

*Team SDMAY19-19: Offensive Security Orchestration*

# Final Project Report

Client: [Redacted by request of client]

Faculty Advisor: Doug Jacobson

Daniel Limanowski - Frontend Lead

Vijay Uniyal - Frontend Developer

Justin Roepsch - Frontend Developer

Paul Chihak - Implant & EDR Testing Lead

Adam Crosser - Implant & EDR Testing Developer

Logan Kinneer - Implant & EDR Testing Developer

Team Email: [sdmay19-19@iastate.edu](mailto:sdmay19-19@iastate.edu)

Team Website: <https://sdmay19-19.sd.ece.iastate.edu>

Revised: April 20th 2019 / Version 1.0

# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Execute Summary</b>	<b>4</b>
<b>Requirements Specification</b>	<b>5</b>
Use-Cases	5
Functional Requirements	5
Bot requirements	5
Command and Control requirements	6
Non-Functional Requirements	6
<b>System Design &amp; Development</b>	<b>6</b>
Design Plan	6
Design Objectives, System Constraints, Design Trade-offs	7
Command-And-Control Service	7
Implant Requirements	8
Cuckoo Sandbox Service	8
Builder Microservice	8
Design Analysis	8
Architectural Diagram, Design Block Diagram -- Modules, Interfaces	9
Description of Modules, Constraints, and Interfaces	10
Command-And-Control Server Interfaces	10
Cuckoo Sandbox interface	11
<b>Implementation</b>	<b>12</b>
Implementation Diagram, Technologies, Software Used	12
Rationale for Technology/Software Choices	13
Applicable Standards and Best Practices	13
<b>Testing, Validation, and Evaluation</b>	<b>13</b>
Test Plan	13
Unit Testing the C2	14
Manually Testing the Bot	14
Integration Testing	15
User-level Testing	16
<b>Project and Risk Management</b>	<b>16</b>
Task Decomposition & Roles and Responsibilities	16
Project Schedule - Gantt Chart (Proposed vs. Actual)	16

Risks and Mitigation	19
Lessons Learned	20
<b>Conclusions</b>	<b>20</b>
<b>Future Work</b>	<b>21</b>
Implement Google BeyondCorp (Zero Trust) Security Model	21
Implement Malleable C2 Framework	22
<b>References</b>	<b>22</b>

# Execute Summary

[REDACTED] engaged SDMay19-19 to develop an Offensive Security Orchestration platform to improve the stealth, effectiveness, and efficiency of red team engagements conducted by [REDACTED].

Red team operations are simulated adversary simulation exercises whereby the third party [REDACTED] simulates the actions performed by a sophisticated adversary targeting the target organization who engaged the services of [REDACTED]. The purpose of the adversary simulation is to simulate an external actor, such as a foreign intelligence agency, competitor, or criminal organization who is attempting to compromise the target organization through any means necessary.

During the assessment [REDACTED] is given a target objective to accomplish which could include things such as theft of intellectual property, credit card numbers, or personal information. While compromising the target environment and moving laterally [REDACTED] will attempt to accomplish the provided objective.

These assessments are completely black box meaning that [REDACTED] has no information on the configuration or overall security posture of the target organization. Furthermore, during a red team operation only a few key stakeholders at the target organization are aware that [REDACTED] has been engaged perform this assessment.

Because of this if [REDACTED] is detected operating within the target environment the network security team at the target organization will think and respond to the attack like it was a real incident. The purpose of this assessment is to determine the readiness of the target to defend against a sophisticated adversary targeting their environment. The red team approach provides a holistic review of the information security capabilities of the customer and allows for identification of areas where processes, technical controls, and detection mechanisms can be improved.

When operating in high security environments it is critical that [REDACTED] is able to gain initial access, move laterally, and operate without being detected. Furthermore, [REDACTED] being a consulting-based services organization wants to improve the efficiency of their processes to reduce costs and improve product delivery and consistency to their customers.

One core problem is that [REDACTED] lacks a custom malware implant that can be used to operate in high security environments. SDMay19-19 has provided a solution to this problem by

developing an minimum viable product (MVP) that can be used to operate in these types of environments.

The initial implementation includes a web-based command and control interface which can be used to control compromised assets within the target organization. A custom implant has been developed that communicates with the command and control server and is used to perform actions on a compromised host. Additionally, the implant supports sophisticated command and control mechanisms such as domain fronting.

SDMay19-19 has also provided an analysis environment which can be used to perform dynamic analysis against the custom implant in order to identify potential IOCs (Indicators of Compromise) which could be used to compromise that could be used to detect the implant running within a target environment.

## Requirements Specification

This section details the function and nonfunctional requirements of the project as well as the expected use-cases based off of discussions with [REDACTED].

### Use-Cases

This project has a use case of our client, [REDACTED], conducting an assessment of a target organizations security. This assessment will include multiple simultaneous users of the software.

### Functional Requirements

The functional requirements of our system are composed of two primary components that work both independently and together but supply their own unique functionality. Those components are the malware (“Bot”) and the command and control frontend (“C2”).

