Securing Script-Based Extensibility in Web Browsers

Vladan Djeric, Ashvin Goel
University of Toronto

Abstract

Web browsers are increasingly designed to be extensible to keep up with the Web's rapid pace of change. This extensibility is typically implemented using script-based extensions. Script extensions have access to sensitive browser APIs and content from untrusted web pages. Unfortunately, this powerful combination creates the threat of privilege escalation attacks that grant web page scripts the full privileges of extensions and control over the entire browser process.

This paper makes two contributions. First, it describes the pitfalls of script-based extensibility based on our study of the Firefox web browser. We find that script-based extensions can lead to arbitrary code injection and execution control, the same types of vulnerabilities found in unsafe code. Second, we propose a taint-based system to track the spread of untrusted data in the browser and to detect the characteristic signatures of privilege escalation attacks. We evaluate this approach by using exploits from the Firefox bug database and show that our system detects the vast majority of attacks with almost no false alarms.

1 Introduction

Most web browsers today provide powerful extensibility features, including native and script-based extensions. Native extensions (or plugins) are typically used when performance is critical (e.g., virtual machines for Java, Flash, media players, etc.), while script extensions ensure memory safety and have the advantage of being inherently cross-platform and amenable to rapid development. Examples of popular script extensions include the Firefox Adblock extension [1] that filters content from blacklisted advertising URLs, and Greasemonkey [4] that allows users to install arbitrary scripts in web pages for customization or to create client-side mashup pages.

Script extensions must have access to both sensitive browser APIs and content from untrusted web pages. For example, Adblock must be able to access the local disk to store its URL blacklist and access web pages to filter their content. This combination is needed for writing powerful extensions, but it creates challenges for securely executing web page scripts. Specifically, when extensions interact with web pages, there is a risk of a privilege escalation attack that grants web page scripts the full privileges of script extensions and control over the entire browser process. Privilege escalation vulnerabilities are perhaps even *more* critical than memory safety vulnerabilities because script-based attacks can often be executed reliably.

Our goals in this paper are two-fold: 1) understanding the nature of script-based privilege escalation vulnerabilities, 2) proposing methods to secure the Firefox browser against them. Privilege escalation vulnerabilities are common in Firefox, and comprise roughly a third of the critical vulnerability advisories. They arise from unsafe extension behaviors or bugs in the Firefox security mechanisms that regulate interactions between trusted native or extension scripts and untrusted web page scripts. These vulnerabilities have appeared regularly in every version of the browser and exist even in the latest versions. This is despite continuing effort from a dedicated team of security developers that have progressively improved the browser security model.

The Firefox security model consists of a combination of stack inspection and one-way namespace isolation. The stack inspection mechanism, implemented at the boundary of the script and native code, regulates accesses to sensitive native interfaces based on the principals of the caller. For example, a local file access is denied if the current stack contains a frame associated with an untrusted principal.[1] Namespace isolation is used to enforce the same-origin policy for web page scripts. This policy limits interactions between scripts and documents loaded from different origins. The namespace isolation is *one way* in that script extensions

[1]A principal represents the code's origin and, for web page scripts, it consists of a scheme, host, port combination.

are privileged and allowed to access content namespaces, but web page scripts should not be able to obtain a reference to the privileged namespace. This policy is designed to enforce the same-origin policy and defend against privilege escalation attacks.

These security mechanisms are well understood, but they have two flaws: 1) relying entirely on principals as a measure of trustworthiness for stack inspection, and 2) depending on one-way namespace isolation to work correctly. In practice, an exploit can leverage browser bugs or vulnerable extensions to confuse the browser into assigning wrong principals to code or executing data or code with wrong principals, thus defeating stack inspection. Second, reference leaks can occur because of interactions between privileged and unprivileged scripts, compromising namespace isolation and allowing unprivileged scripts to affect the execution of privileged scripts. As a result, we find that arbitrary code injection and execution control vulnerabilities that commonly exist in unsafe code can also occur with script-based extensibility.

Based on the flaws described above, our solution for securing the Firefox browser consists of combining tainting with the existing stack-based security model. Our approach guarantees that tainted data will not be executed as privileged code. Tainting *all* data from untrusted origins and propagating the tainted data throughout the browser provides a much stronger basis for making security decisions. In essence, our attack detectors "second guess" the security decisions of the browser by taking into account one additional piece of information, i.e. the taint status. This solution is conceptually simple and well-suited for the browser's security model because namespace isolation already provides a security barrier between the taint sources in content namespaces and privileged code residing in extension namespaces. As a result, we show that it is unlikely that attacks will be detected erroneously, even if we fully taint all data and scripts from web pages.

The contributions of this paper are two-fold: 1) we analyze and classify script-based privilege escalation vulnerabilities in the commonly used Firefox browser, 2) we use taint-based stack inspection to design effective signatures for script-based exploits and evaluate this approach. We use Firefox version 1.0 for the evaluation because it has several privilege escalation vulnerabilities and easily-available exploits. Our results show that we can detect the vast majority of attacks with almost no false alarms and modest overhead.

Below, Section 2 provides background on the Firefox security model. Section 3 presents our classification of privilege escalation vulnerabilities and sample exploits. Section 4 describes our taint-based approach for securing script-based extensibility. Section 5 provides an evaluation of our approach. Section 6 describes related work in the area and Section 7 presents our conclusions and describes future work.

2 The Firefox Browser

In this section, we provide an overview of the Firefox architecture and its security model.

2.1 Architecture

Figure 1 shows a simplified version of the Firefox architecture relevant to this work. The basic browser functionality is provided by native C++ components written using Mozilla's cross-platform component model (XPCOM). XPCOM components implement functionality such as file and socket access, the document object model (DOM) for representing HTML documents, and higher-level abstractions, such as bookmarks, and expose this functionality via the XPIDL interface layer. The Script Security Manager (SSM) is an XPCOM component responsible for implementing the browser's security mechanisms.

The JavaScript interpreter accesses XPCOM functionality via the XPConnect translation layer. This layer allows the interpreter and the XPCOM classes to work with each others data types transparently. XPConnect also serves as the primary security barrier for enforcing the browser's same origin policy and restricting access to sensitive XPCOM interfaces.

Firefox's script extensions and privileged UI scripts, shown in Figure 1, are loaded from local files through URIs with the "chrome" protocol. They are privileged and have access to a greater number of XPCOM interface methods than web page scripts and are not subject to the browser's same origin policy. Similar to other browsers, Firefox also supports native plugins for Java, Flash, etc.

