Detecting Malicious JavaScript Code in Mozilla

Oystein Hallaraker and Giovanni Vigna
Reliable Software Group
Department of Computer Science
University of California, Santa Barbara
hallarak@stud.ntnu.no, vigna@cs.ucsb.edu

Abstract

The JavaScript language is used to enhance the client-side display of web pages. JavaScript code is downloaded into browsers and executed on-the-fly by an embedded interpreter. Browsers provide sand-boxing mechanisms to prevent JavaScript code from compromising the security of the client's environment, but, unfortunately, a number of attacks exist that can be used to steal users' credentials (e.g., cross-site scripting attacks) and lure users into providing sensitive information to unauthorized parties (e.g., phishing attacks). We propose an approach to solve this problem that is based on monitoring JavaScript code execution and comparing the execution to high-level policies, to detect malicious code behavior. To achieve this goal it is necessary to provide a mechanism to audit the execution of JavaScript code. This is a difficult task, because of the close integration of JavaScript with complex browser applications, such as Mozilla. This paper presents the first existing implementation of an auditing system for JavaScript interpreters and discusses the pitfalls and lessons learned in developing the auditing mechanism.

Keywords: *Mobile Code, JavaScript, Web Applications.*

1. Introduction

Mobile code consists of small pieces of software that are transferred across networks and executed on a remote computer. While code mobility improves computing efficiency and reduces latency, it also introduces security issues that have to be dealt with. Different models [11] have been developed to address these issues, including software fault isolation [15], safe interpreters [13], and sand-boxing [3]. While these techniques provide some form of protection, vulnerabilities that can be exploited using mobile code are still common.

JavaScript [7] is a scripting language developed by Netscape to create interactive HTML pages. JavaScript conforms to the ECMAScript standard [1]. Usually, JavaScript code is embedded in HTML code. When a browser downloads a page, it parses, compiles, and executes the script. As with other mobile code schemes, malicious JavaScript programs can take advantage of the fact that they are executed in a foreign environment that contains private and valuable information.

The existing JavaScript security solution is based on sand-boxing, which allows the code to perform a restricted set of operations only. JavaScript programs are treated as untrusted software components that have access to a limited number of resources within the browser. The problem with the current solution is that scripts may conform to the sand-box policy, but still violate the security of the system.

For example, in *cross-site scripting* (XSS) attacks, a malicious web application gathers confidential data from a user. A typical example of a XSS attack is when a user is tricked into clicking on a link hosted by a malicious host (e.g., *www.evil.com*). The link, appears to be pointing to a resource on a trusted site (e.g., *http://www.bank.com/accounts.html*, but, instead, it contains JavaScript code as the resource name (e.g., *http://www.bank.com/<script>sendcookieto(hacker@evil.com)</script>)*. When the resource is requested by the user's browser, the code is sent as part of the HTTP request to the destination of the link (i.e., the trusted web server). Since the requested resource does not exist (e.g., there is no file called "*<script>sendcookie...</script>*"), the trusted web server returns an error message that contains the name of the resource that could not be accessed. As a result, the page containing the error message is interpreted by the client's browser as a page from the trusted site containing some JavaScript code. Therefore, the code is executed in the context of the trusted site and has access to the cookies previously set by the trusted site, including session identifiers and authentication tokens.

Another example is represented by scripts that abuse systems resources, such as opening windows that never close

or creating a large number of pop-up windows.

We propose a novel solution where all operations from a downloaded JavaScript program are monitored and logged. Based on the auditing information, different intrusion detection techniques can be used to evaluate the actions of the script and take appropriate countermeasures if malicious behavior is detected.

To the best of our knowledge this is the first implementation of a security auditing mechanisms that addresses the execution of JavaScript code within a browser.

This paper is structured as follows. Section 2 introduces the current security mechanisms for JavaScript in Mozilla and discusses related work. In Section 3, we describe the architecture of the Mozilla browser. Section 4 and 5 present our auditing system and show how malicious JavaScript code is detected. In Section 6, we present an evaluation of the system. Finally, in Section 7 we conclude and outline future work.

