PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
UNIVERSITY MOHAMED BOUDIAF - M'SILA

FACULTY: Mathematics and computer Science
DEPARTMENT: Computer Science
N°:……

DOMAIN: Mathematics and Computer Science
BRANCH: Computer Science
OPTION: RTIC

Dissertation submitted to obtain Master degree

By: CHAKER TAYYIB RACHID

SUBJECT

MALICIOUS WEB BROWSER EXTENSION DETECTOR

Publicly defended before the jury composed of:

Ms.Bentrissa Rahima	University of M'sila	President
Ms.SAOUDI LALIA	University of M'sila	Supervisor
Mr.Bahache Mohamed	University of M'sila	Examiner

Academic year: 2018/2019

Acknowledgment

First of all, I'd love to show my appreciation and gratitude to Allah the Almighty who made the perfect condition for this work to see the light.

Anxiety and depression were overwhelming while trying to finish this project and it was only possible to fight through it with the support of some people that I want to thank individually.

I shall give the credit of leading this project to my supervisor MS. SAOUDI LALIA, who offered me the opportunity to work on this project and had enough patience to review and correct my work over and over. Let alone sharing her wise guidance and unclouded thoughts to make this project go on the right path. So thank you, it was pleasure to work with you even though I couldn't give my 100%.

Thanks to my mother who was supportive during my educational career over the years, she was the one who had always faith in me despite all.

Thanks to my friends GUESRI TARIQ and ZAHAR YOUCEF for being friends and life mentors, thanks to my class mates who made just the perfect atmosphere to make the past two years bearable.

Thank you all.

CHAKER TAYYIB RACHID

Table of content

List of Figures and Tables

List of figures:

List of tables:

List of equations

Abbreviation table

HTTP	HyperText Transfer Protocol
URL	Uniform Resource Locator,
URI	Uniform Resource Identifier
HTML	HyperText Markup Language
XML	Extensible Markup Language
DOM	Document Object Model
ME	Malicious extension
HTTPS	HyperText Transfer Protocol Secure
XHR	XML http Request
JS	Java script
CSP	Content Security Policy

General introduction

GENERAL INTRODUCTION

Nowadays, the web has taken a major part of people's life. Running a business, communicating, entertainment and many other activities took a place there which made people gravitate towards it and rely on it enough to let their data circulates on it. Web browser were the main cause of this, because they made the experience of surfing the internet easier, fast and relatively secure for the normal user. Due to that fact, big tech companies started a race to claim dominance on the web market by continuously evolving their web browsers through stuffing them with all sort of features. However, that came with a downside, filling the web the browser with variety of features had negative impact on the user experience, because it led to bottlenecking with the machines and network that most of people user to do the surfing which led to the creation of what we know today as browsers extensions. browsers extensions are pieces of software that run on the web browser offering all sort of features that could be performed on the web, those extensions received massive popularity among users who wanted to embrace these mesmerizing invention. But, not only users were attracted those extension, that popularity drawn the attention of attackers to abuse this invention and turn it into a tool that can compromise users' data, launch network and web attacks and execute restricted action on the victim's machine putting users' business and private lives at risk. And that's what we identify as browsers' malicious extensions.

Motivation

Browsers' malicious extensions are spreading like a contagious disease due many factors such as the irresponsibility of developers who let buggy coded extension slip through, unawareness of the normal user of the risks of installing an extension from an unknown source and last but not least, the techniques used by attackers to attract user towards their extensions and the method to bypass the security measures implemented within the browser itself.

Tracking users' activities, compromising personal and financial data and executing events without the user's realization are the sort of maliciousness that users are vulnerable to. Thus, raised a call to found an approach to learn and detect the illness within these pieces of software and prevent further abuse through understanding the fundamental mechanics behind them

Statement of the problem

Detecting malicious browsers' extensions is still an ongoing study currently, the approaches that exists nowadays still suffer from an incompleteness syndrome because of the single themed focus that most approaches adopted. Finding this kind of malice can be done through two unique aspects of detections:

1. **Static Analysis:**

The most common aspect of malice detection is running bunch of static analysis on the code of the examined extension trying to figure the existence of evidence or trace that leads to the finding whether the spotted extension is malicious or benign. And the techniques varieties at this point depending on the perspective of the detector.

In this type we can't detect malicious code that exists into an external file and called from an internal file.

2. **Dynamic Analysis:**

The dynamic aspect to detect malicious extensions offers several techniques for testing resulting the discover of new outputs that could help detector finding the righteous verdict to place right judgement upon the examined extension. This sort of analysis focuses on studying the behavioral reactions of the extension to determine its functionality.

Nowadays codes evolve fast and extensions too which increases the chances to use sophisticated techniques to hide its maliciousness. All existent detection techniques didn't take into consideration malicious extensions defense techniques to bypass any detection tool.

Objective

Pursuing our target to detect malicious extensions, we proposed an updated novel version of malicious features, flexible malicious patterns and adaption of techniques for dynamic analysis that weren't introduced in the previous approaches to detect malicious extensions within Google Chrome's Browser.

Contributions

To fulfill our task, we introduced the following contributions:

- An extended dictionary of malicious features that matches nowadays extensions
- An attack detector that has the flexibility to find patterns or sub-patterns to suggest in-depth analysis for alien patterns
- Detecting all sort API calls that happen within and out of the chrome browser
- Tracking processes triggered by chrome extension
- Detect whether the extension is running within the chrome isolated world or not
- Generating behavior profiles of extensions while surfing the web in real-time through different scenarios
- Overcoming self-defensive mechanism such as cloaking
- Obtained a balanced machine learning model to classify extensions into benign and malicious that is near perfect.

Report outline

This document was divided into four chapters:

- **The First chapter:** revolved around explaining web browsers architecture, chrome's architecture specifically, defining extensions and exploring their architecture, how they work and the implemented security model by Google Chrome for its extensions.
- **The Second chapter**: focused on overview evaluating the already existing approaches used to detect malicious browser extensions and how they do it, closed by a technique based comparison.
- **The Third chapter:** Introduced our claimed approach equipped by our conceptual model, techniques and tweaked methodologies to detect malicious extensions.
- **The Fourth chapter:** contained the implementation and experimentation measures we made to make our approach with scores we achieved.

  And finally, a general conclusion for cloture.

Chapter 1

Browser and extension architecture

1.1 Introduction

Since the dawn of founding the internet as we know it today, Web browsers were and still are the main portal that allows user to access it. therefore, it had to handle the continuous evolving of web application such as websites which became more than bunch of static pages but more like into a rich interactive environment that is built upon different languages and scripts. With this expanding in web, browsers' developers gave the opportunity for others to add extra features to their browsers by installing or programming their own extensions which will add various features to the original browser's ones. However, this came with a cost of making users exposed to malicious extensions and vulnerable ones.

1.2 Browsers Architecture

In order ensure browsers functionalities, there are core components that every browser has. Which are illustrated in Figure 1.1, the core components are [1]:

- **User interface:** which contain the address bar, menus, dynamic buttons such as forward/backward buttons.
- **Browser's engine:** coordinate between the rendering engine and the user's interface.
- **Rendering engine:** displays the content of the requested pages (ex: HTML and CSS content).
- **Networking:** handles network calls such as HTTP requests.
- **UI Backend:** Draws basic widgets, windowing primitives and exposes a platform-free interface. [2]
- **JavaScript Intercepter :** parses and executes JS codes [1].
- **Date Persistence**: meant for locally storing all kind of data such as cookies [1].

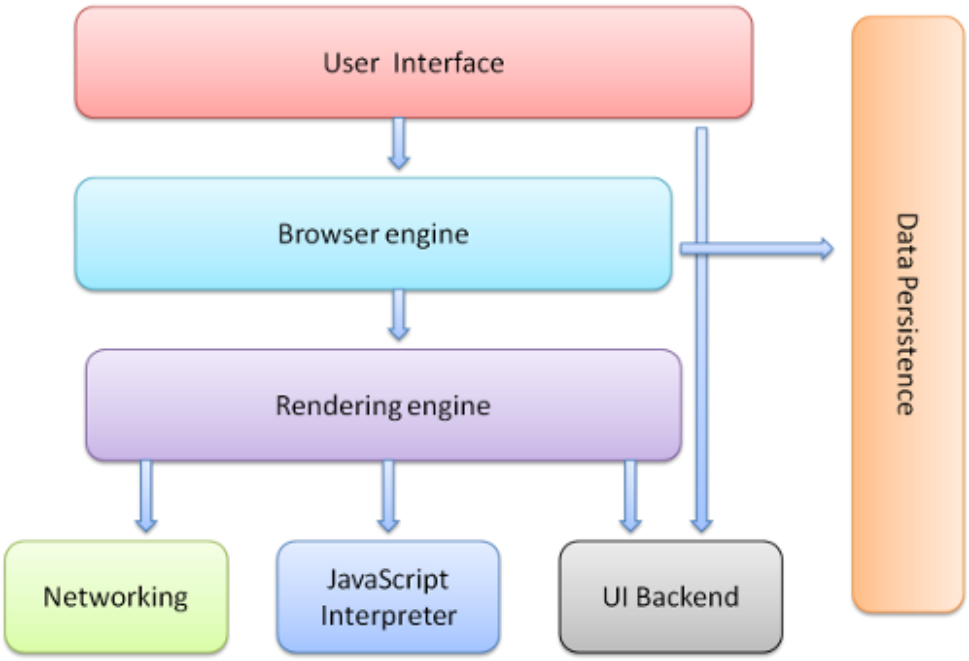

Figure 1.1: browser's core components. [1]

1.2.1 User Interface (UI)

UI is the graphical interface that a web surfer would use to make navigating through the web easier, there are many web browsers with different features out there, such as Google Chrome and Mozilla Firefox.

Most of the web browsers share same external structure as demonstrated in figures 1.2 which contains the following:

- **Back** and **forward** buttons to go back to the previous resource and forward respectively.
- A **refresh** or **reload** button to reload the current resource.
- A **stop** button to cancel loading the resource.
- A **home** button to return to the user's home page.
- An **address bar** to input the Uniform Resource Locator (URL) of the desired resource and display it.
- A **search bar** to input terms into a search engine. In some browsers, the search bar is merged with the address bar.
- A **status bar** to display progress in loading the resource and also the URI of links when the cursor hovers over them, and page zooming capability.

Figure 1.2: Chrome's UI

1.2.2 Data Storage

Data storage also known as **Data persistence** usually saves and manages user's data such as cookies and bookmarks usually stored on the user's drive. When HTML5 got introduced, it became a light complete database in the browser itself.

1.2.3 The browser engine:

The browser Engine plays the role of the coordinator between the User Interface and the rendering engine by marshaling the actions between the two. Due to this a high level interface is provided, beside other methods to load URL and browsing actions such as going forward or backward as defined in UI section.

The browser engine also offers to the user feedback texts, worth to mention like error messages and progress statue

1.2.4 Rendering Engine:

Also known as layout Engine, it's one of the core component of web browsers. It displays markup languages content such as HTML and XML [3] plus other type of data via plug-ins or extensions like PDF content. [2]
Some browser runs an instance of the rendering engine for each tab.[3]

Each browser has its own rendering engine, some of the engines that are worth mention are [3]:

- Gecko used by Firefox
- Webkit used by Safari and Chrome
- Presto by Opera

1.2.5 Rendering Engine Keys

The rendering engine has three major keys:

- HTML parser: parses the html into a parse tree
- DOM: document object model, is the output tree (DOM elements and attribute nodes) [3]
- CCS parsing: a context free grammar[3]

1.2.6 Rendering Engine's basic flow

The rendering engine's basic has four main steps illustrated in Figure 1.3, Step one does the HTML parsing and conversion to DOM nodes into the content tree[3], the second step parses the CSS files and style elements into the render tree[3], the third step is the layout process where each node gets its proper coordinates to be displayed on[3], and the fourth and last step is painting the render tree.

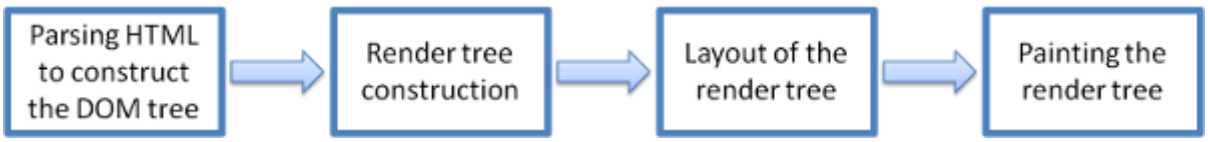

Figure 1.3: Rendering engine's workflow. [1]

1.3 Google Chrome's browser

Google chrome is the most popular web browser in the main time with more than 55% of the market shares. [4] the implementation of this browser isn't fully open sourced however googles provided an open sourced project called "chromium project" [5] which is designed while considering security measures in order to maintain a safe and fast web experience. [6]

Chrome uses **WebKit** as a rendering engine, but later on it changed to **Blink** which is **WebKit** based for more flexible architectural changes and performance enhancing reasons. [7], figure 1.4 show the architecture of **WebKit** rendering engine.

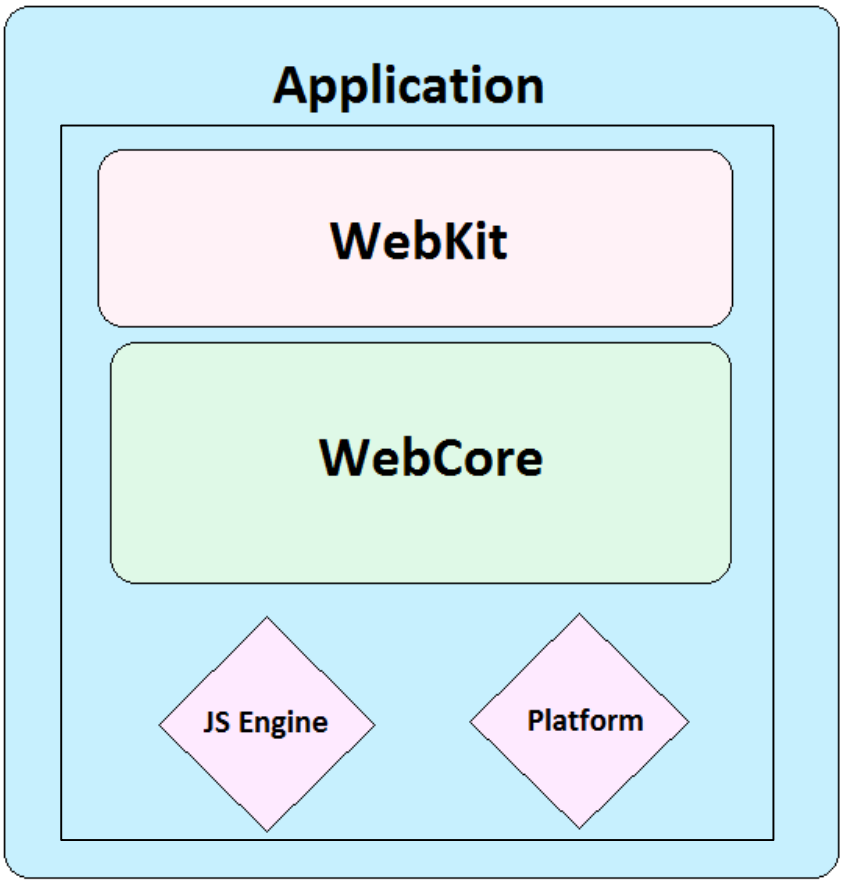

Figure 1.4: WebKit architecture

- **WebKit :** thin layer to seperate from the application
- **WebCore :** contains all network access, layouts, accessiblity support and rendering.
- **JS Engine :** Mainly provided by JavaScriptCore framewrok to handle scripting for WebKit
- **Platform :** bound to platform to implement generic algorithms [10]

Being the most used web browser on the surface, made Chrome the main target of attackers, and since Chrome's extensions became widely used due to the features and functionalities, they started being abused by those attackers.

