Seanonymous: Malicious Web Browser Extension

Team Members: Andrew Jaffie, Rebecca Hassett, Kavan Wadhwa, Ian Buitenkant

Github Repository:

https://github.com/ajaffie/seanonymous

Project Development:

1. Decided that the malicious web browser extension would appear to be a Google Chrome Ad Blocker Extension.
2. Designed database schema for storing client information and blacklisted websites blocked by the extension. Decided to use MariaDB.
3. Split up responsibilities amongst group members, and set up Github repository.
4. Determined frameworks to use to develop web application and to communicate between the server and the extension. Developed the web application using Python Flask. Communication between the server and the extension as well as between the server and the web application used the Flask-SocketIO library.
5. Setup Nginx web server and MariaDB database on Ubuntu 18.10 VM hosted by DigitalOcean.

Server Setup:

The backend is running on an Ubuntu 18.10 VM. While not required technically, I highly recommend you use the included firewall ‘ufw’ to protect the server, allowing ports 443, 80, and 22 (for ssh access). The main dependencies are NGINX, Python 3, Pip3 (python 3 package manager), virtualenv, and MariaDB. The latest versions from the Ubuntu apt repositories are

sufficient, and those in use by our VM. Create a sudo user named seanonymous for seamless use of our files. The MariaDB database should be initialized with a database named cse331, and the root password should be seanonymous (unless you want to change the code). All work should be done from the root directory of the repo.

Run `mysql -u root -pseanonymous cse331 < backend/schema.sql` to create the database. Then, run the script called 'init.sh.' This will create the virtual environment and install all the python dependencies. Copy 'external-files/backend.nginx.conf" to '/etc/nginx/sites-available/backend.conf" and modify to either use the appropriate files (certificate, private key and dhparams) for TLS or disable TLS. Create a symbolic link for this config file in '/etc/nginx/sites-enabled/' to enable it. Do `systemctl enable nginx; systemctl start nginx` to start the reverse proxy. Copy 'external-files/seanonymous-backend.service' into '/etc/systemd/system/' and run `systemctl daemon-reload` to prepare the service. At this point, doing `systemctl start seanonymous-backend` should start the backend service running. Alternatively, and for simpler debugging, just run runserver.sh from the root directory of the repo. A Flask secret and attacker UI credentials can be specified in single-line plaintext files named '.secret' '.user' and '.pass' owned or at least readable by the seanonymous user. You would have to change the url of the server in any part of the code in order to host this yourself, most notably in the extension code and nginx config.

Database Schema:

The database schema is located in the schema.sql file in the backend folder. The database includes a Client entity that stores information specific for a client that has installed the extension. The Client entity stores a client's cell phone number, street address, email address,

social security number, first name, last name, date of birth, city, country, zip code, and state. Clients are uniquely identified using an ID value that is auto-incremented in the database when inserting a new user.

The Credentials entity stores credentials for a specific client for a specific url. The Credentials entity stores the Username, UserPassword, URL, type of Multifactor Authentication for a specific client ID. The client ID, Username, and URL are the primary key because these values uniquely identify a credential. The client ID is a foreign key that references the ID in the Client entity.

The CreditCard entity stores credit card information for a specific client. The CreditCard entity can store a CreditCardNumber, CVC, Expiration Date, and credit card type for a specific client id. The Type in Credit Card is an enum value which can be either ‘American Express’, ‘Mastercard’, ‘Discover Card’, or ‘VISA’. The primary key of CreditCard is CreditCardNumber since a client can have multiple credit cards uniquely identified by their credit card number. The CID in CredtiCard is a foreign key that references the ID in Client.

The Cookies entity stores client cookies. The Cookies entity stores the URL, Content, and Name of the cookie for a specific client id. CID, URL, and Name is the primary key that uniquely identifies Cookies. This is because a website can have multiple cookies with different names. The CID in Cookies is a foreign key that references the ID in Client.

The Security Questions entity stores security questions and answer pairs for signing up for new accounts. The Security Questions entity stores a Question, Answer, URL for a specific client id. CID, Question, and URL are primary keys because a URL can have multiple security questions. CID in Security Questions is a foreign key that references ID in Client.

The BlacklistedWebsites entity stores the URL of the security websites that the clients are redirected away from as well as the RedirectURL which contains the URL users are redirected to. The URL of the BlacklistedWebsites entity is the primary key.

