

# DeadDrop

*a clever command and control framework for penetration testers*

## **Revised Specification and Design (P2)**

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

### **Advisor:**

Shamik Sengupta  
Professor  
University of Nevada, Reno

---

CS 426, Spring 2024

**Instructors:** David Feil-Seifer, Devrin Lee, Sara Davis

Department of Computer Science and Engineering  
University of Nevada, Reno  
February 21, 2024

## Table of Contents

<b>I. Abstract</b>	<b>4</b>
<b>II. Recent Project Changes</b>	<b>4</b>
<b>III. Updated Specifications</b>	<b>6</b>
A. Summary of Changes	6
B. Updated Technical Requirements	6
Functional Requirements	6
Nonfunctional Requirements	9
C. Updated Use Case Modeling	10
Traceability Matrix	12
<b>IV. Updated Design</b>	<b>14</b>
A. Summary of Changes in Project Design	14
B. Updated High- and Medium-Level Design	15
System Diagram and Overview	15
Server – Program Units	19
Framework Interface	19
Task Management	22
Message Generation and Fetching	24
Payload Construction	29
Agent – Program Units	34
Ungrouped Units	34
Message Generation and Fetching	37
Command Execution	40
Direct Forwarding	42
Primary Data Structures	44
Database Tables	44
DeadDrop Standard Message Format	51
Agent Definition	54
C. Updated User Interface Design	55
<b>V. Updated Glossary of Terms</b>	<b>65</b>
<b>VI. Engineering Standards and Technologies</b>	<b>68</b>
A. Standards	68
B. Technologies	68
<b>VII. Project Impact and Context Considerations</b>	<b>71</b>
A. Public Impact	71
B. Accessibility	72
<b>VIII. Updated List of References</b>	<b>74</b>
A. Problem Domain Book	74
B. Reference Articles	74

C. Additional Resources	75
<b>IX. Time Worked</b>	<b>76</b>

## I. Abstract

*We describe the continued implementation of 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. Our work for this semester encompasses the design and implementation of DeadDrop's server and web interface, platform-agnostic payload generation, and video-based covert communication protocols used to communicate with agents. This document outlines the updated design, specifications, and external resources driving the project, as well as new accessibility considerations.*

## II. Recent Project Changes

Since P1, there have been three major changes in the design of individual components.

First, we have settled on implementing a package manager. We originally planned on using Git submodules to add new agents and protocols into fresh installs of DeadDrop's backend, ensuring an internally consistent directory structure. However, we've realized that certain metadata must be generated dynamically for an install to reliably work on arbitrary devices, such as the Dockerfile used to build agents. In turn, it makes more sense to implement our own simple "package manager" to handle any post-install operations that are needed.

Second, we have changed the mechanism through which agents export available commands and other metadata. Agents must expose the commands and the required format of these commands to the backend in some way. Since both our agent and backend were written in Python, we originally planned on simply using Python classes to provide a consistent manner for exposing this information. However, this clearly does not reliably for agents written in other languages, so we've added the expectation that agents generate their supported commands and metadata as a standard JSON document after installation in the package manager.

Finally, we have changed the IPC interfaces for Pygin, the "reference" implementation of an agent that adheres to DeadDrop's interfaces. It depends on a multi-process architecture for tasking and executing commands, which currently involves interprocess communication over a shared Redis database. Redis does not support the storage of arbitrary Python objects without pickling, which is typically discouraged for security reasons. However, the security issue this avoids is moot; the agent already allows arbitrary execution of Python code in its design, and somebody with access to the agent (and therefore the device) could execute arbitrary code to begin with. In turn, we plan to allow IPC using pickled binary data over the JSON messages that we used in the prototype.

The use of pickled Python objects does raise the theoretical issue that an *unauthorized* attacker could add malicious pickled objects to the Redis database and forge malicious commands that appear to have come from an *authorized* attacker using DeadDrop. However, adequate server-side

logging would disprove this action by the authorized attackers (and there are larger issues if an unauthorized attacker already exists in the environment at the time of a penetration test).

As discussed in future sections, no high-level changes to design or implementation have occurred since the fall, and this remains true for P2. The medium- and low-level changes discussed above are reiterated in [Updated Specifications](#) and [Updated Design](#).

### III. Updated Specifications

#### A. Summary of Changes

The original specifications written in the fall were largely based on personal experience and our stakeholder interviews. Since then, we've recognized that there is additional *specific* functionality that must be present in DeadDrop for many of these specifications to make sense. For example, the nonfunctional requirement that DeadDrop maintain operational security necessarily requires functional requirements for ensuring the integrity and confidentiality of messages in transit. Since the fall Specification Document, we have more explicitly enumerated the functionality that we expect from DeadDrop before we can consider it a "complete" product for this semester.

However, as similarly mentioned in [Updated Design](#), the needs and best practices of the penetration testing industry have remained the same. For any given framework, penetration testers still expect feature parity with other well-established frameworks regarding the ability to generate new agents, execute commands on remote hosts, and ensure accountability across every action taken.

Thus, we have not *removed* any specifications outright since the fall. Instead, we have either broken down older specifications to be more precise, or we have added specifications that better map one-to-one to low-level implementations and design. Many of these functional requirements have corresponding GitHub issues or drafts that allow us to confirm that they have been met (or are close to being met); these are tracked in the [unr-deaddrop project board](#).

#### B. Updated Technical Requirements

##### Functional Requirements

ID	L	Description
FR01	1	DeadDrop agents shall issue heartbeat messages at a user-defined interval.
FR02	1	DeadDrop shall allow the user to build new instances of agents, irrespective of the platform used by the server.
FR03	1	DeadDrop shall allow the user to remotely execute shell commands through registered agents.
FR04	1	DeadDrop shall allow users to export logs to a standardized format.
FR05	1	DeadDrop shall allow users to remotely connect to the server interface.
FR06	1	DeadDrop shall encrypt messages using AES-CBC, with a minimum of a 128-bit key length.
FR07	1	DeadDrop shall implement a covert communication protocol that communicates solely over a popular service.

ID	L	Description
FR08	1	DeadDrop shall implement a user authentication system.
FR09	1	DeadDrop shall provide a Svelte web interface, a reference implementation for interacting with all available backend endpoints.
FR10	1	DeadDrop shall provide Pygin, a built-in reference implementation for an agent adhering to all available framework interfaces.
FR11	1	DeadDrop shall record all actions initiated by a user or an agent in a standard format.
FR12	1	DeadDrop shall record all data transferred via communication protocols in a standard format.
FR13	1	DeadDrop shall use the Elliptic Curve Digital Signature Algorithm (ECDSA) to verify the authenticity and integrity of messages.
FR14	1	Pygin shall bundle itself as a PyInstaller-generated executable.
FR15	1	Pygin shall contain portable, platform-specific executables for any dependencies needed.
FR16	1	Pygin shall execute commands concurrently.
FR17	1	Pygin shall provide a command allowing users to execute arbitrary Python code.
FR18	1	Pygin shall use supervisord to manage and configure its multi-process requirements.
FR19	1	The DeadDrop server shall allow users to display and filter logs without requiring the logs to be exported.
FR20	1	The DeadDrop server shall generate agent public/private key pairs in a cryptographically secure manner.
FR21	1	The Svelte frontend shall use automated Lighthouse CI tests for accessibility, the results of which must be saved and tracked over time.
FR22	2	DeadDrop agents shall provide a standard endpoint script for installation that allows for the discovery of available commands and other metadata as a JSON document.
FR23	2	DeadDrop shall allow the user to remove and remotely uninstall agents.
FR24	2	DeadDrop shall implement a “guardrail” system, preventing the framework from acting on user-defined hosts.
FR25	2	DeadDrop shall implement an “allowlist” system, forcing the framework to act only on user-defined hosts.
FR26	2	DeadDrop shall perform a user-defined action in the event an agent is unreachable

ID	L	Description
		for a specified number of attempts.
FR27	2	Pygin shall implement at least one example of a command renderer, whereby the response may be visualized by the frontend.
FR28	2	Pygin shall provide the ability to transfer arbitrary files to and from the infected device.
FR29	2	The DeadDrop server shall implement a package manager allowing for the installation and availability of new agents and protocols without requiring assumptions of the server's directory structure.
FR30	2	When presenting well-formatted responses from agents, DeadDrop may automatically visualize the response as a graph, chart, tree, or other visual listing.
FR31	3	DeadDrop agents shall automatically discover accessible devices from the current device.
FR32	3	DeadDrop agents shall be able to forward messages from other agents that do not have direct internet access.
FR33	3	DeadDrop agents should not assume the availability of platform or distribution-specific binaries.
FR34	3	DeadDrop protocol implementations shall expose an easy method to visit the URLs being used as dead drop locations.
FR35	3	DeadDrop shall allow users to open a live terminal session with an agent using a standard communication protocol.
FR36	3	DeadDrop shall allow users to register "virtual" agents that represent discovered machines with no agent installed.
FR37	3	DeadDrop shall allow users to unregister virtual agents.
FR38	3	DeadDrop shall allow users to write reusable scripts to execute sets of shell or agent-native commands on a particular agent.
FR39	3	The Svelte frontend shall provide users with desktop notifications of completed asynchronous tasks.
FR40	3	Pygin shall implement a command that steals Discord session tokens, if installed, for demonstration purposes.



Nonfunctional Requirements

ID	L	Description
NFR01	1	Covert communication protocols shall be implemented independently of DeadDrop and must include a set of unit tests.
NFR02	1	DeadDrop agents may be trivial to discover and modify at the endpoint level.
NFR03	1	DeadDrop agents should not perform irreversible actions on a device.
NFR04	1	DeadDrop must be difficult for unauthorized attackers to inject information into.
NFR05	1	DeadDrop must contain the information needed for an operator to manually write a detailed penetration report.
NFR06	1	DeadDrop shall adhere to the virtues of ethical hacking.
NFR07	1	DeadDrop shall implement at least one novel communication protocol.
NFR08	1	DeadDrop shall not violate terms of service on public websites.
NFR09	1	DeadDrop should be difficult to detect using network analysis techniques alone.
NFR10	1	DeadDrop should not require users to leverage the web interface as the sole method to interact with the framework.
NFR11	1	Pygin shall be resilient against long-running or failing commands.
NFR12	1	Agents should provide convenience commands for complex but common penetration testing operations.
NFR13	1	The installation of a DeadDrop agent should not immediately cause damage to a device.
NFR14	1	The Svelte frontend shall expose all available API endpoint information in a graphical manner.
NFR15	2	DeadDrop agents must have a contingency plan on loss of contact with the server.
NFR16	2	DeadDrop agents must leave little to no traces of their installation upon self-destruction.
NFR17	2	DeadDrop protocols must be easy to troubleshoot and configure.
NFR18	2	DeadDrop protocols must be resilient against unexpected interactions on chosen dead drop services.
NFR19	2	DeadDrop shall be built in a platform-agnostic manner.
NFR20	2	DeadDrop should be easy to use for users with experience in existing major C2 frameworks.

ID	L	Description
NFR21	2	DeadDrop should bundle log messages to the greatest extent possible, minimizing the frequency of individual dead drop messages dedicated to logging.
NFR22	2	DeadDrop software written in Python shall adhere to PEP8 and must pass static type checks.
NFR23	2	The Svelte frontend must function well on screens meeting the Bootstrap 5 large breakpoint (a width of 992 pixels).
NFR24	2	The Svelte frontend shall uphold, at minimum, the WCAG 2.0 Level A standards.
NFR25	3	DeadDrop agents shall not require internet connectivity to function for a user-specified period of time.
NFR26	3	The Svelte frontend should provide additional statistics and functionality without requiring significant backend strain.

### C. Updated Use Case Modeling

A table containing some (but not all) of DeadDrop's most important use cases is below.

ID	Use Case	Description
UC01	createEndpoint	To generate a new agent, the user may command the server to compile or generate a new agent. The agent should be uniquely identifiable and should register with the server upon installation.
UC02	deleteEndpoint	The user may destroy an existing, registered agent. The agent should uninstall itself, and any dependent agents, from their machines.
UC03	createVirtualEndpoint	Either the user or an automated process may register a "virtual" agent, which is a known, accessible machine that does not have an agent installed.
UC04	deleteVirtualEndpoint	The user may delete any registered "virtual" agents, effectively removing these machines from view. They may be rediscovered if necessary.
UC05	showLogs	The user should be able to view (and export) the logs that have been collected by the server.
UC06	createVirtualEnvironment	The user should be able to create virtual network segments (environments) for the framework to operate in. These can be used to group controlled devices, restrict framework activity, or enforce agreements between the client and the testers.