1.4 Chrome's extensions

Chromes extensions are smalls programs made in order to add more functionality to the browser, they're made using HTML, CSS and JavaScript. An extension should fullfill one and only purpose, however it can contain multiple components as long as it serves a single specific purpose.[9]

1.4.1 Extensions types

Chromes offers three types of extensions[4]:

- Themes: only meant to modify the predefined elements using CSS, all modifications must be declared in the manifest file
- Extensions: can request access to APIs in order to obtain privileged actions, however extensions cannot perform system calls on their own
- Apps: Unlike extensions, apps are not meant to change the browsing behavior, however they're developed with the same technology but they have more access to APIs

1.4.2 Extensions architecture

Chrome's extensions are zipped into.CRX package, it consists two main elements, the content and the manifest file. The extensions will be installed locally, that means they're independent of the web content unlike web applications, Figure 1.5 illustrates the architecture of chrome's extensions.

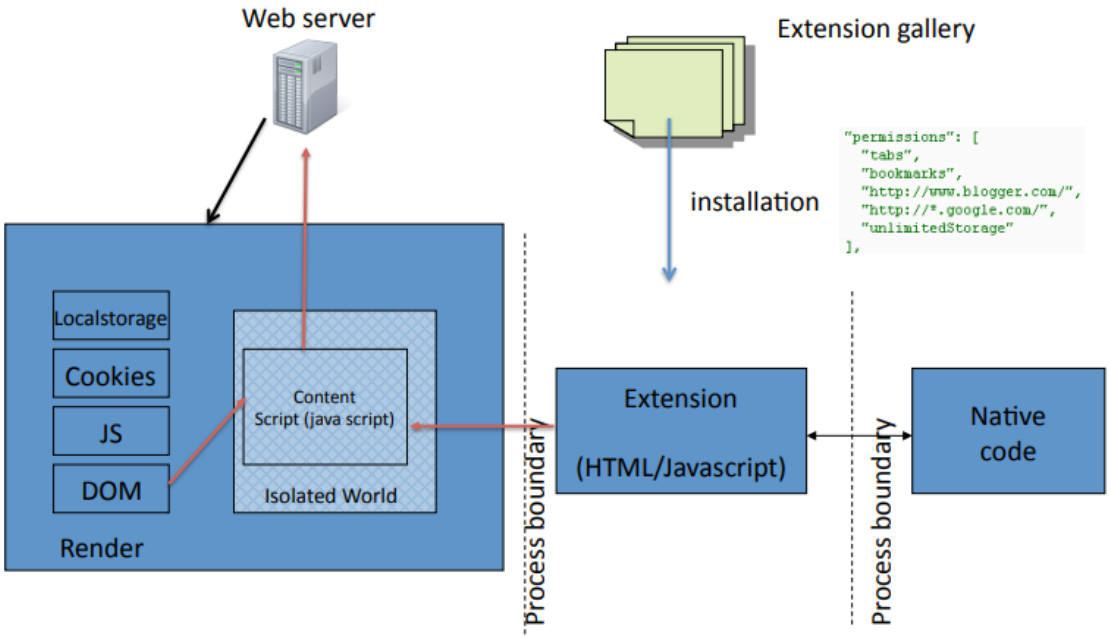

Figure 1.5: Chrome's extensions architecture [10]

1.4.2.1 Content

The content is the code used to program the extension, it's built using HTML, CSS and JavaScript.

- It contains most of the codes such as the design lines to decide how the extension is displayed using HTML and CSS languages and the other design materials
- It has the code written in JavaScript which defines the functionality of the extension, such as the intended text to display or the wanted service from this last. Figure 1.6 shows an example of the **content.js** file to display "hello world" message, Figure 1.7 shows the result of the code.

```
<html>

<head>

<title>Chrome Hello World Extension Example</title>

<style>

body { width: 300px; height: 200px; }

</style>

</head>

<body>

<strong>Hello World</strong>

</body>

</html>
```

Figure 1.6: Hello World extension sample

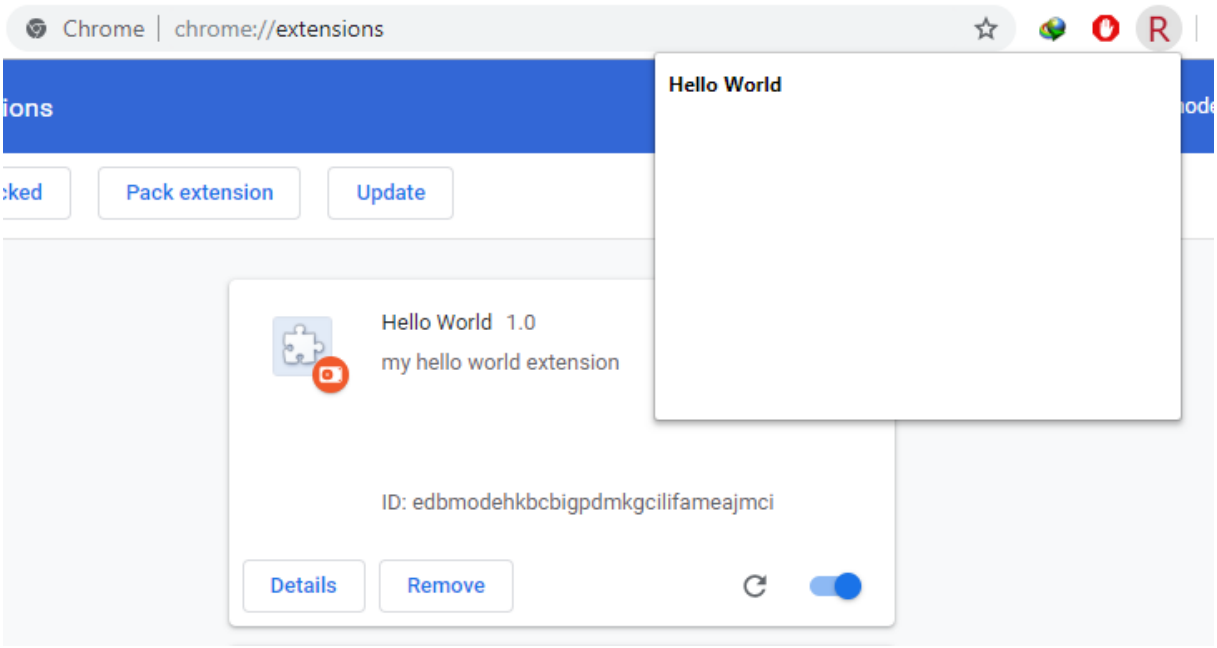

Figure 1.7: Display of hello world extension sample

1.4.2.2 Manifest

The manifest file contains the information about the extension such as the name of the extension, the description, the version, a popup file and icon declared under the field of browser action and all permissions requested. Figure 1.8 shows an example of the manifest file. The manifest file is written in JSON.

```
{
  "name": "Boxcutter",
  "description": "my manifest example",
  "version": "9",
  "manifest_version": 2,
  "permissions": [
    "https://registry.npmjs.org/*",
    "https://rubygems.org/*",
    "https://hex.pm/*",
    "https://bower.herokuapp.com/*",
    "https://packagist.org/*"
  ],
  "content_scripts": [
    {
      "matches": [
        "https://github.com/*"
      ],
      "js": [
        "jquery.min.js",
        "content.js"
      ]
    }
  ],
  "icons": {
    "128": "icon128.png"
  }
}
```

Figure 1.8: manifest.json file

1.5 Chrome' Security Model

A sophisticated security model is a must for a good designed extention to avoid stability issues and vulnerabilities. As of 2014, chrome abandoned extensions with manifest version number 1, and introduced 2nd version which became mandatory, on this version came with two restrcitions on extensions[9]:

- A default CSP (Content Security policy) must be respected by devoloppers
- The resources of chrome's extensions are not web-accessible by default

Each type of chrome's extensions got a slightely different security model

1.5.1 Theme' security model

can only modify colors, background and and very few properties of the browser's layout. Styling has to be declared into the manifest file in JSON structure as shown in Figure1.9.

```
script-src: 'self'; object-src 'self'
```

Figure 1.9: Default CSP for version 2 Extensions

1.5.2 Extensions' security model

Beside the default CSP, all used APIs has to be requested in the manifest file, each access has to be stated in the permission list, so users can see all the requested permissions during installation. "extensions can only include external scripts from their own package. This explicitly disallows any usage of inline scripts and eval-like statements in order to mitigate most XCS attacks. A developer can relax the policy by allowing eval-like statements with the unsafe-eval keyword in a value of its manifest file, whereas the inline scripting restriction cannot be lifted. Additionally, external resources can be allowed but only with severe limitations" [9]

Extensions has to use content script to interact with web content while being enviromentaly seperated. Figure 1.10 shows the security boudries between web content and extentions.

Content script have to use DOM to communicate with web sites and vis versa other wise the website can't access the extension's context either.

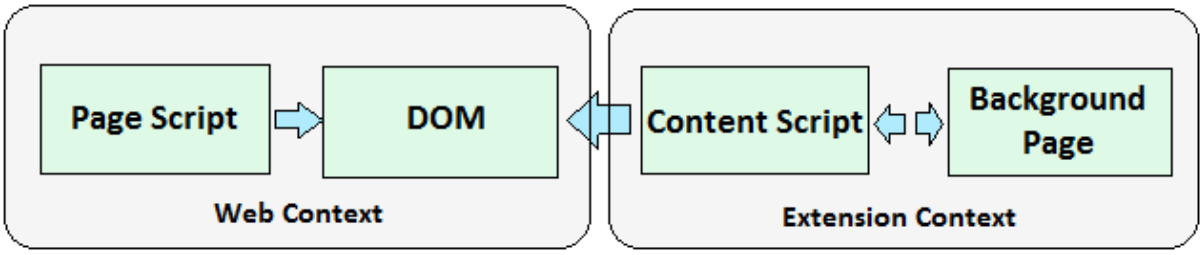

Figure 1.10: security boundries

1.5.3 Apps' security model

Share same restrictions as Extension as mentioned before but has access to privileged APIs Figure 1.11 shows the applied rules. However devoloppers has to use other mechanisms provided by chrome such as *SandBox pages* to be able to use *eval()* in particular [9] due to the fact that those pages don't have high privileges which means they cannot apply a harmful impact.

Communications occur via APIs like *postMessage*, Apps obtain a mechanism to embed untrusted content by using *WebViews* while using *iframe* doesn't allow the embedded code to reconigze being framed or not.

In addion, chrome web store forces a review process upon every extension before reaching the surface, so obvious security flaws would be detect beforehand.

```
default-src 'self';
connect-src * data: blob: filesystem:;
style-src 'self' data: chrome-extension-resources:;
img-src 'self' data: chrome extension-resource:;
frame-src 'slef' data : chrome-extension-resource:;
font-src 'self' data: chrome extension-resource:;
media-src * data: blob: filesystem:;
```

Figure 1.11: Default CSP for version 2 Apps

1.6 Conclusion

On this chapter, we had 4 main sections that were respectively about the architecture of web browsers, precisely google chrome's browser, followed by introducing extensions and finally explaining the security model implemented by chrome for web browser's extensions considering the security flaws that may occur during their development/launch and the techniques that are being used within.

Chapter 2

Related works

2.1 Introduction

Malicious Extensions became viral nowadays and new methods and approaches to detect them became a must in order to ensure the safety of the user while navigating the internet through their preferred browsers. therefore, Multiple analyzing techniques have been introduced in the past few years in order to attempt distinguishing these malicious activities.

In this chapter, we are going to spot on some of the most efficient approaches that are currently used to determine the maliciousness of a browser's extension and put the light of some critical aspect of those last ones.

2.2 Safe Browsing

Safe browsing is a service that was found by Google, it examines content and URLs for the sake of finding unsafe websites or unsafe web content to prevent personal data theft, installing malwares into victims' devices or taking full control of it.

> "Every day, Safe Browsing discovers thousands of new unsafe sites. Many of these are legitimate websites that have been compromised by hackers. Unsafe sites fall into two categories that threaten users' privacy and security: phishing and malware. "[11].

2.2.1 Safe Browsing's vision

Safe Browsing considers any **infected content** (whether intentionally or not) as **malicious** one in order to prevent compromising user's information or machine by blocking any sort of activity such as installing unknown software without the user having the awareness of its impact. Those malwares could be ransomwares, spywares, viruses, worms or even Trojan horses [11].

2.2.2 Safe Browsing Detection method

Safe browsing does periodic scan on the web to try to determine in which the site is being compromised or if it's an attack site, the scanning runs through virtual machines for analyzing the content in order to figure out hint that may lead to identify the sanity of the site. **Figure 2.1** shows the number of compromised and attack sites that have been detected by **Safe Browsing** during July 2018 and march 2019.

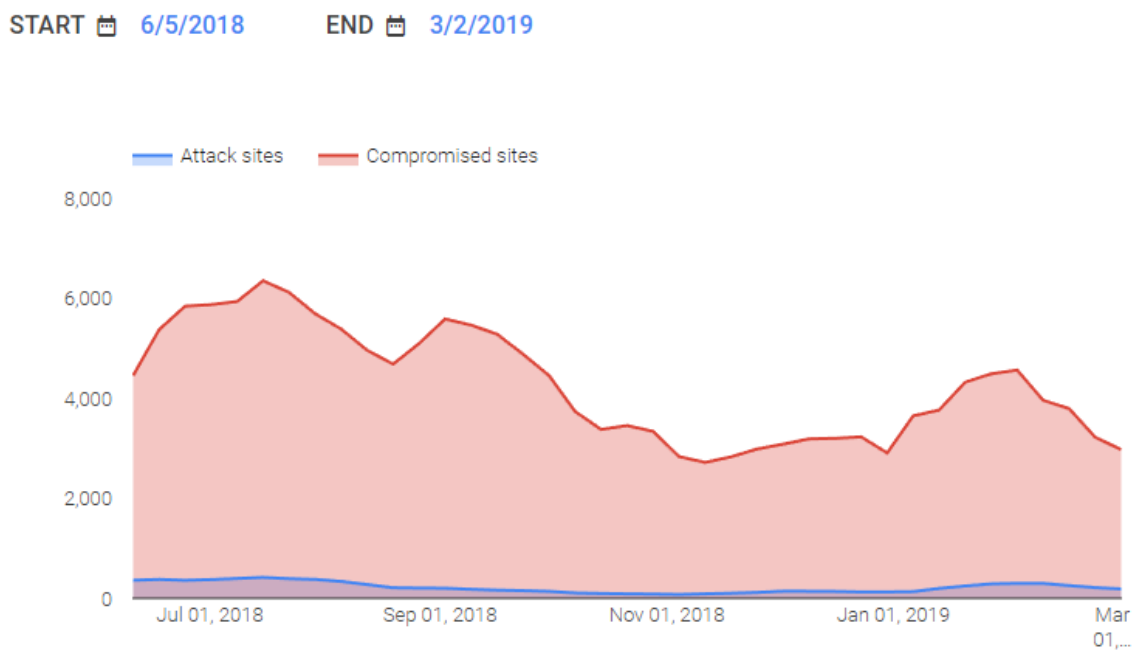

Figure 2.1: site detected by Safe Browsing, July 2018- March 2019 [11]

2.2.3 Attack site

Safe Browsing consider a site as an attack site if it finds signal that proves that its developed intended to spread malicious content throughout his site, meaning he intentionally to cause harm to the user's date or device.

