Università
Ca'Foscari
Venezia

Master's Degree programme – Second Cycle
in Computer Science

Final Thesis

–

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Cookie Protection Through Browser Extensions

Supervisor
Prof. Riccardo Focardi

Co-Supervisor
Dr. Stefano Calzavara

Author
Alberto Carotti
Matriculation Number 838960

Academic Year
2015/2016

Index

1 Introduction

Cookies are the standard mechanism to preserve information about the state of web sessions.
The data they can contain serves a variety of needs, from remembering settings specified for a website to authentication purposes that keep the user logged in as he's browsing through different pages. These important roles make them a common target for web attackers.
If a cookie is successfully stolen from a user, depending on its purpose, the attacker may become able to impersonate the victim on the website the cookie is stolen from, or read some sensitive data.

We take an existing Google Chrome extension named ***Michrome***, focused on the enforcement of security policies through labels associated with elements of active sessions, and build on it new features dedicated to cookie protection [1].
Creating a secure website is hard, since not all developers are careful or knowledgeable enough to be able to deal with every kind of attack, so Michrome operates by assuming websites may be hosting threats, especially *Cross-Site Request Forgeries* (CSRF), and is able to prevent those that break the rules set by its security policies, which indicate what domains a specific domain is allowed to interact with.
The new features for cookie protection are directed on preventing JavaScript attacks, in particular *Cross-Site Scripting* (XSS) exploits, by rewriting the default JavaScript functions used for reading and writing cookies, wrapping them in our own code, so that our security checks can be performed. Thanks to the labels, it is possible to detect when information meant to be kept secret is about to be leaked, or when data may be tainted by untrusted sources, and block the

corresponding action.

This document describes what types of threats the web can present and what security measures exist, with a focus on browser extensions and in particular Michrome, which this work is based on. The ideas behind the new cookie protection layer are then explained, illustrating the main problems of realizing it due to the limitations of extensions. After a thorough description of the technical details of the implementation, we analyze the results of tests and draw the conclusions.

2 Web Security

Since the beginning of the web, malevolent hackers have been exploiting all sorts of security holes to harm oblivious users, trying to gain access to sensitive information in order to impersonate the victim, often with the purpose of stealing money. Several types of threats have arisen through the years.

The most infamous case is **malware**, malicious software that is installed into the user's computer without his consent, or by deceiving him into believing it is legit, and that can perform different unwanted actions, like stealing data and sending it to the attacker over the network.

The term **phishing** is also well-known today, which is the act of tricking a user into communicating information meant to be kept secret, such as passwords, to the hacker, sometimes with methods as simple as imitating a trusted website with a fake one. Despite knowledge about these techniques being common, studies have reported that skillfully crafted phishing websites are sometimes able to fool as many as 90% of their visitors [2].

SQL injections are a threat many websites are vulnerable to. They consist in inserting code as part of a query being asked to a database, so that, when executed, it is read not as an argument of a query, but as actual code, and consequently may allow illicit authentication as administrator or even complete destruction of the database. This exploit targets the website, and its users are only harmed indirectly [3].

We concentrate on two types of attacks that are quite dangerous and common: ***Cross Site Scripting*** (XSS) and ***Cross-Site Request Forgery*** (CSRF).

XSS attacks consist in malicious scripts injected into normally trusted

websites. They can be *persistent* or *reflected*. *Persistent* XSS is stored into a page, by accepting data without validating it: for example, a comment on a discussion board can contain a script that ends up being saved into the page, and whenever other users access that page, the script will be executed in their browsers.
A persistent XSS example may be as simple as inserting a script into a text field meant for comments on a vulnerable discussion board:

```
Nice picture!<script>evil code</script>
```

Reflected XSS attacks happen when a user requests a URL that contains the malicious script as input, instead of otherwise harmless data the website expects: it will not be saved in the page for everyone, but the script will be executed on the computer of the user sending the request. While this implies a long and very fishy URL, link shorteners can be used to conceal it and to avoid explicitly showing the script to the user.[1]
An example of reflected XSS on a vulnerable website could be:
`http://vulnerablesite.com?q=evil code`

It looks suspicious, but with a link shortener it could be made to look as simple as something like `http://short.com/5t34l1nf0`.

Notably, XSS exploits enable the attacker to steal sensitive information such as cookies, which, as explained before, can play very important roles, like identifying a user in a web session [4]. They are very common threats, and even websites as popular as *YouTube* have been found to be vulnerable to them.[2]

1 http://www.hpenterprisesecurity.com/vulncat/en/vulncat/python/cross_site_scripting_persistent.html
2 http://news.softpedia.com/news/Dangerous-XSS-Bug-Found-on-YouTube-146157.shtml

CSRF attacks allow the attacker to generate arbitrary HTTP requests and send them from the victim to a website the victim is authenticated to. Because the website trusts the authenticated user, it can be hard to tell genuine requests apart from malicious ones.

A common case where CSRF attacks may be possible is when a website adopts POST requests. These attacks can allow actions such as illicit money transfers or compromising user accounts. They generally happen when the website receiving the request does not check where it originates from: by making the user visit a website under his control, the attacker can send a precompiled form to the website the victim is authenticated to.

An example of a piece of code in the attacker's website that submits a form to a trusted one which accepts POST requests could be the following:

```
<form name="stealingform" style="display: none" action=
     "http://somebank.com" method="POST">
     <input type="text" name="holder" value="Thief
     McThieves"/>
     <input type="text" name="amount" value="10000"/>
     <input type="submit">
</form>
<script>document.stealingform.submit();</script>
```

CSRF cases are common exploits that even the *New York Times* website was once found to be vulnerable to [5].

There exist solutions that websites can employ to defend themselves and their users from XSS and CSRF attacks.

For example, to prevent XSS attacks, a website can sanitize inputs, removing

characters and tags that are deemed dangerous.[3]
To stop CSRF attacks, tokens can be used, which consist in additional data created in an unpredictable way and only valid for a limited time, stored in the user's session when he first accesses the form and required to be sent back when the form is filled and submitted. This way, forged, malicious requests coming from an external site will be missing the token (or contain invalid ones) and will be discarded, since no corresponding session is found to be open.[4]

Unfortunately there is no guarantee that a website does take advantage of these techniques, and many security measures can actually be bypassed with more complicated attacks. In some cases, caution on the user's part can help, such as inspecting a link's content or avoiding clicking on link shorteners (which allow an attacker to hide the code-injecting part), but of course it is unreasonable to expect every user to have the same expertise and never accidentally click a link without analyzing it first.

3 http://www.ibm.com/developerworks/ibm/library/wa-secxss/
4 http://shiflett.org/articles/cross-site-request-forgeries

3 Browser-side Security

In this section we study what browsers can do to help defend the user from web threats.
Some features are built-in: for example, Chrome contains a *XSS auditor*, capable of detecting and stopping some XSS attacks, but there are ways to circumvent this filter.[5]

Cookies are created and updated either by response headers that follow HTTP requests, or through the JavaScript command *document.cookie='<cookie attributes>'*. Similarly, they are read by the server either from the request headers (all cookies related to a page are automatically appended to request headers by the user's browser) or with the JavaScript command *document.cookie*.
Among the several attributes cookies possess, there is one called *HttpOnly*: when a response header creates a cookie with *HttpOnly* set to *true*, the browser will prevent JavaScript and other non-HTTP APIs from reading or altering it. However it does not allow any granularity in deciding which scripts have access to the cookie: it is either all or none of them; also, it is unfortunate how the spread of this attribute has been slow over the years [6]. In addition, the *Secure* flag, when set to *true*, ensures that the cookie is only attached to requests if they are directed to HTTPS pages [7].
They are only supplemental tools and are not able to thwart every attack, as the protection they provide does not allow for very specific granularity: browser add-ons can give some stronger (although not absolute) security guarantees.

In order to understand existing add-ons, and the problems and solutions behind the implementation of Michrome and its layer for cookie protection, it is

5 http://www.thechromesource.com/tag/xss-auditor-chrome/

important to know how extensions work, in particular on Google Chrome, and what their possibilities and limitations are.
We give an overview of their structure and characteristics and we analyze the original features of Michrome.

3.1 How Browser Extensions Work

Browser extensions, sometimes referred to as add-ons or plug-ins, can modify or add functionalities to a web browser.
The languages used to develop one are HTML, CSS and JavaScript. While each of them has different roles and is, to some extent, used in Michrome, we will focus on JavaScript, as it is the most important one for the implementation of our security goals.

A component that must be present in any extension is the *manifest.json* file, which contains metadata listing name, functionality, version of the add-on but also the permissions it needs and the files it is made of.

The *browser action*, an icon placed near the address bar, is a common but not essential component that, when clicked, performs an operation, usually displaying a pop-up from which the user can take an action that will affect the current page, or edit some settings.

Background page and content scripts, and what differentiates them, are what's most important to understand Michrome and its implementation, as they are the components where most of the work lies in.