#### Bot requirements

The malware has the following functional requirements:

- Communicates with C2 via a secure, encrypted RESTful API
- Is tested and executable on recent versions of Microsoft Windows Operating System
- Is able to disconnect from a C2 and destroy itself
- Supports domain fronting with Amazon Cloudfront and frontable domains
- Demonstrates scheduled task persistence while remaining stealthy
- Endpoint detection and response solution bypass capabilities

## Command and Control requirements

The command-and-control frontend interface has the following functional requirements:

- Communicates over encrypted channel with multiple bots
- Uses Django Python Framework to manage APIs and database queries
- Provides a user interface for sending commands to different bots
- Provides documentation and help to the user
- Allows user creation, deletion, and authentication
- Logs all activity by users
- Has realtime websockets for receiving data from the bots
- Allows building and downloading of malware in-app
- App managed as containers via docker-compose

## Non-Functional Requirements

- Non-regular intervals for heartbeats
- Bot self-destruction upon demand or if it cannot find the C2 server
- Stealthy bots - no virus warnings
- Ease of navigation on C2 server
- Multi-user simultaneous access

# System Design & Development

## Design Plan

There are two primary components of the project, the implant and the web application which is used to control the implant.

There is a one-to-many relationship between the control server and the implants. The control server can control multiple implants, but an implant can only be associated with a single control server.

An “operator” or “attacker” can control an implant using the web application. The operator is able to perform tasks such as download/update files to the system, run commands, move laterally to other systems on the network, and establish persistence.

It is also possible for a “redirector” to be deployed to act as a proxy between the implant and the control server. The purpose of this is to allow for the attacker to hide their control server behind more trusted infrastructure.

The benefits provided by this architecture are that it is more difficult for network security teams to identify the backend infrastructure used by the attacker. Additionally, in instances where “domain

fronting” is utilized an attacker can hide their command and control traffic behind a legitimate and trusted domain.

The design also includes an instance of “Cuckoo Sandbox”. This will be integrated with the control application and will allow operators to analyze various aspects of the implant and automatically extract information on various technical and forensic indicators that are left on a system when the implant is executed in a target environment. This is useful for two reasons:

1. It allows an operator to identify what forensic indicators are left on a system when the implant is executed
2. It is capable of generating an automated report and automatically extracting these indicators which saves a great deal of time compared to manual analysis
3. It allows for semi-automated testing of the implant across various types of platforms and environments.

Outside of this is a builder service which integrates with the command and control server. When the command and control server needs to create a new version of the implant it will make an API call to the builder service and the builder service will return a new version of the implant that is configured to front through a specific domain via domain fronting and route requests to a specific redirector.

## Design Objectives, System Constraints, Design Trade-offs

Before beginning the design process the client laid out all the necessary and desired requirements. These objectives and constraints are as follows.

### Command-And-Control Service

The Command-And-Control system, or C2 for short, will be a single-page web application with a dedicated frontend and backend. The C2 will use the following technologies and frameworks:

#### Frontend

- ReactJS UI framework
- Semantic UI CSS/JS framework

#### Backend

- Django Python web framework
- Django REST Framework

## Implant Requirements

The implant is implemented in C# targeting version 4 of the .NET Framework. The C# programming language was chosen since it allows for more rapid development compared to unmanaged languages, such as C or C++. While Amazon CloudFront was chosen for domain fronting due to the large number of available domains hosted there which could be used for domain fronting.

Additionally, version 4 of the .NET Framework is bundled with most versions of the windows operating system that are currently being used in modern enterprise environments. Outside of this the .NET Framework bundles a large number of useful libraries and functionality which can be used out of the box.

## Cuckoo Sandbox Service

The Cuckoo sandbox service will be used to automatically extract what forensic indicators are created by the implant when it is run on a host. This will be used by operators to identify the various aspects of the implant that an endpoint detection and response solution could flag on/alert on.

## Builder Microservice

The builder allows for customized versions of the implant to be created without recompiling the implant binary. It functions by embedding an XML file into the resources section of the implant. The XML file specifies the domain what should be fronted to and the hostname of the attacker controlled CloudFront redirector.

It was decided that the builder would function as a separate microservice running on Windows Server 2016 Nano Server. This design decision was made because modifying the resources section of a Windows Portable Executable (PE) file requires utilization of proprietary libraries provided by Microsoft which are only available on Microsoft Windows. Therefore it will be used by having the command and control web application make API calls to the microservice whenever a user requests to build a new implant.

One of the advantages of this design is that the attacker can create multiple redirectors and generate multiple implants that do domain fronting through different domains, but they all connect to the same backend control server. However there is the downside of it increasing complexity of the project due to needing to run it as its own separate microservice which will make it harder for the client to give the project to someone else to maintain.

## Design Analysis

Every choice made for the project is backed by research and previous project experience from each of the team members. However, some compromises had to be made to ensure a completed project that satisfies all of the client requirements.



For this reason the project will only support application deployment via microservice containers with Docker. The reasoning behind using Docker is that the client will be using the application in short, engagement-based bursts. Since Red Team Engagements and Penetration Tests only last up to months at a time, and can complete in as short as a few days, the tradeoff to only support Docker instances of the C2 application makes more sense than forcing the client to have a longstanding server just to host it.

