DeadDrop

a clever command and control framework for penetration testers

Project Prototype (P4)

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

Advisor:
Shamik Sengupta
Professor
University of Nevada, Reno

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

Department of Computer Science and Engineering
University of Nevada, Reno
December 15, 2023

Table of Contents

Abstract

We have implemented a prototype for 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. As of our demonstration to the teaching team, we have completed a minimal implementation of each of the major modules in the framework, demonstrating the viability of our architecture and the project as a whole. In particular, we have successfully shown that messages can be transmitted entirely over YouTube, allowing us to execute arbitrary commands on a host adhering to our communication protocol.

I. Introduction

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

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

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

By abusing the "trust" behind large, well-known websites, DeadDrop provides security engineers with new insight into identifying covert attacker techniques and security weaknesses. In doing so, it encourages network security engineers to take new approaches to identifying covert communication methods that could be used by skilled threat actors.

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

Since Project Assignment 3, we have implemented the foundation for the framework as a whole, implementing the major components that provide the services needed for the framework to function. In particular, we have implemented a simple frontend, backend, and agent that leverage our communication protocol over YouTube, which allows us to send and receive commands across remote devices. We plan on integrating and extending these further by implementing asynchronous tasking and payload generation through the backend, completing individual pages on the frontend, implementing more protocols, and more.

Currently, no major changes to the design or requirements of the project have occurred as a result of the development of the prototype. While we have identified required low-level changes during the development of our prototype, such as the introduction of new Django models to support future development, the high-level architecture has remained the same since Project Assignment 3.

II. Prototype Objectives and Functionality

For the prototype, we aimed to make progress on the four fundamental parts of our project:

- the server frontend, which allows users to interact with the framework as a whole;
- the server backend, which abstracts message handling, payload generation, and various other operational activities from the users;
- the protocol handler, which allows us to use YouTube (or any other video platform) as a form of communication; and
- the agent, which provides the routines needed for users to communicate with and execute commands on a device through the server.

We did this with the goal of demonstrating that a complete workflow - where an arbitrary command issued through the frontend could be encoded into a video, automatically uploaded to YouTube, downloaded from a remote agent, decoded back into a JSON document, and used to trigger a shell command through the agent - was possible. This is the absolute minimum that a generic C2 framework *must* do; this workflow forms the backbone of everything that a C2 framework does. Once this backbone is complete, it becomes much easier to extend the framework with the features we had originally planned when developing our requirements. If we could complete such a proof of concept by demo day, we would be confident that the project was at least viable and that we had a solid foundation to build upon.

The remainder of this section describes each of the components implemented for the prototype.

Frontend

The frontend was developed with SvelteKit, a full-stack JavaScript framework for developing single page applications. It includes a host of convenient features including hot module reloading for development, built-in reactive state to display real-time changes, and route grouping with inherited layout and servers to protect routes from unauthenticated access.

The DeadDrop architecture utilizes two full-stack frameworks, SvelteKit and Django. Though perhaps an unusual server architecture, we wanted to leverage Svelte's frontend features and ecosystem where Django's Jinja templates did not suffice (in addition to personal interest). The majority of the components in use (with the exception of the graphs from Chart.js) were developed by hand; the use of a component library or CSS framework such as Tailwind and Bootstrap are planned, but was overlooked during initial development.

Backend

For the Django backend, we aimed to create a RESTful API using Django Rest Framework. This allows users to interact with the framework in arbitrary ways not provided by the Svelte frontend, making it trivial to implement simple Python scripts to automate internal operations without directly modifying the backend source.

Django Rest Framework allows us to quickly write serializers, making it possible to leverage the fetch API used in the Svelte backend to bring data into the Svelte frontend. This allows us to easily make arbitrary requests between Svelte and Django that adhere to the provided API endpoints. This adds to the modularity of the backend, as we can create more endpoints as needed without needing to completely overhaul the frontend. We have also defined the models that we believe are needed for the framework to function, and have implemented tokenization so that only authorized users are able to access sensitive information.

Communication Protocol

For the first of many communication protocols, we decided to implement a technique to encode raw bytes into video data and upload the resulting data to YouTube. This video can then be retrieved by the receiver of the message and decoded using the same technique to inspect the content sof the message.

The data is encoded by taking each bit from the input data and writing an n x n square of pixels to the output video with a white square corresponding to a binary 0 and a black square correlating to a binary 1. The size of the square is configurable so that we can look into optimizing the (de-)compression ratio and fighting YouTube's compression algorithm that may corrupt higher-density data streams.

Agent

For our prototype agent, we opted to implement a simple agent in Python, allowing us to focus on integrating agents into the framework in a platform-agnostic manner. Rather than focus on OS-specific, low-level functionality, such as dumping credentials from RAM or stealing session tokens, we wanted to focus on identifying what data needed to be passed between the server and any arbitrary agent. Our goal was to flesh out the structure of the JSON messages passed between the server and the agent – helping us identify what would be needed on both ends – after which we could focus on agent-specific functionality.

To implement asynchronous tasking, we used Redis (an in-memory database) and Celery (a real-time task queue implementation). This allows us to run multiple tasks and commands in parallel, while still providing a central module for decisionmaking and logging. Additionally, because agent instances are largely unsupervised, these libraries provide mechanisms for retrying commands, setting timeouts, and acting on failed commands, even if they raise an exception.