The **background page**, as the name suggests, is a page silently running

in the background, usually composed of one or multiple JavaScript files, that contols the behavior of the add-on. These scripts can make use of numerous APIs specific to browser extensions: Chrome in particular supports the most compared to other browsers.[6] [7]
Almost all of these APIs use only asynchronous methods, meaning that execution does not wait for the completion of the method before proceeding. This is useful because it minimizes the performance impact caused by extensions, but the complete lack of synchronous methods is also source to a number of problems and complications that we will explain later.

Content scripts are used to interact with webpages, and can see their DOM to read and alter them. They however run in a separate environment from both the background page and the webpage: they do not have direct access to any of the JavaScript variables or functions of either of them.
This means that sometimes, to make changes in the context of the page, they have to inject code into it; also, to access objects and information that belong to the background page, they have to communicate with it through a very limited number of Web Extension APIs available to them.

There are other components to add-ons, such as a page for options, that we do not need to describe in detail, as they don't play a big role in our following explanations.

3.2 Existing Security Extensions

A lot of browser extensions have been developed to help users protect

6 https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API
7 https://developer.chrome.com/apps/api_index

themselves from threats the Web may host.

AdBlock Plus, one of the most popular browser extensions, is known for blocking ads, but it also interrupts and warns the user when he's about to visit domains known for hosting malware, in addition to protecting privacy by blocking some tracking cookies and scripts. Despite some questionable decisions, such as accepting money to list some ads as acceptable, there is an option to stop all ads.[8]

Disconnect blocks tracking cookies and a number of ads, allowing the user to pick individual scripts and elements of a page that he wishes to stop, a feature *AdBlock Plus* provides too. Ads are annoying, but they can also become a threat when the networks that distribute them are compromised, allowing attackers to spread malware after taking control of it, so disabling them also boosts security.[9]

Ghostery is another common add-on aimed at protecting privacy by blocking trackers used for analytic and advertising purposes. It features a simple interface that lists the individual trackers so that more advanced users can whitelist those they deem harmless.

HTTPS Everywhere tries to move to a secure version of any website visited, giving an option to disable the extension for specific websites in case they are accidentally broken as a result.

ScriptSafe blocks all scripts from all websites, which often breaks them, but the user can add exceptions to make the scripts run if he trusts them.

SessionShield defends sessions from being hijacked, by stripping session identifiers away from headers that set cookies, saving them in its own database, and adding them back when an outgoing request is made, so websites still work properly. It acts on the assumption that no trusted script should be interested in session identifiers, so they do not need access to them:

8 http://uk.businessinsider.com/google-microsoft-amazon-taboola-pay-adblock-plus-to-stop-blocking-their-ads-2015-2?r=US&IR=T

9 http://www.pcworld.com/article/2974537/plenty-of-fish-and-exploits-on-dating-website.html

malicious scripts that attempt to read them will not find them and will be unable to hijack the session [8].

CookiExt is an extension aimed at strengthening session security, redirecting to HTTPS requests that carry cookies that are not marked as *Secure* and/or *HttpOnly*, although their purposes suggest so. These attributes are able to give solid security guarantees, but are often ignored by web developers [9].

3.3 Michrome

Michrome is the original extension this work is based on, and to which the cookie protection layer is added. Michrome accepts that the environment is untrusted and XSS and CSRF attacks may be present, providing an approach that protects the user from some of them, based on security policies that include labels associated with websites.

The labels are categorized as **confidentiality** and **integrity**: as the names suggest, the first one serves the purpose of keeping information secret and preventing it from being read by untrusted entities, while the second one is involved in ensuring that information cannot originate from malicious sources, so that its content is reliable.

A label consists in a set of domains and the associated HTTP or HTTPS protocol; for example, a confidentiality label may look like the following:

```
"C": ["Http(example.com)", "Https(example.com)"]
```

Integrity labels follow the same exact format, only their purpose is different.

When a web request is made, by comparing the labels of the current host with the target one, the extension is able to prevent actions such as unwanted redirects.

For example, if the website *W* we are currently visiting on a specific tab has confidentiality label *C* and a request to a different website *W′* with confidentiality label *C′* is made on the same tab, or on a tab opened by it, the operation is allowed only if *C′* is a subset of *C*. Otherwise, the information we want to keep secret within the set *C* could leak to entities not in this set.

In a similar, mirrored way, if *W* has integrity label *I* and *W′* has integrity label *I′*, the transfer is only granted if *I* is a subset of *I′*. The goal of this check is to make sure that *W′* can trust the information received: if some items in *I* were not present in *I′*, it means the data might have been corrupted by an unknown source.

If a label is not explicitly described in the policy, it is assumed to be equal to *TOP*, which is the biggest set that includes every element, meaning that only another *TOP* can be considered its superset.

These checks are able to stop exploits such as CSRF attacks, provided that suitable labels are specified for the involved websites.

4 Protecting Cookies in Michrome

As seen before, session labels indicate what domains information may be known to (confidentiality) or which ones may have written that information (integrity).
To make cookies secure, we also adopt confidentiality and integrity labels, with similar roles, respectively to prevent secret data from being spread and to make sure its contents are reliable.
Cookies are mainly involved in two operations: *get*, which retrieves the names and values of the cookies available to the current domain, and *set*, which creates, updates or deletes a cookie. Both of them make use of the labels.
Two checks are performed during the *get* operation and both need to succeed, otherwise it will be interrupted:

$Confidentiality(session) \leq Confidentiality(cookie)$
$Integrity(cookie) \leq Integrity(session)$

The first one requires that every element present in the confidentiality label of the session is contained within the confidentiality label of the cookie, while the second one needs every element of the integrity label of the cookie to be present in the session's integrity label.

The check on confidentiality comes from the need of protecting the cookie.
Information contained in it is secret and must not be revealed to any domains other than those listed in the cookie's confidentiality label: if a domain wishes to read the cookie's contents, it must not be able to communicate them to domains outside this set, which are considered untrusted.

On the other hand, the session needs the information obtained from the cookie to be reliable: if unexpected sources had tampered with it, the cookie would be tainted and its false information would corrupt the session as well, hence the need for the cookie's integrity label to be a subset of the session's.

Predictably, the two checks performed for the *set* operation are mirrored versions of the ones for the *get*:

$Confidentiality(cookie) \leq Confidentiality(session)$
$Integrity(session) \leq Integrity(cookie)$

The first check requires that the confidentiality label of the cookie is a subset of the confidentiality label of the session, while the second succeeds if the session's integrity label is a subset of the cookie's one.

The reasoning behind these requirements is rather intuitive and simple to understand. We check that the confidentiality of the cookie is a subset of the session's to protect the latter: if the cookie had looser confidentiality restrictions, information about the session, that we want to keep secret, may leak out.
The check on integrity, instead, is to protect the cookie from having its value tainted by an untrusted domain.

To protect cookies from JavaScript attacks, we need to overwrite the methods used to retrieve and set them.
In Javascript, cookies are retrieved with *document.cookie*, while they can be set through *document.cookie = 'key=value'*. These operations invoke respectively the *get* and *set* functions of *document*'s *cookie* property, sometimes referred to as *getter* and *setter* methods.
More precisely, we have to wrap them within our own methods, so that before

deciding whether or not to invoke the original functionality, the legitimacy of the operation can be checked.

While this sounds like an easy task, there are a few important factors and steps to consider:

- Retrieve the original methods for reading and setting of cookies
- We need to wrap such methods before any script in the page does it, ensure that our own methods cannot be wrapped afterwards, and make it so that the original functionality can no longer be retrieved.
- The wrapping methods must have access to the security policies so that they can decide when retrieving and setting operations are allowed.

What makes these operations problematic is that they need to be executed synchronously, before any of the page's own scripts, otherwise possible malicious code could compromise the extension's cookie protection system and effectively render it useless. As stated before, the asynchronous nature of most APIs makes all this particularly complicated. We will now analyze the issues and solutions employed in detail.

4.1 Injecting Code Synchronously

As previously stated, to override cookie *getter* and *setter*, we need to use a content script and inject code into the page's environment. A few steps must be followed to do this properly.

The code containing the wrappers must be executed before anything else in the page. If scripts from the page were executed first, a malicious one could override the *get/set* methods, compromising their functionality and possibly also preventing us from wrapping them.

To achieve this, we first need to specify the *"run_at": "document_start"* line in the manifest file, which indicates the content script has to run before the page is loaded. However, we also need to inject the code into the page in a way that guarantees its immediate execution.
This can be done by converting it into a string and setting it as the text content of a script we append to the document: in particular, the code is written as an *Immediately-Invoked Function Expression* (*IIFE*), obtained with the following format*:* `(function() {<the code>})();`.[10]

The original cookie *get* and *set* functions are obtained and saved in a variable, so that, once the security checks are performed and complete successfully, they can be called to create or read a cookie:

```
var ck =
     Object.getOwnPropertyDescriptor(Document.prototype,
     'cookie');
```

The code snippet that overrides cookie *getter* and *setter* looks like the following:

```
Object.defineProperty(document, 'cookie', {
     get: function() {
          return getter();
     },
     set: function(val) {
          return setter(val);
     }
});
```

10 http://benalman.com/news/2010/11/immediately-invoked-function-expression/

The *getter()* and *setter(val)* functions are our actual wrappers and have rather complicated logic behind, so they will be explained in detail in a later section.