UC07	authenticateUser	The user can enter authentication information to gain access to services provided by the server.
UC08	configureEndpoint	The user or server may request that an agent modify its behavior, such as by changing the communication secret key, the protocol used, its tags, or the name it is internally identified by.
UC09	configureVirtualEndpoint	The user or server may modify “virtual” agents in a similar manner to the above. Since these are loosely defined, this can encompass anything from changing user-written notes to changing their connections in a graph view.
UC10	writeEndpoint	The server or agent may interact with each other over a specified communication protocol. Usually, this involves placing the command at the agreed-upon dead drop on the legitimate service, at which point it is up to the recipient to retrieve it.
UC11	readEndpoint	The server or agent may retrieve (or simply check) new messages placed by the sender over a specified communication protocol. It is up to the recipient to decrypt and decode the message as needed.
UC12	viewAgentStatus	The current status of all agents is shown. This includes their recent logs, state, and location. This will also allow the user to attach to one of the agent’s consoles by opening a reverse shell.

Traceability Matrix

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11	UC12
FR1												
FR2												
FR3												
FR4												
FR5												
FR6												
FR7												
FR8												
FR9												
FR10												
FR11												
FR12												
FR13												
FR14												
FR15												
FR16												
FR17												
FR18												
FR19												
FR20												
FR21												
FR22												
FR23												
FR24												

FR25												
FR26												
FR27												
FR28												
FR29												
FR30												
FR31												
FR32												
FR33												
FR34												
FR35												
FR36												
FR37												
FR38												
FR39												
FR40												

## IV. Updated Design

### A. Summary of Changes in Project Design

Our high-level design remains virtually unchanged when compared to the Design Document written during the fall. This is primarily because the original high-level design was made with the functionality of mature C2 frameworks in mind; if DeadDrop is to provide feature parity with what cybersecurity professionals expect, then certain components must necessarily exist. The process for creating the high-level architecture of DeadDrop was driven by making a list of core features between major C2 frameworks, and then constructing a high-level component that could fulfill each feature. These components were then tied together in the manner we felt made the most sense.

The needs of penetration testers in industry have not changed significantly since the fall; in turn, those components still exist in our high-level design and largely interact with each other in the same way.

However, the medium- and low-level implementations of these components have changed considerably since the fall. The major changes include:

- The use of two full-stack frameworks to implement the server interface, namely Django and Svelte
- The addition of a built-in package manager as part of the server's payload generation component
- The addition and removal of various fields in core data structures, such as messages passed between components or the schema of the Django backend tables

Many of these changes arose as we realized that certain components lacked the information needed for them to fulfill their functionality. For example, we determined that it was far too difficult to "add" agents and protocols to a new installation of the DeadDrop backend without a dedicated subcomponent that updated various build paths and file locations at runtime. In turn, although the actual interfaces between other components have not changed, the [Payload Construction](#) unit has changed considerably in structure.

Other changes were the result of personal interests and practical considerations. For example, the original design called for Django to be the sole framework used in the frontend. However, we felt that Django's built-in templating did not give us the flexibility and efficiency we needed to build our pages. In turn, we opted to use the full-stack Svelte framework to implement the rendering of frontend pages, while Django handled the underlying storage and processing of our models. Django REST Framework drives much of the interface between Django and Svelte.

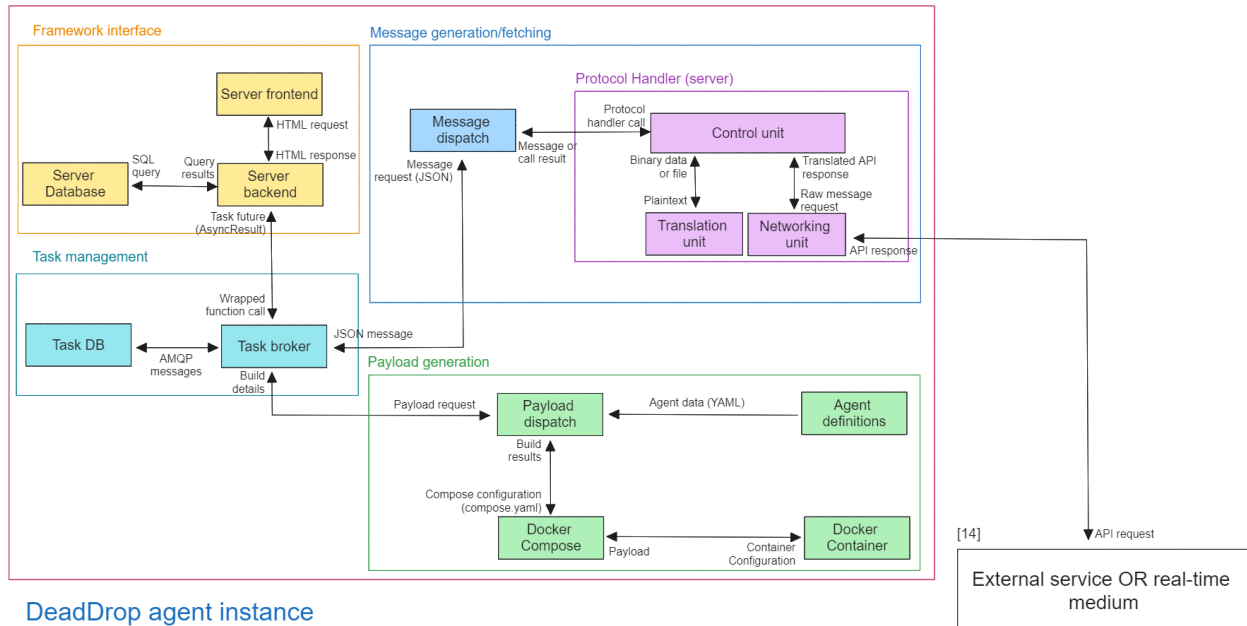
Finally, there have been some minor changes to the underlying data structures composing the interfaces between several high-level components, as well as changes to the database tables in the Django server backend. These changes have been documented in the [Primary Data Structures](#) section.

## B. Updated High- and Medium-Level Design

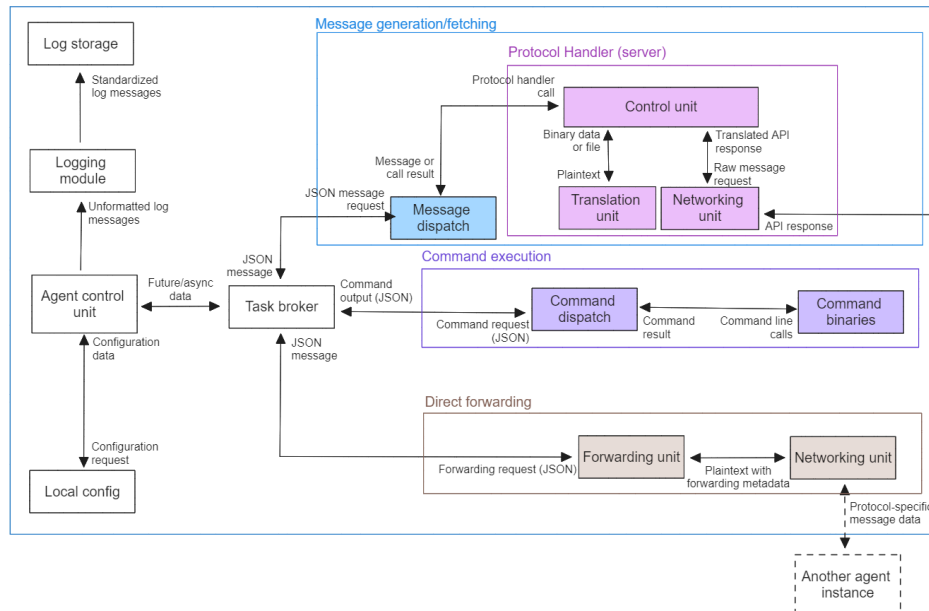
### System Diagram and Overview

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

#### DeadDrop server



#### DeadDrop agent instance



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 → <a href="#">Server Backend</a>	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 → <a href="#">Server Frontend</a>	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 → <a href="#">Server Database</a>	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 → <a href="#">Task Broker</a>	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 → <a href="#">Task Database</a>	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
	<u>Message Dispatch</u>	over a specific protocol.
[7]	Server → Message Generation and Fetching → Protocol Handler → <u>Control Unit</u>	Component of a protocol handler responsible for either: <ul style="list-style-type: none"> <li>- scheduling and generating messages to be sent over a protocol, or;</li> <li>- checking for new or expected messages and reassembling components of these messages over a protocol.</li> </ul>
[8]	Server → Message Generation and Fetching → Protocol Handler → <u>Translation Unit</u>	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 → <u>Networking Unit</u>	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 → <u>Payload Dispatch</u>	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 → <u>Agent Definitions</u>	Static set of files containing the information necessary to build an agent.
[12]	Server → Payload Generation → <u>Docker Compose</u>	Responsible for managing and instantiating the container (or containers) needed to generate the payloads for an agent.
[13]	Server → Payload Generation → <u>Docker Container</u>	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) → <u>External Service or Real-Time Medium</u>	This represents either: <ul style="list-style-type: none"> <li>- The external service used to conceal communication between servers and agents</li> <li>- A "standard" protocol and medium used for real-time communication, like a direct TCP connection</li> </ul>
[15]	Agent → <u>Control Unit</u>	Manages all decisionmaking for the agent; interprets and acts on messages received from the C2 server.
[16]	Agent → <u>Local</u>	Static files dictating various configuration settings for the

#]	Module	Summary
	<a href="#"><u>Configuration</u></a>	agent, such as the agent's ID, any encryption keys in use, logging settings, and so on.
[17]	Agent → <a href="#"><u>Logging Module</u></a>	Centralized logging module that writes logs to external storage based on its configuration settings.
[18]	Agent → <a href="#"><u>Log Storage</u></a>	Arbitrary form of storage used to hold logs generated by any part of the agent.
[19]	Agent → <a href="#"><u>Task Broker</u></a>	Responsible for asynchronously executing and managing tasks and returning their results to the agent control unit.
[20]	Agent → Message Generation/Fetching → <a href="#"><u>Message Dispatch</u></a>	Agent module identical in purpose to the <i>Message Dispatch</i> module of the server.
[21]	Agent → Message Generation/Fetching → Protocol Handler → <a href="#"><u>Control Unit</u></a>	Agent module identical in purpose to the <i>Control Unit</i> module of the server.
[22]	Agent → Message Generation/Fetching → Protocol Handler → <a href="#"><u>Translation Unit</u></a>	Agent module identical in purpose to the <i>Translation Unit</i> module of the server.
[23]	Agent → Message Generation/Fetching → Protocol Handler → <a href="#"><u>Networking Unit</u></a>	Agent module identical in purpose to the <i>Networking Unit</i> module of the server.
[24]	Agent → Command Execution → <a href="#"><u>Command Dispatch</u></a>	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 → <a href="#"><u>Command Binaries</u></a>	Standalone binaries that accept command-line arguments and are designed to achieve a single task on a device.
[26]	Agent → Direct Forwarding → <a href="#"><u>Forwarding Unit</u></a>	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 → <a href="#"><u>Networking Unit</u></a>	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 → <b><i>Server Backend</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Framework Interface → <i>Server Frontend</i></li> <li>- Server → Framework Interface → <i>Server Database</i></li> <li>- Server → Task Management → <i>Task Broker</i></li> </ul>

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	<ul style="list-style-type: none"> <li>- Formatted user inputs (from forms or callbacks) through the frontend</li> <li>- Arbitrary data retrieved from the database</li> <li>- Task information from completed or running tasks from the task broker</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Dynamic webpages rendered through the frontend</li> <li>- SQL queries issued to the server database (as interpreted by Django's ORM)</li> <li>- New tasks issued to the task broker</li> </ul>
Description	<p>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.</p> <p>In particular, this includes:</p> <ul style="list-style-type: none"> <li>- An authentication system that ensures that every user is held accountable for the actions they take through DeadDrop</li> <li>- A permissions system that ensures only specific users can initiate certain actions or view sensitive information gained through the penetration test</li> <li>- 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</li> <li>- Translating requests from users to initiate specific actions into specific tasks that the rest of the framework (abstracted from the user) can understand</li> </ul> <p>We intend to implement the backend with the <a href="#">Django</a> 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.</p>

[2] Server Frontend	
Parent units	Server → Framework Interface → <b><i>Server Frontend</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Framework Interface → <i>Server backend</i></li> </ul>
Summary	Provides users with a web interface to easily interact with the framework, primarily for issuing commands and viewing outputs and logs.

