Offensive Security Orchestration

Project Management Plan

Table of contents

Problem Statement

Our project aims to develop a security orchestration platform for our client which will allow them to conduct red team engagements in a stealth 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.

Project Deliverables and Specifications:

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

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. 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 very 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. 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 deferenses. 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 very manual and can take a lot of time. This can be very expensive give 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

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

The malware has the following functional requirements:

- Communicates with C2 via a RESTful API via HTTP
- Is executable on recent versions of Microsoft Windows Operating System
- Provides system info of its infected host
- Can fetch and drop files to the host
- Can exfiltrate data from the host
- Has the ability to alter heartbeat jitter
- Is able to disconnect from a C2 and destroy itself

C2

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

- Communicates with multiple bots on the same network
- 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 and deletion
- Logs all activity by users
- Has realtime websockets for receiving data from the bots

Constraints considerations

The project and corresponding 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 get versions or trials for. Some proprietary/commercial software is hard to obtain legal licenses for, such as trials.

Technology considerations

The malware was continued 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.

Assessment of Proposed Solution:

Our proposed solution solves a range of troublesome annoyances that arise during red team engagements. We 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 our solution is not without it's strengths and weaknesses.

One of our 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 front end that will allow users to deploy and manage implants without needing to know exactly how they work. Also, since one of our main goals is to save time and money on development, and deployment, we are building the application 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 our strengths is that we leverage existing solutions where possible. When developing a product that will be used continually we want to make sure that maintaining the product is easy even without extensive knowledge regarding how it works. Thankfully when using existing solutions they typically come with great documentation that make picking up the project exponentially easier. Furthermore, by leveraging existing solutions we will be saving ourselves a large amount of time since we will not be required to recreate a product that is already available. Not to mention all the time that will be saved for our client when maintaining the product since the original developers will still be pushing out updates when new issues are discovered.

Additionally, our proposed solution will be highly modular which will allow multiple developers to be working on it simultaneously without conflict. 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. However, no solution is perfect and ours does have a few shortcomings that are important to look out for.

One of the most worrying parts of our design is how we will get access to the results of scanning the implants with different endpoint detection and response solutions. Unfortunately the developers of most business oriented solutions are closed source, and keep their secrets very close. Some of the more popular EDR solutions (such as FireEye) are so protective of their code that they won't even give out copies except to major corporations. Then, once we have the EDR solutions we have to find a way to extract the data from them and post them back to a web application. This will vary from solution to solution and on some it might not be possible to fully access the results of a scan.

Another weakness that we will face is that since we opted for ease of use over functionality we are 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 not a very low level

language and therefore we cannot exploit functionality in windows that malware written in C++ or C would be able to achieve. However, our client does not necessarily need amazingly advanced threats. Their use cases are targeted at being able to run commands on the victim's computer and receive results. While we 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 that we are worried 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 we would have to have multiple machines to check whether or not our traffic is using a CDN to hide itself. This is not necessarily a big deal since our product is still testable, we will just have to manually look at the results.

Validation and Acceptance Test:

Domain Fronting using Amazon Cloudfront

- Proxy cannot see target host through SNI.

Macro to deliver payload malware through automated phishing

- Able to be sent through email, and activates payload when allowed access by user.

Scheduled task persistence via scheduled tasks while avoiding the use of schtasks.exe

- A scheduled task is created without schtasks.exe and leaves few if any obvious artifacts.

Endpoint Detection and Response, to detect unrecognized activity from ports on many machines

- When a collection of machines is observed, and one starts communicating with malware that we give it, the EDR will detect the activity and report it.

User authentication with multiple levels of access

- Admin and regular accounts can be created.
- Regular users cannot access admin-only actions.
- Admin users can access all actions.

Sockets for bi-directional communication between bots and controller

- A socket is established between the controller and each bot, which allows the bot's responses to be shown in real-time without the need to refresh the page.

Encrypted communication (HTTPS)

- A network sniffer watching the packets will not be able to see what commands and responses are being sent between the controller and the bots.