III. Prototype Development

The following section contains various screenshots across all the completed modules of the project. Some of these are not UI screenshots, but we feel that an overview of prototype development would be incomplete without them.

For clarity, a complete non-functional mockup of the frontend was developed to help determine what functionality would need to be provided by the backend, as well as what components would need to be developed in Svelte. The functional Svelte frontend remains as faithful to the mockup as possible but does not directly reuse any mockup components (which comes from an educational interest, as well as the goal of using existing Svelte components as much as possible).

The non-functional mockup can be found at https://deaddrop-v1.bss.design/dashboard.html; the source HTML files are also included in the zip folder submitted with this assignment, as well as the Bootstrap Studio project that was used to create it.

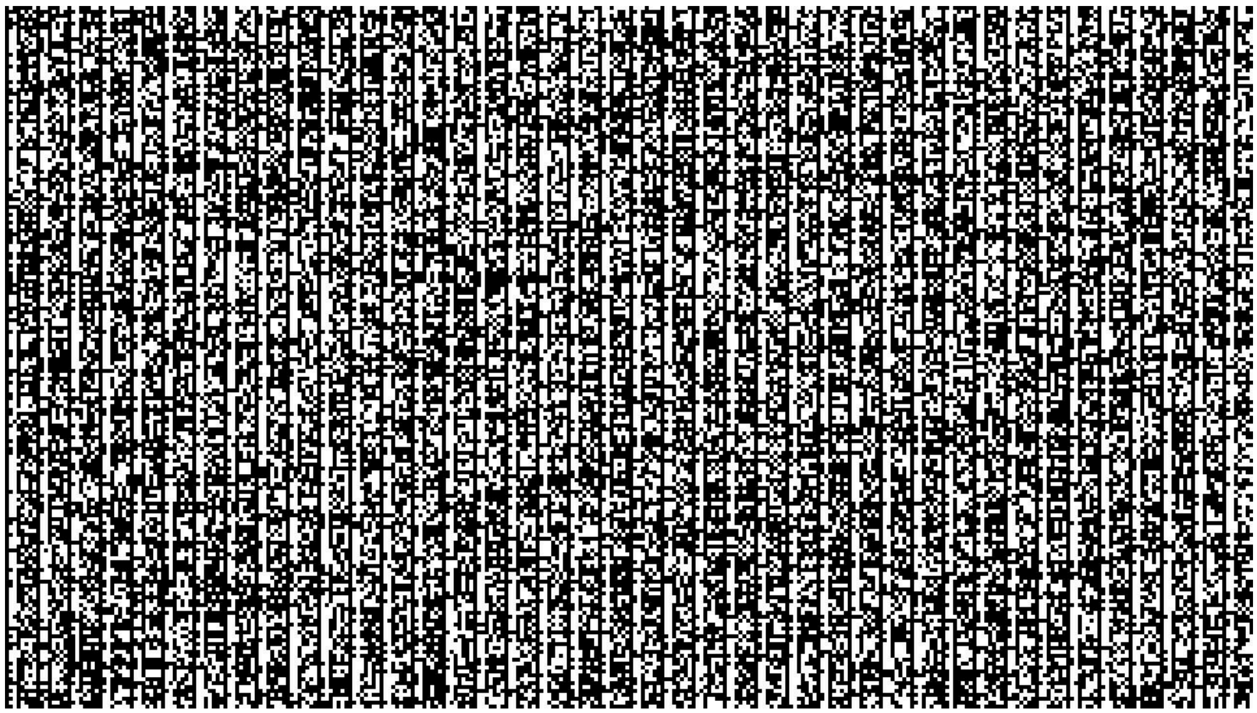


Fig. 1: Example of a single frame from data encoded as video with block sizes of 3 pixels.