Inputs	<ul style="list-style-type: none"> <li>- Arbitrary user inputs through forms, buttons, or other input presented by the web interface</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Formatted user inputs passed to the the server backend to perform a particular action</li> </ul>
Description	<p>The server frontend provides users with a graphical (web) interface to interact with individual parts of the framework. As shown in the <a href="#">User Interface Design</a> section, this includes the following:</p> <ul style="list-style-type: none"> <li>- The ability to issue commands to agents and view their results, possibly as charts, tables, or other formatted visuals</li> <li>- A graphical overview of all devices that the framework either has control over or is aware of</li> <li>- The ability to parse and view logs stored in the backend</li> </ul> <p>We intend to implement the frontend with the <a href="#">Svelte</a> 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.</p>

[3] Server Database	
Parent units	Server → Framework Interface → <i>Server Database</i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Framework Interface → <i>Server Backend</i></li> </ul>
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	<ul style="list-style-type: none"> <li>- Formatted SQL queries as generated by Django's ORM</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Arbitrary, formatted query results passed to Django's ORM</li> </ul>
Description	<p>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:</p> <ul style="list-style-type: none"> <li>- 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</li> <li>- 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.</li> </ul> <p>In our case, we plan on using a <a href="#">Postgres</a> database as the backend. Some</p>

	<p>reasons for doing so include:</p> <ul style="list-style-type: none"> <li>- 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</li> <li>- The extensive number of tools and libraries available for interacting with Postgres database, including Python's psycopg library and the pgAdmin GUI</li> <li>- The ability to easily migrate a local Postgres database to a remote service if a shared cloud database becomes necessary for any reason</li> </ul>
--	---

### 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 → <i><b>Task Broker</b></i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Framework Interface → <i>Server Backend</i></li> <li>- Server → Task Management → <i>Task Database</i></li> <li>- Server → Message Generation/Fetching → <i>Message Dispatch</i></li> <li>- Server → Payload Generation → <i>Payload Dispatch</i></li> </ul>
Summary	Offloads asynchronous tasks from the server interface, reporting results to the server backend and performing on-demand and scheduled periodic tasks.
Inputs	<ul style="list-style-type: none"> <li>- A Task object, which is any arbitrary request to asynchronously execute code from the server</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- An instance of AsyncResult, which represents the result of the asynchronous computation (commonly known as "futures" in other languages and asynchronous frameworks)</li> <li>- Arbitrary requests to check for messages, send messages, or start the construction of a new agent</li> </ul>

Description	<p>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.</p> <p>The most common library used to solve this problem is <a href="#">Celery</a>, 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.)</p> <p>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:</p> <ul style="list-style-type: none"> <li>- Celery supports both multiprocess and multithreaded workers, allowing us to scale background tasking however we’d like</li> <li>- Celery supports scheduled tasks, allowing us to execute tasks based on a simple interval</li> <li>- 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</li> </ul> <p>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.</p>
-------------	--

[5] Task Database	
Parent units	Server → Framework Interface → <b><i>Task Database</i></b>
Interacts with or depends on	- Server → Task Management → <i>Task Broker</i>
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	<p>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.</p> <p>For Celery, two main brokers are supported – <a href="#">RabbitMQ</a> and <a href="#">Redis</a>. 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.</p> <p>From an implementation perspective, our work on the task database involves correctly configuring Redis and observing its performance throughout development.</p>

#### 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 → <i>Message Dispatch</i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Task Management → <i>Task Broker</i></li> <li>- Server → Message Generation and Fetching → Protocol Handler → Control Unit</li> </ul>
Summary	Module responsible for retrieving the information and protocol handler needed to communicate with an agent over a specific protocol.
Inputs	<ul style="list-style-type: none"> <li>- <i>When sending covert messages</i>, 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)</li> <li>- <i>When checking for new covert messages</i>, a request to check for new messages using a particular protocol at a specific location</li> <li>- <i>When used in real-time communication</i>, an arbitrary input (usually a shell command)</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- <i>When sending covert messages</i>, 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 <i>or</i> any errors</li> <li>- <i>When checking for new covert messages</i>, the reassembled plaintext message</li> <li>- <i>When used in real-time communication</i>, an arbitrary output (usually the stdout of a shell command)</li> </ul>
Description	<p>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.</p> <p>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.</p> <p>Additionally, the message dispatch module determines which arguments</p>

	<p>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.</p>
--	--

[7] Control Unit	
Parent units	Server → Message Generation and Fetching → Protocol Handler → <b>Control Unit</b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Message Generation/Fetching → <i>Message Dispatch</i></li> <li>- Server → Message Generation/Fetching → Protocol Handler → <i>Translation Unit</i></li> <li>- Server → Message Generation/Fetching → Protocol Handler → <i>Networking Unit</i></li> </ul>
Summary	<p>Component of a protocol handler responsible for either:</p> <ul style="list-style-type: none"> <li>- scheduling and generating messages to be sent over a protocol, or;</li> <li>- checking for new or expected messages and reassembling components of these messages over a protocol.</li> </ul>
Inputs	<ul style="list-style-type: none"> <li>- <i>When sending or receiving covert messages</i>, 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</li> <li>- <i>When used in real-time communication</i>, an arbitrary input (usually a shell command)</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- <i>When sending covert messages</i>, a log message indicating that the “dead drop” was successfully placed or that it was not, including any responses from the external service.</li> <li>- <i>When receiving covert messages</i>, the reassembled plaintext message.</li> <li>- <i>When used in real-time communication</i>, an arbitrary output</li> </ul>
Description	<p>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.</p> <p>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.</p>

	<p>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.</p>
--	---

[8] Translation Unit	
Parent units	Server → Message Generation and Fetching → Protocol Handler → <b><i>Translation Unit</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Message Generation/Fetching → Protocol Handler → <i>Control Unit</i></li> </ul>
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	<ul style="list-style-type: none"> <li>- A single plaintext message (or binary stream)</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- One or more encoded messages (possibly files or in-memory bytes of data) to be placed at various locations on the target</li> </ul>
Description	<p>The translation unit is responsible for encoding plaintext messages (which may contain protocol-specific metadata beyond that specified in the <a href="#">DeadDrop standard message format</a>) 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.</p> <p>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 <a href="#">ffmpeg</a> 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.</p>

[9] Networking Unit	
Parent units	Server → Message Generation and Fetching → Protocol Handler → <b><i>Networking Unit</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Message Generation/Fetching → Protocol Handler → <i>Control Unit</i></li> <li>- External Service or Real-Time Medium</li> </ul>
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	<ul style="list-style-type: none"> <li>- <i>When sending messages</i>, a single encoded message along with the information needed to place the message at the target external service</li> <li>- <i>When receiving messages</i>, the information needed to request data from the target external service</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- <i>When sending messages</i>, the response from the service as a result of “placing” the dead drop</li> <li>- <i>When receiving messages</i>, the response from the service containing the information (which is expected to contain an encoded message or part of a message)</li> </ul>
Description	<p>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.</p> <p>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:</p> <ul style="list-style-type: none"> <li>- 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</li> <li>- the translation unit would convert the raw message pieces into Base64-encoded strings</li> <li>- the networking unit would actually interact with Reddit’s API to place the messages at the desired locations</li> </ul> <p>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.</p>

[14] External Service or Real-Time Medium	
Parent units	(none) → <b>External Service or Real-Time Medium</b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Message Generation and Fetching → Protocol Handler → <i>Networking Unit</i></li> <li>- Agent → Message Generation and Fetching → Protocol Handler → <i>Networking Unit</i></li> </ul>
Summary	<p>This represents either:</p> <ul style="list-style-type: none"> <li>- The external service used to conceal communication between servers and agents</li> <li>- A “standard” protocol and medium used for real-time communication, like a direct TCP connection</li> </ul>

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	<p>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.</p> <p>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).</p> <p>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.</p>

### 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 → <i>Payload Dispatch</i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Task Management → <i>Task Broker</i></li> <li>- Server → Payload Generation → <i>Agent Definitions</i></li> <li>- Server → Payload Generation → <i>Docker Compose</i></li> </ul>
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	<ul style="list-style-type: none"> <li>- A formatted request to create a particular agent, including any configuration settings that are specific to that agent</li> <li>- The definitions of the specified agent as obtained from the <i>Agent Definitions</i> module</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- The status or build logs associated with the construction of that agent</li> <li>- The (path to the copied) compiled and bundled agent itself</li> </ul>
Description	<p>The payload dispatch module is the primary “jumping point” when a build request for a new agent is received.</p> <p>When it receives a request for a particular agent to be built, it looks up the corresponding data for that agent in the <i>Agent Definitions</i> 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.</p> <p>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.</p>

## [11] Agent Definitions

Parent units	Server → Framework Interface → <b><i>Agent Definitions</i></b>
Interacts with or depends on	- Server → Payload Generation → <i>Payload Dispatch</i>
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	<p>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.</p> <p>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.</p>

[12] Docker Compose	
Parent units	Server → Payload Generation → <b><i>Docker Compose</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Payload Generation → <i>Payload Dispatch</i></li> <li>- Server → Payload Generation → <i>Docker Container</i></li> </ul>
Summary	Responsible for managing and instantiating the container (or containers) needed to generate the payloads for an agent.
Inputs	<ul style="list-style-type: none"> <li>- compose.yaml (which contains information about all of the containers associated with a particular application)</li> <li>- The individual Docker containers associated with a single agent (indirectly, through the YAML configuration file)</li> </ul>
Outputs	- None (it instantiates individual containers from images, but does not pass anything back to the payload dispatch module)
Description	<p>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.</p> <p>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</p>



	<p>designed to make it easier to move information and create connections between multiple containers, as well as defining these relationships in a portable way.</p> <p>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 <a href="#">Direct Forwarding</a> module within agents.</p>
--	---

[13] Docker Container	
Parent units	Server → Payload Generation → <b><i>Docker Container</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Server → Payload Generation → <i>Docker Compose</i></li> </ul>
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	<ul style="list-style-type: none"> <li>- Varies (typically some set of command-line options)</li> <li>- Dockerfile</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Log results (as dictated by the container itself)</li> <li>- A bundled agent, including the agent itself, its dependencies, and the binaries used to execute commands</li> </ul>
Description	<p>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.</p> <p>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:</p> <ul style="list-style-type: none"> <li>- Downloading specific versions of dependencies</li> <li>- Compiling binaries responsible for executing agent commands</li> <li>- Compiling the agent itself, if it must be compiled</li> <li>- Bundling the agent with its libraries, such as with <a href="#">PyInstaller</a></li> <li>- Applying obfuscation techniques, such as encrypting the agent and including a self-decryption routine</li> </ul> <p>An agent may use one or more Docker containers for its build process. Strictly speaking, it is worth noting that Docker containers are not <i>required</i> 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</p>



	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 → <i>Control Unit</i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → <i>Local Configuration</i></li> <li>- Agent → <i>Logging Module</i></li> <li>- Agent → <i>Task Broker</i></li> </ul>
Summary	Manages all decisionmaking for the agent; interprets and acts on messages received from the C2 server.
Inputs	<ul style="list-style-type: none"> <li>- Messages, command outputs, and forwarded messages received from the task broker</li> <li>- Locally stored configuration data</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- New tasks issued to task broker (which may be messages to the C2)</li> </ul>

	server, forwarded messages, or commands to be executed) - Unformatted log messages passed to logging module
Description	<p>The agent control unit serves as the decision-making body for anything that the agent does. Centralized decision-making is good for several reasons:</p> <ul style="list-style-type: none"> <li>- 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</li> <li>- Every single action <i>must</i> pass through the control unit, which implicitly forces every single action to be logged</li> <li>- How the agent reacts to messages from the server can easily be changed without affecting any other component</li> </ul> <p>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.</p>

[16] Local Configuration	
Parent units	Agent → <i>Local Configuration</i>
Interacts with or depends on	- Agent → <i>Control Unit</i>
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	<p>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.</p> <p>Much like the <a href="#">agent definitions</a>, 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.</p>

[17] Logging Module	
Parent units	Agent → <i>Logging Module</i>
Interacts with or	- Agent → <i>Control Unit</i>

depends on	- Agent → <i>Log Storage</i>
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 <i>could</i> be a SQL query)
Description	<p>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 <a href="#">logging</a> 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.</p> <p>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 <a href="#">command_request</a> messages, query the log storage accordingly, and then generate a <code>command_request</code> message.</p>

[18] Log Storage	
Parent units	Agent → <i>Log Storage</i>
Interacts with or depends on	- Agent → <i>Logging Module</i>
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	<p>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.</p> <p>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.</p>

[19] Task Broker	
Parent units	Agent → <i>Task Broker</i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → <i>Control Unit</i></li> <li>- Agent → Message Generation and Fetching → <i>Message Dispatch</i></li> <li>- Agent → Command Execution → <i>Command Dispatch</i></li> <li>- Agent → Direct Forwarding → <i>Forwarding Unit</i></li> </ul>
Summary	Responsible for executing and managing tasks and returning their results to the agent control unit.
Inputs	<ul style="list-style-type: none"> <li>- Depends on implementation; generally, a request for a task to execute</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Depends on implementation; could be the results of tasks or “futures” for tasks</li> </ul>
Description	<p>The purpose of the agent task broker is very similar to that of the <a href="#">task broker described for the server</a>. Because agents may be designed for arbitrary platforms with arbitrary libraries, there is no specific description of the agent task broker’s technologies.</p> <p>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.</p>

### 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 covert communication protocols.

[20] Message Dispatch	
Parent units	Agent → Message Generation/Fetching → <b>Message Dispatch</b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → <i>Task Broker</i></li> <li>- Agent → Message Generation and Fetching → Protocol Handler → <i>Control Unit</i></li> </ul>
Summary	Agent module identical in purpose to the <a href="#">Message Dispatch</a> module of the server.
Inputs	<ul style="list-style-type: none"> <li>- <i>When sending covert messages</i>, 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)</li> <li>- <i>When checking for new covert messages</i>, a request to check for new messages using a particular protocol at a specific location</li> <li>- <i>When used in real-time communication</i>, an arbitrary input (usually the stdout of a shell command)</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- <i>When sending covert messages</i>, 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</li> <li>- <i>When checking for new covert messages</i>, the reassembled plaintext message</li> <li>- <i>When used in real-time communication</i>, an arbitrary output (usually a shell command)</li> </ul>
Description	See <a href="#">[6] Message Dispatch</a> for more information.

