C2 Chopper

Yassine Belkhadem, Mohamed Mongi Saidane
Department of Computer Science and Mathematics Engineering
INSAT, University of Carthage, Tunisia
yassine.belkhadem@insat.ucar.tn, mohamedmongi.saidane@insat.ucar.tn

Abstract

This paper proposes a post-exploitation framework that can help testers manage their sessions and tools efficiently, saving them time and effort. The framework provides a centralized platform for tracking progress and all the hosts. The framework also includes a set of pre-configured tools and techniques for each environment, eliminating the need for testers to prepare for each engagement separately. The proposed framework can significantly improve the efficiency and effectiveness of security engagements and training labs, making them less challenging and more productive.
https://github.com/Chopper-C2-Framework/c2-chopper

Keywords: Security, Post exploitation, Penetration testing, Command and control, Security assessment, Golang, HTTP, GRPC

1 Introduction

1.1 General Introduction

Security researchers, penetration testers, and security enthusiasts face many difficulties in security engagements and training labs. One such difficulty is the enormous size of the client's network, which frequently necessitates the tester to move laterally and horizontally across numerous forests and domains while configuring a number of proxies. It can be difficult to keep track of all the hosts and the progress, and if one proxy chain fails, all sessions might be lost.

The enumeration and post-exploitation phases after gaining initial access are also very important, and most testers follow a checklist for each engagement. However, given the time constraints of engagements, preparing for each environment can be time-consuming because each one requires a particular set of tools and techniques. In this situation, a post-exploitation framework is crucial because it can save testers a lot of time by letting them manage all of their sessions with ease and giving them the tools they need, ready for use.

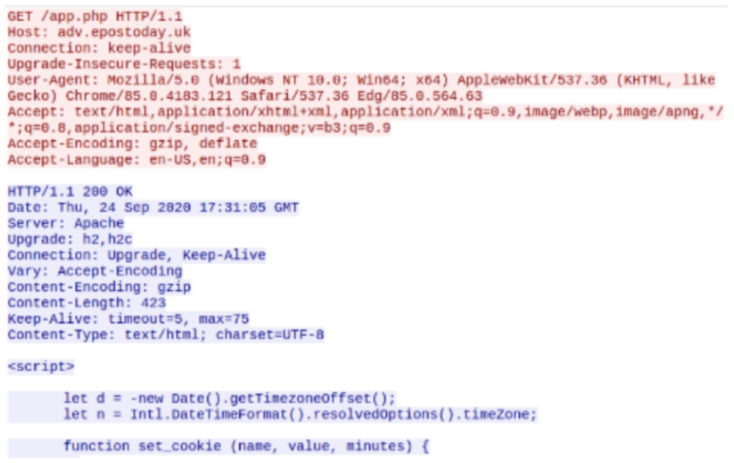
This section's main goal is to familiarize readers with the idea of Command and Control (C2), its elements, and other important ideas. In the sections that follow, these elements will be covered in more detail, giving readers a thorough understanding of the topic. Testers can better understand the value of a post-exploitation framework and how it can streamline their engagements and training labs by becoming more familiar with C2.

1.2 What is a Command and Control (C2)

Command and control (C2) systems are employed to control remote sessions coming from compromised hosts. A command and control program interface allows a security tester to send commands directly from the program or access a remote shell. During a penetration test, a security tester can install a RAT on a compromised host so that it can communicate with a command and control server. Each engagement's specific mechanisms are different, but C2 typically entails one or more covert communication channels between devices in the client organization and a platform under the tester's control. These channels of communication are employed to download additional malicious payloads, send data back to the tester, and issue commands to infected devices. A C2 could use a variety of communication channels, including DNS and HTTP.

In the case of HTTP, communication is done through headers, cookies, or post/get parameters, and is usually encrypted.

For DNS, encrypted data is typically divided into chunks and sent sequentially in the form of subdomains.



```
GET /app.php HTTP/1.1
Host: adv.epostoday.uk
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/85.0.4183.121 Safari/537.36 Edg/85.0.564.63
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

HTTP/1.1 200 OK
Date: Thu, 24 Sep 2020 17:31:05 GMT
Server: Apache
Upgrade: h2,h2c
Connection: Upgrade, Keep-Alive
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 423
Keep-Alive: timeout=5, max=75
Content-Type: text/html; charset=UTF-8

<script>

        let d = -new Date().getTimezoneOffset();
        let n = Intl.DateTimeFormat().resolvedOptions().timeZone;

        function set_cookie (name, value, minutes) {
```

Figure 1: HTTP Communication between Server and Client

1.3 Framework Architecture

The main components of a C2 are:

- C2 Server.
- Agents and Implants.
- Tasks and Commands.

Through a management API, the tester will interact with the C2 Server to carry out tasks like listing implants, assigning them tasks, and receiving results. In order to retrieve tasks and return results, the implant will also communicate with the Team Server using its C2 protocol, such as HTTP. There must be a "link" between these two parts, or a method of transmitting data from one to the other. You can make sure that these components have access to the classes, methods, and data they require to work by using Dependency Injection (DI) in Golang.

"Services" are the items that are added to the DI container. A class that implements an interface will be present in a service.

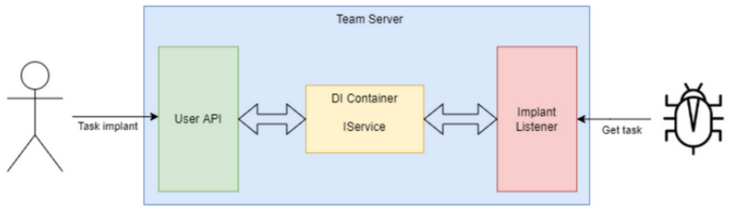


Figure 2: C2 Architecture

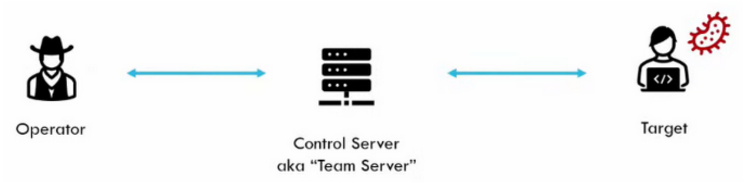


Figure 3: Simpler C2 Architecture

1.4 Command and Control Usage

- Extraction of credentials, operational documents, data, employee records, and other sensitive data can all be copied or transferred to the tester's server.
- Antivirus/Endpoint Detection and Response (AV/EDR) evasion through obfuscation and encrypted end-to-end communication using DNS or HTTP.
- Execute tasks and command through the Agent in the compromised host.
- Move Laterally or Horizontally in the network.
- Automate exploitation techniques and planify tasks.
- Persistence in the compromised host or network.
- Sessions Management.
- Facilitate working as a team of testers by sharing resources.
- Proxying through multiple endpoints.

1.5 Command and Control Models

1.5.1 Centralized

A centralized command and control model works similarly to a client-server relationship. A malware "client" will contact a C2 server and request instructions. In practice, an attacker's server-side infrastructure is frequently far more complex than a single server, and may include re-directors, load-balancers, and defense measures designed to detect security researchers and law enforcement. C2 activity is frequently hosted or masked using public cloud services and Content Delivery Networks.