2. Related Work

JavaScript was developed as a light-weight scripting language with object-oriented capabilities. The current JavaScript security solution is based on executing JavaScript code within a sand-box [2]. The JavaScript sand-box is similar to the Java sand-box, but it is more restrictive since JavaScript does not provide any built-in support for file access. This means that by default, a JavaScript program cannot read files on the local drive or access random XPCOM objects (see Section 3 for a discussion of XPCOM). Other examples of operations that are not allowed are opening a window smaller than 100x100 pixels, using the `History` object to find out recently visited pages, and unconditionally close a browser window. In addition to restricting many operations, Mozilla uses two main JavaScript security policies. The first policy, called the *same-origin policy*, is used to isolate one document from another, while the second one, called the *signed-script* policy, provides means to enforce finer-grained access control.

The same-origin policy prevents documents or scripts loaded from one origin (i.e., a web server), from getting or setting properties of a document from a different origin. In this context, "same origin" means same protocol, host, and port. This policy provides the foundation for isolating one script from another, and ensures that a document downloaded from one source cannot be changed by JavaScript code downloaded from another origin. There is one exception to this rule: a document can set its domain to a suffix of its current domain. This means that a document from `http://store.factory.com` can access a document from `http://factory.com` after setting its domain to `factory.com`. This policy applies to both windows and frames.

Having all JavaScript programs executing within the sand-box created frustration in the JavaScript programmers who desired to have access to more functionality. A second policy, the signed-script policy, was therefore developed to give scripts more functionality and give a user the option to define a finer-grained security policy. Script signing allows a script to get out of the sand-box and is similar to the mechanisms used for signed Java applets. When the browser downloads a script that is digitally signed, the browser first verifies the signature and then extracts the principals of the script. The principals can either be derived from validating the signature of a script or they can be derived from the origin of the script. If a principal is derived from the origin of the script, the principal is called a *codebase principal*. Signed scripts are allowed to ask for extended privileges/capabilities at runtime, using the command `netscape.security.-Privilegemanager.enableprivilege()`. The privilege represents permissions to access specific targets and a prompt will appear in the browser window whenever a script asks for a privilege. The privileges range from automatically bypassing the same-origin checks to reading random files on the local drive.

In addition to these policies, Mozilla also developed a concept called configurable security policies (CAPS). Mozilla's configurable security policies allow users to configure the general security policies and to define different security policies for different security domains. There is currently no user interface for this, and users must manually change the *user.js* configuration file to modify a policy. For example, by editing the *user.js* file, a user can deny all scripts access to certain methods, e.g, `window.open()`, or allow scripts from certain domains only to access some properties.

Sand-boxing is a crucial aspect in JavaScript security, and this technique has been extensively researched by the security community. In [3], different sand-boxing methods are presented that can be used to enforce strict security policies. Janus is another practical tool for application sand-boxing [14]. Janus provides an environment for placing an entire web browser or helper application inside a sand-box, and then monitoring system calls originating from the application contained within the sand-box. The problem with these sand-boxing facilities is that they are too generic. Both the mentioned systems can prevent scripts from accessing private information outside the browser, but they provide no means for protecting information within the browser. In addition, most accesses from the JavaScript engine to the OS are performed through the browser process, making it difficult to determine the origin of an operation.

3. The Mozilla JavaScript Architecture

Mozilla is an open-source, free software project that includes a web browser and an e-mail client. The open-source Mozilla browser has hundreds of contributors, and is constantly evolving.

3.1. The SpiderMonkey JavaScript Engine

SpiderMonkey [12, 6] is the code-name for the implementation of the JavaScript engine embedded in Mozilla. It is a stand-alone JavaScript engine that parses, compiles, and executes JavaScript code. The engine conforms with the ECMAScript standard [1], which is the standardized version of JavaScript. ECMAScript defines built-in types for Undefined, Null, Boolean, Number, and String. In addition ECMAScript defines a collection of built-in objects which include the Global object, the Object object, the Function object, the Number object, the Math object, the Date object, the RegExp object and some Error objects. An application embedding SpiderMonkey may also define its own application-specific objects in addition to the built-in objects. In a browser like Mozilla, the application-specific objects are responsible for providing access to the Document Object Model (DOM) [16] from within the JavaScript engine.

The DOM is a platform and language neutral interface that allows scripts to dynamically access and update the content, style, and structure of web documents. The DOM typically contains an object-instance hierarchy that models the browser window and some browser window information. It also contains an object-instance hierarchy of elements of an HTML document, which is created when the document is loaded into the browser. For example, some of the objects made accessible by the DOM are the `window` object, the `document` object, the `navigator` object, and the `location` object. The `window` object is the global object from which all other objects inherit. The `document` object contains the HTML elements of the current document. The `navigator` object encapsulates information about the browser, while the `location` object contains information about the current URL. Each object has a number of properties which can either be a built-in type, an object, or a method. An example of this is the `href` property accessed using the expression `document.location.href`. A JavaScript program first accesses the `document` object which is a property of the `window` object. The `location` object is a property of the `document` object, and `href` is a property of the `location` object.