2.2.4 Compromised sites

Compromised site contain malicious content but without the ill wiliness of their developers, attackers abuse bugs and security holes to include those malware content or redirect users to theirs.

2.2.5 Safe Browsing's evaluation

Safe Browsing contains the gigantic on growing data base of the attack and comprised sites, however, detecting malicious behavior is only noticeable through the changes on the web content and it doesn't include changes on the browser or the network activities.

2.3 Hulk Approach

Hulk [12] consists a dynamic analyzing method to elicit malicious behaviors in browsers extensions, it uses two-way scan, the first based on **HoneyPages** and the second is a **Fuzzer** for the event handlers. [12], **Hulk** classifies extensions as benign, suspicious and malicious ones.

2.3.1 URL extractions

Usually in order for malicious extensions to bypass their limitation of executing their scripts on the web pages, they include URL to load content from defined origins.

Hulk uses the **manifest file** where there the URLs should be indicated, also determines if the extension has cross origin permissions. **Figure 2.2** shows the top 25 hosts found by Hulk. And the **content file** which has the source code of the extension in order to extract the hardcoded URLs. **Figure 2.3** shows the top 25 hosts detected by hulk.

Hulk visits all the URLs that been extracted from those two sources to try to detect a malicious behavior, and runs a comparison with the **popular sites** that've been targeted by malicious extensions before.

Rank	Top 25 hosts in permissions	# ext.
1	http://*/*	7,319
2	https://*/*	6,395
3	<all_urls >	2,044
4	http://*/	1,126
5	*://*/*	1,025
6	https://*/	665
7	www.flashgame90.com/Default.aspx	224
8	https://api.twitter.com/	200
9	http://localhost/*	161
10	http://127.0.0.1/*	133
11	https://secure.flickr.com/	95
12	*://*.facebook.com/*	91
13	*://*/	89
14	https://www.facebook.com/*	82
15	http://vk.com/*	77
16	http://*.facebook.com/*	77
17	https://mail.google.com/*	71
18	https://*.facebook.com/*	70
19	http://*.google.com/	68
20	https://www.google-analytics.com/	67
21	https://mail.google.com/	64
22	https://*.google.com/	62
23	https://twitter.com/*	61
24	https://www.googleapis.com/	60
25	google.com/accounts/OAuthGetAcc[..]	56

Figure 2.2 top 25 hosts in permissions [12]

Rank	Top 25 hosts in content_scripts	# ext.
1	http://*/*	12,472
2	https://*/*	10,864
3	<all_urls>	4,795
4	*://*/*	1,536
5	https://www.facebook.com/*	520
6	*://*.facebook.com/*	510
7	https://mail.google.com/*	458
8	http://www.facebook.com/*	433
9	https://*.facebook.com/*	344
10	http://*.facebook.com/*	320
11	file://*/*	315
12	https://twitter.com/*	303
13	http://mail.google.com/*	273
14	*://pages.brandthunder.com/[..]	265
15	https://plus.google.com/*	261
16	ftp://*/*	246
17	http://vk.com/*	227
18	http://www.youtube.com/*	211
19	file:///*	207
20	*://mail.google.com/*	189
21	http://twitter.com/*	179
22	*://www.facebook.com/*	178
23	http://ak.imgfarm.com/images[..]	177
24	*://*.reddit.com/*	164
25	https://vk.com/*	164

Figure 2.3 top 25 hosts in content script [12]

2.3.2 HoneyPages

HoneyPages are dynamically generated pages that provides the necessary environment for some extensions to execute, by simulating the requested element such as Ad division, media sections or other content.

Some extensions tend to run only when certain elements are present on the web pages, to offer this, **HoneyPages** override functions used to query the DOM tree of the page [13]. So when the extension queries it, the **HoneyPage** auto generates it and by this, further behaviors could be analyzed. **Listing 2.4** shows the top 15 chrome API called, according to Hulk

Rank	Top 15 chrome.* APIs called	# calls
1	runtime.onInstalled	182,476
2	webRequestInternal.eventHandled	57,466
3	tabs.getAllInWindow	49,312
4	tabs.onUpdated	32,354
5	tabs.create	25,947
6	i18n.getMessage	13,549
7	webRequest.onBeforeSendHeaders	13,213
8	runtime.connect	13,004
9	extension.getURL	11,942
10	storage.get	10,178
11	contextMenus.create	7,816
12	tabs.get	6,970
13	webRequest.onBeforeRequest	6,168
14	runtime.sendMessage	5,847
15	extension.sendRequest	5,454

Figure 2.4: top 15 chrome APIs called [12]

2.3.3 Triggering Event (fuzzer)

Some extensions get triggered when some browser events are occurred due to the fact that those extensions are JavaScript made, and this one is mostly event driven.

Hulk triggers event registered through the **chrome.WebRequest** API to intercept HTTP requests using mock events. By doing this, further extension behaviors could be examined.

2.3.4 Monitoring

After **URL extraction, Honeypagess** and **fuzzing** are done, Hulk uses monitoring hooks such as, monitoring APIs, logs when a **chrome.webRequest** is present[13], It logs all kind of changes on HTTP requests and the JS code, in which would elicit the behavioral functionality of the extension further more.

2.3.5 Profiling Extensions

Hulk labels the extensions after finishing the monitoring and decides whether they're malicious, benign or suspicious. Besides collected data from monitoring hooks Hulk considers any extension that attempts to install or uninstall any extension, Manipulating HTTP headers or inspecting DOM elements as malicious [12].

2.3.6 Hulk's evaluation

Even though Hulk redeems various monitoring methods it still suffers from multiple limitations such as the following:
If the manifest file has **all_urls** or **many URLs** it becomes very difficult for Hulk to examine it

- Loaded pages from same URLs are considered unchanged where in fact they could be changed over time
- Extensions may not show malicious behavior due to **HoneyPages** not providing the latest properties that have been requested
- Hulk does examine the event registered through **EventTarget listeners** which mean Hulk doesn't cover all possible scenarios

2.4 WebEval Approach

WebEval [13] is another tool that was developed in google lab by **Jagpal et al** to detect malicious activities in browser's extensions and it was the first of its kind that return a highly accurate results.

WebEval's approach has several combined techniques to study an extension's behavior, it analysis its code, the publisher's reputation and mostly bases the verdicts on its classifier with the validation of a human expert [14].

2.4.1 Static Analysis

WebEval uses a static code analyzer to enhance its classifier, however the decision whether the extension is malicious or not is fully based on an expert verdict.

It scans the sources code of the browser's extension which includes its HTML, CSS and JS code and the manifest file, also it takes in consideration the identity of the publisher, date of publishing and duplication to prevent republishing of the same extensions by comparing some info located in the manifest file such us the name of the extension and description [4]. And it follows this order:

2.4.1.1 Permission & Content Scripts:

First, the content scripts and permissions are fetched from the extension files since its open sourced, then a scan runs over the code to indicate if there is any sign that tells about the extension capabilities of intercepting or altering web requests or traffic, tab events, intercepting browsing data such as cookies and the extension context.

2.4.1.2 Code Obfuscation:

According to **Kaplan et al**, [13] the scan runs to look for **Minification** which is measuring the distance between the original code and the prettified one, **Encoding** by searching for encoded strings and **Packing** by using regular expression.

2.4.1.3 File and directory structure:

Improving the first step (**Permission & Content Scripts**) by extracting file names and directory to find out if there is duplication.

2.4.2 Dynamic Analysis

The roots of the dynamic analysis are black-box testing and sand-boxed tests in order to simulate near real browsing behaviors while examining the changes that occur during the simulation [13]. It starts by running the extension in a windows virtual machine with system and browser monitoring for the goal of intercepting all type of requesting and reviewing the changes to the system. The simulation is recorded in order for the human expert to examine it for final verdict.

2.4.3 Developer Analysis

On this step, the developer of the targeted extension is the one being examined, reputation of the developer, its logs such us email domain and login location, age of his account, the amount of extension made by this developer. All of this to determine the true identity of the of developer in case the actual account own is being abused by a third party or an attacker. Because usually attacks target developers with good reputation to evade the initial detection by chrome store

2.4.4 Annotation

To enhance the primitive static scan, all the files of the extension are being scanned by multiple anti-viruses and external security experts to examine it further with their own tools in which would improve the verdict accuracy in case something wasn't noticeable through WebEval's analysis and experts.

2.4.5 WebEval's Evaluation

Even though, **WebEval** achieved high accuracy during detecting malice in extensions. Their based foundation is still poor compared to some other approaches [3].

- The static scan cannot determine whether the extension is malicious or not without the human verdict
- The dynamic analysis could be evaded through **Cloacking** techniques which delay the malicious execution behavior until the examination is done

2.5 Model-Based Approach:

This approach is developed by Hosain Shahriar, Koominist Weldemariam, Mohammad Zulkernine and Thibaud Lutellier at the university of Kennesaw State University, USA, their work consists a model-based approach fully depending on existing methods and techniques, it fed with benign and vulnerable characteristics in order to detect malicious extensions [15]. It built upon the Hidden Markov Model (HMM), The detection takes two phases which are **Model Generation** and **Model Matching** led by **Modeling Browser Extensions.**

2.5.1 Modeling Browser Extensions:

“The type of a browser extension can be identified by thoroughly analyzing its distinguishing characteristics while in operation. These characteristics help determine the behaviors of the extension and thereby detect its type automatically”. [15]

Considering that statement, the core of the approach is based on determining the type of extension using its characteristics as compass, for an example benign extensions should only react according to the user’s actions, like clicking event and such while malicious might run otherwise without the control of the user themselves, the following table 2.1 shows the characteristics of the various types and cross types of extensions. Some characteristics overlap among the types.

Table 2.1: Characteristics of benign, vulnerable and malicious browser extensions. [15]

Type	Sample characteristics
Benign	• Visible interfaces • User events are initiates functionalities • APIs can render web content with input filtering, have access to local storage(read/write), generate web requests and provide info to whitelisted sites
Vulnerable	• Visible interfaces might not be present • Events could initiate unintended functionalities • APIs render web content without input filtering and may render requests to whitelisted sites
Malicious	• Visible interfaced are non-existing • User events initiate unintended functionalities • APIs could generate new tabs based on obtained data from remote sites, render web requests to unlisted sites and might access local storage(read/write)
Benign or Malicious	• Visible interfaces may be present • User events might initiate functionalities

	• APIs render web contents with input filtering and may render web requests to whitelisted sites
Vulnerable or Malicious	• Visible interfaces may be present • User events might initiate functionalities • APIs render web requests to whitelisted sites and render web content without input filtering

2.5.2 Model Generation:

In order to determine the different classes of extensions as mentioned before, an HHM is used where extension's features and hidden states are being observed. The algorithm goes as in Figure2.5. [15]

```
Read Operation
for all i=1; i<= N; i++ do
   for all j=1; j<= M; j++ do
      cntOccurence= 0
      K=0
      While (K < ObservationSize) do
          if (vraw [k] == v [j]) then
             ++ cntOccurence
             k++;
          end if
      end while
      b[i] [j] =cntOccurence /ObservationSize
   end for
end for
```

Figure 2.5: Output Probability Matrix Computation

After computing the parameters, the three distinguished models are created according to extension classes mentioned before and therefore, detection models are generated, benign, malicious and vulnerable.

2.5.3 Model Training:

After the signatures are made from the previous phase, a training model is set to adjust and accurate the parameters used for the model, a sequence is made through extracting extension's features.

2.5.4 Model Matching:

The matching is the second phase of the approach, after maintaining the training phase, the 3 detection models are used for cooperation with unknown extension to see which of them matches which. Meaning if an extension matches more the vulnerable model, therefore it's vulnerable.

2.5.5 Approach evaluation

The approach bases its result on categorizing the extension into three types, malicious benign and vulnerable. The method might be efficient to determine the similarity between those types and analyzed extensions; however that doesn't necessarily reflect its accuracy.

2.6 A lightweight malicious extension detection (LMED):

A lightweight malicious extension detection (LMED) system was developed by *Dr. Gauvar Varshney, Dr. Manoj Misra* and *Dr.Pradeep K. Atrey* to achieve better results compared to other ME detection systems. Mainly their work was based to identifying malicious signatures and run a static comparison between the dataset containing the signature and the analyzed extension. [16].

The LMED focuses on finding those ill signatures within the source code of the extension in which a source code reader and an analyzer were required to do the task. The work flow is explained in the following sections

2.6.1 Source Code Reader (SCR)

Chrome extensions have a defined directory where the files containing the core code are located, the default directory goes as the following:

C:/Users/Gaurav/AppData/Local/Google/Chrome/User Data/Default/Extension

The JS file is fetched from the mentioned location and parsed to the JS code Analyzer.

2.6.2 Static JS code Analyzer (JSCA)

After the source code is fetched, a static scan is run over the code to see if there is any matching with the signature dataset, if there is a positive match, the user shall be informed of existence of malice.

2.6.3 Malicious Signature Dataset (MSD):

Knowledge about cyber fraud cyber spying has been gathered during the study of the developers in which they found that those malicious methods have some common content. Putting that in mind, a dataset of regular expression has been stored into the MSD containing all the known common signatures. The study had 500 benign extensions and 20 malicious ones, from where the signatures were extracted.

2.6.4 LMED's Evaluation

The LMED achieved indeed progress considering the previous works, regardless of the fact that is only basing on top of a static analyzer. However, malicious extensions evolve every day and methods used changes overtime. So having a static scanner based on only known signatures wouldn't afford to prevent irregular pattern that may be used in the ME.

2.7 A Combined Static and Dynamic Analysis Approach to detect Malicious Browser Extensions

In this approach, two combined aspects **(static and dynamic analysis)** have been merged to detect malicious extension. The goal of this approach is to extract the features from the JavaScript, html and CSS content of the extension, after those features are extracted; a classifier takes control to determine which feature is malicious, basing its verdict on the features gathered. [17]

2.7.1 Methodology:

The purpose of the study was to obtain a model that can binary classify extensions to either malicious or benign ones using static and dynamic features contained in the extension. The static features were fetched from the files of the extensions and relevant ones as well. while the dynamic ones were found during the execution of the extensions on a testing infrastructure. Later on, a supervised machine learning model is used for the classification.

2.7.2 Approach evaluation

The used model achieved 95.18% accuracy in test phase, which is a really high score. One of the fundamental weaknesses is the environment where the extensions were tested, so the accuracy of test might be questionable for some cases in nowadays extensions.

2.8 Approach Comparison

We choose the criteria mentioned in **table 2.2** to evaluate the efficiency of the approaches reviewed in this chapter, we took in consideration most of the examination points that are being used to detect malwares nowadays and put the approaches on the spot light to see in which they fulfil the requirements.

