Light-Weight Transparent Defense Against Browser Cross-Frame Attacks Using *Script Accenting*

Shuo Chen

March 14, 2007

Technical Report
MSR-TR-2007-29

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Light-Weight Transparent Defense Against Browser Cross-Frame Attacks Using *Script Accenting*

Shuo Chen
Cybersecurity and Systems Management Group
Microsoft Research, Redmond, WA 98052, U.S.A.
shuochen@microsoft.com

Abstract

The browsers' isolation mechanisms are critical to users' safety and privacy on the web. Achieving proper isolations, however, is very difficult at both the policy-specification level and the implementation level. This paper is focused on the implementations of browser isolation mechanisms. As a concrete example, we discuss the enforcement of the well-defined cross-frame isolation policy, which is supposed to prohibit a script from one Internet domain to access objects in a frame of another domain. Historical data show that even for such a seemingly simple policy, the current implementations of the enforcement mechanisms are surprisingly error-prone, and have been exploited on most major browser products. In this paper, we proposed the script accenting *technique as a light-weight transparent defense against the cross-frame attacks. The basic idea is to introduce domain-specific "accents" to the scripts and the object names so that two frames cannot communicate/interfere if they have different accents. The mechanism has been prototyped on Internet Explorer. Our evaluations showed that all known cross-frame attacks were defeated, and the proposed mechanism is fully transparent to existing web applications. The end-to-end measurement about user's browsing experience did not show any noticeable slowdown.*

1. Introduction

Web browsers can render contents originated from different Internet domains. A major consideration of the web security is the appropriate enforcement of the *same-origin principle*. Although it has never been strictly defined, the principle can be loosely interpreted as "a script originated from one domain should not be able to read, manipulate or infer the contents originated from another domain", which is essentially the well-defined *non-interference property* [10] reflected in the web security context. The violation of this principle can result in severe privacy consequences, e.g., a script from an arbitrary website can steal the user's banking information or perform unintended money transfers from the user's account.

The same-origin principle violations can happen due to the insufficient script-filtering of the web application on the server, or due to flaws on the browser domain-isolation mechanisms. The script-filtering flaws are commonly referred to as the cross-site scripting (or XSS) bugs [19], in which malicious scripts from attackers can survive the filtering and later get executed in the same security context as the authentic web application. Researchers have been proposing techniques to find the XSS bugs or defeat the XSS attacks. On the browser, the same-origin principle violations are due to the improper isolation of the contents from different domains. It is challenging to precisely specify policies for the "proper isolation" to guarantee privacy and allow reasonable browser functionalities. For example, the policies for the browser cache, the clipboard, the hyperlink coloring, the *XMLHTTPRequest* object [18] and the *JSONRequest* object [8] are being discussed by researchers. It is clearly a necessity to define and standardize the same-origin policies for the browser.

The focus of this paper, however, is not on how to specify the same-origin policies for the browser. We found that even for a well-specified policy, the implementation of the enforcement mechanism can be surprisingly hard and error-prone. A concrete example is that the cross-frame same-origin policy, which states that the script running inside a frame of http://a.com is not allowed to access the objects inside a

frame of http://b.com. Bugs in the enforcement mechanism of this policy have been discovered on major browsers, including Internet Explorer (IE), Firefox, Opera and Netscape Navigator [1][2][3]. Malicious websites can exploit these bugs to steal users' personal data, spoof the browser graphical interface, and impersonate users to do almost anything that can be done using the browser.

This paper presents a focused study of IE's cross-frame isolation mechanism and the bugs discovered in the past. We found that the current mechanism can be bypassed or fooled because of the navigation mechanism, the function aliasing, the excessive expressiveness of navigation methods, the semantics of user events and IE's interactions with other system components. The attacks are highly heterogeneous, and it would be very challenging to exhaustively reason about every scenario that the cross-frame isolation mechanism may face. Of course, the unsolved challenge suggests that the browser may have new bugs of this type discovered in the future, exactly like the situation that we have with the buffer overrun bugs.

In this paper, we propose a light-weight transparent defense technique to defeat cross-frame attacks. The technique is based on the notion of "script accenting". The basic idea is analogous to the accent in human languages, in which the accent is essentially an identifier of a person's origin that is carried in communications. We slightly modified a few functions at the interface of the script engine and the HTML engine so that (1) each domain is associated with a random "accent key", and (2) scripts and HTML object names are in their accented forms at the interface. Without needing an explicit check for the domain IDs, the accenting mechanism naturally implies that two frames cannot communicate/interfere if they have different accent keys.

