SECURING SCRIPT-BASED EXTENSIBILITY IN WEB BROWSERS

by

Vladan Djeric

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Securing Script-Based Extensibility in Web Browsers

Vladan Djeric

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2009

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 script extensions and control over the entire browser process.

This thesis describes the pitfalls of script-based extensibility based on our study of the Firefox Web browser, and is the first to offer a classification of script-based privilege escalation vulnerabilities. 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 show that this approach is effective by testing our system against exploits in the Firefox bug database and finding that it detects the vast majority of attacks with no false alarms.

Acknowledgements

TODO.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web browsers perpetually need to add new functionality and customizability to keep up with rapidly changing Web technologies. In response, browsers have been designed to 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 [5] that allows users to install additional scripts that in turn dynamically customize HTML-based web pages or generate 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 the script-based attacks can often be executed reliably.

Our aim is to understand the nature of script-based privilege escalation vulnerabilities and propose methods to secure browsers against them. While there has recently been promising work towards securely executing native browser extensions [15, 6], the challenges in securing script-based extensibility are less well understood. Existing solutions widely employ stack inspection to ensure that remote code is sufficiently authorized to perform a security-sensitive operation. For example, a local file access is denied if the current stack contains a frame associated with an untrusted principal.

While stack inspection prevents attacks in which untrusted code is currently on the stack, it can still allow untrusted code no longer on the stack to influence the execution of security-sensitive code [10]. For example, an argument to a sensitive operation may have been changed by code that is currently not on the stack. This problem has been studied in the context of Java with techniques such as history-based access control [10, 12] and information-flow control [24]. For instance, Pistoia et al. [24] have proposed augmenting stack inspection with information flow to ensure that any code guarded by a stack-inspection check is only affected by code that has sufficient permissions to pass the check. This unified technique can detect all integrity violations, but it is highly restrictive and would require many endorsement operations [14] in a browser. For example, script code on a web page can connect to its origin site and transfer any data. The underlying socket connect operation is privileged, but the data being transferred is untrusted. Similarly, a browser uses a history file to track all visited web sites. Updating the history file is a privileged operation, but the contents of new entries are derived from untrusted pages. We discuss other approaches later, but most closely related work in this area has focused on security models, without a corresponding implementation or evaluation on a real system, so the impact of these models on real systems is unclear. Also, stack inspection and its variants can only be a part of the solution for browsers, which need to implement other security policies such as the same-origin policy.

In this thesis, we start by describing the pitfalls of script-based extensibility by providing a classification of script-based privilege escalation vulnerabilities, based on a study of the Firefox

Web browser. We justify using Firefox in the next section. Privilege escalation vulnerabilities are common in Firefox, and comprise roughly a third of the critical vulnerability advisories. These vulnerabilities have appeared regularly in every major 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. This 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 as determined by stack inspection, the principals of the object and the action being performed. Namespace isolation is used to enforce the same-origin policy for web page content scripts. This policy limits interactions between scripts and documents loaded from different origins (domains). The namespace isolation is one way in that script extensions are privileged and allowed to access content namespaces, but content scripts should not be able to obtain a reference to the privileged namespace. This policy is designed to defend against both cross-site scripting and privilege escalation attacks.

These basic security mechanisms are well understood and appropriate, but there are two main weaknesses in the security model: 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. We find that these problems occur because the dynamic and flexible nature of the Javascript language, used to implement the extensions, makes it nearly impossible to implement all the security checks correctly and anticipate all the corner cases in such a large and complicated code base.

1.1 Research Approach

While the problems described above are unique to script-based extensibility in browsers, the solution consists of combining well-known tainting techniques with the existing stack-based security model. Tainting *all* data from untrusted origins and propagating the tainted data throughout the code provides a much stronger basis for making security decisions. Our approach guarantees that tainted data will not be executed as privileged code. In essence, our attack detectors "second guess" the security decisions made by the browser's security mechanisms 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. Later, we further explain why this technique is appropriate for the different classes of privilege escalation vulnerabilities.

It is known that tainting systems can suffer from excessive false alarms when they conservatively mark any influence of a tainted input on an output [28]. This approach 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 content scripts could leak information [29]. In contrast, we do not need to be as strict because we can safely assume that script extensions are fully trusted and are not attempting to launder taints. Furthermore, namespace isolation requires that script extensions and content scripts should never be able to legitimately *invoke* each other directly. As a result, it is unlikely that tainting will occur erroneously, even if we fully taint all data and scripts emanating from web pages.

1.2 Contributions

There has been limited research on script-based privilege escalation vulnerabilities. Our contributions in this area are threefold: 1) we analyze and classify privilege escalation vulnerabilities and exploits, based on a study of a commonly used browser, 2) we evaluate a taint-based stack inspection method on a large, highly extensible application, and 3) we design effective sig-

natures for script-based privilege escalation exploits. We evaluate our approach using Firefox version 1.0 which has several privilege escalation vulnerabilities and easily-available exploits. Our results show that we can detect the vast majority of attacks with no false alarms and modest overhead.

1.3 Thesis Structure

Chapter 2 describes related work in the area and Chapter 3 provides background on the Firefox security model. Chapter 4 presents our classification of privilege escalation vulnerabilities and sample exploits. Chapter 5 describes our approach for securing script-based extensibility. Chapter 6 describes our implementation and Chapter 7 provides an evaluation of our approach. Chapter 8 presents our conclusions and describes future work.