[21] Control Unit	
Parent units	Agent → Message Generation/Fetching → Protocol Handler → <b>Control Unit</b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → Message Generation/Fetching → <i>Message Dispatch</i></li> <li>- Agent → Message Generation and Fetching → Protocol Handler → <i>Translation Unit</i></li> <li>- Agent → Message Generation and Fetching → Protocol Handler → <i>Networking Unit</i></li> </ul>
Summary	Agent module identical in purpose to the <i>Control Unit</i> module of the server.
Inputs	<ul style="list-style-type: none"> <li>- <i>When sending or receiving covert messages</i>, 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</li> </ul>

	dead drop, etc.) needed to encode or decode the message on an external service - <i>When used in real-time communication, an arbitrary input</i>
Outputs	- <i>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.</i> - <i>When receiving covert messages, the reassembled plaintext message.</i> - <i>When used in real-time communication, an arbitrary output</i>
Description	See <a href="#">[7] Control Unit</a> for more information.

### [22] Translation Unit

Parent units	Agent → Message Generation/Fetching → Protocol Handler → <b><i>Translation Unit</i></b>
Interacts with or depends on	- Agent → Message Generation/Fetching → Protocol Handler → <i>Control Unit</i>
Summary	Agent module identical in purpose to the <i>Translation Unit</i> 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 <a href="#">[8] Translation Unit</a> for more information.

### [23] Networking Unit

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

Outputs	<ul style="list-style-type: none"> <li>- <i>When sending messages</i>, the response from the service as a result of “placing” the dead drop</li> <li>- <i>When receiving messages</i>, the response from the service containing the information (which is expected to contain an encoded message or part of a message)</li> </ul>
Description	See <a href="#">[23] Networking Unit</a> 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 → <b><i>Command Dispatch</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → <i>Task Broker</i></li> <li>- Agent → Command Execution → <i>Command Binaries</i></li> </ul>
Summary	Responsible for interpreting requests for specific commands to be executed, generating the arguments and calling the associated binary for that command.
Inputs	<ul style="list-style-type: none"> <li>- An arbitrary request to execute a supported command with specific arguments</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- (to the command binaries) A specific command-line invocation of the binary specifically designed to execute that command</li> <li>- (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</li> </ul>



Description	<p>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.</p> <p>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.</p> <p>At a high level, the command dispatch module must:</p> <ul style="list-style-type: none"> <li>- Interpret a request for a command to be executed, along with any additional data included with the request (which is likely a <a href="#">command request</a> message)</li> <li>- Search for the binary (or binaries) that are responsible for carrying out that command</li> <li>- Issue calls to those binaries based on the request data, adding arguments as needed</li> <li>- Retrieve the outputs (whether written to stderr, stdout, or external storage) and verify that the outputs follow the expected format</li> <li>- Bundle the outputs into a single <code>command_response</code> message and pass it back to the agent control unit (via the task broker)</li> </ul>
-------------	--

[25] Command Binaries	
Parent units	Agent → Command Execution → <i><b>Command Binaries</b></i>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → Command Execution → <i>Command Dispatch</i></li> </ul>
Summary	Standalone binaries that accept command-line arguments and are designed to achieve a single task on a device.
Inputs	<ul style="list-style-type: none"> <li>- Command-line arguments</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Varies greatly; may be new files placed on the device's filesystem, standard output, and so on</li> </ul>
Description	<p>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 <a href="#">Mimikatz</a>, which searches for credentials in volatile memory.</p> <p>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.</p>

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 → <b><i>Forwarding Unit</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → <i>Task Broker</i></li> <li>- Agent → Direct Forwarding → <i>Networking Unit</i></li> </ul>
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	<ul style="list-style-type: none"> <li>- A plaintext message adhering to the <a href="#">standard message format</a></li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- Configuration information necessary for the networking unit to connect to the agent to forward messages to</li> <li>- The plaintext message with any additional metadata attached as needed</li> </ul>
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.

	<p>For example, if a request to forward data to a particular agent via netcat is requested, the forwarding unit likely should:</p> <ul style="list-style-type: none"> <li>- Verify (via the networking unit) that the target agent is reachable</li> <li>- Encode the message as Base64</li> <li>- Pass the base64 message into the networking unit, which will handle the actual process of connecting to the agent and sending the message</li> </ul> <p>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.</p>
--	--

[27] Networking Unit	
Parent units	Agent → Direct Forwarding → <b><i>Networking Unit</i></b>
Interacts with or depends on	<ul style="list-style-type: none"> <li>- Agent → Direct Forwarding → <i>Forwarding Unit</i></li> <li>- (Another agent instance, if present)</li> </ul>
Summary	Responsible for implementing and using the actual transport/application layer protocol needed to interact with another agent.
Inputs	<ul style="list-style-type: none"> <li>- Configuration information and message data from the forwarding unit</li> </ul>
Outputs	<ul style="list-style-type: none"> <li>- A plaintext message sent to or received from another agent</li> </ul>
Description	<p>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.</p> <p>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.</p>

Primary Data StructuresDatabase 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 expect to continue 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 is unclear, **the "id" field is always the primary key of each model**. When no primary key is explicitly declared in a model, an autoincrementing "id" field is added by Django by default.

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 uuid
import datetime

from django.contrib.auth.models import User, Group
from django.db import models
from django.urls import reverse
from django.conf import settings
```

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 is 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):
    name = models.CharField(
        max_length=100, unique=True, help_text="Human-readable name for the
agent."
    )
    # Local path to agent definition root
    definition_path = models.FileField(upload_to="agents")

```

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):
    # Human-readable name
    name = models.CharField(
        max_length=100, unique=True, help_text="Human-readable name for the
protocol."
    )
    # Local path to protocol handler binary (may make sense as a
FileField?)
    handler_path = models.FileField(upload_to="protocols")
```

Protocol example:

id	name	handler_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.

```
id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
# Human-readable name, device hostname, and remote address
name = models.CharField(max_length=100)
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,
related_name="endpoints")
protocols = models.ManyToManyField(
    Protocol, related_name="endpoints"
```

```

)
# 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?
# FIXME: this may be wrong according to
https://stackoverflow.com/questions/39821723/django-rest-framework-many-to-many-field-related-to-itself
connections = models.ManyToManyField("self", blank=True, null=True)

```

Endpoint example:

id	name	hostname	address	is_virtual	agent	protocols	encryption_key	hmac_key	agent_cfg	connections
(UUID)	test	PC1	192.168.0.1	false	1	1,2	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):
    # 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(settings.AUTH_USER_MODEL,
on_delete=models.PROTECT, related_name="tasks", blank=True, null=True)
    endpoint = models.ForeignKey(
        Endpoint, on_delete=models.PROTECT, related_name="tasks",
        blank=True, null=True
    )
    # When was this task started/finished?
    start_time = models.DateTimeField()
    end_time = models.DateTimeField(blank=True, null=True)
    # Is the task still in progress? What was the task? (How do we tie this
    # to Celery?)
    in_progress = models.BooleanField()

```

```

    # Arbitrary data that may be associated with the task. Should be
    plaintext.
    data = models.TextField(blank=True, null=True)

class TaskResult(models.Model):
    # What task is this result associated with?
    task = models.ForeignKey(Task, on_delete=models.PROTECT,
related_name="results")
    # Arbitrary data field; may contain files, a raw message, etc. (Note
    that
    # if a file is present, it *should* be stored with File and not this
    field.)
    data = models.BinaryField(blank=True, null=True)
    # Timestamp. May or may not be different from the end time of the task.
    timestamp = models.DateTimeField(blank=True, null=True)

```

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	timestamp
1	2	b{"status": "OK"}	1700370225

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 Credential(models.Model):
    # Task responsible for creating this credential entry, if any
    task = models.ForeignKey(
        Task,
        on_delete=models.PROTECT,
        blank=True,
        null=True,

```



```

        related_name="credentials",
    )
    # 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(max_length=32)

    # 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=255)

    # When this credential becomes invalid
    expiry = models.DateTimeField(blank=True, null=True)

class File(models.Model):
    # Task responsible for creating this credential entry, if any
    task = models.ForeignKey(
        Task, on_delete=models.PROTECT, blank=True, null=True,
        related_name="files"
    )
    # Path to file; location to be determined
    file = models.FileField(upload_to="files")

    def get_absolute_url(self):
        return reverse("file-detail", args=[str(self.id)])

    def __str__(self):
        return self.file # does this work?

```

*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):
    # Is the log tied to a specific endpoint? (If blank, it's assumed to
    # be
    # the server)
    source = models.ForeignKey(
        Endpoint, blank=True, null=True, on_delete=models.PROTECT,
        related_name="logs"
    )
    # Is the log tied to a user?
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL, blank=True, null=True,
        on_delete=models.PROTECT, related_name="logs"
    )
    # Is the log tied to a specific task?
    task = models.ForeignKey(
        Task, blank=True, null=True, on_delete=models.PROTECT,
        related_name="logs"
    )
    # Is the log tied to a single task result object?
    task_result = models.ForeignKey(
        TaskResult, blank=True, null=True, on_delete=models.PROTECT,
        related_name="logs"
    )
    # 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(max_length=32, blank=True, null=True)
    # Enum field, levels are likely to be 0 - 5
    level = models.IntegerField(blank=True, null=True)
    # 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	source	user	task	task_result	category	level	timestamp	data
----	--------	------	------	-------------	----------	-------	-----------	------

1	2	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": "<DeadDropMessageType>",
  "message_id": "00000000-0000-0000-0000-000000000000",
  "user_id": "00000000-0000-0000-0000-000000000000",
  "source_id": "00000000-0000-0000-0000-000000000000",
  "timestamp": 000000000,
  "digest":
    "f2ca1bb6c7e907d06dafa4687e579fce76b37e4e93b7605022da52e6ccc26fd2",
  "payload": ...,
}
```

The “message\_type” 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
- `heartbeat`, a message used to pass diagnostic information to the server and assert that the server can still be reached
- `init_message`, an arbitrary message used either in registration (i.e. the first time the agent reaches out to the server)

Note that this has changed since the fall, when a dedicated message type for file and credential uploads existed. We have opted to delegate this to the payload of a command response message.

The “message\_id” field is simply a UUID representing this message. Provided all agents and the server uses UUIDv4 on generation (a truly random UUID, not based on the time and input data), it is astronomically unlikely that two UUIDs will ever collide.

The “user\_id” 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 or a null UUID (all zeros).

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 “timestamp” is simply the UNIX timestamp of the message. The timestamp typically reflects the time of completion of the task that caused the message to be sent; it is not necessarily the time that the message was sent.

The “digest” field is a keyed SHA-256 digest of the plaintext message contents, used to assert the integrity of the message. It is signed with either an HMAC key or the private key of the originating client, depending on the cryptographic primitives supported by the agent and server. (Currently, the plan is to use an HMAC key for our current implementations.)