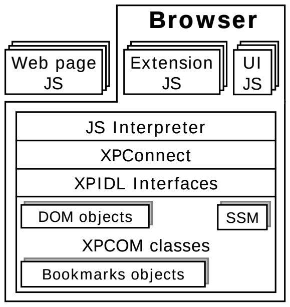

Figure 1: The Firefox architecture.

Although potential security vulnerabilities can exist within plugin implementations, we do not address them. However, with appropriate sandboxing of plugins [14, 23], we would be able to monitor any script interactions with the plugins.

2.2 Security Model

Firefox primarily uses two security schemes, namespace isolation and a subject-verb-object model based on stack inspection. Namespace isolation is used to enforce the same origin policy for web page scripts, and stack inspection regulates access to sensitive XPCOM components. We describe each in more detail below.

2.2.1 Namespace Isolation

The browser runs scripts within an object namespace that defines the objects available to the script. A window object lies at the root of the namespace for each web page. For example, web page scripts manipulate HTML by invoking the DOM methods of the document object that is a property of this window object.

The browser enforces the same origin policy by running web page scripts from different web pages in different namespaces. These scripts are only allowed to access other namespaces from the same origin (described below). Extension scripts are allowed to access all content namespaces. Extension namespaces are hidden from the web page scripts, and extensions are not expected to invoke web page scripts directly.

2.2.2 Subject-Verb-Object Model

Firefox uses a “Subject-Verb-Object” access control model. The subject is the principal of the currently executing code, the verb is one of a limited number of operations (e.g., call a function F, get a property A, set a property B), and the object is the principal of the object that is the target of the operation. This security mechanism is implemented in the Script Security Manager, and invoked by XPConnect to regulate access to sensitive XPCOM interfaces and by the interpreter to limit access to sensitive functions and object properties.

The principal of a web page script is defined by the origin of the document containing the script (its protocol, hostname, and port). The Script Security Manager determines the subject principal by walking down the JavaScript stack until it finds a stack frame with a script principal. The object principal is determined by walking up the object's parent chain (scope chain) in its namespace until an ancestor object with a principal is found. For web pages, the object's parent chain leads to a top-level HTML document associated with the window object.

3 Script-Based Privilege Escalation

Privilege escalation vulnerabilities are created by unsafe extension behaviors or bugs in the Firefox security mechanisms that regulate interactions between privileged and unprivileged code. In this section, we first discuss different classes of script-based privilege escalation vulnerabilities and then describe examples of real vulnerabilities.

3.1 Vulnerability Classification

Our analysis of the Firefox bug database revealed four main classes of privilege escalation vulnerabilities: code compilation, luring, reference leaks and insufficient argument sanitization. Most of the known Firefox vulnerabilities can be attributed to one or more of these classes.

3.1.1 Code Compilation Vulnerabilities

Similar to cross-site scripting (XSS) vulnerabilities that occur in web sites, code compilation vulnerabilities allow arbitrary strings from content namespaces to be compiled into JavaScript bytecode with privileged principals. Unlike a statically typed language such as Java, JavaScript allows arbitrary strings to be converted into byte code at runtime through `eval` and eval-like functions such as `setTimeout`. The `eval` function compiles a string into byte code and executes it with the principal of the calling script, even if the string was obtained from a different namespace. Code compilation vulnerabilities occur if attackers can trick privileged code into compiling strings supplied by the attacker, or if they can find bugs in the rules for assigning principals to newly compiled byte code. For example, it can be dangerous for privileged code to load URIs from untrusted namespaces as the URIs are capable of carrying script code inline. For example, the "javascript" protocol (e.g., `javascript:alert('Hello World');`) allows executing text after the protocol name as a script in the current namespace.

This problem may seem simple, but it has been the cause of several security bugs in Firefox. For example, even after vulnerable code was patched to sanitize URIs before loading them, exploits were possible because they did not account for nested URIs such as `view-source:javascript:`.

3.1.2 Luring Vulnerabilities

Luring vulnerabilities allow malicious scripts to trick privileged code into calling a privileged function of the attacker's choosing instead of the intended callee. Stack inspection prevents unprivileged scripts from calling the privileged functions directly, so malicious scripts must lure privileged code into making these calls.Luring is possible because script extensions routinely access DOM objects in content namespaces. These DOM objects are simply JavaScript wrappers for native XPCOM objects with well-defined, native interfaces. However, JavaScript's flexibility allows web page scripts to modify these wrapper objects. In versions of Firefox after 1.0.3, privileged code is protected by automatically created "safety wrappers" that hide any wrapper changes made by untrusted code. However, if the safety wrapper code contains bugs (as has often been the case), privileged code again becomes vulnerable to luring attacks.

In order to execute privileged code, an attacker can choose one of three possible kinds of callees: 1) an eval-like native function, 2) a privileged function accidentally leaked into the content sandbox (see next section), or 3) a privileged native method that legitimately exists in content namespaces. The third category consists of XPCOM methods that are visible to ordinary web page scripts because they are meant to be invoked by digitally signed web page scripts. For example, the `preference()` method of the `navigator` object allows privileged scripts to read or write the browser's configuration, such as the browser's homepage and security settings. Ordinary web page scripts cannot invoke the sensitive `preference()` method directly, but since every function is also an object in JavaScript, web page scripts can obtain an object reference to this method and potentially trick buggy privileged code into invoking the reference.

3.1.3 Reference Leak Vulnerabilities

Reference leak vulnerabilities occur when web page scripts gain access to references in the extension namespace [11]. These leaks are compromises in the isolation between privileged and unprivileged namespaces. They allow an attacker to modify data or code defined in a privileged namespace and call arbitrary functions within the privileged namespace, potentially leading to arbitrary execution control. Reference leaks are dangerous because privileged code that depends on namespace isolation may become accessible to web page scripts or it may become vulnerable to code compilation or luring attacks. Reference leaks can occur due to bugs in native code that deals with namespaces. Also, careless extensions may place references to privileged objects in an untrusted namespace. Finally, reference leaks can lead to cross-principal confidentiality violations, but we do not address confidentiality in this paper.

