DeadDrop

a clever command and control framework for penetration testers

Design (P3)

Jann Arellano, Brian Buslon, Keaton Clark, Lloyd Gonzales
Team 08

Advisor:
Shamik Sengupta
Professor
University of Nevada, Reno

CS 425, Fall 2023
Instructors: Sara Davis, Devrin Lee, Vinh Le

Department of Computer Science and Engineering
University of Nevada, Reno
November 17, 2023

Table of Contents

Abstract

We describe DeadDrop, a command and control (C2) framework used in post-exploitation activities and penetration testing to create malware payloads, manage compromised devices, and generate operational reports. Unlike existing frameworks that communicate directly with attacker domains, DeadDrop focuses on leveraging features in legitimate websites such as YouTube and Wikipedia to communicate with devices and exfiltrate data, masking its activity within the noise of popular websites. The novelty of this approach is valuable to offensive and defensive cybersecurity professionals alike, as corporate networks and firewalls often "trust" the traffic of large, well-known websites. By abusing this trust, DeadDrop provides security engineers with new insight into identifying covert attacker techniques and security weaknesses. Here, we detail DeadDrop's architecture, major processes, and ongoing development work.

I. Introduction

DeadDrop is a command and control (C2) framework used in post-exploitation activities and penetration testing to create malware payloads, manage compromised devices, and generate operational reports. DeadDrop's primary purpose is to aid legitimate security professionals (the "red team") in an activity known as penetration testing, which assesses the company's ability to defend against attacks and protect its data.

In a penetration test, the red team identifies vulnerabilities in an organization's devices, using these vulnerabilities to install malware that allows them to remotely control and communicate with that device. By leveraging the "trust" of this compromised device, the red team can then plant malware on more valuable devices with sensitive information (some of which may not be directly connected to the public internet), with the goal of exfiltrating this sensitive data out of the network without being caught.

Penetration tests are often coordinated using C2 frameworks, which greatly simplifies the task of managing infected devices while ensuring accountability for actions taken by the red team. The use of a C2 framework in testing greatly benefits security engineers (the "blue team"), as they can identify holes in detecting malicious activity through the reporting and logging functionality provided by the C2 framework.

However, unlike existing frameworks that communicate directly with attacker domains, DeadDrop focuses on leveraging features in legitimate websites such as YouTube and Wikipedia to communicate with devices and exfiltrate data, masking its activity within the noise of popular websites. Messages ("dead drops") can be placed on external services by one party, which can be retrieved by the other party anytime by accessing the external service.

The novelty of this approach is valuable to offensive and defensive cybersecurity professionals alike, as corporate networks and firewalls often "trust" the traffic of large, well-known websites. By abusing this trust, DeadDrop provides security engineers with new insight into identifying covert attacker techniques and security weaknesses. In doing so, it encourages network security engineers to take new approaches to identifying covert communication methods that could be used by skilled threat actors.

In short: DeadDrop provides organizations with the tools needed to develop and test their defenses against these techniques through a highly configurable and extensible framework.

Since Project Assignment 2, we have greatly improved our understanding of the architecture necessary to implement our requirements and overall vision of the project. By taking inspiration from our stakeholder interviews, analyzing the features behind existing frameworks, and early prototyping, we have identified the specific technologies and moving pieces that will need to be part of DeadDrop.

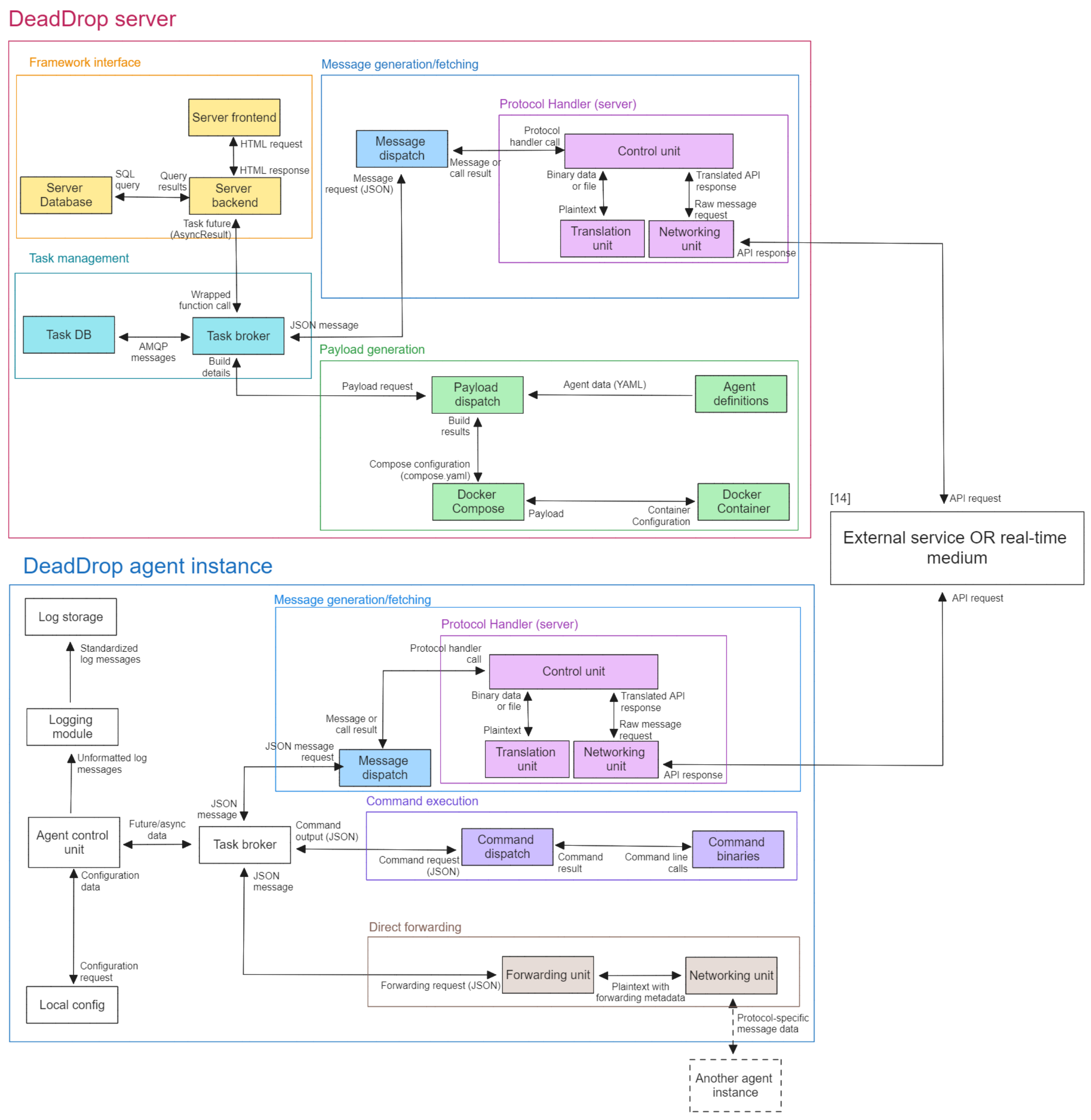
For example, we previously did not have a concrete system for handling asynchronous code or juggling both on-demand and periodic tasks. Through our research, we were able to not only develop a modular system that can withstand rapid changes in requirements, but also identify the specific technologies that could drive this system.

Additionally, we have begun developing internal requirements, expectations, and standards. In addition to enforcing the interfaces between modules that can (and will) be developed in parallel, we believe these will help reduce the number of places where things can go wrong during our winter and spring demonstrations. At this point, we have begun delegating implementation tasks to members of our team, and believe we are in a strong position to present our prototype in the coming weeks.

II. High and Medium-level Design

System Diagram and Overview

A high-level overview of DeadDrop's (non-object-oriented) architectural pattern is shown below.



At the highest level, DeadDrop follows a client-server model, whereby a single attacker-controlled server interacts with one or more agents (clients) installed on remote machines via an arbitrary medium. Each agent is platform-specific and may support different functionality, but must implement certain basic features shared between all agents regardless of implementation language or target platform.

Strictly speaking, the sole purpose of an agent is to accept commands, execute them, and then return their outputs and associated logs. An agent must accept and send messages in the DeadDrop standard message format (described in *Data Structures*, and referred to as "messages" or "plaintext messages" from here onward), which contain the information needed for an agent to execute commands on a remote device. Although the agent may implement this functionality in any form, the architecture for agents presented above is likely to be shared between all agent implementations.

In many cases, the modules of both the server and the agents can be grouped together to represent the specific functionality they provide. The following section has been arranged to describe each group and submodule of DeadDrop in a logical order according to the labels in [brackets] as shown in the diagram above. Please note that **groups of modules/program units all contain an introduction with information that may not be repeated in the individual program unit descriptions.**

A summary of each module has been provided in the table below, with complete descriptions described in the following sections.

[#]	Module	Summary
[1]	Server → Framework Interface → ***Server Backend***	Dynamically generates webpages to reflect the state of the framework; interfaces with the server database to store/access framework information and mediate tasking.
[2]	Server → Framework Interface → ***Server Frontend***	Provides users with a web interface to easily interact with the framework, primarily for issuing commands and viewing outputs and logs.
[3]	Server → Framework Interface → ***Server Database***	Holds the data needed to keep track of individual parts of the framework, such as agents, protocols, and logs. Also holds backend-managed data, such as user and group details.
[4]	Server → Task Management → ***Task Broker***	Offloads asynchronous tasks from the server interface, reporting results to the server backend and performing on-demand and scheduled periodic tasks.
[5]	Server → Framework Interface → ***Task Database***	Database/message broker that may be used by the task broker to store the information necessary to keep track of outstanding tasks.
[6]	Server → Message Generation and Fetching →	Module responsible for retrieving the information and protocol handler needed to communicate with an agent

[#]	Module	Summary
	Message Dispatch	over a specific protocol.
[7]	Server → Message Generation and Fetching → Protocol Handler → ***Control Unit***	Component of a protocol handler responsible for either: - scheduling and generating messages to be sent over a protocol, or; - checking for new or expected messages and reassembling components of these messages over a protocol.
[8]	Server → Message Generation and Fetching → Protocol Handler → ***Translation Unit***	Responsible for translating plaintext messages into one or more binary streams of data to be sent to a particular external service, or vice versa.
[9]	Server → Message Generation and Fetching → Protocol Handler → ***Networking Unit***	Responsible for uploading translated streams of data to one or more target locations (which may involve using an API or specific transport layer protocol), or for retrieving and streams of data from one or more targets.
[10]	Server → Payload Generation → ***Payload Dispatch***	Responsible for interpreting requests to build agents, retrieving data from agent definitions to generate the commands or files necessary for the agent's Docker container to build the agent with the user's configuration settings.
[11]	Server → Framework Interface → ***Agent Definitions***	Static set of files containing the information necessary to build an agent.
[12]	Server → Payload Generation → ***Docker Compose***	Responsible for managing and instantiating the container (or containers) needed to generate the payloads for an agent.
[13]	Server → Payload Generation → ***Docker Container***	Contains the build environment (and any dependencies) that the agent will be constructed in; also generates the final output that is to be installed onto a target device.
[14]	(none) → **External Service or Real-Time Medium**	This represents either: - The external service used to conceal communication between servers and agents - A "standard" protocol and medium used for real-time communication, like a direct TCP connection
[15]	Agent → ***Control Unit***	Manages all decisionmaking for the agent; interprets and acts on messages received from the C2 server.
[16]	Agent → ***Local***	Static files dictating various configuration settings for the

[#]	Module	Summary
	Configuration	agent, such as the agent's ID, any encryption keys in use, logging settings, and so on.
[17]	Agent ⟶ ***Logging Module***	Centralized logging module that writes logs to external storage based on its configuration settings.
[18]	Agent ⟶ ***Log Storage***	Arbitrary form of storage used to hold logs generated by any part of the agent.
[19]	Agent ⟶ ***Task Broker***	Responsible for asynchronously executing and managing tasks and returning their results to the agent control unit.
[20]	Agent ⟶ Message Generation/Fetching ⟶ ***Message Dispatch***	Agent module identical in purpose to the *Message Dispatch* module of the server.
[21]	Agent ⟶ Message Generation/Fetching ⟶ Protocol Handler ⟶ ***Control Unit***	Agent module identical in purpose to the *Control Unit* module of the server.
[22]	Agent ⟶ Message Generation/Fetching ⟶ Protocol Handler ⟶ ***Translation Unit***	Agent module identical in purpose to the *Translation Unit* module of the server.
[23]	Agent ⟶ Message Generation/Fetching ⟶ Protocol Handler ⟶ ***Networking Unit***	Agent module identical in purpose to the *Networking Unit* module of the server.
[24]	Agent ⟶ Command Execution ⟶ ***Command Dispatch***	Responsible for interpreting requests for specific commands to be executed, generating the arguments and calling the associated binary for that command.
[25]	Agent ⟶ Command Execution ⟶ ***Command Binaries***	Standalone binaries that accept command-line arguments and are designed to achieve a single task on a device.
[26]	Agent ⟶ Direct Forwarding ⟶ ***Forwarding Unit***	Responsible for managing "message forwarding" to agents that do not have internet access, but can be contacted through other means. Adds the necessary information for messages to be forwarded to target agents.
[27]	Agent ⟶ Direct Forwarding ⟶ ***Networking Unit***	Responsible for implementing and using the actual transport/application layer protocol needed to interact with another agent.

Server – Program Units

The server, which administers agents (clients) and manages testing data at an administrative level, is composed of four primary modules:

- the *framework interface*, which validates user actions, stores operational logs, and provides a web interface for penetration testers;
- the *task management* module, which asynchronously executes tasks initiated from the framework interface;
- the *message generation/fetching* module, which handles communication with agents over arbitrary protocols, and;
- the *payload construction* module, which generates new agent instances that can be installed on a target device.

Each of these represents discrete parts of the server that may be implemented completely independently of each other, allowing for better delegation (from a project management perspective) and enforcing the interfaces and functionality that each module provides. This also makes it possible to optimize or refactor specific modules independently, although we do not anticipate this being in scope for the project.

Framework Interface

The framework interface is the primary means through which a penetration tester can interact with the framework. The web interface allows penetration testers to observe and keep track of controlled devices in a penetration test, issuing commands to the agents installed on these devices and displaying the result of these commands in a user-friendly manner. Besides interacting with agents, the interface also makes it possible for operators to "deconflict" suspicious activity within the organization – in other words, verifying that suspicious activity during an active penetration test is *actually* a result of the penetration test and not another external attacker.