1.5.2 Peer2Peer

In a P2P CC model, command and control instructions are distributed decentralization, with botnet members relaying messages between one another. Some bots may continue to serve as servers, but there is no central or "master" node. This makes it much more difficult to disrupt than a centralized model, but it also makes it more difficult for the attacker to issue commands to the entire botnet. P2P networks are sometimes used as a backup mechanism if the primary C2 channel fails.

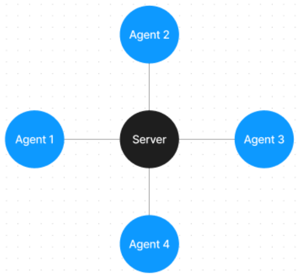


Figure 4: Centralized Model

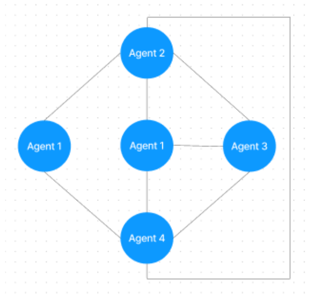


Figure 5: P2P Model

1.6 Example of Known C2 Frameworks

Numerous C2 frameworks are commonly used by professionals on a daily basis. We drew inspiration from the architecture and features of the following frameworks to develop our own framework

1.6.1 Cobalt Strike

Cobalt Strike is a commercial adversary simulation software marketed to red teams ranging from ransomware operators to espionage-focused APTs (APTs). Many network defenders have seen Cobalt Strike payloads used in intrusions, but for those who have not used Cobalt Strike as an operator, understanding the many components and features included in this framework can be difficult.

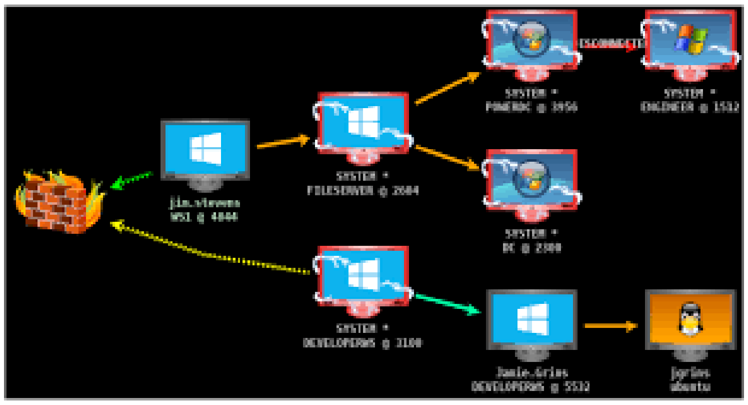
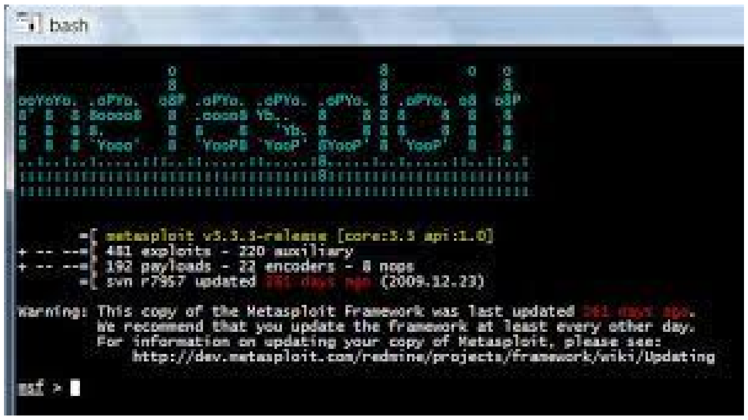


Figure 6: Cobalt strike network mapping

1.6.2 Metasploit

The Metasploit framework is a powerful tool that can be used by both cybercriminals and ethical hackers to investigate systemic vulnerabilities on networks and servers. It can be easily customized and used with most operating systems because it is an open-source framework. The pen testing team can use Metasploit to introduce ready-made exploits into a network to probe for weak spots.



```
       =[ metasploit v3.3.3-release [core:3.3 api:1.0]
+ -- --=[ 481 exploits - 220 auxiliary
+ -- --=[ 192 payloads - 22 encoders - 8 nops
       =[ svn r7957 updated 361 days ago (2009.12.23)

Warning: This copy of the Metasploit Framework was last updated 361 days ago.
         We recommend that you update the framework at least every other day.
         For information on updating your copy of Metasploit, please see:
             http://dev.metasploit.com/redmine/projects/framework/wiki/Updating

msf >
```

Figure 7: Metasploit terminal view

1.7 gRPC

gRPC is an open-source remote procedure call (RPC) framework developped by Google that enables efficient communication and data exchange between distributed systems. At it's core, gRPC uses Protocol Buffers (protobuf) to define the structure of messages, services and data you'll be dealing with. That'll allow gRPC to generate a client/server code in multiple programming languages and different platforms

2 Achitecture

Adhering to best practices in software architecture and to guarantee a robust foundation for the framework, all design phases have been meticulously executed.

2.1 Use cases Diagram

To kick off the project, we began by defining the use case diagram. Our framework not only provides essential C2 functionalities but also offers advanced features that streamline team management across multiple projects. In typical usage, the team would deploy the framework on a cloud instance or local server, allowing all members to access it by launching the framework in client mode on their devices.

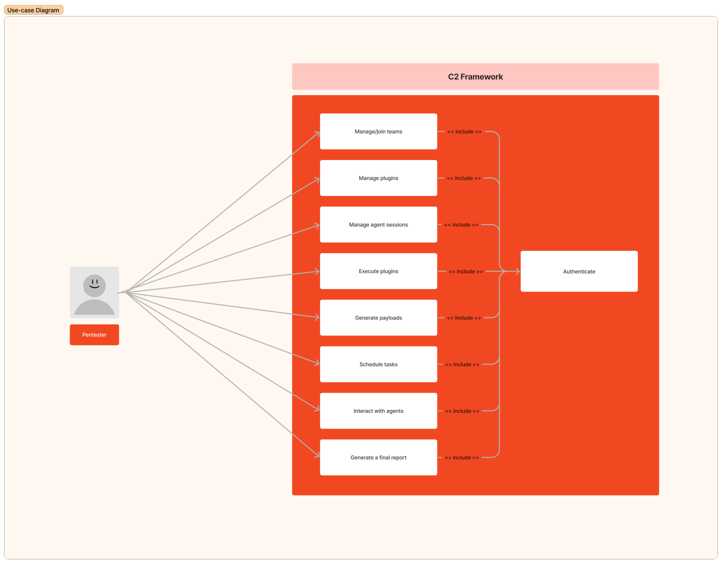


Figure 8: Usecase Diagram

2.2 ERP Diagram

The Entity-Relationship (ER) diagram presents a simplified, high-level view of the framework. In practice, we anticipate more diverse data due to the framework's extensible nature. Our goal is to provide greater flexibility for plugin development and interaction with the main framework.