Protecting our methods from being overwritten is easy, as there is an option, called *configurable*, which when set to *false* prevents the property descriptor from being changed, meaning our wrappers are safe from modifications. *False* is the default value, so there is no need to specify it.

The code above only illustrates how we override the property cookie for *document*s, but we actually do the same for *Document.prototype*.

In fact, *document* actually inherits its cookie *get* and *set* functions from *Document.prototype*, so it is necessary to override them for both. Also, *Document.prototype* needs to be assigned *false* to the *configurable* attribute, as its value is set to *true*.

If we overrode *get* and *set* for *document* but not for *Document.prototype*, an attacker could retrieve the original functionality with the following command:

```
Object.getOwnPropertyDescriptor(Document.prototype,
    'cookie')
```

If we overrode them for *Document.prototype* but not for *document*, our own *get* and *set* functions could be wrapped by defining them for *document*.

4.2 Accessing Security Policies

The wrapper needs to know the security policy in order to perform the checks preceding the *set* and *get* operations. This seemingly simple task

actually spawns a number of issues that are not easily solved.

The script injected into the page becomes completely independent from the extension: it cannot have any references to objects belonging to the content script or background page and it does not have any dedicated "channel" to communicate with them.
Communication between the injected script and the extension is in fact not a viable solution. The *window.postMessage* method would have to be used, which not only would cause issues because it is asynchronous, but it also has no way to guarantee which specific script is sending or responding to the message. Messages are published for anyone listening, with no way to know if the sender is trustworthy, and any response is accepted, meaning malicious code could send fake requests to the extension or, even worse, send responses containing fake policies to the requests made by our injected script.
A feature to allow more direct communication is *externally_connectable*, which at least guarantees who the receiver will be. It however uses asynchronous methods such as *sendMessage* and again cannot guarantee who the sender is, on top of only being available on URLs containing second-level domains, so it is definitely not a solution.[11]

It is hence evident that the code must be injected by the content script together with the policy, which however still poses some problems.
The content script cannot communicate with the background page to receive the policy, as it would involve the *sendMessage* method that we mentioned is asynchronous, resulting in the page loading before the script is injected.
The best solution is to have the policy written in a file, which also helps keeping it tidy and separated from code that implements functions. The problem becomes how to access such file. While the API *chrome.storage* allows

11 https://developer.chrome.com/extensions/messaging#external-webpage

managing files, it is again an asynchronous one and because of it unfit for our needs.

We instead opt to use the *web_accessible_resources* feature of the manifest, which allows to make some extension files available to the web for reading. The content script can then perform synchronous XMLHttpRequests (XHRs) at the beginning to immediately obtain the policies.
For example, to retrieve the policy relative to domains (the process is very similar for the cookies' policy, save for the different file requested):

```
var xhrD = new XMLHttpRequest();
xhrD.open("GET", chrome.extension.getURL
     ('/cookies/domainPolicy.json'), false);
xhrD.send();
var readDomainPolicy = '' + xhrD.responseText;
```

Synchronous XHRs are deprecated because they can potentially be slow, but this is not the case: since there is no real communication over the net (the requested file is on the user's computer), the response time is short.
In case the user wants customized security policies, he will have to edit these files. Note that while files listed as *web_accessible_resources* can be read by any script that knows the extension's ID, they cannot be modified by it so no security threats arise.

Another possible solution would have been using the HTML5 File API, which allows the background page to save a file containing the policy in a virtual file system. The file is reachable by content scripts with a synchronous XHR, and also not readable by foreign, external scripts present in the page.
While this seems ideal, it is unfortunately not a standard API and the W3C has

discontinued work on it. Although Chrome still supports it, there is no telling when it will stop functioning.[12] [13]

4.3 Policy Format

The domain and cookie policies are contained in two separate JSON files, named *domainPolicy.json* and *cookiePolicy.json*. They are declared as *web_accessible_resources* in the manifest file of the extension so that the content script in charge of wrapping cookie *set/get* methods can access them instantly with a synchronous XHR. In order to have customized policies, the user has to edit these files.
As explained in the previous section, this solution was employed as it is the only one that allows us to avoid asynchronous methods, which would break our security guarantees, while also only using standardized technologies.

Being JSON files, policies are written following the JSON format.
For *domainPolicy.json*, domains appear as objects, each associated with a pair *{C, I}*, which indicates the *Confidentiality* and *Integrity* tags. Both *C* and *I* are in turn associated to either the value *TOP* or an array of domains further described by *Http* or *Https*, specifying the protocol.
The structure used for *cookiePolicy.json* is similar but different: domain objects have a series of cookies, each of which associated with its own *{C, I}* pair, detailing the *Confidentiality* and *Integrity* labels.

An example policy for a domain in the *domainPolicy.json* file, with *TOP* as its confidentiality tag and an array of two elements for the integrity tag, is the

12 https://developer.mozilla.org/en-US/docs/Web/API/File_System_API
13 https://dev.w3.org/2009/dap/file-system/pub/FileSystem/

following:

```
"example.com": {
     "C": "TOP",
     "I": [
          "Http(example.com)",
          "Https(example.com)"
     ]
}
```

An example policy for a domain and two of its cookies in the *cookiePolicy.json* file is the following:

```
"example.com": {
     "sample_cookie": {
          "C": "TOP",
          "I": [
               "Http(example.com)",
               "Https(example.com)"
          ]
     },
     "another_sample_cookie": {
          "C": [
               "Https(example.com)"
          ],
          "I": [
               "Https(example.com)"
          ]
     }
}
```

If we were browsing *example.com*, JavaScript would be allowed to both read and write *sample_cookie*: its labels and the session's coincide both for confindentiality and integrity, so *get* and *set* checks are successful.
However, *another_sample_cookie* cannot be read nor written by *example.com*. Regarding the *get*, the confidentiality of the session is *TOP*, which is bigger than the cookie's, as it only includes *"Https(example.com)"*. The other check would succeed, as the cookie's integrity label is a subset of the session's, but both need to pass for the operation to be allowed.
Similarly, the *set* is blocked as the session's integrity label is not a subset of the cookie's, even though the cookie's confidentiality label is a subset of the session's.
These examples are purely for explanatory purposes, they are not recommended policies to use on *example.com*.

While JSON syntax is rather strict compared to JavaScript, there is no need to know JSON to customize policies, as the notation in our files is very simple and intuitive and labels for domains and cookies can easily be written by using those already present as references.
That being said, some attention must be given to write policies properly, such as enclosing objects in curly brackets and not leaving trailing commas, since something as small as an extra (or missing) comma is enough to make the interpretation of the policies completely fail.

5 Implementation

Overriding the *getter* and *setter* involves a few steps:

- Accessing policies and the original *getter/setter* synchronously so that they will be readable by our wrapper methods; this is the easiest step.
- Overriding how cookies are read, a troublesome task due to how difficult it is to identify a cookie since its domain attribute is not returned by the original *get.*
- Overriding how cookies are written, which can be frustrating as the syntax for creating and updating cookies is very relaxed. We have to take into consideration the many fancy ways with which a cookie can be set in an attempt to fool our extension into reading the wrong policy.

5.1 Retrieving Policies and the Original Get/Set

The content script performs two XHRs to retrieve domain and cookie policies from their respective files, defines a function containing the code that will wrap *getter* and *setter*, converts it into string together with the policies as its arguments and injects it into the page by creating a script and appending it to the document.
As explained in previous sections, this is all executed synchronously, before the page is loaded. We will now focus on how the function injected into the page wraps and overrides *get* and *set* functionalities.

The first action performed by the function is the reason policies are written in JSON format: by parsing them with *JSON.parse* they are converted into proper JavaScript objects, which allows them to be easily accessed.

```
var domainPolicy = JSON.parse(domainP);
var cookiePolicy = JSON.parse(cookieP);
```

The object containing policies for cookies will be *cookiePolicy*, while the one for the domains' will be *domainPolicy*. Using them is intuitive: for example, we can refer to the confidentiality label of a cookie of name *name* and domain *dom* simply with *cookiePolicy[dom][name].C*.

The property *cookie* of *Document.prototype* is then retrieved through the use of *getOwnPropertyDescriptor* and assigned to a variable named *ck*.
This way, the original *get* and *set* functions can be preserved.
In the case of *get*, it is invoked with *ck.get.call(document)*, where the use of *call* allows to specify that the *this* argument for the function is *document*, necessary to make it work correctly.
Similarly *ck.set.call(document, val)* is the *set* counterpart, where *val* is a string detailing the cookie being set.
Our wrappers can make the necessary checks based on the cookies' values and the related policies and make use of the original methods in case the results indicate the operation is legit.