Table 2.2: comparison table according to criteria

	Safe Browsing	Hulk	WebEval	Model Based approach	LMED	Combined approach
Fetching source code of the extension		✓	✓	✓	✓	✓
code transformation			✓			
Automated static detection	✓		✓		✓	
Automated Dynamic detection	✓					
Sand box analysis		✓	✓			✓
Real runtime analysis	✓					
Intelligent classifier			✓	✓		✓
Human Expert validation			✓			✓

Safe browsing is the only tool from the six examined ones that run a real time examination during browsing however it that comes with cost of sacrificing lot of key criteria compared to the others such as examining the source code of the extension itself and not only the changed on the web pages

Hulk is based on a unique approach to check all URLs and permissions requested by the extension, however it only statically determine those links and doesn't consider the changes during the runtime beside, it suffers from limitation during fetching all URLs in case there are too many to examine

WebEval, is highly accurate compared to all the existing approaches, it satisfies most of the criteria whether statically or dynamically however it has 2 major flaws as all dynamic analysis, which are the cloaking issue due to the short period when the extension is examined and the fact that it only relies on sand-box test therefore it's not able to detect malicious behaviors while being used in normal conditions by users

The model based detection approaches introduced a technique to classify browser's extensions despite of not having an accurate malicious code pattern, but it lacks analyzing basics and only works mainly on stereotypes of extensions.

In contrary, the approach based on studying fraud and spying techniques have better analyzing techniques applied into it, however, basing the verdict on only known patterns creates a runaway gate for attackers to develop ME that can slip through

The combined static and dynamic approach, focused on defining extension features unlike all the previous approaches, which made it more agile when it comes to determining malice, even though, fundamentally, basing the verdict on features alone and not behavior cannot always be true due to fact that nowadays codes evolve fast and extensions gets continuous updates which increases the chances to sharing lot of features between malicious and non-malicious extensions.

2.9 Conclusion

In this chapter, reviewed existent approaches, explained how they function, evaluated each one of them and summed up with a comparison between them.

Chapter 3

Malicious Web Browser Extension Detector (MWBED)

3.1 Introduction

Inspired from the previous approaches to detect malious extention, we build our model while enhancing, tweaking and tuning the techniques adapted within. Our approach and contribution was hydration of statical and dynamical analysis, however the enviroment and intercepting methods are different, due to the fact that malicious extensions can react differently while being in a closed enviroment or being intercepted by third party tools.

We considered the results of scanning using the well known anti-malwares and spywares as an initial judgement and we proceeded upon the results where there was no accurate detection or benign one.

3.2 Malicious Web Browser Extension Detector (MWBED) mechanism

Our Malicious Web Browser Extension Detector (MWBED) mechanism has multi steps (figure3.1) ; scanning, each focuses on one of the aspects where malicious content of behavior could be detected, starting from the prejudgment of existing methods, to static scanning where malicious patterns are being fetched. passing to extended scanning for addional files, to land on the behavioral test where the final detection phase takes place. The result takes two types of verdicts, either malicious or benign, which eventually will provide our classifier a training model. All the data gathered will be benchmarked to enhance future scans.

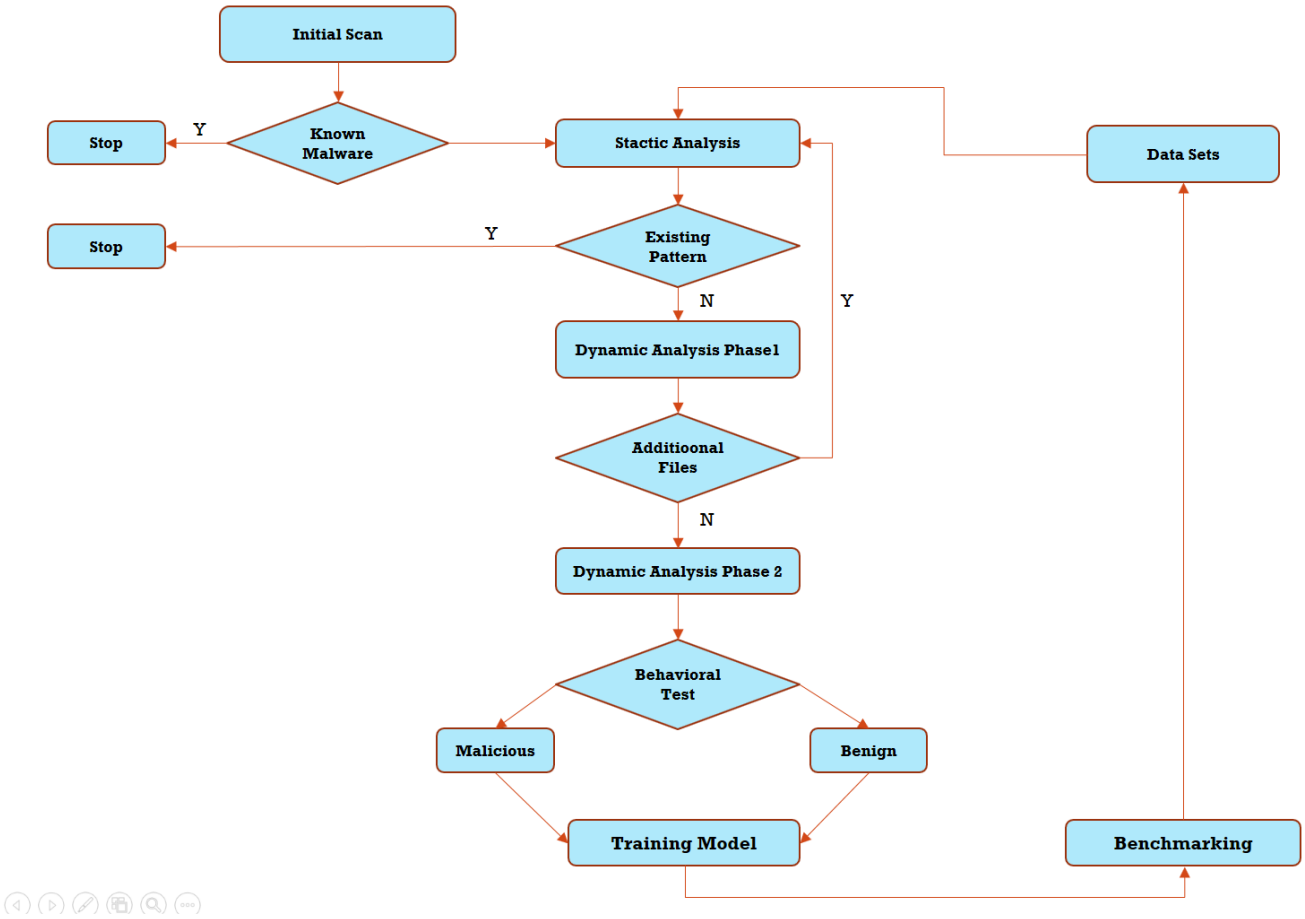

Figure 3.1: MWBED flow chart

3.2.1 Initial Scan

Our initial goal is to detect malicious threat in browser's extension that weren't found beforehand, or detect those which weren't been detected using the traditional methods/approaches. Therefore, we picked *TotalVirus [18]* which is a collaboration tool of some of the most known anti-malwares and anti-spyware such *Kaspersky* in order to enhance our scanning quality, *TotalVirus* results could determine if there is a known threat (specific virus), declare potential threat if there in an unknown or risk such as encrypted content which may or may not be an actual threat, and of course if it's benign where the tools used cannot determine any known threat.

We took in consideration only the last two results mentioned above, the benign and the unknown threat ones, thus we could go further with our analyzing method to see in which case the extension is truly benign or otherwise, and even deeper into the core of the extension with unknown threat.

The two types are later parsed to our first detection step which is the **static analysis**.

3.2.2 Static Analysis

Before jumping up to conclusions about the sanity of the parsed extensions, we had to look in the core code of it, by parsing both the content code (JS script) and the manifest file where requested permissions and data about the extension are located.

The code passes through multiple processing and scanning at this point, starting by extracting the code out of the extension files for sake of obtaining a readable code that can be analyzed later through our automated scanners or human expert.

Scanners uses logical tests based on malicious extension features that were extracted and reviewed previously side to side with newly found ones to be able to identify known malicious patterns or discover new ones.

3.2.2.1 Dumping the extension code:

One of the most common methods to make analyzing a code difficult is adding white space, removing identical outlines; use JSE (JavaScript Encoded) to encrypt code and adding extra symbols to extend code's length.

A method to simplify the code was required to advance to our analyzing, as solution to that, we parsed the known files about the extension to static scanner, but before launching the scan, we used code prettifying method followed by tokenizing the code as shown in **figure3. 2**, thanks to that each element of the code can be viewed separately without any confusion.

Figure 3.2: dumping the code

JSE parts are dealt with separately from the rest of the code due the fact that there is no automated tool that can decrypt those parts without using certain level of logic to determine the encryption base and finding the key.

3.2.2.2 Looking for known sites:

Some of the malicious extensions tend to react to specific sites such as social medias/networks like google, *FaceBook* or *YouTube* , e-shopping sites such as *amazon* and *eBay* by asking for permissions to access those sites or trying to figure out if the visited site

is one of those, so it launches its specialized attacks, to prevent that from happening and as security measure, we sat up our first scanned to look for any known sites that mentioned within the code and compare with our data set collection where we have an extendable collection of possible URLs that refers to one of these. Figure3.3 shows the scheme for process.

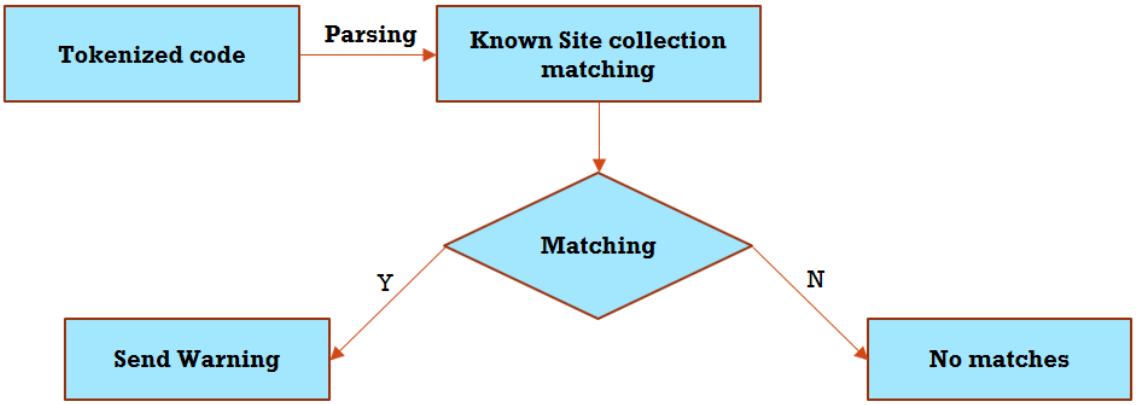

Figure 3.3: known sites detection

Despite requesting access to a known site, we categorized that as a warning and not an actual threat until an ill intention is confirmed, due to the fact that some extension have some sort of functionalities for some of these site without performing any actual malicious activity.

3.2.2.3 Looking for banned sites/servers

Similar to **section 3.2.1,** however, a matching is considered a threat, the attackers use servers with low restriction to hosts their scripts and such in order to launch a cyber-attack or provide attentional scripts to back up their used extensions after being installed, therefor, any of the banned list collections is considered as a threat.

The collection has the ability of expanding according to harvested result from different phases of the scan.

3.2.2.4 Identifying potential malicious features

The pinnacle of our static analysis is the features chosen for scan to identify threats in the extension, we recognized five categories of features that has been used or can be used to ignite any sort of malicious action as the following:

- **Network features:** Attackers use features implemented in JS/Chrome such as webRequest in order to track user's activity by intercepting the various requests fired by the user, listing 3.4 shows the extracted features in this category:

Figure 3.4: network features

- **Permissions and scripts handling:** high permissions or restricted permissions are asked to have privileged access to user's machine or to escape the isolated world implemented by chrome security model, beside trying to run a script that wasn't clarified in the extension. listing 3.5 shows the extracted features in this category:

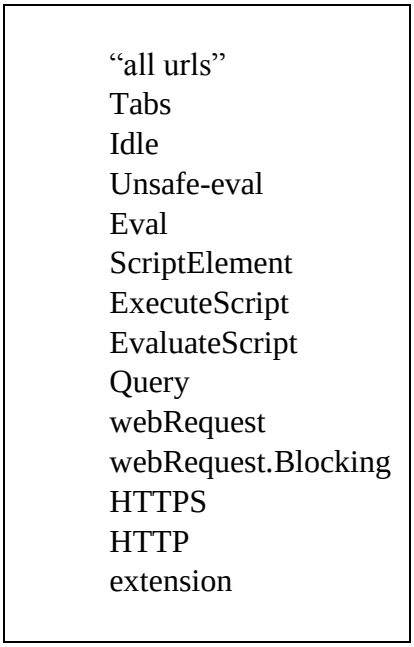

Figure 3.5: permission features

- **Data and Sessions:** another type of features where attackers try to hijack user's sessions, such us social network or emails sessions using features such as *session or cookies,*accessing user storage using features like *unlimitedStorage or browsingData,* listing 3.6 shows the extracted features in this category:

```
BrowsingData
Cookies
Sessions
Storage
unlimitedStorage
```

Figure 3.6: data sessions

- **DOM:** Highly used in ad injecting, by manipulating the content of the displayed web page to the user, where an element of the DOM is replaced using *replaceChild*, removed by *removeChild,* or added to the original content like *appendChild.* However, other DOM features can be used to trigger other kind of malicious actions like locating a specific element of DOM and operating on it, listing 3.7 shows the extracted features in this category:

```
document.foo
Document.createElement
HTMLElement.onload
HTMLScriptElement.src
Window.location
Element.innerHTML
outerHTML
insertAdjacentText
insertAdjacentHTML
insertAdjacentElement
Element.innerText
Element.outerText
Element.removeAttribute("data-description")
Element.getAttribute
```

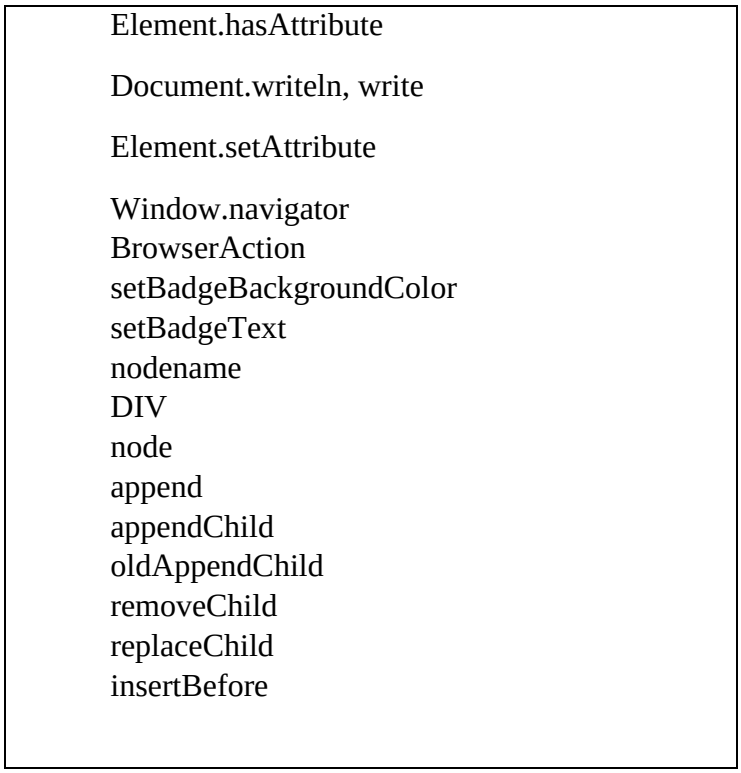

Figure 3.7: DOM features

- **Event listener:** since JS is the language used to develop those extensions, a bunch of event listener can be used to initiate malicious acts, such as the feature *onClick* where it's triggered to run a code whenever the user clicks an element of web. Or *onHover*, which is triggered when the user moves on the mouse of a specific element, listing 3.8 shows the extracted features in this category:

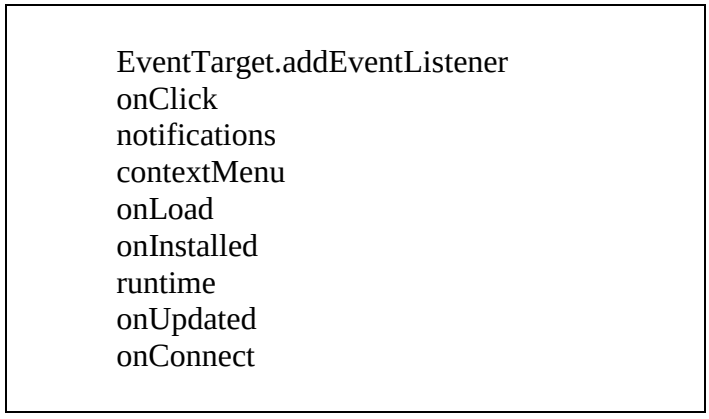

Figure 3.8: Events features

Each of the features explained above has a score system that reflects its threat level based on the sort of action that can be performed through one or a combination of these features, in the interest of helping the case where an alien pattern is found.