SpiderMonkey exposes a public API that applications can use to compile and execute scripts, instantiate host objects, and define properties. The engine does not provide any security *per se*, and all mechanisms to provide access control and safety must be implemented in the embedding application, e.g., the web browser.

3.2. Mozilla and SpiderMonkey

Mozilla [4] is a large and modular software project that is written in both C, C++, and JavaScript. Several technologies are used in Mozilla to break the project into smaller pieces that can be developed independently and efficiently. The main mechanism that supports the integration of the different components is the Cross-Platform Component Object Model (XPCOM) [5, 4], which is similar to Microsoft's Component Object Model (COM). Other technologies used are XPConnect and the Cross-Platform Interface Definition Language (XPIDL). All these technologies will be explained in detail in the following subsections.

Parts of Mozilla are written in JavaScript, which means that the SpiderMonkey interpreter executes both scripts on behalf of a downloaded web page and scripts that are part of the "native" code of the Mozilla browser. The "native" JavaScript code is considered part of the browser code and is not executed within a sand-box.

3.2.1 Cross-Platform Component Object Model (XPCOM)

The main goal of XPCOM is to provide a modular framework that is both platform-independent and language-independent. XPCOM enables a software project to be broken up into smaller modularized pieces that are integrated at runtime, and separates the implementation of an object from its interface. The advantages of developing modular software are many: the code can be reused in many applications, components can be updated without needing to recompile the whole application, and performance can be improved by only loading the modules that are needed at a certain point in time.

The basic idea is that related functionality is gathered in one entity, called a component or a module. The component implements one or more interfaces through which other components can access its functionality. An interface consists of one or more methods and variables. Each component has a unique `classID` and `contractID` that describe the component. In addition, each interface the component implements has a unique `interfaceID` which must be specified before accessing the component. The component manager keeps track of all the components in the system, and is responsible for finding the correct component when a `contractID` or `classID` is specified. An important concept in XPCOM is object ownership, also called "component lifetime". Since a component can implement many interfaces, interfaces must be reference counted.

All XPCOM objects keep track of how many references to it are being used and they delete themselves when the count reaches zero. This functionality is implemented in the `nsISupports` interface, which all XPCOM interfaces inherit. The `nsISupports` interface also provides the `QueryInterface` method, which allows one to find out which interfaces a component supports at runtime.

3.2.2 XPConnect and XPIDL

Another important mechanism for component integration is XPConnect, which enables simple interoperability between XPCOM and JavaScript. XPConnect allows JavaScript objects to transparently access and manipulate XPCOM objects. It also allows JavaScript objects to export XPCOM-compliant interfaces that can be accessed by other XPCOM objects. This mechanism is used when a script accesses the DOM or when scripts access other XPCOM objects in Mozilla. An example of the use of XPConnect is when a native DOM method is called from JavaScript. The arguments passed have no types and it is the job of XPConnect to translate these arguments to the correct C++ types before the method is invoked. In addition, XPConnect must translate the return value from the method to a correct JavaScript value. When accessing the DOM, XPConnect also takes advantage of DOMClassInfo which will be explained in the next section.

All interfaces of an XPCOM object must be declared in XPIDL (Cross Platform Interface Definition Language) in order to work with XPConnect. An XPIDL compiler is used generate both C++ header files and XPConnect *typelib* files. The typelib files are binary representations of one or more interfaces, and are used when XPConnect accesses an interface. A typelib file consists of detailed information about each method and variable an interface consists of, and provides functionality to quickly map an interface id to an interface description.

3.2.3 Interaction

JavaScript is used in three different ways in Mozilla. The first (and most common) way to use JavaScript is to access and manipulate objects in the DOM to create a dynamic environment for documents and to access browser-related information. Because XPConnect uses DOMClassInfo (see the following paragraph) when accessing the DOM's XPCOM objects, a JavaScript programmer does not need to specify which interface of the DOM object he wants to access. On a second level, JavaScript code can access scriptable [1] XPCOM components that are not part of the DOM by using the `Components` object. At this level, the correct component and interface must be specified. The third and last way to use JavaScript in Mozilla is to write entire XPCOM objects in JavaScript. Other XPCOM objects can then call and use an XPCOM object written in JavaScript just like any other XPCOM object. Downloaded JavaScript code embedded in web pages typically falls in the first category only, since, by default, this code does not have access to any of Mozilla's components except for the DOM XPCOM objects. "Native" JavaScript scripts most often fall in the second and third category.