Accessing external references, such as DOM variables or global functions, is often dangerous, because an attacker might have tampered with them [10]. For our purposes, we need to access some of these variables and functions (such as the original *get* and *set* function), but as explained before, thanks to the fact that our code is executed before any other is given a chance to, we are able to make sure that all variables and functions have yet to be altered.
The problem still arises in case one of these global elements needs to be

accessed in a later moment, after executing our code: malicious scripts might have had a chance to override them.
For example, we make use of the *document.domain* property during retrieval and update of cookies to know the current page's domain. A malicious script could alter *document.domain* to make Michrome malfuncion and effectively nullify some of our security guarantees.

This attack is trivially neutralized by saving the original *document.domain* value in a variable we call *docDom* at the beginning of execution, and by replacing with it every instance in which we would otherwise use the global *document.domain*.
For the sake of simplicity and making explanations easier, we will not mention our *docDom* variable in other chapters and instead refer to *document.domain*, although the actual code accesses *docDom*.

5.2 Reading Cookies

Although retrieving cookies might sound simpler than setting them, as often is the case with read operations, in our situation it actually involves much more complicated reasoning.
Pretty much everything that is hard about wrapping the *get* method stems from how cookies are returned by the original *getter*.
The original *getter*, normally called with the *document.cookie* command, is invoked using the *ck* variable, in which we stored it, returning all cookies available to the current domain as a single string with the format *key=value*, separating one cookie from another with a semicolon. We save this string into a variable.

```
var getResult = ck.get.call(document);
```

An example of the string obtained after calling *document.cookie* could be the following:

```
city=New York; food=pizza; color=red
```

The only information returned is the names of the cookies and their values: we know there are three cookies accessible to JavaScript named *city*, *food* and *color*, and their values are respectively *Ney York*, *pizza* and *red*, but no other information, such as their domains, is provided.
Our goal is to retrieve all cookies with the original *get* method, search the returned string to find which cookies the domain is not allowed to read, and return a string containing only the cookies that passed the checks.

5.2.1 Separating the Cookies

The easy step is separating cookies from one another, needed to apply the correct policy to each of them.
The simple pattern used by the string to list cookies suggests adopting the *split(separator, limit)* method, which, given a string, splits it into an array of substrings based on the *separator*. The *separator* argument is the character (or string) that will be interpreted as delimiting one substring from the next one, *limit* indicates how many elements we allow the array to be composed of: every further item will be discarded.
Our implementation does not need the *limit* argument, so we omit it.

```
var cookieArray = getResult.split('; ');
```

Using semicolon followed by a space (*'; '*) as the separator, we obtain an array where every element is a *key=value* string describing a cookie, save from nameless cookies that appear as *value*-only strings.
Cookies may indeed have no name, but this mostly complicates matters during their creation, which will be better explained in the **Setter** section; their impact during retrieval is minimal.
To extract only the name from the *key=value* format we use the *indexOf* method in its simplest way: given a string *s*, *s.indexOf('=')* returns the index of the first instance of the equal sign in *s*, or *-1* if it never appears, which happens when the cookie has no name.

By invoking *substring(start, end)* with values 0 for *start* and the index of the equal sign for *end*, the name of the cookie is extracted. If the equal sign was not found, *end* has value *-1* and *substring* returns an empty string: this is correct since the cookie is nameless.

```
cookieName = cookieArray[i].substring(0,
     cookieArray[i].indexOf('='));
```

As we now have the name of each cookie, we should perform the two necessary checks on them:

Confidentiality(session) ≤ Confidentiality(cookie)
Integrity(cookie) ≤ Integrity(session)

If both succeed, the cookie will be appended to a string which will be returned after all cookies have been checked, if one or both fail, it will not be included in such string.

The string built by our wrapper has the same format of the original *get* method and only differs for the cookie that were discarded; if in the previous example the cookie named *food* failed to meet the requirements, it would look like this:

```
city=New York; color=red
```

5.2.2 The Domain Attribute

We define policies for cookies by grouping them based on the domain. If we want the policy for the cookie *abc* on *example.com*, it is accessed with the command *cookiePolicy['example.com']['abc']*.
If we were to identify cookies solely based on their names, it would soon happen that two cookies with the same name but originating from different websites can no longer be distinguished one from another, and it would also make the policy file very confusing to read.
Using the domain in addition to the name is the most intuitive solution to group and differentiate cookies properly.

We saw before that *document.cookie* returns extremely scarce information, and this poses a problem. Chrome does offer the *chrome.cookies* API, which allows to find more information about the cookies retrieved, but other than being asynchronous, since it is specific to Chrome, it is not even available to scripts that get injected into the context of a page.
With as little data as *key=value* to work with, we cannot truly know what domain a cookie is set for.
In fact, just like a cookie can be set to be readable by the current domain only, it can also be created so that all subdomains may access it as well. It can even be set so that the current domain, its superdomain, and all of the superdomain's subdomains can read it.

This is done by specifying the *domain* attribute while setting the cookie. If left out, the cookie will be available to the current domain only. If present, the domain specified can read it, but the same can be done by all of its subdomains.[14]
We refer to these cookies available to multiple domains as *domain cookies*.

For example, let's assume the current page is on domain *example.com*. A script executes the following command, which generates a new cookie:

```
document.cookie = 'abc=def';
```

A cookie named *abc* is created with the *domain* property set to *example.com*. Only *example.com* will be able to read its value; its subdomains, like *www.example.com* or *forums.example.com*, will not see it.

Assuming *example.com* executes the following script instead:

```
document.cookie = 'abc=def;domain=.example.com';
```

The cookie will instead have domain set to *.example.com*. Not only *example.com* can access it, but also *www.example.com*, *forums.example.com*, other *example.com* subdomains and also all subdomains of those subdomains.

The same script as before can also be executed by *www.example.com*, or any similar subdomains:

```
document.cookie = 'abc=def;domain=.example.com';
```

14 http://javascript.about.com/library/blwcookie.htm

Similarly, the resulting cookie will be set for *.example.com* and be available to the same domains listed before.
Note also that the dot is actually superfluous: as long as the script lists the domain, whether the value is *example.com* or *.example.com* is unimportant and will result in a cookie being saved with domain *.example.com*, that can be read by all subdomains.[15]
The following two lines of code have the same effect.

```
document.cookie = 'abc=def;domain=.example.com';
document.cookie = 'abc=def;domain=example.com';
```

While we are studying Chrome, which is the browser this extension is available for, it is interesting to note that Internet Explorer does not comply with the RFC document that determines what described above: if the domain is not specified, the cookie is made to be available to all subdomains as well.
Internet Explorer staff makes it clear that it is not a mistake and it is intentional, although a reason is not given.[16] [17]

Now let's consider the *get* function. Upon calling *document.cookie* and receiving a cookie *abc*, depending on how deep the current page is within the tree of domains and subdomains, there could be several domains the cookie is coming from.
Reading *abc* from *foo.bar.example.com* means the cookie has a domain set to one out of the following options: *foo.bar.example.com*, *.foo.bar.example.com*, *.bar.example.com*, *.example.com*.
Even reading *abc* from *example.com* is complicated though, as there's no way to tell if it is a cookie unique to *example.com*, or if it is a domain cookie for

15 http://tools.ietf.org/html/rfc6265#section-4.1.2.3
16 https://blogs.msdn.microsoft.com/ieinternals/2009/08/20/internet-explorer-cookie-internals-faq/
17 http://erik.io/blog/2014/03/04/definitive-guide-to-cookie-domains/

.example.com, which could have been set by a subdomain.

5.2.2.1 Issues Arising with Unknown Domains

Not knowing the domain when we retrieve cookies through *document.cookie* is the real reason wrapping the *get* method is so complicated. Because of this, when performing a *get* and obtaining a cookie, we cannot be sure which policy we have to inspect. This is a quite complex problem with some security implications that should not be overlooked, so we analyze possible solutions.

The first fix that might come to mind would be to start from the current domain and search for a policy matching the cookie at hand by **climbing the domain tree** one level at a time.
The idea behind is that, if document.cookie returns a cookie whose policy is not present for the current subdomain, like *bar.example.com*, it might be a domain cookie with its policy defined for a superdomain such as *.example.com*. By going up the domain tree, eventually the policy would be found, if any is specified, otherwise it would be equivalent to having *confidentiality=TOP* and *integrity=TOP*.

However, there is the possibility that the page has access to different cookies, set for different domains, but that have the same name.
Performing the *get* through *document.cookie* might return:

```
abc=def; abc=123; othercookie=somevalue
```

We find ourselves with two distinct cookies named abc, but with different

domains. If we are on the page *foo.bar.example.com*, one might come from *foo.bar.example.com* while the other comes from *.example.com*.
By simply stopping at the first occurrence of a policy in the file, we would be treating the two cookies the same way, blocking a cookie that normally would be read, or, even worse, allowing a cookie that was supposed to be kept secret to be read by an unauthorized page.
Even if we were to continue climbing until we found the second policy, we would have no means of knowing which policy applies to which cookie.