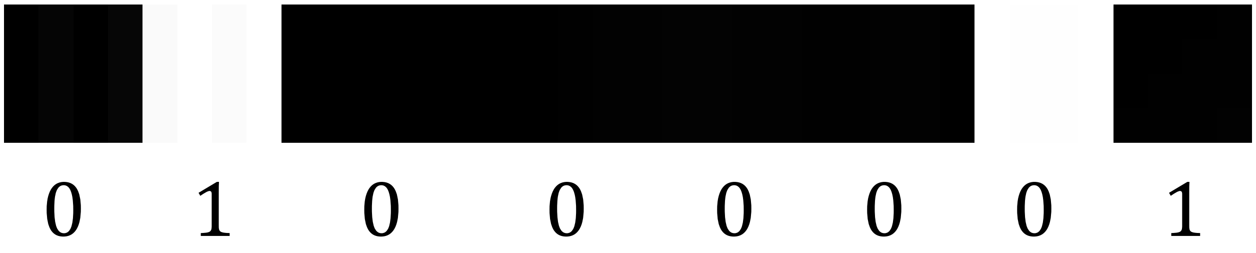
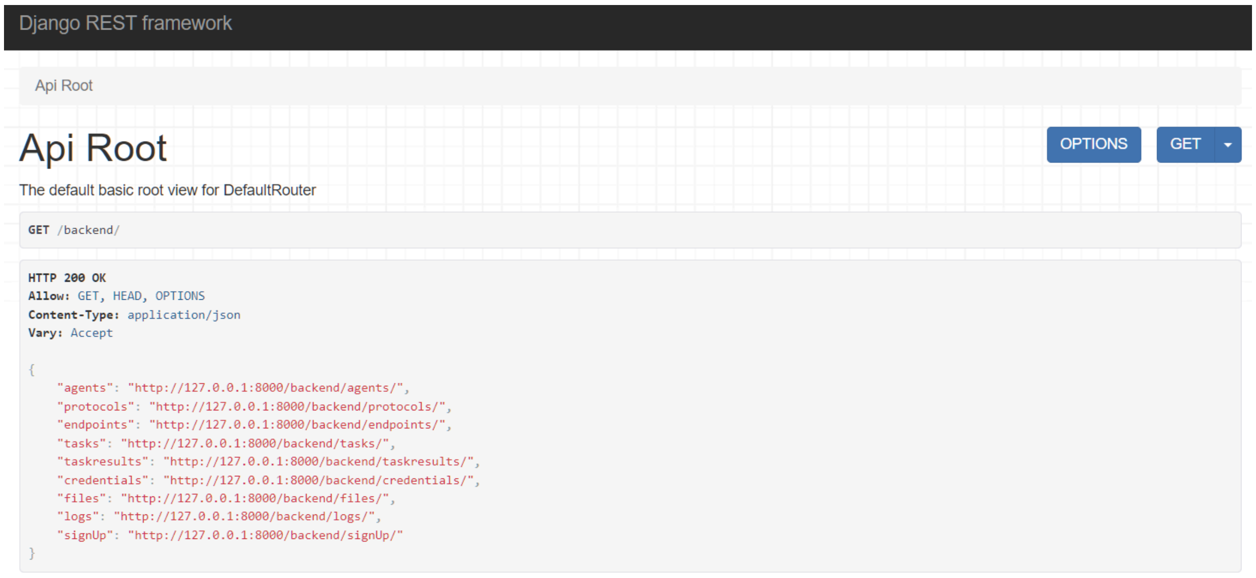


Fig. 2: The first byte from Fig. 1. This is represented as 0x41 in hexadecimal, equivalent to an ASCII "A" character.



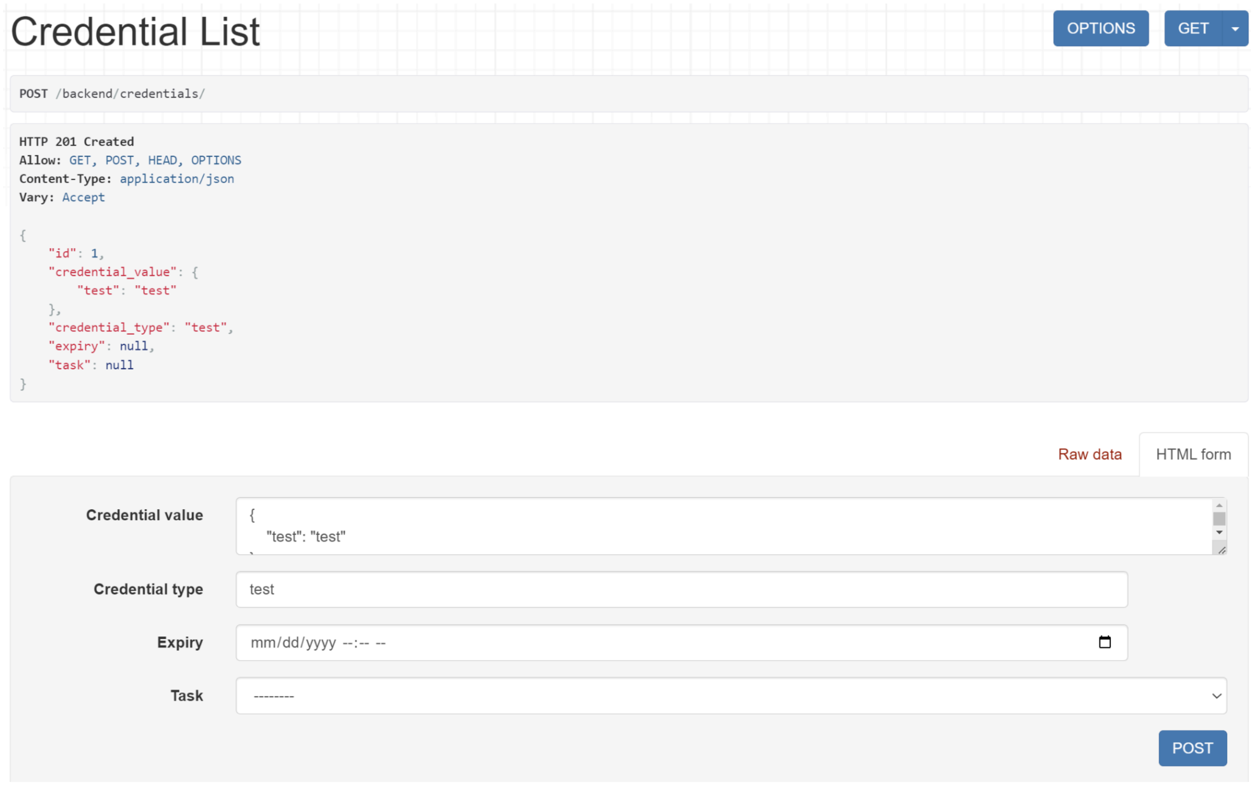
Api Root

The default basic root view for DefaultRouter

```
GET /backend/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "agents": "http://127.0.0.1:8000/backend/agents/",
    "protocols": "http://127.0.0.1:8000/backend/protocols/",
    "endpoints": "http://127.0.0.1:8000/backend/endpoints/",
    "tasks": "http://127.0.0.1:8000/backend/tasks/",
    "taskresults": "http://127.0.0.1:8000/backend/taskresults/",
    "credentials": "http://127.0.0.1:8000/backend/credentials/",
    "files": "http://127.0.0.1:8000/backend/files/",
    "logs": "http://127.0.0.1:8000/backend/logs/",
    "signUp": "http://127.0.0.1:8000/backend/signUp/"
}
```