When a JavaScript program access and manipulates the DOM, DOMClassInfo [9] is used. DOMClassInfo serves two roles: interface flattening and implementing behavior that is not defined in the IDL description of the component. Usually, when one wants to communicate with another XPCOM object, one of the object's interfaces must be specified. This is also the case when the DOM XPCOM objects are accessed. But instead of having the JavaScript programmer specify the interface, XPConnect automatically finds the correct interface by using DOMClassInfo. The second important use of DOMClassInfo is to express things that cannot be expressed in the IDL. All the DOM XPCOM objects have helper classes that inherit a scriptable interface, and these helper classes are contacted whenever XPConnect does not find the correct method to call in any of the interfaces a DOM XPCOM object implements. By using helper classes, a DOM object can appear different when accessed from scripts instead of being accessed from another XPCOM object. An example of the use of DOMClassInfo is implementing array behavior. When a script executes "`history[0]`" this will magically be converted to "`history.item(0)`" by using the helper class of the History object. Another example is when a script tries to set the `window.location` property. In the IDL definition of the `window` object there is no setter function for the `window.location` property. Therefore, in the helper class of the `window` object, `window.location` is converted to `window.location.href`, which is most often what the JavaScript programmer is setting when he tries to change the location property of the window.

When Mozilla starts the SpiderMonkey interpreter, XPConnect builds a number of wrapper objects that are responsible for making all the DOM objects available into the JavaScript engine. Whenever a script tries to access an object in the DOM, the wrapper object is contacted and the wrapper object then calls the correct XPCOM object by using DOMClassInfo. The arguments are converted to the correct type using the code contained in typelib files. An illustrative example of this is when a JavaScript script calls `document.getElementById(...)`. In this case, first XPConnect finds that `document` is a property of the `window` object. It then examines the interfaces for the `window` object to see if a `GetDocument` method is implemented. The return value from `GetDocument`

[1] Scriptable means that it can be accessed from scripts.

is an `nsIDOMDocument`. XPConnect then searches for a method called `getElementById` by looking at the `nsIDOMDocument` interface, and, finally, it calls the method.

An important part of the interaction between SpiderMonkey and Mozilla is the SecurityManager. The SecurityManager is implemented as a XPCOM object and is responsible for enforcing the security mechanisms that all downloaded JavaScript programs are subject to. The SecurityManager keeps information about the script that currently executes within SpiderMonkey. Whenever a script tries to access information outside the engine, the SecurityManager grants or denies the access based on the sand-box policy, the same-origin policy, and the signed-script policy.

3.2.4 LiveConnect

LiveConnect [8] is a technology used to enable communication between JavaScript, Java applets and other plug-ins. Using LiveConnect, JavaScript code can interact with standard Java system classes built into the browser, access downloaded Java applets, and communicate with Java-enabled Navigator plug-ins. The technology also allows applets and Java-enabled plug-ins to interact with JavaScript scripts, reading and writing JavaScript object properties and calling JavaScript methods. The communication between applets/plug-ins and scripts is still restricted by the same-origin policy (see Section2), which means that a script can only call an applet's method if both the script and the applet have the same origin or if the script is signed and has the `UniversalBrowserRead` capability.

LiveConnect is a very powerful technology, and opens new ways for JavaScript programs to interact with their execution environment. A stand-alone JavaScript program cannot access the filesystem, but using LiveConnect, JavaScript programs can access Java classes that allow reading and writing to files and directories. This of course assumes that the script is signed and has the correct privileges.

4. Auditing JavaScript Code Execution

When designing an auditing system there are several issues that have to be addressed: where to integrate the auditing system, what should the auditing information include, and how should the auditing information be represented.