A second idea to ensure security could be to **compute the strongest, safest policy** out of those found for the cookie.
Recalling the checks necessary for the cookie to be read:

Confidentiality(session) ≤ Confidentiality(cookie)
Integrity(cookie) ≤ Integrity(session)

This would mean performing intersection of the cookie's confidentiality labels found, and calculate the union of the cookie's integrity labels. This method, although resulting in preventing some legit reads from happening, is successful in terms of security. In general, by restricting confidentiality we are making data more secret, while by relaxing integrity we are pessimistically assuming data is corrupted. However it is not fit to be used as it results in a far too high number of legit reads being blocked.

What if the cookie came from *.bar.example.com*? The policies we have found are for *foo.bar.example.com* and *.example.com*: however, if the cookie comes from *.bar.example.com*, which has no policy defined, its labels are supposed to be *TOP* both for confidentiality and integrity.
The integrity label will almost always need to be assumed equal to *TOP*,

because it is the biggest set and uniting it with any other set will produce *TOP*. As a result, the cases where cookies are unfairly blocked will be every single situation in which the session has integrity label anything other than *TOP*.

The only expedient which would make this work is that, once a policy is defined for a domain's cookie, it is defined for cookies with the same name in every single superdomain. This causes pollution in the policy file and, although not as badly as assuming every cookie's integrity label to be *TOP*, still leads to legit reads being blocked.

It needs to be clear that we cannot ignore this problem and cannot simply hope that websites do not use multiple cookies with the same name. In the next section we explain why not knowing the domain of a cookie is dangerous even if the website has a unique name for each cookie.

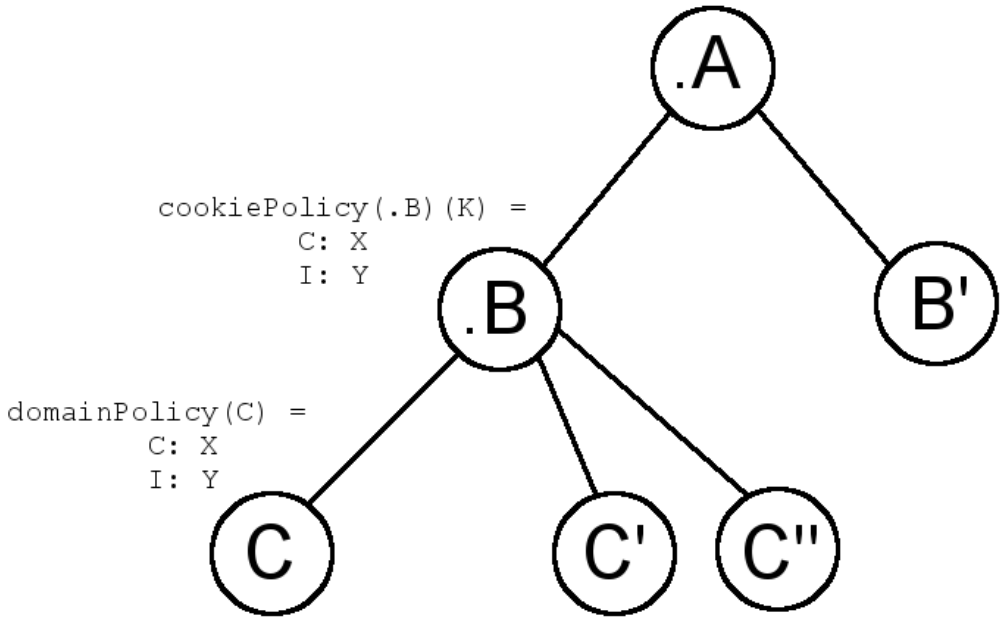
5.2.2.2 Sample Attack with Cookies of the Same Name

We present an attack that manipulates information read by a script by abusing the problem with cookies of the same name.

We consider a tree of domains of an imaginary website *W*, which offers various services, from checking the weather to shopping online.
Leaves are regular domains (like *foo.bar.example.com*) while internal nodes are superdomains starting with a dot (like *.example.com*), meaning cookies defined for an internal node are accessible to the leaves belonging to its subtree.

We use simple names for nodes, like *.A* or *.B* for superdomains with a dot, and *B'* and *C* for regular domains.

For example, in the picture, cookies defined for *.A* are readable by *B'* but also *C*. Cookies with domain set to *.B* are readable by *C* but not by *B'*.



We imagine *.B*'s children, like *C*, *C'*, *C''* etc. to be a series of subdomains dedicated to the shopping section of the website. To make a concrete example, they could be *clothes.shop.example.com*, *books.shop.example.com*, and so on. In this case, *.B* would be *.shop.example.com* and *.A* would be *.example.com*.
W makes use of a domain cookie *K*, defined for the domain *.B*, which contains the address to send the bought products to. The user specifies this address in a form contained in C or any of its brothers. This way, since the cookie is defined for *.B* and shared by all its subdomains, there is no need to indicate the address again when switching to a different shopping subdomain.
Obviously managing delivery like this is very inappropriate and bizzarre, and probably no website handles the process in such a way, but it is suitable for explanatory purposes.

To protect the cookie *K*, which contains our address, we assume to have some policies defined.
In particular, *C* has a session policy defined such that its confidentiality label is *X* and its integrity label is *Y*, where *X* and *Y* indicate a finite set of domains we trust, for example, they could be other shopping subdomains such as *C′* and *C″*.
The cookie *K* instead has a policy defined for its domain *.B*, with confidentiality label the same *X* as the session and also integrity label the same *Y*.
We expect our policies to make the cookie's value safe, because we think that its integrity label *Y* guarantees that the value read is trusted, as in, it can only come from our own input when we specify the mail address in *C* or one of its brothers.

However, imagine *B′* (which could be, for example, be a domain of the website dedicated to weather forecast) to host a malicious script, injected by an attacker, that creates a cookie with the same name *K* in an attempt to steal the things we buy.
Since *B′* is a children of *.A*, it sets the domain of the cookie *K* to *.A*. No policies exist for these domains, which means it is equivalent to having domain policy for *B′* equal to *TOP* for both confidentiality and integrity, and the same for the cookie *K* set for domain *.A*, which would have confidentiality and integrity labels equal to *TOP*. When we visit *B′* to check the weather forecast, the fake cookie *K* is hence set successfully by the malicious script.

We then decide to do some shopping and move to the domain *C*.
When we buy products, the website invokes *document.cookie* to know what address to send the products to, and obtains the following result.

```
K=honest_street_19; K=thief_avenue_97
```

Having found two cookies named *K*, *C* cannot tell which one is coming from the trusted *.B* and which one is from the potentially dangerous *.A*. Unable to distinguish the real address to send items to, it might pick the attacker's.
Even if the website was programmed to be suspicious of cookies of the same name, and warned us we likely had a malicious cookie stored, it would not be a real solution. If we had forgotten to fill in the form with our address, only the fake *K* would be saved, and the attacker would steal the things we buy.

As mentioned before, a possible solution would be to force the user to, whenever he defines a policy for a domain's cookie, also define it for cookies of the same name in all of it superdomains starting with dots. This way, malicious scripts could no longer create fake cookies by exploiting the missing policies, and we could compute the strongest policy out of the defined ones.
While it is secure, it results in a more cluttered, confusing policy file, and can still cause legit read operations to be blocked when the computed strongest policy is stricter than the cookie's own.

5.2.2.3 The Domain Identification Suffix System

Having displayed how dangerous it is to not be able to distinguish the domains, regardless of whether a website uses multiple cookies of the same name or not, it is necessary to find a secure solution.

By getting the cookie *setter* involved in this, it is possible to make the domain known. This can be achieved by having the *set* function append an identification suffix at the end of the name of a domain cookie.
The identification suffix will be composed of two parts: the first will be a specific

string, identical for all domain cookies, that helps the *get* wrapper realize it is handling a domain cookie; for example, it can be *#MCDMCK_*.The second part is the domain itself, which will allow the wrapper to easily tell the domain of the cookie.

A cookie named *abc* set for domain *.example.com* will actually be named *abc#MCDMCK_.example.com*.
If it is not a domain cookie and instead is set for the current domain, it will simply not have a suffix. The cookie named *abc* set for domain *forums.example.com* will keep its *abc* name.

We will explain in a later section exactly how the suffix is added during the creation of the cookie, as right now it is sufficient to simply know that domain cookies have an identification suffix thanks to the extension. For now, we will focus on how the suffix system influences the implementation of the *get* wrapper.

After invoking the original *get* with *document.cookie* and identifying the individual cookies, we look for the presence of the identification suffix by calling *indexOf('#MCDMCK_')* on each cookie: if the returned value is *-1*, it means the suffix was not found, which implies it is a regular cookie, and its domain is set to the current page's domain, obtained by calling *document.domain*. If the suffix is found, the *substring* method allows to extract the domain value.

```
domainIndex = cookieName.indexOf('#MCDMCK_') + 8;
cookieDomain = cookieName.substring(domainIndex);
```

Summing 8 in the first of the two lines above is needed to position the domain index after the identification suffix: 8 is the length of the string

'#MCDMCK_'.