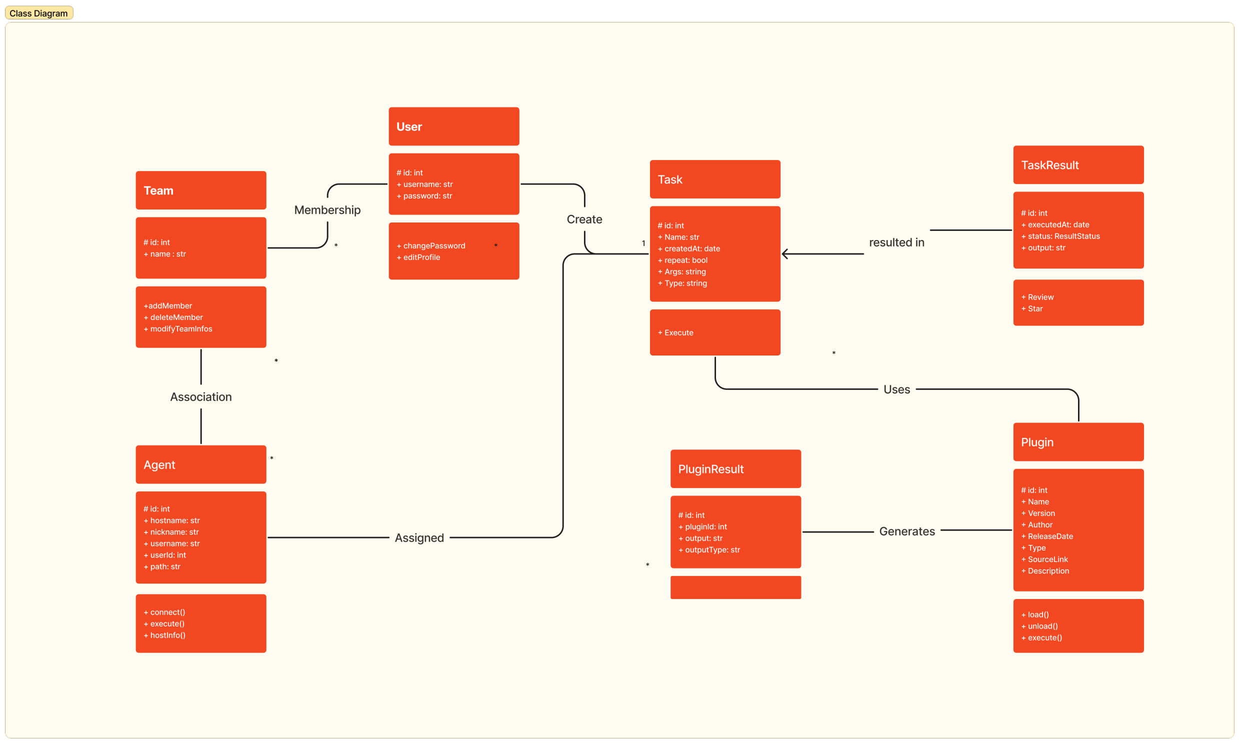


Figure 9: ERP Diagram

2.3 Core

The core module of the framework houses its essential functionalities. All other modules primarily operate independently, with the core module injecting the required services into these modules as needed. This architecture promotes accelerated development time, more concise and readable code, and enhanced debugging efficiency. Moreover, it enables easier bug tracing based on their type, such as logic, communication, or command-line interface (CLI) issues.

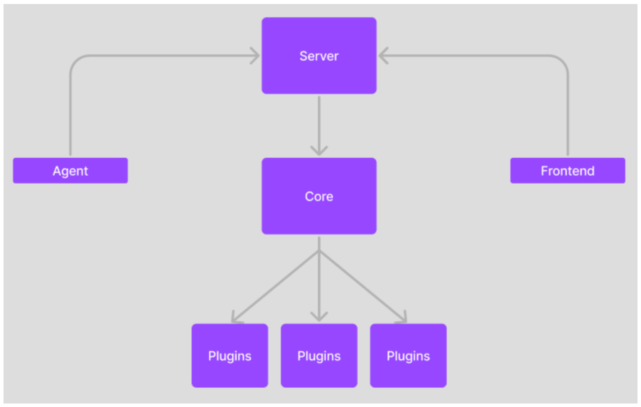


Figure 10: Interactions between modules

2.4 Server

The server module of the framework serves as a central point of communication and coordination by exposing an API using gRPC. It maintains a secure connection with both the clients (Testing team) and the agents by allowing TLS connections. It also plays the role of a communication channel, allowing testers to effectively control and manage the agents remotely, as well as a bridge to access the functionalities offered by the core module.

2.4.1 One code different communication protocols

One of the most critical requirements for our project was to establish seamless communication between the client's frontend and the server. Initially, we attempted to convert the gRPC code into HTTP to enable this communication. However, this approach posed several challenges, such as identifying failed requests and delayed responses.

Fortunately, we discovered an excellent library called gRPC-Gateway. This library is a HTTP JSON proxy that is generated from the same gRPC code and can transform HTTP requests into gRPC and send their results as HTTP responses. The gRPC-Gateway not only simplifies communication between the client and server but also provides a powerful feature that allows our framework to be accessed using both gRPC and HTTP.

Moreover, the HTTP Proxy layer does not hide the gRPC server running behind it, which means that our framework's gRPC server can still be accessed directly. This feature provides greater flexibility and accessibility, making our framework more user-friendly and efficient. Overall, the gRPC-Gateway has been a game-changer for our project, enabling us to achieve our communication goals and deliver a high-quality product to our clients.

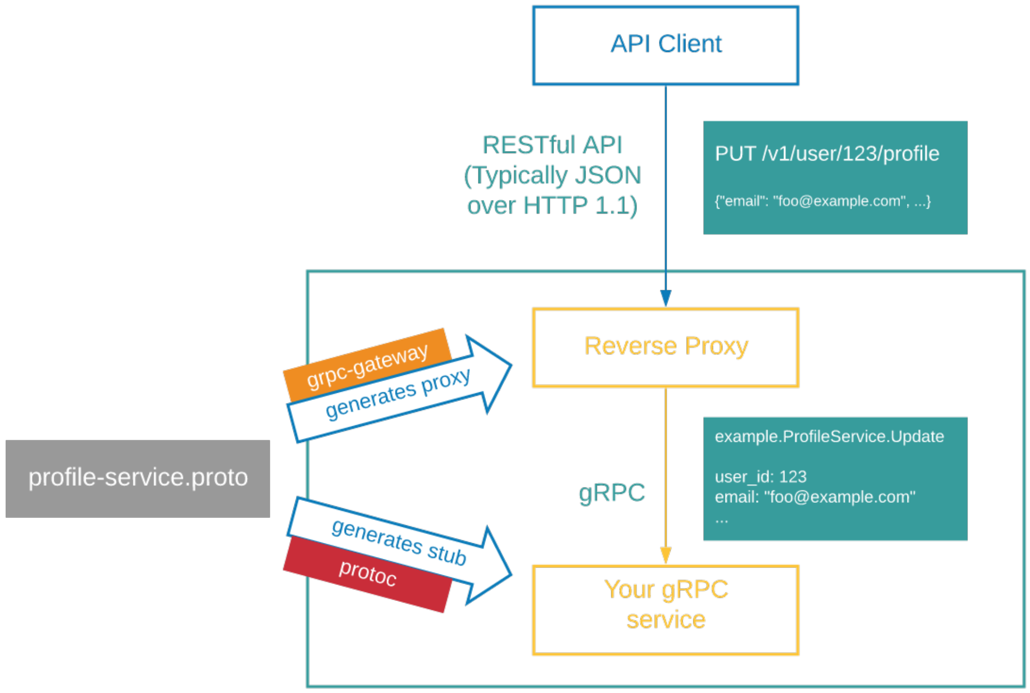


Figure 11: HTTP Gateway into gRPC

2.5 Client

The client module of the framework serves as a user-friendly, easy to use interface for the testing team to interact with the system.