3.2.3 Pattern Checking

After running the two types of analysis mentioned in **section 3.2.1** and **section 3.2.2,** we run a logical based scan on the final code to match one or some of the patterns we implemented into our code, if a certain combination of features is present, we can identify the type of attack in the browser's extensions through a logical statement.

We collected most of accessible patterns that we could lay our hands on from malicious extension and expert reviews, figure 3.9 shows an example matching a *youtube*'s phishing pattern.

```
//youtube
let found5 = pattern.find(element => element == "yt-user-info");
let found6 = pattern.find(element => element == "element");
let found7 = pattern.find(element => element == "nodename");
let found8 = pattern.find(element => element == "channelName");
let found9 = pattern.find(element => element == "DIV");
if(found5 && found6 && found7 && found8 && found9){console.log("we found youtube Phishing attack ");
```

Figure 3.9: youtube phishing pattern

3.2.3.1 Potential Pattern Handling

In pursuance of enhancing our patterns identifying technique, we looked for possible patterns that relates to our existing ones, any piece of code that contains one or some of the main pattern elements is considered as potential threat that takes a different shape. thus, process a more specialized analysis towards that type of attacks.

3.2.3.2 Potential Threat handling

The last step if the static analysis triggers when all scanners are done analyzing all potential threats, that may leave us with array containing all the features that could be used to launch a malicious action from the extension code sided by a score, the higher the score is, the higher the risk is.

In case of no pattern or confirmed threat is found, the features found are examined by an expert to discover possible malicious code.

3.2.4 Dynamic Analysis

Once the static analysis is done and no threat was recognized, the dynamic analysis starts, we adopted various techniques to examine, intercept and analyze the extension during execution.

This analysis is spited into two phases, phase 1 where the extension is running, we look for additional files that are called to pass them to our static analyzer if they weren't before, phase 2 where we run different behavioral test to see if there is a noticeable malicious action ran by the extension.

3.2.4.1 Phase 1 : Additional files :

By running the browser's extension, its full code would be executed and during that we may witness the call for an additional file that was hidden in the extension or location or being downloaded through another server, we intercept those files using the build in tool in google chrome to gather all the JavaScript files downloaded or called during execution.

The second static scan initiates if those files exist, and analyzing flow changes upon the result we obtain from feeding those script to the scanner.

3.2.4.2 Phase 2: behavioral test:

Our second phase of dynamic analysis focuses on the output and the behavior of the extension, we learned that some malicious extension uses self-defending mechanism where the malicious code won't execute in some cases, we mention some of them:

- If the environment we are holding the test on is virtual like when the elements of browser are auto generated to fit user browsing routines according to the extension demands, some of malicious extensions run a test to see if it's run on a proper browser due not having the ability to cover all the possible requested elements by the extension.
- If there is third party interceptor (API/web requests) is present that may lead the extension to put its malicious behavior on hold.
- If a virtual network is detected, which in our case the use of virtual machines to contain the extension.
- Cloaking mechanism: the extension can hide its malicious behavior for a period of time after installation to avoid being detected.

- Absence of network connection: in some rare cases, if there is no access to the internet (for security measures), the extension would also hold executing its code, and we wouldn't be able to observe it.

To overcome those challenges, we tried to maintain the least possible of analyzing tools and running the extension on actual machine while making sure that the extension is always running in isolated mode for the safety of the system.

Our main intercepting tool, is the build in google developer tool in chrome, which allowed us to intercept web and network requests, files being loaded/called and downloaded. However, doing this task manually led to dealing with massive amount of data because not only the extensions are being intercepted, by the browser itself is.

To make the analysis more doable, we generated profiles of normal usage of the browser, by surfing some of the known social media, e-shopping sites and such, during the generating, no extension was installed into the browser, and to make sure nothing is affecting the normal scenario of loading those websites or using the browser in the normal case.

Once all profiles are generated and saved, we install our targeted extensions each at a time while performing an actual browsing routine, each profile represent a scenario where the extension is running during surfing one site or firing the browser, those scenarios are saved as well and a comparison will be run after gathering all needed data.

For complex extensions, we kept them running for several days, while generating profile on different period of times to see if there are changes through the multiple profiles of that extension

Running the extension on an actual machine without third party programs offered us a clearer view of the extension behavior overtime which allowed us to learn more about methods used by the attackers

3.3 Classification

After satisfying all conditions from the static and dynamic analysis phases, the results containing whether the extensions are malicious or benign are then parsed to our classifier.

Our model contained initially eighty (80) features, the features were chosen according to their impact on the functionality of the extensions and how often they were used.

In order to enhance the accuracy of our results and the performance of our classification process, a feature-selective algorithm was required to determine which feature has an impact on predicting the correct verdict.

3.3.1 Feature Selection:

Our model contained a massive number of features compared to previous approaches, so reducing their numbers was needed to optimize the analyzing time and to confirm the impact value of each feature.

To do so, we took in consideration three factors, the total number of occurrence for each feature, the number of its occurrence in malicious extensions and benign one to calculate the general average of occurrence of those features.

The idea revolved around finding the features that would make an extensions distinguishable, meaning if the average obtained is higher than zero, that means the feature appeared more on one of the two types of extensions we have (malicious and benign). The algorithm and mathematical form we used are illustrated in **equation 3.1** and **figure 3.10.**

We refer to the average by **AVG**, it only takes positive values, the closer it gets to 0 the less impact it has on the classification

numberOfOccurenceM refers to the occurrence of the feature in malicious extensions.
numberOfOccurenceB refers to the benign one
numberOfOccurenceT refers to the sum of both.

$$AVG = \frac{|numberOfOccurenceM - numberOfOccurenceB|}{numberOfOccurenceT}$$

Equation 3.1: Calculating Occurrence Average

```
Algorithm:feature subset selection

While(true)
{
    nM(fi)= calculates of occurence in malicious extensions of the feature fi
    nB(fi)= calculates of occurence in benign extensions of the feature fi;
    nT= nB(fi)+nM;
    Avg= Calculates the positive average;

    AccuOG= invokes classifier and calculate accuracy;

    nF= features set in the model;
    tempNF= nF;

        for i=0 to nF
        {
            remove(min) from tempNF;
            currentAccu= invokes classifier and calculates accuracy of tempNF;

            if(currentAcci>AccuOG)
            {   nf=tempNF;
                write("new set:" nF);
            }
        }
}
```

Figure 3.10: feature subset selection algorithm

3.3.2 Classifier Selection.

In order to demonstrate the effectiveness of our detector we experiment our data set with four machine learning algorithms: Support Vector Machine (SVM), Bayesian Network (BN), Logistic Regression (LR), and Multilayer Perceptron (MLP). From our dataset we extracted 75% of it as training set and the remaining 25% as test set to evaluate the four models. By comparing and analyzing the results of the models, we can choose the optimal classifier as our malicious extension detection classifier.

3.3.2.1 Support Victor Machine (SVM) Classifier

SVM [20] is a support-vector machine that construct a set of multi-dimensional hyperplanes that is used in regression and classifications.

"a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class (so-called functional margin), since in general the larger the margin, the lower the generalization error *of the classifier" [20]*

The SVM can solve both linear and non-linear problems.to do so, the classifier requires finding a hyperplane between two classes by taking the input and output line in order to separate the two classes.

Points that closest to the line between both classes are identified as support vectors, the distance between the support vectors and the line is known as the margin. The goal is to find the hyperplane in which the margin reaches its peak.

If the data isn't separated linearly, the equation of the circle is being used to make linear separator higher in dimension which will allow classifying data then projecting decision back to the original dimensions through mathematical transformation

3.3.2.2 Naïve Bayes Classifier

Naïve Bayes [21] is a model for the probabilistic machine learning which used for classifications. It's based on the theorem in **equation 3.2.**

$$p(A\backslash B) = \frac{\mathrm{p(B\backslash A)p(A)}}{\mathrm{p(B)}}$$

Equation 3.2: Naïve Bayes formula

We can find the probability of *A* happening, given that *B* has occurred. *B* plays the role of the evidence while *A* represents the hypothesis. We assume that the features are independent and the existence of a single feature doesn't not affect the other which led to the naming ***Naïve.***

This algorithm [22] is basically used for cases such as:

- **Real time prediction:** since the Naïve Bayes algorithm is fast, it gave it the capacity to perform real-time predictions
- **Multi-class prediction:** it is possible to predict the posterior probability for several classes of targeted variables
- **Text, spam and sentiment analysis:** it's the most used classifier in the mentioned cases because of the refined multi-class prediction ratio.
- **Recommendation Systems:** mostly used machine learning and data science to do filtering of unseen data and make predictions

3.3.2.3 Logistic Regression (LR)

Logistic Regression [23] mainly used in social sciences and medical related fields in order to perform predictions in the mentioned fields. It handles cases where the observed outcome is a dependent variable that can only be either 0 or 1.

It measures the relationship between the categorical dependent variable and one or a set of independent ones though estimating probabilities *P*. **equation 3.3** illustrates the formula of the logistic regression function.

$$Logit\ p = ln.\frac{p}{1-p} \quad for\ 0 < p < 1$$

Equation 3.3: logistic regression

3.3.2.4 Multilayer Perceptron (MLP)

MLP [24] is one of the artificial neural networks, it has 3 later: the input, the hidden and the output layer. Both the hidden and the output layers are neurons. The MLP uses a supervised learning model that's called backpropagation as training.

Mainly the MLP is used to solve research problems which could offer a high accurate solution for high complicated problems, it can create regression analysis for classification if the response variable is categorical. In other words, it makes it perfect for classifying.

3.4 Benchmarking

As we finish all the mentioned processes that were introduced in this chapter, we wanted to gather every split of information gathered from the process. Usually benchmarking tests are used to test the performance, but for our case we wanted to review the changes that occurred during the tests.

Those variations in the browser will serve later to extract more features that may help us understand malicious extension functioning. The results are later saved in on our database collections.

3.5 Detection Criteria

When it comes to detecting malicious extension content or behavior, several aspects have to be taken into consideration such as:

- **Looking for included malicious code:** malicious features or patterns.
- **Monitoring performed processes:** overview the processes performed during the examination time.
- **Monitoring calls for system APIs**: checking which APIs were/are called during execution to determine restricted action or what sort of behavior the extension willing to perform
- **Looking for requests to access to privileged permissions:** mainly found within the extension manifest, where it asks for permissions such as storage. In which we compare if the requested permissions match the claimed functionality
- **Looking for local Scripts:** within the extension code, an additional script file could be called during the execution.
- **Looking for external Script:** an external script file could be called or downloaded later by the extension
- **Performing different environmental tests(cloaking):** some extensions tend to execute different while being intercepted, others delays their malicious behavior and others detects whether they're running on a virtual environment (virtual machine or virtual browser).
- **Network and Web requests:** in most cases malicious extensions don't rely on what it's hard coded into the extension, more often they request access to servers where malicious functions are store or redirect user to bad links.

While considering those criterial factors, we made a comparison to see what aspect we could covered through our approach compared to the currently existent ones mentioned from **section 2.3** to **section 2.7**

Table 3.1 shows which analyzing areas were covered by our approach and existent approaches:

Table 3.1: approach comparison

Approaches	Hulk	WebEval	Model Based approach	LMED	Combined approach	**MWBED**
Looking for included malicious code		✓	✓	✓	✓	✓
Monitoring processes						✓
Monitoring APIs Calls						✓
Checking permissions		✓			✓	✓
Looking for local Scripts		✓			✓	✓
Looking for external Script				✓	✓	✓
Anti-cloaking						✓
Network/web requests	✓					✓

Regardless of serving the same purpose, the techniques used by other approaches were more theme focused when it comes to detection malicious extension. From our perspective, we wanted to deal with the browser malicious extensions same way as other type malwares which made us adapt other analyzing technique that other approaches aren't much familiar with.

3.6 Conclusion

In this chapter we presented our Malicious Extension Detector mechanism and explained in detail the followed steps to implement and evaluate our detector in the next chapter.

Chapter 4

Implementation and Experimentation

4.1 Overview

This chapter contains the implementation and experimentation of our approach which is explained in the previous chapter. In this process, we have a web browser (chrome), a Mango data base that had 4 different collections, a static analyzer, a dynamic analyzing tools that serves multiple purpose according to need, google dev-tool, a process monitor and API monitor. A feature subsect selector, a supervised classifier based on four algorithm of data mining tool which in our case was WEKA and a benchmarking tool to review the alterations in the browser. We choose to run MED on a nodeJS server for future plans to run our detector on multi-platform while having the flexibility to do so.

We kept the ratio of examined extensions around 50% for both malicious and benign ones while taking in consideration only the extensions that were relatively reach with features, most of benign extensions were picked from top chrome store extension. While the malicious one were a mix of experimental ones and others that still on the chrome store.

A total of 66 extensions were included in our model, few of the malicious ones were taking down from chrome store during the study period. Most of the malicious extension were targeting user data of well-known sites such as Youtube, facebook, google and twitter.

4.2 Experimentation setup

We run all our tests on a custom built computer while disabling all sort of processes that has may affect the performance one way or another.

- CPU: intel i7 4790k @4.0Ghz
- RAM: 16 GB @2400Hz
- Operating System: Microsoft Windows 7 Pro 64 bit
- API monitor: API Mon
- Process monitor: Process Hacker and Process Monitor
- Built in monitoring tool: google dev tool
- Data mining tool: Weka 3.8
- IDE: visual studio code x64-1.35.1
- Web browser: Google Chrome
- Server environment: NodeJS

4.3 Implementation tools

In this section we'll mention the tools used in the experiment and the reasons behind our choices.

4.3.1 Google Chrome Browser:

We choose google Chrome browser for our study because of two main reasons, the fist reason was because chrome is most used web browser currently according to the chart in figure 4.1 dominating most of market shares, and the second reason is the fact the chrome is open sourced friendly, meaning we could usually manipulate the extensions within and access their codes when needed.