3.1.4 Insufficient Argument Sanitization

Vulnerabilities can also occur if a browser extension uses unsanitized data from untrusted namespaces as arguments to privileged XPCOM APIs. For example, if an extension used to download Flash videos from web pages uses the name of the movie file on a web page as part of the local filename to which the file is saved, it may be open to directory traversal attacks (e.g., using "../" to access normally inaccessible directories) that would not be detected by the browser's stack inspection mechanism. If the overwritten file were an extension JavaScript file, it would lead to a privilege escalation attack. This specific class of vulnerability has not been documented in the Firefox bug database, but we consider it a

```
onLinkIconAvailable:  function(Href)
{
  if (favIcon && ...)  {
    favIcon.setAttribute("src",
      Href);
  }
}
```

Figure 2: Target code invoked when a LINK tag is found in the current web page.

```
evalCode = "clonedFunction = \
    function deliverPayload(){...}; \
    clonedFunction()";
myElem = document.getElementById
                ("myMarquee");
xbl_object = myElem.init.call;
eval(evalCode, xbl_object);
```

Figure 3: Exploit code that allows untrusted functions to be associated with privileged principals.

likely vulnerability for extensions.

3.2 Examples

We describe some examples of privilege escalation vulnerabilities from the Firefox bug database to show that these vulnerabilities can be subtle and easy to overlook.

3.2.1 URI Code Injection

Figure 2 shows an example of browser JavaScript containing a code compilation vulnerability that can lead to URI code injection (Bug 290036). This GUI code displays a favicon (16x16 pixel icon) image next to the browser's URL bar. Normally the icon's URI, which is specified by the current web page, would be the HTTP address of the favicon image, but a malicious web page can specify a "javascript" protocol URI. When the privileged UI code attempts to load the image by setting the `src` property of the icon container to the `Href` URI, it will inadvertently execute script code. This code will be compiled with the unprivileged principals of the URI, but it will have access to the privileged UI namespace, allowing reference leaks, which can then be used for other attacks (e.g., see Section 3.2.4). This vulnerability occurs because the native code implementing the icon container and the compilation function are unaware of the origin of the `Href` argument.

3.2.2 Compilation with Wrong Principals

Figure 3 shows code that exploits a code compilation and a reference leak vulnerability to create a dynamically-defined function (`clonedFunction`) with elevated privileges. The `eval` function compiles and executes the `evalCode` string with the unprivileged principal of the web page. However, the attacker has also supplied a second argument that specifies the namespace for name resolution during the string evaluation. Normally, this argument does not cause a problem because it belongs to the same namespace as the caller's namespace. However, `xbl_object` is a benign reference leak from a privileged namespace.

Exposing `xbl_object` is a reference leak, but it is not sufficient for an attack because the interpreter invokes `eval` with the correct caller's principals. However within `eval`, once run, the `evalCode` byte code gets access to a privileged namespace. This access by itself is still not a problem because `evalCode` runs with the web page principals, and thus will not be able to get past the stack inspection checks. Similarly, invoking `deliverPayload` directly within `evalCode` would not be problematic.

The exploit occurs when `evalCode` creates a function referenced by `clonedFunction`. The interpreter creates a new function object in the privileged namespace that is a clone of `deliverPayload`. When a function is created by cloning, its principal is set to its object principal, as described in Section 2.2.2. When the cloned function is invoked, it executes its payload with elevated privileges. In effect, this exploit attaches a user-supplied function to a privileged namespace, making it appear privileged to the security manager. This vulnerability occurs because the implementation of `eval` did not check that it was compiling code from one principal and executing it within the namespace of a more privileged principal.

The patch for this vulnerability added a check to `eval` to ensure that the principal of the caller subsumes the object principal of the second argument. However, it was discovered that this patch could be bypassed by invoking `eval` indirectly using the timer method `setTimeout`. When the natively-

```
var code = "... payload ...";
document.body.__defineGetter__
  ("localName", Script(code));
```

Figure 4: Simplified exploit code for Bug 289074.

implemented timer fires, there are no JavaScript frames left on the stack, so the caller's principal is the fully privileged principal of the native timer code. The next patch prevented `eval` from being called directly by native code. Further patches were needed to fix other attacks on `eval`.

3.2.3 Luring Privileged Code

Figure 4 shows the exploit code for a luring attack. This exploit would trigger if the `document.body.localName` property is read by the UI code. This code tricks the privileged code into working with a different property than the one it expects by associating a getter function with a native DOM object property (`localName`). Furthermore, the `Script` object behaves like an eval-like function that allows strings to be precompiled and executed with the privileges of the caller's principal.[2] The consequences are equivalent to privileged JavaScript executing a string of the attacker's choosing, although no code is compiled in the privileged namespace. This vulnerability occurs because the caller accesses an overridden property.

This problem was so widespread in Firefox 1.0 that it motivated developers to implement the "safety wrapper" mechanism that allows privileged scripts to work with native DOM objects without being exposed to any modifications made by web page scripts. However, even the latest releases of Firefox continue to suffer from bugs in assigning wrappers, thus allowing privileged scripts to interact with tampered DOM methods and properties [6].

3.2.4 Privileged Reference Leaks

Figure 5(a) shows code that exploits a reference leak vulnerability in the QueryInterface XPCOM object. A flaw in the XPConnect code for setting up safety wrappers for native objects inadvertently sets a privileged object as the prototype of the safety wrapper for QueryInterface in untrusted namespaces. Malicious code can use this leak to reach the global

[2]This Firefox-specific object has been deprecated since Firefox 3.0, presumably due to security risk.

```
var leaked =
  QueryInterface.__proto__.__parent__;
var cid = {equals: Script(payload)};
leaked.foo.getClassObject(cid);
```

(a) Simplified exploit code.

```
var foo = {
  getClassObject: function(aCID) {
    if (aCID.equals(value))
      return this._objects[key];
  }
};
```

(b) Simplified target code.

Figure 5: Exploit and target code for Bug 294795.

object of a privileged namespace. The exploit calls the script method `foo.getClassObject` in the privileged namespace with a specially-crafted argument to carry out a luring attack.

The `getClassObject` method shown in Figure 5(b) relies on namespace isolation and thus expects to be called from other privileged functions with safe arguments. However, when it calls the `equals` method of its `aCID` parameter, it inadvertently invokes the Script object defined by the attacker, executing it with full privileges.

3.2.5 Loading Privileged URIs

There are also attacks that use a combination of a bug that allows unprivileged pages to load higher privilege documents (e.g., "chrome" protocol URIs) and a cross-site scripting (XSS) bug to inject their own scripts into these pages. Bug 306261 allowed untrusted pages to bypass restrictions on loading privileged URIs of the "about" protocol (which allows setting browser configuration values) by using a malformed URI. We do not address XSS bugs or violations of URI loading policies, but our system is able to detect this category of attacks because it leads to code injection.

3.3 Comparison With Memory Safety

JavaScript extensions have many clear benefits, but they suffer from risks posed by these four classes of vulnerabilities. As a result, Firefox users have been victims of real-world privilege escalation attacks and the Firefox bug database shows that the incidence rate for these types of vulnerabilities is comparable to memory-safety vulnerabilities (more on this in Section 5.1).