The concept of script accenting provides a higher assurance for the implementation of the cross-frame defense. We are able to confidently define the *script ownership* and the *object ownership*, which are easily followed in our implementation without any confusion. A prototype of the technique has been implemented for IE. The evaluation showed that all known cross-frame attacks[1] were defeated. Moreover, because the accenting mechanism only slightly changes the interface between the script engine and the HTML engine, it is fully transparent to web applications. Our stress test showed a 3.6% worst-case performance overhead, but the end-to-end measurement about user's browsing experience did not show any noticeable slowdown.

The rest of the paper is organized as follows: Section 2 discusses the related work. We briefly introduce the basic of IE's cross-frame isolation mechanism in Section 3. Section 4 presents a case study of real-world attacks. We discuss the design and the implementation of the script accenting mechanism in Section 5, followed by the evaluations in Section 6. Section 7 concludes the paper.

2. Related Work

Researchers have been studying security issues related to the same-origin principle, among which the cross-site scripting (XSS) problem has attracted much attention. Although it is not the focus of this paper, we summarize a few interesting projects here. Livshits and Lam proposed a static analysis technique to find XSS bugs in Java applications [16]. Johns studied XSS attacks and identified the prerequisites for the attacks to hijack sessions. He proposed the *SessionSafe* approach that removes the prerequisites to protect browser sessions [12]. Because XSS attacks are due to the failures of script filtering, Xu et al proposed using taint tracking to detect the attacks [17]. The browser cross-frame attacks discussed in this paper are a different type of attacks. They exploit flaws in the browser isolation mechanism rather than the script-filtering mechanism on the web application.

Interesting research has also been done about the policies of the browser isolation mechanism. Significant effort is spent on the discussion and the standardization of the browser's mechanisms to securely retrieve data from servers, among which *XMLHTTPRequest* [18] and *JSONRequest* [8] are the

[1] There is one cross-frame bug reported to Microsoft that we cannot reproduce.

representatives. In addition to the effort on data retrieval mechanisms, researchers also found that the timing characteristics of caches and the coloring of visited links allow malicious scripts to infer certain browser states and thus track users' browsing histories. Accordingly, they refined the same-origin policies for browser caches and visited links [5][9][11].

The idea of script accenting resembles a number of randomization-based security techniques that defeat memory corruption attacks. The *PointGuard* technique protects return addresses and function pointers using the XOR operation with a random number generated at runtime so that memory-based exploits cannot succeed in tempering with the control flow [7]. Instruction Set Randomization (ISR) is a processor-level technique to defeat binary-code injection attacks. By introducing a key-register, the processor is able to decode and execute the binary code that was encoded at the load time, but not the code injected during the execution [4][15]. As we discuss later, the primitive of the accenting operation is also XOR. Nevertheless, our technique falls in the areas of access control and sandboxing, while the above randomization techniques are developed to ensure data/code authenticity.

3. The Cross-Frame Isolation Mechanism of IE

We start the technical discussions with a short introduction of the cross-frame isolation mechanism. In IE, each HTML document is hosted in a frame (or an inline frame)[2]. A browser window is the top-level frame, which hosts the top-level document that may contain other frames. Although many web applications need cross-frame communications to coordinate the actions between the frames from the same Internet domain, it is a basic security requirement that the frame from different domains cannot communicate with each other. IE implements a security mechanism to check whether a cross-frame communication is permitted.

Figure 1 shows `Frame1` and `Frame2` that represent two frames in the browser. The document in `Frame1` is downloaded from http://a.com. The objects in the frame are stored in a DOM tree (i.e., a Document Object Model tree). The root of the DOM tree is a `window` object. Note that "`window`" in the DOM terminology actually represents a frame, which is not necessarily the entire browser window. The children of `window` include: `location`, which is the URL of the document; `event`, which is the event received by this frame; `document`, which is the parsed HTML document contents; `history`, which is a collection of the URLs having been visited in this frame. The objects `body` and `scripts` have the common parent object `document`. The `body` object contains primarily the static contents to be rendered in the frame. `Scripts` is a collection of scripts that manipulate the DOM tree of its own frame and communicate with other frames. These scripts are compiled from the script source text embedded in the HTML file or passed from another frame. They are in a format of "byte-code", essentially the instruction set of the script engine.