Logging of actions taken by every user

- Every command sent from the controller is logged into a file that is visible by admins.
- Each logged action contains time sent, user's name, user's IP, bot's name, bot's IP, and message.

Admin action page for actions such as user management

- An admin can access the page and actions while a non-admin cannot.
- Administrative actions, such as user creation, deletion, and escalation of privileges are available.

Dockerize application to standardize deployment

- A docker image with a different version of a requirement can be created and ran. The reason for one with different requirements is to show that after our dockerization, we have the ability to update the requirements and create an image with those updates completed.
- A docker image with the current requirements can be created.

Creation of help pages for common actions a user will take

- Every major feature or action will have a section or page in the help pages.
- A new user will be able to use the help pages to understand and run the project.

In-App Malware Builder that allows users to quickly test different payloads

- A user can write malware inside of the Webapp, which is then packaged and able to be delivered.

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.

Fig 1: Gantt Chart for Semester 1

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.

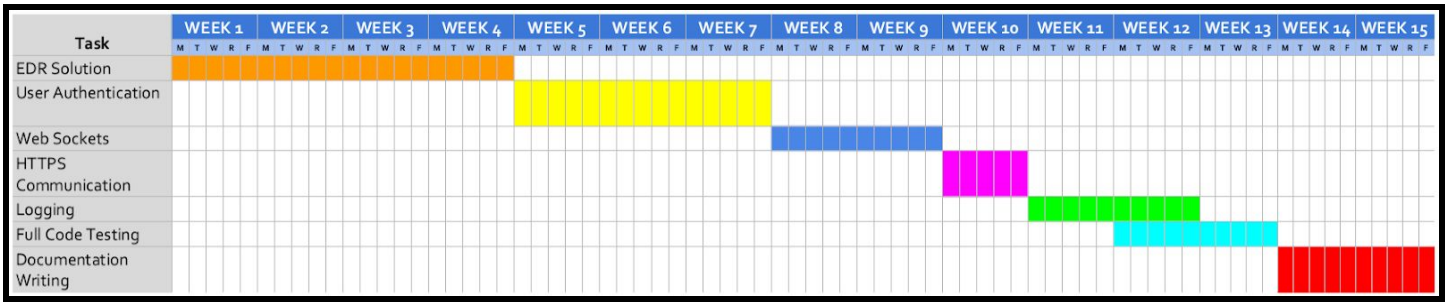

Fig 2: Gantt Chart for Semester 2

Semester 2 requires much less planning and time will be primarily spent on implementing features for the project. Near the end of the semester we will test all of the code and write documentation for the client.

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. 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 around $30 monthly for 8 VMs. We ourselves will not incur any personal costs regarding the project.

Costs: Time and AWS access (amazon web services basically cloud hosting) but company will pay for it
One for C2 server, one for Cloudfront CDN instance. Probably 3 credits $30 monthly for 8 VMs

Risk will be malware crashing clients
Not having proper security for a malware
Not meeting deliverables on time
Over-promising on deliverables

Hardest feasibility
Getting results from EDR (endpoint detection and response)
In-App malware builder, using web app to compile malware with variables

Standards:

One of the main concerns for this project is setting up standards to prevent us from leaking our code to third parties. There are a few reasons why we wouldn't want to leak code. If any malware we write ends up being leaked it may be signitured by antivirus or used for purposes which we do not condone. In addition the company will lose any competitive edge this code gives them over their competitors in the marketspace if their competitors also have the code. To ensure that we do not leak code we will be using only our private git repository and other encrypted channels of communication to share the code.

This project will be used as a tool to reveal to organizations' weaknesses in their cybersecurity. One of the tenants of the IEEE code of ethics is "to hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, and to disclose promptly factors that might endanger the public or the environment;" Our product will directly support the health, safety, and welfare of the organization by facilitating processes which help to reveal an it's security weaknesses and provide advice on how to become better and more secure.

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.

Conclusions:

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.