Top 10 Web Browsers		
1	Chrome 74	28.10%
2	Safari 12	9.90%
3	Chrome 73	7.72%
4	UC 12	4.29%
5	IE 11	2.60%
6	Samsung 9	2.35%
7	Chrome 63	2.35%
8	Firefox 66	2.33%
9	Chrome 72	1.93%
10	Chrome 71	1.68%

Figure 4.1: web browser ranking according to w3counter global stats [25]

4.3.2 Weka

Weka [26] is a collective machine learning algorithms for data mining. It provides tools for classification, regression, clustering and even visualizing data.

Reason why we picked Weka is mostly because of its reliability in previous studies and open sourced license provided by Weka

4.3.3 API mon

API Mon [27] is a free software that offers functionalities to view and control API calls. It provides displayed summary about those called API by referring the source and ID of API callers.

Knowing which API and who made the call helps viewing the extensions functionality within our testing machine. Figure 4.2 shows a sample of monitoring API initiated by Chrome's DLLs.

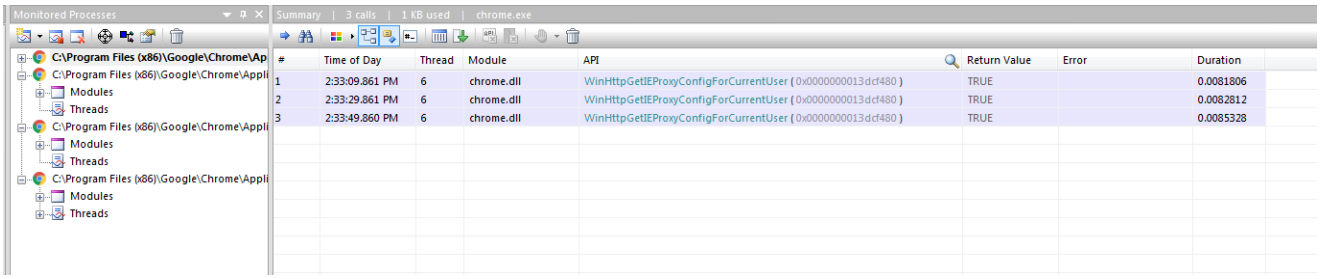

Figure 4.2: API Mon monitoring on Chrome

4.3.4 Process monitor

Process monitor [28] is one of Microsoft external tools to view process, acuities and registry in real-time which includes filters and some details about the process initiator. Also, for our case, this tool was used to capture momentum scenario during the execution of some extensions. Figure 4.3 shows a sample of process monitor being in use.

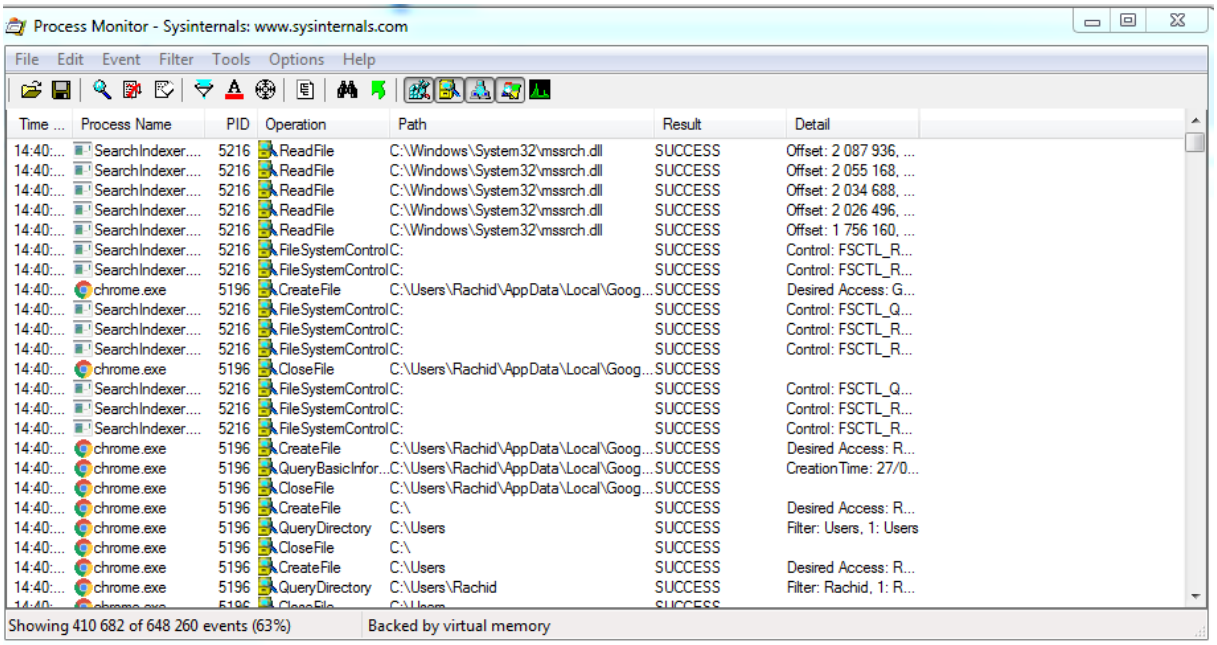

Figure 4.3: Process monitor use sample

4.3.5 Process Hacker

Process Hacker [29] is another free multi-purpose tool to run some tasks on process. It offers the following:

- detailed overview of system activity with real time highlighting.
- Tracking the users of the different files.
- Graphical tracker for resource hogs and runaway processes.
- Checks process network activity and shut it down if needed
- Information about disc access
- Edit and control services

The tool was used to track and alter chrome extensions activities. Figure 4.4 and Figure 4.5 shows a sample of the tool being in use.

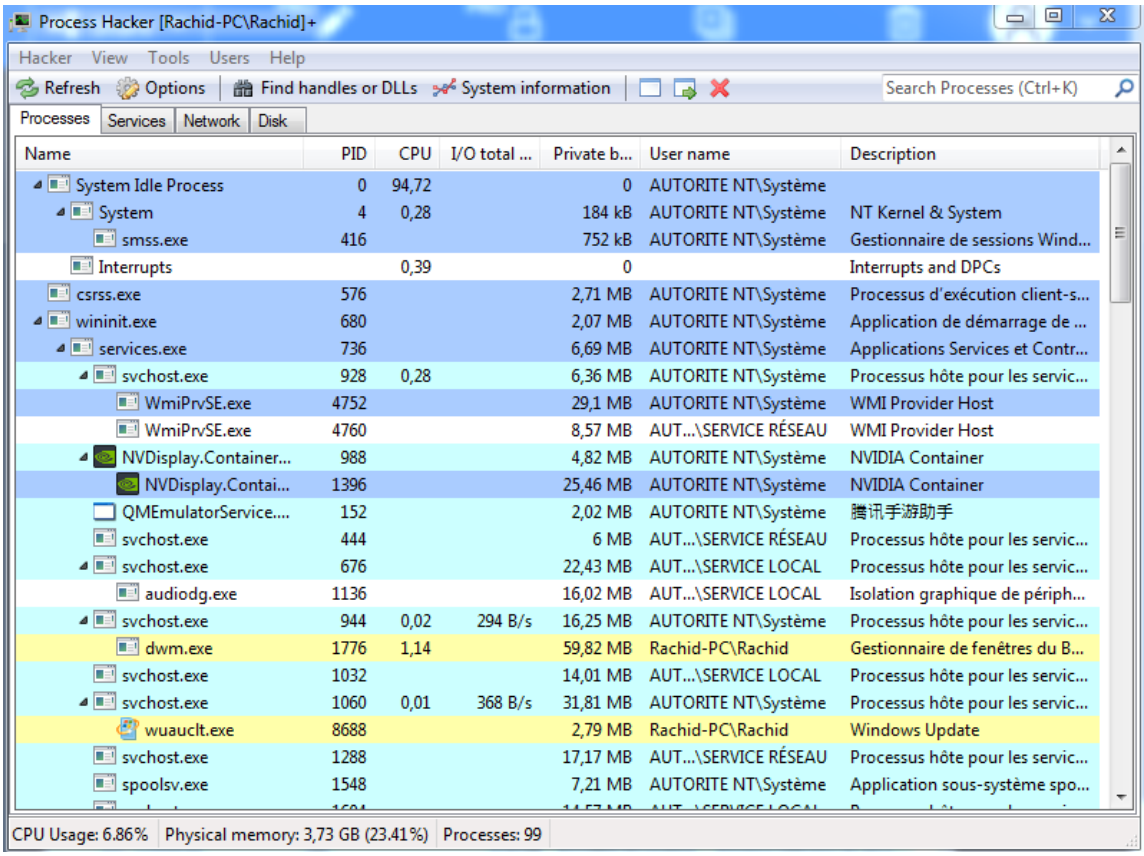

Figure 4.4: process hacker

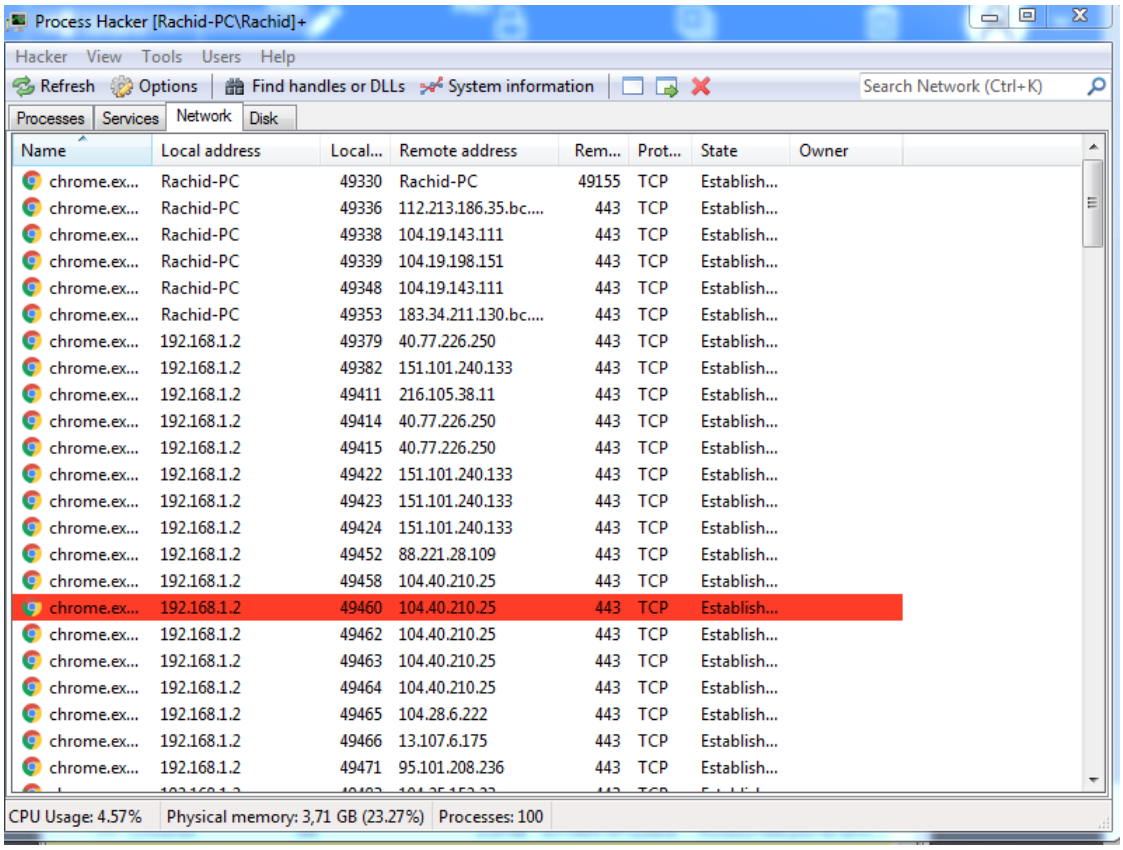

Figure 4.5: Process hacker real-time overview on chrome

4.3.6 Chrome's dev tool

Chrome's dev-tool [30] is one of the built in tools that chrome browser has. It was lately introduced on the chrome browser as a developer tool, after it was only used on chromium (the open sourced chrome project)

The tools offer, network, view-points (XHR request, JS calls, all time of files loaded ...etc.) while surfing the web without disturbing the functionality of extensions nor the visited sites during execution. It also provides time stamps to see what happen at any given moment.

The tool was used mainly to record the different scenario of network activity during the use of extensions and during the off-use for scenario comparison and to analyze extension behavior without a third party monitor. Figure 4.6 and Figure 4.7 demonstrate the dev tool being in use while surfing the web and extensions being run.