Each frame has a script runtime, which includes a stack, a heap, a program counter pointing to the current instruction in the `scripts` object, and a set of window references (to be discussed in the next paragraph). When the script accesses a DOM object, the script runtime executes an "`LoadMember baseObj,nameString`" instruction to get the object's reference. For example, to access `document.body`, the script runtime executes "`LoadMember RefDocument, 'body'`", where `RefDocument` is a reference to the `document` object. `LoadMember` is an instruction to look up a child object name and return the object's reference.

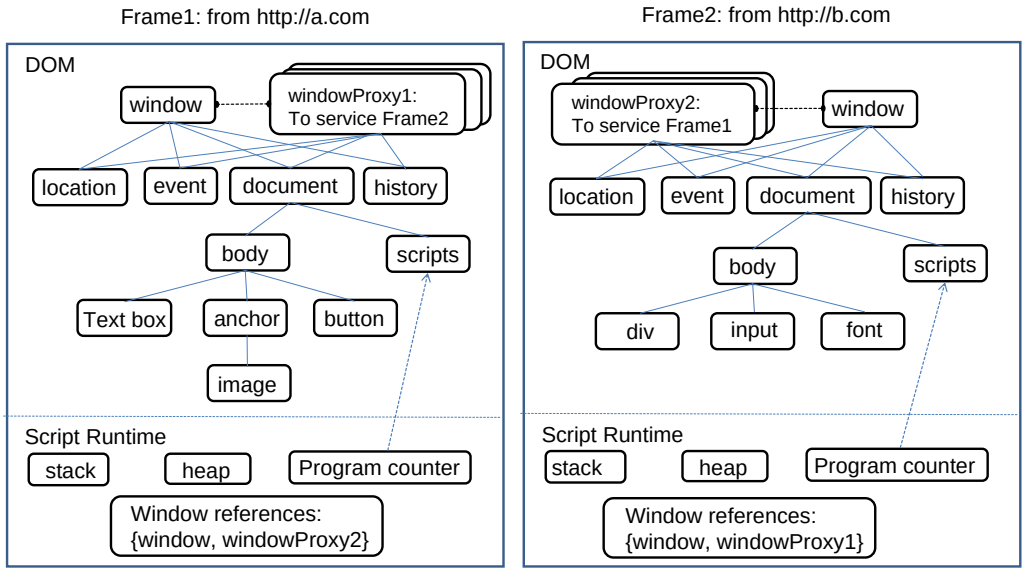
The script runtime keeps a `window references` object. The reference to the `window` object of `Frame1` is in the `window references` of `Frame1`'s runtime, so any script running in `Frame1` can get the reference to every object in its own DOM and manipulate it. Hypothetically, if a script running in

[2] The security aspect of the inline frame is very similar to that of the frame. In the rest of this paper, the term "frame" also refers to the inline frame.

`Frame2` from http://b.com had a reference to the `window` object of `Frame1`, the script could also totally control the DOM of `Frame1`, which violates the same-origin policy. It is therefore a crucial security requirement that the reference to the `window` object should never be passed outside its own frame. Instead, `Frame2` has a window proxy `windowProxy1` to communicate with `Frame1`. The window proxy is similar to the window object, but it is specifically created for `Frame2` to access `Frame1`. The window proxy is the object in which the cross-frame check is performed: for any operation[3] to get the reference of a child of `windowProxy1`, a domain-ID check is made to ensure that the domains of `Frame1` and `Frame2` are identical. For example, assuming a script is running in `Frame2`, and `windowProxy1` is represented as `WND1` in the script, then the script expression "`WND1.document`" will fail with an access-denied error, because `WND1` (i.e., `windowProxy1`) is the proxy between two frames from different domains. The domain-ID check in the window proxy is simply a string comparison to check if the two domains expressed in the plain text format are identical. Because of the simplicity, the string comparison per se is robust.

The above described mechanism seems to provide a flawless isolation between frames from different domains. However, in the next section, we discuss a number of real attacks that bypass or fool it to allow a malicious script to control a frame of a different domain.

Figure 1: Cross-Frame References and Frame Isolation Between Frame1 and Frame2



4. Real-World Cross-Frame Attack Examples

The isolation mechanism presented in Figure 1 is designed to prevent a script from http://a.com to access the DOM from http://b.com. The implicit assumptions are (1) every cross-frame communication must go through the window proxy, (2) the window proxy has the correct domain-IDs of the accessor frame and the accessee frame. We studied Microsoft's internal product security database, and found that all discovered cross-frame bugs are because of the invalidity of these assumptions. There are secret execution paths in the system to bypass the check or feed incorrect domain-IDs to the check. These exploit scenarios take advantage of the navigation mechanism, IE's interactions with other system components, the function aliasing in the script runtime, the excessive expressiveness of frame navigations, and the semantics of user events. In this section, a number of real attacks are discussed to show that it is very hard to exhaustively reason about all possible execution scenarios.

