Team SDMAY19-19: Offensive Security Orchestration

PROJECT MANAGEMENT PLAN

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 3.0

Table of Contents

Table of Contents 1
- List of Figures 2
- List of Tables 2
- List of Symbols 2
- List of Definitions 2

Introduction 3
- Problem Statement 3
- Project Deliverables and Specifications 4
  - Deliverables for the Implant Team 4
  - Deliverables for the Command and Control Team 4
- Previous Work / Literature Review 5

Proposed Approach and Statement of Work 7
- Functional requirements 7
  - Bot requirements 7
  - Command and Control requirements 7
- Project Architecture 7
- Constraints considerations 8
- Technology considerations 9
- Alternative considerations 9
  - FruityC2 9
  - Splunk 9
- Proposed Design Diagram 10
- Assessment of Proposed Solution 10

Validation and Acceptance Test 13

Project Timeline 16

Risks / Feasibility Assessment, Cost considerations: 18

Standards 20

Test Plan 21
- Implant Team 21
- Command and Control Team 21

Conclusion 23

References 24

List of Figures

- **Figure 1:** Project Architecture
- **Figure 2:** Project Design Diagram
- **Figure 3:** Gantt Chart Legend

List of Tables

- **Table 1:** Semester 1 Gantt Chart
- **Table 2:** Semester 2 Gantt Chart

List of Symbols

- **C2:** Command and Control Server

List of Definitions

- **Red Team:** Independent group that challenges an organization’s security and response capabilities
- **The Company:** To protect our client from any potential legal issues they will remain unnamed and only referenced as our client or the company
- **Command and Control Server (C2):** The web application that controls, manages, and manipulates our implants
- **Penetration test:** Focuses on finding as many vulnerabilities/misconfigurations as possible
- **Endpoint Detection and Response (EDR):** tools primarily focused on detecting and investigating suspicious activity on hosts
- **Implant:** malware that is planted on a victim machine also referred to as a bot
- **Phishing:** technique to coerce victims into giving up sensitive information or run malicious programs
- **Institute of Electrical and Electronics (IEEE):** Organization of technical professionals seeking educational and technical advancement of electrical and computer disciplines
- **Server Name Indication (SNI):** Extension to TLS protocol handshake where the client indicates what hostname it is attempting to connect to
- **Single-page application (SPA):** A web application that interacts with the user by dynamically rewriting a single page
- **RESTful API:** an application that is based on representational state transfer (REST) technology to handle communications
- **Information Technology (IT):** study or use of computer systems for storing, retrieving, and sending information
- **Vulnerability:** weakness capable of being exploited to gain information or control
- **Virtual Machine (VM):** an emulation of a computer system

Introduction

Problem Statement

Our team will be working with our client, PwC, during our senior design project to develop a security orchestration platform which they can utilize during red team engagements. Our client is a Big Four consulting firm which provides a plethora of services from accounting and financial services to cybersecurity consulting services.

One of the service lines provided by our client is something known as a “red team” engagement. The red team engagement can be thought of as a capstone exam designed to test a mature blue team in their ability to detect and respond to an advanced attack on their network. Unlike a penetration test, the red team aims for “depth not breadth” of findings and is often targeted at identifying higher-level problems with the security program itself, whereas a penetration test tends to focus on individual vulnerabilities or misconfigurations.

The red team is tasked with working towards a given objective as specified by the client organization. For instance, an electronics company worried about a Chinese competitor stealing design documents and HDL code for their next-generation GPU would engage the red team with the task of stealing this HDL code. Another example is a healthcare company worried about an attacker breaching their database of twenty-two million social security numbers would engage the red team to gain access to this database and exfiltrate this data.

The network defense team (i.e. blue team) would be unaware that the engagement is taking place and would be expected to react to and defend against the attacks performed by the red team as if they were a real adversary. Red teaming is most effective when only one or a few high-level executives are aware of the engagement, and the red team is given unrestricted access to act as a realistic adversary. The deliverable to a client after a red team is a detailed timeline of the steps the red team took to accomplish the objective and strategic recommendations the client can take to improve their information security program.