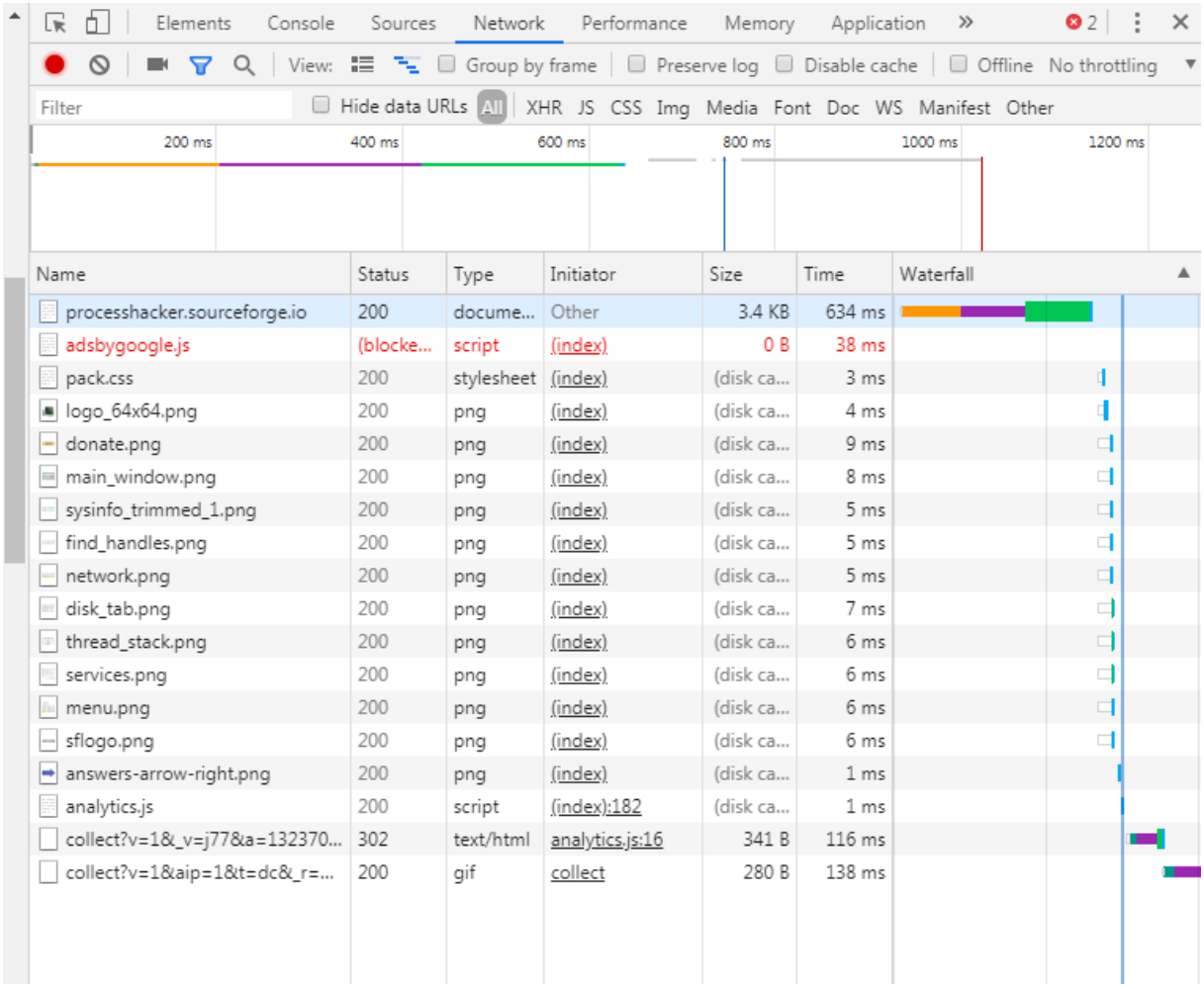

Figure 4.6: real-time viewing of network activity within chrome

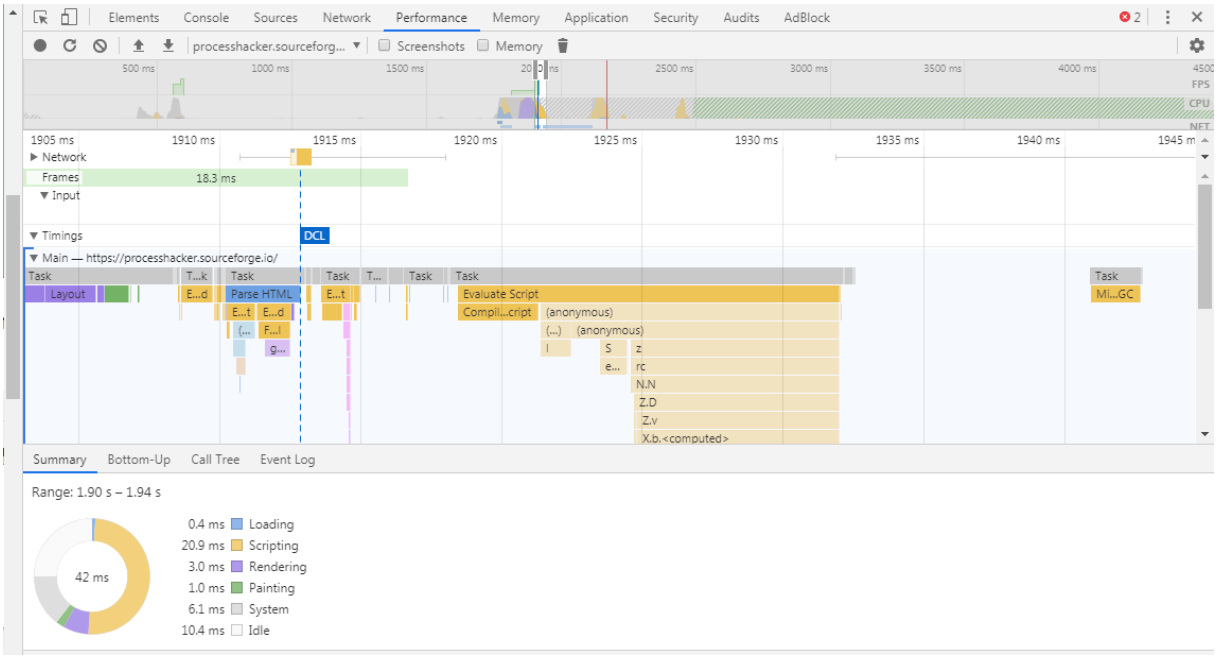

Figure 4.7: a recording (scenario) of surfing web while using a chrome extension

4.4 Analysis methodology

In persuasion to achieve to detect the malicious extensions, we ran several layered analyzing techniques. At each layer we consider what the extension is capable of doing and what kind of information we can extract from the analysis. Each technique is meant to extract every piece of data possible to obtain regardless of the mechanism used by the malicious extension.

4.4.1 Implementation of static analyzer

The static analyzer was used compare the data that was initially gathered in our database to compare it with the extracted codes/files of the examined extensions. The analyze compared data from database and extension to find matching of:

- Extension feature
- Call for known site
- Call for banned sites/servers
- Stored Attack patterns

In addition, it provides a score that is calculated depending on what features were found. And provides suggestions if one or some of the main attack patterns element were found

but no confirmed verdict was found to hint for an in-depth analysis. Figure 4.8 and figure 4.9 shows sample code for feature fetching and finding attack patterns.

```
Signature.find({}, function(err, signatures) {
  signatures.forEach(signature => {
    //   console.log("firstmsg this is for sinature : "+signature);
    myobj.forEach(element => {

      if (element.value === signature.sinature) {
        sum += signature.cof;

        pattern.push(signature.sinature);
      }
    });
  });
```

Figure 4.8: fetching extension's that matches the malicious features collection

```
if (found5 && found6 && found7 && found8 && found9) {
  console.log("we found youtube Phishing attack ");
  count++;
}else if(found5||found8){
  console.log("Potentional pattern to youtube phishing")
}
```

Figure 4.9: sample of matching a YouTube phishing attack pattern/potential pattern.

4.4.2 Performing Dynamic Analysis

In order to examine the behaviors of the extensions, we put the extension under different conditions (modes):

4.4.2.1 Fully supervised mode

Basically, we run the chrome extensions while monitoring the processes, network activity and tracking all API calls. Most of the extensions ran normally. This mode helped determining whether the extension is being executed within the chrome's isolated world or not, whether the extension affects the testing machine or not and track its network activity within the browser.

However, some didn't show any sort of malicious behavior despite the fact that they are, which led to conclude that those kind of extensions uses cloaking (self-defensive mechanism when being intercepted) which meant another method was required. Process Monitor, API mon and process hacker were used on this mode

4.4.2.2 API monitored mode

This is a thinner mode where processes weren't supervised by any third party tool, only API mon was used to track API calls during launch, this mainly was targeting extensions that might react differently while being intercepted or delay the malicious behavior for defined time period.

4.4.2.3 No third party mode

This is the thinnest layer we obtained; however not using any sort of a third party interceptor or monitor but the google dev-tool provided a clarified view on the functionality of the extensions and their workflow, as long as the extension doesn't have access out of chrome's browser. Google dev-tool plays the role of both the monitor and recorder during the execution of our extensions. it was also used to save the different profiles mentioned in **section 3.2.3.2**

4.5 Feature selection results:

After using our suggested algorithm, we found 10 features that had no impact upon the results of the accuracy. Table 4.1 shows the original features before the selection and Table 4.2 shows the excluded features after the selection phase. Although no significant improvement was noticed, just time based enhancement, However, they're kept in a different dataset collection for future analysis due to the ongoing changes of the extensions

Table 4.1: original feature set

Feature	weight
All	0.121
setTimeout	0.47
webRequest	0.53
postMessage	0.045
chrome	0.758
onBeforeRequest	0.106
webRequestBlocking	0.515
XML	0.197
HttpRequest	0.01
onReadyStateChange	0.242

open	0.303
send	0.197
statusText	0.167
ResponseText	0.152
webNavigation	0.227
connect	0.121
getURL	0.197
HTTPS	0.53
HTTP	0.409
URLS	0.106
URL	0.561
Tabs	0.606
Idle	0.091
unsafe-eval	0.53
Eval	0.136
ScriptElement	0.01
ExecuteScript	0.258
EvaluateScipt	0.01
Query	0.288
extension	0.288
BrowsingData	0.015
Cookies	0.53
Sessions	0.01
Storage	0.576
unlimitedStorage	0.333
document	0.652
foo	0.091
createElement	0.621
Element	0.273
onLoad	0.197

src	0.53
HTML	0.394
location	0.318
innerHTML	0.333
outerHTML	0.01
insertAdjacentText	0.01
insertAdjacentHTML	0.01
insertAdjacentElement	0.01
innerText	0.197
outerText	0.01
removerAttribute	0.01
getAttribute	0.333
hasAttribute	0.03
write	0.03
writeln	0.01
setAttribute	0.394
navigator	0.167
window	0.485
BrowserAction	0.409
setBadgeBackgroundColor	0.167
setBadgeText	0.121
nodename	0.242
DIV	0.348
EventTarget	0.01
addEventListener	0.561
onClick	0.318
notifications	0.152
contextMenu	0.167
onInstalled	0.318
runtime	0.53

onUpdated	0.167
onConnect	0.03
node	0.182
append	0.258
appendChild	0.485
oldAppendChild	0.01
removeChild	0.242
replaceChild	0.136
insertBefore	0.394
all_urls	0.227

Table4.3: neglected features.

HttpRequest
EvaluateScipt
Sessions
outerHTML
insertAdjacentText
insertAdjacentHTML
outerText
writeln
EventTarget
oldAppendChild

4.6 Experimentation

In this section, we show performance of our ME detection framework, exactly by testing our web browser extensions classifier modules with big amount of data.

4.6.1 Dataset generation:

As we mentioned in the previous chapter, a new dataset was collected to train and test our own classifiers.

We offered the Weka tool our analyzing result of 66 extensions while marinating a near average of 50% of each type (benign and malicious extensions), for the sake of making sure that the features we choose could make malicious extensions distinguishable from benign ones without relying on mere luck through an unbalanced training model.

4.6.2 Experimental Results

To show the performance of our ME detection framework, we use instances of two types of web browser extensions: benign extensions and malicious extension.

Training dataset is divided into 75% of the entire dataset while the remaining 25% used for testing classifiers.

We obtained an accuracy of 98.48% with 1.51% error ratio using Random Tree. Figure 4.10 shows the result obtained by Weka. While using Naïve Bayes, MLP, LR and SVM reached 100% shown in figure 4.11, figure 4.12, figure 4.13 and figure 4.14 respectively.

We conclude that Random Tree works better with a big dataset, while the others reached a perfect score regardless complexity of our feature set.

```
Time taken to build model: 0.04 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          65               98.4848 %
Incorrectly Classified Instances         1                1.5152 %
Kappa statistic                          0.9695
Mean absolute error                      0.0152
Root mean squared error                  0.1231
Relative absolute error                  3.0531 %
Root relative squared error             24.7077 %
Total Number of Instances               66

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               1,000    0,028    0,968      1,000    0,984      0,970    0,986     0,968     0
               0,972    0,000    1,000      0,972    0,986      0,970    0,986     0,987     1
Weighted Avg.  0,985    0,013    0,985      0,985    0,985      0,970    0,986     0,978

=== Confusion Matrix ===

  a  b   <-- classified as
 30  0 |  a = 0
  1 35 |  b = 1
```

Figure 4.10: Weka's Random Tree Results

```
Time taken to build model: 0.03 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          66               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               66

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     0
               1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     1
Weighted Avg.  1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000

=== Confusion Matrix ===

  a  b   <-- classified as
 30  0 |  a = 0
  0 36 |  b = 1
```

Figure 4.11: Weka's Naïve Bayes Results

```
Time taken to build model: 1.58 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          66               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0.0035
Root mean squared error                  0.0056
Relative absolute error                  0.7041 %
Root relative squared error              1.1176 %
Total Number of Instances               66

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     0
               1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     1
Weighted Avg.  1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000

=== Confusion Matrix ===

  a  b   <-- classified as
 30  0 |  a = 0
  0 36 |  b = 1
```

Figure 4.12: Weka's MLP Results

```
Time taken to build model: 0.09 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          66               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               66

=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     0
               1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     1
Weighted Avg.  1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000

=== Confusion Matrix ===

  a  b   <-- classified as
 30  0 |  a = 0
  0 36 |  b = 1
```

Figure 4.13: Weka's LR Results

```
Time taken to build model: 0.04 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances          66               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               66

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     0
                 1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000     1
Weighted Avg.    1,000    0,000    1,000      1,000    1,000      1,000    1,000     1,000

=== Confusion Matrix ===

  a  b   <-- classified as
 30  0 |  a = 0
  0 36 |  b = 1
```

Figure 4.14: Weka's SVM Results

4.6.3 Evaluation

After obtaining the result sheet, we need to highlight three of the parameters which are Accuracy, precision and recall. Those three and identified and calculated according to the following: [31]

Accuracy:

The accuracy mainly reflects the percentage of the correct answers out of the overall answer to give a description about of the systematic errors

Precision:

It represents the relevant instances out of all retrieved ones. And it's calculated as the following in **equation 4.1.**

$$precision = \frac{\{relevant\ instances\} \cap \{retrieved\ instances\}}{\{retrieved\ instances\}}$$

Equation 4.1: precision formula

Recall:

Represents the relevant instances retrieved out of all relevant ones that exist. And it's calculated as the following in **equation 4.2**

$$Recall = \frac{\{relevant\ instances\} \cap \{retrieved\ instances\}}{\{relevant\ \ instances\}}$$

Equation 4.2: recall formula

In the following table 4.3 we mention the scores obtained of the accuracy, recall and precision of our approach. Using five (5) different classifiers which the Random Tree, Naïve Bayes, SVM, MLP and LR.

Table 4.3: Accuracy/Precision/Recall results of our model

Model	Accuracy	Precision	Recall
Random Tree	98.48%	96.8%	100%
Naïve Bayes	100%	100%	100%
SVM	100%	100%	100%
MLP	100%	100%	100%
LR	100%	100%	100%

4.6.4 Comparison

The following charts in figure 4.15, figure 4.16 and figure 4.17 shows comparison between our approach and the closest one in terms of methodology (combination of static and dynamic analysis from **section 2.7**) using the common classifiers between the two.