The main challenges in developing an auditing system for SpiderMonkey/Mozilla are to achieve completeness and correctness as much as possible. When the JavaScript execution environment is initialized, a large number of function callbacks are registered in the JavaScript engine. These hooks are used when scripts try to access a specific property or method that are not native in the engine. This means that

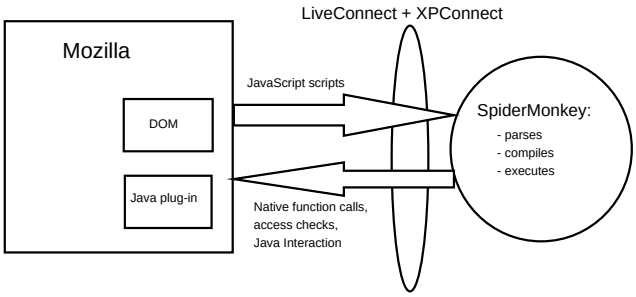

Figure 1. Basic interaction between SpiderMonkey and Mozilla.

a JavaScript program that executes in SpiderMonkey communicates with Mozilla in many different ways. Though most calls are directed to the DOM and forwarded using XPConnect, some calls go directly to the SecurityManager or are transferred through LiveConnect to the Java Virtual Machine (JVM). It is up to the embedding application, in this case the Mozilla browser, to define and implement all the correct function callbacks. To ensure completeness in our auditing system, all these calls must be intercepted and logged.

The auditing system must also be capable of differentiating between "native" scripts that execute on behalf of the browser, and scripts that are downloaded as part of HTML pages. Since "native" JavaScript code executes with all privileges set, these scripts can perform any operation that is allowed to the browser program itself and should not be audited. SpiderMonkey is a stand-alone JavaScript engine without explicit knowledge of whether the currently executing script is "native" or downloaded. Most of the auditing is therefore done in XPConnect, which is the layer between the JavaScript engine and Mozilla [2]. In addition, some auditing is also done in DOMClassInfo, in LiveConnect, and in the SecurityManager. Figure 1 shows the basic interaction between SpiderMonkey and Mozilla. As explained, our auditing mechanism is mainly implemented in the XPConnect and LiveConnect layer, though some calls do not use these layers and must be audited elsewhere.

Another important choice in designing the auditing system is deciding what to audit and how to represent the logging information. Due to the nature of the communication between SpiderMonkey and Mozilla, we decided to focus on the auditing of method calls and property getters and setters. As mentioned earlier, there is a mediated communication between SpiderMonkey and Mozilla, and auditing must be able to discern what are the security-sensitive part of this chain of invocation. An example of this is

[2] With the notable exception of communication with the Java Virtual Machine through LiveConnect.

when a script executes `document.location.href = "somewhere"`. If the document property has never been accessed before, XPConnect first gets the `document` object, then gets the `location` object, and finally sets the `href` property. The fact that XPConnect first gets the `document` object and `location` objects is not important, since these get operations are only intermediate operations that are done in order to set the `href` property. Intermediate operations are consequently not logged.

When SpiderMonkey communicates with its environment all arguments in getters/setters and method calls will be passed as *jsvals*, which is an internal JavaScript engine type. These arguments must be converted to the correct type, before any logging can be done. A jsval is one of 7 different types: string, null, void, double, int, boolean, or object. When a JavaScript script executes a method call, the types of the arguments given are compared to the types of the arguments in the corresponding method signature. Depending on the comparison between these types, different conversions are done. The conversion also depends on which technology is used in the intermediate communication, either LiveConnect or XPConnect. An example is when a script calls a method that expects a string argument, but instead passes an object. XPConnect will automatically call the `toString` method on the object to get a string representation of the object. The conversion will be different if the method call is forwarded through LiveConnect to the Java Virtual Machine. In our system, all types are converted to a string representation before they are written to the audit log.

Figure 2 shows a simple output from the auditing system when a script executes the instruction `document.location = "http://www.newlocation.com";`. The audit information produced is XML-encoded and includes a timestamp, the origin of the script, the method, the method arguments, and the return value. The *time* element is the time when the operation was executed. The *host* element is the origin of the script that executes the operation. The *method* element includes information about the name of the getter/setter/method call, and which object the getter/setter/method was called on. It is important to include the *object* information, since there are examples where there exist different methods with the same name, e.g. `window.open(..)` and `document.open(...)`. The *arguments* element give all the arguments with their corresponding type and value. If the *type* of the *argument* is "object", the corresponding class of the object is written in the *type* element. The *return* element contains the result of the operation. This element has two attributes: *bool*, which says if the operation was successful or not, and *param*, which gives the type and value of the return value.

```
<call>
  <time>Thu Jul 1 15:58:34 2004</time>
  <host>http://www.simplehost.com</host>
  <method>
    <name>location</name>
    <object>HTMLDocument</object>
  </method>
  <arguments>
    <argument>
      <type>STRING</type>
      <value>http://www.newlocation.com</value>
    </argument>
  </arguments>
  <return>
    <bool>true</bool>
    <param>
      <type>STRING</type>
      <value>http://www.newlocation.com</value>
    </param>
  </return>
</call>
```

Figure 2. XML encoding of JavaScript events.