Knowing the cookie's name and now also the domain it is coming from, the implementation has no issues finding exactly what policy is to be applied.
If the cookie is indeed allowed to be retrieved, the wrapper takes care of silently removing the identification suffix before returning the cookie, so that the whole operation is done transparently. This way, the extension is able to protect cookies while also making sure websites are not broken by unexpected cookie names.

```
newCookieName = cookieName.substring(0, indexOfSuffix);
```

Let's consider again the attack illustrated in the previous section, and see step by step how this solution solves the issue while keeping the website's functionality intact.
When the malicious script injected into *B'* attempts to create a cookie *K* for domain *.A*, it is actually saved with the name *K#MCDMCK_.A*.
Having filled out the form with our address while browsing *C*, the cookie *K*, saved with the domain attribute equal to *.B*, has been renamed into *K#MCDMCK_.B*.
After confirming that we are ready for our purchases to be delivered, the website calls our wrapper's get method to check the address. The wrapper initially calls the original *document.cookie getter* and obtains:

```
K#MCDMCK_.B=honest_street_19; K#MCDMCK_.A=thief_avenue_97
```

The cookies no longer share the same name, and their domains are explicit to the extension. By extracting the domains from the cookies' name, the wrapper can tell what policy applies to what cookie, and figures that

K#MCDMCK_.B's value can be trusted, while *K#MCDMCK_.A* should be blocked.
The second cookie is hence blocked, and the first is returned without the suffix:

```
K=honest_street_19
```

The website's functionality has been left intact while the attack has been prevented.

Since some cookies are used by JavaScript but also as HTTP cookies, it is necessary to implement the identification suffix system for the latter as well: if we did not incorporate suffixes for cookies exchanged through headers, we would end up with all sorts of inconsistencies. To avoid breaking the flow of the wrappers' explanations, HTTP cookies will be explained in detail in a later section.

5.2.3 Performing the Checks for the Get

In the section about the labelling system we explained that two checks are carried out by comparing the current domain's policy with the policy for the cookie about to be read. The two checks are:

Confidentiality(session) ≤ Confidentiality(cookie)
Integrity(cookie) ≤ Integrity(session)

Respectively, they are implemented with two methods called *cSessionLTcookie* and *iCookieLTsession*, which both return either *true*, if the check was successful, or *false* if it failed. If even only one of them returns *false*,

the operation is interrupted, otherwise the cookie can be read. Both checks are performed for every cookie obtained by calling the original *document.cookie getter*. Thanks to the domain identification suffixes, these methods can be implemented in a simple and intuitive way.

cSessionLTcookie(name, dom) returns true if the confidentiality label of the session is less than or equal to that of the cookie of name *name* and domain *dom*. Every element of the session's confidentiality label has to be included in the cookie's.

```
function cSessionLTcookie(name, dom) {
     if (cookiePolicy[dom] == null || cookiePolicy[dom]
[name] == null)
          return true;
     var polCookieC = cookiePolicy[dom][name].C;
     if (polCookieC === 'TOP')
          return true;
     if (domainPolicy[docDom] == null)
          return false;
     var currentDomC = domainPolicy[docDom].C;
     if (currentDomC === 'TOP')
          return false;
     if (isArraySubset(currentDomC, polCookieC))
          return true;
     return false;
}
```

The variable *cookiePolicy* contains all defined policies for cookies. If *cookiePolicy[dom]* returns *null*, it means there is no policy defined for cookies of

the domain *dom*; similarly, if *cookiePolicy[dom]* is not *null*, but *null* is instead returned by *cookiePolicy[dom][name]*, it indicates that while *dom* has some cookies with a policy, the cookie *name* does not have one.
Both cases translate to a confidentiality (and integrity) label equal to *TOP* and, since *TOP* is the set that contains all elements, *true* is returned. Obviously, if the policy is defined but equal to *TOP*, again the result is *true*.
If execution gets past these checks, and instead either cannot find a policy for the session, or finds its confidentiality label to be equal to *TOP*, it returns *false*, as it cannot be a subset of the cookie's label.
Finally, if the last check is reached, the method *isArraySubset(currentDomC, polCookieC)* is called and returns *true* if the first argument is a subset of the second, otherwise *false*.

iCookieLTsession(name, dom) works in a similar way for the integrity label, now inverting the roles of cookie and session.
If the integrity label of the session is not defined or equal to *TOP*, it returns *true*. Otherwise, it checks the cookie's label and returns *false* if it is not found, equal to *TOP*, or not a subset of the session's. If it is a subset, *true* is returned.

5.3 Creation, Update and Deletion of a Cookie

JavaScript can create, update and delete cookies with a command that usually looks similar to *document.cookie = 'key=value';*, which we call the *setter*. The name of the cookie is indicated by *key*, its value is *value*. Optionally, a number of additional properties can be added while setting the cookie, separated by semicolons, such as domain or expiration date.
The only attribute required is the **key** (name) with the associated **value** of the cookie, which always has to be the first: if we create a cookie with something

like *document.cookie = 'domain=example.com;<etc.>'*, we will not make a cookie for the domain *example.com*, but instead a cookie named *domain* and with value *example.com*.

The policy is based on the *name* and *domain* of cookies, so we need to extract that information from the string passed to *document.cookie* in order to perform the security checks.
The format for *document.cookie = 'key=value'* and its possible optional attributes is very loose and accepting, and it can become very frustrating to discern the exact name and domain, since the original *getter* permits multiple whitespaces, and even tolerates the presence of invalid strings, simply removing the attribute that was not specified correctly and assigning it a default value.

Dealing with these unpredictable strings is troublesome and makes the resulting code confusing, and at first glance might seem to only serve the purpose of supporting those few websites that accidentally define cookies in such weird ways.
However, ignoring these situations actually opens glaring security holes that can be exploited and, unless we take all the necessary precautions to find the right name and domain of cookies even in case of irregular strings, we risk mismatching cookies and policies and exposing the extension to attacks that take advantage of this.

For example, let's compare the following two lines of code, and suppose that **the second one is a script** that has been injected into *foo.example.com*:

```
document.cookie = 'abc=def; domain=.example.com
document.cookie = 'abc=def; domainn=.example.com
```

The two lines of code are **not equivalent**. The first one creates a domain cookie named *abc* for domain **.example.com**.
This is not the case for the second line. The word "domain" actually has an extra *n* and is spelled as "domainn": the original method for setting cookies does not throw an error; instead, it ignores the "domainn" attribute and generates the cookie *abc* with the default value for domain: the current page's domain ***foo.example.com***.
Since we want to preserve its functionality, our wrapper also has to ignore incorrect attributes.
With a poorly written domain value extraction algorithm, we might not notice the extra *n*, assume the cookie's domain will be *.example.com* and, as a result, check the wrong policy. Much in the same way, an improper algorithm can be manipulated into reading the wrong name. Noticing this or similar flaws, malicious scripts can be successful in creating a specific cookie despite not holding the permissions to do so.

Unlike what we do in the *getter* wrapper, we do not use the *split* method, as we are only interested in two attributes.
Instead we find key and domain by looking for specific characters in the string, which delimit a field from the next one or its name from its value. Attributes are separated by a semicolon, while the equal character associates name and value.
Correctly extracting name and domain from the argument of the *set* function is complicated because of the issues described above, and numerous function calls and variables are involved to achieve this, making the code rather hard to understand. We explain the reasoning behind the implementation to make it less confusing.

An important method we use in this process is again *indexOf(searchvalue, start)*, which we now make more extended use of: it returns the index of the first instance of *searchvalue* (which can be either a single character or a string) starting from the position *start*, which, if omitted, will use the default value 0, meaning the search starts from the beginning of the string.
The *substring(start, end)* method is also again useful: as we have seen before, it returns a substring of the given string starting from the character at index *start* and up to the one at index *end* (the *end* character is however not included). If *end* is omitted, it returns the rest of the string.

5.3.1 Extracting the Cookie's Name

In an attempt to forge a malicious cookie, an attacker aware of our extension could try to exploit the identification suffix system to create a cookie that will confuse the *get* wrapper and induce it into reading the wrong policy.
The attack could simply involve creating a cookie for the current domain (by not specifying any), but with a name that ends with the *#MCDMCK_* suffix followed by the domain we want to induce the *get* wrapper into reading the policy of.
This is easily taken care of by using *indexOf* to look for the presence of the *#MCDMCK_* string: if it is found, the operation is blocked.

```
if (val.indexOf('#MCDMCK_') != -1) {
     console.log('<warn user>');
     return;
```

The main issue that we need to address when extracting the cookie's name is that it is possible to set cookies that do not have a name by simply not

specifying any.
For example, *document.cookie = '=abc'* sets a nameless cookie of value *abc*; alternatively, the equal sign can also be left out.
This complicates implementation a little, but we are still able to notice a nameless cookie is being set by carefully analyzing where other characters are, like the first occurrence of a semicolon:

```
var cookieName;
var indexOfEqual = val.indexOf('=');
var indexOfSemicolon = val.indexOf(';');
if (indexOfEqual == -1 || (indexOfSemicolon != -1 &&
indexOfEqual > indexOfSemicolon))
     cookieName = '';
```

The code above realizes that the cookie is nameless if there is no equal sign, or if it is present but it appears after the first occurrence of the semicolon character. If the cookie is set with something similar to *document.cookie = '=abc;<etc.>'*, where the equal sign is present before the first occurrence of the semicolon character, our implementation still works properly, by saving the empty string as the cookie name:

```
else
     cookieName = (val.substring(0, indexOfEqual)).trim();
```

No matter the amount of whitespaces before and after the name of the cookie, it will all be ignored during the creation, hence the use of the *trim()* method, which removes whitespaces at the beginning and end of strings. Even in the case of something such as the following, which is equivalent to a nameless cookie of value *123*, *trim()* correctly results in assigning the empty

string to the *cookieName* variable:

```
document.cookie = '            =123';
```

5.3.2 Extracting the Cookie's Domain

Identifying the domain also implies some thoughtful considerations and is overall even more complicated.
The simplest case is when the domain is not specified: the value will automatically be set to the page's current domain, easily obtained through *document.domain*. If it is set, we have to take into consideration that it is no longer the first attribute specified and it can be in any position.
To know that a domain is specified, we will need to look for the substring *';domain'*: note how the semicolon is included to avoid picking invalid strings that simply end with the word *domain*, but do not actually decide the domain.