2.6 Plugin

Plugins are external code modules that are dynamically loaded and executed during runtime to add additional functionality to the framework.

2.7 Agent

Agents are software implants that are deployed onto a victim's machine to establish a connection between the attacker's server and the compromised system. Once deployed, agents can be used to remotely control the victim's machine, exfiltrate data, or perform other malicious activities.

2.8 Architecture problems

By adopting this architecture, we were able to implement the principle of dependency injection throughout the application's code. To gain a better understanding of the relationship between the various components of the framework, let's examine the components diagram.

As depicted in the diagram, both the server and client rely on the configuration service provided by the core module. However, the server is more closely coupled with the core module, as it consumes a greater number of services. At one point, we considered revising the diagram to reflect the following changes:

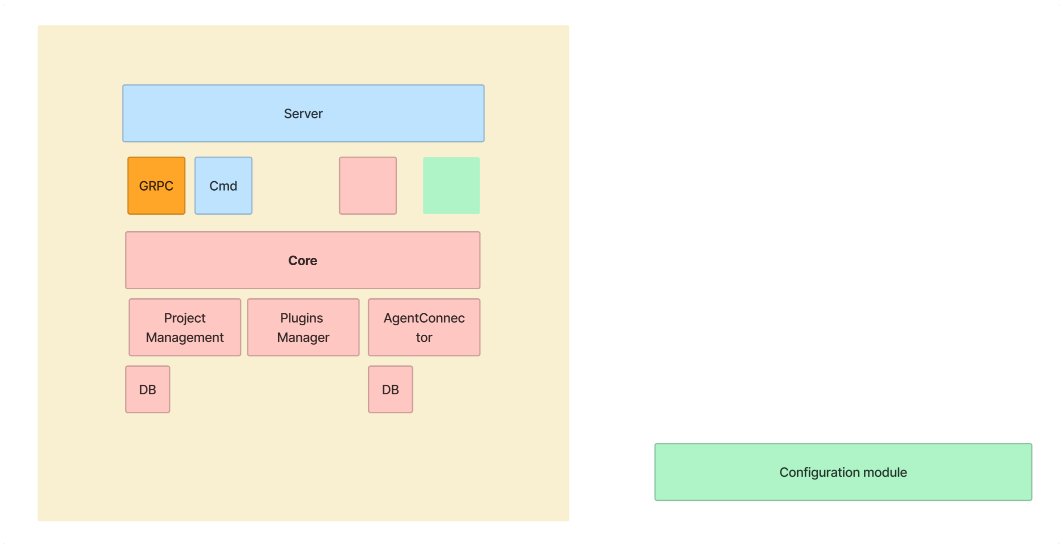


Figure 12: Separating configuration from core

However, we aimed to make the core module self-contained, with all functionalities included. This way, the server could modify the communication methods and the way services are exposed as needed.

3 Realization and tools

3.1 Core services

3.1.1 Configuration

Our objective was to create a configurable framework that provides numerous powerful functionalities without attempting to conceal its inner workings or wrest control from the user or team. The framework's configuration is straightforward, covering only the essential aspects, such as:

- Ports' configuration
- JSON Web Token (JWT) secret
- Plugins' directory

The configuration file is automatically generated in the user's home directory. Additionally, the framework provides a command-line interface (CLI) utility to verify the accuracy of custom configurations.

3.1.2 Plugins manager

The plugin manager serves as the core of our plugin system, encapsulating all functionalities, including:

- Load specific plugin
- List All plugins
- Execute plugin
- Get plugin's info

The plugin system is relatively straightforward, and we will delve into its details in the plugins section.

3.1.3 Providers

In the context of software development, providers serve as the interfaces that offer the primary business logic of an application and expose it to be consumed by controllers, regardless of the protocol used, whether it be gRPC, HTTP, or another. These providers act as the backbone of the application, encapsulating the core functionality and making it available for use by the controllers. By separating the business logic from the controllers, providers enable greater flexibility and modularity in the application's design, allowing for easier maintenance and scalability.

To implement the plugin provider, we had to break the rule of separation between core services. This was a necessary step to enable the provider to access the core services and utilize their functionality. While this may seem like a violation of best practices, it was a deliberate decision made to ensure the smooth integration of the plugin provider and the core services. By doing so, we were able to create a more cohesive and efficient system that delivers the desired functionality without compromising on quality or performance.

```
func InitServices(db *orm.ORMConnection, frameworkConfig config.Config) services.Services {
userService := services.NewUserService(db)
return services.Services{
TeamService:         services.NewTeamService(db),
AgentService:        services.NewAgentService(db),
HostService:         services.NewHostService(db),
TaskService:         services.NewTaskService(db),
ReportService:       services.NewReportService(db),
PluginResultService: services.NewPluginResultService(db),
AuthService:         services.NewAuthService(userService, frameworkConfig),
}
}
```

3.2 Listener's API

The ability to handle requests via a TCP/UDP connection is the most important feature of a C2 server. In this case, a Listener is required to monitor the traffic coming to the server via a specific port, which could be the implant's heartbeats, command results returned from the compromised machine, or simply a new agent added to our list. The listener can simply be a socket waiting for an entity to connect to (as in TCP-based implants) or a rogue DNS server that will receive communication via the subdomain section of a URL, parse it, and resume the message. When in listening mode, the utility netcat is an example of a listener. Where we could monitor traffic passing through a particular port.

In our case, the listener is implemented in gRPC. It doesn't keep the connection open instead it works as an endpoint that receives data periodically and update it the database accordingly. This makes the agent more hard to detect.

3.3 Agent API

The Server component of our C2 framework exposes an Agent API that serves as a communication interface between the Server and the deployed agents. Through this API, the Server establishes secure connections with the agents, allowing for data transfer and command execution. The Agent API provides a standardized set of methods and protocols that enable the Server to effectively communicate instructions, retrieve information, and receive responses from the agents.