Fig. 3: Example of the API root view of the Django Rest Framework API router. This provides users with a direct way to issue RESTful API requests to the Django backend, as well as a convenient way of documenting available endpoints.


Credential List

OPTIONS GET

```
POST /backend/credentials/
```

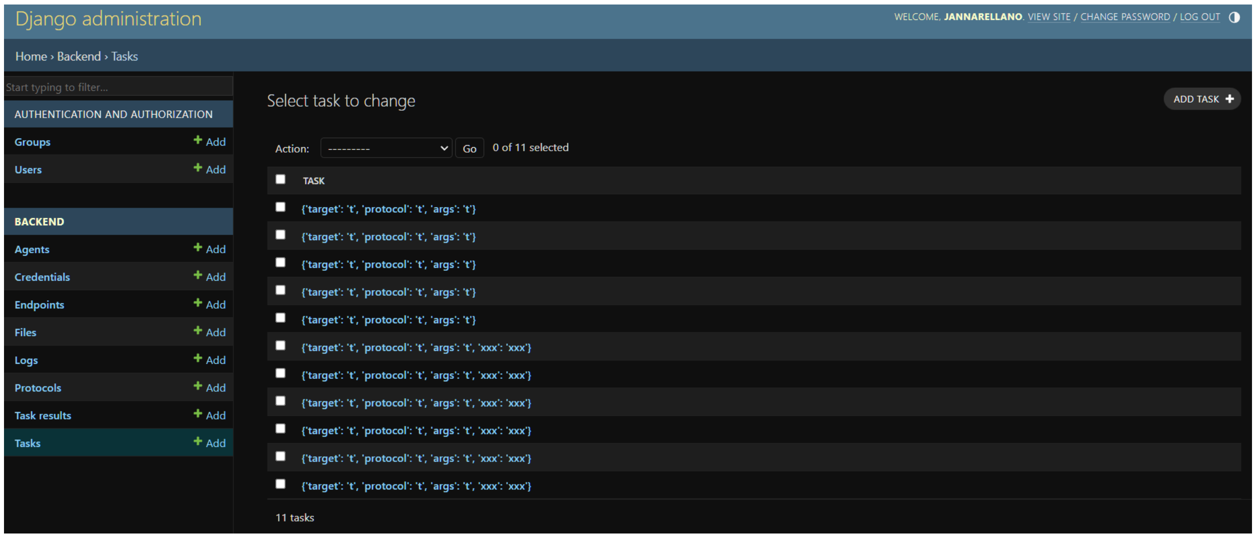
```
HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "credential_value": {
        "test": "test"
    },
    "credential_type": "test",
    "expiry": null,
    "task": null
}
```

Raw data | HTML form

Credential value	{ "test": "test"
Credential type	test
Expiry	mm/dd/yyyy --:-- --
Task	--------

POST

Fig. 4: Example of a GET and POST request initiated with Django Rest Framework to the Django backend. Currently, this directly results in the creation of a new model instance (i.e. database entry) without validation or asynchronous tasking; this is planned for the winter/spring.


Django administration

WELCOME, **JANNARELLANO**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Backend › Tasks

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION
- Groups + Add
- Users + Add

BACKEND
- Agents + Add
- Credentials + Add
- Endpoints + Add
- Files + Add
- Logs + Add
- Protocols + Add
- Task results + Add
- Tasks + Add

Select task to change

ADD TASK +

Action: --------- Go 0 of 11 selected

```
TASK
{'target': 't', 'protocol': 't', 'args': 't'}
{'target': 't', 'protocol': 't', 'args': 't'}
{'target': 't', 'protocol': 't', 'args': 't'}
{'target': 't', 'protocol': 't', 'args': 't'}
{'target': 't', 'protocol': 't', 'args': 't'}
{'target': 't', 'protocol': 't', 'args': 't', 'xxx': 'xxx'}
{'target': 't', 'protocol': 't', 'args': 't', 'xxx': 'xxx'}
{'target': 't', 'protocol': 't', 'args': 't', 'xxx': 'xxx'}
{'target': 't', 'protocol': 't', 'args': 't', 'xxx': 'xxx'}
{'target': 't', 'protocol': 't', 'args': 't', 'xxx': 'xxx'}
{'target': 't', 'protocol': 't', 'args': 't', 'xxx': 'xxx'}
```

11 tasks

Fig. 5: Example of the Django administration dashboard, which provides an easy way for experienced operators to edit database values. This is equivalent to “superuser” status, as it allows users to directly modify sensitive values.

