions are used to access items in Beacon's Data Store.
245 * BeaconDataStoreGetItem returns NULL if the index does not exist.
246 *

```

```

247 *   The contents are masked by default, and BOFs must unprotect the entry
248 *   before accessing the data buffer. BOFs must also protect the entry
249 *   after the data is not used anymore.
250 *
251 */
252
253 #define DATA_STORE_TYPE_EMPTY 0
254 #define DATA_STORE_TYPE_GENERAL_FILE 1
255
256 typedef struct {
257     int type;
258     DWORD64 hash;
259     BOOL masked;
260     char* buffer;
261     size_t length;
262 } DATA_STORE_OBJECT, *PDATA_STORE_OBJECT;
263
264 DECLSPEC_IMPORT PDATA_STORE_OBJECT BeaconDataStoreGetItem(size_t index);
265 DECLSPEC_IMPORT void BeaconDataStoreProtectItem(size_t index);
266 DECLSPEC_IMPORT void BeaconDataStoreUnprotectItem(size_t index);
267 DECLSPEC_IMPORT size_t BeaconDataStoreMaxEntries();
268
269 /* Beacon User Data functions */
270 DECLSPEC_IMPORT char * BeaconGetCustomUserData();
271
272 /* Beacon System call */
273 /* Syscalls API */
274 typedef struct
275 {
276     PVOID fnAddr;
277     PVOID jmpAddr;
278     DWORD sysnum;
279 } SYSCALL_API_ENTRY, *PSYSCALL_API_ENTRY;
280
281 typedef struct
282 {
283     SYSCALL_API_ENTRY ntAllocateVirtualMemory;
284     SYSCALL_API_ENTRY ntProtectVirtualMemory;
285     SYSCALL_API_ENTRY ntFreeVirtualMemory;
286     SYSCALL_API_ENTRY ntGetContextThread;
287     SYSCALL_API_ENTRY ntSetContextThread;
288     SYSCALL_API_ENTRY ntResumeThread;
289     SYSCALL_API_ENTRY ntCreateThreadEx;
290     SYSCALL_API_ENTRY ntOpenProcess;
291     SYSCALL_API_ENTRY ntOpenThread;

```

```

292  SYSCALL_API_ENTRY ntClose;
293  SYSCALL_API_ENTRY ntCreateSection;
294  SYSCALL_API_ENTRY ntMapViewOfSection;
295  SYSCALL_API_ENTRY ntUnmapViewOfSection;
296  SYSCALL_API_ENTRY ntQueryVirtualMemory;
297  SYSCALL_API_ENTRY ntDuplicateObject;
298  SYSCALL_API_ENTRY ntReadVirtualMemory;
299  SYSCALL_API_ENTRY ntWriteVirtualMemory;
300  SYSCALL_API_ENTRY ntReadFile;
301  SYSCALL_API_ENTRY ntWriteFile;
302  SYSCALL_API_ENTRY ntCreateFile;
303  } SYSCALL_API, *PSYSCALL_API;
304
305  /* Additional Run Time Library (RTL) addresses used to support system calls.
306  * If they are not set then system calls that require them will fall back
307  * to the Standard Windows API.
308  *
309  * Required to support the following system calls:
310  *     ntCreateFile
311  */
312  typedef struct
313  {
314      PVOID rtlDosPathNameToNtPathNameUWithStatusAddr;
315      PVOID rtlFreeHeapAddr;
316      PVOID rtlGetProcessHeapAddr;
317  } RTL_API, *PRTL_API;
318
319  typedef struct
320  {
321      PSYSCALL_API syscalls;
322      PRTL_API      rtls;
323  } BEACON_SYSCALLS, *PBEACON_SYSCALLS;
324
325  DECLSPEC_IMPORT BOOL BeaconGetSyscallInformation(PBEACON_SYSCALLS info, BOOL
  resolveIfNotInitialized);
326
327  /* Beacon System call functions which will use the current system call
  method */
328  DECLSPEC_IMPORT LPVOID BeaconVirtualAlloc(LPVOID lpAddress, SIZE_T dwSize,
  DWORD flAllocationType, DWORD flProtect);
329  DECLSPEC_IMPORT LPVOID BeaconVirtualAllocEx(HANDLE processHandle, LPVOID
  lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);
330  DECLSPEC_IMPORT BOOL BeaconVirtualProtect(LPVOID lpAddress, SIZE_T dwSize,
  DWORD flNewProtect, PDWORD lpflOldProtect);

```