Before doing so we invoke *replace(/;\s+/g, ';')*: this confusing command looks for any instance of semicolon followed by one or more spaces and replaces it with only a semicolon: as the whitespaces are ignored during the setting of cookies, no functionality is broken, and it helps detecting the *';domain'* string in case spaces are present (for example *'; domain'*, or even, with multiple spaces, *'; domain'*), as they would prevent us from finding it. We also invoke *toLowerCase()* in order to avoid worrying about matching case. In the snippet of code below, *v* is the original string passed to *document.domain='<v is this>'*.

```
var val = v.replace(/;\s+/g, ';');
var domainIndex = (val.toLowerCase()).indexOf(';domain');
```

After finding the *';domain'* string, we check, starting from its index, the position of the first instance of the equal sign.

```
var domainEqualIndex = val.indexOf('=', domainIndex);
```

If *domainEqualIndex* is *-1* it means no equal sign could be found and we are actually facing an invalid attribute, so the domain will be set to the default (the current page's domain, *document.domain*). Between the *domain* word and the equal sign, there could be whitespaces, which will be ignored, but if a different character is present in the middle, it is again an invalid attribute.
We can notice this situation by checking if, by replacing all whitespaces in the string following *';domain'* and preceding the equal sign, the empty string is obtained: if it is not empty, there is a different character in the middle and we fall back to the default. We can find such string by retrieving the substring starting from *domainIndex+7* (7 is the length of *';domain'*) and ending at *domainEqualIndex*.

```
var betweenDomEq = val.substring(domainIndex + 7,
     domainEqualIndex);
```

The following code checks if *betweenDomEq* is either empty or only made of whitespaces, in which case it saves in *domainStart* the length of the substring starting from the beginning of the *setter*'s argument up to and including the equal sign related to the domain attribute: this number will be useful when extracting the domain's value.

```
if (!betweenDomEq.replace(/\s+/g, '').length)
     var domainStart = domainEqualIndex + 1;
```

If not empty (or if the equal sign for the domain was never present to begin with), we assign *-1* to the variable *domainIndex*, which makes sure in a later check that the default will be used.
By looking for a semicolon, we check if other attributes are defined afterwards: if they are, the domain value is from *domainStart* up to the semicolon at *domainEnd*, if no other attributes are present, it is up to the end of the string. After all these precautions, we are able to extract the domain associated with the cookie.

As we have seen in a previous section, if the domain attribute is listed, the cookie is set not only for that domain, but also for all of its subdomains, something that is denoted by a dot at the beginning of the domain name once the cookie is saved by the browser.
Summing up with some examples what was said previously:

- if the page *forums.example.com* sets a cookie through JavaScript without specifying its domain, it will be saved with *domain = forums.example.com*, and only be read by *forums.example.com*
- if *forums.example.com* sets a cookie specifying domain *.example.com*, the cookie will be saved with domain *.example.com* and be accessible by *example.com* and all its subdomains, such as *forums.example.com* or *account.example.com*. Even if the dot is not present, it will be added to the domain attribute's value by the browser, so *domain=example.com* has the same effect as *domain=.example.com*.

Our implementation also takes these possibilities into account, adding the dot if missing, to properly recognize the domain.

```
if (cookieDomain.charAt(0) != '.' && cookieDomain != '')
     cookieDomain = '.' + cookieDomain;
```

Note that a website cannot set a cookie for any domain it wants: *example.com* cannot set a cookie for *google.com*. If this was possible, it would be a severe security threat, but we do not need to worry about it as the original *set* function already takes care of preventing this.

5.3.3 Performing the Checks for the Set

With the cookie's name and domain, our wrapper can now check for a corrispondence in the policy by calling the *iSessionLTcookie* and *cCookieLTsession* methods, which perform the necessary checks before deciding whether the cookie violates the policy and needs to be blocked or if it can be set.

As we explained in a previous section while explaining cookie labels, two checks need to succeed for a script to be allowed setting a specific cookie.
These checks are:

Confidentiality(cookie) ≤ Confidentiality(session)
Integrity(session) ≤ Integrity(cookie)

Our extension implements them through the *cCookieLTsession* and *iSessionLTcookie* methods, which are similar to the methods used for the *getter*.

cCookieLTsession(name, dom) checks that the confidentiality label for the cookie of name *name* and domain *dom* is less than or equal to the confidentiality label of the current domain, meaning that every element in the

cookie's label also appears in the domain's.
The mehod looks for the current domain's policy with the command *domainPolicy[document.domain]* and, if it cannot find it, it means it is equal to *TOP* and returns *true*, as *TOP* is the set that contains all elements.
Similarly, true is returned if the policy is found but the confidentiality label is *TOP*.
The cookie's policy is then searched in *cookiePolicy[dom][name]* and, if not found or equal to *TOP*, *false* is returned, as reaching these checks means the domain policy is smaller than *TOP* and hence the cookie's confidentiality label cannot be smaller.
If neither of them is equal to *TOP*, *isArraySubset(polCookieC, currentDomC)* is called, which checks that the first argument is a subset of the second.

iSessionLTcookie(name, dom) checks that the integrity label of the current domain is less than or equal to the integrity label of the cookie of name *name* and domain *dom*.
Similar to the previous case, if the integrity label of the cookie is not found or is equal to *TOP*, it returns *true*, as TOP is the biggest set.
After getting past this check, if instead the domain's integrity label is not found or is *TOP*, *false* is returned.
Finally, *isArraySubset* returns *true* if every element in the domain's integrity label is contained in the cookie's one.

If a script causes a policy violation, the cookie will not be set, and the extension prints a message in the developer console for the current page, informing what the name and domain were for the cookie that the script attempted to create or update.
If the checks are successful and the page is allowed to set the cookie, the *setter* can now decide whether or not to add the identification suffix. If the

domain was not specified, there is no need to add it. If it was, the suffix is appended to the end of the cookie's name with a simple line:

```
var newCookieName = cookieName + '#MCDMCK_' +
     cookieDomain;
```

The old name is then replaced with the new one, and the cookie is set.
As mentioned before, it is important to preserve consistency between situations where cookies are exchanged via headers and where they are read or created with JavaScript. The next section explains how the identification suffix system is handled for HTTP cookies.

5.4 The Domain Identification Suffix System for HTTP Cookies

In order to preserve consistency with the suffix system used to make domain cookies identifiable by our wrappers, HTTP cookies need to implement suffixes too.
This functionality was added by modifying the already existing file *events.js*, which contains listeners for web requests and responses.

Chrome offers an API called *webRequest* that allows to analyze and intercept web requests. It is among the very few Chrome APIs that offer synchronous methods, essential to achieve our goals.
The first listener we are interested in is called *onBeforeSendHeaders*, which, as the name suggests, is triggered right before the headers of a web request are sent. It allows to specify a callback function that can read the headers (found in

the *details* object) and, if the *blocking* option is set, as in our case, it becomes synchronous and can also modify them before they are sent over the network.
Request headers contain an element called *Cookie*, which works very similarly to the JavaScript *document.cookie* command, since it contains all cookies that are being sent to that page in a *key=value* format.
The function looks for this header element, locates cookie names and removes the identification suffixes from them if found. This process is rather similar to the one adopted by the JavaScript *get* wrapper.

```
if (details.requestHeaders[i].name.toLowerCase() ===
'cookie')
     <removes domain identification suffixes>
```

The other listener is *onHeadersReceived*, triggered when a response header is received. If the *blocking* option is set, the headers can be modified before they are accepted as a response.
This listener was already present in the original Michrome for other purposes, but the suffix system implementation can be added to it without any issue.
Response headers contain the *Set-cookie* field which, unsurprisingly, sets cookies, and does so in a similar way to JavaScript's *document.cookie = 'key=value'* command.
We are able to detect if the cookie being set is a domain cookie, and in this case add a suffix to the cookie's name in the same way we do it for cookies set via scripts.

```
if (details.responseHeaders[i].name.toLowerCase() ===
'set-cookie')
     <adds domain identification suffix to domain cookies>
```

The already existing file *utils.js* was also modified, adding some useful functions necessary for extracting HTTP cookies' names and domains. Since this process works much like with script cookies, we will not describe its implementation.