The framework interface allows users to initiate long-running tasks, such as generating new agents (payloads) or messages. However, the framework interface itself does not directly handle the execution of tasks, as this would cause the interface to block on virtually any user action. Instead, these long-running tasks are delegated to the task management module, which returns a "future" that allows the framework interface to report on the status of a currently executing task.

In many ways, the framework interface is much like the mission control room – it is not the rocket that is being launched into space to do work. But without it, the rocket isn't going anywhere to begin with.

[1] Server Backend	
Parent units	Server → Framework Interface → ***Server Backend***
Interacts with or depends on	- Server → Framework Interface → *Server Frontend* - Server → Framework Interface → *Server Database*

	- Server → Task Management → *Task Broker*
Summary	Dynamically generates webpages to reflect the state of the framework; interfaces with the server database to store/access framework information and mediate tasking.
Inputs	- Formatted user inputs (from forms or callbacks) through the frontend - Arbitrary data retrieved from the database - Task information from completed or running tasks from the task broker
Outputs	- Dynamic webpages rendered through the frontend - SQL queries issued to the server database (as interpreted by Django's ORM) - New tasks issued to the task broker
Description	The server backend largely represents the "core" of the framework, keeping track of all of the users, agents, and other data associated with a single penetration test. In addition to dynamically generating the contents of the frontend, it also provides various administrative features – those that do not directly aid in executing a penetration test, but are critical to its success and value to an organization. In particular, this includes: - An authentication system that ensures that every user is held accountable for the actions they take through DeadDrop - A permissions system that ensures only specific users can initiate certain actions or view sensitive information gained through the penetration test - Providing a central source of "truth" for logs and agents, ensuring that operators have a clear understanding of the penetration test at any given moment - Translating requests from users to initiate specific actions into specific tasks that the rest of the framework (abstracted from the user) can understand We intend to implement the backend with the Django web framework for Python. Much of our team has existing experience with both Python and Django, and believe that we can extend its functionality as necessary to server our purposes.

[2] Server Frontend	
Parent units	Server → Framework Interface → ***Server Frontend***
Interacts with or depends on	- Server → Framework Interface → *Server backend*

Summary	Provides users with a web interface to easily interact with the framework, primarily for issuing commands and viewing outputs and logs.
Inputs	- Arbitrary user inputs through forms, buttons, or other input presented by the web interface
Outputs	- Formatted user inputs passed to the the server backend to perform a particular action
Description	The server frontend provides users with a graphical (web) interface to interact with individual parts of the framework. As shown in the *User Interface Design* section, this includes the following: - The ability to issue commands to agents and view their results, possibly as charts, tables, or other formatted visuals - A graphical overview of all devices that the framework either has control over or is aware of - The ability to parse and view logs stored in the backend We intend to implement the frontend with the Svelte component framework. This largely comes down to team preference – one member of our team is familiar with Svelte, while another member is interested in learning a new framework.

[3] Server Database	
Parent units	Server → Framework Interface → ***Server Database***
Interacts with or depends on	- Server → Framework Interface → *Server Backend*
Summary	Holds the data needed to keep track of individual parts of the framework, such as agents, protocols, and logs. Also holds backend-managed data, such as user and group details.
Inputs	- Formatted SQL queries as generated by Django's ORM
Outputs	- Arbitrary, formatted query results passed to Django's ORM
Description	The server database is (primarily) a Django-managed module, thanks to Django's object-relational mapper, which allows the user to write Python code that is translated to database-specific queries without having to worry about the underlying technology used for the database. This comes with many benefits, including: - The ability to use Django's built-in migrations system, which allows us to add to the database models as needed without having to manually modify the table ourselves - The ability to use an arbitrary backend of our choice depending on our scaling and performance needs; we can easily switch between a single-file, local lightweight database and a cloud-hosted solution

	based on observed performance during development. In our case, we plan on using a Postgres database as the backend. Some reasons for doing so include: - Our team's familiarity with Postgres (through the CS 457 class), which allows us to issue raw queries (bypassing Django's ORM) and make low-level changes if needed - The extensive number of tools and libraries available for interacting with Postgres database, including Python's psycopg library and the pgAdmin GUI - The ability to easily migrate a local Postgres database to a remote service if a shared cloud database becomes necessary for any reason

Task Management

The task management module addresses the challenge of managing multiple asynchronous methods that are a mix of on-demand and periodic tasks. The framework interface (more specifically, Django) cannot handle long-running tasks on its own, as Django operations are executed synchronously and block the server. In turn, one long-running operation would make it appear as if the server has become unresponsive; ideally, we would instead inform the user that the task has been started, allow the user to perform other actions, and then (eventually) inform the user of the results of the task.

It is true that we could write our own middleware to spin up and manage subprocesses, using Python's built-in asynchronous libraries to implement "futures" that can be passed back to the framework interface and used in reporting. However, using a dedicated, mature task management module is much more robust and is less likely to suffer from edge cases.

[4] Task Broker	
Parent units	Server → Task Management → ***Task Broker***
Interacts with or depends on	- Server → Framework Interface → *Server Backend* - Server → Task Management → *Task Database* - Server → Message Generation/Fetching → *Message Dispatch* - Server → Payload Generation → *Payload Dispatch*
Summary	Offloads asynchronous tasks from the server interface, reporting results to the server backend and performing on-demand and scheduled periodic tasks.
Inputs	- A Task object, which is any arbitrary request to asynchronously execute code from the server
Outputs	- An instance of AsyncResult, which represents the result of the asynchronous computation (commonly known as "futures" in other

	languages and asynchronous frameworks) - Arbitrary requests to check for messages, send messages, or start the construction of a new agent
Description	Django does not inherently provide any functionality for offloading tasks to the "background" – that is, a separate process or worker – nor does it provide any functionality to execute periodic, scheduled tasks. That said, we still want Django to be responsible for interpreting and starting tasks, as any requests made through the frontend should still be processed and validated by Django. The most common library used to solve this problem is Celery, which is specifically designed as a "task queue" – a mechanism to distribute work. In our case, the client (the Django server) adds a message to the task queue. Then, a broker delivers that message to one or more available workers, which monitor the task queue for new work to perform. There can be an arbitrary number of workers and clients, allowing the system to scale arbitrarily. (Strictly speaking, the "broker" as used by Celery is what we call a "task database"; however, since Celery mediates all tasking, we refer to Celery as the "broker" in our architecture instead.) Celery makes it very easy to add tasking to many Python projects, including specific bindings for Django. Besides its ease of use, it also has many features that make it an ideal solution for our use case: - Celery supports both multiprocess and multithreaded workers, allowing us to scale background tasking however we'd like - Celery supports scheduled tasks, allowing us to execute tasks based on a simple interval - Celery has many built-in guardrails, including task timeouts, resource leak protection, and task rate limiting that prevent tasks from causing resource exhaustion or causing the server to crash But most importantly, it largely abstracts away the issue of dealing with asynchronous code and reporting on its status. When a task is issued, Celery immediately returns a Celery.result.AsyncResult object, which can be arbitrarily checked to see the status of an ongoing task. This allows the server to continue execution – and by extension, allows the user to do other things while a task is running.

[5] Task Database	
Parent units	Server ⟶ Framework Interface ⟶ ***Task Database***
Interacts with or depends on	- Server ⟶ Task Management ⟶ *Task Broker*
Summary	Database/message broker that may be used by the task broker to store the information necessary to keep track of outstanding tasks.

Inputs	- Advanced Message Queuing Protocol (AMQP) messages containing tasking information (but may vary depending on the broker)
Outputs	- Arbitrary (at a high level, the information needed for a worker process or thread to initiate a new task)
Description	The "task database" is a message broker used by Celery to store the information necessary to implement a task queue. Although Celery will largely dictate the usage of the broker, there is still some configuration that must be done by us for Celery to function correctly. For Celery, two main brokers are supported – RabbitMQ and Redis. In our case, we currently plan to use Redis as the broker. In many ways, RabbitMQ is superior to Redis; RabbitMQ was specifically designed as a message broker and is known to scale much more efficiently than Redis. However, Redis is significantly easier to configure and troubleshoot, and is much more suitable for a medium-scale (in terms of usage) project like DeadDrop. From an implementation perspective, our work on the task database involves correctly configuring Redis and observing its performance throughout development.

Message Generation and Fetching

The message generation/fetching module is responsible for sending and receiving messages over arbitrary communication protocols. From here onward, "plaintext messages" refer to the DeadDrop standard message format as described in the *Primary Data Structures* section.

From the server interface's perspective, this module allows it to send and receive messages using just the DeadDrop standard message format. The server interface does not need to know how a YouTube- or Reddit-based protocol operates or how messages are split up and reassembled; it simply needs to know that an agent uses a particular protocol, and the message generation/fetching module will handle the rest.

Internally, this module is composed of two submodules:

- The *message dispatch module*, which is responsible for interpreting requests to communicate with agents. It determines the protocol handler that must be used, as well as the arguments passed to the protocol handler.
- The *protocol handler*, a protocol-specific module (or standalone binary) that implements the construction and reassembly of messages for a single, specific communication protocol. It is composed of three submodules that are described below.

This architecture makes it trivial to add support for new protocols by simply adding new protocol handlers and registering them with the message dispatch module. For example, if we wanted to develop a covert communication protocol in the future that operates over Gmail, we could simply implement a new protocol handler for the agent and server without needing to touch any other aspect of the framework.

It is important to note that **a similar message generation/fetching module exists for the agent as well**. The implementation of a particular protocol handler does not need to be identical between the server and the agent; for example, the server might use a simple Python library, while the agent might use a dedicated binary compiled from C.

[6] Message Dispatch	
Parent units	Server → Message Generation and Fetching → ***Message Dispatch***
Interacts with or depends on	- Server → Task Management → *Task Broker* - Server → Message Generation and Fetching → Protocol Handler → Control Unit
Summary	Module responsible for retrieving the information and protocol handler needed to communicate with an agent over a specific protocol.
Inputs	- *When sending covert messages,* a standardized request to send a plaintext message to a particular agent using a particular protocol (which may include any configuration options supported by that protocol) - *When checking for new covert messages*, a request to check for new messages using a particular protocol at a specific location - *When used in real-time communication*, an arbitrary input (usually a shell command)
Outputs	- *When sending covert messages*, a log message containing the responses from downstream modules (those in the Protocol Handler group) which may represent either a confirmation that the message has been placed in the dead drop *or* any errors - *When checking for new covert messages*, the reassembled plaintext message - *When used in real-time communication,* an arbitrary output (usually the stdout of a shell command)
Description	The message dispatch module exposes two major features to the server, namely the ability to "send" and "receive" messages to and from agents. Because of its modular design, the server interface only has to worry about three things when sending messages – the (name of the) protocol being used, the configuration settings for the protocol, and the message itself. Naturally, when receiving messages, the server only has to worry about the first two items. The message dispatch module is responsible for determining which protocol handler is necessary to communicate with an agent. This may be implemented by storing protocol handler information within the message

	dispatch module itself, or by the server interface passing the desired information within the message request. Additionally, the message dispatch module determines which arguments need to be issued to the protocol handler. In other words, the message dispatch module unpacks a message that the server interface wants to send, formats it into the specific arguments that the corresponding protocol handler wants, and then runs the protocol handler with those arguments. The same is true if the server instead wants to check for a new message over a particular protocol at a specific location.

[7] Control Unit	
Parent units	Server → Message Generation and Fetching → Protocol Handler → ***Control Unit***
Interacts with or depends on	- Server → Message Generation/Fetching → *Message Dispatch* - Server → Message Generation/Fetching → Protocol Handler → *Translation Unit* - Server → Message Generation/Fetching → Protocol Handler → *Networking Unit*
Summary	Component of a protocol handler responsible for either: - scheduling and generating messages to be sent over a protocol, or; - checking for new or expected messages and reassembling components of these messages over a protocol.
Inputs	- *When sending or receiving covert messages*, protocol-specific configuration settings (such as the encryption key, the credentials used to log into an account on an external service, the location of the dead drop, etc.) needed to encode or decode the message on an external service - *When used in real-time communication*, an arbitrary input (usually a shell command)
Outputs	- *When sending covert messages*, a log message indicating that the "dead drop" was successfully placed or that it was not, including any responses from the external service. - *When receiving covert messages,* the reassembled plaintext message. - *When used in real-time communication*, an arbitrary output
Description	The control unit represents the decision-making body of the protocol handler, deciding how to best "deliver" a plaintext message to an agent based on the protocol it implements. After making any high-level decisions, it relies on the translation and networking units to actually form the message that will be uploaded to an external service. For example, if a protocol calls for messages to be no larger than 1 kilobyte in size, it is up to the control unit to determine the best logical way to split up

	the message and place it at locations that the agent will actually discover. It must then call the translation unit for each piece accordingly, then ask the networking unit to place each piece at the desired locations. Its role is similar in the case of receiving messages; given configuration details, the control unit must determine where to check for new messages, asking the networking unit to reach out to the external service accordingly. As the control unit receives information from the networking unit, it can pass this back to the translation unit to decode the messages and determine if message components are missing.

[8] Translation Unit	
Parent units	Server → Message Generation and Fetching → Protocol Handler → ***Translation Unit***
Interacts with or depends on	- Server → Message Generation/Fetching → Protocol Handler → *Control Unit*
Summary	Responsible for translating plaintext messages into one or more binary streams of data to be sent to a particular external service, or vice versa.
Inputs	- A single plaintext message (or binary stream)
Outputs	- One or more encoded messages (possibly files or in-memory bytes of data) to be placed at various locations on the target
Description	The translation unit is responsible for encoding plaintext messages (which may contain protocol-specific metadata beyond that specified in the *DeadDrop standard message format*) into binary streams or files that will be uploaded to the external service. Where necessary, the translation unit also includes any dependencies necessary to perform the encoding/decoding process. For example, if the protocol intends to upload a database to YouTube by encoding it as a video, the translation unit is likely to include ffmpeg and other libraries to convert the file into individual frames. The resulting video (or path to the video) is then passed back to the control unit.

