Team SDMAY19-19: Offensive Security Orchestration

DESIGN DOCUMENT

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

Revised: December 3 2018 / Version 2.0

Table of Contents

Table of Contents 1

Introduction 3

Problem and Project Statement 3

Operational Environment 3

Intended Users and Uses 3

Assumptions and Limitations 3

Expected End Product and Deliverables 4

Deliverables for the Implant Team (C# Back-End) 4

Deliverables for the Command and Control Team (Web App Front-End) 4

Design Specification and Analysis 5

Design Requirements 5

Command-And-Control Requirements 5

Implant Requirements 5

Design Overview 6

Design Diagram 7

Domain Fronting Redirector 8

Design of Command-And-Control Element 9

Implant Design 10

Cuckoo Sandbox Service 11

Builder Microservice 11

Design Analysis 11

Potential Downsides of Design Decisions 12

Testing and Implementation 14

Interface Specifications 14

Hardware and Software 14

Functional Testing 14

Non-Functional Testing 15

Implementation Issues and Challenges 16

Results 17

Closing Material 18

Conclusion 18

References 18

Introduction

Problem and Project Statement

Our project aims to develop a security orchestration platform for our client which will allow them to conduct red team engagements in a stealthy and efficient manner. Since our client often uses widely deployed and standardized tools during red team engagements they frequently run into issues with the tools they use being flagged by network security teams. By developing a custom implant which integrates into the security orchestration platform we will be able to bypass the tools which flag on more common implants. The other issue our client encounters is that developing red team processes is a manual and time consuming endeavor. This is a result of needing to test implants against multiple endpoint detection solutions to see if the implant will be discovered. To address this issue the security orchestration platform will automate the deployment of implants so that our client can rapidly develop and test different payloads.

Operational Environment

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”).

Intended Users and Uses

The intended users of our project will be the company red team. We will have various levels of security and user authentication to ensure there is no misconduct with the tool. There will be a history log of all actions done along with Admin and Regular accounts. They will be able to utilize our project to conduct engagements in a stealthy manner. This tool will be able to bypass normal security tools that commonly flag regular implants. Also with our automated deployment system, they will be able to rapidly test multiple different payloads to streamline testing.

Assumptions and Limitations

The project does have a cost of needing AWS credits so we are under the assumption that our company client will be paying for the costs. AWS access stands for Amazon Web Services which is a cloud hosting system that’ll allow us to host our server/application as well as utilize potentially 3 VMs.

Another assumption is that we will be able to complete all of the deliverables on time. As we reviewed all of them we found out that we had two very difficult deliverables. Those were, getting results from the EDR (endpoint detection and response) and the In-App malware builder

that'll use the web app to compile malware with variables. Being able to complete these on time isn't an easy task due to the difficulty they present.

The project and corresponding system will comply with the following limitations:

- The C2 will use ReactJS framework for the user interface SPA
- The C2 backend will use Django
- The C2 will communicate to the bots via REST APIs and be considered "RESTful"
- The EDR solution is restricted to whatever endpoint protection services we are able to get versions or trials for. Some proprietary/commercial software is hard to obtain legal licenses for, such as trials.

Expected End Product and Deliverables

Deliverables for the Implant Team (C# Back-End)

- Domain Fronting using Amazon Cloudfront
- Macro to deliver payload malware through automated phishing
- Scheduled task persistence while remaining stealthy
- Bypassing endpoint detection and response solutions

Deliverables for the Command and Control Team (Web App Front-End)

- User authentication with multiple levels of access
- Sockets for bi-directional communication between implants and controller
- Encrypted communication (HTTPS)
- Logging of actions taken by every user
- Admin action page for actions such as user management
- Dockerize application to standardize deployment
- Creation of help pages for common actions a user will take
- In-App Malware tester that allows users to quickly test different payloads against EDR solutions

Design Specification and Analysis

This section describes the overall design of our solution as well as an analysis of the design which discusses the various design considerations we had to choose from when designing the overall architecture plan for our project.