Chapter 2

Related Work

Our work is heavily influenced by previous research on providing safe environments for remote code execution. A necessary component in such an environment is memory safety, traditionally implemented using hardware-based protection domains (via privilege levels and memory protection). The cost of crossing hardware protection domains is typically several orders of magnitude more than simple function calls, and can have significant impact on programs that communicate heavily across domains. This limitation has led to several software-based approaches such as software fault isolation [30] and type-safe languages for enforcing memory safety. However, memory and type safety only form a part of the solution for providing secure services [31, 10]. For example, Java supports components that execute with different privileges. This facility is used to execute remote code in a sandbox with limited access to local files and network sockets. Memory safety ensures that components cannot directly access each other's memory, but remote code may still be able to trick a vulnerable privileged component into executing code on its behalf, and thus be able to access sensitive files or sockets indirectly. Below, we discuss various proposed methods for securely sandboxing untrusted code and other areas relevant to this work.

2.1 Stack Inspection

Stack inspection is widely used by many web browsers and 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. Stack inspection determines principals based on the "origin" or source of the code, rather than the subject running the code. The code origin can be established using different methods such as digitally signed code or the domain from which the code is obtained. The basic stack inspection technique associates a principal with each stack frame on the call stack. A sensitive operation, such as accessing a local file, is denied if the current stack contains any frame associated with an untrusted principal. More generally, the run-time permission of any code is the intersection of the static permissions of all code on the stack. This simple approach does not allow an invocation from a trusted component, when it is run safely on behalf of remote code. This problem is resolved using extended stack inspection in which trusted code can enable specific permissions and the sensitive operation checks for these permissions on the stack. To prevent luring attacks, the operation is denied if any stack frame between the sensitive operation and the permission-enabling frame is untrusted. Wallach et al. [31] provide instructive background on stack inspection, and Erlingsson and Schneider [17] describe various implementations. Wallach et al. [31] also describe capability-based sandboxing, which has not found much acceptance because it requires significant retrofitting of library and application code, and controlling how capabilities are shared.

Abadi and Fournet [10] propose a novel technique in which access control decisions are based on the execution history of a program. The run-time permission of any code is computed by taking the intersection of the static permissions of all previously executed code, obviating the need for stack inspection. This method is safe but more restrictive than necessary because sensitive operations may be denied even when untrusted code in the execution history has no impact on the sensitive operations. This problem is resolved with an *Accept* programming construct that restores specific permissions after running untrusted code, but it is sensitive and its use increases the possibility of security pitfalls. This scheme has been analyzed [12], but

as far as we are aware, it has not been implemented or evaluated for a real system, and so it is unclear how often untrusted code in extensible applications would have to be "accepted" for correct operation.

Pistoia et al. [24] augment stack inspection, by ensuring that any sensitive code guarded by a stack-inspection check is only affected by code that has sufficient permissions needed for the check. This unified information-flow technique is sound because it can detect all integrity violations. However, it may cause false alarms because it uses program slicing, which is conservative. While our approach has some similarities with this work, there are important differences. First, the unified technique is restrictive, and in practice, would require many endorsement operations [14] in our environment. For example, script code on any untrusted web page can connect to its origin site and transfer any data. The connect operation is privileged, but the data being transferred is untrusted. Similarly, a browser uses a history file to track all visited web sites. The update to the history file is a privileged operation, but the data in the history file is derived from untrusted web pages. Our detection method is pragmatic and avoids these problems by only ensuring that untrusted data is not executed in a trusted context, and it ignores implicit information flows [27, 26]. The evaluation in Chapter 7 shows that our technique is able to detect exploits effectively, without causing false alarms. Second, Pistoia et al. present a static-analysis based formal security model, illustrated using Java examples. Our focus is on dynamically-typed scripting languages, which present several challenges such as `eval` statements that allow compiling and executing data. Despite much progress with static analysis, its application to dynamically-typed languages generally uses an easier-to-analyze subset of the language [33, 19], making it impractical for large, extensible applications.

2.2 Taint Analysis

Taint analysis helps determine whether untrusted data may influence data that is trusted by the system. Its use became popular with the Perl language, which allows detecting tainted

variables. Newsome and Song [22] use dynamic taint analysis to taint data originating or derived from untrusted sources, such as the network. 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. [29] use script tainting in a browser to track sensitive browser data, such as browser cookies or the URLs of visited pages. They detect attacks when tainted data is transferred to a third party (e.g., the attacker's site). While this work aims to detect cross-site scripting attacks, in practice it detects privacy leaks, including third-party sites that collect web site statistics or perform user tracking with the consent of the web site owner. Monitoring privacy leaks requires tracking implicit information flows conservatively, causing many false alarms, and requiring extensive whitelisting. In our threat model, privileged scripts can be buggy but are not under the complete control of the attacker (i.e., not malicious), and hence we do not track implicit flows because they are unlikely to be a significant avenue of privilege escalation attacks.

While we focus on securing browsers, several systems have used taint analysis [23, 33, 19, 34] to enforce web application security. Xu et al. [34] use taint tracking to detect attacks that subvert legitimate access privileges, e.g., an attack that subverts an FTP server to download the password file. They use source-to-source transformation to implement taint propagation and policies. overhead is high.

Zhao and Boyland [36] use type annotations to track data flow. They statically ensure that untrusted inputs are not passed to security-sensitive methods when they are invoked in a trusted context, and confidential values are not revealed to untrusted code. This approach requires fine-grained access-control annotations at the level of fields and methods for classes in trusted code, and it cannot prevent control-dependent information flow.

2.3 Same Origin Policy

The same origin policy is the basic sandboxing method used by web browsers. This policy limits interactions between scripts and documents loaded from different origins. Given the complexity of browsers, the same origin policy is hard to enforce correctly and has led to several browser bugs that allow violating the policy, leading to cross-site scripting attacks.

An effective method for implementing the same origin policy is script accenting [13]. Script code is accented (using the XOR operation) with the source of the window frame that supplies the code. When the code is compiled to be run, the frame in which it is compiled must be same as the frame supplying the code. Similarly, objects are accented with the frame that hosts the DOM tree of the object. When the object is accessed, the frame accessing the object must be the same as the frame of the object.