3.3.1 Agent's communication with the server

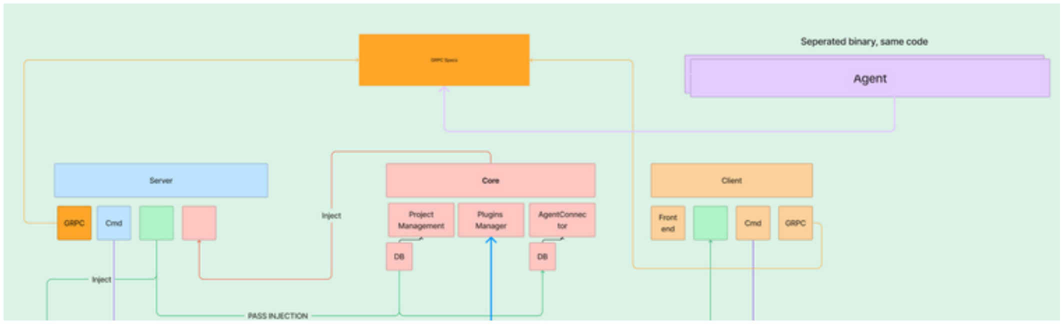


Figure 13: Component diagram - Agent interaction

For the server-agent interaction, we'll have mainly 2 type of interactions:

- Initializing connection
- Agent identification

During agent initialization, the framework registers the agent in the database, making it known to the system. This enables us to assign tasks to the agent by passing the UUID assigned during the registration process. This approach simplifies the process of tracking the agent's location throughout the network. However, it does not guarantee that the agent still exists, which means that dispatched tasks may never be executed, causing the penetration tester to wait indefinitely. Therefore, we recommend treating agents as non-existent if they fail to connect within a few seconds. The communication with the agent is based on gRPC.

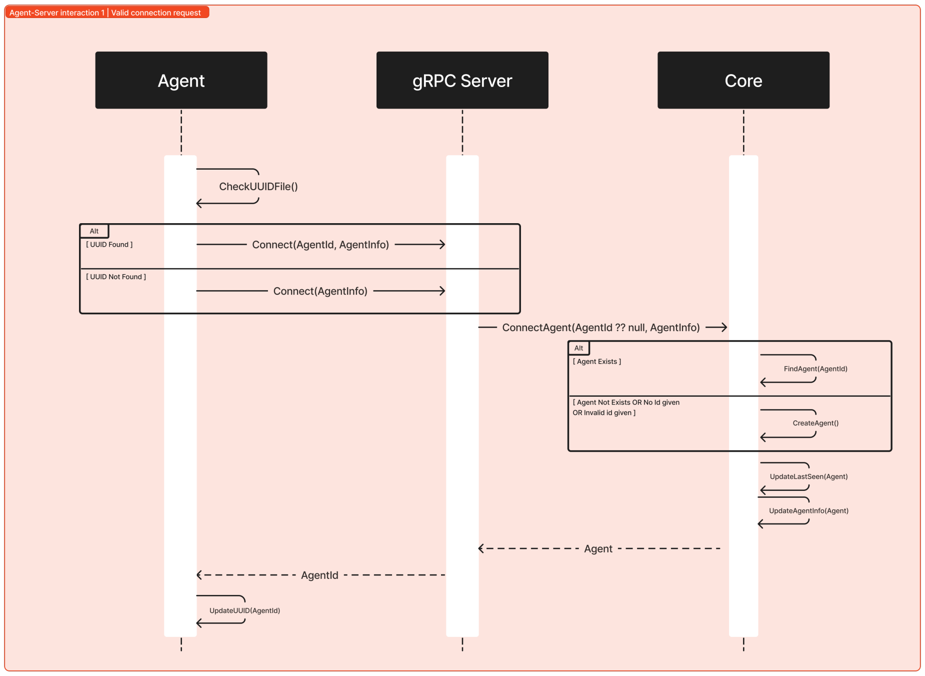


Figure 14: Sequence Diagram - Agent Connection

```
agent, err := s.AgentService.ConnectAgent(
    agentInfo.GetId(),
    &entity.AgentModel{
        Username: agentInfo.GetUsername(),
        Uid:      agentInfo.GetUserId(),
        Hostname: agentInfo.GetHostname(),
        Cwd:      agentInfo.GetCwd(),
    },
)
....
```

3.3.2 Agent's maintaining connection and executing tasks

As we said the connection between agent and the server is not a stream but it is periodic. Actually, the agent connects in a periodic manner to the server and checks if there are any tasks for it to execute. If there are, it will fetch them, execute them on a thread then collects the execution's results and send them back to the server. All new results received by the server are marked as unseen, so the user's will be informed of their existence.

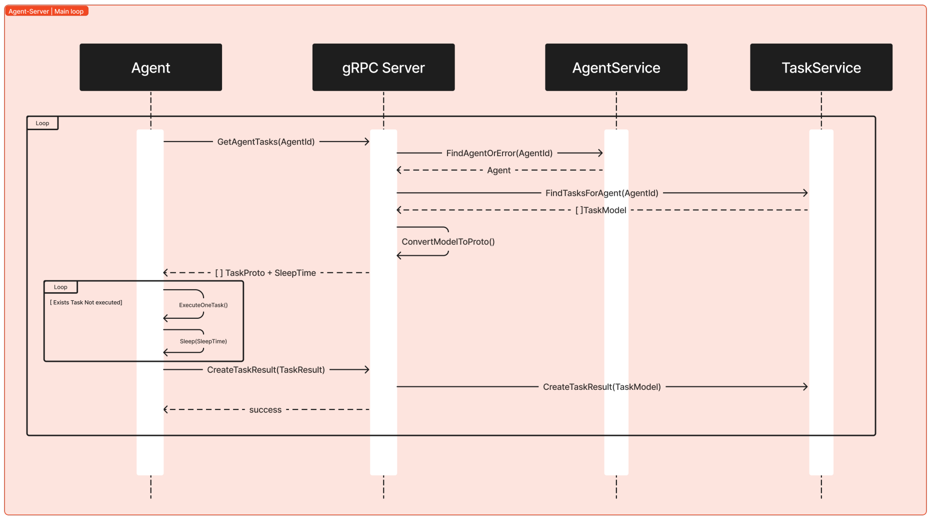


Figure 15: Sequence Diagram - Agent Connection

As previously mentioned, the connection between the agent and the server is not a continuous stream but rather periodic. Specifically, the agent connects to the server at regular intervals to check for any assigned tasks. If there are tasks, the agent fetches them, executes them on a thread, collects the results, and sends them back to the server.

Any new results received by the server are marked as unseen, ensuring that users are notified of their existence.

```
for {
    tasks, sleep, err := FetchTasks(services)
    if err != nil {
        log.Panic("Unable to fetch tasks")
    }
    fmt.Println("Fetched", len(tasks), "tasks")

    for _, task := range tasks {
        // This can become multithreaded in the future
        // But will require sync between { SendResult, Sleep } blocks
        result, err := ExecuteTask(task)
        if err != nil {
            log.Panic("Unable to execute task")
        }
        err = SendResult(services, result)
        if err != nil {
            log.Panic("Unable to submit task result")
        }

        time.Sleep(time.Duration(sleep * uint32(time.Second)))
    }
    time.Sleep(time.Duration(sleep * uint32(time.Second)))
}
....
```

4 Plugins System

The framework's primary feature is the ability to reuse enumeration and exploitation scripts across different projects. Additionally, users can leverage pre-existing scripts to jumpstart their assessments. These capabilities are made possible by the plugin system, which provides a foundation for building a limitless framework on top of the basic C2 functionalities.

In this section, we will delve into the design and implementation of the plugin system, highlighting the challenges and design decisions we encountered along the way.