Design Requirements

Before we began the design process we first met extensively with the client to lay out all the desired and necessary requirements. From these meetings we identified that our design must meet the following requirements in order to satisfy the needs of the client.

Command-And-Control Requirements

- User authentication with multiple levels of access
- Sockets for bi-directional communication between implants and controller
- Encrypted communication (HTTPS)
- Logging of actions taken by every user
- Admin action page for actions such as user management
- Dockerize application to standardize deployment
- Creation of help pages for common actions a user will take
- In-App Malware tester that allows users to quickly test different payloads against EDR solutions

Implant Requirements

- Domain Fronting using Amazon Cloudfront
- Macro to deliver payload malware through automated phishing
- Scheduled task persistence while remaining stealthy
- Bypassing endpoint detection and response solutions

Design Overview

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 “attack” can control an implant using the web application. The operator will be 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 environment.s

Outside of this there will be 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. The functionality of the builder is described in more detail in later sections of the design document.

Design Diagram

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.

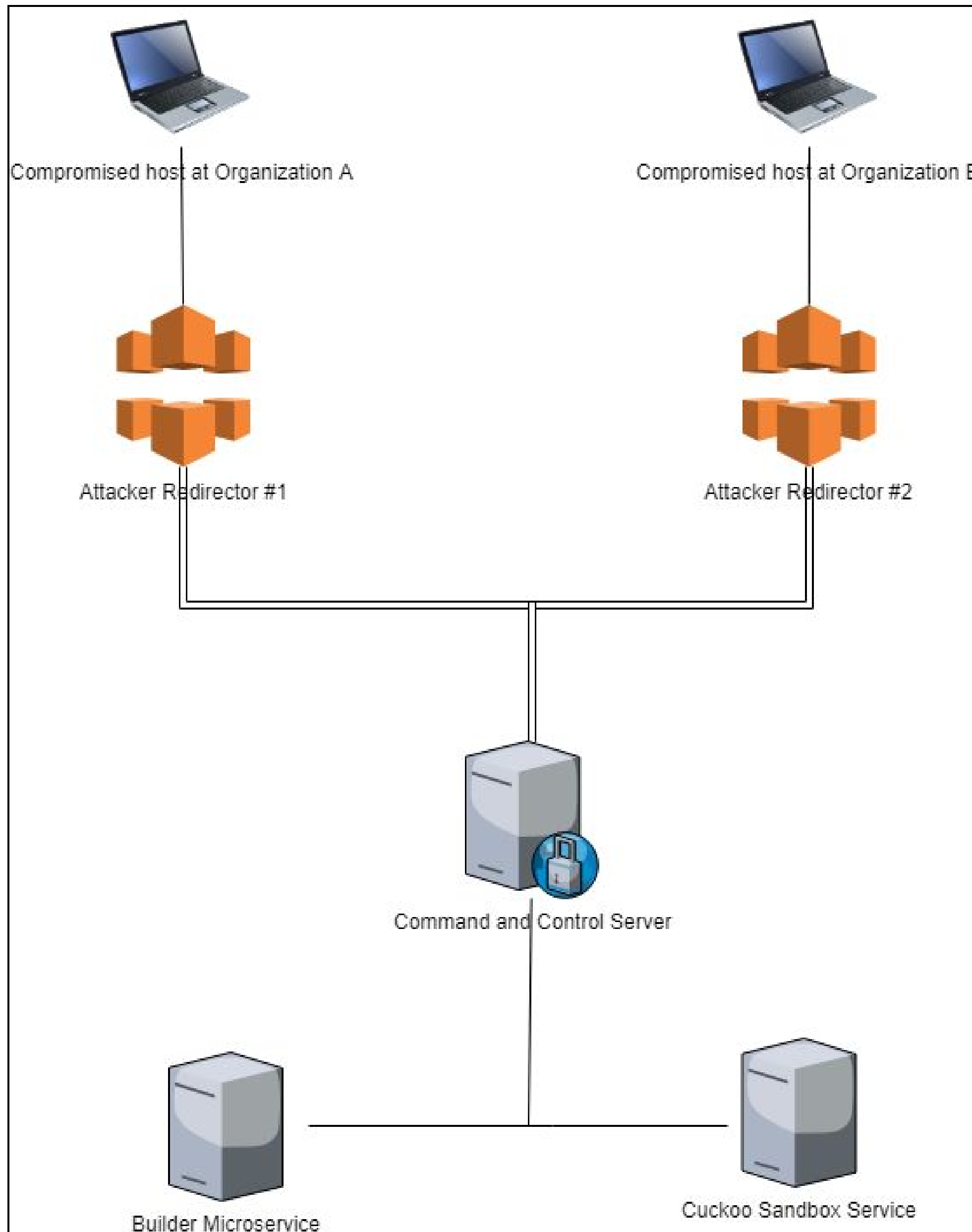

Figure 1: Design Diagram

Domain Fronting Redirector

Our current implementation only supports one type of redirector. This redirector can be used by the implant to perform domain fronting. Domain fronting is a technique that can be utilized to “hijack” a legitimate and trusted domain and use it for command and control purposes.

Domain fronting only works in scenarios where the victim domain is hosted on a CDN, such as Amazon CloudFront. The technique works by exploiting the way CDNs handle TLS termination. Due to the computational overhead of handling TLS connections encryption is typically handled by specialized hardware devices specialized for these purposes.