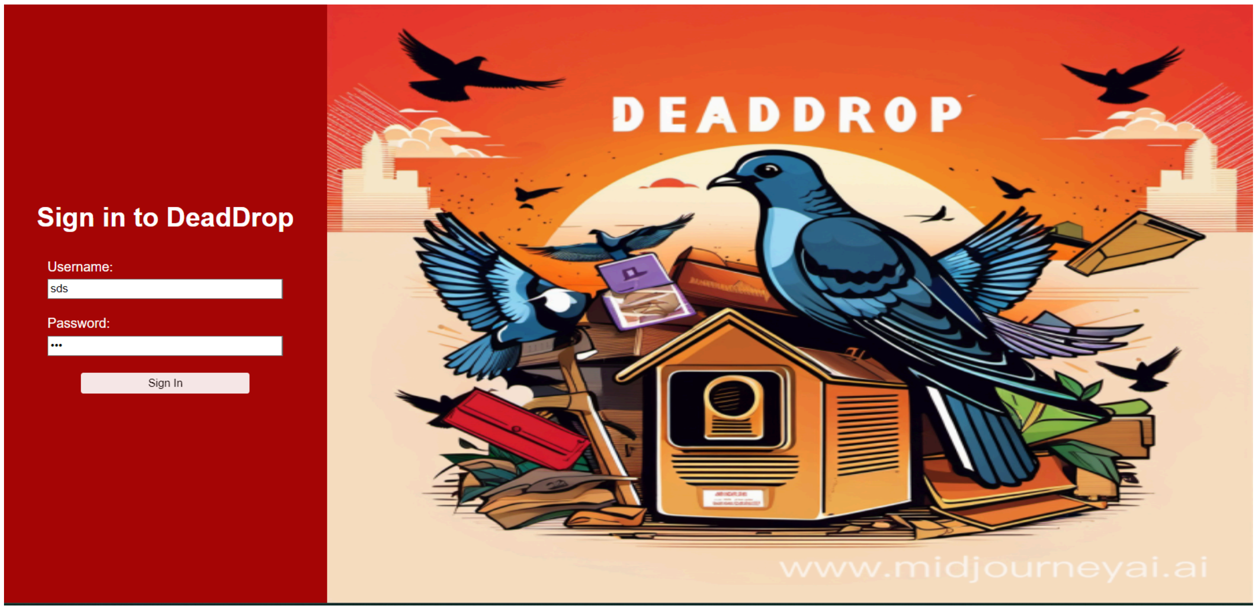
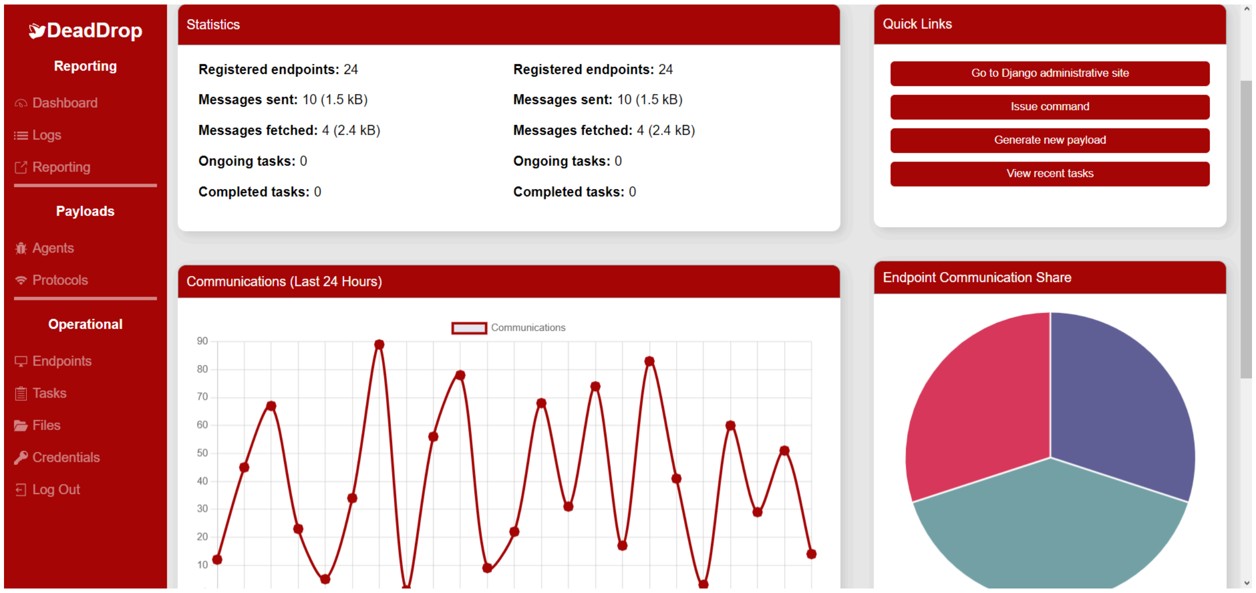