With extensible browsers, privileged script code is allowed to access scripts and data from different domains and thus bypasses the same origin policy restrictions. As a result of this sharing, it is important to allow communication while still tracking untrusted data. Furthermore, script or data accenting cannot be used or else the privileged scripts would be unusable. For example, a privileged UI script in the Firefox browser sets the title of a tab based on the title of a web page. This code would clearly not work correctly with accented web page contents. Even so, our compilation detectors have some similarities with code accenting.

The same origin policy is too strict for certain web applications such as mashup web sites. For such applications, Mashup OS provides abstractions that allow limited communication while protecting the different principals associated with mashup content [32]. For example, an integrator web site can specify a sandbox policy, and the web browser can then enforce the policy for unauthorized, third-party content. This approach has similarities with sandboxing content scripts, but it is primarily used to defend against cross-site scripting attacks, while our focus is on privilege escalation attacks. However, our work aims to improve the security of script sandboxing, which will be useful for the Mashup OS abstractions as well.

Since browser extensions typically run with unrestricted privileges, a malicious extension

can serve as a powerful attack vector. Louw et al. [20] propose access control for limiting extension privileges. For example, certain extensions may be restricted by the same origin policy or have no access to the password manager.

2.4 Browser-Based Defenses

BrowserShield filters the content of dynamic web pages to defend against browser vulnerabilities [25]. This vulnerability-driven filtering method offers an easier-to-deploy alternative to installing or patching security updates. While this approach can handle a more comprehensive set of attacks against the browser (e.g., buffer overflows), it requires known vulnerabilities.

Several research projects have focused on detecting cross-site scripting and related exploits in the browser. Hallarker and Vigna [18] audit script code at the interface between the Javascript engine and the core browser and detect known intrusions such as cookie stealing or unwanted pop-ups by using attack-specific signatures. Yu et al. [35] use program rewriting and pluggable security policies to achieve similar goals. A useful by-product of their work is an operational semantics of a core subset of Javascript, called CoreScript. The security policies that were implemented for CoreScript also detect cookie stealing and pop-up based attacks. Given the subtle flaws in script sandboxes and the range of attacks that are possible, these signature and policy based approaches are likely to provide only limited defences against privilege escalation attacks. Erlingsson et al. [16] make a strong case for end-to-end web security, and describe browser-side Javascript policies for enforcing a web application's security policies. Nadji et al. [21] describe a cross-site scripting defense in which a browser confines untrusted user-generated data on a web page based on server-specified policy.

Chapter 3

The Firefox Browser

In this chapter, we provide background on the Firefox browser. We start by justifying the use of Firefox for this work, then provide an overview of the Firefox architecture, and finally describe the Firefox security model.

3.1 Why Firefox?

We chose Firefox for our study for several reasons. First, it is one of the most popular browsers with over 20% market share [9], and it has a large number of third-party developers providing popular and commonly used script extensions. There are over 6000 extensions available across all versions of Firefox from the Firefox Add-ons web site [8]. Second, its broad user base, together with the detailed security reports and bug fixes that are available, makes it an excellent code base for investigating real and recurring issues that arise with securing untrusted code. Third, Firefox has one of the most powerful script-based extensibility features among the well-known browsers, making it a challenging target for any security mechanism. For instance, its user interface is specified using markup and script languages and is fully extensible. This approach enables portability across platforms and allows a script extension to enhance the GUI (e.g., to provide an extension-specific options menu).

Privileged script-based extensions are supported in all major browsers, other than Internet

Explorer. For example, the Gnome Epiphany browser supports powerful Python-based script extensions, and the Safari browser has a GreaseKit plugin that enables user scripting. Soon after this plugin was released, a privilege escalation vulnerability was reported, and its fix regrettably involved reducing the functionality of the plugin [7]. The Google Chrome browser has announced that they plan to support script extensions and have rudimentary support in development [4]. Interestingly, the Opera browser supports user scripts with limited additional privileges, primarily because of the potential for security vulnerabilities, even though there is demand for more functionality. We believe that all major browsers will eventually incorporate powerful script-based extensibility methods, based on demand and to remain competitive. Our goal, from a security perspective, is to help inform this process.

3.2 Architecture

Figure 3.1 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 is implemented as a separate library. Javascript scripts access XPCOM functionality via the XPConnect translation layer. This layer allows the interpreter and the XPCOM classes to work with each other's 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. This mechanism is unlike Java in which sensitive resources directly invoke the security manager.

The Firefox user interface is specified using the XML User Interface Language (XUL)

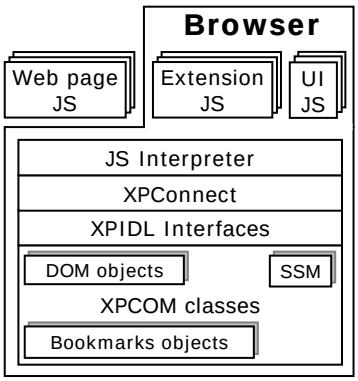

Figure 3.1: The Firefox architecture.

markup language, while privileged UI scripts, shown in Figure 3.1, provide interactivity. GUI and web page designers can also use the XML Binding Language (XBL) markup language to create new GUI widgets based on existing XUL and HTML tags.

Firefox's script extensions are usually written using a combination of Javascript, XUL and XBL and 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 content 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. Although potential security vulnerabilities can exist within plugin implementations, we do not address them. However, with appropriate sandboxing of plugins [15, 6], we would be able to monitor any script interactions with the plugins.

3.3 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 content scripts, and stack inspection regulates access to sensitive XPCOM components. Firefox also provides support for extended stack inspection for signed scripts, but this feature is seldom used. We describe each in more detail below.