4.1 Plugins feature design

To implement the plugin system, we needed to provide plugin developers with the freedom to code the desired functionality while ensuring compatibility with the framework. Additionally, plugins should be loaded on demand, as loading a large set of plugins during framework launch, without actually needing them later, can cause significant performance issues and resource usage.

To begin, we established the following questions that the framework needed to address:

- How can we ensure that plugins are compatible with the framework and do not cause conflicts or errors?
- How can we ensure that plugins are loaded on demand, without impacting the framework's performance or resource usage?
- How can we enable users to easily manage and configure plugins within the framework?

4.2 Dynamic code execution in Golang

The Go programming language has a mature and comprehensive ecosystem that provides developers with all the necessary packages and tools to develop solutions quickly. In our case, we were able to leverage the plugin package from the standard library to meet our needs. Although the package is not very mature and may exhibit unexpected behavior in some edge cases, it allowed us to load plugins and execute them in a straightforward and efficient manner. Overall, the plugin package proved to be a valuable asset in the development of our framework, enabling us to easily incorporate plugins and extend the functionality of the system

4.3 Plugin's unified interface

Our first step in implementing the plugin system was to define a general interface for the plugins. This interface serves as the only means by which a plugin can function within the framework. Any violation or misimplementation of this interface will result in the plugin failing to load.

By establishing a clear and well-defined interface, we were able to ensure that all plugins adhere to a consistent set of standards and can be easily integrated into the framework. This approach also enabled us to maintain a high level of compatibility and flexibility, allowing for the seamless incorporation of new plugins as needed.

4.3.1 Identifying plugins

To identify a plugin before loading it, the only available method is to use its filename. Once loaded, the framework retrieves the necessary data to use the plugin as if it were a part of its code, effectively integrating it dynamically. To facilitate this process, we defined a struct for all plugins that encapsulates the plugin's metadata and function-related information.

By using this struct, we were able to store all relevant information about the plugin in a single location, making it easier to manage and integrate into the framework.

```
func New(service services.ITaskService) plugins.IPlugin {
    return &EvilPlugin{
        Plugin: plugins.Plugin{
            Metadata: plugins.Metadata{
                Version:     "1.0",
                Author:      "Evil Corp",
                Tags:        []string{"evil", "malware"},
                ReleaseDate: "2023-05-01",
                Type:        1,
                SourceLink:  "https://evilcorp.com/plugins/evil",
                Description: "This plugin does evil things",
            },
            PluginInfo: plugins.PluginInfo{
                Name: "EvilPlugin",
                Options: map[string]string{
                    "target":  "string",
                    "agentId": "string",
                },
                ReturnType: "string",
            },
        },
        TaskService: service,
    }
}
```

4.3.2 Handling arguments in dynamically executed code

Due to the dynamic nature of code execution, argument types can be lost in the process. To address this issue, we implemented our own type system, which places the responsibility of passing the correct types to the framework on the user, while still allowing the plugin to validate these types. By implementing our own type system, we were able to ensure that the framework could handle a wide range of argument types.

```
func (p *EvilPlugin) SetArgs(args ...interface{}) error {
    fmt.Println("Setting args")
    arg1, ok := args[0].(ArgValue)
    if !ok {
        return errors.New("Bad first argument")
    }
    ...
```

4.3.3 Plugin’s types

Also the plugin’s types is divided into two:

- Session opener: Executing this type of plugins might result opening a remote session on the victim’s machine. In other terms, the plugins is able to gain access to a another machine by exploiting a vulnerability in the victim’s system. After the exploitation, the framework will upload an agent to the victim and will use it to open a connection between them.

- Info Retriever: Executing this type of plugins will return an output.

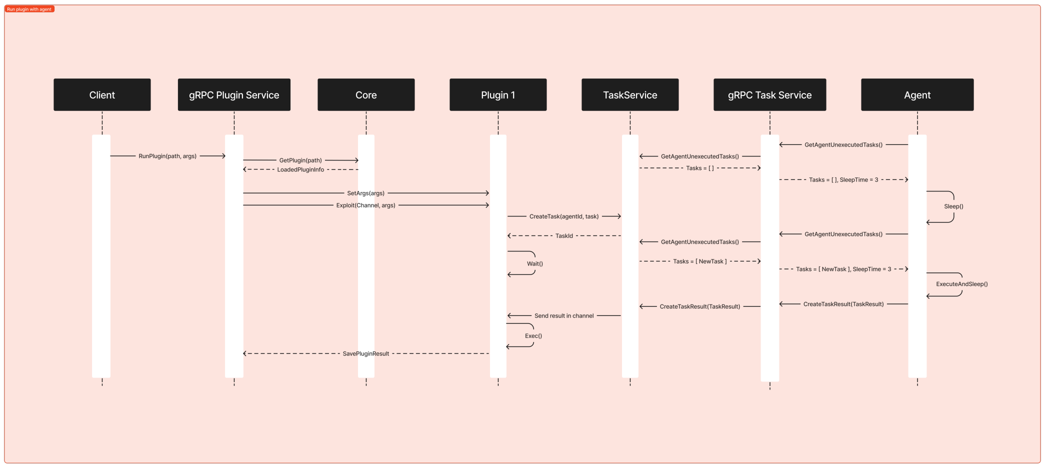


Figure 16: Session opener plugin sequence diagram

4.3.4 Plugin’s execution

All plugins have the same starting point which is an exploit function and setArgs to set arguments. To launch a specific plugin, the framework loads it into a map data structure inside the memory and whenever it’s called it will check if it can retrieve it otherwise it will load it again.

```
func (p *EvilPlugin) Exploit(Channel chan *entity.TaskResultModel,
args ...interface{}) []byte {
    // Do evil things with the args
    fmt.Println("Do evil things to", p.targetIp)

    return []byte(fmt.Sprint("EvilPlugin attacking", p.targetIp, "\nOutput:", result.Output))
}
```