```

331 DECLSPEC_IMPORT BOOL BeaconVirtualProtectEx(HANDLE processHandle, LPVOID
lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect);
332 DECLSPEC_IMPORT BOOL BeaconVirtualFree(LPVOID lpAddress, SIZE_T dwSize, DWORD
dwFreeType);
333 DECLSPEC_IMPORT BOOL BeaconGetThreadContext(HANDLE threadHandle, PCONTEXT
threadContext);
334 DECLSPEC_IMPORT BOOL BeaconSetThreadContext(HANDLE threadHandle, PCONTEXT
threadContext);
335 DECLSPEC_IMPORT DWORD BeaconResumeThread(HANDLE threadHandle);
336 DECLSPEC_IMPORT HANDLE BeaconOpenProcess(DWORD desiredAccess, BOOL
inheritHandle, DWORD processId);
337 DECLSPEC_IMPORT HANDLE BeaconOpenThread(DWORD desiredAccess, BOOL
inheritHandle, DWORD threadId);
338 DECLSPEC_IMPORT BOOL BeaconCloseHandle(HANDLE object);
339 DECLSPEC_IMPORT BOOL BeaconUnmapViewOfFile(LPCVOID baseAddress);
340 DECLSPEC_IMPORT SIZE_T BeaconVirtualQuery(LPCVOID address,
PMEMORY_BASIC_INFORMATION buffer, SIZE_T length);
341 DECLSPEC_IMPORT BOOL BeaconDuplicateHandle(HANDLE hSourceProcessHandle,
HANDLE hSourceHandle, HANDLE hTargetProcessHandle, LPHANDLE lpTargetHandle,
DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwOptions);
342 DECLSPEC_IMPORT BOOL BeaconReadProcessMemory(HANDLE hProcess, LPCVOID
lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize, SIZE_T *lpNumberOfBytesRead);
343 DECLSPEC_IMPORT BOOL BeaconWriteProcessMemory(HANDLE hProcess, LPVOID
lpBaseAddress, LPCVOID lpBuffer, SIZE_T nSize, SIZE_T
*lpNumberOfBytesWritten);
344
345 /* Beacon Gate APIs */
346 DECLSPEC_IMPORT VOID BeaconDisableBeaconGate();
347 DECLSPEC_IMPORT VOID BeaconEnableBeaconGate();
348
349 /* Beacon User Data
350 *
351 * version format: 0xMMmmPP, where MM = Major, mm = Minor, and PP = Patch
352 * e.g. 0x040900 -> CS 4.9
353 *      0x041000 -> CS 4.10
354 */
355
356 #define DLL_BEACON_USER_DATA 0x0d
357 #define BEACON_USER_DATA_CUSTOM_SIZE 32
358 typedef struct
359 {
360     unsigned int version;
361     PSYSCALL_API syscalls;
362     char custom[BEACON_USER_DATA_CUSTOM_SIZE];
363     PRTL_API rtls;

```

```
364     PALLOCATED_MEMORY allocatedMemory;
365 } USER_DATA, * PUSER_DATA;
366
367 #ifdef __cplusplus
368 }
369 #endif // __cplusplus
370 #endif // _BEACON_H_
```

C Code for Checking Beacon API Usage in GitHub Repository

The following Python tool was used in Chapter 4 to analyze which beacon APIs are used per BOF GitHub repository. It uses the GitHub Search API and requires a valid Personal Access Token (PAT). The tool's creation was aided by ChatGPT using the GPT-4o model.