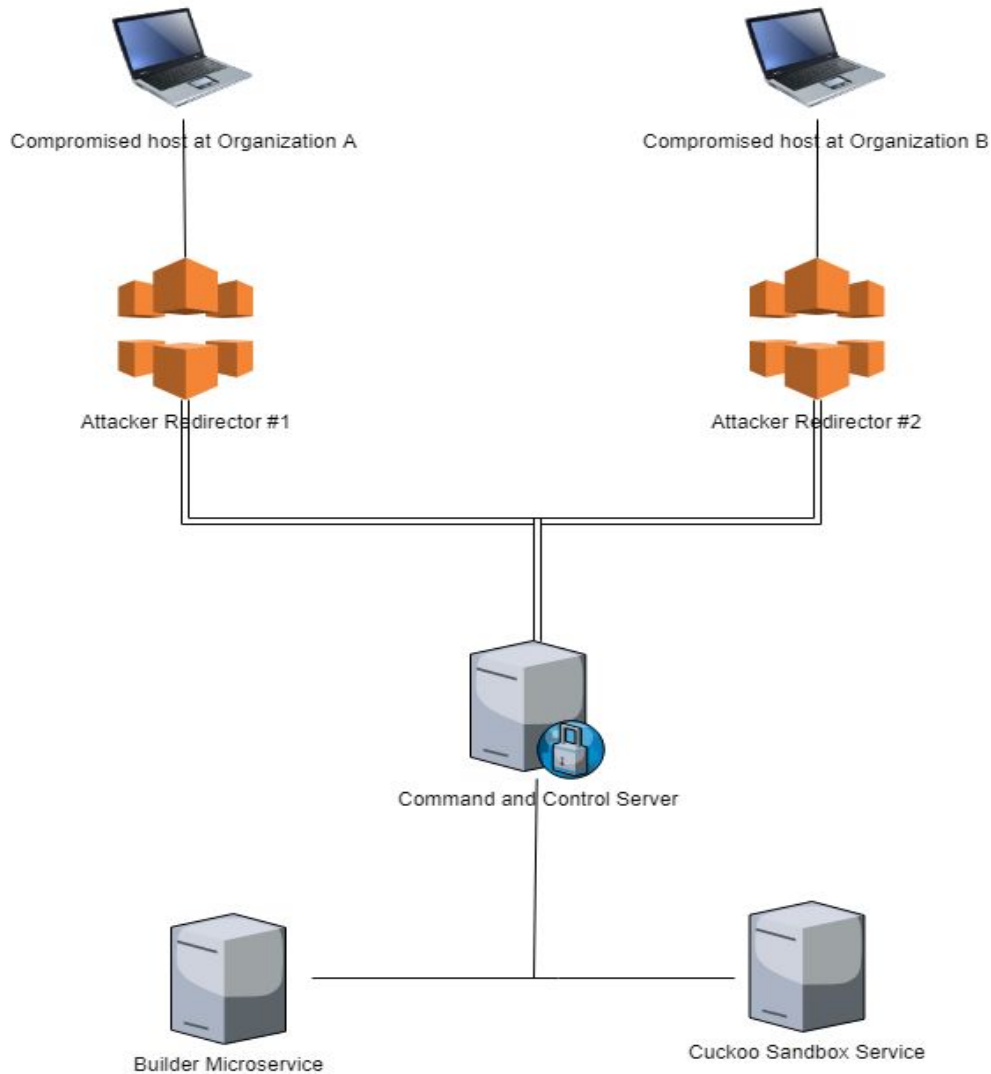
In the rare event that the client wanted a more permanent installation of the application, they will have to migrate the project from Docker containers to individual servers for each of our microservices.

Due purely to time constraints and to reduce complexity of the project, we have made the design decision to only support one type of malware payload at this time, and that payload being the provided C# bot that the client initially has developed. This reduces the overall flexibility of the project, in that only one piece of malware can be deployed and monitored, rather than using a plugin suite of malware, but due to the vast requirements in other areas of this project, we had to compromise and restrict the bots that we will support. While this is less flexible, it will allow for more stability with the existing malware as we only have to test against a single, albeit configurable, flavor of malware.

Another compromise that had to be made was that originally the project was going to give a simple yes or no answer as to whether or not a payload would be detected by a specific endpoint detection and response solution. However this would have required the client to buy multiple expensive EDR solutions and also would have required extensive reverse engineering in order to retrieve the necessary data. Therefore due to both time, and cost constraints the project shifted instead to using Cuckoo Sandbox which can extract forensic artifacts so that even if the client will not necessarily know which EDR solution will detect the payload they will at least know what evidence the payload is leaving behind.

## Architectural Diagram, Design Block Diagram -- Modules, Interfaces

Below is a high-level architecture diagrams which shows how each of the components of the system interact. The builder service allows for multiple implants to be configured to do domain fronting through multiple redirectors fronting through different domains. The command and control server application calls two different backend services. One of them is the builder microservice and the other is the Cuckoo Sandbox Service.



## Description of Modules, Constraints, and Interfaces

### Command-And-Control Server Interfaces

The frontend and backend will communicate almost exclusively via HTTP/S REST APIs, provided by the Django REST Framework (DRF), a wholly supported, financed, and dedicated framework to providing stable RESTful Application Programmer Interfaces (APIs). The image below details some of the APIs we have already developed and tested with regards to the malware we were provided.

- **bots/{key}**
  - GET: returns expanded details from a single bot, identified by bot.key
- **heartbeat/**
  - POST: given a key, updates when the bot last checked in
- **init/**
  - POST: creates a new bot, bot calls this to register with CNC
- **botupload/**
  - POST: allows a bot to upload a file to CNC
- **message/**
  - POST: allows bot to return output to the CNC
- **command/{key}**
  - GET: returns raw-text response of the current command for the bot

*Figure 3: Bot APIs for the C2*

The backend database will store existing bots and their corresponding files and data. The database will also store logs from the users' actions and user authentication details. Because this application is going to be used in the short-term (one week to a few months), a single-file SQLite database may reduce complexity and make it easier to backup or extract the data for the client.