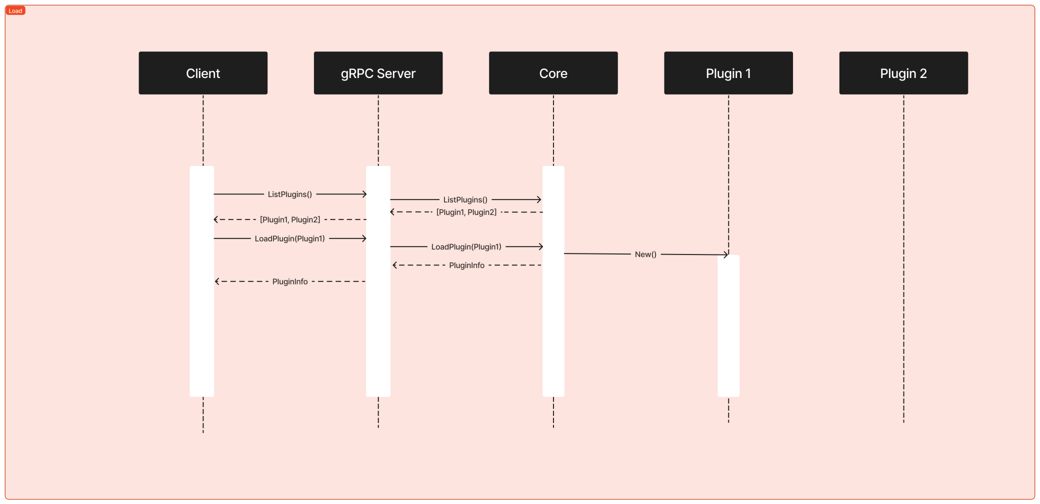


Figure 17: Loading plugin sequence diagram

5 Tasks and commands

The majority of communication between the C2 server and the compromised host will be from launching tasks and receiving their results. . Tasking is an important aspect of a C2, as it allows the tester to run shell commands or pre-coded snippets of code in the compromised host. Each task will have its own id. A command type to run, arguments for that command and the agent Id for the agent that will run the task.

```
const (
    TASK_TYPE_PING  TaskType = "PING"
    TASK_TYPE_SHELL TaskType = "SHELL"
)

type TaskModel struct {
    UUIDModel
    Name      string
    Type      TaskType ‘json:"type" sql:"type:ENUM(’PING’, ’SHELL’)"‘
    Args      string
    AgentId   uuid.UUID ‘type:"uuid"‘
    Agent     AgentModel
    CreatorId uuid.UUID ‘type:"uuid"‘
    Creator   UserModel ‘gorm:"foreignKey:CreatorId"‘
}
```

The task’s types are either:

- PING: to see if the agent or the target is still accessible or not.
- SHELL: to execute a shell command

We will also require a way to receive and store the results of the commands. Each Result will have a unique id and string containing the data.

```
type TaskResultModel struct {
    UUIDModel
    ExecutedAt time.Time
    Status     int32
    TaskID     uuid.UUID ‘type:"uuid"‘
    Task       *TaskModel
    Output     string
    Seen       bool ‘gorm:"default:false"‘
}
```

5.1 Nature of task execution

Task execution is a critical component of any C2 framework, as it enables the attacker to maintain access to the compromised machine after exploitation. Remote code execution is a particularly dangerous vulnerability, as it allows the attacker to gain control of the machine and potentially escalate privileges, pivot to other hosts on the network, and carry out further attacks.

To ensure that task execution is carried out in a controlled and secure manner, it is important to implement well-defined scripts for exploitation, enumeration, and scanning. This functionality is one of the most complex aspects of the framework, as it requires careful consideration of security, performance, and scalability.

Before discussing the current implementation of task execution in our framework, it is important to distinguish between plugins and agents. Agents are pieces of code that are deployed on the victim's machine and should expect all types of tasks to execute. On the other hand, plugins are external pieces of code that can have various functionalities, their tasks are already set by their developer. Despite their differences, we treat both entities in the same manner when it comes to their execution within our framework.

5.2 process of task execution

The task execution flow for both them is as follows :

- Start execution in another thread
- When finished, the plugin or the agent will push the results into the channel shared between it and the main thread
- Now, the framework will consume the results from the channgel and add it to the databse.
- The user is notified of the new incoming resutls

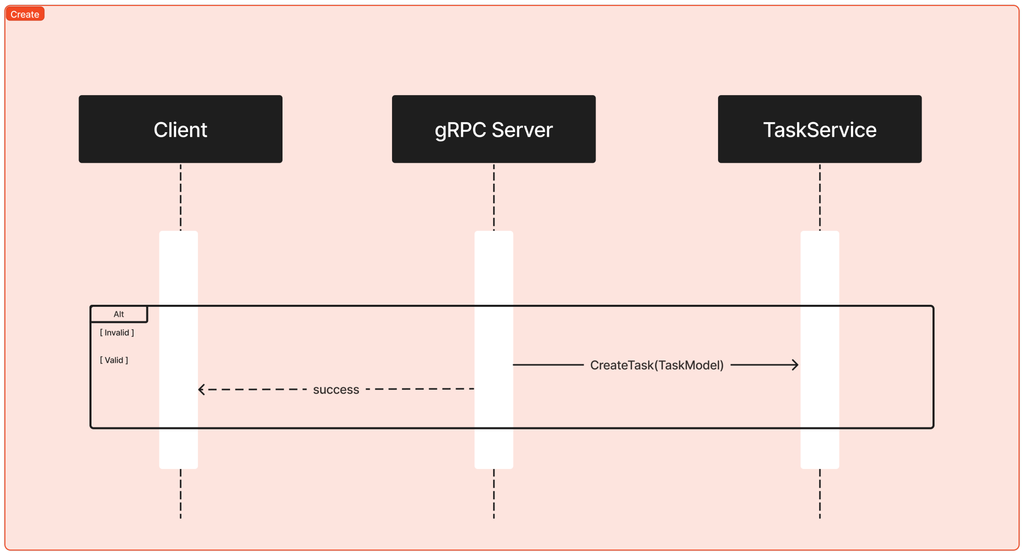


Figure 18: Create task sequence diagram

```
     taskProto := in.GetTask()
fmt.Println(in)
err = ValidateTaskProto(taskProto)
if err != nil {
return &proto.CreateTaskResponse{}, err
}

args := ""
if len(taskProto.GetArgs()) != 0 {
args = taskProto.GetArgs()[0]
}
var task = entity.TaskModel{
Name:    taskProto.GetName(),
Args:    args,
Type:    entity.TaskType(taskProto.GetType().String()),
AgentId: agentId,
// CreatorId: ,
}

err = s.TaskService.CreateTask(&task)
```

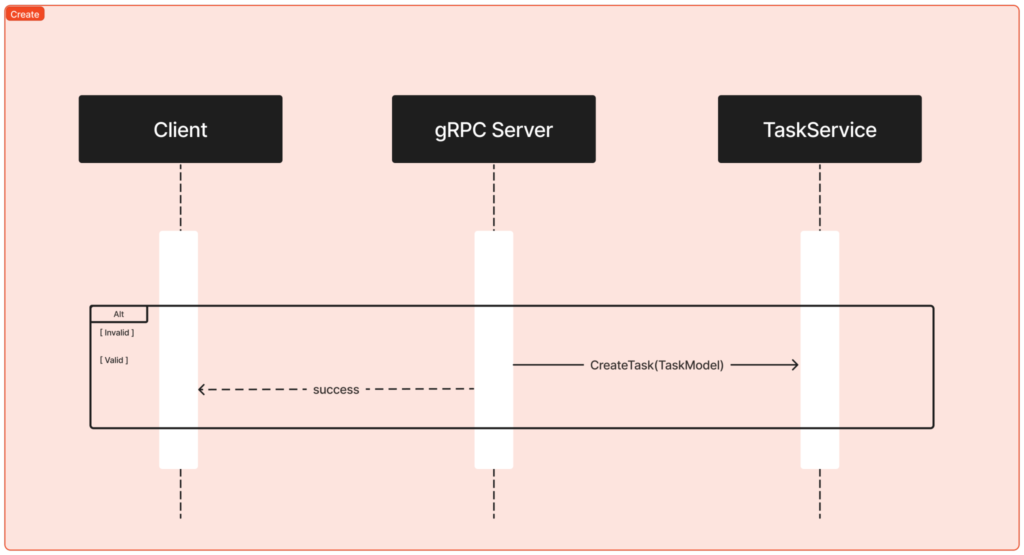
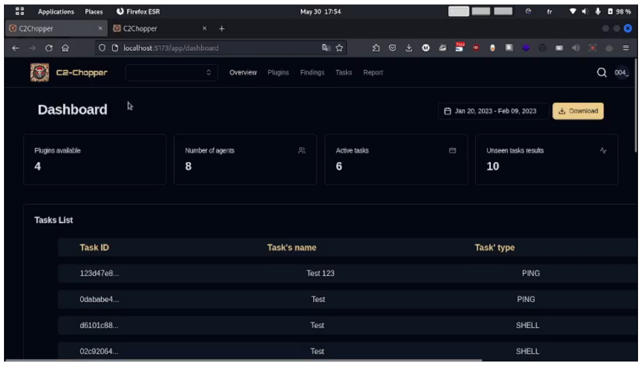