```
1 import requests
2 import time
3
4 GITHUB_TOKEN = "<redacted>"
5 TARGET_REPO = "zyn3rgy/smbtakeover" # example
6
7 HEADERS = {
8     "Authorization": f"token {GITHUB_TOKEN}",
9     "Accept": "application/vnd.github.v3+json"
10 }
11
12 FUNCTIONS = [
13     # Data API
14     "BeaconDataParse",
15     "BeaconDataPtr",
16     "BeaconDataInt",
17     "BeaconDataShort",
18     "BeaconDataLength",
19     "BeaconDataExtract",
20
21     # Format API
22     "BeaconFormatAlloc",
23     "BeaconFormatReset",
24     "BeaconFormatAppend",
25     "BeaconFormatPrintf",
26     "BeaconFormatToString",
27     "BeaconFormatFree",
28     "BeaconFormatInt",
29
30     # Output Functions
31     "BeaconOutput",
32     "BeaconPrintf",
33
34     # Token Functions
35     "BeaconUseToken",
36     "BeaconRevertToken",
37     "BeaconIsAdmin",
38
```

```

39 # Spawn+Inject Functions
40 "BeaconGetSpawnTo",
41 "BeaconInjectProcess",
42 "BeaconInjectTemporaryProcess",
43 "BeaconSpawnTemporaryProcess",
44 "BeaconCleanupProcess",
45
46 # Utility Functions
47 "toWideChar",
48
49 # Beacon Information
50 "BeaconInformation",
51
52 # Key/Value Store Functions
53 "BeaconAddValue",
54 "BeaconGetValue",
55 "BeaconRemoveValue",
56
57 # Beacon Data Store Functions
58 "BeaconDataStoreGetItem",
59 "BeaconDataStoreProtectItem",
60 "BeaconDataStoreUnprotectItem",
61 "BeaconDataStoreMaxEntries",
62
63 # Beacon User Data Functions
64 "BeaconGetCustomUserData",
65
66 # Beacon System Call Functions
67 "BeaconGetSyscallInformation",
68 "BeaconVirtualAlloc",
69 "BeaconVirtualAllocEx",
70 "BeaconVirtualProtect",
71 "BeaconVirtualProtectEx",
72 "BeaconVirtualFree",
73 "BeaconGetThreadContext",
74 "BeaconSetThreadContext",
75 "BeaconResumeThread",
76 "BeaconOpenProcess",
77 "BeaconOpenThread",
78 "BeaconCloseHandle",
79 "BeaconUnmapViewOfFile",
80 "BeaconVirtualQuery",
81 "BeaconDuplicateHandle",
82 "BeaconReadProcessMemory",
83 "BeaconWriteProcessMemory",

```

```

84
85 # Beacon Gate APIs
86 "BeaconDisableBeaconGate",
87 "BeaconEnableBeaconGate",
88 ]
89
90 def search_repo_function_usage(function_name, repo):
91     url = f"https://api.github.com/search/code?q={function_name}+repo:{repo}"
92
93     while True:
94         response = requests.get(url, headers=HEADERS)
95
96         if response.status_code == 200:
97             results = response.json().get('items', [])
98             return results
99
100        elif response.status_code == 403:
101            reset_time = int(response.headers.get(
102                "X-RateLimit-Reset", time.time() + 60))
103            wait_time = reset_time - int(time.time())
104            print(f"RateLimited, waiting {wait_time} seconds...")
105            time.sleep(wait_time + 1)
106
107        else:
108            print(f"Error at {function_name}: {response.status_code}")
109            return []
110
111 if __name__ == "__main__":
112     for function in FUNCTIONS:
113         results = search_repo_function_usage(function, TARGET_REPO)
114
115         if results:
116             print(f"{len(results)} results for {function}:")
117             for result in results[:5]: # Show max 5 results
118                 repo_name = result['repository']['full_name']
119                 file_path = result['path']
120                 html_url = result['html_url']
121                 print(f" - File: {file_path}")
122                 print(f"   URL: {html_url}")
123         else:
124             print(f"no results for {function}")
125
126     print()

```

D Code for converting a file to a C/C++ char buffer

The following code was used to statically embed BOF files into C/C++ code. In this way, the developed BOF runtime could be tested more easily, as loading the BOF file was impossible due to the restriction that the C standard library was unavailable. This code loads an arbitrary file and generates a char array (“buffer”) from it, which can be statically compiled into the test program. As a result, a previous transfer by Mythic via the C2 profile is simulated without such an infrastructure having to exist.