[9] Networking Unit	
Parent units	Server → Message Generation and Fetching → Protocol Handler → ***Networking Unit***
Interacts with or depends on	- Server → Message Generation/Fetching → Protocol Handler → *Control Unit* - External Service *or* Real-Time Medium

Summary	Responsible for uploading translated streams of data to one or more target locations (which may involve using an API or specific transport layer protocol), or for retrieving and streams of data from one or more targets.
Inputs	- *When sending messages*, a single encoded message along with the information needed to place the message at the target external service - *When receiving messages*, the information needed to request data from the target external service
Outputs	- *When sending messages,* the response from the service as a result of "placing" the dead drop - *When receiving messages*, the response from the service containing the information (which is expected to contain an encoded message or part of a message)
Description	The networking unit is responsible for uploading encoded messages to the desired service (and downloading encoded messages from the same service) based on the protocol it supports. Its main role is to abstract away the API-specific calls and formatting that might be necessary to actually interact with the external service that the protocol uses. For example, if a service uses Base64-encoded Reddit comments to communicate: - the control unit would determine which (links to) Reddit threads to use as the dead drop location, as well as how many comments to use - the translation unit would convert the raw message pieces into Base64-encoded strings - the networking unit would actually interact with Reddit's API to place the messages at the desired locations The same is true in the case the networking unit must retrieve data from an external API. In all cases, it returns the response from the external service back to the control unit, which may be an error or confirmation that the data was successfully retrieved or placed.

[14] External Service or Real-Time Medium	
Parent units	(none) → **External Service** or **Real-Time Medium**
Interacts with or depends on	- Server → Message Generation and Fetching → Protocol Handler → *Networking Unit* - Agent → Message Generation and Fetching → Protocol Handler → *Networking Unit*
Summary	This represents either: - The external service used to conceal communication between servers

	and agents - A "standard" protocol and medium used for real-time communication, like a direct TCP connection
Inputs	- Varies greatly by service (but usually a file, API response, HTTP response, or sustained connection)
Outputs	- Varies greatly by service (but usually a file, API response, HTTP response, or sustained connection)
Description	The external service represents an arbitrary service used as a dead drop location between the agent and server. Again, the motivation behind DeadDrop is to discover a way for agents and servers to communicate without ever having to actually connect to each other. This could be Wikipedia, YouTube, Reddit, Instagram, or any other major website that is likely to be trusted by an organization's network. In practice, it is likely that the external service would not be under our control. However, as part of this project, we do intend to stand up "mock" services in an effort to avoid breaking the law (as an example, it's very likely that this use of YouTube is outside of their terms of service). This is likely to simply involve us standing up our own instance of an open-source video streaming platform, or hosting our own instance of MediaWiki (which powers Wikipedia). We do also note that a "real-time medium" may also be used here as well. While the focus of DeadDrop is on these indirect forms of communication, it should always be possible for the server to communicate directly with its agents, even if such communication is likely to be caught or blocked. A "real-time medium" could simply be a direct SSH connection over the open internet.

Payload Construction

The payload construction module is responsible for creating and registering new agents. It accepts requests to generate new agents from the server interface, which contains any user-specific configuration information as well as the desired agent itself.

Although the architecture of an agent is described in the following section, an agent is generally composed of a centralized command module, one or more messaging units, and one or more command execution units. Each of these must be bundled into a single "payload" that can be delivered to a remote host for installation, thus allowing the server to interact with the newly created agent remotely.

Constructing payloads comes with two major challenges:
- Payloads are platform-specific, and there is no guarantee that a target device will use the same platform as the one currently hosting the server.

- Payloads may not assume that any dependencies are available on the host, except for those that are guaranteed by the platform. That is to say, an agent must come bundled with every single dependency that it requires to function properly, which may encompass many different binaries and libraries – some of which may require other libraries to properly be "bundled."

In turn, payload construction is orchestrated through Docker containers, which contain the build environment necessary for the agent to be built, configured, and bundled. Then, regardless of the operating system on which the server is running, it becomes possible to generate agents for arbitrary platforms. Additionally, the underlying Docker image can be distributed to arbitrary users with the expectation that it will come with all of the dependencies needed for the agent to build.

[10] Payload Dispatch	
Parent units	Server → Payload Generation → ***Payload Dispatch***
Interacts with or depends on	- Server → Task Management → *Task Broker* - Server → Payload Generation → *Agent Definitions* - Server → Payload Generation → *Docker Compose*
Summary	Responsible for interpreting requests to build agents, retrieving data from agent definitions to generate the commands or files necessary for the agent's Docker container to build the agent with the user's configuration settings.
Inputs	- A formatted request to create a particular agent, including any configuration settings that are specific to that agent - The definitions of the specified agent as obtained from the *Agent Definitions* module
Outputs	- The status or build logs associated with the construction of that agent - The (path to the copied) compiled and bundled agent itself
Description	The payload dispatch module is the primary "jumping point" when a build request for a new agent is received. When it receives a request for a particular agent to be built, it looks up the corresponding data for that agent in the *Agent Definitions* and determines what must be done to prepare for that agent to be built. In most cases, this consists of interpreting the agent request into specific commands or arguments passed into the Docker container responsible for building and bundling the agent. While this can be more complex, we believe that this workflow is likely to be most common within our project. This is necessary because the server interface is only concerned with providing "plaintext" options that the operator specifies – it does not know about any of the specific arguments that the Docker container accepts. If a user specifies that they want a particular encryption key to be used in

	communications, it is up to the dispatch module to figure out how to get that into the container so it can be added to the agent's static configuration.

[11] Agent Definitions	
Parent units	Server → Framework Interface → ***Agent Definitions***
Interacts with or depends on	- Server → Payload Generation → *Payload Dispatch*
Summary	Static set of files containing the information necessary to build an agent.
Inputs	- None (after initial configuration and manual registration by the user)
Outputs	- Structured Python classes and other configuration files containing the information necessary to build an agent
Description	The agent definitions are a standardized set of instructions for building and defining agents. They contain the metadata and other build information necessary for locating build dependencies (if any), the associated agent's Docker container, and other agent-specific information. As of right now, we do not anticipate this module to have much functionality (if any) beyond accepting the name of a particular agent and returning its associated configuration files. This could realistically change in the future, such as if the "definitions" are used as high-level templates for generating more concrete definitions that are then passed to the payload dispatch module.

[12] Docker Compose	
Parent units	Server → Payload Generation → ***Docker Compose***
Interacts with or depends on	- Server → Payload Generation → *Payload Dispatch* - Server → Payload Generation → *Docker Container*
Summary	Responsible for managing and instantiating the container (or containers) needed to generate the payloads for an agent.
Inputs	- compose.yaml (which contains information about all of the containers associated with a particular application) - The individual Docker containers associated with a single agent (indirectly, through the YAML configuration file)
Outputs	- None (it instantiates individual containers from images, but does not pass anything back to the payload dispatch module)
Description	Docker Compose is used in defining and quickly instantiating applications

	that depend on multiple Docker containers (that interact with each other). This is true of any application that leverages multiple Docker containers. For DeadDrop, we anticipate this to be used for agents that require multiple containers when building their modules (such as when the agent itself is built independently of its command binaries). Docker Compose is specifically designed to make it easier to move information and create connections between multiple containers, as well as defining these relationships in a portable way. Although we currently have no plans to do so, we also believe that Docker Compose makes sense in testing agents against a simulated environment. Because Docker Compose has full networking between separate containers, it may be possible to simulate actions between multiple devices and observe how they interact. This is particularly relevant for the Direct Forwarding module within agents.

[13] Docker Container	
Parent units	Server → Payload Generation → ***Docker Container***
Interacts with or depends on	- Server → Payload Generation → *Docker Compose*
Summary	Contains the build environment (and any dependencies) that the agent will be constructed in; also generates the final output that is to be installed onto a target device.
Inputs	- Varies (typically some set of command-line options) - Dockerfile
Outputs	- Log results (as dictated by the container itself) - A bundled agent, including the agent itself, its dependencies, and the binaries used to execute commands
Description	Docker containers are core to DeadDrop's functionality – and, more generally, any modern C2 framework. Docker containers make it possible to build agents for arbitrary platforms from a single host machine, eliminating the need to pre-compile agents or spin up platform-specific servers. An individual Docker container is responsible for building an agent according to the configuration that it receives. The exact process can vary greatly, but can include: - Downloading specific versions of dependencies - Compiling binaries responsible for executing agent commands - Compiling the agent itself, if it must be compiled - Bundling the agent with its libraries, such as with PyInstaller - Applying obfuscation techniques, such as encrypting the agent and including a self-decryption routine

	An agent may use one or more Docker containers for its build process. Strictly speaking, it is worth noting that Docker containers are not *required* for the framework to function properly; as long as an agent (regardless of how it was created) adheres to the server-client architecture, it will still be accepted by the framework. However, this is largely only viable in a development environment, as Docker containers are much more portable and are arguably easier to maintain.

Agent – Program Units

The agent is a piece of software installed on a target device that remotely executes commands on behalf of the server. The agent may range in complexity from a simple Python or Powershell script to a feature-filled application capable of dumping credentials from memory, stealing session cookies, and opening a reverse shell to the machine. Agents are very abstract in form, and may be implemented in arbitrary languages for arbitrary systems; the only underlying requirement is that they respond to standardized DeadDrop messages.

However, for an agent to be useful in the context of a penetration test, we expect each agent to have certain program units. Generally speaking, its components can be grouped into four distinct units:

- Ungrouped units, which represent various standalone pieces of the agent
- Message generation/fetching, which handles communication with the server over a covert communication protocol
- Command execution, which executes and manages specific “tasks” by invoking dedicated (independent) binaries bundled with the agent
- Direct forwarding, which handles message forwarding to other agents in the network that may not have direct internet access

As with the server, these modules are sufficiently independent that they can be developed in parallel and interchanged, so long as they implement the interfaces described in this section. Furthermore, this means that the functionality of an agent can be arbitrarily extended by simply adding more protocol handlers or command binaries, which can greatly improve its usability.

Ungrouped Units

The ungrouped units contribute to the functionality of the agent at a high level, but do not provide any inherent functionality that would be desired from an operator interacting with the agent. Instead, these provide and manage the information necessary for the agent to be useful in a penetration test, including logging and configuration.

[15] Control Unit	
Parent units	Agent → ***Control Unit***
Interacts with or depends on	- Agent → *Local Configuration* - Agent → *Logging Module* - Agent → *Task Broker*
Summary	Manages all decisionmaking for the agent; interprets and acts on messages received from the C2 server.
Inputs	- Messages, command outputs, and forwarded messages received from the task broker - Locally stored configuration data

Outputs	- New tasks issued to task broker (which may be messages to the C2 server, forwarded messages, or commands to be executed) - Unformatted log messages passed to logging module
Description	The agent control unit serves as the decision-making body for anything that the agent does. Centralized decision-making is good for several reasons: - Logical flaws in agent operations can always be traced back to this single module, while flaws in command execution can be attributed to the binaries instead - Every single action *must* pass through the control unit, which implicitly forces every single action to be logged - How the agent reacts to messages from the server can easily be changed without affecting any other component Again, the control unit is much like a command center; it doesn't do much from a functional perspective, but is essential for the agent to be useful in the context of a penetration test.

[16] Local Configuration	
Parent units	Agent → ***Local Configuration***
Interacts with or depends on	- Agent → *Control Unit*
Summary	Static files dictating various configuration settings for the agent, such as the agent's ID, any encryption keys in use, logging settings, and so on.
Inputs	- None (after initial setup by the user and the Docker build container)
Outputs	- Arbitrary, agent-specific configuration settings
Description	The local configuration of the agent may be stored inline (as part of a single, compiled agent) or as separate configuration files that can be accessed by the agent control unit as necessary. These may include encryption keys, logging settings, the agent's ID, and other information that may be duplicated server-side. Much like the agent definitions, we do not anticipate this module to have much functionality, if at all; however, breaking it off as a separate module (as opposed to baking it in with the control unit) makes it possible to reconfigure an agent after it has already been built, even if the agent control unit malfunctions.

[17] Logging Module	
Parent units	Agent → ***Logging Module***

Interacts with or depends on	- Agent → *Control Unit* - Agent → *Log Storage*
Summary	Centralized logging module that writes logs to external storage based on its configuration settings.
Inputs	- Unformatted log messages issued by the control unit
Outputs	- Arbitrary, well-formatted log messages sent to storage (which vary in form; for example, this *could* be a SQL query)
Description	The logging module is an arbitrary adapter that accepts unformatted log messages and stores them in external storage. For an agent made with Python, this might simply be the built-in logging module, which supports multi-threaded applications and can write to arbitrary log formats. This is independent of the agent control unit, which makes it possible to change the log format or logging configuration without needing to change the actual messages issued by the control unit. Where useful, the logging module should also support the retrieval of logs, possibly with filters applied. In this case, the logging module may need to accept command request messages, query the log storage accordingly, and then generate a command_request message.

[18] Log Storage	
Parent units	Agent → ***Log Storage***
Interacts with or depends on	- Agent → *Logging Module*
Summary	Arbitrary form of storage used to hold logs generated by any part of the agent.
Inputs	- Arbitrary, well-formatted log messages
Outputs	- Groups of log messages (if the module supports querying); else, the entire log history of the agent
Description	Much like the local configuration module, this is a largely static service whose sole purpose is to store logs generated by the agent. This may range in complexity from a simple text file to a fully-fledged database with daemons. Again, if the functionality of the logging system ever needs to be extended, breaking this off as a separate module makes it possible to "upgrade" the log format without needing to change the rest of the agent. For example, if we decide that the log storage should be a proper database instead of just a simple text file, it is sufficient to change the logging module and log storage.

[19] Task Broker	
Parent units	Agent → ***Task Broker***
Interacts with or depends on	- Agent → *Control Unit* - Agent → Message Generation and Fetching → *Message Dispatch* - Agent → Command Execution → *Command Dispatch* - Agent → Direct Forwarding → *Forwarding Unit*
Summary	Responsible for executing and managing tasks and returning their results to the agent control unit.
Inputs	- Depends on implementation; generally, a request for a task to execute
Outputs	- Depends on implementation; could be the results of tasks or "futures" for tasks
Description	The purpose of the agent task broker is very similar to that of the task broker described for the server. Because agents may be designed for arbitrary platforms with arbitrary libraries, there is no specific description of the agent task broker's technologies. It is expected that agents are able to handle both on-demand and periodic tasking and can execute these tasks in parallel. However, there are no requirements that this is the case; in turn, the task broker may simply be responsible for formatting requests and generating the correct calls to the messaging, command, or forwarding units.

Message Generation and Fetching

The message generation/fetching module is responsible for communication with the server. It is identical in purpose and architecture to the server module of the same name, although it does not necessarily need to be implemented in the same manner. (As a result, we treat it as a separate module in our client-server architecture, as it is *very likely* to actually be different between the agent and server implementations of the same protocol).

The primary difference between the server and the agent implementations of the messaging module is that the agent is likely to send a greater number of messages at a higher volume, assuming that the covert communication protocol is used as the primary means of communication (as opposed to a more direct but less covert protocol, such as SSH). Not only does the agent have to send command response messages of arbitrary size, but it must also regularly provide a "heartbeat" message, provide log messages, and may need to exfiltrate files of large size or volume. The server only needs to issue command requests.