Once the TLS layer is removed the request is then routed by the backend infrastructure using the host header to point to a specific CloudFront redirector. The redirector will then either cache the response or forward the response to the backend server.

It is possible for an attacker to exploit this by setting up their own CloudFront redirector and connecting to another domain hosted on the CDN. The host header will specify the CloudFront redirector, but at the network level everything will look like a TLS connection is being made to the victim domain.

When the attacker connects to the domain that is being used for domain fronting over HTTPs the TLS certificate associated with the domain is used and any network appliance that is performing is not performing TLS inspection will not be able to tell that this malicious connection is taking place.

When the security appliance calculates the trust level of the domain it takes into account the reputation of the domain that is being used for domain fronting. So if, for instance, the domain being used for domain fronting is associated with a reputable company, the attacker will inherit the same level of trust as the victim domain.

This is a very stealthy technique and is very difficult to detect as it requires that an inline web proxy be deployed that is capable of performing TLS inspection. The network defender must then identify instances where the Host header being used does not match the domain associated with the TLS certificate. There are a number of legal and technical hurdles to performing TLS inspection as it is necessary to account for both privacy laws as well as it requires that all computers have a root CA installed that can be used by the proxy to decrypt and then re-encrypt TLS connections.

Amazon has stated that they have taken actions to prevent domain fronting, but if you look at the image below on December 2nd, 2018 it was still possible to perform domain fronting. This example shows how we are able request resources hosted on an amazon subdomain while making it look like we are actually connecting to press.pwc.com by mismatching the host

headers. When we try to access the resource without setting the host header to a different value we get a 404 error as the resource does not actually exist on the press.pwc.com site.