The 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 their tools being flagged by network security teams. By developing a custom implant which integrates into the security orchestration platform we will be able to provide custom tools which will be more difficult for the blue teams to identify. 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.

Project Deliverables and Specifications

Deliverables for the Implant Team

- 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
- In-App Malware tester that allows users to quickly test different payloads against EDR solutions

Deliverables for the Command and Control Team

- User authentication with multiple levels of access
- Sockets for bi-directional communication between implants and controller
- Encrypted communication
- 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
- Download and execute new payload

Previous Work / Literature Review

There are a number of industry standard toolkits used for red team engagements. One such toolkit is the “beacon” payload which is part of the Cobalt Strike framework [1]. The beacon payload is designed to be an asynchronous agent which can be deployed to a target system in order to allow for stealthy command and control. The approach used by beacon was innovative compared to other solutions at the time, such as metasploit's meterpreter, as the asynchronous nature of the tool allowed for a more stealthy approach to communication. Other tools at the time were often noisy in regard to network communication as they would beacon at a regular interval to a control server at a very fast pace (e.g. beacon once a second at a fixed interval). This would allow blue teams to detect malware infected systems using techniques such as discrete fourier transforms, in order to detect a process connecting to a command and control server at a known interval using tools such as the RITA platform provided by Black Hills Information Security. Additionally, certain next generation firewall appliances, such as the Palo Alto Next-Gen Firewall, employ similar tactics which allow for the detection and automatic blocking of this type of beaconing behavior.

The novel technique used by beacon allowed for asynchronous communication with a randomized jitter value. The jitter value would randomize the interval used for the connection which allows for bypassing of tools such as RITA [2]. Additionally, the operator can set the beacon interval and jitter time to a custom value and the asynchronous control mechanism allowed for dynamic queueing and un-queueing of pending jobs to perform on an infected system.

The primary problems we have identified with beacon are that due to its widespread use it is often easily detected. Furthermore, the builtin commands provided by the tool often use outdated techniques that are easily detected by modern defenses. Additionally, since the tool is a closed source solution it is more difficult to customize and “cruel hacks” must be used to instrument additional functionality on top of a closed source solution.

We have also conducted a review of common frameworks used for automation of red team quality assurance testing. One such toolkit is the malice tool which is meant to be an open source alternative to VirusTotal which does not distribute samples. The purpose of the tool is to test the evasion capabilities of payloads to ensure that red team payloads are not signatured before being deployed on a target network. This process is normally extremely manual and can take valuable hours of time. This can be expensive given the high salaries typically paid to red team operators and the business case for automation quickly becomes apparent.

The malice tool supports a number of static signature scanning techniques, but does not support dynamic testing or analysis of red team payloads. This is an obvious shortcoming as modern security tools have moved away from purely static signature checking and have moved to support behavioral detection techniques.

There are also a number of tools which support behavioral analysis, but do not include the static signature analysis portion that malice does. The Cuckoo Sandbox tool does support dynamic analysis of payloads, but lacks a number of features included in malice. From our research it appears like it would be ideal to combine these tools into a unified platform which supports automated quality assurance of red team payloads.

Proposed Approach and Statement of Work

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
- Has a dedicated ReactJS single page application for managing 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

Project Architecture

The below image describes the architecture with all components at a high-level overview. The diagram shows the bots running on compromised hosts. Those bots send information through attacker redirectors accomplished via domain fronting to our C2. The client connects to and uses the C2 to control and see the bot activity.