The “payload” field is a variable-content field containing a JSON dictionary of arbitrary keys to values. The structure of this 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, possibly to be passed to a command renderer
- For “heartbeat” and “init\_message”:
  - Any diagnostic data that may optionally be supplied by the agent

Finally, a “forwarding” field is planned to contain forwarding information, in plaintext. 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. Because each agent may

encrypt its own messages, this causes layers of encryption to be applied to the message multiple times. This is currently in development.

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 contains 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* (also referred to as *renderers*), 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 purpose 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.

For Pygin, the sole agent currently in development, these are *implemented* as subclasses of several Python abstract base classes that express commands, protocols, and other core DeadDrop objects. However, because agents can be implemented in any language or platform so long as they adhere to the DeadDrop interfaces, agents must “expose” this information in a set of standardized JSON files containing the information above. In the case of Pygin, this is achieved by parsing the Python source code with a separate utility to generate these JSON documents for the server as needed.

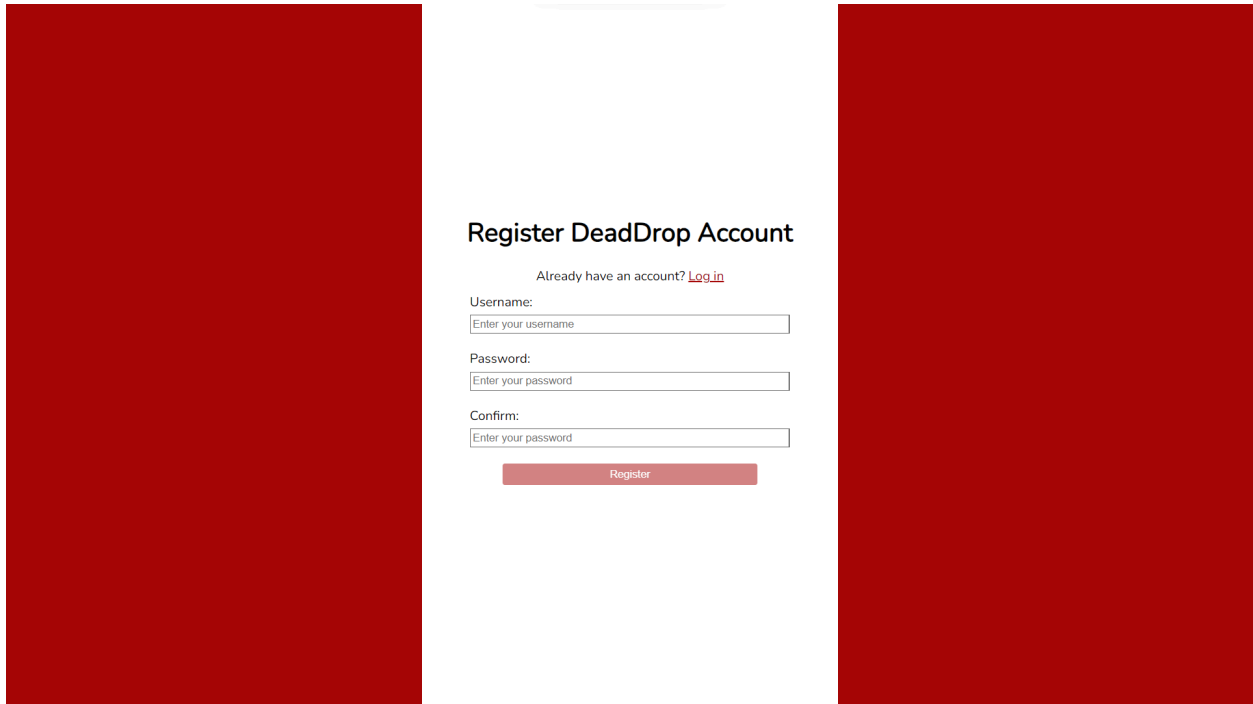
### C. Updated User Interface Design

As mentioned before, the Svelte web interface interacts with a Django backend using RESTful API calls; it serves as the “standard” way of interacting with DeadDrop as a whole, but is not tightly bound to Django. Although users of modern C2 frameworks expect a web interface, DeadDrop also exposes several APIs to allow users to manually develop their own interfaces or scripts.

The following images all demonstrate the web interface used to interact with DeadDrop, allowing the user to initiate actions and view internal results in a graphic manner.



*Sign in page to DeadDrop featuring artwork and a link to sign up. The sign in button is disabled until both username and password have an entry in their field; clicking on the button executes a POST request initiated by Svelte that is then passed to the associated Django endpoint.*



**Register DeadDrop Account**

Already have an account? [Log in](#)

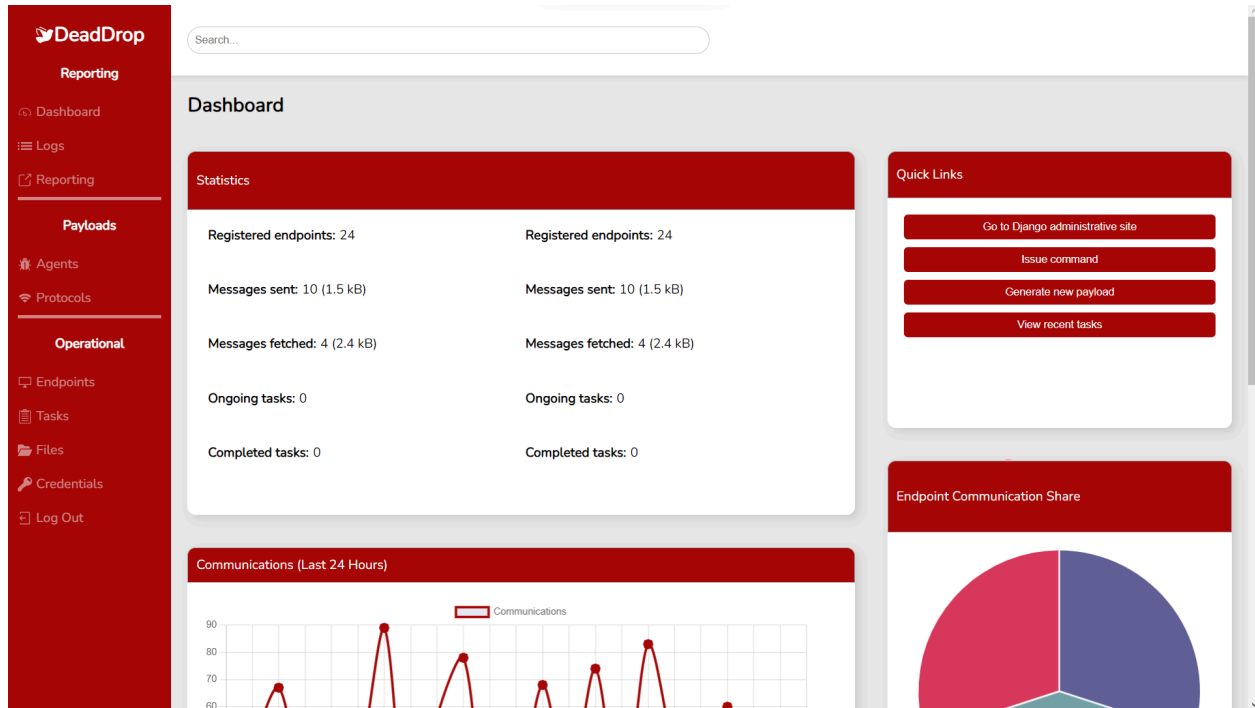
Username:

Password:

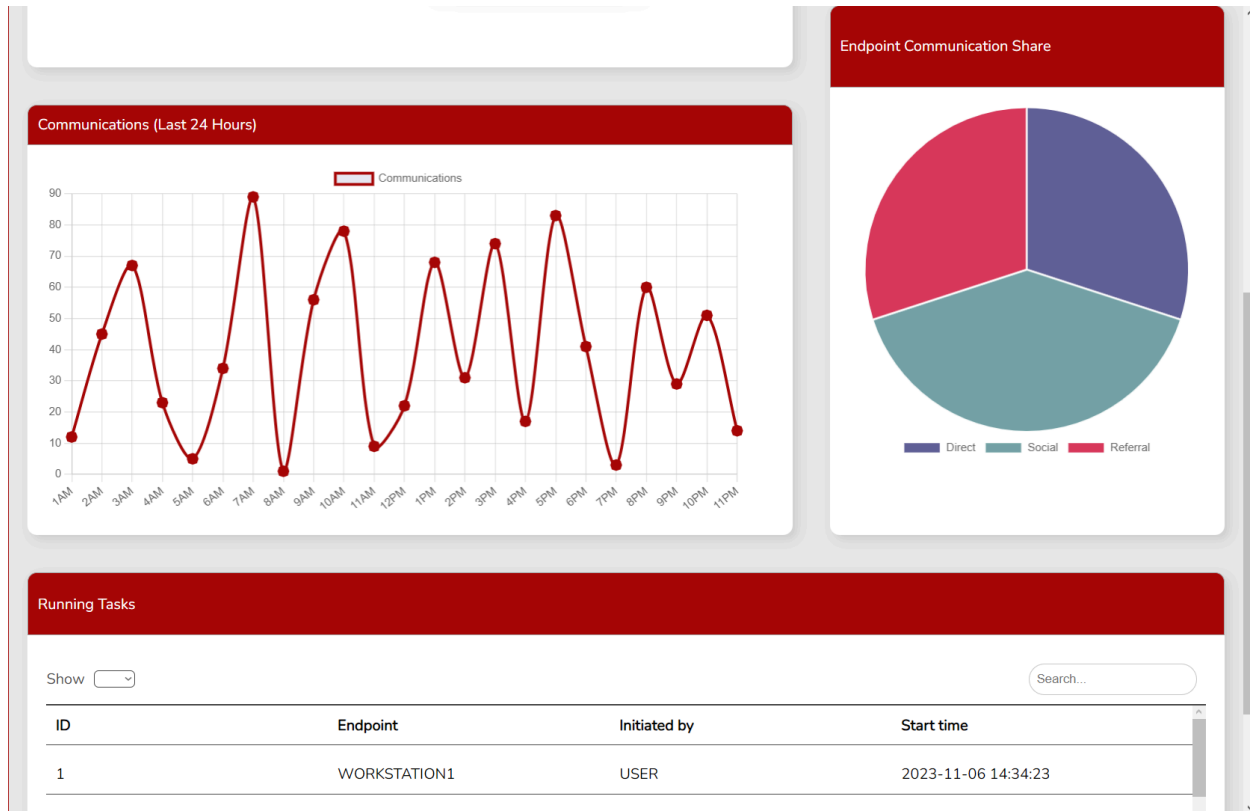
Confirm:

*Sign up page for DeadDrop. Client-side checks assert that the username and password fit Django requirements, disabling the button until they are met. Clicking the button similarly initiates a request to Django, which may respond with an error message (such as the username being taken) that is re-rendered by the web interface.*





*Side Panel and upper half of dashboard page. Sidebar icons light up upon hover and can navigate to any page on the server. Dashboard page presents at-a-glance graphs and statistics for monitoring agent performance. Quick Links menu provides easy access to common actions for users on the server.*



*Dashboard continued. Displays a summary view of endpoint communications over the past 24 hours and distribution communications by endpoint type. Uses Chart.js for rendering graphical information; the table was hand-built out of personal interest.*

**DeadDrop**

**Reporting**

- Dashboard
- Logs
- Reporting

**Payloads**

- Agents
- Protocols

**Operational**

- Endpoints
- Tasks
- Files
- Credentials
- Log Out

### Issue Command

#### Standard Options

Target Agent:

Protocol:

Arguments:

[Export Commands](#)

#### Available Commands

Select a command to issue. You can view the command reference by selecting the target endpoint and then using the Command Reference below.