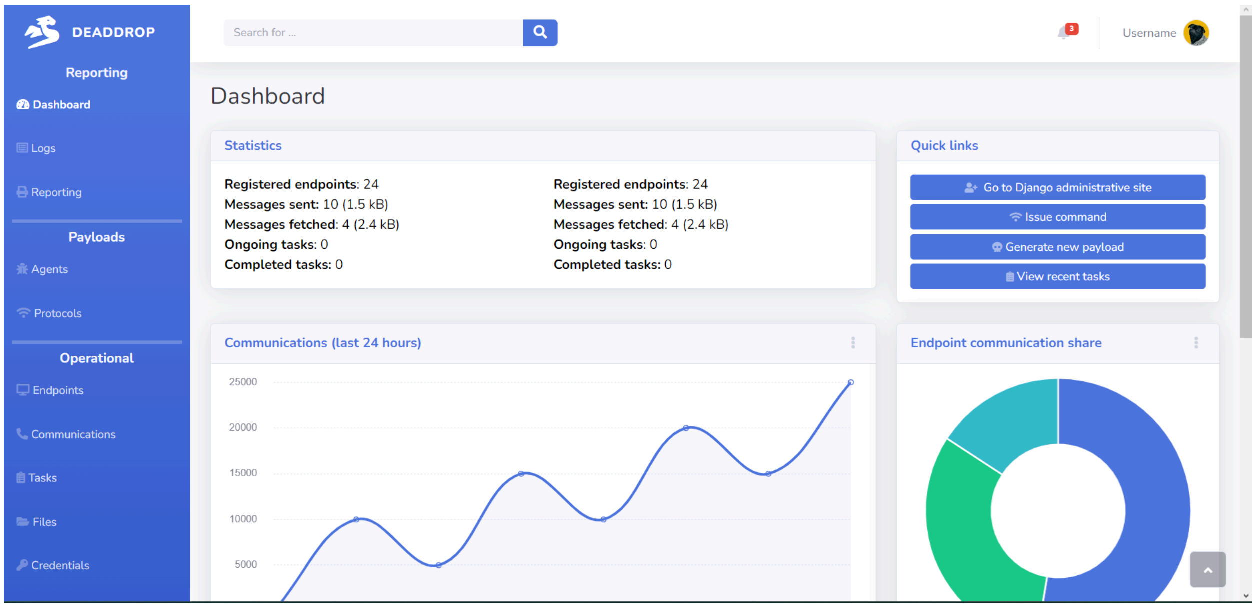
Fig. 6: Login page for the frontend, as implemented with Svelte. This directly interfaces with Django, which handles sessions and user management.



Registered endpoints: 24
Messages sent: 10 (1.5 kB)
Messages fetched: 4 (2.4 kB)
Ongoing tasks: 0
Completed tasks: 0

Registered endpoints: 24
Messages sent: 10 (1.5 kB)
Messages fetched: 4 (2.4 kB)
Ongoing tasks: 0
Completed tasks: 0

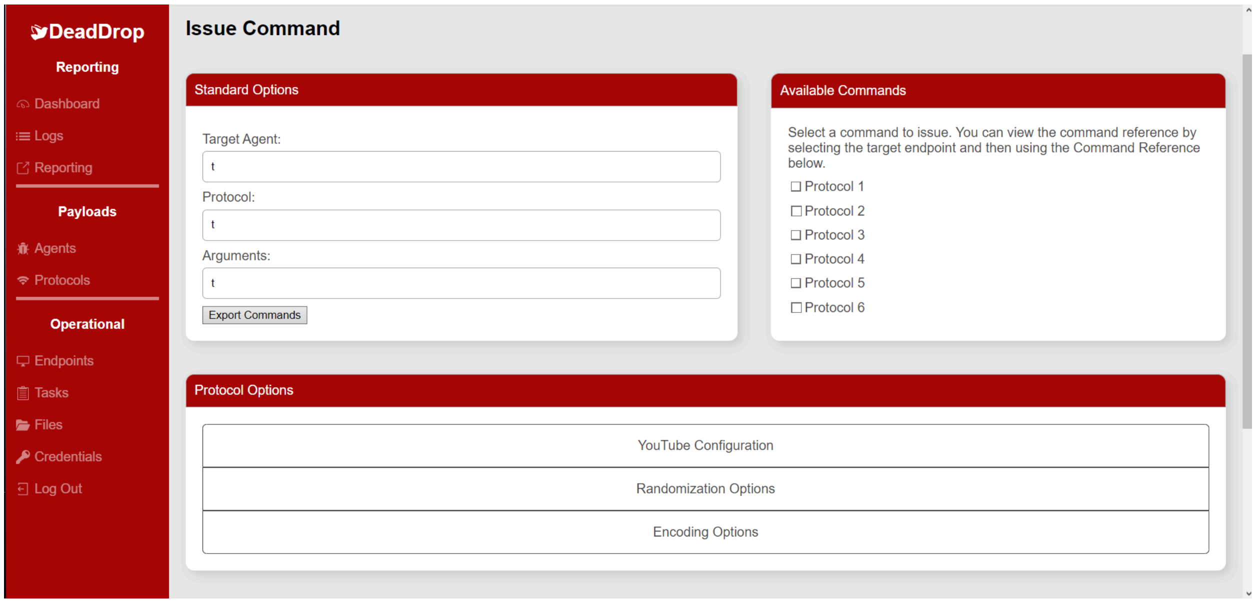
Fig. 7: Implemented dashboard prototype in Svelte, providing users with a high-level overview of recent framework activity. Charts generated with Chart.js.