3.3.1 Namespace Isolation

The browser runs scripts within an object namespace that defines the objects available to the script. At the root of each namespace is a global object, and for web pages, the global object is called the window object. For example, content scripts manipulate HTML by invoking the DOM methods of the document object that is set as a property of the window object.

The browser enforces the same origin policy by running content scripts from different web pages in different namespaces. These scripts are only allowed to access other namespaces from the same origin (described below). Script extensions are allowed to access all content namespaces. For example, the window object of a script extension may have a `content` property that allows accessing the currently loaded web page. The extension's namespace is hidden from the content scripts, and it is expected that extensions will never invoke content scripts directly.

3.3.2 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 some sensitive functions and object properties.

The principal of a content script is defined by the origin of the document containing the script (its protocol, hostname, and port). Such scripts only have access to the properties and methods of objects with the same origin. The principals associated with script extensions and native code are called the System and the NULL principal. These principals are fully trusted and allowed access to all available XPCOM interfaces, documents from different origins and objects with any principal.

The security manager calculates subject and object principals differently. The subject prin-

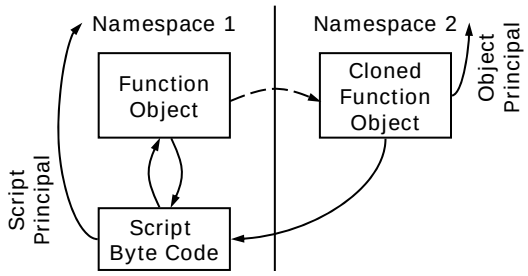

Figure 3.2: Cloned functions.

cipal is determined by performing a stack inspection of the Javascript stack. The security manager starts from the current frame and walks down the stack until it finds a stack frame with a principal.[1] It does not go any further down the stack because namespace isolation ensures that content scripts cannot invoke script extensions. If no principal is found in any of the frames on the stack, then it uses the principal of the global object in the current namespace.

The security manager determines an object principal by walking up the object's parent 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.

A stack frame's principal is determined by the principal of the script associated with the stack frame. There is one complication in the assignment of script principals. The principal of a script is normally the principal assigned to its byte code at compilation time, which would typically be the principal of the document in which the function was compiled. However, sometimes the script byte code is heavily shared among documents of different principals. For example, XBL bindings can be used to extend UI elements in both privileged and unprivileged documents. In this case, the script functions in an XBL document are compiled once and then *cloned* into all content namespaces by setting the function object's parent property to be an object in the destination namespace, as shown in Figure 3.2. The principal of a cloned function

[1] A stack frame may have the NULL principal when it is associated with a native function, which is assumed to be invoked on behalf of the scripted caller. It is necessary for native functions to have their own Javascript stack frames as they may invoke other script functions. For example, the native `ParseInt` string method invokes the native `toString` method when passed an object argument, and the `toString` method of an object can be overridden with a script function.

is defined by its object principal rather than its byte code principal.

3.3.3 Extended Stack Inspection

Firefox also employs extended stack inspection to support trusted content scripts. With extended stack inspection, trusted code can enable specific permissions and the stack inspection check looks for these permissions on the stack. To prevent luring attacks, in which trusted code with some enabled privileges is tricked into calling into untrusted code, the check fails if any stack frame between the sensitive operation and the permission-enabling frame is untrusted [31].

Trusted content scripts need to be digitally signed, and the user must give consent before they can execute with some of the elevated privileges available to script extensions, a feature meant primarily for Intranet applications. The versions of Firefox we have studied have not had security bugs in the extended stack inspection code, which is unsurprising because signed scripts are used infrequently. Later, we will discuss whether script extensions can benefit from extended stack inspection.

Chapter 4

Script-Based Privilege Escalation

In this chapter, we first discuss different classes of script-based privilege escalation vulnerabilities and then describe examples of real vulnerabilities.

4.1 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.

4.1.1 Code Compilation Vulnerabilities

Statically typed languages, such as Java, clearly separate code from data. For example, the class loader is the only component in Java that allows loading new code. Java associates each class with its class loader, allowing security code to reliably determine the origin of any byte code.

Unlike 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.

There are many situations when script extensions need data from a web page. For example, when a user hovers her cursor over a URI link, the status field in the browser UI is updated to display the target URI of the link. Similarly, if a user chooses an image from a web page as her desktop wallpaper, the "Set Desktop Background" dialog will ask for the user's confirmation by loading the image's URI in a dialog to create a preview of how it might look on the user's desktop. However, 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, special protocols such as "javascript" (e.g., `javascript:alert('Hello World');`) allow 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 recursive URIs such as `view-source:javascript:"some attack code"`.

4.1.2 Luring Vulnerabilities

Javascript is a prototype-based language in which an object can be set as the "prototype" of another object. Objects in the prototype chain form an inheritance hierarchy, helping resolve methods and properties if the object does not contain their definitions. Javascript allows scripts to dynamically change the object inheritance chain, redefine methods and properties, and associate getters and setters with member variables. Javascript also uses lexical scopes, such as nested functions, to define the object namespace, and allows eval-like functions to execute strings in different scopes.

These powerful features provide great flexibility that is routinely tapped by experienced Javascript programmers, but it is troublesome for privileged code when it needs to access ob-

jects from unprivileged namespaces. Privileged functions cannot be certain about the type of object they are interacting with, whether any of the methods they are invoking have been overridden with malicious versions, or if the variables they access have getters and setters.

This problem is severe because script extensions routinely access properties and methods of DOM objects in content namespaces. This operation should be straight-forward because script DOM objects are wrappers for XPCOM objects with well-defined interfaces, but Javascript's flexibility allows content scripts to make significant changes to these objects. This flexibility can lead to unintentional execution of unprivileged code, a luring attack, or compromise the correctness of the returned values. In object-oriented terms, script extensions need access to final or base class member functions, but Javascript does not support them natively.