The PendingPayloads entity stores the Javascript attack commands and Phishing commands that the attacker wants to send to a specific client once they are online. The ClientID is the primary key that uniquely identifies the payloads that will be sent to the client. The PendingPayloads entity also stores the Payload as a TEXT type because the Javascript and Phishing commands are stored as a JSON dictionary with a pattern field for the url that the command is sent to and a command field for the command being sent to the client.

The FormIDMappings is an entity that maps database schema attributes to the remote definition of an element on a specific url form. The FormIDMappings entity stores the URL of the website where the mapping is relevant, the LocalDef value which is the database schema attribute being mapped to, and the RemoteDef value which is the remote definition of the element containing the stolen information in the url form. The LocalDef value must be an enum that matches the attribute of an entity in the database. For example, LocalDef values include ‘Username’, ‘UserPassword’, ‘CreditCardNumber’, ‘Email’, ‘FirstName’, ‘Question’ as well as other database attributes. The purpose of the FormIDMappings is to dynamically process information on a website entered by a client, and to store this information into the database. Once a FormIDMapping has been made for a URL, it can be used for many different clients at the same URL to process their information. The FormIDMappings have default mappings set by the attacker mapping common remote definitions to the database local definitions. The attacker web application also has the ability to obtain client information linked to unknown remote definitions,

to store this client information into the correct location in the database manually and to manually add a formIDMapping between the unknown RemoteDef and one of the LocalDef enum values to aid in further processing of data on that website. The unknown RemoteDef values are gathered by the attacker web application using the ComplexForms table.

The ComplexForms entity stores the key-value pairs sent in the JSON file from the extension to the server that could not be mapped to a local database schema definition using the FormIDMappings table. The key values stored in the ComplexForms entity can be mapped to the local database schema definitions manually by the attacker web application calling server side functions to add these mappings into FormIDMappings. The ComplexForms entity stores the URL of the website from which this information was retreived, the clientid for which this information was obtained, and a JSONFORM attribute stored as a string which contains all of the key-value pairs that were not stored in the database because no mappings existed for that key.

Storing Client History:

The seanonymous directory of the project contains a history-files directory where every client has their history stored. Each client's history is stored in a separate file titled "[clientid]-hist.txt" where the [clientid] is replaced by the id of the client whose history is stored in that file. The history file stores the urls of websites that the clients have visited along with timestamps of when these urls were stored by the malicious web browser extension.

Phishing Commands and Javascript Execution Commands:

The malicious web browser extension program contains two phishing commands that have already been set by the attacker and are reusable. The phishing commands are stored in the phish-cmd directory in the seanonymous directory. This directory contains the phish-1.js

Javascript file containing the first phishing command and the phish-2.js Javascript file containing the second phishing command. When the attacker selects the “phish1” button in the web application for a specific user, the javascript command from the phish-1.js file is either stored in the PendingPayloads table in the database for that specific client id if the client is offline or it is stored in the pending_payloads dictionary to be loaded, sent to, and executed by the extension if the client is online. The same functionality is implemented for the “phish2” button in the web application. The attacker web application also has the option for the attacker to dynamically enter Javascript executable code that will be sent to the extension to be executed. This is performed in the same way that the phishing commands are performed using the PendingPayloads database entity to store the Javascript executable if the user is offline or the pending_payloads dictionary to store the Javascript to be sent to and executed by the client if the user is online.

The phishing scripts themselves share a good portion of code. This code searches for the main content of the page and replaces it with our malicious content. First, it tries to use the ARIA role ‘main’ or an empty role. Then, it looks for an id of ‘content’ or ‘main.’ The last attempt searches for a div with class ‘container’ that is not contained within a header tag. This seems to be successful on several popular sites, but as a last resort the script simply replaces the entire body tag. Then, all h1’s and sidebar-like elements are removed and the title is changed appropriately. Links are also disabled as a small effort to increase the likelihood of users falling for the phishing attempt. The idea behind all of this is to try and remove the main content of the page so that it seems like the malicious content is supposed to be there. The HTML to be inserted is stored in the script in a template literal. This is a relatively new feature of

ECMAScript, but it is implemented in Chrome. Phish-1 is a traditional phishing attack where the user is tricked into providing personal information. We basically try to sell them on, ironically, an identity theft monitoring service for 'free.' Phish-2 is also based in irony in that it tries to trick the user into downloading 'free anti-virus software.' For everyone's safety in this project, the file is not actually a virus and we encourage you to run it for a good laugh.