[3] The write operation to the `location` object is an exceptional case. It does not follow the same-origin policy. The cross-domain check is explicitly bypassed for this operation.

In these examples, we assume the user's critical information is stored on the website http://payroll, and the user visits an unknown website http://evil. The goal of http://evil is to steal the payroll information and/or actively change the direct deposit settings of the user, for example. We use "`doEvil`" to represent a piece of malicious Javascript payload supplied by http://evil that does the damages. In the following discussion, the attacker's goal is to execute `doEvil` in the context of http://payroll.

4.1 Exploit of the Interactions Between IE and Windows Explorer

IE and Windows Explorer[4] have tight interactions. For example, if we type "`file:c:\`" in the address bar of IE, the content area will load the folder of the local C drive. Similarly, if we type "`http://msn.com`" in the address bar of Windows Explorer, the content area displays the homepage of MSN. On Windows XP prior to Service Pack 2, this convenient feature gave the attacker a path to bypass the security check.

Attack 1. Figure 2 illustrates an attack where the script of http://evil loads a frame for http://payroll and manipulates it by injecting `doEvil` into the frame. The script of http://evil, running in `Frame1`, first opens the http://payroll page in `Frame2`, and then navigates `Frame2` to the URL "`file: javascript:doEvil`". Because the protocol portion of the URL is "`file:`", IE passes the URL to Windows Explorer. Windows Explorer treats it as a normal file-URL and removes "`files:`" from it, and treats the remainder of the URL as a filename. However, the remainder is "`javascript:doEvil`", so Windows Explorer passes it back to IE as a javascript-URL. According to the "javascript:" protocol, navigating `Frame2` to such a URL is to add `doEvil` into the scripts of `Frame2` and execute it [14]. Normally, one frame navigating another frame to a javascript-URL is subject to the same-origin policy. For example, the statement `open("javascript:doEvil","frame2")` will result in an access denied error. However, by passing the URL to the Windows Explorer, `Frame2` receives the script as if it was passed from the local machine, not from the Internet, which bypasses the same-origin policy check.

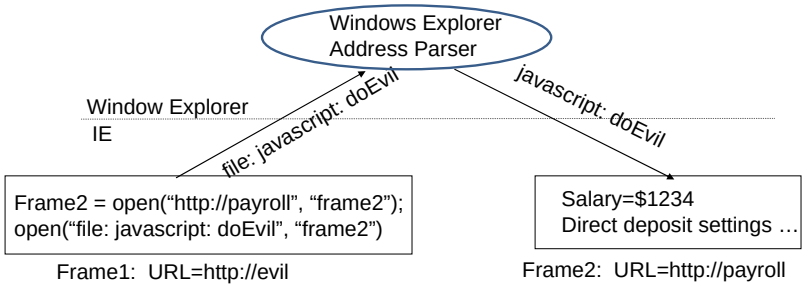


Figure 2: Illustration of Attack 1

4.2 Exploit of the Function Aliasing in the Script Runtime

In Javascript, a method (i.e., a member function) itself is also an object, and thus its reference can be assigned to another object, which is essentially an alias of the function. The aliasing combined with the frame navigation could result in a very complicated scenario where the real meaning of a script statement is difficult to obtain based on its syntactical form.

Attack 2. The attack shown in Figure 3 has four steps: (1) `Frame1` loads the script from http://evil, which sets a timer in `Frame2` to execute a statement after one second; (2) the script makes `frame2.location.assign` an alias of `window.location.assign`. According to the DOM specification, the method `location.assign(URL)` of a frame is to navigate the frame to `URL`; (3) the script navigates its own frame (i.e., `frame1`) to http://payroll; (4) when the timer is expired, `location.assign('javascript:doEvil')` is executed in `Frame2`. Because of the aliasing, the statement really means `frame1.location.assign('javascript:doEvil')`. Despite that

[4] *Windows Explorer* is the application to display the local folders and files. It is sometimes referred to as the *Shell*.

it is physically a cross-frame navigation to a javascript-URL, the operation is syntactically an intra-frame operation, which does not trigger the cross-frame check. Then, `doEvil` is merged to the scripts of the http://payroll DOM, and get executed.