Another difference from Java is Javascript's lack of private and package scoping of class variables and methods. Javascript does not natively support these encapsulation methods. It is possible to imitate the functionality of private functions using Javascript's nested functions but this practice is almost non-existent. The lack of private variables makes it hard to force control to flow through security checks and thus implement reference monitors [31].

4.1.3 Reference Leak Vulnerabilities

Given the complexity of browsers, it is hard enough implementing complete namespace isolation to enforce the same origin policy. For example, Chen et al. [13] describe several cross-site scripting vulnerabilities in the IE browser. Firefox's security model assumes a weaker, one-way isolation model between privileged and unprivileged namespaces. Bugs in any native code that deals with namespaces can lead to unprivileged code gaining access to references in privileged namespaces. Also, careless extensions may place references to privileged objects in an untrusted namespace.

Reference leak vulnerabilities are critical because they compromise namespace isolation. They allow an attacker to modify data or code defined in a privileged namespace, call arbitrary functions within the privileged namespace, potentially providing arbitrary execution control.

Reference leaks are also dangerous because privileged code that depends on namespace isolation may become vulnerable to code compilation or luring attacks. For example, naively luring privileged functions into calling regular unprivileged functions does not lead to a successful exploit. The byte code of an ordinary function is tagged with the principals of the document in which the function was declared. Even if such a function is invoked from a privileged namespace, the principals of its stack frame will still be correct, and the security manager would not be fooled when determining the subject's principal. However, with a reference leak, an attacker can defeat security checks, which are mostly based on the principals of the caller (the subject principal), by tricking privileged code into calling privileged target functions of the attacker's choosing.

4.1.4 Insufficient Argument Sanitization

Vulnerabilities can also occur if a browser extension uses unsanitized data from untrusted documents 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 the content page as part of the local filename to which the file is saved, it may be open to directory traversal attacks 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 vulnerability has not been documented in the Firefox bug database, but we consider it a likely vulnerability for extensions.

4.2 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.

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

Figure 4.1: Target code invoked when a LINK tag is found in the current web page.

4.2.1 URI Code Injection

Figure 4.1 shows an insufficient argument sanitization and code compilation vulnerability in the browser code that allows URI code injection (Bug 290036). This browser UI code displays a favicon (16x16 pixel icon) image next to the browser's URL bar. Normally, the icon's URI would be the Web 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 4.2.4). Note that the stack inspection check allows invoking the `setAttribute` function. This vulnerability occurs because the native code implementing the icon container does not perform sufficient argument sanitization and the compilation functions are unable to determine the safety of the `Href` argument.

4.2.2 Compilation with Wrong Principals

Figure 4.2 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

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

Figure 4.2: Exploit code that allows untrusted functions to be associated with privileged principals.

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 reference to a special `call` function of an XBL object called `myElement`. XBL objects can legitimately be created to extend the UI in privileged or unprivileged documents, but their `call` function lies in a privileged XBL document namespace.

Exposing the `call` function is a reference leak, but it is not sufficient for an attack because the interpreter invokes this function with the correct caller's principals, so access to `xbl_object` by itself is not enough. However, within `eval`, 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 not problematic.

The exploit occurs when `evalCode` creates a function referenced by `clonedFunction`. The interpreter creates a new function object in the privileged XBL document namespace that is a clone of `deliverPayload`, as described in Section 3.3.2 and Figure 3.2. The attacker can then invoke the cloned function to execute its payload with elevated privileges. In effect, this exploit attaches a user-supplied function to a privileged namespace, thus making it appear

privileged to the security manager. This code compilation and reference leak vulnerability occur because the implementation of `eval` did not check that it was compiling code from one principal that could execute within the namespace of a different 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-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`.

4.2.3 Luring Privileged Code

Figure 4.3 shows the exploit code for a luring attack. This exploit would trigger if the UI code reads the `document.body.localName` property. 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. 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.[1] 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 the developers eventually implemented "safety wrappers" in the browser that ensure that only the originally defined methods and members of HTML elements in web pages are visible to extensions, but even the latest releases of Firefox continue to suffer from luring attacks.

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

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

Figure 4.3: Simplified exploit code for Bug 289074.

```
var leaked = QueryInterface.__proto__.__parent__;
var cid = { equals : Script(payload) };
leaked.gModule.getClassObject(null, cid, iid);
```

Figure 4.4: Simplified exploit code for Bug 294795.

4.2.4 Privileged Reference Leaks

Figure 4.4 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 access this prototype object and its parent property, which leads it to the global object of a privileged namespace. The exploit calls the scripted method `getClassObject` of the `gModule` object defined in the privileged namespace using a specially-crafted argument to carry out a luring attack.

The `getClassObject` method shown in Figure 4.5 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, granting it full privileges. This exploit targets the reference leak and luring vulnerabilities.

4.2.5 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)

```
var gModule = {
  getClassObject:function(..., aCID, ...) {
    if (aCID.equals(this._objects[key].CID))
      return this._objects[key].factory;
  }
};
```

Figure 4.5: Simplified target code for Bug 294795.

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 by using a malformed URI. We do not address cross-site scripting bugs or violations of URI loading policies, but our system is able to detect this category of attacks because it leads to code injection.

4.3 Discussion

In Section 3.3.2, we had mentioned that script extensions and native code execute with full privileges that are permanently enabled. Employing least privilege techniques would not necessarily be beneficial. For example, when data is compiled with wrong principals, the compilation privilege is already enabled. It also does not address luring attacks because the target could be a privileged scripted function that enables the privilege. It would also be cumbersome for extension code to enable and disable XPCOM access privileges as sensitive XPCOM functions are accessed frequently in script extensions.