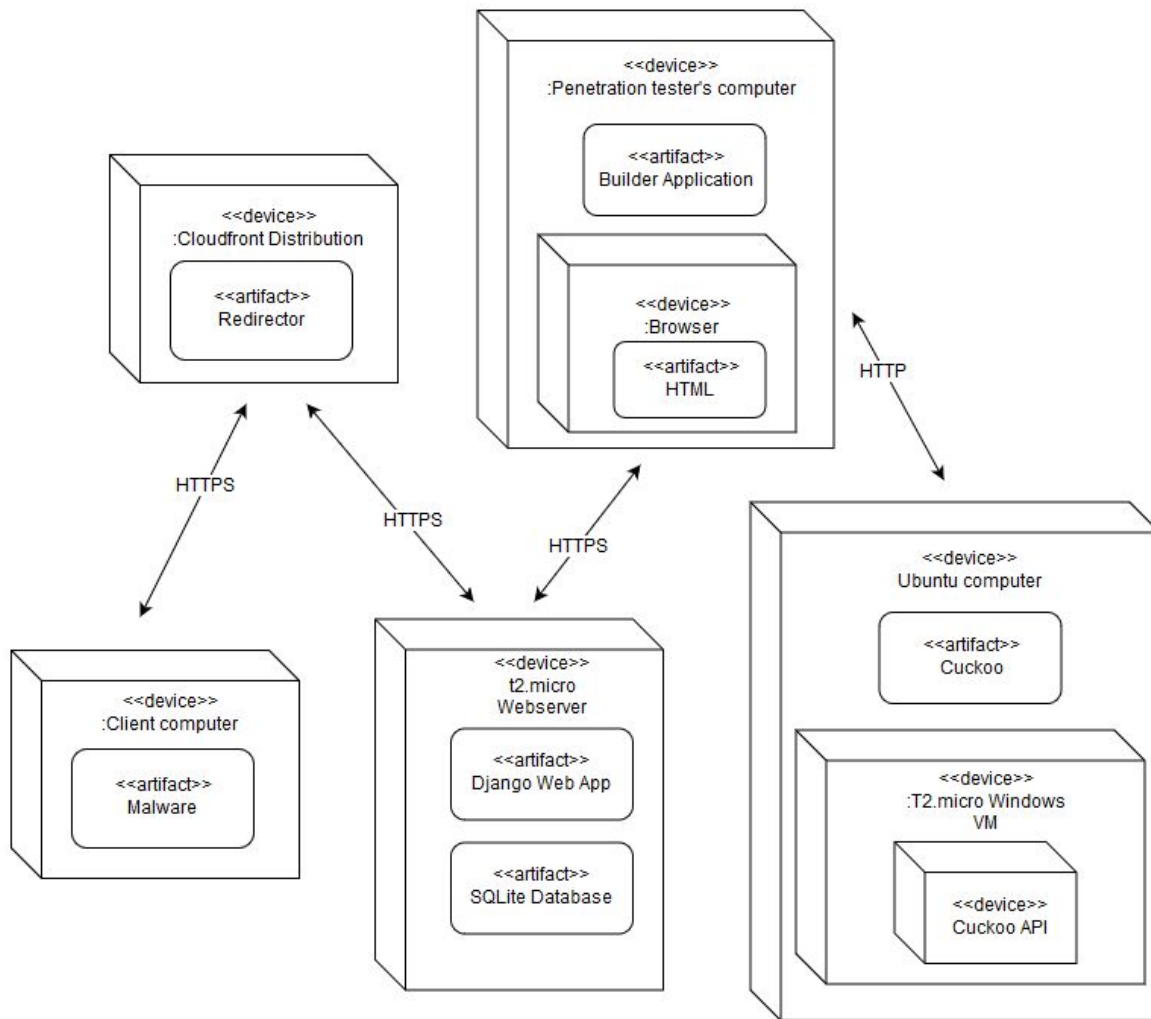
Furthermore, to reduce complexity for the client, the project will utilize Docker for easy production-ready deployment of this application in terms of microservices. This will allow the client to spin up an instance of the C2 anywhere (a laptop, a dedicated server, in the cloud, etc.), and without any initial configuration or tedious setup steps needed.

## Cuckoo Sandbox interface

The Cuckoo Sandbox instance is hosted on a Linux 16.04 server with Virtualbox installed. Furthermore Cuckoo requires a Windows Virtual Machine to be created and snapshotted so that the Cuckoo agent distribute and execute the payload before reverting the Virtual Machine back to its original state. In order to access the Cuckoo Sandbox instance there is simply a button on the web application that will redirect over to it and then the client is able to test any payloads they want.

# Implementation

## Implementation Diagram, Technologies, Software Used



The malware component of the project is capable of being deployed to a client's computer using a phishing email with a malicious attachment. From the the malware establishes communication with the command and control server and allows the penetration testers to issue commands to the malware. The malware was written in C# such that the DotnetToJScript tool could be used to convert the code in a manner that it could be embedded in Microsoft Office files. This was desired by the client for use in engagements. Penetration testers can optionally send the malware they have compiled using the malware builder to a machine with our Cuckoo setup on it to test if the malware works as intended.

The command and control server is written in using django. This server waits for the malware to connect to it then provides a user interface for the penetration testers to interact with the malware. It provides

secure communication between itself, the malware, and the penetration testers. Actions performed by the penetration testers are logged for reporting and accountability.

## Rationale for Technology/Software Choices

Our rationale for using the .net framework to write the malware is that the malware can then be compiled to a Jscript which can be embedded in a word document allowing for easy distribution of the malware. We chose to use Cuckoo to test our malware because it can be run locally guaranteeing that the malware does not get signitured by antivirus software. Additionally Cuckoo can be further tweaked by the client as they invent new tests they would like to perform on the runtime qualities of the malware.