- ☐ Protocol 1
- ☐ Protocol 2
- ☐ Protocol 3
- ☐ Protocol 4
- ☐ Protocol 5
- ☐ Protocol 6

#### Protocol Options

YouTube Configuration

Randomization Options

Encoding Options

#### Command Reference

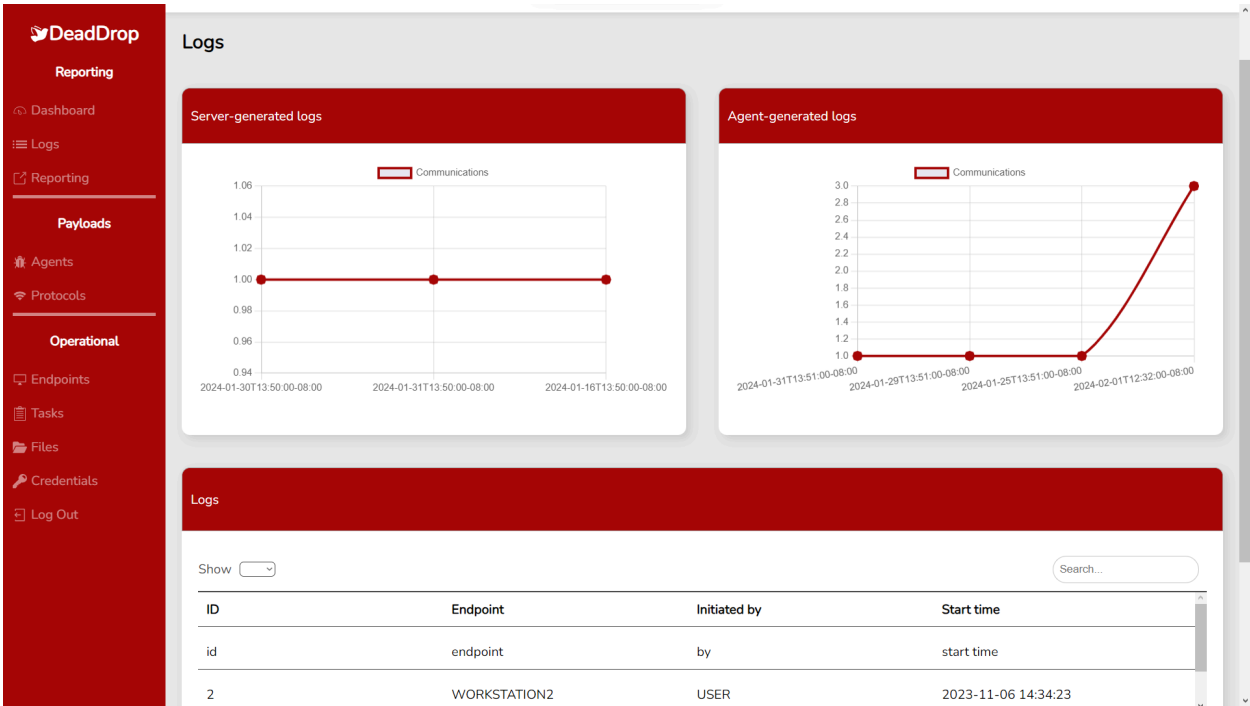
*Issue command page for executing commands on an infiltrated system through a target agent using a specified protocol. Form elements are generated dynamically based on the agent and protocol specified at the time this page is requested; this is not currently implemented, though the general layout of the form is reflective of the final design.*

The screenshot displays the DeadDrop web application interface. On the left is a red sidebar with the 'DeadDrop' logo and a navigation menu. The menu is organized into three sections: 'Reporting' (containing Dashboard, Logs, and Reporting), 'Payloads' (containing Agents and Protocols), and 'Operational' (containing Endpoints, Tasks, Files, Credentials, and Log Out). The main content area is white and features a top section with a text input field containing 't', an 'Arguments:' label, another text input field containing 't', and an 'Export Commands' button. To the right of this is a list of checkboxes for Protocol 3, Protocol 4, Protocol 5, and Protocol 6. Below these are two large red header sections. The first, 'Protocol Options', contains three white accordion-style boxes labeled 'YouTube Configuration', 'Randomization Options', and 'Encoding Options'. The second, 'Command Reference', contains a white table with three rows: 'ls', 'calculator', and 'dump\_creds'.

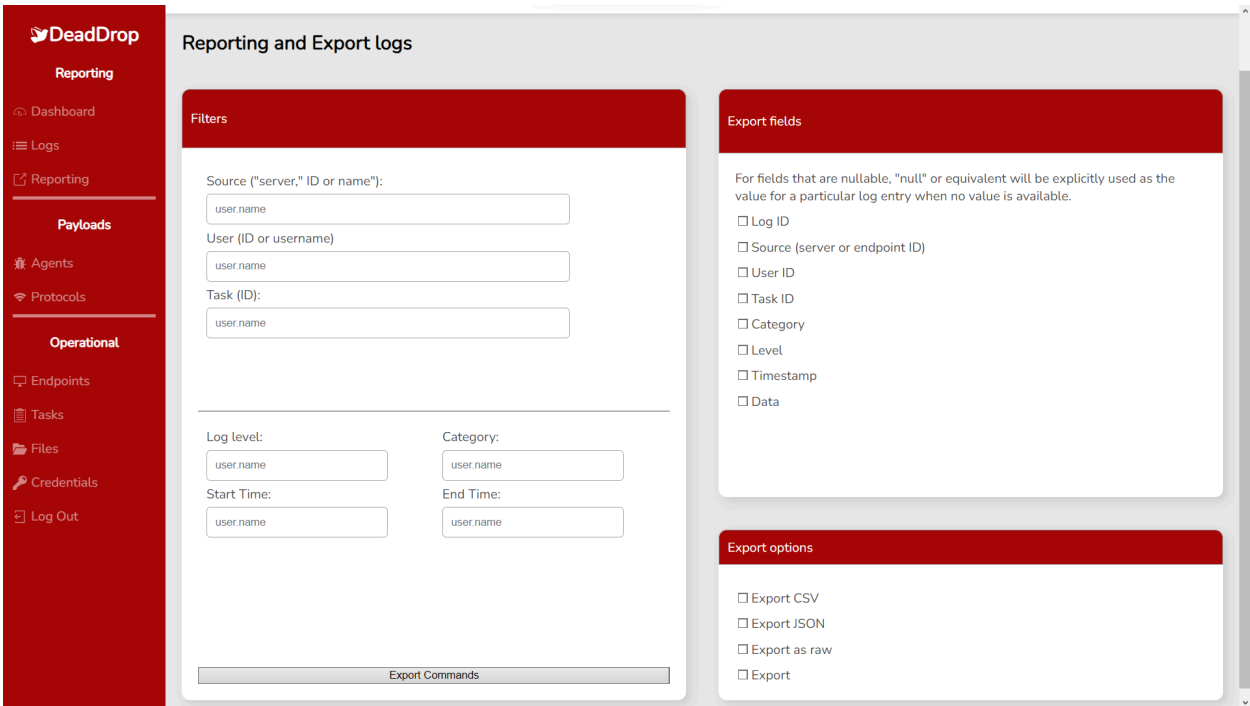
Protocol Options	
	YouTube Configuration
	Randomization Options
	Encoding Options

Command Reference	
	ls
	calculator
	dump_creds

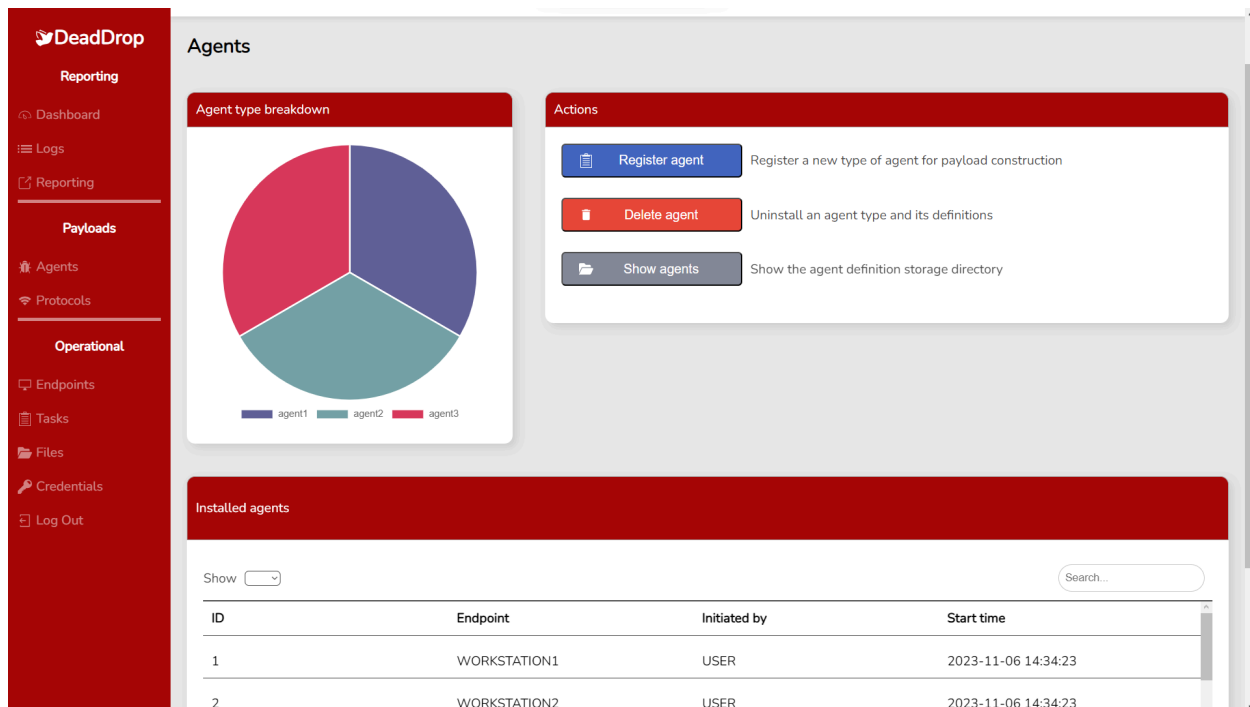
*References for protocol options and commands stored in individual accordion elements. These provide the user with easy access to protocol and agent documentation, which is required to be exposed in a standardized manner by protocols and agents. This allows rich text content (and possible code blocks and images) to be easily viewable.*



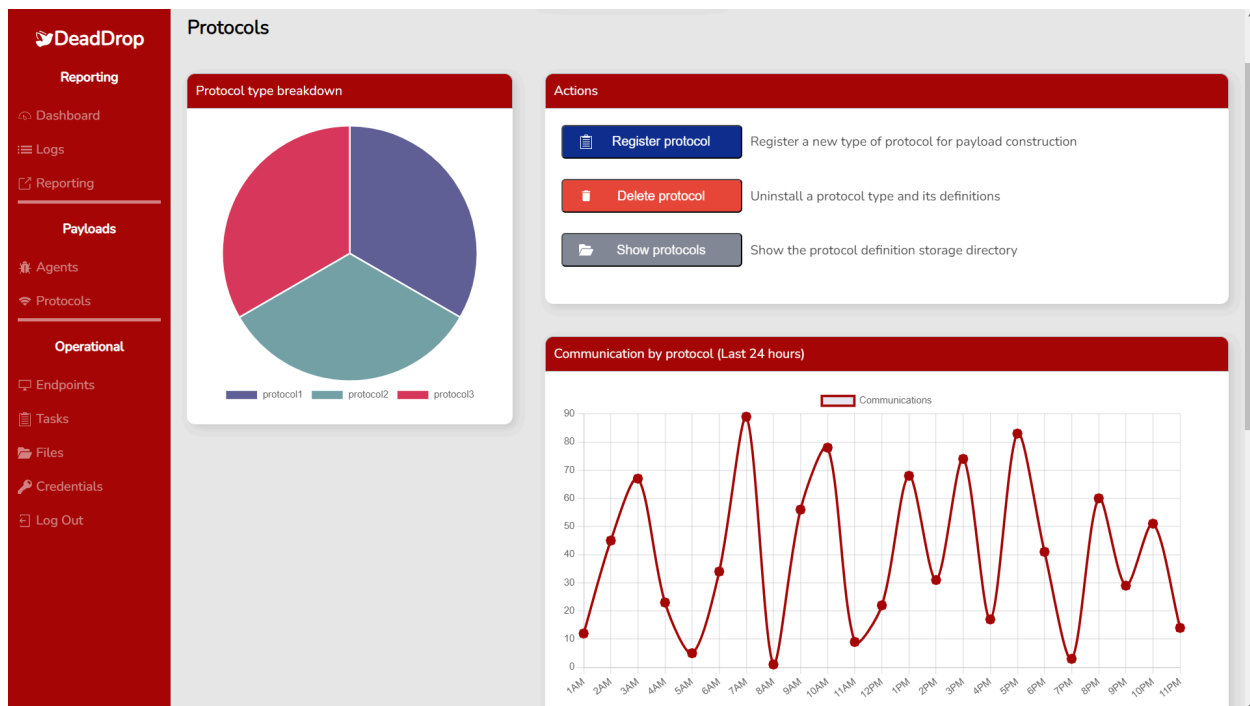
Log page displaying an overview server-generated versus agent generated logs, as well as a placeholder table for recent information. Server-side pagination is used to calculate statistics and display table entries.