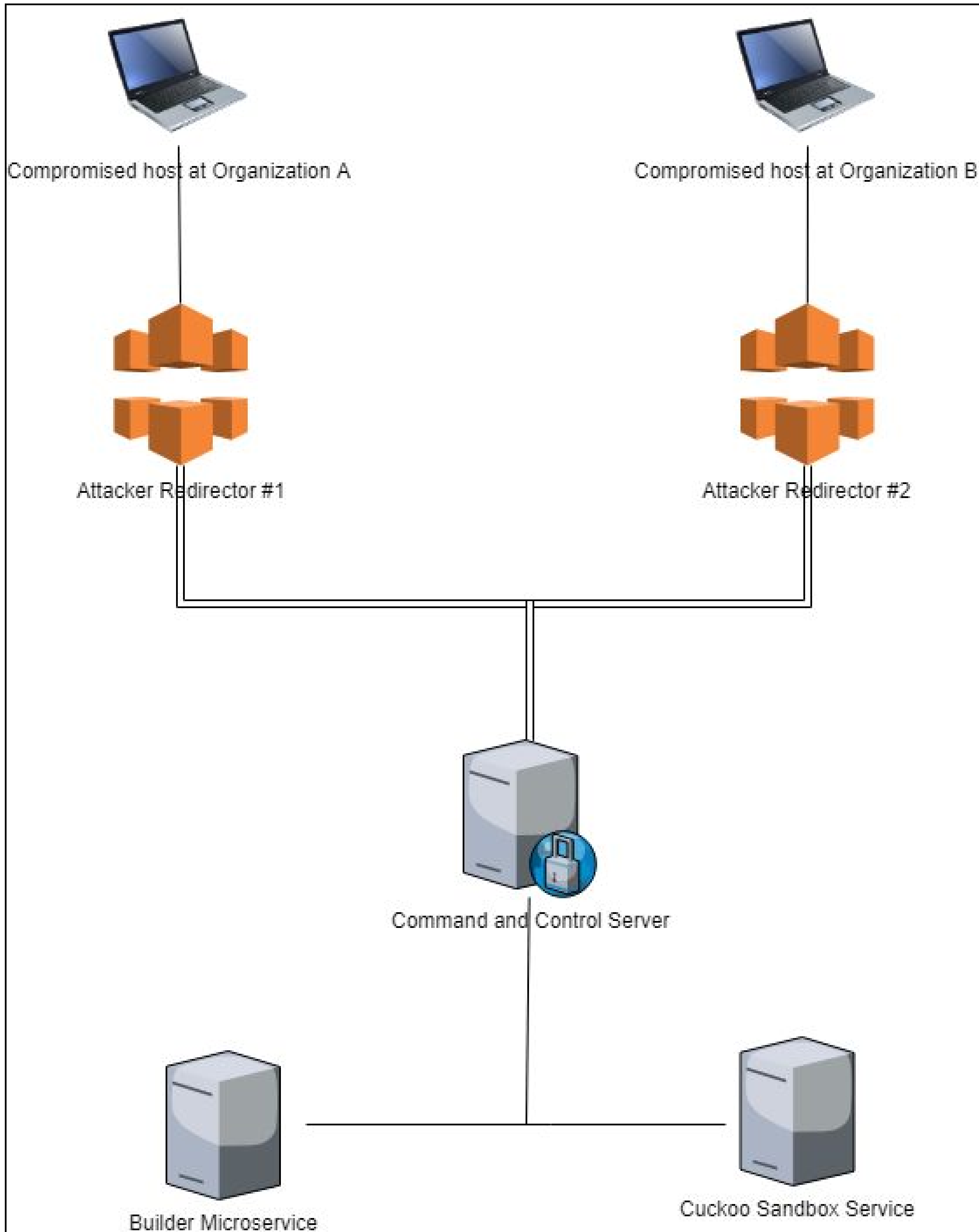

Figure 1: Project Architecture

Constraints considerations

The project system will comply with the following constraints:

- 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 obtain versions or trials for. It is difficult to obtain legal licenses for some proprietary/commercial software.

Technology considerations

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.

We decided to use ReactJS for the frontend because it allowed us to pursue a single-page-application, or SPA. An SPA does not require page refreshes as every changing item is dynamically handled in React’s virtual Document-Object-Model (DOM). An SPA fits perfectly in with the real-time nature of this project.

Django, a Python Web Framework, is excellent at handling database queries and database models. Because our application is “data-heavy,” having a tested framework to handle the data storage is much more stable than directly dealing with SQL or non-SQL-based databases.

Alternative considerations

During our extensive research and throughout our conversations with the client, we discovered several alternative paths to providing a solution to the client’s problem. Below we highlight on these alternative solutions, discussing pros and cons and ultimately why the given solution was not chosen.

FruityC2

There exists a Command-And-Control framework called FruityC2 that is open-source and readily available on Github [3].