Initially when planning this project we had planned to use ReactJS on the front end of the command and control server, however we as we began to work through the project we realized that React JS was not necessary for the project, created additional work when modifying the command and control server, and posed a risk to the maintainability of the project over time. For these reasons our client agreed that it would be best if the React was removed from the project, and a simpler UI was put in place.

## Applicable Standards and Best Practices

The project implements a number of standards which are critical to keep the confidential data our project deals with confidential, as well as to make the project more useful to penetration testers. Some of these standards include the TLS standard defined in RFC 5246 which keeps data private while in transit between our command and control server and the other components it connects to over the internet. One of the ways our project is made better is through the implementation of domain fronting which is accomplished by sending unusual but perfectly valid HTTP requests. These requests conform to the HTTP standards laid out in the RFC's.

## Testing, Validation, and Evaluation

### Test Plan

Our project is unique in that it requires testing of two independent components, the Command and Control (C2) server and the configurable bot that runs on Microsoft Windows endpoints. We opted for a combination of automated and manual testing to ensure our system as a whole operates with robustness and exhibits fault tolerance. Due to the sensitive and private nature of the operation of our project in Red Team environments, we must ensure the C2 and bots are free of bugs that may cause them to crash or fail to communicate between each other.

## Unit Testing the C2

Our C2 application (frontend and backend) uses the Django web framework, written in Python. The Django framework allows programmers to leverage the built-in Python Unit Testing library, which our team took full advantage of.

Each model and view was written with a “test-driven” development cycle, meaning we initially wrote our unit tests to define how we wanted the application to work, and crafted Django code that would pass said tests. Test-driven development was more of a time investment upfront, but it saved us time throughout the evolution of our project by allowing us to immediately identify bugs as we made changes to the codebase.

The screenshot below shows unit tests being run with a Python script. We simply need to run `python manage.py test` to kick off all of our unit tests for the entire C2. Any failures are immediately brought to attention and can be located quickly within the project. The unit tests ensure that all database models and transactions operate as expected. Additionally, the unit tests test that all views have proper permissions, locations, and use the proper templates for rendering.

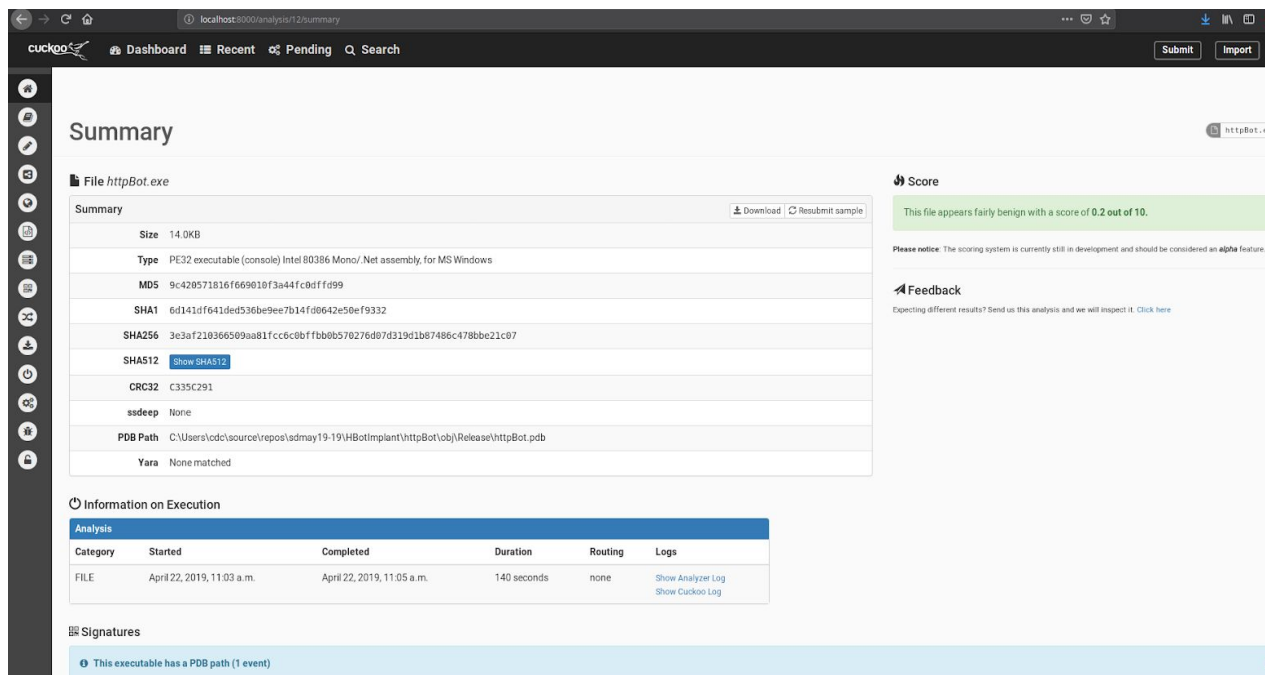
```
(venv) dlimanow@gazelle:~/seniordesign/sdmay19-19/c2$ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced)
.....
-----
Ran 7 tests in 7.134s
OK
Destroying test database for alias 'default'...
```

*Figure: Running all (grouped) unit tests. If any failed, we would see a detailed error message and exact test that failed.*

## Manually Testing the Bot

Because the Bot can run on multiple configurations of Microsoft Windows, which may have different antivirus and endpoint detection/response software installed, it was very difficult to automate testing for the bot, forcing us to perform manual testing.