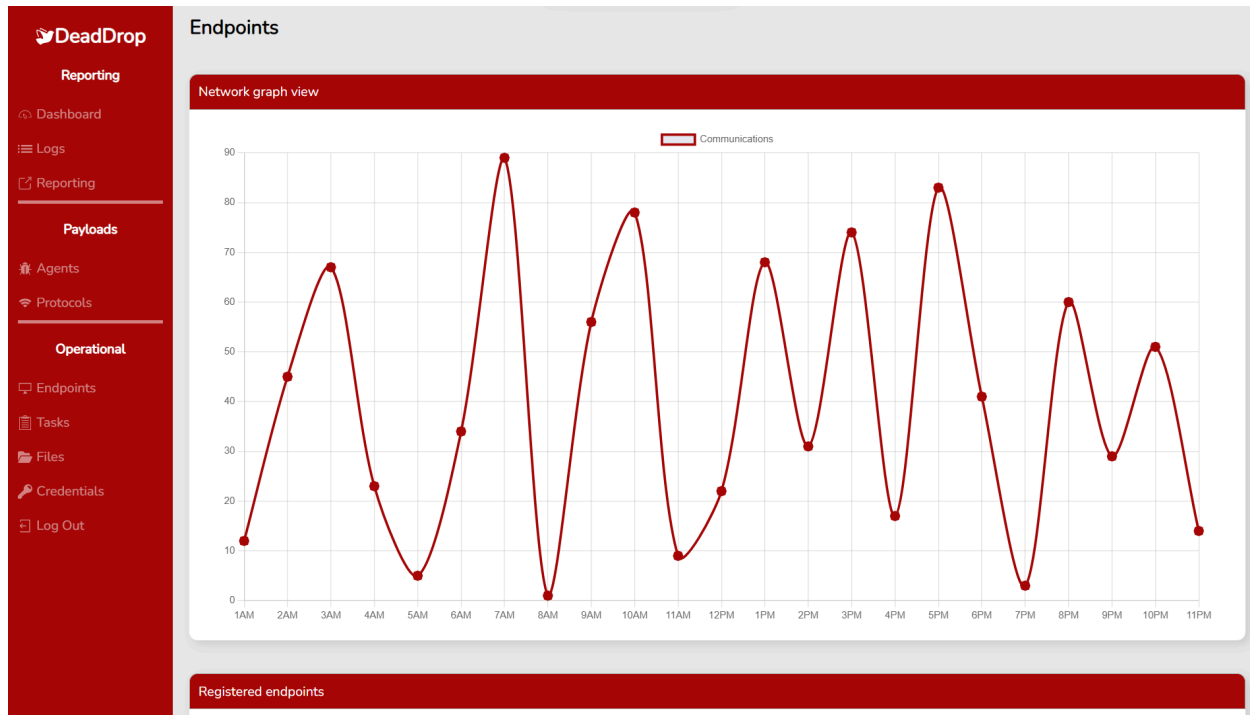
Reporting page for manually exporting logs through the server. These are simply passed as JSON to the Django backend, which sanitizes and parses the fields before executing the relevant filters specified by the user.



*Agent breakdown displaying the distribution of deployed agent types. Action tab buttons darken on hover, and allow users to download, delete, and manipulate new agent types.*



*Protocol breakdown displaying the distribution of deployed protocols. It is functionally and visually similar to the agents page, in large part because the internal package manager treats them as similar objects.*



*Endpoints page summarizing the number of communications over the last 24 hours, broken down by endpoint time (of which only one is shown here). Not shown is a table detailing individual endpoints. Other statistics may be added based on stakeholder feedback.*

**DeadDrop**

**Reporting**

- Dashboard
- Logs
- Reporting

**Payloads**

- Agents
- Protocols

**Operational**

- Endpoints
- Tasks
- Files
- Credentials
- Log Out

### Tasks

Running Tasks  
13

Open Callbacks  
12

Total Tasks  
111

Completed Tasks  
85

Running Tasks

Open Callbacks

Periodic Tasks

All tasks

Show  Search...

ID	Endpoint	Initiated by	Start time
1	WORKSTATION1	USER	2023-11-06 14:34:23
2	WORKSTATION2	USER	2023-11-06 14:34:23

Showing items 1-3 of 3

<< 1 2 3 >>

*Tasks page displays the number of asynchronous tasks being executed by agents, as well as their current status. This page covers both server tasks (being executed by Celery within the Django backend) as well as “callbacks”, which are tasks being executed by agents that have yet to result in a visible response at the dead drop location.*



## V. Updated Glossary of Terms

#	Term	Definition
1	advanced persistent threat	Refers to well-funded attackers (threat actors), often nationally-backed groups, that engage in stealthy, complex attacks. They may be undetected for a long period of time, and often target national industries. Frequently referred to as an APT.
2	antivirus evasion	Any malware mechanism that allows the malware to go undetected by antivirus solutions such as Windows Defender. This is often achieved through <b>polymorphic code</b> .
3	attack surface	Refers to all of the possible points where an attacker can attempt to <b>exploit</b> a potential <b>vulnerability</b> . For many organizations, this can include the company website, any public-facing servers, and individual user machines.
4	beacon(ing)	Commonly refers to the process of a <b>C2 agent</b> communicating with its associated <b>C2 server</b> . In a strict sense, it refers to the agent periodically reaching out to the server for new commands.
5	black box testing	Refers to penetration tests in which the attackers have virtually no knowledge about the organization or do not have any existing entry points into the organization. This better simulates real-world attacks, but makes it more difficult to discover internal vulnerabilities.
6	blue team	Collective term used to refer to cybersecurity professionals that defend a (particular organization's) network. Contrasts with the <b>red team</b> .
7	C2 agent	A piece of software ( <b>malware</b> ) installed on a device that allows an attacker to remotely control that device as part of a <b>C2 framework</b> . It may execute arbitrary commands and forward messages to other agents installed in a network.
8	C2 framework	Software used to administer and organize penetration tests (and illegal cyber attacks). Comprised of a <b>server</b> hosted by the attackers and one or more <b>agents</b> planted on target devices, such that the server communicates in a standardized manner with the agents.
9	C2 server	Software used by an attacker as part of a <b>C2 framework</b> to administer and organize a penetration test. It is primarily used to compile, distribute, and communicate with <b>agents</b> installed on remote devices.
10	Common Vulnerabilities and Exposures (CVE)	A free, public <b>vulnerability</b> database often used by organizations to track and address vulnerabilities in software used by the organization. Integrates with many other platforms and resources for testing older platforms and identifying vulnerable software.

#	Term	Definition
11	Cyber Kill Chain	Attack framework developed by Lockheed Martin that describes the typical “life cycle” of an attack, detailing the steps that attackers complete to <b>exfiltrate</b> data out of a network.
12	domain	In this context, refers to a publicly accessible network that is identified by a unique name. Attackers will typically set up new domains to host a <b>C2 server</b> ; traffic to the new domain may be suspicious due to the longevity of most legitimate domains.
13	endpoint security	Software that contributes to the security of individual devices on an organization’s network. Common endpoint security solutions include antivirus (Microsoft Defender), live threat analysis (CrowdStrike), and device log collection (osquery).
14	exfiltration	The act of taking sensitive information outside of an organization’s network (where it is intended to be kept secure). The goal of most cyber attacks is to exfiltrate valuable data and either sell it or hold it for ransom.
15	(to) exploit	The act of abusing a <b>vulnerability</b> (flaw) in software to cause it to behave in an unintended manner. Can be used to install malware on a vulnerable device.
16	intrusion detection system	Refers to any hardware or software that monitors a network for malicious activity (usually indicating the presence of an attacker within a network).
17	malware	<p>Malicious code or software that executes on a computer typically without the user’s knowledge, performing arbitrary actions as designed by the attacker. Typically, malware is delivered by <b>exploiting a vulnerability</b> in one or more systems.</p> <p>In this document, malware is generally interchangeable with <b>C2 agent</b>.</p>
18	MITRE ATT&CK	A publicly available set of models that detail common attack techniques used by <b>threat actors</b> that have been observed “in the wild”. Sponsored by the MITRE Corporation.
19	payload	Broadly refers to any malicious code that executes a specific action on a target system; payloads may be delivered through a variety of means (email, SSH, etc.), and may take a variety of forms (a Word doc, an executable, a shell script, etc.). Common payloads allow attackers to run arbitrary commands on a target or install malware.
20	persistence	The quality of a running program (typically malware) to survive a restart of the machine. Typically, these programs leverage operating system features to allow the program to be run again on startup. In

#	Term	Definition
		the case of malware, this also implies that it can survive a restart of the machine without detection.
21	polymorphic code	Any code (or software) that is capable of changing the instructions that it executes without changing its functionality. By doing so, the software is able to obfuscate malicious code that might already be known to an antivirus solution, contributing to <b><i>antivirus evasion</i></b> . Sometimes achieved through decrypting instructions at runtime.
22	red team	Collective term used to refer to cybersecurity professionals associated with attacking a network and its services and identifying gaps in an organization's security. Contrasts with the <b><i>blue team</i></b> .
23	threat actor	Refers to any individual or group that engages in malicious activity (and represents a risk to a particular organization). Threat actors vary greatly in skill and funding.
24	traffic mirroring	The act of inspecting, copying, storing all (unencrypted) traffic that travels in and out of an organization's network. Often used in both forensic investigations (i.e. after an attack has occurred) and proactive identification of suspicious activity.
25	vulnerability	Commonly used to refer to any flaw in a piece of software that can be used to cause the software to behave in an unintended manner. May also refer to any general weakness in computer systems, procedures, or configurations that could be used for malicious gain.
26	white box testing	Refers to penetration tests in which the testers have complete access to and knowledge of an organization's software and network. Typically, this includes network diagrams and application source code. These are more likely to discover internal vulnerabilities, but often do not accurately represent an external attacker's knowledge.

## VI. Engineering Standards and Technologies

DeadDrop leverages a variety of technologies to meet modern penetration testing needs, allowing DeadDrop to adhere to industry-recognized standards and ensuring it meets the expectations of professionals.

### A. Standards

#### PTES

The PTES Technical Guidelines outline a systematic approach to penetration testing. The process includes defining engagement parameters, gathering intelligence, modeling threats, analyzing vulnerabilities, actively exploiting weaknesses, assessing post-exploitation capabilities, and documenting findings in a comprehensive report. Following remediation efforts, a final report is generated, and documentation is archived for future reference, ensuring a structured and ethical penetration testing procedure to enhance overall security.

Penetration testing comes with many risks that must be minimized or avoided. By following the Penetration Testing Execution Standard DeadDrop is able to provide a familiar standard for professionals using this software and reduce the amount of risk inherent in this field.

#### WCAG 2.0

The Web Content Accessibility Guidelines (WCAG) 2.0 provide a comprehensive framework for creating accessible web content. It emphasizes principles of perceivability, operability, understandability, and robustness to ensure that digital information and services are accessible to individuals with disabilities. WCAG 2.0 offers guidelines and success criteria, addressing various aspects such as text alternatives, keyboard navigation, time-based media, and adaptable content. The goal is to make web content more inclusive and usable for a diverse audience, promoting equal access and usability across different devices and user needs.

### B. Technologies

#### Docker

Docker is a platform that enables developers to automate the deployment of applications inside lightweight, portable containers. Containers are standalone executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools. Docker uses containerization technology to encapsulate applications and their dependencies, ensuring consistency across different environments.

In DeadDrop, we use Docker during both development and runtime. During development we use Docker so that our builds are reproducible and we are working in a standard environment. This also has the added benefit in that each member of our team is able to easily switch to and run a section of the project that they are not familiar with with minimal setup time. During runtime, Docker is used to build agents in their target environment. This allows the agents to avoid compatibility issues once they are deployed on the target machine.

DeadDrop's front end is a web based application. We are adhering to the WCAG 2.0 standard to ensure that our application is accessible to the largest range of users.

### Svelte

Svelte is a JavaScript framework designed for building user interfaces with a unique and compiler-based approach. It shifts much of the work from runtime to build time, resulting in highly optimized JavaScript code during compilation. Unlike traditional frameworks, Svelte uses a declarative syntax, making it easy for developers to describe the desired outcome of components. It embraces reactivity, allowing for automatic updates when state variables change. Not relying on a virtual DOM, Svelte directly manipulates the DOM at build time, contributing to smaller and more efficient bundles. With smaller bundle sizes, faster load times, and a framework-agnostic nature, Svelte offers an innovative and easy-to-learn solution for creating modern web applications.

Svelte is the frontend framework for the DeadDrop project. Svelte provides templating for interacting with the services provided by the server.