This framework provides a near-single-page-application experience and is modular. However, the application doesn’t have support for our baseline malware and that would require code refactoring. Additionally, our existing C2 already performs all actions on our malware, and learning an entirely new framework would be a waste of time.

Splunk

Splunk is a massively successful application and log management tool. It can be expanded to manage just about any existing software or data input and allows for creation of dynamic dashboards.

Splunk would be able to handle and manage our malware. However, Splunk is a commercial tool that requires licensing. This licensing cost increases with the amount of data and usage of Splunk. Because our client wants to keep costs as low as possible, we decided Splunk was not a reasonable framework for our project.

Proposed Design Diagram

The diagram below describes at a high-level how our project will function. The diagram details the communication between our client, the C2 application, and our malware bots. The communication occurs over HTTP API Calls, and one of our primary goals this semester is to research and implement a secure, encrypted channel between the C2 and its “users” (client and bots).

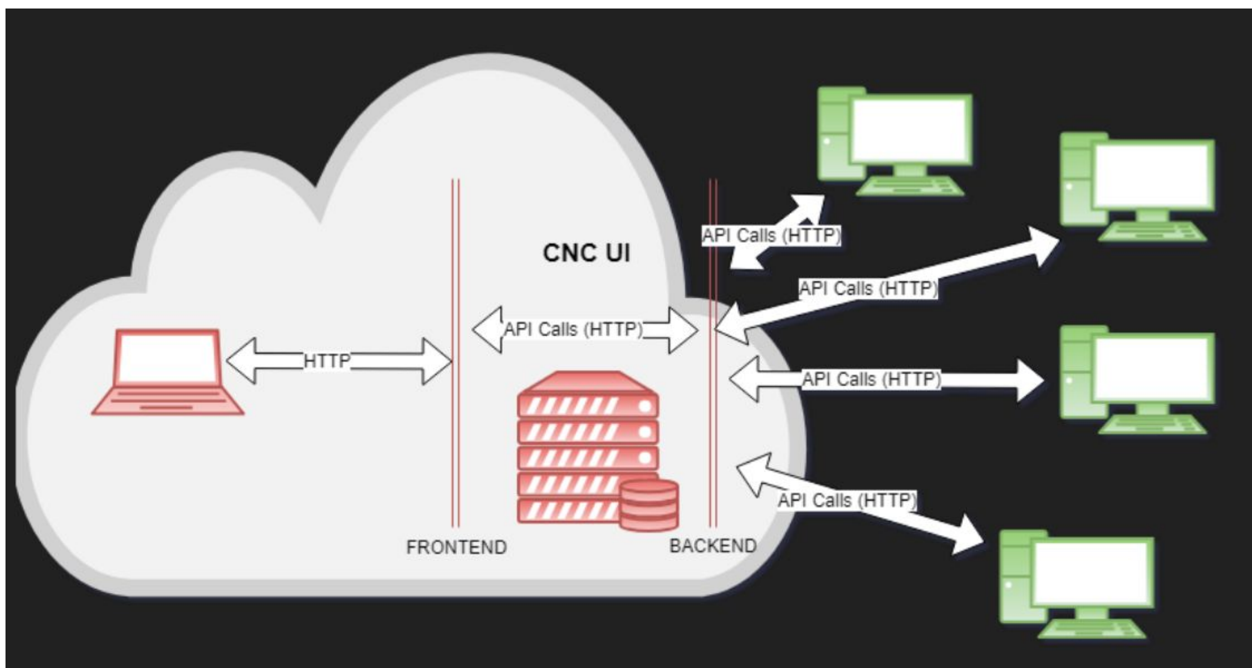

Figure 2: Project Design Diagram

Assessment of Proposed Solution

The proposed solution solves a range of troublesome annoyances that arise during red team engagements. It will offer a way to give red teams a foothold within a secured network using automated phishing campaigns that distribute macro-enabled word documents. This allows a red team to gain access without needing to conduct extensive, and expensive, reconnaissance on the target to discover remote vulnerabilities. However, the solution is not without its strengths and weaknesses.

One of the biggest strengths is how easy the solution will be to use. It is highly customizable and easy to learn as a result of being written in a relatively simple programming language (C#).