5. Detection

In this section we describe how malicious JavaScript code can be detected leveraging the implemented auditing facility and a simple intrusion detection system (IDS). Intrusion detection systems [10] can be separated into two main categories: anomaly detection systems and misuse detection systems. Anomaly detection systems compare the behavior of a script to the "normal" behavior of scripts, and interpret deviations from the "normal" behavior as a problem. Misuse detection systems are based on comparing the operation of a script to some predefined attack scenarios (also called "signatures"). The auditing system we have developed does not restrict the IDS to be of any specific type. However, the IDS used in the following two examples is a misuse detection system. We show how malicious JavaScript code can be detected by comparing the output of the auditing system to known attack signatures.

5.1. Example 1

In the first example we look at a simple cross-site scripting attack (see Section 1). A user visiting a malicious web site, say *www.evil.com*, can be tricked into clicking on the following link: *<a href="http://www.trusted.com/<script>document.location='http://www.evil.com/cookie.cgi?'+document.cookie</script>">Click here to collect your prize*. When the user clicks on the link, an HTTP request will be sent to *trusted.com*, requesting the page "*<script>document.location='http://www.evil.com/cookie.cgi?'+document.cookie</script>*". The trusted host will receive the request and check if it has the resource which is being requested. When the *trusted.com* host does not find the requested page, it will send a return

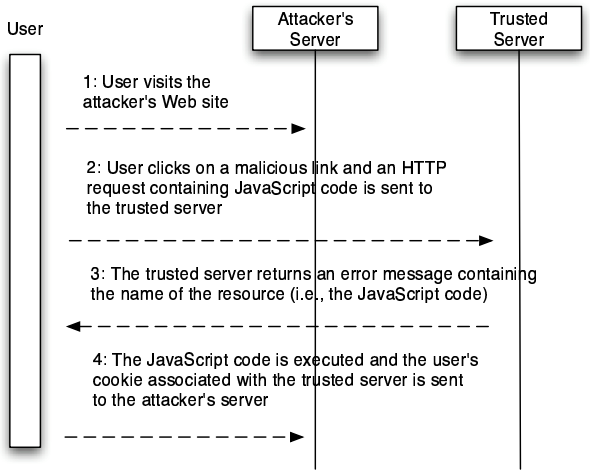

Figure 3. A typical cross-site scripting scenario.

message with the error code 404: "File does not exist". The web server may also decide to include the requested filename in the return message, to specify which file was not found. If this is the case, the filename (which is a script) will be sent from the *trusted.com* web server to the user's browser and will be executed in the context of the *trusted.com* origin. When the script is executed, the cookie set by *trusted.com* will be sent to the malicious web site as a parameter of the invocation of the *cookie.cgi* server-side script. The cookie will be saved and later used by the owner of the *evil.com* site to impersonate the unsuspecting user with respect to *trusted.com*. Figure 3 shows this scenario. This attack exemplify how a malicious hacker can perform an attack that bypasses the same-origin check to execute JavaScript code with the privileges of someone else. Figure 4 shows the script that is executed and Figure 5 shows the audit information produced by the auditing system.

Based on the information we have about cross-site scripting attacks, the intrusion detection system is implemented as a state-transition model that includes the following check:

```
if((event.method.name==cookie)&&(event.arguments==0)){
  state1.add(event.host);
}
if((event.method.name==location)&&(event.arguments!=0)
    &&(state1.includes(event.host))){
  log("You may have been exposed to
       cross site scripting");
}
```

Whenever a script reads the cookie associated with the script's origin, the origin of the script is added to the list

```
<script>
document.location='http://www.evil.com/cookie.cgi?'+
document.cookie;
</script>
```

Figure 4. Cross-site scripting example.

state1. To separate between property getters and property setters, we check if the property operation has any arguments. If the property operation has arguments, it is a set operation. If the property operation does not have any arguments, it is a get operation. When a script now sets the location of the document, the IDS checks if the origin of the script that performed the operation is already in the list *state1* and detects a possible cross site scripting attack.

This is a very simplified signature that addresses one specific event only. However, the example shows how the audit information can be used to evaluate a script's behavior with respect to a predefined attack scenario.