At first, this may seem counter-intuitive: components written in a memory-safe, interpreted language should be more secure than their native equivalents. This intuition may be true in single-principal applications, but Firefox must execute JavaScript from multiple principals concurrently and must arbitrate over many possible interactions, which raises the specter of bugs leading to privilege escalation attacks.

In fact, the classes of vulnerabilities we found for the multi-principal Firefox script environment are similar to memory-safety vulnerabilities found in single-principal native code. The code compilation vulnerabilities are not unlike buffer overflows: data is executed as code, allowing for arbitrary code execution. The luring vulnerabilities allow attackers to call existing functions of their choosing, similar to return-to-libc attacks [5].

4 Approach

Script-based extensibility in the Firefox web browser is a powerful feature and is highly valued by its users. However, it leads to privilege escalation vulnerabilities precisely because of the dynamic and flexible nature of the script language used to implement the extensions. The language features allow leveraging browser bugs or vulnerable extensions to confuse the browser into assigning wrong principals to code, thus bypassing stack inspection.

Privilege escalation vulnerabilities also arise because Firefox's implementation of one-way namespace isolation is inherently error prone. The browser fully trusts script extensions, but these scripts can interact with data from unprivileged sources in unsafe ways, compromising namespace isolation. One-way namespace isolation will not disappear from extensible browser architectures, as extensions will always need to read and modify untrusted web pages. One method of improving the security of one-way namespace isolation is to provide stronger isolation guarantees. For example, Google Chrome [10] divides an extension into separate processes, one for for accessing privileged interfaces, and another for interacting with untrusted web pages, while only allowing IPC between the two processes. This architecture requires increased implementation effort from the extension developer and is completely incompatible with the Firefox extension model.

Instead, our solution is to use tainting to augment the browser's security mechanisms. We use tainting because it helps detect when untrusted content can affect privileged code. Furthermore, it is fully compatible with the current Firefox extension model. Unfortunately, many tainting-based systems suffer from endemic false alarms and thus are unusable in practice [18]. In this section, we show that our tainting-based solution, while being conceptually simple, is well-suited for the browser's security model because namespace isolation already provides a security barrier between the taint sources in content namespaces and privileged code in extension namespaces.

4.1 Threat Model

We define a privilege escalation attack as tainted data executing as privileged code. Tainted data is executed as privileged code if it is compiled into script byte code tagged with the wrong principals, or if tainted data is used as a reference to execute privileged code. Both scenarios lead to a failure of the browser's security mechanism for guarding access to sensitive interfaces, allowing untrusted web pages to gain the ability to modify the host system.

We add security checks and augment stack inspection to look for the characteristic signature of privilege escalation attacks. To do so, we rely on the memory safety of the browser as well as the browser's ability to correctly assign a principal to a web page when it is *first* loaded, before any web page scripts begin executing. Assigning this principal is straightforward as it only depends on the web page's URI. We do not depend on the correctness of the rest of the code that assigns principals, or code that interprets principals. Instead, we "second guess" browser security code by auditing its security decisions with the additional taint status information.

4.2 Tainting

We consider all documents fetched from remote sources or local documents opened with the "file" protocol as untrusted and taint them because the browser does not assign them a privileged principal. When documents are loaded into the browser, they are parsed into a tree of native DOM objects, representing individual markup elements and their attributes. All nodes of the tree are individually marked tainted, including the text of any scripts defined inside the document, such as in event handlers or in SCRIPT tags, and taints are tracked separately for each attribute of a DOM element.

Our tainting system uses *different* policies based on the privilege level of the executing script. Unprivileged code is completely untrusted and may be malicious, so we must unconditionally taint all script variables created or modified by executing scripts originating from untrusted (tainted) documents. For privileged scripts, we use standard taint propagation rules that mark the output of JavaScript instructions as tainted when the instruction inputs are tainted. Tainting allows us to mark and track the influence of untrusted code throughout the browser.

Tainting systems can suffer from excessive false alarms when using control-dependent tainting. Control-dependent tainting taints the output of any code whose execution depends on tainted data. For example, all outputs of an if-branch would be tainted if the condition variable were tainted. Control dependence is necessary when the code processing the tainted data may itself be malicious. For example, detecting cross-domain information leaks requires accounting for implicit flows, since malicious web page scripts could leak information [19]. We do not use control-dependent tainting on the privileged side because we assume that the privileged scripts are trusted. We consider it highly unlikely that privileged script code would accidentally launder taints through control flow and then execute the laundered data as privileged code.

It is necessary to track taint both in the native code and inside the script interpreter. For example, when a new HTML document is loaded into a tab, privileged UI script code reads the tainted document's title property and sets it as the caption of the tab element. This requires taints from native DOM objects associated with the HTML document to propagate to script variables in the UI code and then back to DOM objects associated with the UI document. On the native side, we track the taint status of string properties of XPCOM objects. Tainting code in XPConnect taints any JavaScript references to unprivileged DOM elements and propagates taints between the XPCOM and JavaScript environments.

4.3 Attack Detection

We define a privilege escalation attack as tainted data executing as privileged code. We implement two classes of attack detectors to detect this condition: compilation detectors and invocation detectors. Compilation detectors ensure that tainted data is never compiled into byte code tagged with privileged principals, while invocation detectors monitor the stack for tainted references to function objects creating privileged frames. Compilation detectors map closely to code compilation vulnerabilities, while invocation detectors are best suited for preventing luring attacks.

4.3.1 Compilation Detectors

We use compilation detectors as a proactive measure to prevent tainted data from being compiled to privileged byte code, even if it is never executed. These detectors are well suited for securing eval-like functions that compile strings into byte code, because the string's taint status informs these functions of the string's origin. These detectors allow defending against compilation bugs such as the wrong principal attack (see Section 3.2.2). If native XPCOM code compiles the strings, as in the URI code injection attack (see Section 3.2.1), or the XSS attacks (see Section 3.2.5), the detectors will use the taint status of XPCOM string objects to detect and prevent exploits. Our compilation detectors are placed before all calls to compilation functions, such as those defined by the JavaScript API.

4.3.2 Invocation Detectors

Invocation detectors monitor script execution for situations where tainted references to script or native functions are invoked inside the interpreter and result in the creation of privileged stack frames. This policy catches luring attacks in which privileged scripts are tricked into invoking functions of the attacker's choice. It also detects when an unprivileged script uses a reference leak to directly call a privileged JavaScript function from an extension.