In turn, the agent should implement a messaging module that is optimized for sending messages, possibly of large size; the server has no need for this, as the only “large” file that is required to be sent is the initial agent. The agent itself must be manually planted by the penetration tester via direct means and is not brokered by the convert communication protocols.

[20] Message Dispatch	
Parent units	Agent → Message Generation/Fetching → ***Message Dispatch***
Interacts with or depends on	- Agent → *Task Broker* - Agent → Message Generation and Fetching → Protocol Handler → *Control Unit*
Summary	Agent module identical in purpose to the *Message Dispatch* module of the server.
Inputs	- *When sending covert messages,* a standardized request to send a plaintext message to a particular agent using a particular protocol (which may include any configuration options supported by that protocol) - *When checking for new covert messages*, a request to check for new messages using a particular protocol at a specific location - *When used in real-time communication*, an arbitrary input (usually the stdout of a shell command)
Outputs	- *When sending covert messages*, a log message containing the responses from downstream modules (those in the Protocol Handler group) which may represent either a confirmation that the message has been placed in the dead drop *or* any errors - *When checking for new covert messages*, the reassembled plaintext message - *When used in real-time communication,* an arbitrary output (usually a shell command)
Description	See [6] Message Dispatch for more information.

[21] Control Unit	
Parent units	Agent → Message Generation/Fetching → Protocol Handler → ***Control Unit***
Interacts with or depends on	- Agent → Message Generation/Fetching → *Message Dispatch* - Agent → Message Generation and Fetching → Protocol Handler → *Translation Unit* - Agent → Message Generation and Fetching → Protocol Handler → *Networking Unit*
Summary	Agent module identical in purpose to the *Control Unit* module of the server.

Inputs	- *When sending or receiving covert messages*, protocol-specific configuration settings (such as the encryption key, the credentials used to log into an account on an external service, the location of the dead drop, etc.) needed to encode or decode the message on an external service - *When used in real-time communication*, an arbitrary input
Outputs	- *When sending covert messages*, a log message indicating that the "dead drop" was successfully placed or that it was not, including any responses from the external service. - *When receiving covert messages,* the reassembled plaintext message. - *When used in real-time communication*, an arbitrary output
Description	See [7] Control Unit for more information.

[22] Translation Unit	
Parent units	Agent → Message Generation/Fetching → Protocol Handler → ***Translation Unit***
Interacts with or depends on	- Agent → Message Generation/Fetching → Protocol Handler → *Control Unit*
Summary	Agent module identical in purpose to the *Translation Unit* module of the server.
Inputs	- A single plaintext message
Outputs	- One or more encoded messages (possibly files or in-memory bytes of data) to be placed at various locations on the target
Description	See [8] Translation Unit for more information.

[23] Networking Unit	
Parent units	Agent → Message Generation/Fetching → Protocol Handler → ***Networking Unit***
Interacts with or depends on	- Agent → Message Generation/Fetching → Protocol Handler → *Control Unit* - External Service *or* Real-Time Medium
Summary	Agent module identical in purpose to the *Networking Unit* module of the server.
Inputs	- *When sending messages*, a single encoded message along with the information needed to place the message at the target external service

	- *When receiving messages*, the information needed to request data from the target external service
Outputs	- *When sending messages,* the response from the service as a result of "placing" the dead drop - *When receiving messages*, the response from the service containing the information (which is expected to contain an encoded message or part of a message)
Description	See [23] Networking Unit for more information.

Command Execution

The command execution module provides all of the complex functionality that an agent offers. Broadly speaking, "commands" are simple requests that lead to specific actions being performed on the target device. Typical commands in other C2 frameworks include:

- The ability to enumerate the entire filesystem – that is, list all of the files stored on the device, including their filename and location
- The ability to take a dump of volatile memory, which may contain sensitive information or user credentials
- The ability to open a reverse shell, whereby the target device initiates an outgoing connection to the attacker that provides a standard shell that the attacker can execute commands with

In DeadDrop, each of these commands are implemented as discrete binaries, though a binary may be used to support one or more commands. The command dispatch module is responsible for interpreting requests to execute one of these commands, generating the command-line arguments necessary to execute the command and returning the command's result back to the agent control unit.

[24] Command Dispatch	
Parent units	Agent ⟶ Command Execution ⟶ ***Command Dispatch***
Interacts with or depends on	- Agent ⟶ *Task Broker* - Agent ⟶ Command Execution ⟶ *Command Binaries*
Summary	Responsible for interpreting requests for specific commands to be executed, generating the arguments and calling the associated binary for that command.
Inputs	- An arbitrary request to execute a supported command with specific arguments
Outputs	- (to the command binaries) A specific command-line invocation of the

	binary specifically designed to execute that command - (to the task broker) The output of the command as determined at this level, which may be a newly-generated file, the standard output of the command, or some other side effect
Description	Much like other dispatch modules, the command dispatch module is responsible for interpreting requests to execute a command into specific calls to stored binaries. It must then interpret the raw results generated by these binaries (if any), and generate a formatted response back to the agent control unit with the results of executing the commands. This may include additional tasking that is outside the scope of the commands themselves; for example, if the user requests that a file be generated, the command dispatch module might be responsible for asserting that the file was actually created by the associated binary. At a high level, the command dispatch module must: - Interpret a request for a command to be executed, along with any additional data included with the request (which is likely a command_request message) - Search for the binary (or binaries) that are responsible for carrying out that command - Issue calls to those binaries based on the request data, adding arguments as needed - Retrieve the outputs (whether written to stderr, stdout, or external storage) and verify that the outputs follow the expected format - Bundle the outputs into a single command_response message and pass it back to the agent control unit (via the task broker)

[25] Command Binaries	
Parent units	Agent ⟶ Command Execution ⟶ ***Command Binaries***
Interacts with or depends on	- Agent ⟶ Command Execution ⟶ *Command Dispatch*
Summary	Standalone binaries that accept command-line arguments and are designed to achieve a single task on a device.
Inputs	- Command-line arguments
Outputs	- Varies greatly; may be new files placed on the device's filesystem, standard output, and so on
Description	The command binaries are responsible for implementing individual commands. These can range greatly in functionality, complexity, and intensity; a simple command might just generate a directory listing, while a more complex command might involve Mimikatz, which searches for credentials in volatile memory.

	It is up to the command dispatch module to determine where these binaries are, how to correctly call them, and how to interpret their results or side effects.

Direct Forwarding

The direct forwarding module is responsible for forwarding messages to agents that do not have an internet connection and therefore cannot use a convert communication protocol to communicate. This is typical of "layered" networks in which certain devices are isolated from the internet, but (naturally) must be connected to other devices in the network to provide an internal service.

However, because these devices often contain data that would be of interest to an attacker, it is necessary to provide some way to control these devices in a way that does not require a direct internet connection. This module implements the most typical method of defeating a layered defense, which is to use "outer" devices as a way to communicate with "inner" devices by forwarding messages through arbitrary protocols. This is achieved through two submodules:

- The forwarding unit, which converts plaintext messages into encoded messages and determines the networking unit configurations necessary to start communication with the receiving device, and;
- The networking unit, which actually implements the API or protocol necessary for communication with the target device.

Naturally, this protocol will vary greatly from platform to platform. Perhaps a simple TCP socket may be sufficient; in other cases, it might be necessary to leverage OS-specific or organization-specific features. As with all other independent modules, a networking unit can be developed independently of the rest of the agent and interchanged depending on the penetration test's needs.

[26] Forwarding Unit	
Parent units	Agent → Direct Forwarding → ***Forwarding Unit***
Interacts with or depends on	- Agent → *Task Broker* - Agent → Direct Forwarding → *Networking Unit*
Summary	Responsible for managing "message forwarding" to agents that do not have internet access, but can be contacted through other means. Adds the necessary information for messages to be forwarded to target agents.
Inputs	- A plaintext message adhering to the standard message format
Outputs	- Configuration information necessary for the networking unit to connect to the agent to forward messages to

	- The plaintext message with any additional metadata attached as needed
Description	Like other dispatch units, the forwarding unit is responsible for interpreting requests to forward messages to other clients and leveraging the networking unit to do so. In most cases, this consists of calling the relevant networking unit used for forwarding and passing in formatted data. For example, if a request to forward data to a particular agent via netcat is requested, the forwarding unit likely should: - Verify (via the networking unit) that the target agent is reachable - Encode the message as Base64 - Pass the base64 message into the networking unit, which will handle the actual process of connecting to the agent and sending the message In some cases, it may make sense for the forwarding unit to be a fully daemonized application, particularly if the associated protocol for forwarding requires that the client is always listening for new connections.

[27] Networking Unit	
Parent units	Agent → Direct Forwarding → ***Networking Unit***
Interacts with or depends on	- Agent → Direct Forwarding → *Forwarding Unit* - (Another agent instance, if present)
Summary	Responsible for implementing and using the actual transport/application layer protocol needed to interact with another agent.
Inputs	- Configuration information and message data from the forwarding unit
Outputs	- A plaintext message sent to or received from another agent
Description	The networking unit is an arbitrary module that implements the specific API or protocol needed to connect to another agent. This is likely a subset of the protocols supported by the networking unit in the messaging module, likely including raw TCP connections, UDP messaging, and SSH connections. The networking unit should abstract away the dependencies and network-layer information needed to pass messages; that is to say, the forwarding unit should not concern itself with the handshaking process of a TCP connection.

Primary Data Structures

Database Tables

The database tables used in this project (excluding internal databases used by dependencies) will be administered through Django's ORM, which provides a database-agnostic API to interact with the underlying data as Python objects. Individual columns of the underlying database are represented as "fields", which use Django-specified types that are converted to SQL queries for the underlying database management system. As a result, the definitions for the database tables are presented here using Django model definitions, using the built-in field types. (That said, it is still physically possible to execute raw SQL queries through Django as needed. We anticipate using Postgres as our database backend for this reason, as it is the DBMS we are most familiar with in the event manual intervention is needed.)

Note that most foreign keys have the "models.PROTECT" restriction to help prevent accidental deletion of operational data; if an agent is deleted, the logs associated with it should not be deleted. In case the bolding in table examples isn't clear, **the "id" field is always the primary key of each model.**

Each of the following model definitions below is what we *expect* to need as the backend is gradually developed. Certain field details (such as the max length of a character field) have not been finalized, though the required information in each model is unlikely to change. Unless otherwise stated, assume that the following imports in models.py have been performed:

```
import datetime
import uuid

from django.contrib.auth.models import User, Group
from django.db import models
```

The **User** and **Group** models are administered by Django. They are used in user authentication and authorization, allowing users to take specific actions against the framework. To the best of our knowledge, there is no need to extend the User and Group models beyond what is provided by Django. Their available fields are summarily defined below, though it is possible that Django internally uses different fields:

```
class User:
      # Administered by Django.
      username: str
      first_name: str
      last_name: str
      email: str
      password: str
      groups: models.ManyToManyField # to Group
      user_permissions: models.ManyToManyField # to Permission
      is_staff: bool
      is_active: bool
      is_superuser: bool
      last_login: datetime.datetime
      date_joined: datetime.datetime

class Group:
      # Administered by Django.
      name: str
      permissions: models.ManyToManyField # to Permission
```

No table provided, as Django manages this table.

The **Agent** model represents an abstract definition of an agent (not an instance of an agent; this is internally referred to as an “endpoint”). It contains the information needed to uniquely identify and build a particular agent type.

```
class Agent(models.Model):
      id = models.AutoField(primary_key=True)
      # Human-readable name
      name = models.CharField(max_length=32)
      # Local path to agent definition root
      definition_path = models.FilePathField()
```

Agent example:

id	name	definition_path
1	example_agent	/agents/example_agent/

The **Protocol** model represents an abstract definition of a protocol (again, not a specific protocol handler). It contains the information needed to uniquely identify a particular protocol and determine its associated serverside protocol handler.

```
class Protocol(models.Model):
      id = models.AutoField(primary_key=True)
      # Human-readable name
      name = models.CharField(max_length=32)
      # Local path to protocol handler binary (may make sense as a
FileField?)
      handler_path = models.FilePathField()
```

Protocol example:

id	name	definition_path
1	example_protocol	/protocols/example_protocol/server_handler

---

The **Endpoint** model represents a specific instance of an agent. It contains various common high-level details about the agent, supporting agent-specific configuration details that may be used in communication or issuing commands. It also supports the notion of “virtual” agents, which are devices that have been discovered (through arbitrary means) but are not directly interactable through the framework.

```
class Endpoint(models.Model):
      id = models.UUIDField(
            primary_key=True,
            default=uuid.uuid4,
            editable=False
      )
      # Human-readable name, device hostname, and remote address
      name = models.CharField(max_length=32)
      hostname = models.CharField(max_length=100)
      address = models.CharField(max_length=32)
      # Does this device actually have an agent installed?
      is_virtual = models.BooleanField()
      # What protocol and agent does this endpoint use, if any?
      # Also, block endpoints from destruction if an agent or protocol is
deleted
      agent = models.ForeignKey(Agent, on_delete=models.PROTECT)
      protocol = models.ForeignKey(Protocol, on_delete=models.PROTECT)
      # Encryption key used in communications, if any
      encryption_key = models.CharField(
            max_length=64,
            blank=True,
            null=True
        )
        # HMAC key used in communications, if any
        hmac_key = models.CharField(
            max_length=64,
            blank=True,
            null=True
        )
        # Additional JSON configuration object
        agent_cfg = models.JSONField(
            blank=True,
            null=True
        )
        # What other endpoints does this endpoint have direct access to?
        connections = models.ManyToManyField("self")
```

Endpoint example:

id	name	hostname	address	is_virtual	agent	protocol	encryption_key	hmac_key	agent_cfg	connections
(UUID)	test	PC1	192.168.0.1	false	1	1	null	null	{“a”: 1}	null

---

The **Task** model represents a single task, whether issued on demand or periodically. These are closely tied to tasks within Celery’s task queue, and contain information needed for debugging and reporting. Similarly, the **TaskResult** model reflects any information received as a result of the task’s completion.

```
class Task(models.Model):
        id = models.AutoField(primary_key=True)
        # What user and endpoint, if any, is this task associated with?
        # Theoretically, every task was caused by *someone*, even if it's
        # a periodic task
        user = models.ForeignKey(
            User,
            on_delete=models.PROTECT
        )
        endpoint = models.ForeignKey(
            Endpoint,
            on_delete=models.PROTECT
```

```
      )
      # When was this task started/finished?
      start_time = models.DateTimeField()
      end_time = models.DateTimeField()
      # Is the task still in progress? What was the task? (How do we tie
this to Celery?)
      in_progress = models.BoolField()
      data = models.BinaryField()

class TaskResult(models.Model):
      id = models.AutoField(primary_key=True)
      # What task is this result associated with?
      task = models.ForeignKey(Task, on_delete=models.PROTECT)
      # Arbitrary data field; may contain files, a raw message, etc.
      # Is there a more suitable field?
      data = models.BinaryField()
```