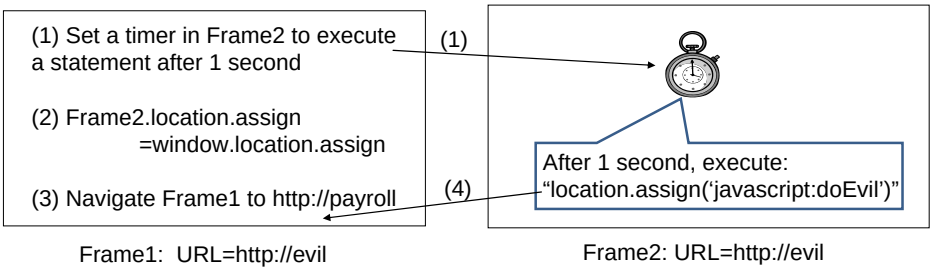


Figure 3: Illustration of Attack 2

4.3 Exploit of the Excessive Expressiveness of Frame Navigation Calls

The syntax of frame navigation calls can be very expressive. An attacker page can exploit the excessive expressiveness to confuse IE about who really initiates the operation.

Attack 3. Shown in Figure 4, `Frame0` from http://evil opens two frames, both loading http://payroll. These two frames are named `Frame1` and `Frame2`. Then the script running in `Frame0` executes a confusing statement `Frame2.open("javascript:doEvil", Frame1)`. This is a statement to navigate `Frame1` to the URL "`javascript:doEvil`", but the critical question is who initiates the navigation, `Frame0` or `Frame2`? In the unpatched versions of IE, `Frame2` is considered the initiator, because the `open` method is under `Frame2`. Therefore, the cross-frame check is passed because `Frame1` and `Frame2` are both from http://payroll. Similar to all previous examples, `doEvil` is then merged into `Frame1`'s scripts and get executed.

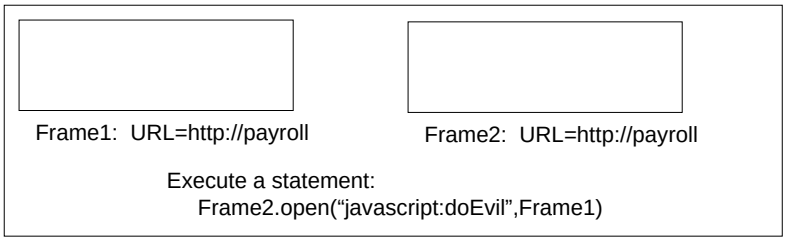


Figure 4: Illustration of Attack 3

4.4 Exploit of the Semantics of User Events

We have discussed a number of attacks in which a piece of script from the attacker frame can be merged into the scripts of the victim frame. The other form of attacks is to merge the victim's DOM into the attacker's DOM so that the attacker's script can manipulate it.

Attack 4. The DOM objects have the `setCapture` method to capture all mouse events, including those outside the objects' own screen regions. In the attack shown in Figure 5, the script from http://evil in `Frame0` creates `Frame1` to load http://payroll, then calls `document.body.setCapture()` to capture all mouse events so that they invoke the event handlers of the `body` element of `Frame0` rather than the element under the mouse cursor. When the user clicks inside `Frame1`, the event is handled by the method `body.onClick()` in `Frame0` because of the capture. Suppose the user clicks on the `font` object in `Frame1`, the DOM object `event.srcElement` in `Frame0` becomes an alias to the `font` object, according to the dentition of `event.srcElement`. Therefore, the script of `body.onClick()` can traverse in `Frame1`'s DOM tree as long as the traversal does not reach the window proxy level. In other words, `Frame1`'s `document` subtree is merged into `Frame0`'s DOM tree,

so the script can reference to the `document` object using `F1Doc = event.srcElement.parentElement.parentElement`. In particular, the execution of `doEvil` in `Frame1` can be accomplished by the simple assignment `F1Doc.scripts(0).text= doEvil`.