The invocation detectors vary depending on whether the invoked functions are scripted or native.

Namespace isolation limits script functions to calling other script functions within the same namespace. Therefore, our detectors watch for namespace pollution, namely callers invoking tainted function references that result in a privileged callee stack frame, as in the luring attack (see Section 3.2.3). This detector is able to intercede before any function code is executed with elevated privileges.

For native functions, it is not as straightforward to come up with a policy for detecting attacks. It can be perfectly safe for privileged scripts to invoke natively defined methods of tainted object references. For example, an extension script could call the native `toLowerCase` string method on a web page's title string. The reference to the title string will be tainted, and the function reference to the `toLowerCase` method will also be tainted because it is accessed as a method of a tainted string, but this operation should not raise a privilege escalation alert because, in and of itself, it does not represent a privilege escalation threat even if it is called from a privileged context. However, if the native function called through the tainted reference is a native XPCOM method that is only accessible to privileged callers, then a security violation needs to be raised as it indicates a luring attack.

Thus, it is important to know whether the native callee is sensitive and whether the caller will be interpreted as privileged. We get this information by letting the call proceed, and if it reaches XPConnect, the security manager establishes the sensitivity of the target XPCOM method or property and performs a stack inspection to determine the effective subject principal of the caller. We augment the security manager to signal an attack whenever it computes a privileged subject principal, but a tainted function reference is found on any stack frame during the stack walk.

4.3.3 Reference Leaks

As demonstrated in Section 5, we can detect and stop the vast majority of proof-of-concept exploits in the Firefox bug database based on reference leaks. We achieve these results by detecting attempts to directly invoke or lure privileged code with our invocation detectors, as in the reference leak attack (see Section 3.2.4), and by detecting malicious attempts to compile tainted strings with our compilation detectors. However, we are unable to detect and prevent reference leaks. For example, in Figure 5(a), we cannot rely on the object reference's taint status to detect the privileged reference leak, because our tainting rules require that properties of tainted objects, such as QueryInterface, also be marked tainted.

Although we do not prevent reference leaks, attacks employing reference leaks will not be able to escape our tainting. Any data modified by untrusted scripts is still marked tainted, and invoking or compiling tainted data will trip the detectors. Therefore, attackers will not be able to mount a privilege escalation attack, in which untrusted data is executed as privileged code. At most, if the reference leak allows access to arbitrary global variables in the privileged namespace, attackers may be able to devise control dependent attacks and compromise the integrity of extension logic.

Barth et al. [11] propose a system for detecting reference leaks between different security origins. Although their work aims to prevent cross-origin attacks made possible by reference leaks, it could also be integrated with our system to detect reference leaks from privileged namespaces. We should note that reference leaks are not a requirement for mounting luring attacks. As previously described in section 3.1.2, the target of any luring attack can also be a call to an eval-like function (such as the Script object) or a reference to a sensitive method of an XPCOM object legitimately present in the content namespace.

4.3.4 Unsafe XPCOM Arguments

We are currently conducting a study to determine the extent of this class of vulnerability. We plan to create a list of sensitive parameters of security-sensitive XPCOM interfaces known to the security manager to mitigate the threat of tainted XPCOM arguments. We would need to provide untainting functionality to allow privileged scripts to indicate that a tainted argument has been sanitized. Other systems, such as Saner [9], allow validating sanitization routines.

4.4 Implementation

In this section, we describe the implementation of our tainting system in the JavaScript interpreter and the XPCOM classes and our attack detectors. In our system, we are most concerned about the taint status of strings and function references because privilege escalation attacks require either luring privileged code or compiling attacker strings. We chose not to use an existing system-level tainting solution because control dependent tainting is not required in our system and low-level tainting systems tend to produce a large number of false positives.

4.4.1 JavaScript Interpreter

JavaScript tainting requires associating a notion of taint with each script variable. JavaScript variables can hold the values of primitive data types such as booleans and integers, or they can hold references to heap allocated data, such as objects, strings, and doubles (hereafter collectively referred to as "objects"). All accesses to object variables are done by reference. We transparently convert all tainted primitive variables to doubles (a reference type) so that our tainting code exclusively deals with reference types. For reasons which we will discuss shortly, we do not taint the actual heap object pointed to by the reference (e.g. the floating point value of a double variable), but instead we only ever taint the individual references (pointers). For example, it is possible to have both a tainted and an untainted reference (pointer) to the same string. Therefore, variables of all data types are tainted in the same way, i.e. by tainting individual references.

When we implemented our tainting system, we had a choice between associating taint status with objects or with references to objects. We believe that it is a mistake to associate taint with objects because objects can be safely shared across privileged and unprivileged namespaces. For example, if a string variable were to be defined in a privileged namespace and then assigned to a variable in an unprivileged namespace, and unprivileged code were then to copy it into another variable, the original reference and the copy should not have the same taint status although they reference the same heap object. The value of the copied variable was clearly influenced by untrusted code, whereas the original variable was not. Note that strings and doubles are immutable, so there is no risk of modification by untrusted code. In other words, whenever a string or a double is modified, a new object is created with the new value and the original remains unchanged. For mutable JavaScript objects, our policy is to taint individual property references when they are modified by untrusted code. If we were to taint by object instead of by reference, we would run the risk of excessive, unnecessary taint propagation. For example, if an extension stores a tainted value in a property of a commonly used object, the object itself would become tainted. Therefore, any existing fields or methods of the object would also become tainted without receiving any tainted data. Such tainting could lead to false positives. The most egregious example of such unnecessary taint proliferation occurs when an extension copies a tainted variable into its global namespace, which is itself an object. Tainting the global object instead of merely tainting the property reference would unnecessarily taint all existing variables in the trusted extension namespace.

Therefore, we implemented variable tainting by storing a taint bit inside each variable. Internally, JavaScript variables are a machine word with a few of the least significant bits reserved for a type tag used for dynamic typing. We set aside an extra bit in the type tag for the taint status. The upper bits of primitive variables contain the variable's value, while the upper bits of references contain a pointer to a memory-aligned heap object. A downside of our reference tainting approach is increased memory use because heap objects now have to align at bigger boundaries. Specifically, we can store half as many JavaScript objects within a single memory page. This may seem like a large overhead for our approach, but the heap-allocated data structures are very small because the data structures use *unaligned* pointers to point to their actual contents. For example, the aligned, heap-allocated string data structure consists of two member variables: the string length and a pointer to an *unaligned* character array stored elsewhere on the heap. In practice, we find the overhead is not significant because JavaScript heap memory accounts for only a small portion of the Firefox memory footprint. Empirical measurements confirm that the increase in Firefox's data resident set size is less than 10% in everyday browsing, even on JavaScript-heavy sites such as GMail.