5.2. Example 2

The second example is a script that calls `window.open()` every time the user unloads the document. The HTML source is shown in Figure 6, and parts of the auditing information produced is shown in Figure 7. The audit information shown will be logged every time the user tries to exit the document. This is an example of a script that does not necessarily cause a security threat to the user, but is abusing the browser's resource (and can be problematic for an inexperienced user). In the state-transition based IDS we add the following checks:

```
if((event.method.name==open)&&
    (event.method.object=="window")){
  if(stateW4.includes(event.host)){
    log("Script has opened 5 windows.
        Possibly a malicious script!")
  }
  else if(stateW3.includes(event.host)){
    stateW3.delete(event.host);
    stateW4.add(event.host);
  }
  else if(stateW2.includes(event.host)){
    stateW2.delete(event.host);
    stateW3.add(event.host);
  }
  else if(stateW1.includes(event.host)){
    stateW1.delete(event.host);
    stateW2.add(event.host);
  }
  else{
    stateW1.add(event.host);
  }
}
```

The IDS checks how many times a script calls `window.open()` and identifies the script as malicious if it opens more than five windows.

```
<call>
  <time>Tue Jul 13 11:29:39 2004</time>
  <host>http://trusted.com</host>
  <method>
    <name>cookie</name>
    <object>HTMLDocument</object>
  </method>
  <arguments>
  </arguments>
  <return>
    <bool>true</bool>
    <param>
      <type>STRING</type>
      <value>My_cookie</value>
    </param>
  </return>
</call>
<call>
  <time>Tue Jul 13 11:29:39 2004</time>
  <host>http://trusted.com</host>
  <method>
    <name>location</name>
    <object>HTMLDocument</object>
  </method>
  <arguments>
    <argument>
      <type>STRING</type>
      <value>
      http://www.evil.com/cookie.cgi?My_cookie
      </value>
    </argument>
  </arguments>
  <return>
    <bool>true</bool>
    <param>
      <type>STRING</type>
      <value>
      http://www.evil.com/cookie.cgi?My_cookie
      </value>
    </param>
  </return>
</call>
```

Figure 5. Audit information produced by the script in Example 1.

```
<html>
<body onUnload="reopen()">
  <script>
    function reOpen(){
      window.open(self.location,");
    }
  </script>
</body>
</html>
```

Figure 6. HTML containing a script that causes the window to never close.

```
<call>
  <time>Tue Jul 20 10:05:34 2004</time>
  <host>http://www.somehost.com</host>
  <method>
    <name>open</name>
    <object>window</object>
  </method>
  <arguments>
    <argument>
      <type>OBJECT:location</type>
      <value>
      http://www.cs.ucsb.edu/~hallarak/
      </value>
    </argument>
    <argument>
      <type>STRING</type>
      <value>
      </value>
    </argument>
  </arguments>
  <return>
    <bool>true</bool>
    <param>
      <type>OBJECT:window</type>
      <value>
      [object window @ 0x8865950]
      </value>
    </param>
  </return>
</call>
```

Figure 7. Audit information produced every time the user tries to unload the document in Example 2.

5.3. Example 3

Figure 8 shows an HTML document containing a script that calls `window.alert()` every 100 milliseconds. This causes an alert box to appear in the browser window constantly, and the user cannot exit the window. Figure 9 shows the audit information produced every time the user tries to close the alert box. As in the previous example with `window.open()`, this malicious script is detected using a signature that keeps track of the number of times a certain method is called.

6. Evaluation

The overhead introduced by our auditing system is highly dependent on the JavaScript code that executes in SpiderMonkey. Scripts that do not communicate with the DOM or other parts of Mozilla will not be audited at all, and the overhead is, in these cases, very small. However, most useful JavaScript programs interact with their host environment to make static HTML pages more interactive.

To evaluate the performance of the auditing system, we designed three scripts that interact with the DOM in different ways. All the scripts consist of only one method call,

```
<html>
  <body>
  <script language=JavaScript>
    function alrt() {
      alert("Never Close!");
    }
    dummy=setInterval("alrt()","100");
  </script>
  </body
</html>
```

Figure 8. HTML containing a script that causes an alert box to constantly appear in the browser window.

```
<call>
  <time>Fri Jul 30 10:47:33 2004</time>
  <host>http://www.somehost.com</host>
  <method>
    <name>alert</name>
    <object>window</object>
  </method>
  <arguments>
    <argument>
      <type>STRING</type>
      <value>Never Close!</value>
    </argument>
  </arguments>
  <return>
    <bool>true</bool>
    <param>
      <type>VOID</type>
    </param>
  </return>
</call>
```

Figure 9. Audit information produced in Example 3.