Web Sockets:

The Flask-SocketIO websockets library was used to communicate between the server and the extension as well as communicate between the server and the Flask web application. The extension loads the Socket.IO library and establishes a connection to the server using "socket = io.connect('https://cse331.andrewjaffie.me/socket.io')". When a new client installs the extension or an existing client opens their web browser, a connection event handler on the server side specified by "@socket.io('connect', namespace='/socket.io')" is called. A connection event handler on the client side is called as well. When a client exits their web browser, a disconnection event handler specified "@socket.io('disconnect', namespace="/socket.io")" is called on the server side. This function removes the client id and session id from the list of connected clients and it stores pending payloads that have not been executed yet for that client in the database. There is also an event handler that is called by the attacker web application to store new FormIDMappings.

The server also contains an event handler that is called by the extension to send payloads to the server. This function obtains the client id from the payload, checks that the payload is valid, and handles new client installation or existing client responses. If the client id in the payload is zero, this means the client is new. The client must be added to the list of connected

clients, and the client id must be emitted by the server to the web application to add a new row for that client in the web application. If the client id is not zero, this means the client is an existing client. The client's id is stored in the list of connected clients, the client's information is stored appropriately in the database, and a response is sent to the extension containing the pending payloads for the client and the blacklisted websites.

Web Application:

The attacker UI was implemented using Python Flask, Jinja2, Bootstrap, HTML, CSS, Javascript, and JQuery. Flask integrates Jinja2 templating very well and comes with many useful global variables, for example, .url_for(), which was used to redirect the attacker between the different pages and templates associated with the frontend. The login screen consists of a form asking the user to enter his/her credentials to access the UI. If the credentials do not match those specified by the single plain text files, .user and .pass, the user is prompted again. All endpoints are secure, meaning that trying to access ../attackmode, which is the main page for the attackerUI, will redirect to the login screen. This was implemented using the Flask-Login module. Typically, Flask-Login is used for user management. Although only one set of credentials is needed to access the attacker UI (in other words, only one user), Flask-Login was still very helpful in setting up a secure login system without much difficulty. The module comes with an @login_required decorator ensuring that functions cannot be executed/accessed unless a user is logged in.

Once logged in, the attacker is presented with his/her dashboard. There is a welcome message towards the top of the screen ("Welcome Seanonymous. Attack at your will!") with a

logout button beneath this greeting. The logout button marks the user as logged out (thanks to Flask-Login, this is one line of code) and redirects to the login screen.

Below the login button is a button that toggles information about blacklisted websites. This includes the current blacklist as well as a form for adding a new domain to the blacklist. As such, the blacklist can be updated dynamically.

The entire attacker dashboard is a singular table, constructed using Bootstrap and its associated jQuery/JavaScript plug-in, DataTables. Bootstrap comes with many base styles for different elements making styling more simplistic to implement. Additionally, Bootstrap has a responsive grid system, making it easy to position elements; this was used particularly for the form that appears under the information for a user when "view info" is clicked. DataTables comes with ordering and filtering entries pre-built. It also displays the number of entries at the bottom and allows the user to select the maximum number of entries per page, thereby allowing the table to be spread out across pages.

Each row of the table contains the user's clientid (if available, the name of the user is displayed instead), the user's current status (online/offline) with a green/red background respectively, an input field for URL, an input field for JS commands, a button group of 2 for phishing commands, and a view info button. The clientid/name acts as a button; when clicked it opens a new tab with all the user's information (general, credentials, credit cards, forms, etc.) which is queried from the seanonymous database. The queried information is then rendered using Jinja2 and html (result.html file). The JS command input box has a button attached to it. When clicked, it grabs the URL input information, the JS command input, and the clientid and sends this information to the database to be processed and sent to the extension. Like JS commands,

phishing commands also send the URL input and clientid to the database; however, instead of a script, the numbers 1 or 2 are sent specifying which phishing file to obtain the command from.

The information for the commands are sent as a POST request to the corresponding app.route (/sendjs or /phish) via ajax. In these routes, the information is requested, stored, and sent to the appropriate method in database.py for processing.

Clicking on the view info button in the last column of a row results in that row collapsing downward. A new row is opened with all the information collected for that particular user. This was implemented using Bootstrap's collapse plugin, used for toggling visibility of elements. It involves adding collapse to the classList of an element and referencing that element by id in the button being used to toggle it. To integrate the idea of a row expansion with the DataTables plugin, I made use of child rows, for which DataTables has a class (child rows attach to parent rows). The expanded row presents the data very simply, with a header for each section. Not all browsing history is shown in this expansion, but there is a view all button next to the browsing history tab that opens the full browsing history in another tab.

