COS 333 Internals: Parsnip

Shuyang Li, Timothy Seah, James Wang, David Zhao

Our app is a Google Chrome extension packaged as a .crx file, just like any other extension you can download from the Chrome store. The source files are written in a combination of HTML, CSS, and JavaScript. Within the .crx file we have the following source files:

Client Side

manifest.json (required for chrome extensions): a manifest file that holds the app together. It specifies what permissions the app has (for example, our app can access information from the user's open tabs), what files are linked to the extension, etc.

index.html: the user interface for our app is stored in index.html. index.html is hosted on the website (www.princeton.edu/~shuyangl). When the user clicks our app's icon, our app opens a new window with www.princeton.edu/~shuyangl as the URL.

mystyle.css: defines the style for index.html

macrodef.js: macrodef.js defines logic for index.html. When the user clicks on any of the buttons specified in index.html, macrodef.js sends a message to background.js for further processing.

webpageinject.js: when the user presses the "record" button, a message is sent to background.js telling it to inject webpageinject.js into all active tabs. Webpageinject.js attaches event listeners to all of the DOM elements on the page. When triggered, the event listener sends a message to background.js containing information about the action. Our app uses this information to play the macro later on.

The sent information contains two things: the JavaScript code to perform the action and the URL associated with the action. The bulk of the JavaScript code attempts to identify the element to perform the action on. We cannot simply address an element by its index because an element's position on a webpage may change from day to day. Our app identifies elements by matching on important attributes such as tag, id, class, href, and src.

background.js: background.js runs as long as our extension is active. Background.js is the "glue" of our app. It performs the following functions:

a. When the user clicks on our app's icon, background.js receives a message and opens a window containing the GUI for our app. When the user closes our app's window, background.js terminates all of our app's processes (except for a process that listens for the user to click on our app's icon).
b. Recording: when the user presses the "record" button, macrodef.js sends a message to background.js telling background.js to start recording. Background.js then injects webpageinject.js into all of the user's currently open pages and any other pages that the user might open. When the user presses "record" a second time, background.js stops injecting webpageinject.js into all new pages.
c. Storage: when the user presses the "save" or "save with condition" buttons, macrodef.js sends a message to background.js. background.js stores the macro using the chrome.storage API, which stores the information on the user's computer as a JSON object in Chrome local storage. If the user wants to remove their saved macros, they can click "delete" or "wipe memory," which use the chrome.storage API to delete the macro whose name is specified in the "Macro Name" field or delete all the macros in local storage. When the user presses "load," macrodef.js sends a message to background.js. background.js then loads the macro from memory into a global variable and displays it in the UI.
d. Clearing forms: if the user clicks "clear form," macrodef.js sends a message to background.js telling it to delete the current macro and clear it from the user interface. This includes both the macro and the associated conditions. If the user clicks "clear conditions," macrodef.js sends a message to background.js telling it to delete the conditions and remove them from the user interface. If the user hits "play" or "play with repeat" when there is no macro in the user interface, nothing will happen.
e. Playing macros: if the user clicks "play" or "play with repeat," macrodef.js sends a message to background.js telling it to play the macro. Depending on which button is pressed (play vs play with repeat) and whether or not a condition was saved,

background.js decides to either play the macro immediately, play it when the time condition is met, or play it at every interval. Our app plays a macro using the following basic logic:

i. Open a window containing the URL associated with the first action.
ii. Add a listener for when the URL changes. When the URL changes, play all the actions associated with that URL.
iii. Refresh the page, triggering the listener for the first time.
iv. Our app plays all the actions associated with a URL until it changes, after which it plays all the actions associated with the new URL. It continues to do this until there are no more actions to perform, after which our app removes listeners and terminates execution.

Server Side

The server side of Parsnip is hosted on an Amazon Web Services Elastic Compute Cloud instance. The server used SQLite3 as its database system because of its very quick deployability. SQLite3 was able to be hosted as a part of the EC2 layer, rather than occupying its own server.

sniff.py was the main program that implemented packet sniffing and macro storage capabilities. Packet sniffing was enabled through the Scapy python library. sniff.py recieved macro information through XMLHtttpRequests called by the Chrome extension and then parsed the http and TCP information though regular expression searches. Then, sniff.py inserted whole macros as strings into the SQLite3 database for future use, accompanied by IP Address and macro name information.

execute.py was the more experimental side of the backend. It utilized the Python libraries of BeautifulSoup, Splinter, and the headless web browser PhantomJS, in order to attempt preliminary implementation of macro execution. Unfortunately, it wasn't as simple as executing a single macro script, because the auto-generated macro scripts required confirmation of web page state before each segment of JavaScript could be executed. For the most part, execute.py was a preliminary exploration in backend execution functionality.