Furthermore, it is also fully integrated with a web application frontend that will allow users to deploy and manage implants without needing to know exactly how they work. This is demonstrated in Figure 1 where each implant communicates with the C2 infrastructure using API calls. Also, since one of the main goals is to save time and money on development and deployment, the application is being built with automation in mind so that users will have a limited amount of work required to get up and running with the tool.

Another one of the strengths of our project is that the solution will leverage existing tools where possible. When developing a product that will be used continually it is important to ensure that maintaining the product is easy even without extensive knowledge regarding how it works. Thankfully, when using existing solutions, they typically come with detailed documentation that make picking up the project exponentially easier. Furthermore, by leveraging existing solutions, a large amount of development time will be saved since it will not be required to recreate a product that is already available. Not to mention all the time that will be saved for the client when maintaining the product since the existing solution's developers will presumably still be pushing out updates when new issues are discovered.

We have determined to use Cuckoo Sandbox for the EDR portion of the proposed design. This is because EDR as a field is always advancing and our client does not want to be responsible for keeping up with all the changes. So by using a prebuilt solution the workload is shifted to the maintainers of Cuckoo.

Additionally, the proposed solution will be highly modular which will allow multiple developers to be working on it simultaneously without conflict. This is shown in Figure 1 by each major component being seperated and communicated through API calls. So if the implant code changes it will not affect the web app front end so long as the API calls remain the same. This saves both development time, and makes it easier to add new functionality to the final design since modifying the codebase in one area will be unlikely to break things in another. Furthermore, in order to add new functionality to the implants all that has to be done is create a new API call and handle on the web application. However, no solution is perfect and there are a few shortcomings that are important to look out for.

One issue is that it is not feasible for the product to receive a direct yes or no detection answer from an EDR solution. This is a result of a few different factors. One of those factors is that every EDR solution will have a different method of reporting threats. Furthermore, EDR solutions have no reason to want to make it easy for third parties to pull information from their solution, and so it can require a large amount of reverse engineering to learn how to do that.

Another factor is that EDR solutions are expensive, and a lot of the large commercial ones are not interested in selling one off copies for single users, instead they want a corporation to buy services for their entire company. Thus our product would not be able to test against some of the most common EDR solutions that are deployed in the real world. Therefore, as a way to bypass the limitations of having to reverse engineer results, and only having results from

smaller and cheaper EDR solutions the product will instead focus on what end points the malware is using. In order to calculate this the product will leverage existing solutions to determine what endpoints are being activated, and when, as well as any C2 traffic that could be flagged, trace api calls, and even send memory dumps of the infected host. To compensate for these two issues with the EDR we chose to use Cuckoo because it will give generic information about identifiers for the payload. This makes it so that while our client will not know directly whether or not a specific EDR will detect the payload, they will at least know what identifiers the payload has so that if it is detected they can deduce what portion of the payload triggered the EDR solution to flag it as a virus.

Another weakness is that as a result of choosing ease of use over functionality the product is limited on adding certain factors that other, more sophisticated, malware would have. This is not something that is easy to get around since C# is a high level language and therefore cannot exploit functionality in windows that malware written in C++ or C would be able to achieve. However, the client does not necessarily need amazingly advanced malware. Their use cases are targeted at being able to run commands on the victim’s computer and receive results. While the product will strive to remain stealthy, in the grand scheme of things the implant will only be on the targets computer for a limited amount of time, and is designed only to be an initial foothold.

The final weakness to be concerned about is that automating the testing of certain functionality may not be entirely feasible. This is a result of the work being highly specialized in nature and therefore not necessarily standardized. For instance testing if domain fronting is working would require more work than actually implementing domain fronting itself since it would have to have multiple machines to check whether or not the traffic is using a CDN to hide itself. This is not necessarily a big deal since the product is still testable, it will just require manually looking at the results.

Validation and Acceptance Test

Domain Fronting using Amazon Cloudfront
Validation: Ensure that we are able to use domain fronting using AWS to hide the endpoint of connections to our webapp.
Acceptance Tests:

- A server we set up as a proxy for our webapp cannot see target host through it's SNI.
- We are able to set values used for the HTTP host header and the domain to connect to.