Registered endpoints: 24
Messages sent: 10 (1.5 kB)
Messages fetched: 4 (2.4 kB)
Ongoing tasks: 0
Completed tasks: 0

Registered endpoints: 24
Messages sent: 10 (1.5 kB)
Messages fetched: 4 (2.4 kB)
Ongoing tasks: 0
Completed tasks: 0

Fig. 8: Mockup dashboard created with Bootstrap Studio.



Issue Command

Standard Options

Target Agent: t

Protocol: t

Arguments: t

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

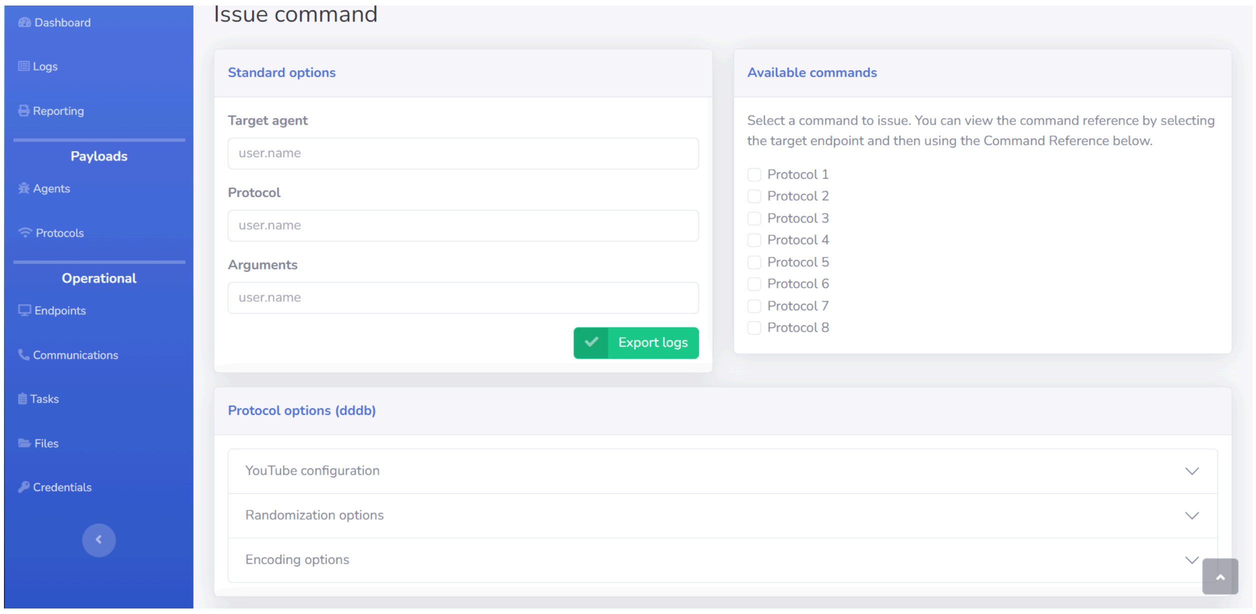
Protocol Options

YouTube Configuration

Randomization Options

Encoding Options

Fig. 9: Command page implemented with Svelte, which allows operators to specific commands to a target agent/endpoint over a specified protocol.



Issue command

Standard options

Target agent: user.name

Protocol: user.name

Arguments: user.name

Export logs

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 7
- ☐ Protocol 8

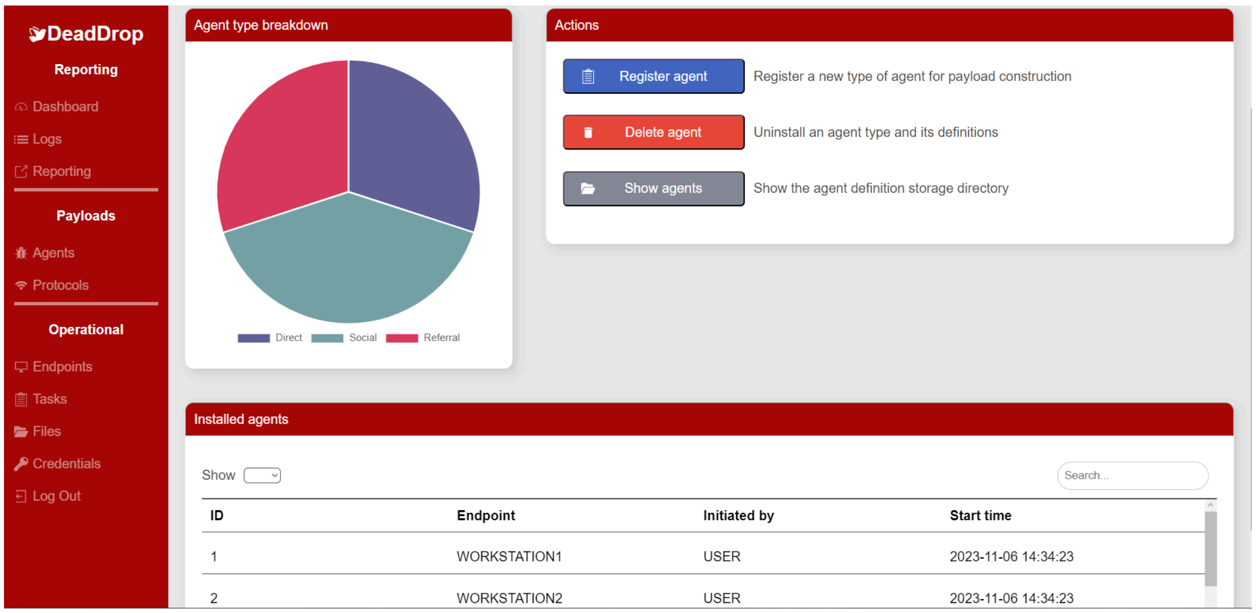
Protocol options (dddb)

YouTube configuration

Randomization options

Encoding options

Fig. 10: The same command page as originally designed in the mockup.