For forms, there are two buttons under every form header (each form header consists of "Form #" and the URL that the form is from). The open button is self explanatory; the edit button makes visible "select" elements, which are html dropdown elements. Each "select" element has all possible LocalDef enum values. The attacker can select per their desires and submit the form to add new mappings. New mappings are only sent for those field which the attacker has selected a value (there is a blank option in the "select" element).

The DataTable defaults were overridden so that the preset sorting was descending in the second column (status). This means all online users appear first. The DataTable is also orderable on the ID/name field. The search bar filters the data based on the ID/name field.

Making use of Flask-SocketIO and JavaScript's SocketIO, there are real time updates made to the table without refreshing/reloading. This includes when a new client installs the extension as well as when users connect and disconnect ('online'/'offline'). When the extension notifies the server that a client has connected/disconnected, the clientid is emitted and using "socket.on" the emitted event, the status field and color are changed for that particular user. The new client install emit was described previously.

Extension:

"SeAnonymous uses revolutionary 'SmartBlock' technology, that analyzes sites that you commonly visit to better detect and remove advertisements that you would otherwise be exposed to. Backed up with an up-to-date database of known ad services, SeAnonymous gives you the best of the internet without the ads. We also analyze HTTP cookies to thwart persistent advertisers who wish to track you over the web and serve targeted ads."

- Chrome Store Overview for SeAnonymous

Since the extension disguises its behavior behind an ad blocker, the user should be convinced that the extension actually does its job, otherwise they will just remove it. We used an online tutorial for setting up a simple ad blocker that uses a blacklist of ad servers and webRequest blocking. The tutorial was created by Adrian Stoll, and can be found here (https://adrianstoll.com/dyi-adblocker/). In this section, we will explain how the extension fulfills each of the project specifications in such a way that the user does not become suspicious of their ad blocker. In general, the extension communicates with the server using a SocketIO object, and

sends payloads of information at regular intervals. This reduces web traffic that may affect the user's bandwidth if it is sent every time new information is found.

1. Leaking Browser History

   Our extension keeps a queue of information to be sent to the remote server at regular intervals. Just before a payload of information is sent to the server, the extension pulls a large amount of the user's history and adds it to the payload. To prevent duplicate history objects for the same site, we keep track of the time of last payload and filter out objects with a "lastVisitTime" that is less than this value.

2. Steal Username and Password

   We set up a listener for the webRequest event, and whenever the event is fired, we check to see if it is an HTTP POST request. If so, we get the "formData" object (in the form of a JSON) and iterate over all of its keys, checking to see if they are recognized as a username or a password. All other elements are stored in a "complex form". If the form elements are named in a way that we did not consider, the attacker has the option of mapping these names to known credential types. For example, if the form has an element called "pwd", the attacker will see this value in the a complex form, and can choose to map all future uses of "pwd" to "password". We decided to take all form fields, as these may be useful for further work with recognizing other sensitive information sent by the user.

3. Steal Cookies from HTTP Request

   We set up another listener for the webRequest event. This one simply takes all cookies associated with the url for this HTTP request and adds them to a payload. When

the cookies reach the database, if one already exists with the same “domain” and “name”, the one in the database will be overwritten. This allows us to update cookies whose state have changed since the last payload, and also ignore duplicate cookies. This required the use of the “cookies” permission, which we explain in our advertising slogan as necessary because “Seanonymous analyzes HTTP cookies to thwart persistent advertisers who wish to track you over the web and serve targeted ads”.

4. Stop the User from Visiting Security Websites

    Rather than hard coding a list of sites and corresponding locations, we decided to allow the attacker to determine these mappings for himself. The attacker can add new mappings through the UI, and the new map is sent to every victim whenever one communicates with the attacker. This is not ideal in case where the attacker has created a large mapping, but we could not find a good way of removing mappings or updating existing mappings once the victim already has these redirection rules in place.

