Malicious Website Detection

January 21, 2023

1 Background

This research proposed a malicious behavior suspected website detection system with static analysis and dynamic analysis based on JavaScript, Jquery, and HTML that analyzes code density, the kind of method that code used, entire JavaScript entropy, and entropy of each characteristic.

A function named myWebguard is defined. which has a constant object named 'builtins', this builtin object has property value pairs like URL:URL, parseJson:JSON.parse, apply:Function.prototype.apply, etc.

```
function myWebGuard() {
    const builtins = {
        URL: URL,
        Date: Date,
        Error: Error,
        Promise: Promise,
        setTimeout: setTimeout,
        parseJson: JSON.parse,
        stringify: JSON.stringify,
        window: window,
        sessionStorage: window.sessionStorage,
        apply: Function.prototype.apply,
        defineProperty: Object.defineProperty,
        hasOwnProperty: Object.prototype.hasOwnProperty,
        getOwnPropertyDescriptor: Object.getOwnPropertyDescriptor,
    }
```

An object named Utils is created with different functions within it, it works like a key pair value, so any function with the utils can be called to run or it can be executed with the execution of utils object. There are 6 functions respectively.

1) The function() prints Mywebguard along with the argument that it takes. PrintVerbose is the key.

2)The function() with getTopOrigin as the key Returns a reference to the topmost window in the window hierarchy.it works within the try catch functionality to get the url of the current window.

```
const utils = {
printVerbose: function () {
  console.log('[MyWebGuard]', ...arguments)
},
getTopOrigin: function () {
  let url
  try {
    url = builtins.window.top.origin
  } catch {
    url = builtins.window.origin
  }
  return this.getOrigin(url)
},
getOrigin: function (url) {
  try {
    return new builtins.URL(url).hostname
  } catch {
    return null
  }
},
isCrossOrigin: function (origin) {
  try {
    return this.getTopOrigin() !== origin
  } catch {
    return false
  }
},
isUrlCrossOrigin: function (url) {
  try {
    return this.getTopOrigin() !== this.getOrigin(url)
  } catch {
    return false
  }
},
sleep: function (ms) {
  const start = new builtins.Date()
  let current = null
  do {
        current = new builtins.Date()
      }
      while (current - start < ms)
    },
  }
```

2 JavaScript Objects and Properties

The named values, in JavaScript objects, are called properties.
Objects written as name value pairs are similar to:

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

source: www.w3schools.com/js/js$_object_definition.asp$

JavaScript objects are dynamic ”bags” of properties (referred to as own properties). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

Methods are actions that can be performed on objects. Object properties can be both primitive values, other objects, and functions. An object method is an object property containing a function definition.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
FullName	function() return this.firstName + ” ” + this.lastName;

JavaScript objects are containers for named values, called properties and methods.

A JavaScript object is a collection of unordered properties.

Properties can usually be changed, added, and deleted, but some are read only.

JavaScript objects inherit the properties of their prototype.

In JavaScript, the this keyword refers to an object.

Any JavaScript object can be converted to an array using:

Object.values()

Object. getOwnPropertyDescriptor

Object. defineProperty

```
old_alert= window.alert;
window.alert=
function(){
        console.log(" alert is being monitored");
       const obj = this;
       const args = arguments;
       old_alert.apply(obj, args);
       CountOfApis['alert']+=1;
  }
```

The key difference between properties of the object being used/created as well as creating new instance of a given function so that it can be monitored and policies/other function calls can be embedded in them.Using this method we can look at the flow(steps) and also enforce other calls or statements(if,else conditions).We can also find out how many times the function was called.

Background The Aim of this thesis is to create a Dynamic Malware detection model, which is a updated version of the already working Static Malware Detection Model (Webguard browser add-on). A machine learning model will be deployed to achieve the desired results.

Approach/Plan Collect data/information from the browser by using functions calls. This collected information will then be passed to machine learning model which in turn will be able to help in applying/enforcing policies, preventing/avoiding malicious website, links etc.

Outcomes/results Using the above mentioned approch we should be able to monitor functions and their properties that are being used through out the website and collect information from them, this in turn helps to train the machine learning model that is being implemented.

Concepts/Topics learned So far i was able to understand the key difference between properties of the object being used/created as well as creating new instance of a given function so that it can be monitored and policies/other function calls can be embedded in them. This concept of intercepting the function call and applying/enforcing policies or other function calls helps to give users more control of their data as well as blocking unwanted re-direction which prevents drive by downloads and other unwanted installations.

Related work A NOVEL APPROACH FOR ANALYZING AND CLASSIFYING MALICIOUS WEB PAGES link: https://etd.ohiolink.edu/apexprod/rws$_etd/send_file/send$?
accession=dayton1620393519333858disposition=inline

API flow Documentation Browser$_action$:

A browser action is a button that your extension adds to the browser's toolbar. If you supply a popup, then the popup is opened when the user clicks the button, and your JavaScript running in the popup can handle the user's interaction with it. If you don't supply a popup, then a click event is dispatched to your extension's background scripts when the user clicks the button.

Background:

Background scripts or a background page enable you to monitor and react to events in the browser, such as navigating to a new page, removing a bookmark, or closing a tab.

Persistent – loaded when the extension starts and unloaded when the extension is disabled or uninstalled. Background scripts run in the context of a special page called a background page. This gives them a window global, along with all the standard DOM APIs provided by that object.