We added code to propagate taint between the inputs and outputs of each of the 154 opcodes in the JavaScript interpreter as well as code to unconditionally taint all outputs produced by unprivileged scripts. In addition to the aforementioned data types, scripts can also make use of a number of built-in objects and top-level properties and functions defined by the JavaScript language. Some built-in objects provide more advanced data types such as the "Date" and "Array" objects, while other built-ins provide utility functionality such as the "Math" object and the "encodeURI" function. Instead of painstakingly modifying each of these methods and functions individually to propagate taints, we conservatively taint the return values from any built-in function or method if any supplied arguments are tainted. For example, the returned values from `Math.sqrt(X)` or `encodeURI(X)` will be tainted if `X` is tainted. Finally, we had to make a few manual changes in the interpreter code to prevent loss of taint. For example, object references were sometimes converted into raw pointers and then the same raw pointers were converted back into object references without restoring the taint bit in the type tag.

4.4.2 XPCOM

We track the taint status of string objects in the XPCOM code because it is possible for native and interpreter code to compile strings into attack code. We also pay special attention to tracking taint in DOM string properties as these properties are the initial taint source and a very common taint sink.

We have borrowed the XPCOM string-tainting implementation from Vogt et al. [19]. This implementation adds taint flags to XPCOM string classes and modifies string class methods to preserve taint. We extended it to more string classes and made a small number of manual changes to account for the taint laundering that occurs in the code base when raw string pointers are extracted from string objects and used to create new string objects.

The XPCOM implementations of markup elements, representing the contents of the browser UI and web pages, do not store all their string properties within XPCOM string classes. The string properties of these DOM elements are a significant source and propagation vector for tainted data, so we needed to associate each string property of a DOM element with a taint status. To this end, we modified a small number of base classes from which DOM elements of all types are derived. DOM classes redirect calls to get or set individual properties to a handful of methods in these base classes, allowing us to add taint-propagation behavior and to automatically taint string properties of elements in unprivileged documents.

Adding taint tracking for every type of XPCOM property is difficult because there is no elegant way to associate taint status with primitive data types in the native XPCOM code. However, it is straightforward to taint all script references to unprivileged DOM objects. We added a taint bit to the "wrappers" used to reflect XPCOM objects into the JavaScript environment as well as the wrappers used to reflect JavaScript objects into XPCOM code. The first time XPConnect is asked to reflect a given object between the two environments, it creates a new wrapper object in the destination environment. For wrappers around XPCOM objects, we alter the wrapper creation process to check whether the wrapped object is a DOM node and if so, if it belongs to an unprivileged document. When the wrapper is placed in a JavaScript namespace, we make sure its object reference is tainted. The tainting rules in the interpreter automatically taint the values obtained from reading tainted objects' properties, effectively tainting all string and non-string properties of unprivileged DOM elements. Similarly, when a JavaScript object or function reference is wrapped for the XPCOM environment (e.g., a JavaScript callback function), we make sure its taint status is preserved and therefore propagated during a property read or a function call.

4.4.3 Attack Detectors

Once we determined the detection policies described in sections 4.3.1 and 4.3.2, implementation of the attack detectors became straightforward. The compilation detector code was added to the native functions that turn strings into bytecode (such as "eval"), while the invocation detector code was added to the code that implements JavaScript function calls. The only challenge was in finding the appropriate sites to install the detectors so that all JavaScript compilation and function invocations could be audited. The detectors had to be close enough to the low-level compilation and invocation code to intercept all the relevant call paths, but at the same time sufficiently high-level to easily retrieve principals and taint status.

5 Evaluation

We have implemented the approach described above in the Firefox browser. In this section, we evaluate our system by demonstrating its effectiveness against privilege escalation attacks. We start by showing how well it prevents attacks on known Firefox vulnerabilities. These vulnerabilities are documented in Firefox's Bugzilla bug database, which provides detailed security reports, proof-of-concept exploits and any available bug fixes. Next, we show that our system has minimal impact on normal usage by evaluating any false alarms that are raised and the performance overhead.

We evaluated against proof-of-concept attacks from Mozilla's bug database because the vulnerabilities are well cataloged and the proof of concept attacks are readily available. Most extension authors do not invest as much effort as Mozilla into documenting security issues in their code, thus making it difficult to evaluate our system against attacks on specific extensions. However, the same vulnerabilities could be leveraged against extensions.

We have implemented our system on Firefox version 1.0.0, which we use for all the experiments. We chose this version because it has the largest number of known privilege escalation bugs, allowing more extensive testing of our system. Also, the Firefox security team has a policy of embargoing reports for recent vulnerabilities, except for exploits already available in the wild. As a result, recent versions of Firefox have far fewer available privilege escalation exploits. For example, as of the end of 2009, the current version of Firefox (v3.5) has several privileged escalation vulnerabilities as shown below but no publicly available exploits for them. We plan to port our system and evaluate our results for newer versions of Firefox as exploits become available in the bug database.

5.1 Vulnerability Coverage

Table 1 shows the continuing threat posed by privilege escalation (PE) vulnerabilities in the Firefox browser. This table shows the total number of critical vulnerabilities and the number of critical PE vulnerabilities in the various major versions of the browser. The last column shows the percentage of PE vulnerabilities. Most PE vulnerabilities are generally classified as critical, and thus we do not show the statistics for non-critical vulnerabilities. Table 1 shows that PE vulnerabilities comprise 2/3 of all critical Firefox 1.0 vulnerabilities. All other versions continually have about 1/3 PE vulnerabilities. The main reason is that Firefox 1.5 implements safety wrappers that limit the opportunities for unsafe interactions between privileged code and web content, as described in Section 3.2.4.

Firefox	**Critical**	**Critical PE**	**%**
Version 1.0	27	18	67
Version 1.5	44	13	30
Version 2.0	43	16	37
Version 3.0	30	8	27

Table 1: Vulnerability Statistics.

Table 2 shows all the 19 privilege escalation advisories affecting Firefox 1.0.0, with some advisories containing multiple bug reports. Note that there are 26 such advisories in Firefox 1.0 (of which 18 are critical as shown in Table 1), but the other seven do not run on Firefox 1.0.0 and so we are unable to reproduce them. We were unable to test our system against 5 out of the 19 advisories because exploits were not available for them. The last column shows the types of vulnerabilities exploited in each advisory. For reference leaks, we also show whether the leak is leveraged to compile code (C) with the wrong principals or execute a luring attack (L).