5. Preset Phishing and Arbitrary JS Execution

    Preset Phishing and Arbitrary JavaScript Execution happen through the same mechanism: a list of pairs of {website, JS command} that can be added to by the attacker through payloads. We set up a chrome.tabs.onUpdated listener that fires whenever the current tab is updated (via navigation, clicked link, etc). This was necessary because a webRequest listener would not offer us access to the DOM of the current tab. With this access, we are able to run the chrome.tabs.executeScript function in the current tab, allowing for the execution of any JavaScript code that is passed into the function. For

more information about the preset commands see page 5, Phishing Commands and Javascript Execution Commands.

6. Miscellaneous functionality

The “browser_action” is to display a popup that allows the user to “disable” the ad blocker in case they want to view something that was unintentionally blocked. This is also detailed in the tutorial by Stoll, but we chose to leave it in our extension as a means of promoting the legitimacy of our extension in the eyes of the victim. If the victim thinks that he has a choice in how the extension operates, he will be more likely to trust the extension. It also provides a way for the victim to view blocked content besides uninstalling the extension.

Primary Responsibilities:

- Ian Buitenkant: Extension
- Kavan Wadhwa: Attacker Web Application
- Andrew Jaffie: Back End Web Sockets/Server Setup/Phishing Commands
- Rebecca Hassett: Back End Database Transactions

References:

- MariaDB Documentation:
    - https://mariadb.com/kb/en/library/documentation/

- Weekend Project (Part 2): Turning Flask into a real-time websocket server using Flask-SocketIO:
    - https://secdevops.ai/weekend-project-part-2-turning-flask-into-a-real-time-websocket-server-using-flask-socketio-ab6b45f1d896
- Flask-SocketIO Documentation:
    - https://flask-socketio.readthedocs.io/en/latest/
- Flask Documentation:
    - http://flask.pocoo.org/docs/1.0/
- Easy Web Sockets with Flask and Gevent: Posted by Miguel Grinberg
    - Reference used for learning about Flask-SocketIO library
    - https://blog.miguelgrinberg.com/post/easy-websockets-with-flask-and-gevent?fbclid=IwAR1jKoakcMKgxy32CQQVk-5gI5ZhtVaCqHvJXfcHXI4emGrzIXM9hcM7yh8
- The Flask Mega-Tutorial: Posted by Miguel Grinberg
    - Reference used for learning about Flask framework
    - https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world?fbclid=IwAR1scoztf1iVDQS_PlwsiJupOMJgTsHewounhkuj4-ckH0PVqjBnHnio8bo
- StackOverflow

    - Reference used for many, many questions about JavaScript, Bootstrap, JQuery, etc.
    - https://www.stackoverflow.com
- Chrome Web Extensions API Reference
    - Used for referencing the Chrome Web Extensions API
    - https://developer.chrome.com/extensions/api_index
- Bootstrap Documentation:
    - https://getbootstrap.com/docs/4.1/getting-started/introduction/
- Shane Lynn's Asynchronous updates to a webpage with Flask and Socket.io:
    - Used for implementing real time table updates for changes in client connectivity and new client installations without refreshing webpage
    - https://www.shanelynn.ie/asynchronous-updates-to-a-webpage-with-flask-and-socket-io/
- Flask-Login Documentation:
    - https://flask-login.readthedocs.io/en/latest/
- Flask Jinja2 Documentation:
    - Referenced for integrating Jinja2 with flask
    - http://flask.pocoo.org/docs/1.0/templating/
- StackOverflow Code Snippet:

    - Used for centering “add blacklist” form
    - https://stackoverflow.com/questions/44389464/align-the-form-to-the-center-in-bootstrap-4
- DataTable Documentation:
    - Referenced for implementing attacker UI dashboard table and child rows for each user.
    - https://datatables.net/
- W3Schools HTML Element Reference:
    - Used for styling, tag, javascript, etc. questions
    - https://www.w3schools.com/tags/
- jQuery.post() API Documentation:
    - Referenced for sending information and instructions from script in html files to frontend.py, where information retrieval/storage occurred
    - https://api.jquery.com/jquery.post/
- DIY Adblocker Tutorial:
    - Used as a starting point for building our extension and learning about basic JavaScript
    - https://adrianstoll.com/dyi-adblocker/
- Chrome Developer Documentation:

    - Referenced frequently to access Chrome APIs such as History, Cookies, WebRequests, and Storage
    - https://developer.chrome.com/extensions/
- Collecting Form Data with a Chrome Extension:
    - Referenced to get an idea of how to pull form data from POSTs. Mostly just used the concept of listening for all requests, and looking closely at those with method = POST
    - https://spin.atomicobject.com/2017/08/18/chrome-extension-form-data/