```
1 import sys
2
3 def binary_to_c_array(file_path, array_name="binary_data"):
4     try:
5         with open(file_path, "rb") as binary_file:
6             binary_data = binary_file.read()
7
8         array_content = ", ".join(f"0x{byte:02x}" for byte in binary_data)
9         array_size = len(binary_data)
10
11        c_array = f"unsigned char {array_name}[] = {{\n    {array_content}\n}};\n"
12        c_array += f"unsigned int {array_name}_len = {array_size};\n"
13
14        output_file = file_path + ".c"
15        with open(output_file, "w") as c_file:
16            c_file.write(c_array)
17
18        print(f"Converted {file_path} to {output_file}")
19    except Exception as e:
20        print(f"Error: {e}")
21
22    if __name__ == "__main__":
23        if len(sys.argv) != 2:
24            print("Usage: python convert_to_c_array.py <binary_file>")
25        else:
26            binary_to_c_array(sys.argv[1])
```

An exemplary output looks like this:

```
1 unsigned char binary_data[] = {
2     0x69, 0x6d, 0x70, 0x6f, 0x72 // ...
3 };
4 unsigned int binary_data_len = 851;
```

E Code for the discontinued Use-it-all BOF

This is the source code for the “Use-it-all” BOF, which was intended to test all of Cobalt Strike’s beacon APIs. Development was discontinued during the course of this thesis, because the implementation proved too complex to serve as a simple example.

For documentation purposes and potential further development, this is the latest version of the Use-it-all BOF. In this version, the Token API and Data Store API are only partially implemented, while the Key/Value Store API implementation is not functional. Parts of the examples were taken from Cobalt Strike’s BOF template repository [63].

```
1 #include "beacon.h"
2
3 // Test Dynamic Function Resolution (DFR)
4 WINBASEAPI int WINAPI KERNEL32$strlenA(LPCSTR lpString);
5
6 /* function to take the src and modify it by some offset and
7 * save it into the dest buffer.
8 *
9 * returns number of characters modified.
10 */
11 int somefunc(char *src, char *dest, int destLen, int offset) {
12     int i;
13     for(i = 0; i < destLen && src[i] != 0x0; i++) {
14         dest[i] = src[i] + offset;
15     }
16     return i;
17 }
18
19 void go(char *args, int alen) {
20     //// Beacon API Tests ////
21
22     // Format and Output API
23     formatp buffer;
24     BeaconFormatAlloc(&buffer, 1024);
25     // fill the buffer with stuff
26     for (int i = 0; i < 5; i++) {
27         BeaconFormatPrintf(&buffer, "counter is currently at %d\n", i);
28     }
29     BeaconPrintf(CALLBACK_OUTPUT, "%s\n", BeaconFormatToString(&buffer, NULL))
30     BeaconFormatFree(&buffer);
31
32     // Data Parser API
33     datap parser;
```