Our system guards against 13 out of the 14 vulnerabilities described in the advisories. We do not detect an attack on the vulnerability in advisory #6. In this attack, an untrusted HTML string is parsed by the HTML parser to generate new HTML elements in a privileged document. Currently, we lose taint because we have not implemented taint propagation within the HTML parser.

5.2 False Positive Evaluation

We also tested our system by installing the top 10 most popular extensions that were available for Firefox 1.0.0, and then we manually browsed the Web. These extensions are Adblock Plus, FoxyTunes, NoScript, Forecastfox, Add N Edit Cookies, PDF Download, StumbleUpon, 1-Click Weather,

#	Advisory	Advisory Name	Type of Vulnerability	Detection
1	2006-25	Privilege escalation through Print Preview	Compilation	Yes
2	2006-16	Accessing XBL compilation scope via valueOf.call()	Leak (C)	Yes
3	2006-15	Privilege escalation using a JavaScript function's cloned parent	Leak (C)	Yes
4	2006-14	Privilege escalation via XBL.method.eval	Leak (C)	Yes
5	2005-56	Code execution through shared function objects	Leak (C), Leak (L)	Yes
6	2005-49	Script injection from Firefox sidebar panel using data://	Compilation	No
7	2005-44	Privilege escalation via non-DOM property overrides	Luring	Yes
8	2005-43	"Wrapped" javascript: URLs bypass security checks	Compilation	Yes
9	2005-41	Privilege escalation via DOM property overrides	Luring	Yes
10	2005-39	Arbitrary code execution from Firefox sidebar panel II	Compilation	Yes
11	2005-37	Code execution through javascript: favicons	Compilation	Yes
12	2005-35	Showing blocked javascript: pop-up uses wrong privilege context	Compilation	Yes
13	2005-31	Arbitrary code execution from Firefox sidebar panel	Compilation	Yes
14	2005-12	javascript: Livefeed bookmarks can steal private data	Compilation	Yes
Embargoed, or exploit not available				
15	2006-24	Privilege escalation using crypto.generateCRMFRequest	N/A	N/A
16	2006-05	Localstore.rdf XML injection through XULDocument.persist()	N/A	N/A
17	2005-58	Firefox 1.0.7 / Mozilla Suite 1.7.12 Vulnerability Fixes	N/A	N/A
18	2005-45	Content-generated event vulnerabilities	N/A	N/A
19	2005-27	Plugins can be used to load privileged content	N/A	N/A

Table 2: Vulnerability Coverage.

MR Tech Toolkit and FLST. A user, who is not associated with the project, browsed the Web for 5 hours, specifically visiting the top 100 most heavily visited web sites, as ranked by Alexa [2]. The user interacted extensively both with the web sites as well as with the extensions (e.g., directly invoking extension functionality by setting preferences).

The user's testing caused one alarm. This alarm was caused by Forecastfox, which displays the current weather forecast for a city of the user's choice. When a user searches for his city while setting his preferences, Forecastfox queries `accuweather.com` for cities matching the search string. When the user selects his city from the search results, Forecastfox concatenates several strings together including the full city name fetched from the web site and `eval`'s this expression to set the city option. Since the city name string originates from an untrusted web page, and the expression is evaluated in a privileged context, the alarm is raised. This code is unsafe because if the web site were compromised, the browsers of all Forecastfox users could be exploited. After seeing this alarm, we researched and found that Forecastfox for Firefox 3.0 has removed the `eval` statement.

We also performed automated testing by writing a Web crawler extension for Firefox. The crawler extension takes as input a list of web sites to visit and directs Firefox to load any HTML or JavaScript links found in the web site in depth-first order and interacts with each loaded page in Firefox to mimic the behavior of a human user. On each page, the crawler chooses multiple events to send to the page (e.g. mouse clicks, key strokes) and fills out and submits any HTML forms. The crawler exercises the JavaScript in the browser UI by performing one of several scripted GUI actions such as viewing the web page's HTML source code. We also installed AdBlock and Flashblock extensions and had the crawler randomly add and remove AdBlock filters on each page visited. The full crawler test visited 100 pages from each website in the Alexa Top 200.

The automated testing resulted in the discovery of one false positive, triggered by selecting "Page Source" from Firefox's "View" menu. The offending UI JavaScript retrieves a (tainted) reference to a window object from the content namespace. The window object implements multiple interfaces and some of these are sensitive interfaces inaccessible to web page scripts. The UI script casts the reference to the window object to a sensitive interface, further propagating the taint. When the privileged code calls a sensitive method of this interface through the tainted reference, our detectors flag it as a luring attack. This is not likely an exploitable vulnerability, but it would be safer if privileged JavaScript obtained references to sensitive interfaces without going through a content namespace.

While our testing is limited to heavily visited web sites, we believe that our system will not generate many false positives with other web sites. We find that privileged scripts are careful when operating on untrusted data and they are selective about the strings they compile in their privileged context (i.e., compilation false positives). Second, namespace isolation works well enough in non-malicious environments, and thus it is difficult for privileged function references to become tainted (i.e., luring false positives). Similarly, web pages don't expect to have access to privileged references and thus are unlikely to access them legitimately (i.e., reference leak false positives).

5.3 Performance

During regular browsing, we did not notice any degradation in page load times or responsiveness. We also conducted experiments to quantify the performance overhead of our system. We ran the Dromaeo JavaScript Tests and the DOM Core Tests from Mozilla's performance test suite [3]. These tests are micro-benchmarks that measure 1) the performance of basic operations of the script interpreter, and 2) the performance of common DOM operations. Our experiments were run on Ubuntu 8.04 Linux on an Intel Core 2 Duo 2.4 GHz processor, with 2 GB of memory. Our browser had 28% overhead for the JavaScript tests and 32% overhead for the DOM tests. Although the overhead witnessed in these micro-benchmarks does not visibly influence the browsing experience, the overhead may become an impediment to the adoption of our system at a time when JavaScript performance is becoming a competitive feature for modern browsers. One possible research direction would be to investigate how to efficiently integrate our tainting system with the just-in-time compilation systems present in modern JavaScript engines.

5.4 Security Analysis

Our system effectively detects nearly all available proof-of-concept attacks with few false positives. Admittedly, these proof-of-concept attacks were not designed with our detection system in mind. In order to defeat our defenses, an attacker would need to find a means of removing taint from untrusted objects. It would be difficult to remove taint in the JavaScript interpreter as the tainting rules are straightforward. The most likely target for laundering taint would be the native XPCOM methods.