Macro to deliver payload malware through automated phishing
Validation: Ensure that a macro for Word has been created that is able to deploy payload, and is able to be automated.
Acceptance Test:

- Send a document with the macro to an email that when opened and enabled content by a bot, executes the payload and connects to the C2 server.

Scheduled task persistence while remaining stealthy
Validation: Ensure that we are able to schedule a task for the malware that does not use schtasks.exe in order to avoid the high visibility that it brings.
Acceptance Test:

- A scheduled task is created by the malware without use of schtasks.exe.

Bypass Endpoint Detection and Response
Validation: Ensure that our malware is not detected by common EDR solutions
Acceptance Test:

- After implementing an EDR solution, Cuckoo, into our application, we will run it while our C2 server and bots are communicating. We will ensure that it does not detect and report the activity.

User authentication with multiple levels of access
Validation: Ensure that users are only able to access the levels of the application that they are supposed to have access to
Acceptance Tests:

- Have multiple levels of access possible, and that are assigned to each user account.
- People not logged in as a user cannot access any actions
- Regular users cannot access admin-only actions.
- Admin users can access all actions.

Sockets for bi-directional communication between bots and controller
Validation: Ensure that sockets are used for communication between the C2 server and bots running the malware.

Acceptance Test:

- When running the C2 server as it is connected to a bot, responses from the bot do not require the user to refresh the page or query the bot to look for any new responses.

Encrypted communication (HTTPS)
Validation: Ensure that HTTPS protocol is used for communication between the C2 server and the bots.
Acceptance Test:

- While running Wireshark on the C2 server, none of the packets captured show what commands are run between the C2 server and the bots.

Logging of actions taken by every user
Validation: Actions and commands taken by users on the C2 server are recorded and able to be reviewed by users with admin access to the application.
Acceptance Tests:

- Running a command using each level of user access, and ensure that they are logged and can be viewed by only a user with admin access.
- Each logged action contains information including the time sent, user's name, user's IP, bot's name, bot's IP, and message.

Admin action page for user management and logs
Validation: Ensure that there is a page that allows users with admin access to do user management tasks like adding and changing user statuses. Also allows the ability to view logs.
Acceptance Tests:

- An admin can access the page and actions while a non-admin cannot.
- An admin can create a new user, change it's access level, and remove it.
- An admin can view logs generated

Dockerize application to standardize deployment
Validation: Ensure that the application is dockerized to allow the ability to easily create a new version of the application with current requirements.
Acceptance Tests:

- A new docker image that has a different version of a single requirement is created and ran successfully.
- A docker image that has the current requirements is created and ran successfully.

Creation of help pages
Validation: Ensure that help pages are created that include common actions that a user may take
Acceptance Tests:

- A list of the major features of the application has a section or page in the help pages for each feature.

- Starting with a computer that does not include the application, a new user or existing user feigning ignorance will be able to use only the help pages to understand and run the project.

In-App Malware Builder
Validation: Ensure that an In-App Malware Builder exists that allows users to quickly test different payloads on the bots
Acceptance Test:
- A user can write malware inside of the Webapp, which is then packaged and able to be delivered.

Download and execute new payload
Validation: Ensure that a user is able to have the Implant download and execute a different payload than the original
Acceptance Test:
- Have a bot execute a payload. After execution is complete, have it download and execute a different payload

Project Timeline

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.

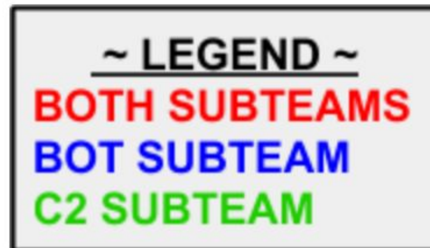

Figure 3: Gantt Chart Legend

Semester 1 will require extensive planning, meeting with the client, and setting up necessary infrastructure and tools. We plan to conduct research on the technologies we will be using for the project as well as proper procedures for successfully completing the project. Our subteams will work closely together and help each other learn the different programming languages and frameworks that we will be using to build this fully-custom application and malware bot. We will update our client throughout and implement any feedback we receive. Finally, we will write 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 1: Semester 1 Gantt Chart