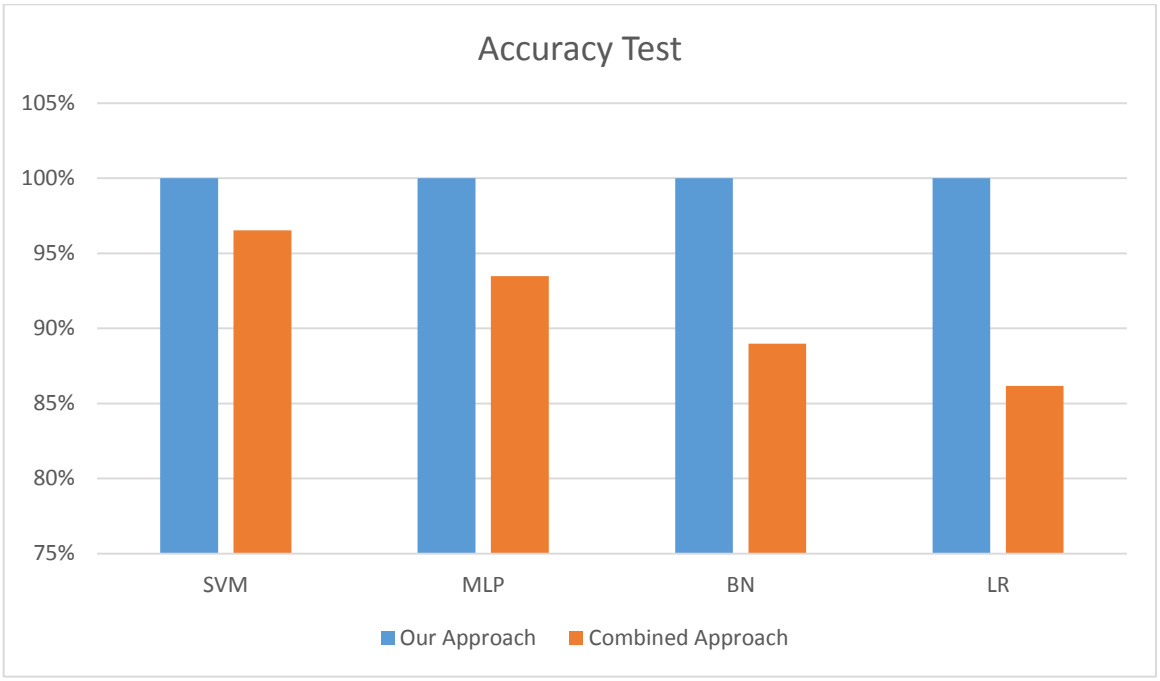

Figure 4.15: Accuracy comparison

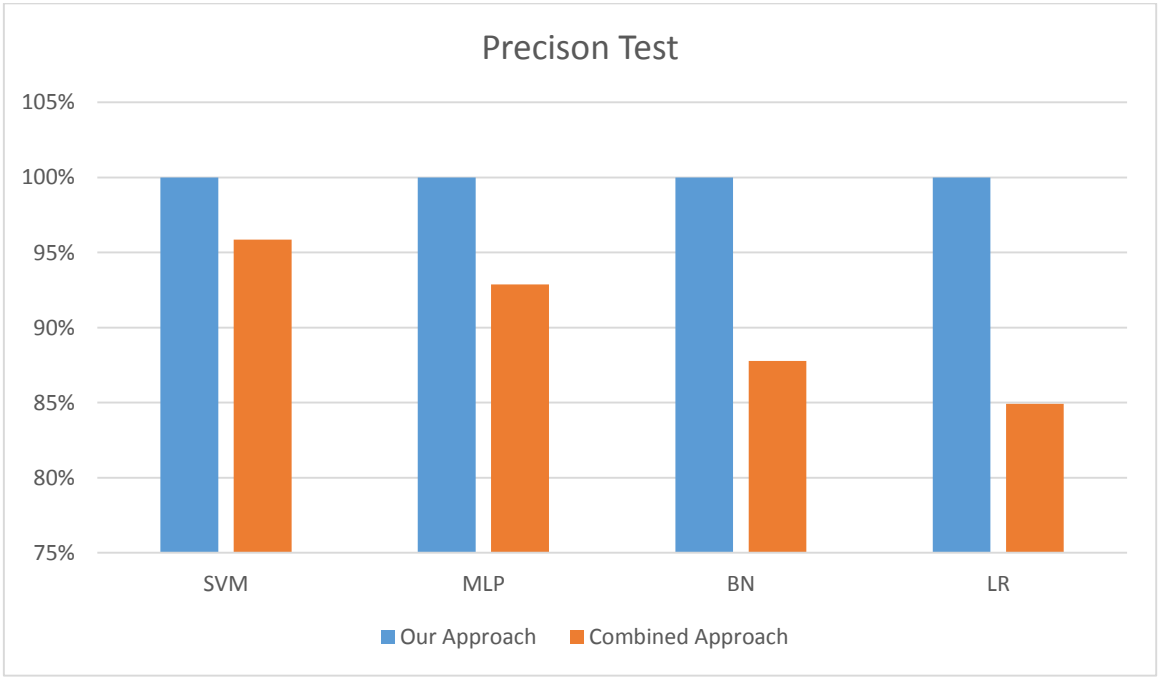

Figure 4.16: Precision Comparison

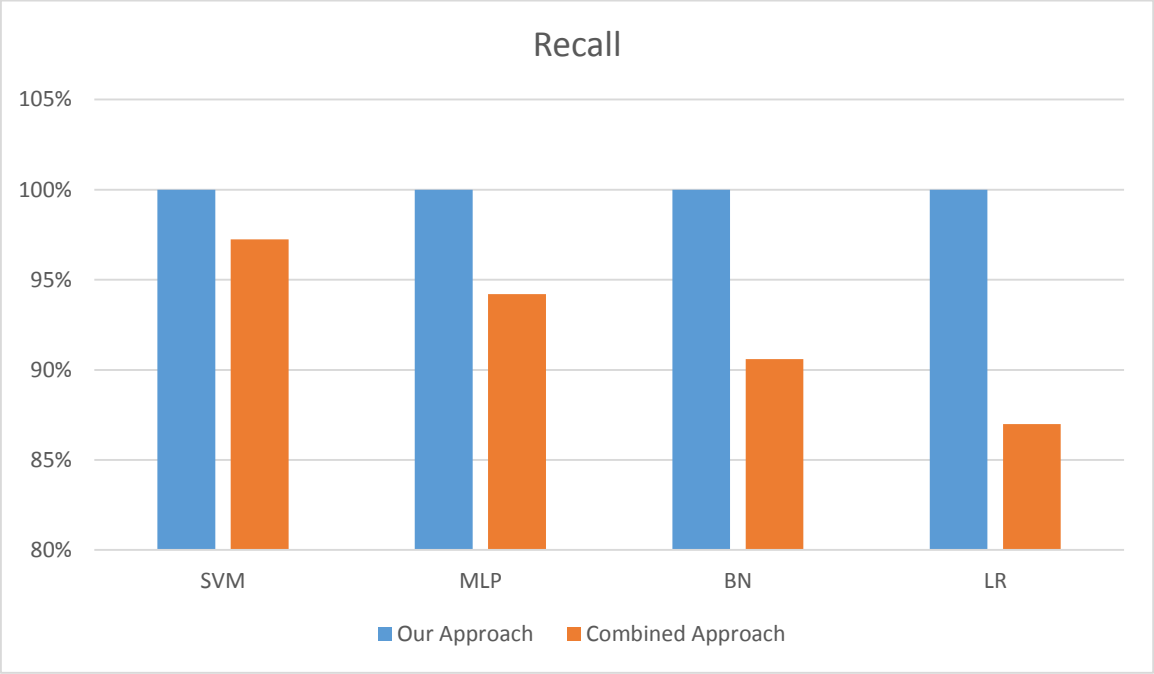

Figure 4.17: Recall Comparison

4.7 Result Discussion

Through our model, we maintained perfect score on 4 out of the five (5) classifiers used for the evaluation. The high accuracy, precision and recall that we scored was due to our web extension analysis step. to extract relevant features, a wide range of relevant features is used to classify the extensions, the possible combination of features helped improving those results.

Having the virtue of flexibility and scalability in our feature set allows to go neck to neck with evolving of malicious extensions

4.8 Limitations

We ought to say that our major limitations were technical due to the massive data loads we had to statically examine in which our tool build upon a NodeJS server framework (which is still an evolving technology) couldn't handle parsing all data once (at least for the current moment with affordable tools). This was caused because of nowadays extensions became more complex compared to the early days and the amount of element and code included within rivals the other type of applications.

As temporary solution for the complex extensions, we implemented fragmentizing methods in order to handle identical parts of the extensions to parse it to the static analyzer.

4.9 Conclusion

In this chapter we've shown the implementation and experimentation of our approach for web browsers malicious extensions detection, our approach was based upon obtained data from actual extensions analysis that initially weren't classified as malicious or had unknown threat.

The experimental study clearly shows the performance of our framework in detection of malicious extension, our classifiers prove their performance with an accuracy of 100%.

General Conclusion

GENERAL CONCLUSION

Malicious Browser Extensions spread out widely and quickly invading thousands of browsers putting users' data at high risks by making them vulnerable to all sort of malwares and spywares. In our project we demonstrated that existent approaches to detect malicious browser extensions are still incomplete and biased by limited perspectives. Therefore, the need to breakthrough traditional detection techniques was required. Thus, led us to provide an enhanced hybridization of static and dynamic analyzing techniques that equals the malicious extensions threat at every aspect by taking environmental and external variables into consideration while dealing with those extensions.

Our approach contributed through maintaining the following critical points:

- An extended dictionary of malicious features that matches nowadays extensions
- An attack detector that has the flexibility to find patterns or sub-patterns to suggest in-depth analysis for alien patterns.
- Detecting all sort API calls that happen within and out of the chrome browser
- Tracking processes triggered by chrome extension
- Detect whether the extension is running within the chrome isolated world or not
- Generating behavior profiles of extensions while surfing the web in real-time through different scenarios
- Overcoming self-defensive mechanism such as cloaking
- Obtained a balanced machine learning model to classify extensions into benign and malicious that is near perfect.

In this work, we collected a new dataset that includes modern type of malicious extension, which weren't used in previous researches.

During our experimentation, we focused on real-life threats that still exist in Chrome's extension that some of them still exist in the Chrome Store, we performed our analysis over real malicious and benign extensions that either were considered benign at first or malicious without known threat. Our work focused running legit test on a legit environment to witness the actual functionalities of the examined extensions for the sake of defying fake behavioral reactions as explained through this document.

Our approach reached almost perfect scores at classifying extensions by examining existent malicious ones and benign ones from top ranked Chrome Store's extensions.

Perspectives

Our approach focuses on the unstudied aspects of detection malicious extensions, which was fundamentally great according to the obtained results. However, it could technically be improved to enhance fluidity during dynamic analysis by getting access to larger number of APIs that could help us out perform our desired tasks (mostly because of the lack of open sourced APIs), Statically by founding a new tool that can handle mass data during testing relatively big extension or load of extension files and finally we propose to give the user the possibility to detect vulnerable web extension (benign but can be exploited by malicious users) in our detection tools.

References

References

Articles:

[2] Alan Grosskurth and Michael W. Godfrey, A Reference Architecture for Web Browsers, ICSM, 2005.

[10] Lei'Liu, Xinwen'Zhang, Guanhua'Yan, Songqing'Chen, Chrome' Extensions:Threat'Analysis and Countermeasures, 2012.

[14] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab and Kurt Thomas, Trends and Lessons from Three Years Fighting Malicious Extensions, at google. 12th of August 2015.

[15] Hossain Shahriar, Komminist Weldemariam, Mohammad Zulkernine and Thibaud Lutellier, Effective detection of vulnerable and malicious browser extensions, Computers & security. volume 47, November 2014.

[16] Gaurav Varshney, Manoj Misra and Pradeep K. Atrey, Detecting Spying and Fraud Browser Extensions, Multimedia Privacy and Security, Pages 45-52, 30th of October 2017.

[17] Yao Wang, Wandong Cai,1 Pin Lyu, and Wei Shao, A Combined Static and Dynamic Analysis Approach to Detect Malicious Browser Extensions, Security and Communication Networks. Volume 2018, May 2018

Thesis:

[4] Nicolas Golubovic, Attacking Browser Extension, RUHR-UNIVERSITAT BOCHUM, 3rd of May 2016.

[13] Michael Cypher, Intercepting Suspicious Chrome Extension Actions, Imperial college Longdon, 5th of July 2017.

Web sites:

[1] How browsers work, https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#Introduction, visited on February 2018

[3] Web Browser Architecture, https://fr.slideshare.net/quangntta/web-browser-architecture-49196378 visited on February 2018

[5] Chromium project, https://dev.chromium.org/, visited on February 2018

[6] Chrome security model, https://www.chromium.org/chromium-os/chromiumos-design-docs/security-overview, visited on February 2018

[7] Rendering engine, https://lunascape.zendesk.com/hc/en-us/articles/215401928-What-is-a-rendering-engine-, visited on February 2018

[8] Webkit, https://www.paulirish.com/2013/webkit-for-developers/, visited on February 2018

[9] Chrome extensions, https://developer.chrome.com/extensions, visited on February 2018

[11] Safe browsing, https://transparencyreport.google.com/safe-browsing/overview?hl=en, visited on May 2019

[12]Hulk: eliciting malicious behavior in browser extensions, https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kapravelos, visited on April 2019

[18] TotalVirus, https://www.virustotal.com, visited on 18th March 2019

[19] Random tree, http://desktop.arcgis.com/fr/arcmap/latest/tools/spatial-analyst-toolbox/train-random-trees-classifier.htm, visited on 28th of June 2019

[20] NB wiki, https://en.wikipedia.org/wiki/Naive_Bayes_classifier, visited on 28th of June 2019

[21**]** NB,https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c, visited on 28th of June 2019

[22] SVM, https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989, visited on 28th of June 2019

[23] https://en.wikipedia.org/wiki/Logistic_regression, visited on 28th of June 2019

[24] MLP, https://en.wikipedia.org/wiki/Multilayer_perceptron, visited on 28th of June 2019

[25] Global stats https://www.w3counter.com/globalstats.php visited on June 2019

[26] Weka, https://www.cs.waikato.ac.nz/ml/weka/ visited on June 2019

[27] API mon, http://www.rohitab.com/apimonitor, visited on 10th of April 2019

[28] Process monitor, https://docs.microsoft.com/en-us/sysinternals/downloads/procmon, visited on 26th of March 2019

[29] Process hacker, https://processhacker.sourceforge.io/, visited on 15th of May 2019

[30] Google dev tool, https://developers.google.com/web/tools/chrome-devtools/, visited on 18th March 2019

Abstract

Malicious browser extensions raised a global threat towards web users, their tremendous spreading made internauts vulnerable to all sort of attacks that could be performed those extensions. multiple approaches and techniques were used by security experts to prevent and detect those ill extensions. In this report we propose a hybridization approach of static and dynamic techniques, geared with a machine learning model. The approach focuses on retrieving relevant malicious features, matching malicious pattern and defining new ones through examining the extensions behaviors in real-time on a legit environment with multi factors that work as a trigger to witness the various behaviors possible performed by the extension on the spot light. For our training model, we examined some of the top chrome store extensions, group of malicious extensions discovered by experts but mainly extensions that weren't studied previously, which were detected by us later. The validation test reached 100% accuracy on several classifiers.

Keywords: Web, Browser, Extension, Malicious extension, Detector.

ملخص

تمديدات المتصفح الخبيثة تسببت بتهديد عالمي لمستخدمي الويب, إنتشارهم الهائل جعل مستخدمي الإنترنت عرضة لجميع أنواع الهجمات التي يمكن تنفيذها بواسطة هاته التمديدات. عدة مقاربات و تقنيات تم استخدامها من طرف خبراء الأمن لكي يكشفوا و يوقفوا هاته التمديدات الخبيثة. في هذه المذكرة نطرح مقاربة لاكتشاف تمديدات الواب الخبيثة تعتمد على تهجين للتقنيات الساكنة و الديناميكية, مجهزة بنموذج تعلم الآلة. هذا المنهج يركز على إيجاد خصائص خبيثة ذات صلة, مطابقة النمط الخبيث و تحديد أنماط جديدة خلال فحص سلوكات التمديدات في وقت حقيقي في بيئة شرعية مع عدة عوامل تعمل كمحرض لمشاهدة عدة سلوكات مختلفة ممكنة تم تنفذيها بالتمديدة التي تم تسليط الضوء عليها. من أجل نموذجنا الخاص بالتدريب, قمنا بفحص بعض التمديدات ذات مراتب أولى في متجر كروم للتمديدات, مجموعة من التمديدات الخبيثة التي تم الكشف عنها من طرف خبراء لكن اعتمدنا بشكل أساسي على تمديدات لم يتم دراستها من قبل, و التي تم الكشف عنها من طرفنا. وصلت دقة إختبار التحقق من صحتها إلى 100% على عدة مصنفات.

كلمات مفتاحية : واب، متصفح، امتدادة، امتدادة خبيثة، كاشف.