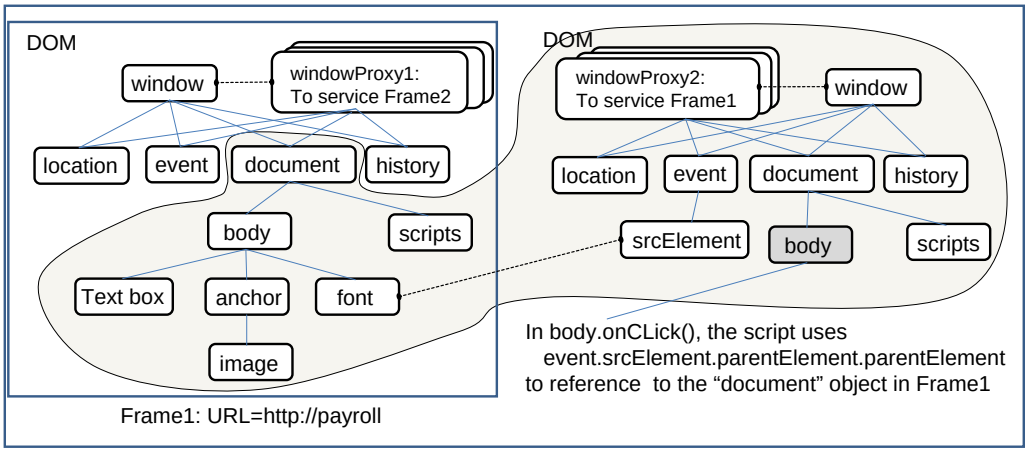


Figure 5: Illustration of Attack 4

4.5 Other Attacks

We studied other bug reports in the Microsoft product security database and were able to successfully reproduce other attacks, which we do not discuss in details in this paper. **Attack 5** can be launched when the victim page has at least one sub-frame. The attack script forces the sub-frame to receive the focus and then injects a script into it. **Attack 6** is similar to Attack 4, although literally it is not a cross-frame attack. It is an attack to cross the isolation boundary of the XML object, whose access control should also conform to the same-origin policy. Because of a race condition in the redirection mechanism of the XML object, the attack script is able to access the internal objects of an XML object from a different domain.

5. Design and Implementation of the Script Accenting Mechanism for Defense

We have discussed a number of real attacks in Section 4. The isolation failures are not because of any errors in the cross-frame check discussed in Section 3, but because of two reasons: (1) there exist unexpected execution scenarios to bypass the check; (2) the current mechanism is a single-point check buried deep in the call stack – at the time of check, there are confusions about where to obtain the domain-IDs of the script and the object. It is challenging for developers to anticipate all these highly heterogeneous scenarios because many code modules are involved, including the scripting engine, the HTML engine, the navigation mechanism, the event handling mechanism, and even non-browser components like Windows Explorer. Each of them has a large source code base which has been actively developed for more than 10 years. It is clearly a difficult task to guarantee that the checks are performed exhaustively and correctly.

We propose the *script accenting* as a defense technique. The technique takes advantage of the fact that the IE executable has a clean interface between the component responsible for the DOM (the HTML engine *mshtml.dll*) and the component responsible for the Javascript execution (the Javascript engine *jscript.dll*). Because by definition the cross-frame attack is caused by the script of one domain accessing the DOM of another domain, if both components carry their domain-specific accents in the communications at the interface, the communications can succeed only when the accents are identical. To achieve this, we assign each domain an *accent key*, which is only visible to the HTML engine and the Javascript engine, but not to the Javascript code. The main task of the implementation is thus to appropriately place accenting and de-accenting operations in the browser executable.

5.1 The Accenting/De-Accenting Primitive Operation

A simple design choice is about the primitive operation for accenting and de-accenting. In the current implementation, we generate a 32-bit random number (four-bytes) as the accent key for each domain. The primitive operation to accent/de-accent a message string is to XOR every 32-bit word in the string with the accent key. When there are one, two or three bytes remaining in the tail of the string, we mask the accent key to the same length, and apply the XOR operation.

We choose the XOR operation because of its simplicity and runtime efficiency. Of course, there can be many other primitive operations, as long as the operation is performed based on the accent key and the accented message is not forgeable by other domains. Nevertheless, choosing the primitive operation is orthogonal to the focus of our following discussion about how to apply the operation in the browser executable.

5.2 Accent Key Generation and Assignments

We keep a lookup table in *mshtml.dll* to map each domain name to an accent key. The keys are generated in a *Just-In-Time* fashion: immediately after the `document` object is created for each frame, we look up the table to find the key associated with the domain of the frame (if not found, create a new key for the domain), and assign the key to the `window` object (i.e., the frame containing the document).

When the `scripts` object is created, it copies the key from the `window` object. This is for the sake of runtime efficiency when the script runtime references the key later. Otherwise, it would be time-consuming for the script runtime to retrieve the key from the DOM because the script runtime and the HTML engine are implemented in different DLLs.

IE has the support for a frame to change its domain during the page rendering and the execution of its script [5]. For example, the *Virtual Earth* application (at http://map.live.com) initially runs in the domain http://map.live.com, and later changes its domain to http://live.com in order to communicate with other http://live.com services. To be compatible with this feature, we redo the key generation/assignment operations when the `document.domain` attribute is changed. Note that http://map.live.com and http://live.com are two unrelated domains once the domain changing operation is done, so each of them has its own accent key.