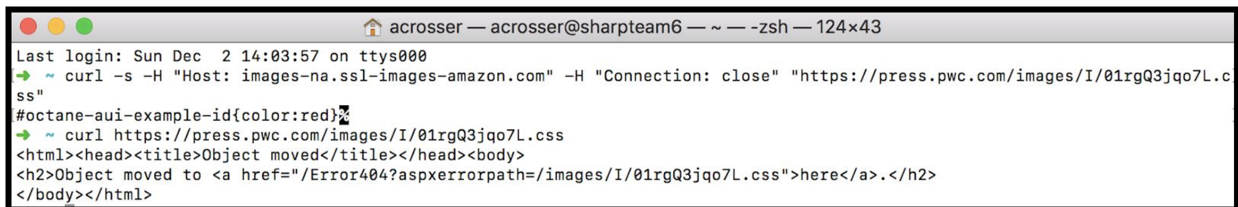

Figure 2: Domain Fronting Tests

Design of Command-And-Control Element

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

Frontend

- ReactJS UI framework
- Semantic UI CSS/JS framework

Backend

- Django Python web framework
- Django REST Framework

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. The database will be SQLite to begin with, and if resources demand it, we will look at moving to a dedicated database service such as MariaDB or Postgres. Because this application is going to be used in the short-term (one week to a few months), a single-file database may reduce complexity and make it easier to backup or extract the data for our client.

We 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.

Implant Design

The implant will be 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++. 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.

We chose to use Amazon CloudFront for domain fronting due to the large number of available domains hosted there which could be used for domain fronting. 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 in order to modify the resources section of a Windows Portable Executable (PE) file you must utilize proprietary libraries provided by Microsoft which are only available on Microsoft Windows.

During the research phase we were not able to identify any third parties libraries that were readily available that would satisfy this use-case using a unix-based microservice. We decided that it would probably be better to just run a windows-based microservice written in C# that implements the builder instead of taking the time to develop a library for modifying the resources section of portable executable files.

The command and control web application will make API calls to the builder microservice when the 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.

Future expansion will allow for the builder to be used to tailor the functionality of the implant to a specific target. While we will not be implementing this functionality during the course of our project, we are laying the groundwork for future teams to expand on the project and add additional customization options to the builder component.

Design Analysis

At this point in time, our code base consists of a simple reconnaissance implant in C# that was provided by the client. The implant can retrieve and send data to our C2 prototype server as well as run commands on an infected host. In addition to the implant, we have a working model

of the C2 application prototype. This barebones application utilizes our previously-researched technologies and frameworks.

Our team has met with the client to discuss what they would like completed during the course of this project. So far we have, in depth, defined the scope of the project and desired outcomes.

Besides defining scope, our team has been meticulously researching methods of implementation for the each of the demanding requirements for this project. Our C2 and implant modules require significant low-level knowledge of the Windows Operating System as well as web application development, so at this time everyone is familiarizing themselves with the technologies they will be using before implementation. Any missteps or wrong paths taken during this project can severely hinder progress down the road with this project, so research is being taken very seriously in these initial stages.

Research is beginning to pay off - our team is having more in depth discussions about proper ways to implement certain features, and members are now understanding the technologies they will be working with.

Our thoughts on the design are as follow: we believe we have a lot of highly-technical and challenging work ahead of us, but we have carefully selected the proper technologies and implementation routes to ensure that we provide our client with a stable and professional product that they can use in their Red Team Engagements.

Potential Downsides of Design Decisions

Every choice made for this project is backed by our personal 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.

We have decided to support application deployment only via microservice containers with Docker. The reasoning behind using Docker is that we figure the client will be using our 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, we made the tradeoff to only support Docker instances of the C2 application.

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.

Testing and Implementation

Interface Specifications

The initial loader will be distributed through a word macro and it will therefore need to interface directly with Microsoft Word. From there the loader will download the implant and write it to C:\Windows\Temp. Communication between the command and control server will be handled with HTTP/S and will allow for the implant to receive commands from the control server using REST API's.

Hardware and Software

There are numerous tools that we will be using to test our design however since most of them cannot be automated we will have to test them manually. The first of these tools is that we will be relying heavily on Microsoft Word to test the macros generated by DotNetToJScript. This is because the results that DotNetToJScript outputs are not easy to read and so it makes more sense to just load the macro and test it. Another tool that we will be using is wireshark and tcpdump to determine if our implant are using the fronted domain. We chose to use these tools because our team is familiar with them and they provide an easy interface to view where the packets are being sent as well as seeing if the proper REST API's are being used to send data. We will also being using several Windows management tools to ensure that our implant exists in the correct context. These tools are Windows Process monitor and Process Explorer. Both of these are free tools from the Windows Sysinternals and they monitor and display in real-time all the file system activity on a Windows operating system. Finally we will also be using Cuckoo Sandbox to allow us to learn what identifiers each payload we develop has and use those identifiers to determine the likelihood of an EDR solution detecting the payload.