Script	Number of operations	Time without auditing	Time with auditing	Percent overhead
Script1	10	0.013sec	0.016sec	23%
Script2	250	0.255sec	0.327sec	28%
Script3	500	0.498sec	0.669sec	34%

Table 1. Overhead introduced by the auditing system.

that is, `document.write("message")`. The only difference is how many times this method is called. We performed our experiments on an Intel Pentium 4, 2.4GHz processor and a WD Caviar 7200rpm hard disk. The operating system is Red Hat Linux, kernel version 2.4.22. Our auditing system is developed for Mozilla version 1.7 Beta. Table 1 shows the time it takes to run the three different scripts with and without the auditing system. Each script has been run 100 times. The number of operations is the number of audited operations, i.e., the number of `document.write()` invocations that appear in the audit log. The overhead caused by the auditing is mainly due to file I/O, when writing auditing information to the log file. There is a trade-off between the time overhead and how fast an attack can be detected. Using more buffering, the overhead can be reduced, while increasing the detection time. This parameter can be configured in our system to match the security requirements of different installations. As the figure shows, the overhead caused by the auditing system increases with the number of operations. Note that we have not focused on optimizing the auditing system, and the overhead can be easily decreased using more efficient I/O buffering techniques.

7. Conclusions and future work

In this paper we have presented a novel auditing facility for JavaScript code execution. This mechanism is combined with an intrusion detection system to detect malicious JavaScript code, including the very common cross-site scripting attacks. In designing and implementing the auditing mechanism, we found that the complexity of the interaction between the JavaScript interpreter and the browser required careful evaluation of design trade-offs.

We implemented our system and evaluated the overhead introduced. Even though, our system introduces a somewhat substantial overhead with respect to the execution time of non-audited scripts, the security benefit may be worthwhile in security-critical environments.

Future work will focus on implementing more efficient

auditing techniques and in extending our intrusion detection system with more sophisticated signatures.

Acknowledgments

This research was supported by the National Science Foundation under grants CCR-0209065 and CCR-0238492. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation or the U.S. Government.

References

[1] ECMA-262, ECMAScript language specification, 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[2] V. Anupam and A. J. Mayer. Secure Web Scripting. *IEEE Internet Computing*, 2(6):46–55, 1998. citeseer.ist.psu.edu/anupam98secure.html.

[3] R. P. D. Peterson, M. Bishop. A flexible containment mechanism for executing untrusted code. 2002. http://nob.cs.ucdavis.edu/~bishop/papers/2002-sandbox/2002-sandbox/2002%-sandbox.html.

[4] I. C. P. D.Boswell, B.King. *Creating Applications with Mozilla*. September 2002. http://books.mozdev.org/chapters/.

[5] I. O. Doug Turner. Creating XPCOM Components. December 2003. http://www.mozilla.org/projects/xpcom/book/cxc/index.html.

[6] B. Eich. Embedding the JavaScript Engine, A Bare Bones Tutorial. February 2000.

[7] D. Flanagan. *JavaScript: The Definitive Guide, 4th Edition*. December 2001.

[8] S. Furman. Java Method Overloading and LiveConnect 3. http://www.mozilla.org/js/liveconnect/lc3_method_overloading.html.

[9] F. Guisset. The Mozilla DOM hacking Guide, February 2004. http://www.mozilla.org/docs/dom/mozilla/hacking.html.

[10] R. Kemmerer and G. Vigna. Intrusion Detection: A Brief History and Overview. *IEEE Computer*, pages 27–30, April 2002. Special publication on Security and Privacy.

[11] J. T. Moore. Mobile Code Security Techniques. Technical Report MS-CIS-98-28, 1998. citeseer.ist.psu.edu/moore98mobile.html.

[12] mozilla.org. JavaScript C Engine Embedders's Guide.

[13] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl security model. *Lecture Notes in Computer Science*, 1419:217–??, 1998. citeseer.ist.psu.edu/ousterhout97safetcl.html.

[14] D. A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, December 1999.

[15] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993. citeseer.ist.psu.edu/wahbe93efficient.html.

[16] L. Wood. Document Object Model (DOM) Level 1 Specification. Technical Report REC-DOM-Level-1-19981001, W3 Consortium, 1998. Version 1.0.