5.3 Design Principles

The attack examples show that it is challenging to guarantee the correctness of the current isolation mechanism because the developers need to reason about it as a system-wide property. Reasoning about the correctness of the script accenting mechanism is significantly easier because we only need to guarantee that every object is known by its accented name and that every script executes and travels in its accented form. In particular, the implementation needs to conform to three principles.

Principle of Script Ownership: One of the difficulties in the current window-proxy-based check is that at the time when the check is performed, it is hard to determine the origin of the script. Attack 2 and Attack 3 exemplify this difficulty. Our implementation follows the rule that the script always carries its owner frame's identity. The principle of script ownership states that *a script is owned by the frame that supplies the source code of the script, and should be accented at the time when its source code is supplied*. The rationale is that the source code supplier defines the behavior of the script, so we need to guarantee that the script is illegible to the frames from domains other than the source code supplier's domain. We will discuss in Section 6.1 that this principle eliminates the attacker's possibility of using wrong domain-IDs to fool the check.

[5] The detailed policy about domain changing is out of the scope of this paper. An article about this subject is located at http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/reference/properties/domain.asp.

Principle of Object Ownership: The principle of object ownership states that *every object is owned by the frame that hosts the DOM tree of the object, and is always known by its accented name*. The rationale of this principle is that an object can be referenced in many ways due to aliasing, so it is error-prone to determine the object's origin based on its syntactical reference. Instead, an object's origin should be only determined by the `window` object (i.e., the frame) of its DOM tree, because this ownership relation is established at the DOM creation time.

Principle of Self Defense: The isolation of our mechanism should not rely on the global knowledge about the browser implementation, but should assume the complicated logic of navigations and object references to be unknown and arbitrary. The principle of self defense states that *every DOM object has the responsibility to prevent itself from being accessed by a script from another domain, and each "scripts" object has the responsibility to resist any attempt of merging a script from another domain.*

5.4 Locating Single-Entrant Chokepoints for Accenting/De-accenting Operations

In addition to the design principles, correctly placing accenting/de-accenting operations in the browser executable requires the notion of "single-entrant chokepoint". We present an abstract illustration in Figure 6, and will discuss it with concrete scenarios in Section 5.5 and Section 5.6.

Let's assume Figure 6 is a call graph in which we want to select a function to perform an accent operation. There are multiple functions that produce `info`, which is the information to be accented. This information is passed along the arrows in the graph. To find an appropriate location to perform the accenting operation, we need to find a function that is a single-entrant chokepoint, i.e., a function called once and only once on every call path. Function A is not a chokepoint because it does not sit on the call path starting from Z. If we place the accenting operation here, there is no guarantee that `info` is accented. Although Function C is a chokepoint, it is not single-entrant because it can be called more than once. For our XOR-based primitive, applying the primitive twice on the same information would nullify the accenting operation. In the call graph, only function B is a single-entrant chokepoint. In our implementation, the call stacks of different scenarios were collected to form the call graphs to identify the single-entrant chokepoints.