Functional Testing

Since our team has determined the cost/benefit ratio of implementing automated testing does not have a high enough payoff we will be conducting mainly manual testing. There are several functional requirements that we have identified and will be testing to ensure that they are met. On the implant side we will be testing to ensure that the implant communicates with C2 over a secure, encrypted API, and also that the implant is using a fronted domain. This will be tested using wireshark or tcpdump to view the packets and make sure they are both encrypted with an SSL certificate, and they are going to the correct APIs. Next due to request by our client we want to ensure that the implant has the means to destroy itself. This has been implemented via a command that will cause the implant to delete its files and then destroy the process. We can test this using Process Monitor to ensure that the process is destroyed, and File Explorer to ensure that the necessary files are deleted.

Furthermore, we want to test that our implant can demonstrate persistence while remaining stealthy. For this we have a two part test case. For part 1 we want to ensure that the implant can maintain persistence across a reboot. This can easily be tested by having the implant set up its persistence routine, and then rebooting the machine and seeing if the implant opens a connection on boot. Then we want to make sure that the persistence mechanism is stealthy. In order to test this we have to make use of another portion of our project to test against EDR identifiers. To achieve this we will have a Cuckoo Sandbox test the implant and see what identifiers the implant leaves behind after running. Then we will determine what the likelihood of an EDR solution detecting on those identifiers is.

Additionally there are also several C2 functional requirements that we have to test. These requirements will almost entirely be tested by running the application and ensuring that the correct commands are being sent to the implant using either the logs generated by the application or by using wireshark to view the packets. For instance ensuring that the UI is a single page application, uses an encrypted communication, has multi-user creation/deletion/authentication, logs all activity by users, uses realtime websockets for receiving data, and includes documentation for new users can all be done by having a tester run the application and ensure that each portion is functional. We are heavily debating the use of ReactJS unit tests for testing the front end because that is the portion of our project that would make sense to automate the testing of since it is the only part that doesn't require multiple technologies to test.

Non-Functional Testing

For testing non-functional requirements we were not able to determine anyway to automate the test cases so for both the C2 and implant everything will be tested manually. There are several test cases on the implant side that will be fairly hands on to test. For instance making sure that there is not predictable network traffic requires someone to run wireshark and check that the heartbeat is not being sent out at regular intervals. Next we want to make sure that when installing the implant that it will not be detected by the victim user. To test this we will be running the word macro to kick off the implant and then making sure that there are no virus warnings or command prompts being opened. After that we want to make sure that the bot can destroy itself if it cannot locate the C2 server. To do that we would put it on an isolated network and test if it will actually destroy itself or just sit there trying to contact the C2 server which would against be tested using Wireshark. Finally we want the implant to be configurable and support multiple deployment options which we were not able to determine a good method of testing but we plan on having a XML document embedded in the implant that will have all the deployment options and then any modifications to that XML will then create a new deployment option.

On the C2 side of things we want the user to be able to navigate the application freely without interrupting any ongoing processes. So they should not be able to switch between implants without losing any information or interrupt the response of previous commands. This will be tested by setting up multiple different implants and then issuing commands to both and then

using multiple sections of the web application at once. Then we can check to see if the results of the commands came back as they should or if they were lost. Next we want multiple users to access the application simultaneously which can be tested by simply creating multiple users and having them both issues commands and try and use the application without interrupting service. Finally we want to ensure that the C2 can be quickly deployable in a temporary state which can be tested using docker-compose to launch the application within a docker container and then can easily be torn down by closing the container.