Task example:

id	user	endpoint	start_time	end_time	in_progress	data
1	2	3	1700370224	null	true	b”data”

TaskResult example:

id	task	data
1	2	b”{“status”: “OK”}

In addition to task results, the **StoredCredential** and **StoredFile** models describe credentials and files transferred to the server from an agent via arbitrary means. Because these may contain sensitive user information (in a real penetration test) that is subject to legal requirements, they are represented as independent classes. Besides its use in reporting, this also makes it possible to restrict users from viewing this information.

```
class StoredCredential(models.Model):
       id = models.AutoField(primary_key=True)
       # Task responsible for creating this credential entry, if any
       task = models.ForeignKey(
             Task,
             on_delete=models.PROTECT,
             blank=True,
             null=True
       )
       # Enum field, probably something like "session_token", "userpw", etc
       # Again, it may make sense to create an arbitrary tag model for this
       credential_type = models.CharField()
       # The actual value of the session token, username/password combo,
etc -
       # in case the credential is composed of multiple things, it should
have a
       # clear delimiter based on credential_type
       credential_value = models.CharField(max_length=100)
       # When this credential becomes invalid
       expiry = models.DateTimeField(blank=True, null=True)

class StoredFile(models.Model):
      id = models.AutoField(primary_key=True)
      # Task responsible for creating this credential entry, if any
      task = models.ForeignKey(
             Task,
             on_delete=models.PROTECT,
             blank=True,
             null=True
      )
      # Path to file; location to be determined
      file = models.FileField()
```

StoredCredential example:

id	task	credential_type	credential_value	expiry
1	2	userpw	admin:password	null

StoredFile example:

id	task	file
1	2	/storage/tasks/2/etc_passwd

Finally, the **Log** model represents a standardized, arbitrary log message. Although various components may generate their own logs, the Log model contains logs relevant *in any way* to the penetration test. Modules should be built with the philosophy that it is better to send logs to the server than not, but it may not always make sense to send every single log.

```
class Log(models.Model):
      id = models.AutoField(primary_key=True)
      # Is the log tied to a user?
      user = models.ForeignKey(
            User,
            blank=True,
            null=True,
            on_delete=models.PROTECT
      )
      # Is the log tied to a specific task?
      task = models.ForeignKey(
            Task,
            blank=True,
            null=True,
            on_delete=models.PROTECT
      )
      # Is the log tied to a single task result object?
      task_result = models.ForeignKey(
            TaskResult,
            blank=True,
            null=True,
            on_delete=models.PROTECT
      )
      # Enumerable field; may make sense to create a "Tag" model and allow
      # it to be associated with arbitrary models, could help with
organization
      category = models.CharField()
      # Enum field, levels are likely to be 0 - 5
      level = models.CharField()
      # Time of log (ideally, relative to server time; these should be
timezone aware)
      timestamp = models.DateTimeField()
      # Actual log message
      data = models.TextField()
```

Log example:

id	user	task	task_result	category	level	timestamp	data
1	3	null	null	user_created	5	1700370224	b"this is a test"

DeadDrop Standard Message Format

The DeadDrop standard message format is an internal plaintext JSON document that all agents and the server must support. They are passed into the message generation/fetching module of their respective implementations, where the message is then converted into one or more pieces that are suitable for the external service being used as a dead drop.

All messages adhere to the following top-level format shown below:

```
{
      message_type: "string",
      initiated_by: "string",
      source_id: "string",
      timestamp: 000000000,
      details: {
            ...
      },
      forwarding: {
            ...
      }
}
```

The “message_field” field is one of five message types:

- log_message, which represents an arbitrary message from an agent or server component that should be stored in the server’s log database
- command_request, a server-only message that represents a request for a command to be executed on a specific agent
- command_response, an agent-only message that represents the response to a specific command_request message
- file_upload, an agent-only message that represents (part of) the contents of a single file being sent to the server
- overhead, an arbitrary message used for non-functional tasks such as heartbeats and initial registration (i.e. when an agent communicates with the server for the first time)

The “initiated_by” field contains the ID of the user who is directly associated with the message if the action was user-initiated (whether on-demand or periodic). If it is system-initiated, the field is empty.

The “source_id” field contains the ID of the agent directly associated with the message. If the server is the source of the message, the field is empty. If the message is being forwarded, this contains the ID of the *original* agent that constructed this message.

The “details” field is a variable-content field containing a JSON dictionary of arbitrary keys to values. Its structure depends on the specific type defined in “message_field”, and is described below.

Finally, the “forwarding” field contains forwarding information. This may contain either the desired agents through which the message should be forwarded (if a path is known ahead of time), configuration settings for forwarding, or the agents that have previously forwarded the message. This is allowed to be an empty dictionary or null.

The structure of the “details” field varies depending on the message type. Although we have not decided on the exact format of these fields, we expect the following data to be present in each type:

- For “log_message”:
    - The “level” of the message (debug, info, warning, error)
    - The message itself
    - Additional details, if available (such as tracebacks)
- For “command_request”:
    - The task ID associated with this command
    - The name of the command being executed on the agent
    - A nested dictionary of keys to values, representing each of the configuration options available for the command (typically command-line arguments interpreted by the agent’s command dispatch module)
- For “command_response”:
    - The task ID of the “command_request” message associated with this response
    - The stdout of the binary used to fulfill the command
    - The stderr of the binary used to fulfill the command
    - Any additional command-specific information
- For “file_upload”:
    - The task ID of the “command_request” message associated with this file
    - The full filepath of the file being uploaded
    - The “piece” number of the file that this message contains (note that this is the only message type for which its contents may be broken up)
    - The total number of “pieces” associated with this file
- For “overhead”:
    - Likely nothing, as the message is only used for initial check-in and status checks

Agent Definition

The final major data structure used by DeadDrop is the agent definition, a standardized format that contains the necessary information used to assemble an agent into a payload. It exposes various metadata about the agent, including the commands that the agent supports, configuration options that can be passed into the agent's Docker container, and other overhead such as the agent's creator and version number.

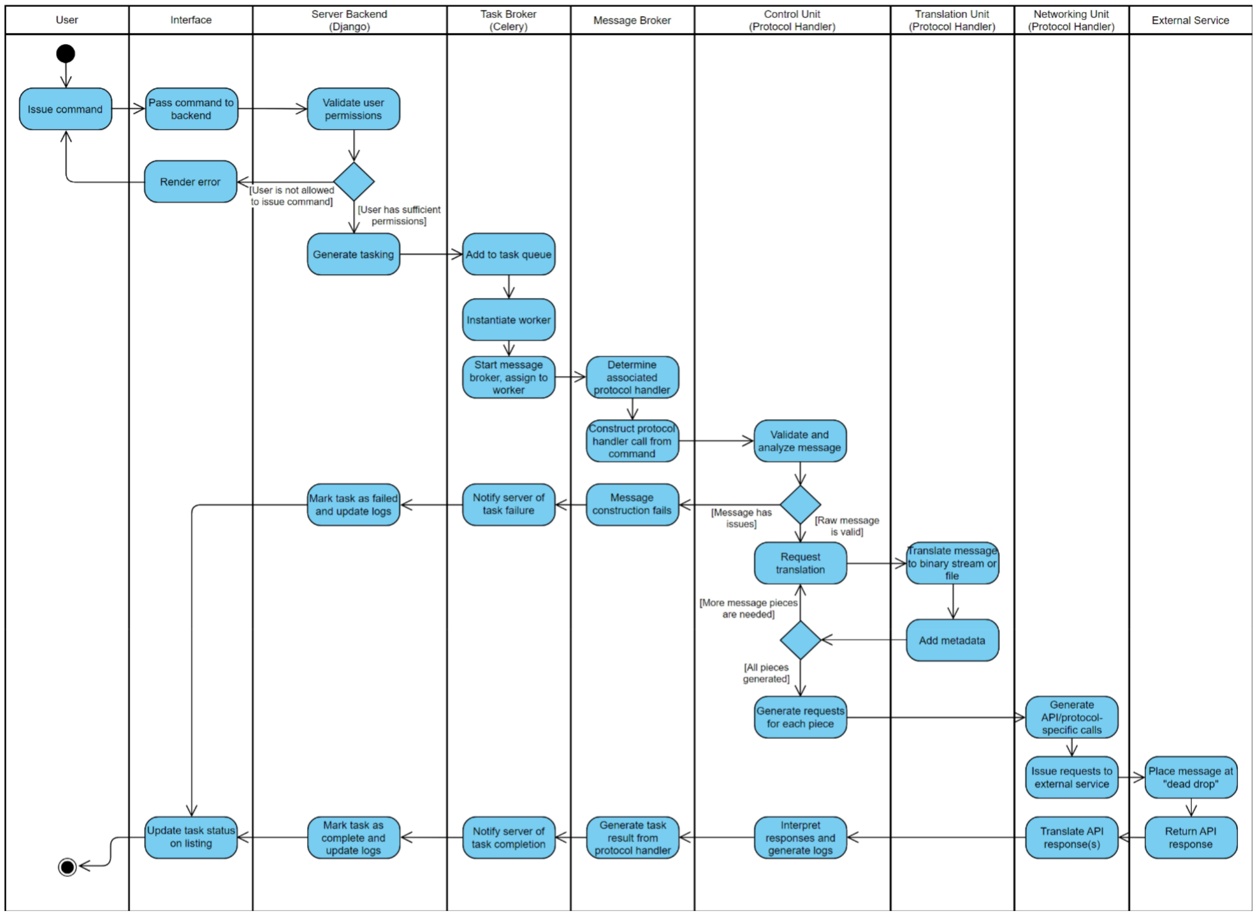
More specifically, the agent definition must contain the following information:

- The commands that the agent supports – more precisely, the "command_request" messages that the agent is capable of translating into specific calls to bundled binaries, generating "command_response" messages based on the results.
    - Optionally, these may also include *message interpreters*, inspired by the Mythic C2 framework's notion of *browser scripts*. These are server-side templates that can be used to convert "command_response" messages received from a particular agent for a specific command into a styled HTML element. These may contain visual representations of a command's output, formatted tables, or other user-friendly visualizations.
- The protocol handlers that the agent supports, if restricted for any reason. Agents should generally not be restricted from supporting every agent, as the protocols are intended to be designed as an opaque form of TCP.
- The binaries that are bundled with the agent for the propose of executing commands
- The build definitions – more precisely, the arguments that can be passed into the Docker container orchestrating its complex for the purposes of configuration. This typically includes encryption keys, the random seed used to determine the location of message dead drop locations, and the agent's ID.
- The docker configuration and container itself
- Other non-functional metadata, such as who developed the agent, the version of the agent, and the platforms it supports.

We currently expect to implement these requirements as a Python abstract base class, but are considering developing a standalone file format for organizing this information.

III. Detailed Design

Command Dispatch/Request (server-side)


User
Interface
Server Backend (Django)
Task Broker (Celery)
Message Broker
Control Unit (Protocol Handler)
Translation Unit (Protocol Handler)
Networking Unit (Protocol Handler)
External Service
Issue command
Pass command to backend
Validate user permissions
Render error
[User is not allowed to issue command]
[User has sufficient permissions]
Generate tasking
Add to task queue
Instantiate worker
Start message broker, assign to worker
Determine associated protocol handler
Construct protocol handler call from command
Validate and analyze message
Mark task as failed and update logs
Notify server of task failure
Message construction fails
[Message has issues]
[Raw message is valid]
Request translation
Translate message to binary stream or file
[More message pieces are needed]
Add metadata
[All pieces generated]
Generate requests for each piece
Generate API/protocol-specific calls
Issue requests to external service
Place message at "dead drop"
Update task status on listing
Mark task as complete and update logs
Notify server of task completion
Generate task result from protocol handler
Interpret responses and generate logs
Translate API response(s)
Return API response

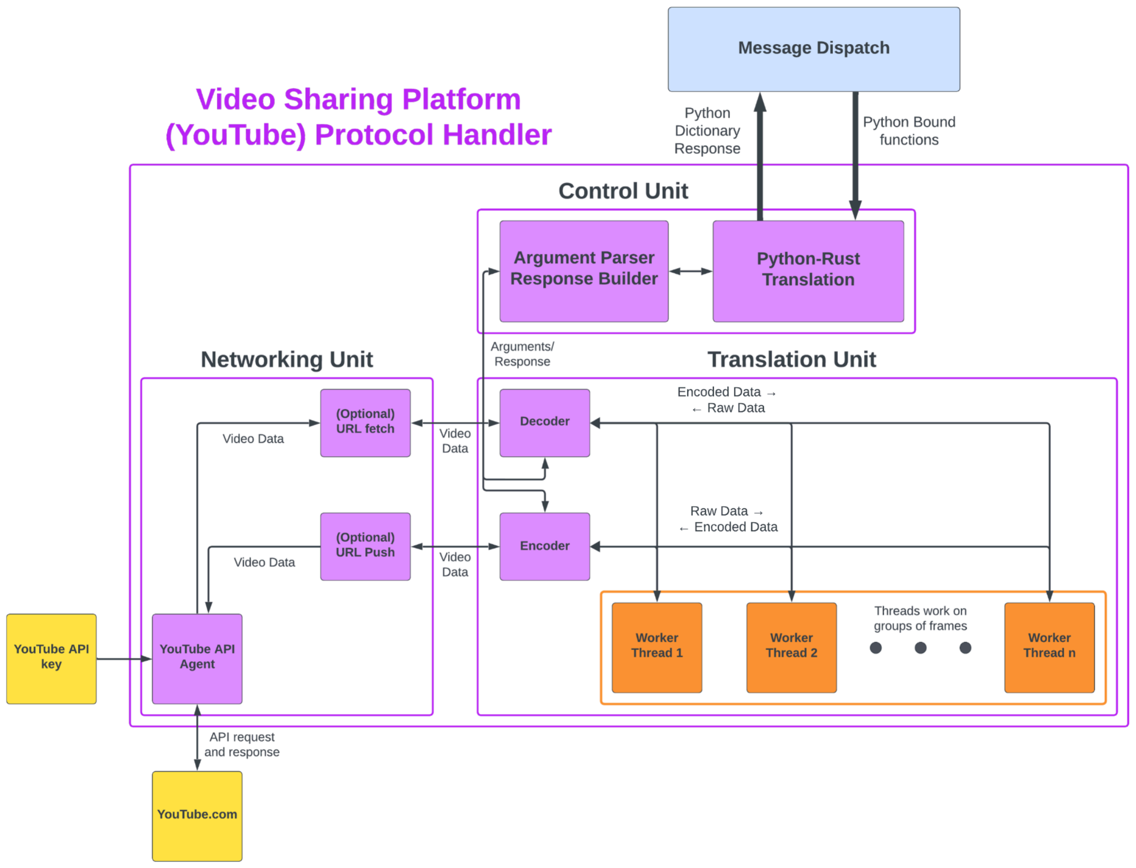
At a high level, the process can be described as following:

- A user issues a command to a specific agent through the web interface.
- The web interface generates a request to the server backend (Django).
- Django validates that the user has sufficient permissions to initiate a command request. (Django does little protocol- or command-level validation; these are mostly delegated to the protocol handler and the agent itself respectively.)
- If the user has sufficient permissions to initiate a command request, a task is generated to request that the corresponding message is issued to the target agent, which is passed to the task broker.
- The task is then added to the task queue, after which a worker is assigned to the task and starts an instance of the message dispatch module.
- The message dispatch module locates the associated protocol handler binary, constructs a protocol handler call from protocol definitions, and issues the call.
- The protocol handler validates the message to ensure that it can be sent. (If not, the protocol handler immediately exits and the task finishes early.) It then analyzes the message to determine if it must be split up; for each message piece, the translation unit is called to convert it into a message compatible with the target external service.