Figure 19: Getting task's results sequence diagram

```
    func (s *TaskService) CreateTaskResult(ctx context.Context, in *proto.CreateTaskResultRequest
agentInfo := in.GetInfo()
if agentInfo != nil {
s.AgentService.ConnectAgent(
agentInfo.Id,
&entity.AgentModel{
Username: agentInfo.GetUsername(),
Uid:      agentInfo.GetUserId(),
Hostname: agentInfo.GetHostname(),
Cwd:      agentInfo.GetCwd(),
},
)
}
```

6 Results



Plugins available	Number of agents	Active tasks	Unseen tasks results
4	8	6	10

Tasks List

Task ID	Task's name	Task' type
123d47e8...	Test 123	PING
0dababe4...	Test	PING
d6101c88...	Test	SHELL
02c92064...	Test	SHELL

Figure 20: Dashboard view

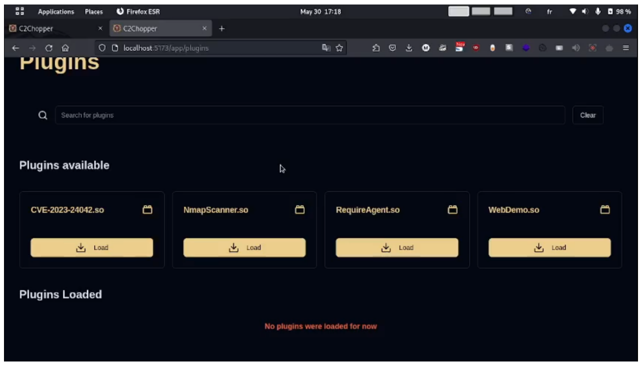
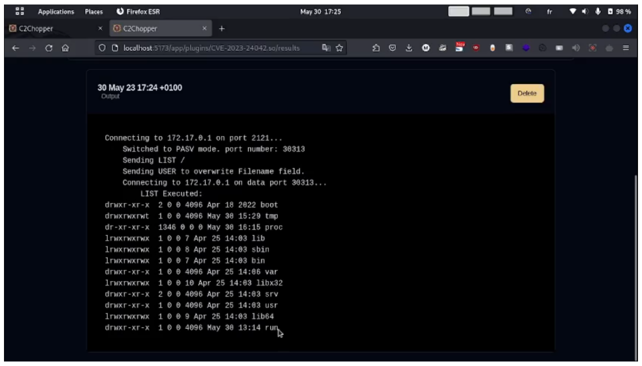


Figure 21: Plugins listing



```
Connecting to 172.17.0.1 on port 2121...
    Switched to PASV mode. port number: 30313
    Sending LIST /
    Sending USER to overwrite Filename field.
    Connecting to 172.17.0.1 on data port 30313...
        LIST Executed:
drwxr-xr-x  2 0 0 4096 Apr 18 2022 boot
drwxrwxrwt  1 0 0 4096 May 30 15:29 tmp
dr-xr-xr-x  1346 0 0 0 May 30 16:15 proc
lrwxrwxrwx  1 0 0 7 Apr 25 14:03 lib
lrwxrwxrwx  1 0 0 8 Apr 25 14:03 sbin
lrwxrwxrwx  1 0 0 7 Apr 25 14:03 bin
drwxr-xr-x  1 0 0 4096 Apr 25 14:06 var
lrwxrwxrwx  1 0 0 10 Apr 25 14:03 libx32
drwxr-xr-x  2 0 0 4096 Apr 25 14:03 srv
drwxr-xr-x  1 0 0 4096 Apr 25 14:03 usr
lrwxrwxrwx  1 0 0 9 Apr 25 14:03 lib64
drwxr-xr-x  1 0 0 4096 May 30 13:14 run
```

Figure 22: Task output

7 Vision

Our team is thrilled to announce that we are planning to make our tool open source. By doing so, we hope to attract more people to the project and build a library of plugins that can enhance its functionality.

We recognize that the framework is still in its early stages and has a lot of room for improvement. While it currently provides the minimum functionalities promised, we are committed to enhancing the user experience and providing better means of setup and installation. To achieve this goal, we plan to leverage the power of Golang, which can be compiled into cross-platform binaries. This feature will enable us to create a more user-friendly installation process and make the framework more accessible to a wider audience. We are thrilled about the potential of this project and look forward to collaborating with the open-source community to make it even better.

In addition, we are excited about the potential of building pipelines with plugins and enhancing them with filters and map reduce functions. This feature will make the automation of the penetration testing process easier and more enjoyable in the long term. One crucial aspect of this feature will be the ability to share recipes, enabling users to collaborate and share their knowledge.

8 Contributions

Yassine Belkhadem served as the product owner for this project, defining the necessary functionalities and prioritizing them based on their importance. Meanwhile, Mongi Saidane played a crucial role in making technical decisions, particularly in the development of the agent's algorithms.

Mongi Saidane was responsible for developing the remote agents, executing tasks, and creating two plugins for testing. On the other hand, Yassine Belkhadem played a significant role in developing the client, managing plugins, creating entities, and developing the CLI.

Both team members played a critical role in the success of the project, leveraging their unique skills and expertise to deliver a high-quality product. Their collaboration and teamwork were essential in ensuring that the project met all requirements and exceeded client expectations.

9 Conclusion

The goal of this project was to develop a fully functional command and control framework that can execute commands through an API that is hosted on a server, leverages gRPC communication for communication between the compromised host and the host, and is extensible for other communication techniques. A different security researcher can add his own commands and listeners to the Team- Server and Agent without having to completely redo them because they are both extensible. Finally, the opportunity to work with the Golang programming language and gRPC protocol made this project very rewarding for us. We will benefit more from this first experience when creating Command and Control Frameworks in the future that are more complex and obfuscated.

List of acronyms

- DNS - Domain Name Service
- JSON - JavaScript Object Notation
- RAT - Remote Access Trojan
- APT - Advanced Persistent Threat
- DI - Dependency injection
- CLI - Command line interface

Bibliography

[1] Go in action ,William Kennedy with Brian Ketelsen and Erik St. Martin Foreword by Steve Francia

[2] Black Hat Go, Go Programming for Hackers and Pentesters by Tom Steele, Chris Patten, and Dan Kottmann

[3] Efficient Go by Bartlomiej Plotka

[4] Building C2 Implants in C++: A Primer: Introduction (https://shogunlab.gitbook.io/building-c2-implants-in-cpp-a-primer/)