Semester 2 requires much less planning and time will be primarily spent on implementing features for the project. We will kick the semester off by working as a whole to implement secure communication between the Command-and-Control Server (C2) and the bot. For this, we will

need to write code on both parts of our project and test to make sure that they mesh and the resulting communication prevents outsiders from snooping on the bot communication traffic. Next, the individual sub-teams will break off and implement their respective functionality. Near the end of the semester we will test all of the code and write 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															
Implement domain fronting															
Bypass EDR															
Expand bot persistence															
Implement bot destruction															
User authentication															
Detailed action logging															
Create malware builder (in-app)															
Enable websockets for realtime updates															
Dockerize the app															
Code testing															
Documentation															

Table 2: Semester 2 Gantt Chart

Risks / Feasibility Assessment, Cost considerations:

The main challenges our group will be facing regarding this project will be ensuring we can provide all the deliverables stated, while ensuring we have a safe and secure software for our client to properly use. There's a variety of risks that our group is aware of during this project. First off we need to ensure that we have proper security for this project. As with all projects making sure your code is safe, secure, and won't fall into the wrong hands is a basic concept. With ours specifically, due to the nature of our project being malware, we would want to especially focus on the importance of security and user authentication. Another risk would be the usability of the software and ensuring that the malware wouldn't crash the client's PC.

Although security is a big risk, we also have to consider the feasibility of the deliverable we have promised. Each statement we made about the final deliverables of the project counts as a risk if we fail to deliver or any of those deliverables aren't realistic/feasible. It's a big blunder to be naive and over-promise on what we can actually provide at the end of these two semesters. Time constraints are hard to calculate early on and as we progress through our projects and run into obstacles, it's easy to end up diminishing the final product quality as a trade-off for completing the project on time.

In terms of Feasibility Assessment we were quite thorough with understanding the risks of the deliverables we were promising. After meeting in person and discussing the deliverables thoroughly we came to the conclusion that the majority were quite feasible. Although they would take some time none were completely out of the scope of what we could accomplish. That being said there were a few that stood out as significant amounts of work. Therefore they'd be the highest risk factor in our project. The first one was getting proper results from our EDR. This consisted of getting our endpoint detection and response to detect unrecognized activity from ports on many machines. It would easily be the most challenging feature for our Implant Team to implement as well as the most time consuming. On the other hand the most difficult risk for the Webapp team would have to be the In-App Malware Builder that would allow users to quickly test different payloads. The difficulty in this would be ensuring that the web app would be able to compile malware with different variables with ease.

This project's costs can be divided into two categories, time and money respectively. First off all of these deliverables cost a certain amount of time, and the higher risk ones as outlined earlier also have the highest time costs. Progressing on these deliverables and only amounting to failure would cost heavily in terms of a failed timesink. Therefore getting a proper response from our EDR and creating an In-App Malware Builder will have the greatest time commitment and risk. Not falling into a scheduling risk on these is one the most important parts of this project. The money costs of our project is much more minimal as we have no hardware. The main costs will be AWS access which stands for Amazon Web Services. It is simply cloud hosting for our server/application, but we will not be personally paying the cost. Our client(company) will be paying for it. We will need approximately three credits which'll cost about

$20/month. $5 for the CloudFront redirect for Domain Fronting, $7.59 for the Linux t2.micro for Web Application prototype, and $5.91 for the Windows t2.nano for Malware Builder.

Standards

Over the course of the semester our team has looked at many standards and plans to implement a subset of the standards we have looked at. The first of these standards is the "International Standard for Software Engineering - Software Life Cycle Processes - Maintenance" which has an ID number of 14764-2006 - ISO/IEC/IEEE. This standard helps describe methods of effectively maintaining software [4]. We will use this as a reference at the end of the project to help plan for future software maintenance. This standard focuses on both the execution of maintenance activities and the planning of these activities. Since our team will not be the party performing maintenance activities we have focussed on the planning of these activities and on how to make a maintainable product. Our ability to implement these activities is limited however as no one on our team currently works for our client.

Another set of standards we have consulted is the IEEE Guide for Developing System Requirements Specifications which has an ID number of 1233-1998 - IEEE. This document lays out methods for determining system requirements [5]. Throughout the semester our team has talked with our client to help determine their needs and set forth expected system functionality for out project. This document has been useful in helping to set expectations.