In order to ease the testing, the team made a decision to integrate the Cuckoo Sandbox into the project. Cuckoo Sandbox is an open-source project that allows for automated malware analysis. By deploying and integrating Cuckoo into the C2 server, a Red Team operator can configure malware with the malware builder and then immediately submit the malicious binary to Cuckoo for detection analysis. Cuckoo tells us which indicators of compromise therein exist (and therefore the malware's probability of detection). This allows for faster manual testing of changes to the bot which also provides valuable intelligence to the Client (outside of providing value to the programmer(s)).



*Figure: Cuckoo showing analysis results for our custom malware implant binary. Note that Cuckoo thinks the binary is benign, this is exactly what we want it to report as EDR solutions will likely think this as well and not report it.*

## Integration Testing

The C2 and Bot must be able to communicate well with each other for the system to function as anticipated. We performed integration testing between these two components via virtual machines running various version of Windows 7 and Windows 10. The C2 server was placed on a private network with Windows workstations and malware was executed on the workstations with varying configurations. Each command supported by the bot was tested, one-by-one, by one of our engineers. Results were placed in a spreadsheet and any failures were investigated by the team. This was a tedious but necessary step to identifying several bugs within the system.

## User-level Testing

We worked with the Client to procure funds to run test instances of our project in Amazon's Web Services (AWS) cloud infrastructure. The Client was able to access the project remotely and securely to view the progress and perform their own user-level testing. Additionally, our team thoroughly used the C2 frontend to test and provide feedback on the system as a whole. This cloud-hosted solution allowed all of us to test releases of code simultaneously. Feedback on user experience was provided to the C2 from both the Client and the Team and design changes were made to improve usability.

## Project and Risk Management

### Task Decomposition & Roles and Responsibilities

For our project we split the team into two sub teams that would both focus on a separate area and then merge together towards the end. The roles and responsibilities were as follows:

Daniel Limanowski - Frontend Lead  
Vijay Uniyal - Frontend Developer  
Justin Roepsch - Frontend Developer  
Paul Chihak - Implant & EDR Testing Lead  
Adam Crosser - Implant & EDR Testing Developer  
Logan Kinneer - Implant & EDR Testing Developer

As we can see we had the split our group into an Implant Team and a Command & Control Team. The Implant teams main goal was to implement Domain Fronting, deliver payload malware, stealth, and bypass EDR. While the purpose of the C&C team was to implement User authentication, websockets, encrypted HTTPS communication, and logs. Although each team had even more tasks these were just the basic deliverables.

### Project Schedule - Gantt Chart (Proposed vs. Actual)

The timeline is split into two 15-week semesters (excluding Finals week, in which no work will take place). The timeline is annotated via Gantt Charts that highlight the beginning and end of each major task for the project.

Our proposed Gantt charts are as following:



Task	WEEK 1 AUG 20	WEEK 2 AUG 27	WEEK 3 SEP 3	WEEK 4 SEP 10	WEEK 5 SEP 17	WEEK 6 SEP 24	WEEK 7 OCT 1	WEEK 8 OCT 8	WEEK 9 OCT 15	WEEK 10 OCT 22	WEEK 11 OCT 29	WEEK 12 NOV 5	WEEK 13 NOV 12	WEEK 14 NOV 26	WEEK 15 DEC 3
Develop project description	█	█													
Define scope with client			█	█											
Create/expand detailed project plan					█	█	█	█	█	█	█	█	█	█	█
Research technologies					█	█	█	█	█	█					
Research concepts to be implemented											█	█	█	█	█
Write test code using technologies												█	█	█	█
Acquire access to repo, cloud, tools									█	█	█				
Set up development infrastructure, some of which in cloud														█	█

Table 1: Semester 1 Proposed Gantt Chart

Semester 1 will required extensive planning, meeting with the client, and setting up necessary infrastructure and tools. We spent time conducting research on the technologies we would be using for the project as well as proper procedures for successfully completing the project. Our subteams worked together and helped each other learn the different programming languages and frameworks that we used to build this fully-custom application and malware bot. We updated our client throughout the semester and implemented any feedback we received. Finally, we wrote code using our official technologies in order to advance towards actual implementation of project goals.

Task	WEEK 1 AUG 20	WEEK 2 AUG 27	WEEK 3 SEP 3	WEEK 4 SEP 10	WEEK 5 SEP 17	WEEK 6 SEP 24	WEEK 7 OCT 1	WEEK 8 OCT 8	WEEK 9 OCT 15	WEEK 10 OCT 22	WEEK 11 OCT 29	WEEK 12 NOV 5	WEEK 13 NOV 12	WEEK 14 NOV 26	WEEK 15 DEC 3
Develop project description	█	█													
Define scope with client			█	█											
Create/expand detailed project plan					█	█	█	█	█	█	█	█	█	█	█
Research technologies					█	█	█	█	█	█	█				
Research concepts to be implemented											█	█	█	█	█
Write test code using technologies												█	█	█	█
Acquire access to repo, cloud, tools									█	█	█	█	█	█	█
Set up development infrastructure, some of which in cloud														█	█

Table 2: Semester 1 Actual Gantt Chart

As we can see in our Proposed vs Actual charts the timelines vary only slightly for Semester 1. Most importantly is that we did not receive access to our repo, cloud, tools until we were already in Semester 2 extending that timeline quite a bit. We also took another week for researching technologies, but other then that we were quite close to our proposed Gantt chart.