Similarly, extended stack inspection (see Section 3.3.3) cannot solve privilege escalation vulnerabilities. Namespace isolation ensures that unprivileged code never calls privileged directly, therefore extended stack inspection would only be able to detect attacks where unprivileged code calls privileged directly through a reference leak. Attackers could bypass this mechanism by calling the privileged reference indirectly, through luring attacks or by setting

up callbacks with native methods such as the setTimeout timer method. Fundamentally, the problem is that once principals are wrongly assigned, stack inspection by itself will not work correctly.

Chapter 5

Approach

Script-based extensibility in web browsers 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 the browser's 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.

While these vulnerabilities are unique to script-based browser extensibility, they arise primarily because untrusted content is able to affect privileged code. Our solution therefore consists of augmenting the browser's security mechanisms with tainting. In this chapter, we show that while this solution is conceptually simple, it is well-suited for the browser's security model.

5.1 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 content scripts begin executing. We do not depend on the correctness 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.

5.2 Sources of Taint

The browser is started in a completely untainted state. The initial sources of taint in our system are any untrusted documents loaded in the browser. We consider it safe to base our tainting on this initial source because the documents are tainted before any content scripts have begun executing. We consider all documents fetched from remote sources or local documents opened with the "file" protocol as untrusted because the browser does not assign them the privileged System Principal. Untainted documents are typically loaded via the chrome URI, and consist of script extensions, user interface scripts and XUL documents, running with the System 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, with the window and document objects at the root of the tree. 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.

5.3 Taint Propagation

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 code reads the tainted document's title property and sets it as the caption of the tab element. Recall that the UI is specified using XUL markup language documents. The UI code sets the caption property of a DOM element representing the current tab in its XUL document. 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 XUL document. It also requires that DOM elements be capable of separately storing the taint status for each of their attributes. Tainting code in XPConnect propagates taint between the DOM and script environments.

5.3.1 Taint Propagation in Javascript

Our tainting system uses different policies based on the privilege level of the executing script. We unconditionally taint all script variables created or modified by executing scripts from tainted documents, but we use standard tainting rules for privileged scripts.

Control dependent tainting is not required in our system, allowing us to have fewer false positives. We can safely assume that privileged scripts are non-malicious and that they are not attempting to fool our tainting system. This assumption is reasonable since privileged scripts come from either the browser or from fully trusted browser extensions. Furthermore, it seems highly unlikely that privileged script code would accidentally launder taints through control flow and then execute the laundered data as privileged code.

To determine whether a script is from an untrusted document, we consult the security manager for the script's effective principals, because it is not sufficient to check the principals assigned to the script byte code when it was compiled. The reason is that shared script functions can be compiled in a privileged namespace and then cloned into unprivileged namespaces. This approach does not increase our reliance on the security manager since our tainting sys-

tem ensures that principals are assigned correctly. We rely only on the initial assignment of principals to documents and the correctness of our system.

5.3.2 Taint Propagation in XPCOM

XPCOM objects can be a taint source or a taint sink. The XPCOM objects that represent DOM elements in unprivileged documents are the initial taint source, so their properties always need to be tainted. XPCOM objects, including DOM elements of privileged documents, can also be taint sinks, as in the URI code injection attack described in Section 4.2.1.

XPCOM objects need to track the taint status of each of their properties because each property can be independently set to a tainted value. Therefore, we added a taint flag to the definitions of data structures that represent property values (such as XPCOM strings) and modified their methods to preserve taint. We also modified DOM elements to unconditionally taint their properties if the element exists within an unprivileged document. We modified XPConnect to taint any Javascript references to unprivileged DOM elements and to propagate taints between the XPCOM and Javascript environments. For implementation reasons explained in Chapter 6, we do not track the taint-status of non-string XPCOM data types.

5.4 Attack Detection

We define a privilege escalation attack as tainted data executing as privileged code. This attack occurs when tainted data is used to create a privileged stack frame, as determined by the security manager. This can happen only if a script compiled from tainted data is executed with privileged principals, or if a tainted function pointer is used to create a stack frame with privileged principals.

We implement two classes of attack detectors to detect these conditions: 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.

5.4.1 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 4.2.2). If native XPCOM code compiles the strings, as in the URI code injection attack (see Section 4.2.1), or the XSS attacks (see Section 4.2.5), 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 defined by the Javascript API.

5.4.2 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.

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 4.2.3). This detector is able to intercede before any function code is executed with elevated privileges. The detector establishes that the callee's stack frame will

be privileged by querying the security manager for the subject principal of the newly extended stack.

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 allowing the call to 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. Each stack frame keeps track of the function reference used to create the frame. 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.

We assume that native functions already accessible to unprivileged scripts, such as methods of DOM objects from the same origin and native functions implemented by the interpreter, are not worthwhile luring targets. We have seen exploits lure scripts from privileged contexts into calling eval-like functions such as the DOM's setTimeout method, but these attacks are already caught by our compilation detectors. It might be possible to lure privileged scripts into calling these functions with untainted arguments of the privileged script's choosing, but such attacks would be of limited usefulness. If such attacks were shown to be useful, we could defend these eval-like methods with the security manager's stack inspection, which would detect the luring

attempts on the stack.

One final subtlety for invocation detectors involves the interpreter's use of intermediate helper functions when a script attempts to use an object as if it were a function. Certain types of objects such as Script objects can be invoked as if they were functions. Internally, the interpreter needs to go through intermediate steps to convert the object reference into a reference to a script function. These helper functions create empty stack frames and manipulate the links between frames, and so it is not simple to determine the future privilege level of the scripted callee. However, since the helper functions are only used to execute objects with eval-like semantics, we can be strict and signal an attack whenever a privileged caller invokes a tainted reference requiring a helper function.