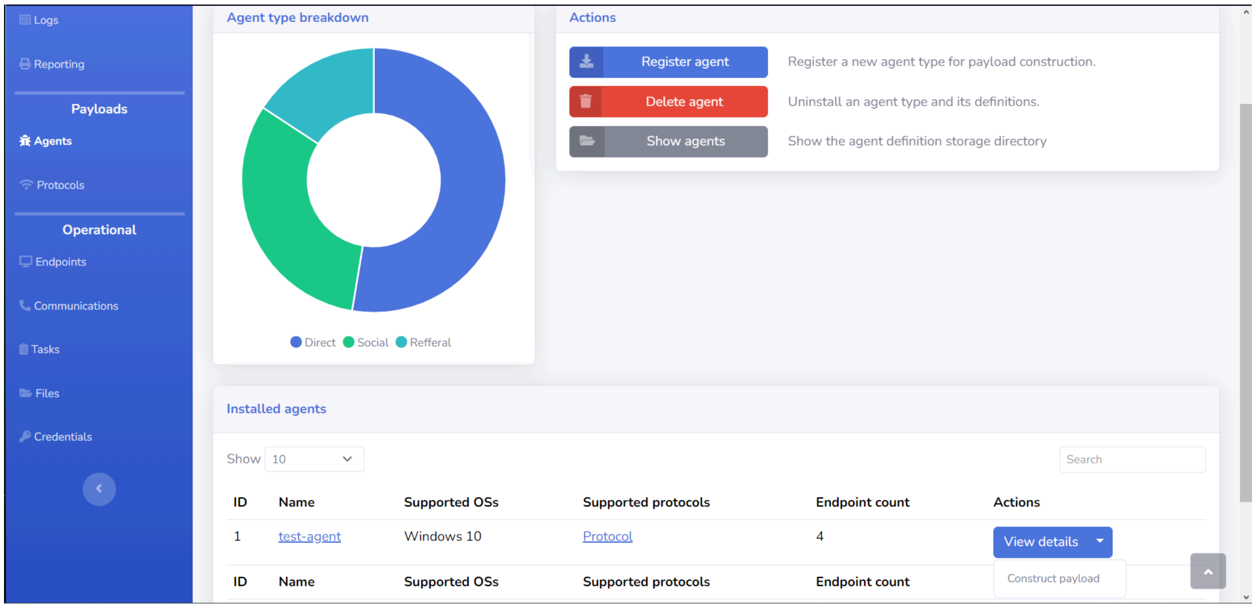
Register a new type of agent for payload construction

Uninstall an agent type and its definitions

Show the agent definition storage directory

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

Fig. 11: Agent listing page implemented with Svelte, which allows users to manage agents and create new endpoints.



Register a new agent type for payload construction.

Uninstall an agent type and its definitions.

Show the agent definition storage directory

ID	Name	Supported OSs	Supported protocols	Endpoint count	Actions
1	test-agent	Windows 10	Protocol	4	View details
ID	Name	Supported OSs	Supported protocols	Endpoint count	

Fig. 12: The same agent listing page as originally designed in the mockup.

IV. Prototype Demo

Team 08 presented the prototype of DeadDrop to the CS 425 teaching team on Monday, December 11, 2023, at 11:30 a.m. We received positive feedback regarding our current progress, and intend to continue with our current pace and project management approach. Additionally, the teaching team provided two general recommendations that we intend to act on:

- The teaching team provided a few interesting ideas for communication protocols, including the use of Slack as a "dead drop" platform and increased data density through sound and subtitle encoding. YouTube's limitations and compression algorithm have largely hampered efforts in this respect, but we plan on exploring these on different platforms (such as Zoom) that may not use aggressive compression.
- It was suggested that we reach out to our advisor, Shamik Sengupta, regarding our current progress with this project, as well as future plans for the project from a research perspective.

Finally, the following paraphrased comment from the session:

> *"Get in trouble… It would be cool if your group got the whole school banned from YouTube."*

has been duly noted.

V. Time Worked

This time is inclusive of time spent implementing the prototype, preparing for the demo, and working on this document. All team members spent approximately 20 hours together for work associated with the prototype, which is reflected below.

Hyperlinks go to the associated repositories for each code section contributed.

Team Member	Hours	Sections Contributed
Jann Arellano	60	- Code: - Frontend (Svelte backend) - Backend (Django) - Document: - Prototype Development
Brian Buslon	50	- Code: - Frontend (Svelte frontend) - Document: - Prototype Objectives and Functionality
Keaton Clark	55	- Code: - Protocol handler (dddb) - Document: - Prototype Development - Prototype Objectives and Functionality - Prototype Demo
Lloyd Gonzales	55	- Code: - Frontend (Bootstrap Studio mockup) - Backend (Django) - Prototype agent - Document: - Abstract - Introduction - Prototype Objectives and Functionality - Prototype Demo