Task	WEEK 1 JAN 14	WEEK 2 JAN 21	WEEK 3 JAN 28	WEEK 4 FEB 4	WEEK 5 FEB 11	WEEK 6 FEB 18	WEEK 7 FEB 25	WEEK 8 MAR 4	WEEK 9 MAR 11	WEEK 10 MAR 25	WEEK 11 APR 1	WEEK 12 APR 8	WEEK 13 APR 15	WEEK 14 APR 22	WEEK 15 APR 29
Secure API channel	Red	Red													
Implement domain fronting			Blue	Blue	Blue	Blue	Blue								
Bypass EDR						Blue	Blue	Blue	Blue	Blue					
Expand bot persistence								Blue	Blue	Blue	Blue	Blue	Blue		
Implement bot destruction								Blue							
User authentication			Green	Green	Green										
Detailed action logging				Green	Green	Green	Green								
Create malware builder (in-app)							Red	Red	Red	Red	Red	Red			
Enable websockets for realtime updates							Green	Green	Green						
Dockerize the app									Green	Green	Green	Green	Green		
Code testing	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red
Documentation	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red

Table 3: Semester 2 Proposed Gantt Chart

Semester 2 required much less planning and time was primarily spent on implementing features for the project. We kicked off the semester by working as a whole to implement secure communication between the Command-and-Control Server (C2) and the bot. For this, we had to write code on both parts of our project and test it to make sure that they mesh and the resulting communication prevents outsiders from snooping on the bot communication traffic. Next, the individual sub-teams broke off and implemented their respective functionality. Near the end of the semester we tested all of the code and wrote documentation for the client.

Task	WEEK 1 JAN 14	WEEK 2 JAN 21	WEEK 3 JAN 28	WEEK 4 FEB 4	WEEK 5 FEB 11	WEEK 6 FEB 18	WEEK 7 FEB 25	WEEK 8 MAR 4	WEEK 9 MAR 11	WEEK 10 MAR 25	WEEK 11 APR 1	WEEK 12 APR 8	WEEK 13 APR 15	WEEK 14 APR 22	WEEK 15 APR 29
Secure API channel	Red	Red													
Implement domain fronting			Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue					
Bypass EDR						Blue	Blue	Blue	Blue	Blue	Blue				
Expand bot persistence								Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Implement bot destruction								Blue							
User authentication			Green	Green	Green										
Detailed action logging				Green	Green	Green	Green	Green							
Create malware builder (in-app)							Red	Red	Red	Red	Red	Red			
Enable websockets for realtime updates											Green	Green	Green	Green	
Dockerize the app										Green	Green	Green	Green	Green	Green
Code testing	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red
Documentation	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red

Table 4: Semester 2 Actual Gantt Chart

For the actual Gantt 2 chart we can see a few differences that occurred. First off the Domain fronting/EDR/Bot persistence took a little more time to finish. The biggest difference is that websockets

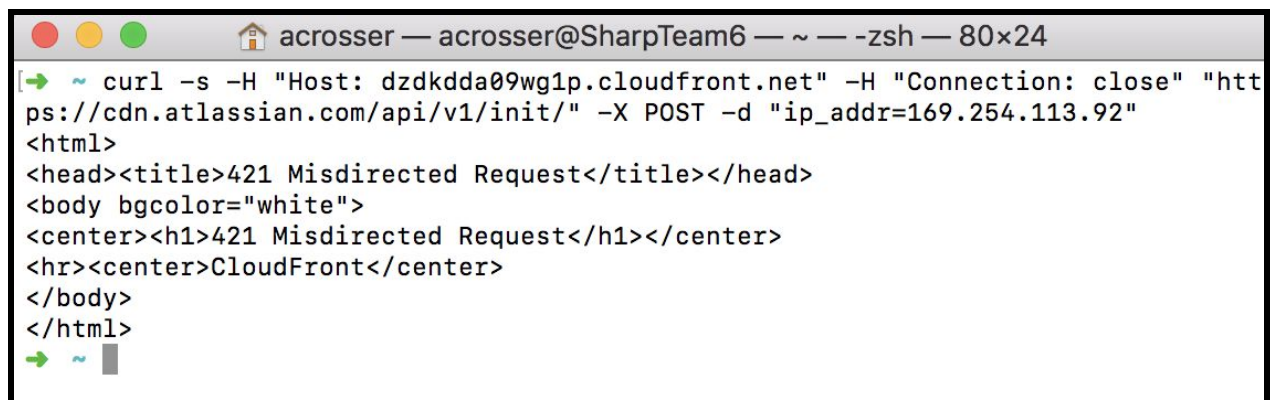
were pushed back due to other features that had to be implemented first. As such the dockerize was pushed back also. Otherwise we were fairly close to the proposed semester 2 chart.

## Risks and Mitigation

One interesting risk that materialized during the course of our project is Amazon's mitigation of domain fronting in the Cloudfront Content Delivery Network (CDN) available in Amazon Web Services. Amazon had previously published an article that they had fixed domain fronting [1], however, we will still be able to successfully perform domain fronting almost a year after that article was published.

A few weeks before handling in the deliverable for our project Amazon actually fixed domain fronting in Cloudfront even though it had worked for a year after publishing the article that they "fixed it". To do this they added in a check that the domain being used for domain fronting is actually associated with the cloudfront redirector specified in the HTTP host header.

If this does not match then Cloudfront returns an HTTP 421 error code. Example evidence given below:



```
acrosser — acrosser@SharpTeam6 — ~ — -zsh — 80x24
[➔] ~ curl -s -H "Host: dzdkdda09wg1p.cloudfront.net" -H "Connection: close" "https://cdn.atlassian.com/api/v1/init/" -X POST -d "ip_addr=169.254.113.92"
<html>
<head><title>421 Misdirected Request</title></head>
<body bgcolor="white">
<center><h1>421 Misdirected Request</h1></center>
<hr><center>CloudFront</center>
</body>
</html>
➔ ~ █
```

This is something that represents an unknown unknown in our project. We unfortunately did not anticipate this as being a risk and the risk materialized. Fortunately, with the framework we have developed as part of this project it will be easy to implement domain fronting for another CDN. However, this was not done during the project since we were already at the end of the semester and did not have time to implement any additional features.