5.4.3 Reference Leaks

As demonstrated in Chapter 7, 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 lure privileged code with our invocation detectors, as in the reference leak attack (see Section 4.2.4), and by detecting malicious attempts to compile tainted strings with our compilation detectors. However, we are unable to detect and prevent privilege leaks from occurring. For example, in Figure 4.4, 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.

For performance reasons, we also cannot check whether every object accessed in an unprivileged sandbox comes from a privileged namespace. Every input to an opcode would require a call to the security manager to perform an object principal check. Additionally, it is conceivable that an exploit could set up a luring attack to a leaked reference without ever touching the reference directly, thereby bypassing any reference leak detector. For example, this could happen if the leaked reference were a method of an unprivileged object. Therefore, to cover all cases, it would be necessary for every opcode to check not only whether an input is a reference

leak, but also whether any of its properties are reference leaks. The performance penalty of this approach renders it infeasible.

Although we cannot 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.

5.4.4 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 security-sensitive XPCOM interfaces and their security-sensitive parameters 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 [11], allow validating sanitization routines.

Chapter 6

Implementation

In this chapter, we describe the implementation of our tainting system in the Javascript interpreter and the XPCOM classes, our attack detectors, and the limitations of our design. 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.

6.1 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 referred to as "objects"). Therefore, we have a choice between associating taint status with objects or with references to objects.[1] We believe that it is a mistake to associate taint with objects because objects can be safely shared across privileged and unprivileged namespaces. For ex-

[1] Primitive data types can be transparently upgraded to reference types by converting primitive types to doubles.

ample, if a string variable were to be defined in a privileged namespace and then assigned to a variable in an unprivileged namespace, the two references should not have the same taint status although they reference the same heap object. Note that strings and doubles are immutable, so there is no risk of modification by untrusted code. If a property of a shared Javascript object is modified by untrusted code, the affected property reference will be tainted.

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.

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.

6.2 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. [29]. 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 HTML, XUL and XBL elements, representing the contents of the 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 behaviour 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.

6.3 Attack Detectors

Once we determined the detection policies described in sections 5.4.1 and 5.4.2, implementation of the attack detectors became straightforward. 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.

6.4 Limitations

Firefox is a large and complex code base that was never meant to support a dynamic tainting system nor a notion of origin for all of its data types. Our retrofitted security system attempts to provide a system-level security guarantee but existing design choices and practices raise the specter of taint loss that needs significant manual effort to correct. For example, the developers' frequent use of primitive data types such as char and char* in common IDL interfaces and XPCOM methods (e.g., `nsGenericDOMDataNode::SetText(char*)`) makes it difficult to associate strings with additional information as they travel through the code base. For implementation reasons, we do not explicitly track the taint status of primitive data types on the C++ side and across the XPConnect boundary.

Furthermore, we've also come across several DOM elements that do not implement the standard DOM interfaces and they also do not provide any methods that could be used to determine their origin or trustworthiness. One such example is the CSSStyleDeclaration class. When such an object is wrapped before being reflected into Javascript, there is no reliable way to determine whether the wrapper should be marked as tainted or not.

Finally, we are also unable to handle any taint laundering that occurs outside the XPCOM classes and the Javascript interpreter, such as inside plugins or files.

Chapter 7

Evaluation

We have implemented the approach described above in the Firefox browser. In this chapter, 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 have implemented our system on Firefox version 1.0.0, which we use for all the experiments. We chose this version for two reasons. First, this version has the largest number of known privilege escalation bugs, allowing more extensive testing of our system. Second, 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 March 2009, the current version of Firefox (v3.0) has virtually no available exploits. We plan to port our system and evaluate our results for these versions as exploits become available in the bug database.

7.1 Vulnerability Coverage

Table 7.2 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 7.2 shows that PE vulnerabilities comprise 2/3 of all critical Firefox 1.0 vulnerabilities. All other versions continually have 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 4.2.4.

Table 7.1 shows all the 20 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 7.2), but the other six 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 20 advisories because arbitrary code execution 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 12 out of the 15 vulnerabilities described in the advisories. We do not detect the attacks on vulnerabilities in advisories #6, #7 and #15. The reason for not detecting #6 and #15 is that we have not yet implemented taint sources for inputs other than web pages. For example, the vulnerability in #6 consists of an external application such as a standalone media player directing Firefox to open javascript protocol URIs, which we would detect if we marked URIs supplied by external programs as tainted sources. The vulnerability in #7 occurs due to our system losing track of taint in the HTML parser where we have not yet implemented our taint propagation.

#	Advisory	Advisory Name	Type of Vulnerability
1	2006-25	Privilege escalation through Print Preview	Compilation
2	2006-16	Accessing XBL compilation scope via valueOf.call()	Leak (C)
3	2006-15	Privilege escalation using a Javascript function's cloned parent	Leak (C)
4	2006-14	Privilege escalation via XBL.method.eval	Leak (C)
5	2005-56	Code execution through shared function objects	Leak (C), Leak (L)
6	2005-53	Standalone applications can run arbitrary code through the browser	Compilation
7	2005-49	Script injection from Firefox sidebar panel using data://	Compilation
8	2005-44	Privilege escalation via non-DOM property overrides	Luring
9	2005-43	"Wrapped" javascript: URLs bypass security checks	Compilation
10	2005-41	Privilege escalation via DOM property overrides	Luring
11	2005-39	Arbitrary code execution from Firefox sidebar panel II	Compilation
12	2005-37	Code execution through javascript: favicons	Compilation
13	2005-35	Showing blocked javascript: pop-up uses wrong privilege context	Compilation
14	2005-31	Arbitrary code execution from Firefox sidebar panel	Compilation
15	2005-12	javascript: Livefeed bookmarks can steal private data	Compilation
Embargoed, or exploit not available			
16	2006-24	Privilege escalation using crypto.generateCRMFRequest	N/A
17	2006-05	Localstore.rdf XML injection through XULDocument.persist()	N/A
18	2005-58	Firefox 1.0.7 / Mozilla Suite 1.7.12 Vulnerability Fixes	N/A
19	2005-45	Content-generated event vulnerabilities	N/A
20	2005-27	Plugins can be used to load privileged content	N/A

Table 7.1: Vulnerability Coverage.