- Once all pieces have been created and metadata has been attached to each piece, the networking unit is called for each piece. The networking unit implements the necessary API calls needed to interact with the external service and place the message pieces at the specified dead drop location(s).
- The external service *should* respond with the result of each API call. These responses are analyzed by the protocol handler to determine if the message was successfully placed (as a safety measure, failed messages are not automatically retried by default).
- The result is passed through all upstream components, eventually resulting in the task's completion. The server is notified of the completed task, allowing the task to be displayed as complete the next time the frontend requests it.

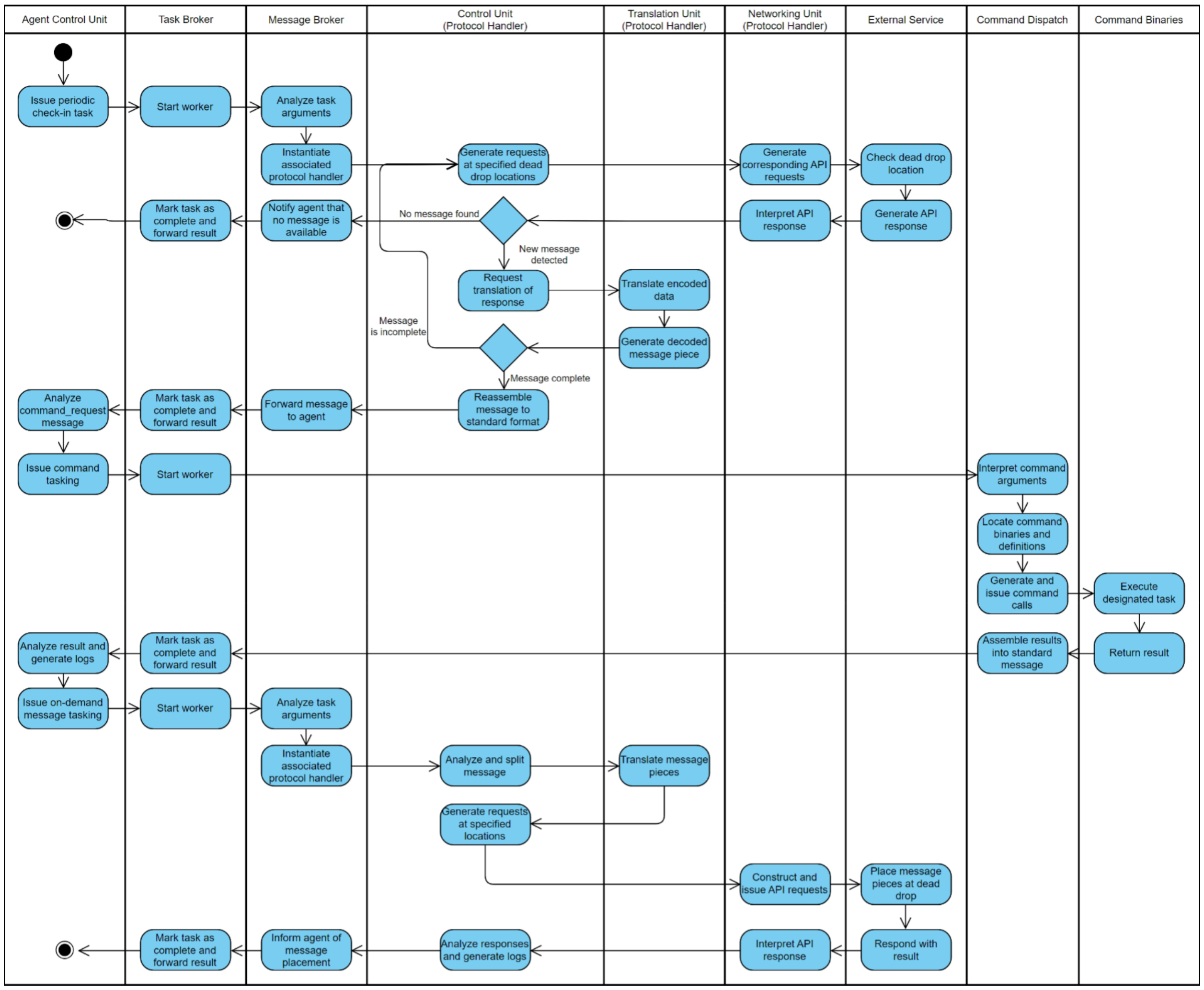
It is worth noting that there is no permission checking past Django – that is, it is theoretically possible for somebody with access to the server files to forge a message using the stored configuration files and encryption keys associated with the agent. However, this assumes that the device hosting the DeadDrop server has been compromised, and is outside the scope of application-level security.

Covert Protocol Implementation



- A message is dispatched to the protocol handler by using the Python function bindings and passing a Python dictionary with the required entries as the argument.
- This dictionary is translated into a Rust struct and is passed to the argument parser.
- The struct is parsed for the requested operation, formatting details, and optional push/pull flags. Either the decoder or the encoder is called with the requested arguments, depending on whether the message dispatcher is fetching or sending a message.
- If the encoder is called:
    - The file data is passed either by raw bytes or filename. The encoding details are written to the first frame of the video. The raw data is then split among *n* threads (defined in the arguments) and each thread begins a job encoding pursuant to the arguments.
    - The resulting frames are then reassembled and saved to disk.
    - If the push flag is set, the file is then sent to a YouTube API wrapper. Using the provided API key, the video is then uploaded to the owner of that key's channel.
    - Any return data is then passed back to the Python-Rust translation unit as a native Python object, passing data to the message dispatch unit as a standard message.
- If the decoder is called:
    - If the pull flag is set, the requested URL is retrieved by the YouTube API wrapper, saved to disk, and passed by filename back to the decoder.
    - The decoder reads the first frame for the encoding details and splits up the rest to the *n* worker threads along with the encoding details.
    - The threads decode the frames and return the raw data.
    - The raw data is reassembled and either saved to disk or returned to the Python-Rust translation unit.
    - Any return data is converted into a Python dictionary and returned to the message dispatch unit as a standard message.

Command Execution/Response (client-side)


Agent Control Unit
Task Broker
Message Broker
Control Unit (Protocol Handler)
Translation Unit (Protocol Handler)
Networking Unit (Protocol Handler)
External Service
Command Dispatch
Command Binaries
Issue periodic check-in task
Start worker
Analyze task arguments
Instantiate associated protocol handler
Generate requests at specified dead drop locations
Generate corresponding API requests
Check dead drop location
Generate API response
Interpret API response
No message found
Notify agent that no message is available
Mark task as complete and forward result
New message detected
Request translation of response
Translate encoded data
Generate decoded message piece
Message is incomplete
Message complete
Reassemble message to standard format
Forward message to agent
Mark task as complete and forward result
Analyze command_request message
Issue command tasking
Start worker
Interpret command arguments
Locate command binaries and definitions
Generate and issue command calls
Execute designated task
Return result
Assemble results into standard message
Mark task as complete and forward result
Analyze result and generate logs
Issue on-demand message tasking
Start worker
Analyze task arguments
Instantiate associated protocol handler
Analyze and split message
Translate message pieces
Generate requests at specified locations
Construct and issue API requests
Place message pieces at dead drop
Respond with result
Interpret API response
Analyze responses and generate logs
Inform agent of message placement
Mark task as complete and forward result

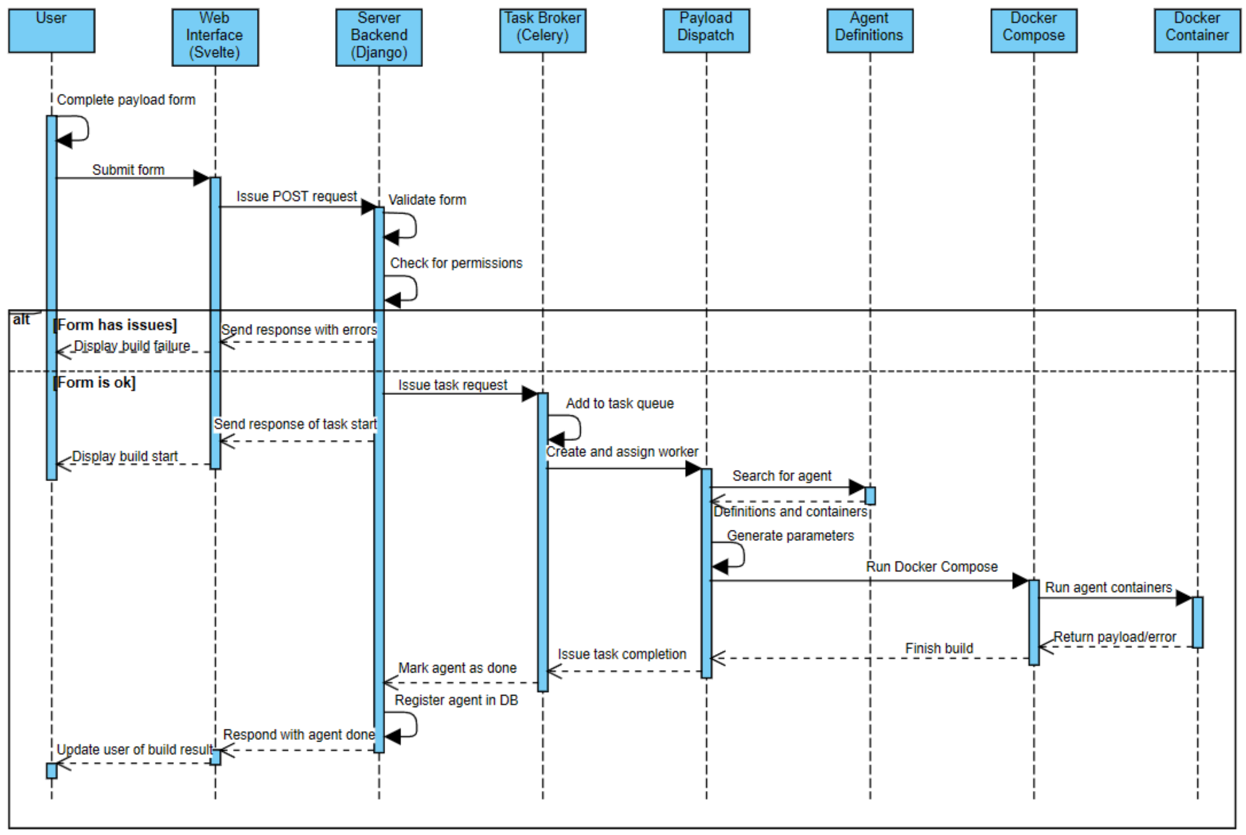
The process of executing a command on an agent is as follows:

- The agent makes a scheduled check-in at the specified dead drop location to see if any new messages have been placed. How these periodic check-ins are scheduled may vary, but one option is for the agent control unit to issue a periodic task to its task broker, after which the task broker starts a message dispatch worker after a specified period of time.
- The message dispatch module interprets the arguments originally set by the agent control unit when the task was first scheduled, calling a specific protocol handler and passing the result to its protocol control unit. The control unit then calls on the protocol networking unit to issue an API request to the external service to see if a new message has been placed.
- The external service responds to the networking unit, which decodes the result and forwards it to the protocol control unit. If a message *does not exist*, the task simply exits early. However, if a message is detected, the control unit passes the result to the translation unit, which decodes the message into (part of) a plaintext message.
    - If the protocol allows for messages to be split up, this process may repeat an arbitrary number of times until the message is reconstructed.

- The protocol control unit returns the reassembled plaintext message to the message dispatch module, which returns the reuslt to the agent control unit through the task broker. At this point, the task broker should observe that the message is a command_request message and schedule a task for the command dispatch module accordingly.
- The task broker receives the on-demand task and starts a worker for the command dispatch module. Based on the contents of the command_request message, the dispatch module identifies the command binaries responsible for fulfilling the command, forms arguments from the message, and then calls the binaries. It is up to the dispatch module to interpret the results and generate a command_response message.
- The message is sent back to the agent control unit, which should then start a new on-demand task to send a message back to the server. As with before, the task broker starts a worker for the message dispatch module, which calls a supported protocol handler. The protocol handler's translation unit converts the message into a format understood by the external service, which is then sent by the protocol handler's networking unit.

At this point, the agent's job is complete, and it is up to the server to retrieve the results of the command.

Payload Construction



The process of building a new payload used to install an agent on a target device is as follows:

- The user completes and submits a form to request a new payload for a particular agent.
- The frontend issues a formatted POST request to the backend.