### Django

Django is a high-level web framework for building robust and scalable web applications using the Python programming language. It follows the model-view-controller (MVC) architectural pattern and provides a set of tools and conventions to facilitate rapid development. Django includes an ORM (Object-Relational Mapping) system for database interactions, a templating engine for designing user interfaces, and a built-in administrative panel for managing application data. It emphasizes the DRY (Don't Repeat Yourself) and convention over configuration principles, promoting code reusability and reducing redundancy. Django's batteries-included philosophy means it comes with many built-in features such as authentication, URL routing, and form handling, allowing developers to focus on building application-specific features. The framework also encourages best practices in terms of security, scalability, and maintainability, making it a popular choice for developers building a wide range of web applications.

DeadDrop's backend server utilizes Django to serve front end web pages, act as an API for controlling and retrieving data from the running system.

### ffmpeg

FFmpeg is a powerful open-source software suite designed for handling multimedia data, including audio and video files. It provides a comprehensive set of libraries and tools for encoding, decoding, transcoding, streaming, and manipulating multimedia content. FFmpeg is a command-line tool, allowing users to perform a wide range of tasks, such as converting between different audio and video formats, adjusting video quality, cutting or joining media files, and streaming content over networks.

DeadDrop's server and agents utilize the avcodec library of ffmpeg to manipulate video and images for some of the communication protocols.

## VII. Project Impact and Context Considerations

### A. Public Impact

DeadDrop offers substantial open-source value to the public as it aims to offer all of the features of major commercial and open-source C2 frameworks while introducing a unique approach to evading detection. Our mission is to provide organizations with the tools needed to defend against these attacks. Although DeadDrop lacks significant direct cultural or global impacts due to its cybersecurity-focused nature, its indirect impact on organizations can be immense. Hospitals, power grids, and other critical infrastructure have all been the target of cybersecurity attacks in the past, placing the lives of many at risk. Discovering the vulnerabilities that make these attacks possible through DeadDrop has the potential to save lives by keeping these services accessible.

As DeadDrop is free and open-source, it is likely to reduce the economic burden on some companies and corporations should they decide to use our framework. Furthermore, our existing code can be freely forked to develop an ecosystem of interchangeable packages (and even new servers) through the use of standardized interfaces with well-documented messaging formats.

However, the free and open-source licensing of DeadDrop makes the source code readily available to both well-intentioned and malicious actors. It is entirely possible for a malicious user to utilize or build off DeadDrop's code to exploit vulnerabilities in a system. This could have a social impact; an increase in cybercrime as a result of DeadDrop would indirectly lead to more awareness regarding data theft and espionage. (There are also the direct damages of data breaches and cyber attacks, as well, such as the loss of services or sensitive information.)

Addressing the potential for misuse, DeadDrop agents are constructed in an open manner, meaning that agents deployed by DeadDrop are not obfuscated and leave behind logs of all activities undertaken. That is, DeadDrop is aimed at evading network-level defenses, not evading endpoint defenses or causing destruction; agents can be easily found and reverse-engineered on devices. Furthermore, the repository doesn't come with any tools for infiltration – only the ability to execute commands from the server on an already infiltrated device. This leaves out the possibility for any “out-of-the-box” exploitation, increasing the difficulty of misusing the framework.

Though DeadDrop has a limited environmental impact, we believe that the novel (and perhaps “flashy”) nature of our project will raise awareness about modern security threats, allowing organizations to improve their cybersecurity practices and better protect the data of users. The inherent nature of DeadDrop, as both a free and open-source command and control framework intended for penetration testing, raises a host of legal and ethical concerns regarding the potential abuses of its functionality.

In particular, there are many concerns associated with simply testing and developing our communication protocols, even before their use in the general public. The protocols being developed for DeadDrop are intended to utilize large, whitelisted platforms such as Twitter or YouTube to bypass security measures and communicate covertly between the server and infected

devices. In turn, developing and testing these communication protocols involves using these platforms for unintended purposes, likely skirting or violating their terms of service. This does have legal implications for users of DeadDrop - not every organization will have the resources to request permission from the relevant platform owners, and therefore may not be able to use DeadDrop due to the legal risks involved.

Although DeadDrop is unlikely to have a significant impact on the health, safety, and welfare of the public in any context, cybersecurity as a whole has become a core focus of many organizations' concerns. There are direct consequences for failing to secure the data and availability of services - especially critical infrastructure - which DeadDrop can help mitigate through its effective use in penetration tests.

## **B. Accessibility**

The primary concerns regarding accessibility are largely tied to the web interface of the C2 server; all other interactable components of the framework involve source code development or API calls. For components outside of the web interface, their accessibility is dictated largely by the user's choice of program for software development.

Developing with web accessibility in mind is largely dictated by the Web Content Accessibility Guidelines (WCAG), which express these guidelines as functional and non-functional requirements. Applicable functional requirements for DeadDrop include keyboard navigation for interactable components, text alternatives for non-text content (such as figures), and focus indicators for keyboard-focused components, which are essential to the "mechanical" elements of accessibility - that is, the website's support for interacting with page elements. Non-functional requirements for DeadDrop include rigorous browser compatibility, high color contrast, and responsive design, which contribute to a seamless experience across different users. These are typically higher-level requirements that are not implemented on a page-by-page basis, but rather across the website as a whole.

As with most modern JavaScript frameworks, SvelteKit (the framework in use for the frontend), can leverage Accessible Rich Internet Applications (ARIA) attributes to enhance the accessibility of dynamic content and UI components. SvelteKit's ARIA features, in addition to openly available prebuilt components, help maintain a high degree of accessibility throughout our development cycle. Furthermore, we intend to use openly available accessibility analyzers, such as SiteImprove and Google Lighthouse, to identify potential accessibility issues and track compliance with specific items of the WCAG standards.

SiteImprove allows us to generate rich accessibility reports informing us of accessibility issues such as color contrast, the absence of alt text, and other WCAG-specific guidelines. Similarly, the Lighthouse extension creates a report on SEO performance and other important accessibility issues such as text contrast and mobile friendliness. The reports detail important metrics such as first contentful paint (FCP), largest contentful paint (LCP), total blocking time (TBT), cumulative layout



shift (CLS), and speed index (SI). These are immensely helpful during the development of individual pages, allowing us to catch potential accessibility (and usability) issues early on.

Another method that we will use is Lighthouse CI, which makes sure that our website is continuously meeting proper accessibility standards throughout every commit. Lighthouse CI is an automated tool to track any regressions that we may have on a page after finishing designing them. Furthermore, we will also utilize Lighthouse which runs the Lighthouse extension on all pages. It generates a consolidated report that makes it easy to analyze the overall accessibility and usability of our web app as a whole. This allows us to track accessibility throughout the website over time, allowing us to determine if feature changes could have an impact on site-wide accessibility (despite not changing the underlying page templates themselves).

To be explicit, we intend to use SiteImprove and Google Lighthouse for manually testing accessibility during the design of individual pages. Lighthouse CI and Unlighthouse will be used for automated regression testing on major commits.

From manual analysis and initial testing using these tools, we were able to identify three accessibility issues in the web interface that should be resolved in the near future. The largest priority is the absence of focus indicators for graph elements, which contain on-hover tooltips that are not accessible via keyboard navigation. Because these graph elements are the only way to view high-level aggregate statistics through the web interface, this makes important information inaccessible to non-visual users.

Additionally, we found that various icons and figures throughout the interface lacked descriptive alternative text. While some of these icons are decorative (despite not being marked as such - another accessibility issue), other icons convey meaningful information that is inaccessible to a non-visual user. Finally, certain focus indicators either do not function correctly or do not operate as expected; some focus indicators are hidden by other elements due to z-axis CSS rules, while others overlap with other elements on the page and are obscured.

## VIII. Updated List of References

*Indicate a related “problem domain” book, at least 3 reference articles (journal, conference or web scientific publications), and at least 3 project-related websites with useful resources, all with brief descriptions (40 to 70 words each).*

### A. Problem Domain Book

[Learn Penetration Testing](#) is an introductory book to both the technical and non-technical principles of penetration testing. It covers not only the specific techniques that attackers use to identify and exploit vulnerabilities, but also the overall “life cycle” of an attack. This is particularly important; although the novelty of this project comes from its technical implementation of a covert communication protocol, the project as a whole is only useful if it can be used to orchestrate an attack from start to finish.

### B. Reference Articles

Each of the publications listed below detail penetration testing at the organizational level; that is, rather than focus on specific technical approaches, these articles detail the general procedures and ethics that teams follow when conducting penetration tests in a variety of different environments. Additionally, these articles also discuss the general impact of penetration testing from a governance and risk perspective, as well as the “human” aspect of cybersecurity (such as social engineering).

- [An overview of vulnerability assessment and penetration testing techniques](#): This paper gives a general overview of both vulnerability assessments (where vulnerabilities are identified but not exploited) and penetration testing methodologies. It also covers many common vulnerabilities found in industry, detailing their frequency and severity. This can be used to guide the development of our agents and the backend, as it allows us to prioritize what offensive features to implement.
- [A study on penetration testing process and tools](#): This paper details the process for penetration tests against individual components in an organization, such as routers, endpoint devices, firewalls, applications, and more. It also covers the organizational process of starting a penetration test, including defining the rules of engagement, performing initial information gathering, and developing a usable technical and executive report. While DeadDrop focuses on post-exploitation, it is still important to recognize that a C2 framework should allow users to store information relevant to the full penetration test.
- [MTC2: A command and control framework for moving target defense and cyber resilience](#): This paper describes the architecture of a novel C2 framework suitable for attacking organizations using “Moving Target Defense”, or MTD. The idea behind MTD is that an organization’s landscape constantly changes, including the network layout itself, applications, the underlying data, and so on, such that an attacker must expend more effort to identify and exploit vulnerabilities. The architecture described by the authors can be adapted to DeadDrop, as the idea of both our project and MTC2 are similar - we want to circumvent robust, cutting-edge network defenses using novel techniques.

### C. Additional Resources

- The [Mythic C2 documentation](#) details payload and agent development for the Mythic C2 framework, which details Mythic's requirements for what new agents must implement. This is particularly helpful, as it helps inform us on what data is needed for communication between the server and multiple agents, as well as what their message format looks like. It also contains an extensive amount of general documentation on Mythic's architecture, components, and configuration..
- A [list of various open-source C2 frameworks](#) is available on GitHub, ranging from fully-fledged frameworks (like Mythic) to proofs of concept (usually just two scripts, one for the server and another for the agent). This allows us to evaluate what functionality is common among many different frameworks (and therefore desirable), as well as how other well-designed frameworks accept new functionality.
- The [Sliver C2 documentation](#) details both the architecture of Sliver as well as various other resources related to implementing specific penetration testing techniques (such as antivirus evasion) that are not covered in the Mythic documentation. This includes links to the [Veil Framework](#) and [PEzor](#), both of which are automated solutions to developing payloads that evade common antivirus solutions.

A [list of resources for implementing malware persistence](#) (that is, malware that survives a restart of the computer) and detecting and analyzing persistent malware techniques, is available on GitHub, along with a general overview of how persistence techniques work. This can be particularly helpful in implementing our C2 agents in a "real world" context; although VMs in the cloud might not restart often, real user machines do.

## IX. Time Worked

Note that the recorded time includes time spent performing research, writing code, and implementing the modules described in this document (since P1). This also includes meetings to discuss the architecture and design as a team, verifying that the systems are still reasonably designed and achievable.

Team Member	Hours	Sections Contributed
Jann Arellano	7	<ul style="list-style-type: none"> <li>- Project Impact and Context Considerations               <ul style="list-style-type: none"> <li>- Public Impact</li> <li>- Accessibility</li> </ul> </li> </ul>
Brian Buslon	11	<ul style="list-style-type: none"> <li>- Updated Design               <ul style="list-style-type: none"> <li>- Updated User Interface Design</li> </ul> </li> <li>- Project Impact and Context Considerations               <ul style="list-style-type: none"> <li>- Accessibility</li> </ul> </li> </ul>
Keaton Clark	8	<ul style="list-style-type: none"> <li>- Updated Specifications               <ul style="list-style-type: none"> <li>- Updated Use Case Modeling</li> </ul> </li> <li>- Engineering Standards and Technologies</li> <li>- Updated List of References</li> </ul>
Lloyd Gonzales	15	<ul style="list-style-type: none"> <li>- Abstract</li> <li>- Recent Project Changes</li> <li>- Updated Specifications               <ul style="list-style-type: none"> <li>- Summary of Changes</li> <li>- Updated Technical Requirements</li> </ul> </li> <li>- Updated Design               <ul style="list-style-type: none"> <li>- Summary of Changes in Project Design</li> <li>- Updated High- and Medium-level Design</li> </ul> </li> <li>- Updated Glossary of Terms</li> </ul>