```

34 BeaconDataParse(&parser, args, alen);
35 char *str_arg = BeaconDataExtract(&parser, NULL);
36 int num_arg = BeaconDataInt(&parser);
37 BeaconPrintf(CALLBACK_OUTPUT, "First arg (string): %s, second arg (int):
%d\n", str_arg, num_arg);
38
39 // Token API
40 char *is_admin = BeaconIsAdmin() ? "yes" : "no";
41 BeaconPrintf(CALLBACK_OUTPUT, "Beacon has admin token?: %s\n", is_admin);
42
43 // Utility functions
44 char *util_src = "Hello, World\n";
45 wchar_t util_dst[20];
46 if (!toWideChar(util_src, util_dst, sizeof(util_dst))) {
47     BeaconPrintf(CALLBACK_ERROR, "couldn't convert to wide string\n");
48 }
49
50 // Key/Value Store API
51 // This does not work currently, maybe use structs as in https://github.com/
Cobalt-Strike/bof_template/blob/main/examples/cs_key_value/cs_key_value.c
52 BeaconPrintf(CALLBACK_OUTPUT, "searching for key \"test_key\"\n");
53 int *kv_int = (int*)BeaconGetValue("test_key");
54 if (kv_int == NULL) {
55     BeaconPrintf(CALLBACK_OUTPUT, "key not found, adding a new one with 1\n"
56     int kv_int_new = 1;
57     BeaconAddValue("test_key", &kv_int_new);
58 } else {
59     BeaconPrintf(CALLBACK_OUTPUT, "key found, value is %d, increasing by
1\n", *kv_int);
60     BeaconRemoveValue("test_key");
61     int kv_int_new = *kv_int + 1;
62     BeaconAddValue("test_key", &kv_int_new);
63 }
64
65 // Data Store API
66 size_t ds_max_entries = BeaconDataStoreMaxEntries();
67 BeaconPrintf(CALLBACK_OUTPUT, "beacon data store can hold %d entries\n",
ds_max_entries);
68 //PDATA_STORE_OBJECT obj = BeaconDataStoreGetItem(0);
69 //BeaconDataStoreProtectItem(0);
70 //BeaconDataStoreUnprotectItem(0);
71
72 // Syscall API
73 BEACON_SYSCALLS syscalls;
74 BOOL status = BeaconGetSyscallInformation(&syscalls, TRUE);

```

```

75  PBEACON_SYSCALLS p_syscalls = (PBEACON_SYSCALLS)&syscalls;
76  PSYSCALL_API_ENTRY entry = &p_syscalls->syscalls->ntAllocateVirtualMemory;
77  BeaconPrintf(CALLBACK_OUTPUT,
78     "ntAllocateVirtualMemory\n\tfnAddr: %p\n\tjmpAddr: %p\n\tsysnum: %lu\n",
79     entry->fnAddr,
80     entry->jmpAddr,
81     entry->sysnum
82  );
83
84  // Beacon Information
85  BEACON_INFO info = { 0 };
86  info.version = 0x041000; // Cobalt Strike 4.10
87  if (!BeaconInformation((PBEACON_INFO)&info)) {
88     BeaconPrintf(CALLBACK_ERROR, "couldn't get beacon information\n");
89  }
90  // print generic info
91  BeaconPrintf(CALLBACK_OUTPUT,
92     "Beacon info:\n\tBase address: %p\n\tText size: %lu\n\tTotal size: %lu\n",
93     info.sleep_mask_ptr
94  );
95  // print memory info
96  PALLOCATED_MEMORY_SECTION mem_info = (PALLOCATED_MEMORY_SECTION)&info;
97  BeaconPrintf(CALLBACK_OUTPUT,
98     "Memory info:\n\tBaseAddress: %p\n\tVirtualSize: 0x%x\n",
99     mem_info->BaseAddress,
100    mem_info->VirtualSize, mem_info->VirtualSize,
101    mem_info->CurrentProtect
102  );
103
104  //// Generic C Tests ////
105
106  // test if we can call a local function
107  char stuff[20] = { 'Z', 'Z', 'Z', 'Z', 'Z', 'Y', 'Y', 'Y', 'Y', 'Y', 'X',
108  'X', 'X', 'X', 'X', 'W', 'W', 'W', 'W', 0x0 };
109  BeaconPrintf(CALLBACK_OUTPUT, "Orig string:%s\n", stuff);
110  int stuff_modified_count = somefunc(stuff, stuff, sizeof(stuff), 1);
111  BeaconPrintf(CALLBACK_OUTPUT, "New string: %s, Modified: %d\n", stuff,
112  stuff_modified_count);
113
114  // test if we can call a DFR'ed kernel32 function
115  int stuff_len = KERNEL32$lstrlenA(stuff);
116  BeaconPrintf(CALLBACK_OUTPUT, "Len of new string: %d\n", stuff_len);
117  }

```