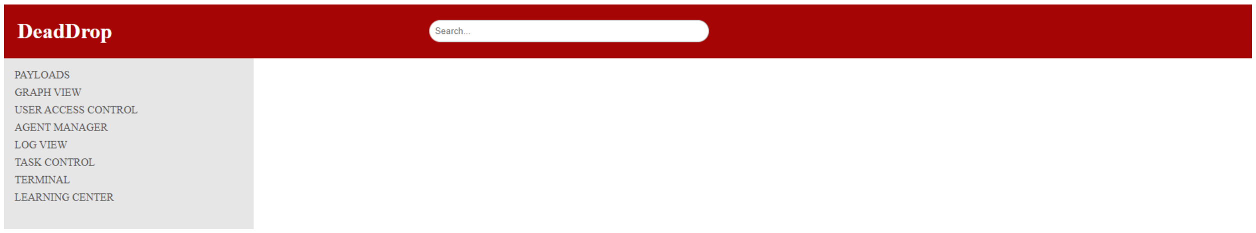
- The backend validates the form and asserts that the user has the permissions needed to generate and register new agents. (If the user does not have sufficient permissions, the flow ends immediately and the user is informed of the build failing.) If all checks pass, the backend issues a new task to the task broker and responds to the POST request with details of the task starting.
- The task broker adds the newly-created task to the task queue and starts a worker. When a worker is available, it is assigned to start an instance of the payload dispatch mechanism with the user's parameters.
- The payload dispatch module searches for the agent's details in the agent definitions listing. Based on these details, the user's parameters are interpreted into specific Docker calls and configuration files that are to be included with the final payload. Then, Docker Compose is called to orchestrate the containers responsible for building the payload.
- Based on compose.yaml, Docker Compose begins running the containers responsible for building the agent, setting up networking between the containers as needed. Each container builds each part of the agent, bundling and returning the result in an agent-specific manner.
- The payload dispatch module is informed of the build completing, forwarding both the payload and any relevant logs or errors. Celery is informed of the task's competition, which then forwards this information to the backend.
- The backend marks the task for building a new agent as done, and registers the agent in the database. Where applicable, a response is issued to the frontend to inform the user that the build has completed (or that errors have occurred).

At this point, it is the user's responsibility for getting the payload to the target device. In some real-world cases, this means exploiting some vulnerability that gives them access to the device and allows them to run arbitrary programs; in reality, this can range from dropping infected flash drives and physically breaking into the organization to sending out phishing emails. Once the agent runs for the first time, it will issue a message (using the protocol handler it was built with) to the server informing it of its successful installation.

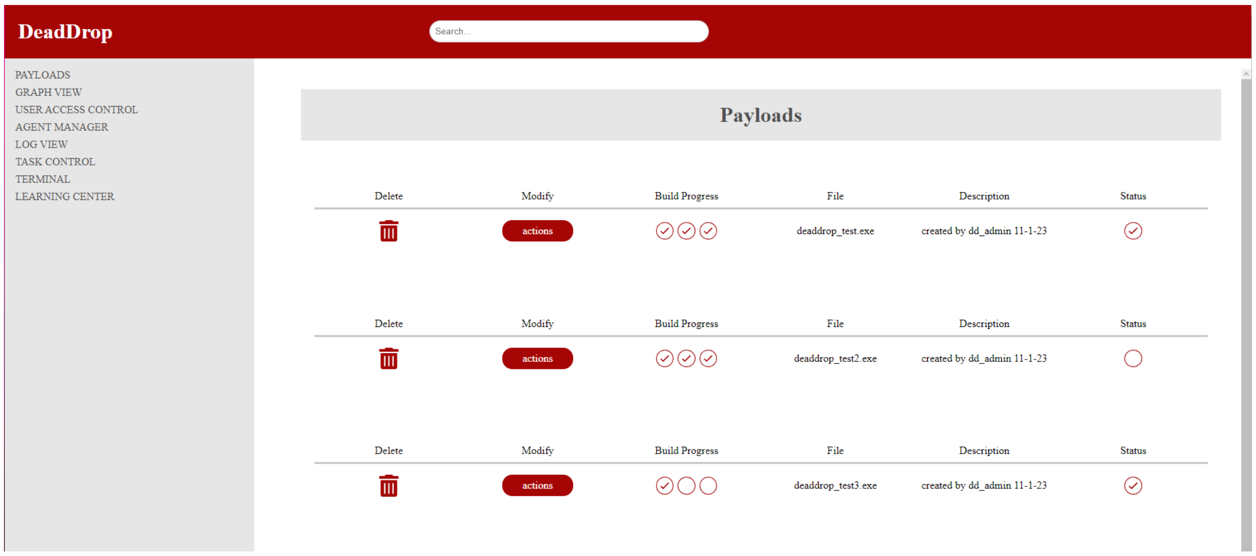
IV. User Interface Design

Here, we present ten snapshots of DeadDrop's server UI through which penetration testers can interact with the framework. Each of these provides some way to interact with the required features of DeadDrop.

Note that these snapshots are largely provided as a "proof-of-concept" layout of the Svelte components required to implement the functional requirements and identify the endpoints that need to be exposed by the Django server for Svelte to correctly reflect the state of the framework. As a result, the styling has been kept to a minimum until we finalize the Svelte-Django API.



DeadDrop navigation: Users can navigate through the DeadDrop C2 framework through the sidebar and search bar. The above image is an isolated view of the navigation, without any content rendered.

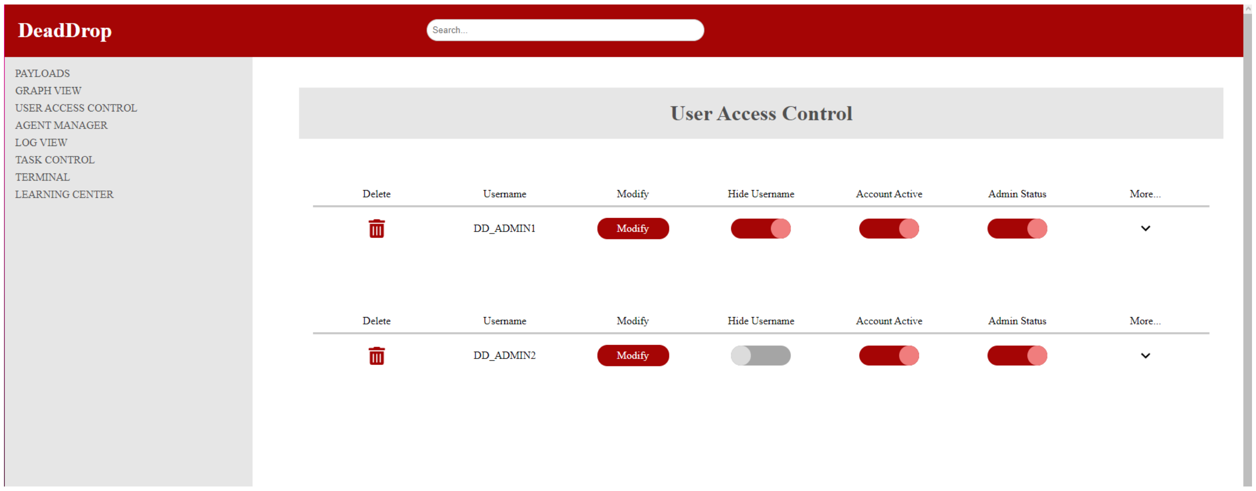


Delete	Modify	Build Progress	File	Description	Status
🗑	actions	✓✓✓	deaddrop_test.exe	created by dd_admin 11-1-23	✓

Delete	Modify	Build Progress	File	Description	Status
🗑	actions	✓✓✓	deaddrop_test2.exe	created by dd_admin 11-1-23	○

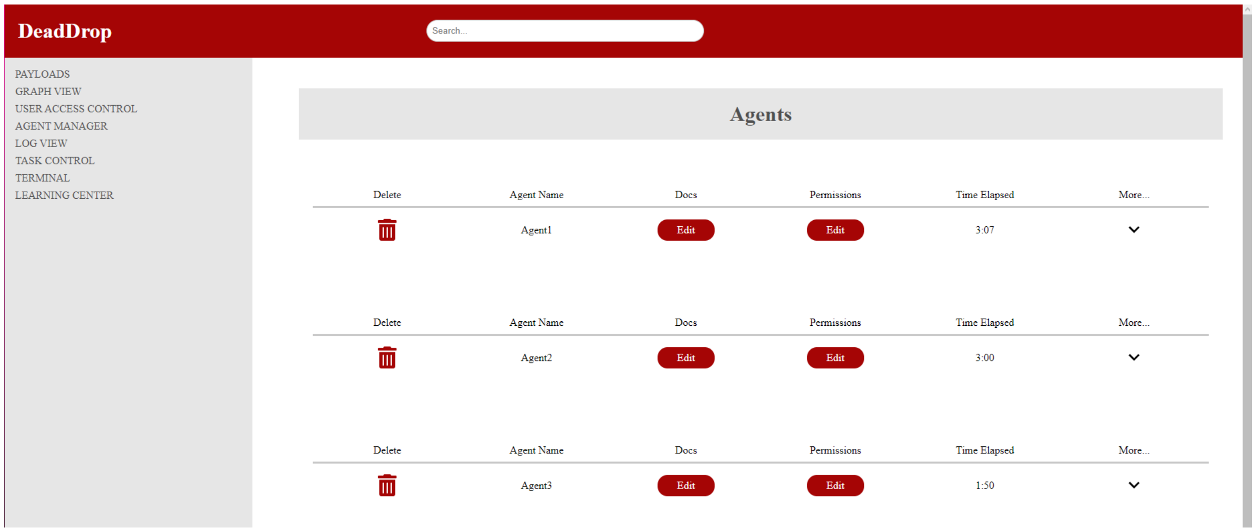
Delete	Modify	Build Progress	File	Description	Status
🗑	actions	✓○○	deaddrop_test3.exe	created by dd_admin 11-1-23	✓

DeadDrop payload management: Users can construct and manage payload instances, which carry the malware or exploits that can be used to accomplish tasks on compromised systems.



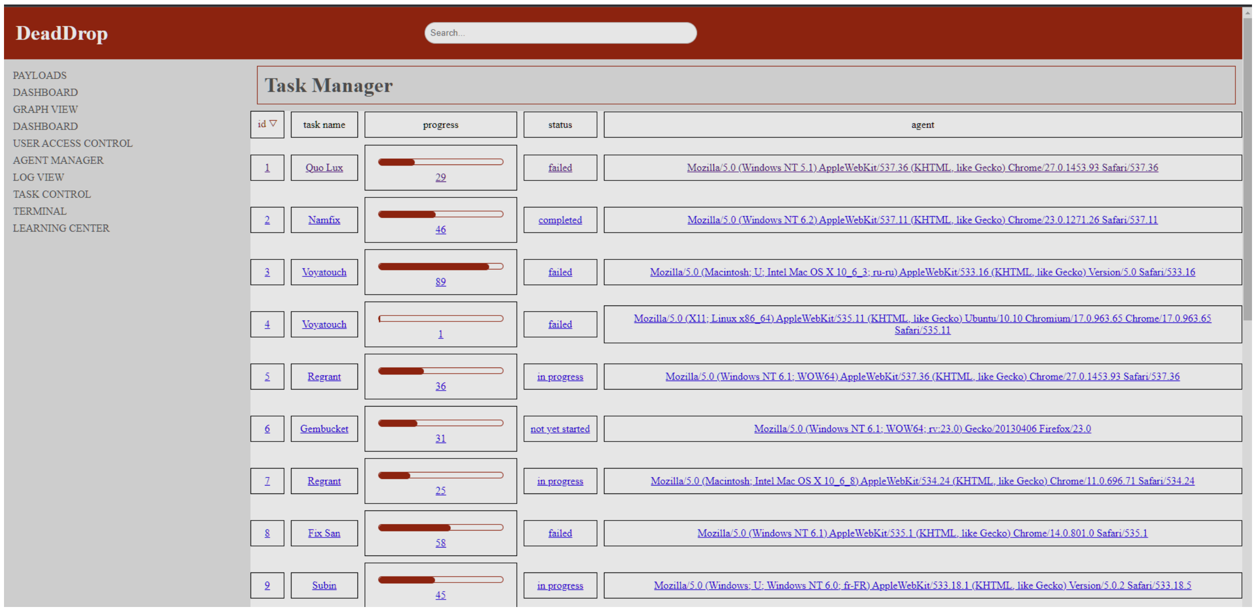
Delete	Username	Modify	Hide Username	Account Active	Admin Status	More...
🗑	DD_ADMIN1	Modify	●	●	●	⌄
🗑	DD_ADMIN2	Modify	○	●	●	⌄

DeadDrop User Access Control: Users can be created and managed from the UAC panel of the C2 framework. Visibility, permissions, active status, and administrative status can be managed for each user from the tab.



Delete	Agent Name	Docs	Permissions	Time Elapsed	More...
🗑	Agent1	Edit	Edit	3:07	⌄
🗑	Agent2	Edit	Edit	3:00	⌄
🗑	Agent3	Edit	Edit	1:50	⌄

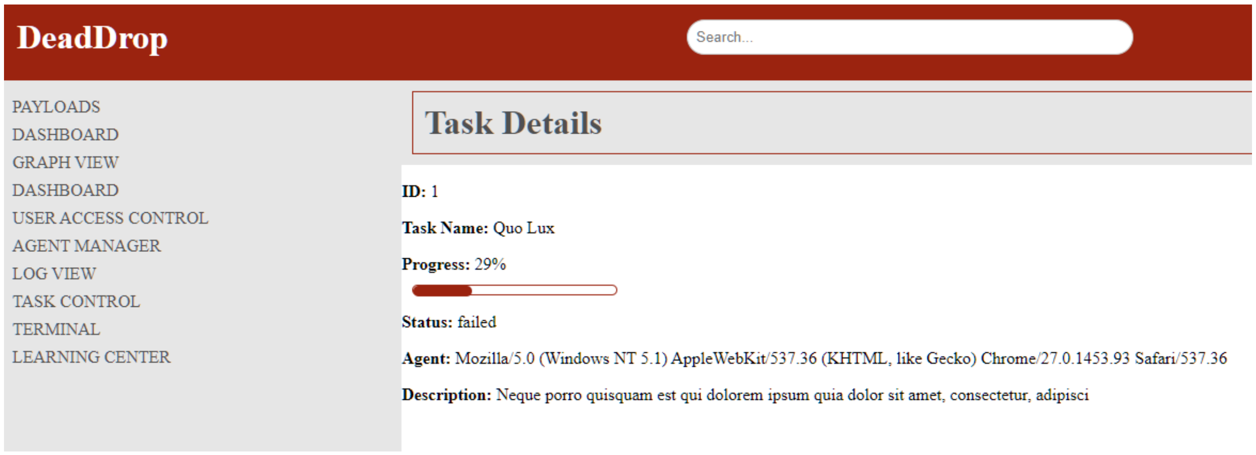
DeadDrop agent management: Users can construct and modify “agents” through this panel of the C2 framework. Agents are responsible for communicating between the C2 framework and the payload which executes code.