Firefox	Critical	Critical PE	%
Version 1.0	27	18	67
Version 1.5	44	13	30
Version 2.0	42	15	36
Version 3.0	14	5	36

Table 7.2: Vulnerability Statistics.

7.2 Manual Testing

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, 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 is not a false positive because untrusted data is being executed with full privileges. 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.

While our manual testing is limited to heavily visited web sites, we believe that our system

will not generate many false positives with other web sites. Unlike other tainting systems, our detectors precisely look for activities that should never occur during normal browser operation. Specifically, 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). As future work, our system needs to be evaluated for false positives using a more rigorous, automated testing method.

7.3 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. Our code has not been tuned, and we expect that this overhead can be reduced with tuning.

Chapter 8

Conclusion

Script-based privilege escalation attacks are a serious and recurring threat for extensible browsers such as Firefox. In this thesis, we describe the pitfalls of script-based extensibility by presenting a classification of privilege escalation vulnerabilities. Then, we propose a tainting-based system that specifically targets each class of vulnerability. We implemented such a system for the Firefox browser and our evaluation shows that it detects the vast majority of attacks in the Firefox bug database with no false alarms and reasonable overhead.

8.1 Future Work

Currently, we are porting our system to Firefox 2.0. We suspect that browser defenses, exploit techniques, and extensions have all advanced and become more varied and sophisticated since Firefox 1.0 was released. For example, we know that versions of Firefox after v1.0.2 include the automatic "Native Wrapper" mechanism which protects extensions from unsafe interactions with DOM objects from untrusted web pages. We also know that attackers have found ways to defeat this protection mechanism. It should be interesting to find out how well our current system fares against these more advanced techniques and what changes, if any, are required to cope with new threats. Also, in the near future, Google Chrome will officially debut its script extension functionality and we are curious to test the generality of our vulnerability

classification and defenses.

We could also turn our fortified version of Firefox into a Web crawler and conduct a study of the prevalence of script privilege escalation attacks on the Web. We are not aware of any studies of the privilege escalation threat posed to ordinary surfers. It would be useful to understand the characteristics of the payloads in real attacks and to find ways to mitigate them if it is not possible to offer perfect protection.

A third possible extension of our work is to investigate the feasibility of attacks based on the XPCOM "insufficient argument sanitization" class of vulnerabilities. Currently, we are not aware of any real or proof-of-concept exploits, but the size and diversity of XPCOM and the Firefox extension ecosystem nearly guarantee the existence of many vulnerable extensions. It would not be sufficient to set up taint detectors around sensitive XPCOM parameters because extensions may legitimately call sensitive XPCOM methods with tainted arguments. For example, an image downloader extension may need to create a new local file with the file name derived from the URI of a Flickr image. Endorsement functions are a potential solution.

Our system could be adapted to monitor the execution of extensions for dangerous behaviours. For example, we could raise security warnings whenever an extension breaks namespace isolation by sharing object references or private information across different domains. This behaviour is problematic because it creates the possibility of cross-site scripting or data theft.

To prevent control flow attacks from leaked references (described in Section 5.4.3), we could extend our tainting system to detect attacks on namespace isolation. By adding another taint bit, we could distinguish between tainted variables created by the execution of unprivileged and privileged scripts. If we found that a tainted variable declared in a privileged namespace was directly modified by an unprivileged script, we would raise a security warning.

Finally, it may be useful to investigate what design principles are required and what tradeoffs are necessary to develop a browser that tags each piece of data with its origin and the types of policies and issues that may arise.

Bibliography

[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] Google Chrome. http://en.wikipedia.org/wiki/Google_Chrome.

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

[6] Native Client (Google Code). http://code.google.com/p/nativeclient.

[7] GreaseKit GM APIs Vulnerability, December 2007. http://8-p.info/greasekit/vuln/20071226-en.html.

[8] Firefox Add-ons, March 2009. https://addons.mozilla.org/en-US/firefox/.

[9] Usage share of web browsers, March 2009. http://en.wikipedia.org/wiki/Usage_share_of_web_browsers.

[10] Martin Abadi and Cedric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium*, pages 107–121, 2003.

[11] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analy-

sis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[12] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programing*, 15(2):131–177, 2005.

[13] Shuo Chen, David Ross, and Yi-Min 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.

[14] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the USENIX Security Symposium*, pages 1–16, 2007.

[15] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob 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.

[16] Ulfar Erlingsson, Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *Proceedings of the USENIX Workshop on Hot topics in Operating Systems*, pages 1–6, 2007.

[17] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 246, 2000.

[18] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, 2005.

[19] Nenad Jovanovic, Christopher Kruegel, , and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006. Short Paper.

[20] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, August 2008.

[21] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[22] James Newsome and Dawn 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*, February 2005.

[23] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[24] Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 149–163, 2007.

[25] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2006.

[26] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[27] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security (TISSEC)*, 3(1):30–50, 2000.

[28] Asia Slowinska and Herbert Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the EuroSys conference*, April 2009.

[29] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[30] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

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

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

[33] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, 2006.

[34] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*, 2006.

[35] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 237–249, 2007.

[36] Tian Zhao and John Boyland. Type annotations to improve stack-based access control. In *Proceedings of the IEEE workshop on Computer Security Foundations*, pages 197–210, 2005.