5.5 Further Michrome Changes

While this work focuses on cookies, a few other changes were made to some of the already existing files and features of Michrome, listed here for the sake of completeness.

The browser action used to simply create a new tab when clicked, but it was changed to display a popup by modifying the *browser_action.html* and *browser_action.js* files.
The popup retrieves information from the *options* page to present the active tab's labels. This helps the user figure out a possible policy to define for the domain, which can be a rather complicated and time-consuming process.
The file *model.js* has also been modified to add the *toPossiblePolicy* method, which prints labels in a more JSON-like way for easier copying and pasting.
A button to open a new tab is also present in the popup for the sake of conserving the original functionality.

Domain policies used to be defined in JavaScript code directly in the *policy.js* file, but with the need to have policies available in a JSON file to make them easy to edit and available to the content script, *policy.js* was modified to retrieve them from *domainPolicy.json* through a XMLHttpRequest.

6 Tests and Results

Various tests were performed on a website built for the occasion to confirm the correct functioning of the extension.
Policies are read synchronously thanks to the XHR and interpreted by *JSON.parse* before injecting the code. The injected code is executed before any of the page's own, guaranteeing that no global variable has been tampered with before reading it. The *get* and *set* methods are successfully wrapped and protected so that their original functionality cannot be retrieved by other scripts.
The domain suffixes work as desired, indicating the correct domain in the cookies' names and being added and removed transparently so that websites are not broken by them.
Regardless of how inconveniently a cookie's name and domain are indicated by a script that sets a cookie, the algorithm for extraction has always derived the correct ones.
The checks executed on labels were successful in allowing the right operations to take place while stopping those that did not abide by the policies.

Some tests have also been conducted on some popular websites to make sure that the extension does not break them while also preventing JavaScript attacks.
Due to the high amounts of cookies adopted by websites and the difficulty in finding their purposes, only a few websites were tested.
Although it was easy to accidentally specify an incorrect policy or forget a small detail (like a comma) in the definition of the labels, once these mistakes were corrected, functionality never appeared to be broken.
Attacks were simulated by executing the "malicious" script in the browser's developer console: as they run in the context of the page, they can be

considered analogous to injected scripts. Michrome was successful in stopping the reading of cookies or creation of them when the policies did not allow it.

The extension has been tested on the following websites: *amazon.com*, *facebook.com*, *google.com*, *instagram.com*, *msn.com*, *neocities.org*, *twitter.com*, *wikipedia.org*, *yahoo.com*, *youtube.com*.
Depending on the website, policies sometimes needed to be very big to keep all of its features running properly, as there can be numerous domains involved, but all tests were successful.

7 Conclusions

The extension is able to accomplish all of its intended tasks and also preserve its correct behavior even in websites containing scripts that try to impede it, thanks to the fact that our code is executed before any of the page's own and due to the measures taken to protect it from later attempts at tampering with it such as overriding of global variables.
When properly defined, the policies are successful in stopping malicious scripts that try to leak secret data or alter its value, granting the security guarantees we desired.

The results are satisfying, as they supply the layer of security we wanted to provide, but it would have been preferable if less restrictions were imposed on browser add-ons and more tools were available to them, as they would have allowed for better usability. In the next section we analyze what these limitations are and how loosening them would give us the chance to improve the extension.

7.1 Limitations

Although Michrome gives a number of security guarantees, it is subject to some limitations which we now analyze.
Due to the restrictions on browser extension, it is impossible to distinguish two scripts running in the same page.
This means that if we trust a page with managing a certain cookie and include it in the policies so that it may do so, if an attacker's script is running in that page, it might be able to steal the cookie, as there is no way to stop the malicious

script while allowing legit ones to execute.
Furthermore, if the user accidentally installed a malicious browser add-on, preventing it from causing harm is outside the possibilities of our extension.

Michrome changes some cookies' names: this means that, if the user disables it and continues to use the browser afterwards, some cookies may no longer be recognizable, so there might be need to authenticate or change settings again in the associated websites.
Similarly, if the extension is enabled after the creation of cookies, it might be unable to relate them to the corresponding policies, as the identification suffixes will be missing. In general, it is not a good idea to disable (or enable) it in the middle of a session.
Note that conflicts may occur in case other add-ons that manage cookies are installed, as they might be confused by the identification suffixes, or they might create domain cookies without the suffix that will not properly be handled by our extension. Also, issues can arise if another extension wraps the cookie *getter* or *setter* methods.
This is again due to the fact that the possible cross-extension interactions are minimal.

7.2 Future Works

The extension acts as an additional layer of security, capable of stopping some attacks that would otherwise steal cookies or corrupt their values, but it is not to be intended as a shield impenetrable to any attacks.
We hope that in the future Chrome, and browsers in general, will implement new APIs that give extensions more possibilities, as in the current situation even some very basic actions such as reading a file can be unnecessarily

complicated and require unexpectedly long times to find a working solution. As a result Michrome adopts some pretty convoluted and counterintuitive techniques to reach its goals, and is overall hard to use, especially for those who are not familiar with domains, cookies or other concepts related to the extension.

It would be desirable to see more synchronous APIs, especially for message passing or storage access, as it would make it easier to exchange information between extension components. Asynchronous methods definitely help keeping the performance impact of add-ons to a minimum, but if no simple synchronous equivalent is available when there is need for it, finding an alternative is time-consuming and often leaves no choice but settling for an intricate solution that might need to abandon some features and even results in performance degradation.
In particular, communicating from the background page to content scripts in a synchronous way is something that would have allowed us to implement a simple click-to-add policy creation system, and also supply the tools for managing policies in a cleaner and more convenient way.

Equally useful would be being able to find a cookie's domain in a simple, synchronous way: it would make the renaming of their keys unnecessary and some of the possible conflicts and issues described in previous paragraphs would disappear.

8 References

Many references to the official documentation of APIs, which describes their mode of operation, have been left out to avoid the cluttering that would have resulted from an excessive use of footnotes, which would have also been pointless as they often refer to the same website.

We hence list the links to the documentation, from where the pages corresponding to the specific API of interest can be accessed, but also a number of other websites that were helpful in general in the processes of studying and developing.

W3Schools, website containing documentation, tutorials and examples to help developing for the web:
http://www.w3schools.com/

A list of Web APIs with the corresponding documentation by *Mozilla*:
https://developer.mozilla.org/en-US/docs/Web/API/

The extensions section of *developer.chrome.com*, containing a large amount of information, ranging from explanations on how browser extensions work and how to develop one to documentation for numerous Web Extension APIs:
https://developer.chrome.com/extensions

Tutorials Point, a website containing various tutorials for web development:
http://www.tutorialspoint.com/index.htm

Stack Overflow, an online community of programmers where questions are asked and answered, helpful in understanding what is not properly explained or sometimes completely absent from official documentation: http://stackoverflow.com

Last is a list of interesting and related papers on the subject of web security, that have been consulted during the writing of this document.

[1] R. Focardi, S. Calzavara, N. Grimm, M. Maffei, "Micro-Policies for Web Session Security", to appear in Proceedings of the 29th IEEE Computer Security Foundations Symposium, June 27 – July 1, 2016, Lisboa, Portugal

[2] R. Dhamija, J. D. Tygar, M. Hearst, "Why Phishing Works", Harvard University, UC Berkeley, in Proceedings of CHI-2006: Conference on Human Factors in Computing Systems

[3] C. Dougherty, "Practical Identification of SQL Injection Vulnerabilities", Carnegie Mellon University, United States Computer Emergency Readiness Team, 2012

[4] A. Klein, "Cross Site Scripting Explained", Sanctum Security Group, June 2002

[5] W. Zeller, E. W. Felten, "Cross-Site Request Forgeries: Exploitation and Prevention", Princeton University, October 18th 2008

[6] Y. Zhou, D. Evans, "Why Aren't HTTP-only Cookies More Widely Deployed?", University of Virginia, 2010, Web 2.0 Security and Privacy (W2SP), Oakland, CA, May 20th 2010

[7] J. S. Park, R. Sandhu, "Secure Cookies on the Web", George Mason University, 2000, IEEE Internet Computing vol. 4, no. 4, July-August

[8] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, W. Joosen, "SessionShield: Lightweight Protection against Session Hijacking", Katholieke Universiteit Leuven, SAP Research – CEC Karlsruhe, 2011, Engineering Secure Software and Systems (ESSoS)

[9] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, "Automatic and Robust Client-Side Protection for Cookie-Based Sessions", Università Ca' Foscari Venezia, 2014 International Symposium on Engineering Secure Software and Systems (ESSoS)

[10] K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, "Language-based Defenses against Untrusted Browser Origins", INRIA Paris-Rocquencourt, Imerial College London, In Proceedings of the 22th USENIX Security Symposium, 2013