Task Manager

id ▽	task name	progress	status	agent
1	Quo Lux	29	failed	Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.1453.93 Safari/537.36
2	Namfix	46	completed	Mozilla/5.0 (Windows NT 6.2) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.26 Safari/537.11
3	Voyatouch	89	failed	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_3; ru-ru) AppleWebKit/533.16 (KHTML, like Gecko) Version/5.0 Safari/533.16
4	Voyatouch	1	failed	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.11 (KHTML, like Gecko) Ubuntu/10.10 Chromium/17.0.963.65 Chrome/17.0.963.65 Safari/535.11
5	Regrant	36	in progress	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.1453.93 Safari/537.36
6	Gembucket	31	not yet started	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:23.0) Gecko/20130406 Firefox/23.0
7	Regrant	25	in progress	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.71 Safari/534.24
8	Fix San	58	failed	Mozilla/5.0 (Windows NT 6.1) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.801.0 Safari/535.1
9	Subin	45	in progress	Mozilla/5.0 (Windows; U; Windows NT 6.0; fr-FR) AppleWebKit/533.18.1 (KHTML, like Gecko) Version/5.0.2 Safari/533.18.5

DeadDrop task management: Agents can be assigned "tasks", which are long-running commands that do not require instant responses. In most cases, these are scripts that perform common but complex tasks, such as dumping all the credentials in the volatile memory of a particular device. Users can view past, running, and scheduled tasks from this view.



Task Details

ID: 1

Task Name: Quo Lux

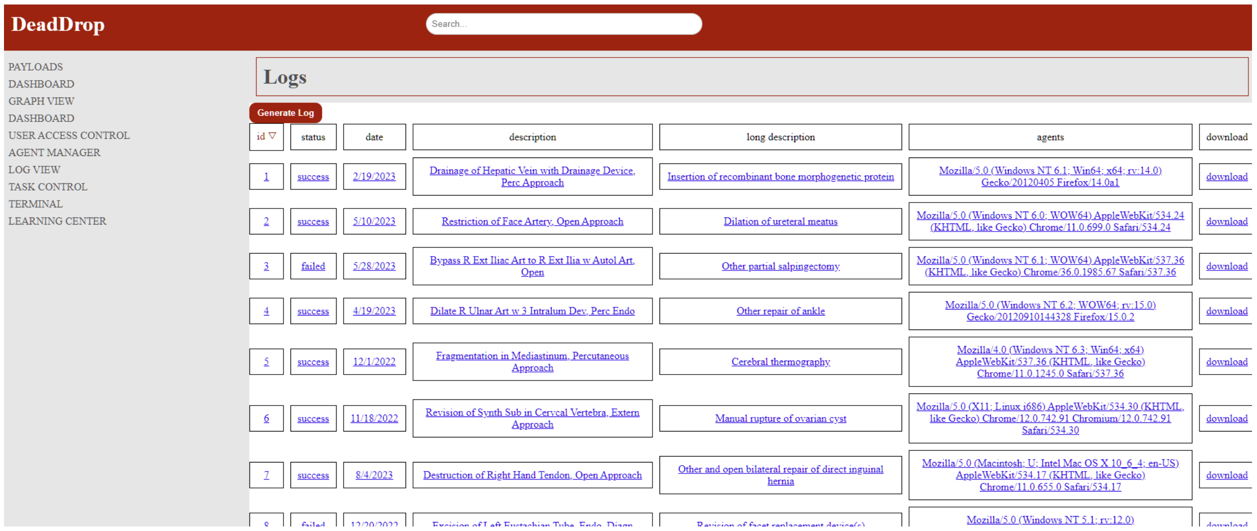
Progress: 29%

Status: failed

Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.1453.93 Safari/537.36

Description: Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci

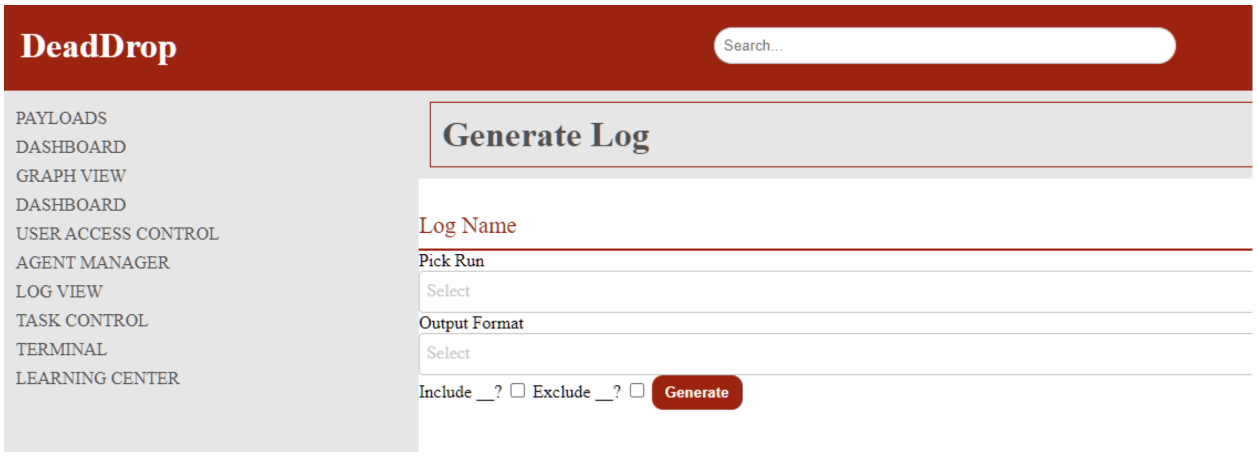
DeadDrop task details: Users can drill down into individual tasks to view their raw results. If the task has explicit support for HTML rendering, it can be presented in a graphical view for usability, such as the use of a directory listing for the output of a file enumeration command.



Logs

id ▽	status	date	description	long description	agents	download
1	success	2/19/2023	Drainage of Hepatic Vein with Drainage Device, Perc Approach	Insertion of recombinant bone morphogenetic protein	Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:14.0) Gecko/20120405 Firefox/14.0a1	download
2	success	5/10/2023	Restriction of Face Artery, Open Approach	Dilation of ureteral meatus	Mozilla/5.0 (Windows NT 6.0; WOW64) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.699.0 Safari/534.24	download
3	failed	5/28/2023	Bypass R Ext Iliac Art to R Ext Ilia w Autol Art, Open	Other partial salpingectomy	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.67 Safari/537.36	download
4	success	4/19/2023	Dilate R Ulnar Art w 3 Intralum Dev, Perc Endo	Other repair of ankle	Mozilla/5.0 (Windows NT 6.2; WOW64; rv:15.0) Gecko/20120910144328 Firefox/15.0.2	download
5	success	12/1/2022	Fragmentation in Mediastinum, Percutaneous Approach	Cerebral thermography	Mozilla/4.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/11.0.1245.0 Safari/537.36	download
6	success	11/18/2022	Revision of Synth Sub in Cervcal Vertebra, Extern Approach	Manual rupture of ovarian cyst	Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.91 Chromium/12.0.742.91 Safari/534.30	download
7	success	8/4/2023	Destruction of Right Hand Tendon, Open Approach	Other and open bilateral repair of direct inguinal hernia	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-US) AppleWebKit/534.17 (KHTML, like Gecko) Chrome/11.0.655.0 Safari/534.17	download
8	failed	12/20/2022	Excision of Left Eustachian Tube, Endo, Diagn	Revision of facet replacement device(s)	Mozilla/5.0 (Windows NT 5.1; rv:12.0)	download

DeadDrop logs: Users can view all logs collected by the server, allowing them to troubleshoot and identify potential risks within the penetration test. Users are able to filter and sort by each provided field within the DeadDrop log table.



Generate Log

Log Name

Pick Run

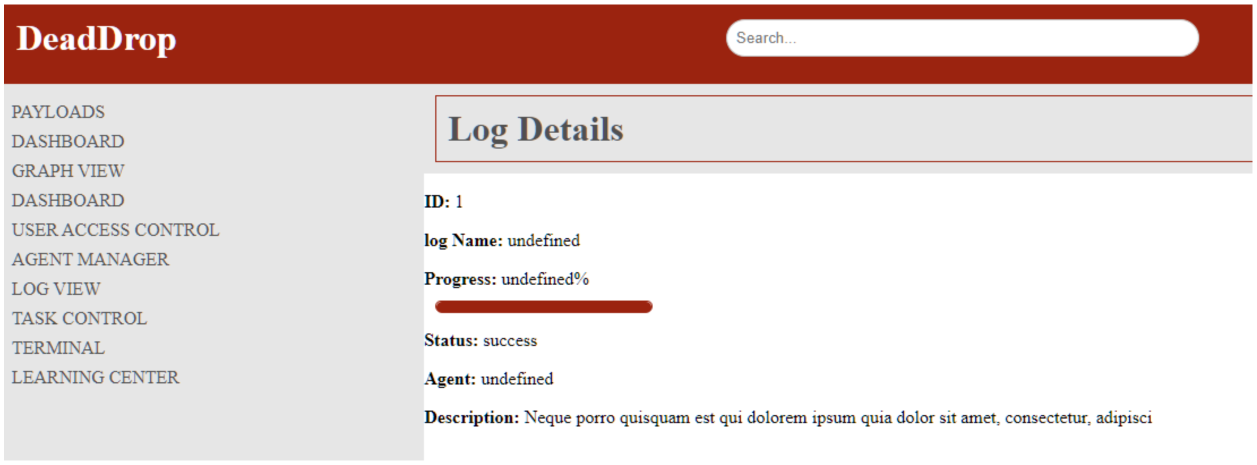
Select

Output Format

Select

Include __? ☐ Exclude __? ☐

DeadDrop log generator: In addition to viewing the raw, standardized logs collected and parsed by the server, users can also generate standalone log collections and reports that may be used in later analysis or demonstrations to organizational leadership.



ID: 1

log Name: undefined

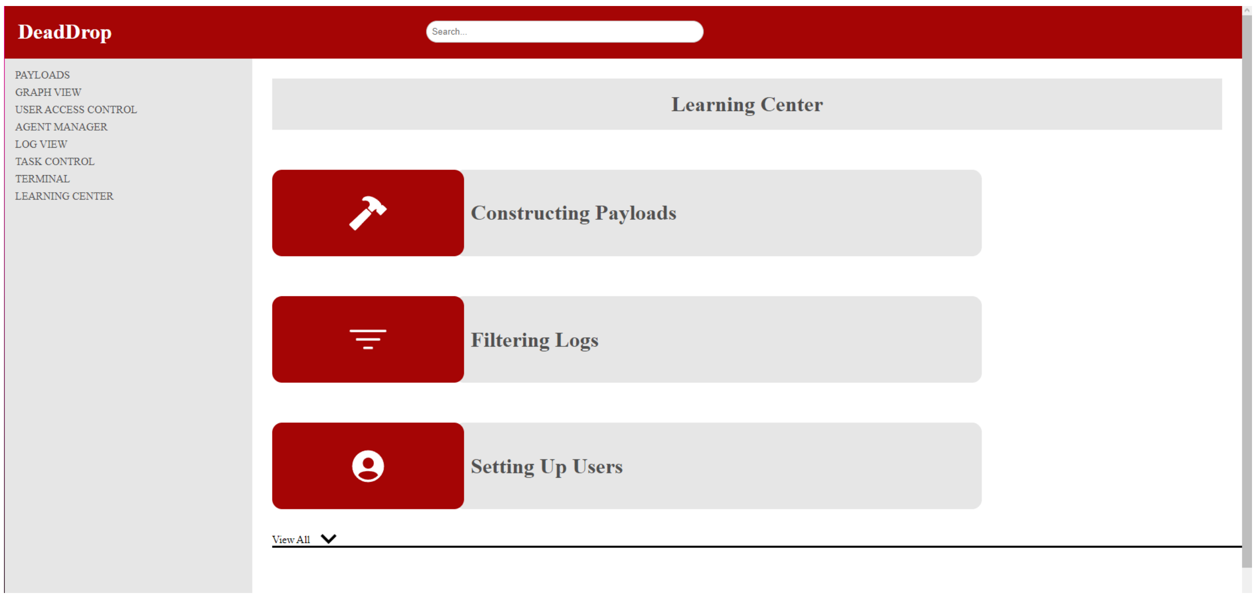
Progress: undefined%

Status: success

Agent: undefined

Description: Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci

DeadDrop log details: Users can drill down into individual logs as needed. This is particularly important for large log messages that are often created as a result of payload generation or communication; since they represent one coherent action, they are grouped into a single log object. However, it may be necessary to view the actual log contents for troubleshooting.



DeadDrop learning center: The learning center provides a centralized space for documentation regarding the uses and functions of DeadDrop. As with many other C2 frameworks, the documentation for the framework as a whole – as well as individual agents – are provided and generated within the server itself.

V. Version Control and Software Management

This project will be hosted and administered on GitHub through https://github.com/unr-deaddrop. Using an organization (as opposed to a single repository) gives us the flexibility needed to implement each of the modules described in *High and Medium-level Design* in a parallel manner.

This is particularly important, as it enforces the interfaces provided by servers and agents and avoids any unnecessary coupling between the two. That is to say, so long as agents and servers adhere to the standard messaging format and leverage protocol handlers as a black box, it should always be possible to develop new agents independently of the server (or any arbitrary implementation of the server).

Additionally, we plan to use this organization as our primary project management platform. We intend to use GitHub's issues and project management features to strongly tie commits to specific requirements, allowing us to track the progress of our project.

VI. Time Worked on Document

Note that the recorded time includes time spent performing research, writing code, and implementing the modules described in this document. This also includes meetings to go over the architecture as a team and verify that the systems are resonably designed and achievable.

(Time spent writing code is included for non-UI work because without it, we wouldn't have identified the problems that the architecture in this document fixes - code is not words on paper, but this paper doesn't exist without it. Naturally, the UI work is exclusively code-related, and therefore is included.)

Team Member	Hours	Sections Contributed
Jann Arellano	23.0	- UI Snapshots (*includes design discussions with Brian and code implementation*)
Brian Buslon	17.0	- UI Snapshots (*includes design discussions with Jann and code implementation*)
Keaton Clark	16.5	- Detailed Design - Covert Protocol Implementation (*includes code implementation*)
Lloyd Gonzales	28.0	- Abstract (from P1) - Introduction (mostly from P2) - High and Medium-level Design - Detailed Design - Command Dispatch/Request - Command Execution/Response - Payload Construction - Version Control and Software Management