One possible way for the browser to lose taint is to store tainted objects outside the browser. For example, if a user saves a malicious URL string from a web page as a bookmark, the bookmark is stored in a bookmarks file and the URI's taint is no longer present when the browser is restarted. A second, more involved method may be to launder taint through XPCOM method arguments. The attack begins by tricking an extension into passing a tainted, privileged object (a luring target) to an XPCOM function. If this function then natively calls a privileged native method of the tainted argument, our system would not detect this as a luring attack. This is because the extension JavaScript did not directly invoke a privileged method through a tainted reference. Similarly, if an XPCOM function were to accept a tainted object as an argument but then return a different, but related untainted object, it may be accurate to say the taint was laundered. Note that in these examples, the arguments and return values could not be strings as taint is always propagated during XPCOM string operations.

Although laundering taint is theoretically possible within our system, our system greatly raises the bar for potential attackers. The attackers now not only need to find a privilege escalation vulnerability in the browser, they also require extension JavaScript that interacts with specific XPCOM methods in such a way as to launder taint from crucial attack variables.

6 Related Work

This work focuses on securely executing untrusted scripts by using taint-based stack inspection. Stack inspection is widely used by modern component-based systems, such as Java and Microsoft .NET Common Language Runtime, to ensure that remote code is sufficiently authorized to perform a security-sensitive operation. Wallach et al. [20] provide instructive background on stack inspection.

Taint analysis helps determine whether untrusted data may influence data that is trusted by the system. Newsome and Song [16] use dynamic taint analysis to taint data originating or derived from untrusted network sources. An attack is detected when tainted data is used in a dangerous way, such as overwriting a return address. We use a similar approach to ensure that dirty data is not executed in a trusted context. Vogt et al. [19] use script tainting in a browser to track sensitive browser data, such as browser cookies or the URLs of visited pages.

The same origin policy is the basic sandboxing method used by web browsers. An effective method for implementing the same origin policy is script accenting [12], which uses simple XOR encryption to ensure that code is loaded and run, and data is created and accessed, by the same principal. Several recent projects [22, 17] attempt to enforce the same origin policy by separating different origins into different processes. In order to adopt this architecture, the extension model needs to be redesigned to accommodate extensions' interactions with pages from different principals [10]. The same origin policy is too strict for mashup web applications. For such applications, Mashup OS provides abstractions to allow limited communication while protecting the different principals associated with mashup content [21]. Interestingly, Mashup OS introduces the same set of problems as privileged extensions interacting with untrusted content and thus would benefit from our solution.

In concurrent work, Barth et al [10] propose a new browser extension model for Google Chrome. Extensions and web page scripts are isolated using processes and "isolated worlds" so that they never exchange JavaScript pointers. This architecture raises the bar for perpetrating a successful privilege escalation attack as multiple components now need to be compromised. Their design has obvious advantages, but the threat of privilege escalation attacks has not been completely eliminated. For example, Google recently fixed a vulnerability that incorrectly allowed JavaScript to be executed in the context of a Chrome extension [7].

Since browser extensions typically run with unrestricted privileges, a malicious extension can serve as a powerful attack vector. Louw et al. [15] propose access control for limiting extension privileges. For example, certain extensions may not be allowed access to the password manager. Dhawan and Ganapathy [13] propose adding an information-flow tracking system to Firefox to assist in determining whether a JavaScript extension maliciously compromises browser confidentiality or integrity. Although we are also interested in misuses of low-integrity data, their system is not an online attack detector and it requires human analysis.

Recent versions of Firefox use security wrappers (e.g., XPCNativeWrappers, XPCChromeObjectWrappers, etc.) to regulate interactions between JavaScript and XPCOM objects from different namespaces [8]. Unfortunately, implementation bugs in creating and manipulating wrappers are fairly common. Our system adds another layer of security on top of wrapper techniques by effectively second guessing wrapper security decisions.

7 Conclusion

Script-based privilege escalation attacks are a serious and recurring threat for extensible browsers such as Firefox. In this paper, we describe the pitfalls of script-based extensibility in Firefox and show that the privilege escalation vulnerabilities are similar to arbitrary code injection and execution control vulnerabilities found in unsafe code. Then, we propose a tainting-based system that specifically targets each class of vulnerability. We implemented such a system for the Firefox 1.0 browser and our evaluation shows that it detects the vast majority of attacks in the Firefox bug database with almost no false alarms and moderate overhead.

Our vulnerability classification and our proposed defense system are inevitably linked to the Firefox browser. However, one-way namespace isolation must exist in browser extension architectures because extensions need access to restricted APIs and they also need to read and modify untrusted web pages. As such, we expect our analysis and results to be applicable to other script-extensible browsers.We plan to test the generality of our vulnerability classification and defenses against other browsers, especially Google Chrome as it also provides powerful script extension functionality.

Acknowledgments

We would like to thank our shepherd, Helen Wang, and the anonymous reviews for their insightful comments on the paper.

References

[1] Adblock. http://en.wikipedia.org/wiki/Adblock.

[2] Alexa the web information company. http://www.alexa.com.

[3] Dromaeo JavaScript performance test suite. https://wiki.mozilla.org/Dromaeo.

[4] Greasemonkey. http://en.wikipedia.org/wiki/Greasemonkey.

[5] Return-to-libc attack. http://en.wikipedia.org/wiki/Return-to-libc_attack.

[6] setTimeout loses XPCNativeWrappers, July 2009. http://www.mozilla.org/security/announce/2009/mfsa2009-39.html.

[7] Incorrect execution of JavaScript in the extension context, May 2010. http://googlechromereleases.blogspot.com/2010/05/stable-channel-update.html.

[8] XPConnect wrappers, May 2010. https://developer.mozilla.org/en/XPConnect_wrappers.

[9] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[10] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[11] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proceedings of the USENIX Security Symposium*, Aug. 2009.

[12] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2–11, 2007.

[13] M. Dhawan and V. Ganapathy. Analyzing information flow in Javascript-based browser extensions. In *Proceedings of the Annual Computer Security Applications Conference*, 2010.

[14] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 339–354, 2008.

[15] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, Aug. 2008.

[16] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[17] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the EuroSys conference*, 2009.

[18] A. Slowinska and H. Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the EuroSys conference*, Apr. 2009.

[19] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[20] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 116–128, 1997.

[21] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 1–16, 2007.

[22] 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*, 2009.

[23] B. Yee, D. Sehr, G. Dardyk, J. B. 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*, pages 79–93, 2009.