One of our big anticipated risks was the feasibility of the deliverables we have promised. Each statement we made about the final deliverables of the project counts as a risk if we failed to deliver or any of those deliverables ended up not being realistic/feasible. It would've been a huge blunder to be naive and over-promise on what we end up providing at the end of these two semesters. Time constraints are hard to

calculate early on and as we progressed through our projects and ran into obstacles, it's easy for the final product quality to be diminished as a trade-off for completing the project on time.

What actually ended up happening was we delivered quite well on our deliverables. Although some things took longer than others, we budgeted our time well to allow a buffer so we were able to complete everything by the time the project is due. The biggest way we mitigated any time delay was simply strong communication between teammates and constant meetings with our client to ensure we were all on the same page.

## Lessons Learned

Overall we as a team learned an incredible amount throughout the process of this project. A lot of us were inexperienced in the technologies used, so the obstacles we overcame just for the research part of this project weren't easy. We were able to create a custom implant that could be deployed to various client machines through an easy to use web-application. All while making sure we had correct security measures in place. I'd say the most important lesson we learned was how vital communication is to such a large scale project. Not being on the same page can easily create huge setbacks for the team.

## Conclusions

SDMay19-19 thoroughly enjoyed working with [REDACTED] during the process of designing and developing the custom implant. It is hoped that the deliverables provided as part of this senior design project can be extended by [REDACTED] and used to improve the red team services line at [REDACTED].

SDMay19-19 is available to answer any questions related to the provided implant and other deliverables that [REDACTED] may have while working to operationalize the tools provided as part of this project.

It is hoped that [REDACTED] will continue to utilize Iowa State University's skilled engineering students to supplement internal projects and service-line capabilities.

## Future Work

The purpose of this section is to provide guidance to [REDACTED] on operationalizing and continued improvements of the deliverables provided by SDMay19-19. The two most important things that SDMay19-19 recommends that [REDACTED] focus on are implementing a Zero Trust/Google BeyondCorp model for securing access to Red Team infrastructure and implementing a malleable C2 framework for the implant.

### Implement Google BeyondCorp (Zero Trust) Security Model

SDMay19-19 recommends that [REDACTED] implement the Google BeyondCorp/Zero Trust security model [3] to secure access to red team infrastructure. This includes placing the command and control application behind an identity aware proxy which performs device and user authentication (password + FIDO U2F as a second factor) in order to control access to the implant control server.

The core principles of this security model state that both user and device authentication should occur as part of the authentication process and that all authentication should flow through a centralized identity aware proxy responsible for calculating a trust score used to determine what applications a user is allowed to access.

This allows for things such as only allowing a user to access the implant control server when logging in from a [REDACTED] managed device using U2F authentication. Less secure methods of authentication such as logging in from a mobile phone using SMS based two-factor authentication can then be restricted to only allow access to less sensitive services such as email or slack. Additionally, SSH access to servers should be provisioned using time-limited signed SSH certificates as opposed to hardcoded SSH keys. This can be accomplished by utilizing services such as those provided by ScaleFT or Pritunl-SSH.

Only allowing users to authenticate from specific devices which have been assigned to them makes it more difficult for an attacker using stolen credentials to utilize them successfully. Furthermore, U2F provides a high degree of security as it cannot be relayed like other authentication mechanisms. This is accomplished by signing the URL of the site the token is created for so even if a user is phished and tricked into submitting a U2F token to the site it cannot be used to impersonate the user on another site.

This can be accomplished by utilizing services such as Okta which provides both device authentication, FIDO U2F, and BeyondCorp style capabilities such as conditional access control

and trust based authentication. An identity aware reverse proxy can then be placed in front of the implant control application.

This type of security model can also be implemented by utilizing services such as Azure AD Conditional Authentication depending on the tech stack utilized by [REDACTED].

Taking these steps is critical as sensitive data is often exfiltrated from a target environment during a red team operation to prove successful completion of the objective. Furthermore, the implant server allows for control of computers and other assets owned by the customer and irreparable reputation damage could occur if [REDACTED]'s infrastructure is successfully compromised by an adversary.

## Implement Malleable C2 Framework

SDMay19-19 recommends that a malleable C2 framework [2] is implemented in order to allow for the implant to continue to function in the long term. In the short term this is less important due to the custom nature of the implant the C2 traffic will not be signed. However, with continued use of the implant work will need to be done in order to allow for continued evasion of network security monitoring solutions such as Palo Alto NGFW, Symantec/Bluecoat Proxies, etc.

## References

- [1] MacCarthaigh, Colm. "Enhanced Domain Protections for Amazon CloudFront Requests | Amazon Web Services." *Amazon*, Amazon, 27 Apr. 2018, [aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/](https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/).
- [2] Mudge, Raphael. "Rsmudge/Malleable-C2-Profiles." *GitHub*, 7 Apr. 2018, [github.com/rsmudge/Malleable-C2-Profiles](https://github.com/rsmudge/Malleable-C2-Profiles).
- [3] Ward, Rory, and Betsy Beyer. "BeyondCorp: A New Approach to Enterprise Security." ;*Login: The Usenix Magazine*, Dec. 2014, pp. 6–11.