Implementation Issues and Challenges

Our current setup has several implementation issues that we are attempting to overcome. Many of these issues pertain to testing against EDR solutions. The biggest issue was simply that we did not have the funding to obtain a copy of the major EDR solutions that corporations are using and therefore would not be able to test against them. The second biggest issue is that each EDR solution has its own proprietary method of giving results and so in order to pull each result from those would require reverse engineering the solution. Which would only work until that solution updated the method that results are shown through in which case the new method would also have to be reverse engineered which can be a very time consuming process (Caparas, Marianne). Therefore we decided that instead of trying to pull a direct yes or no answer on whether or not a certain EDR solution will detect a payload we decided to instead take the payload and run it through the open source tool Cuckoo Sandbox which will give identifiers based on the payload. These identifiers are modifications that the payload makes to the operating system as well as network traffic that the payload creates (Guarnieri, Claudio). Using these identifiers and some knowledge of what EDR solutions will typically alert on our client can deduce whether or not a certain payload will be detected.

Another implementation issue that we have come across is that we will need multiple servers setup which will make automating the build process significantly harder. For instance we will need a machine for Domain Fronting, a Cuckoo Sandbox instance, and a machine that Cuckoo Sandbox can test payloads against, in addition to the docker container that will run the web application. One of the functional requirements for our project is that it should be easy for our client to deploy the project and have everything just work. However to do that we will need to find some way to automate the provisioning of all of those different servers as well as our client will have to have all of those additional servers ready to go in order to fully use the application thus increasing the cost.

Finally the last major issue that we have encountered is that C# and DotNetToJScript are not very friendly when it comes to developing features that use Windows at a low level. Therefore we are heavily limited on what features are possible to be implemented. For instance while attempting to implement process injection to increase the stealthiness of the application we found that it was impossible to execute win32 api calls from JScript and therefore couldn’t inject the process at all (Christensen, Lee). Therefore we have to find new ways to exploit Windows, or limit the functionality of our implant.

Results

This project is a two-semester project and is still in progress. Therefore we do not have conclusive results. However, we do have preliminary calculations, estimations, and details from our research and time spent developing this project so far. The current results we have obtained can be found in the Design Analysis and Testing and Implementation sections of this document.

Closing Material

Conclusion

So far, our team has done extensive research, discussion, and some amount of testing various methods for the implementation of our deliverables. This process allowed verification that our different ideas were technically feasible, which was extremely important for the implant element, as it requires less frequently used techniques.

Our overall goals are divided into two main categories: Command and Control, and Implant. Goals in the Command and Control category consist of requirements that result in making a webapp more secure, more user-friendly, have more functions, and be easier to set up. Goals in the Implant team consist of setting up domain fronting, creating a malicious macro for Word, allowing tasks to persist stealthily, and bypassing endpoint detection and response.

To achieve these goals, the first step in our plan was to split up the team into two subteams to work on either category of the goals. This was chosen to be done as the requirements needed for each can be done without much reliance on the other one. This allows us to get work done in parallel, and the separation into subteams formalizes this. For the Command and Control team, our plan is to start with Docker before working on the other requirements. This is because getting Docker set up correctly will make the process of working on everything else go faster. For the implant team, our plan is to start with the malware loader, as it is a prerequisite to work on the other requirements.

References

1. Caparas, Marianne, and Nick Schonning. “Overview of Endpoint Detection and Response Capabilities.” Capabilities | Microsoft Docs, 2 Sept. 2018, docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-atp/overview-endpoint-detection-response.
2. Christensen, Lee. “EmpireProject/PSInject.” GitHub, 11 Apr. 2017, github.com/EmpireProject/PSInject.
3. Guarnieri, Claudio. “Cuckoo Sandbox Book.” Cuckoo Sandbox Book - Cuckoo Sandbox v2.0.6 Book, The Honeynet Project, 20 Sept. 2017, docs.cuckoosandbox.org/en/latest/.
4. Limanowski, Daniel. “b1tst0rm/Django-React-Boilerplate.” GitHub, 8 July 2018, github.com/b1tst0rm/django-react-boilerplate/blob/master/README.md.