We have also looked into IEEE 16085-2006 - ISO/IEC 16085:2006, Standard for Software Engineering - Software Life Cycle Processes - Risk Management. This standard set forth by the IEEE is used help manage risk. This standard emphasizes the importance of documenting and putting plans in place to mitigate risk [6]. In addition to this risk must be communicated to stakeholders in the project to ensure everyone has an excellent understanding of the probability of project success. We have worked towards having excellent communication with our clients throughout the project to help us manage risk, and in addition to this we have documented many of the potential risks inherent in our project.

Test Plan

Our project has many different components that all have a low level of coupling with each other. Because of this any side effects of modifications to the codebase should be relatively easy to discover. This combined with a relatively low number of branching paths in the codebase make it easy to test the code manually. The intended users of this product will also be proficient programers and will have the ability to do some of their own debugging or fall back on old methods if out project fails. Given these circumstances we believe that manually performing each of our validation and verifications tests throughout the process of development and prior to giving the solution to the client will be sufficient to test the code well.

Our testing Process will involve testing each of Project Deliverables

Implant Team

- Domain Fronting using Amazon Cloudfront
  Domain Fronting will be tested by using the Domain fronting and determining What URL it looks like the client is connecting to.
- Macro to deliver payload malware through automated phishing
  Our Macro will be tested by running it and seeing if a payload can be delivered to a target machine.
- Scheduled task persistence while remaining stealthy
  We will test if we have accomplished this by attempting to establish scheduled task persistence through our application and seeing if EDR solutions detect it.
- Bypassing endpoint detection and response solutions
  We will test our evasion of EDR by running an EDR solution on a target machine and determining if it detects our implants.

Command and Control Team

- User authentication with multiple levels of access
  We will test our authentication by observing the authentication process with wireshark.

- Sockets for bi-directional communication between implants and controller

  We will test that communication can be both sent and received from the implants and controller
- Encrypted communication (HTTPS)

  We will observe our web applications traffic in Wireshark to verify that it is indeed sending using HTTPS.
- Logging of actions taken by every user

  We will perform a list of actions as a user and ensure they are all logged.
- Admin action page for actions such as user management

  Check that the admin page can perform every action listed on it.
- Dockerize application to standardize deployment

  We will check that the application can be easily deployed using Docker on some operating system.
- Creation of help pages for common actions a user will take

  Check that an explanation of how to perform any common action has been created.
- In-App Malware tester that allows users to quickly test different payloads against EDR solutions

  Check that the malware testing solution gives the same results as the EDR solution.
- Download and execute new payload

  Check that a payload can be downloaded and executed on a victim machine.

Conclusion

The solution we propose will be an easy to use web application that allows users to easily deploy malware to a number of client machines and determine how various EDR products will react to a piece of malware. The solution will be modular allowing for the use of different malware and EDR products. Our solution must implement stringent security measures to keep the company's client's data safe including use of https and authentication. The solution will also be able to log actions performed by it's users and detect unauthorized access to malware to provide accountability for the company's client's data. This product should greatly reduce the amount of work required to perform a red team engagement.

References

[1] Mudge, R. (2018). *Beacon – An Operator's Guide*. [online] Strategic Cyber LLC. Available at: https://blog.cobaltstrike.com/2013/09/12/beacon-an-operators-guide/ [Accessed 15 Oct. 2018].

[2] Black Hills Information Security. (2018). *RITA - Black Hills Information Security*. [online] Available at: https://www.blackhillsinfosec.com/projects/rita/ [Accessed 18 Oct. 2018].

[3] xtr4nge (2018). *FruityC2*. [online] GitHub. Available at: https://github.com/xtr4nge/FruityC2 [Accessed 26 Nov. 2018].

[4] *International Standard for Software Engineering - Software Life Cycle Processes - Maintenance,* ISO/IEC/IEEE 14764:2006, 2006.

[5] *IEEE Guide for Developing System Requirements Specifications,* IEEE 1233:1998, 1998.

[6] *Standard for Software Engineering - Software Life Cycle Processes - Risk Management,* IEEE 16085:2006, 2006.