Web Extension APIs- Background scripts can use any Web Extension APIs, as long as their extension has the necessary permissions.

Cross-origin access- Background scripts can make XHR (XMLHttpRequest (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh.) requests to hosts they have host permissions for.

Web content- Background scripts do not get direct access to web pages. However, they can load content scripts into web pages and communicate with these content scripts using a message-passing API.

Content security policy- Background scripts are restricted from certain potentially dangerous operations, such as the use of eval(), through a Content Security Policy.

```
const fs = require('fs');
const path = require('path');
const puppeteer = require('puppeteer');

const runScriptOnWebsites = async websites => {
  const pathToExtension = path.join(process.cwd(), 'extension');
  const browser = await puppeteer.launch({
    headless: false,
    dumpio: true,
    args: [
      `--disable-extensions-except=${pathToExtension}`,
      `--load-extension=${pathToExtension}`,
    ],
  });

  // Create a CSV with a header row
  let csv = 'URL,API,Count\n';

  for (const website of websites) {
    const page = await browser.newPage();
    page.on('console', msg => console.log(msg.text()));///continious flow of data
    await page.goto(website, { waitUntil: 'networkidle0' });
    //await page.waitForNavigation({waitUntil: 'networkidle2' });
    const entries = await page.evaluate(() => window.entries);
    console.log('PAGE LOG:', entries);


    for (let i = 0; i < entries.length; i++) {
      const api = entries[i][0];
      const count = entries[i][1];

      csv += `${website},${api},${count}\n`;
    }
  }
  console.log(csv);
  fs.writeFileSync('entries_log.csv', csv);
};

const websites = ['https://www.yahoo.com', 'https://www.bing.com', 'https://www.msn.com'
(async () => { await runScriptOnWebsites(websites) })();
```

The above code is a JavaScript script that utilizes the Node.js run time environment and several modules, including fs, path, and puppeteer, to scrape data from a list of websites and write the data to a CSV file. The script begins by importing the required modules at the top of the file. fs is the built-in file system module, path is the built-in path module, and puppeteer is an external library for controlling a headless Chrome browser. The script then defines an async function called runScriptOnWebsites, which takes an array of websites as its argument. The function uses puppeteer to launch a headless Chrome browser, with the dumpio option set to true and the args option set to load an extension from a directory called extension. The script then creates an empty string variable called csv which later will be filled with the data from each website.

The script then enters a for loop, where it iterates through the array of websites passed as an argument to the function. For each website, the script creates a new page in the browser, navigates to the website, and waits for the network to be idle before evaluating the JavaScript on the page. This JavaScript is expected to have a variable called entries which the script is pulling out and adding to the csv variable. Finally, the script writes the contents of the csv variable to a file called entries log.csv using the fs.writeFileSync method and logs the contents of the csv variable to the console. The script then calls the runScriptOnWebsites function and passes an array of websites as an argument. The script also wraps the function call in an immediately invoked function expression to execute the function immediately.

```
{
  "manifest_version": 2,
  "name": "My Extension",
  "version": "1.0",
  "description": "counts the call of API’s.",
  "content_scripts": [
    {
      "all_frames": true,
      "js": [
        "background.js",
        "inject.js"
      ],
      "matches": [
        "http://*/*",
        "https://*/*"
      ],
      "run_at": "document_start"
    }
  ],
  "browser_action": {
    "default_popup": "popup.html",
    "default_title": "API COUNT"
  },
  "content_security_policy": "script-src ’self’ ’unsafe-inline’; object-src ’self’",
  "permissions": [
    "tabs",
    "storage",
    "unlimitedStorage",
    "webNavigation",
    "<all_urls>"
  ],
  "web_accessible_resources": [
    "/web_accessible_resources/*"
  ]
}
```

This code is a manifest file for a Google Chrome extension. The manifest file is a JSON file that contains information about the extension, including its name, version, description, and permissions. The manifest version property is set to 2, indicating that this is the second version of the extension manifest format. The name property is set to "My Extension", which is the name of the extension. The version property is set to "1.0", indicating that this is the first version of the extension. The description property is set to "counts the call of API's" The content scripts property is an array that specifies the JavaScript files that the extension will run on certain pages. The all frames property is set to true, indicating that the JavaScript files will be injected into all frames of the page, not just the top-level frame. The js property is an array that lists the JavaScript files that will be injected into the page. In this case, the files are "background.js" and "inject.js". The matches property is an array of URL patterns that specify the pages on which the JavaScript files will be injected. In this case, the extension will be injected on all URLs starting with "http://" or "https://". The run at property is set to "document start", indicating that the JavaScript files will be injected into the page as soon as the document starts loading. The browser action property sets the default title and default popup html file for the extension, it is set to "API COUNT" and "popup.html" respectively. The content security policy property is set to "script-src 'self' 'unsafe-inline'; object-src 'self'", indicating that the extension can load scripts from its own source and allow unsafe inline scripts. The permissions property is an array that lists the permissions that the extension requires. In this case, the extension requires access to the tabs, storage, unlimited storage, web navigation, and all URLs. The web accessible resources property is an array that specifies resources that can be accessed by pages outside of the extension. In this case, resources in the directory "/web accessible resources/*" can be accessed.

References

[?] [?] [?] [?] [?] [?] [?] [?] [?]