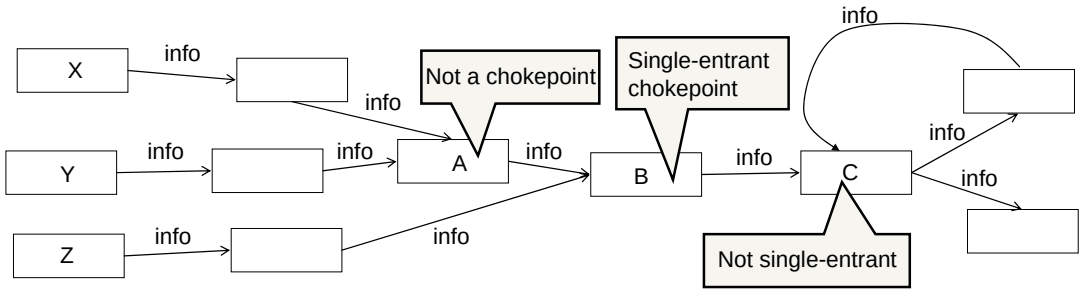


Figure 6: Locating the Single-Entrant Chokepoint

5.5 Accenting the Script Source Code to Defeat Script Merging Attacks

Many cross-frame attacks are because of script merging, as we showed in Section 4. In the browser, a text string can be sent to another frame and compiled as a script by (1) calling certain methods of the `window` object, including `execScript(ScriptSrc)`, `setTimeout(ScriptSrc,…)` and `setInterval(ScriptSrc,…)`, where `ScriptSrc` is the text string, or (2) navigating the frame to a Javascript-URL. The format of the Javascript URL is "`javascript:ScriptText`", where `ScriptText` is the script source code in the plain text format. There are many ways to navigate to a javascript-URL, such as the method calls "`open(…)`", "`location=…`", "`location.replace(…)`", "`location.assign(…)`" and HTML hyperlinks "`<base href=…>`", "`<a href=…>`", etc. Note that the Javascript function `eval` is to evaluate a text string in the current frame, so it is not a cross-frame operation.

For each invocation or navigation scenario, we obtained a call path. These paths form a call graph shown in Figure 7, where we omit many intermediate functions (represented as clouds). We observed that a single-entrant chokepoint for `execScript`, `setTimeout` and `setInterval` is `InvokeMemberFunc`, and a single-entrant chokepoint for all Javascript URL navigations is `InvokeNavigation`. Therefore, we can insert the accenting operation before `InvokeMemberFunc` and `InvokeNavigation`. At these two functions, it is straightforward to conform to the principle of script ownership: since the caller script supplies the source code of the script to be sent to another frame, the accent key should be taken from the frame hosting the caller script.

The call graph of script receiving is much simpler. In the receiver frame, because the `scripts` object in the DOM is in the "byte-code" format, any received script source code needs to be compiled before being merged into the `scripts` object of the receiver frame. Function `Compile` is the single-entrant chokepoint for this purpose, and it is an ideal location to perform the de-accenting operation.

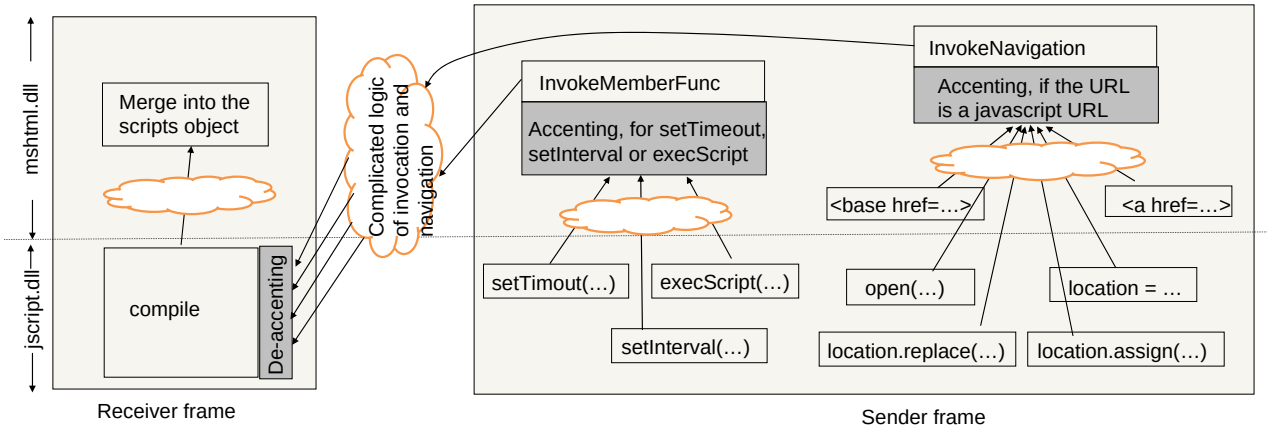


Figure 7: Accenting and De-Accenting of Script Source Code

Our implementation also reflects the principle of self defense. The logic of invocation and navigation, which we represent as a cloud, is highly complicated. As we discussed in Section 4, exploitable bugs have been discovered in the past in the logic. The advantage of our implementation is that the security does not rely on the logic correctness of the cloud because if the accented script is erroneously sent to a frame of a different domain, the compilation guarantees to fail.

5.6 Accenting the Object Name Queries to Defeat DOM Merging Attacks

Cross-frame attacks can also be caused by DOM merging, in which case an object can be directly accessed by a script running in another domain without going through the window proxy object.

A script references an object (e.g., `window.location`), an attribute (e.g., `window.status`) or a method (e.g., `window.open`) by name. The distinction between the terms "object", "attribute" and "method" is not important in our later discussion, so we use the term "object" for all of them.

To reference to an object, the script runtime iteratively calls into the DOM for name lookups. For example, the reference `window.document.body` is compiled into a segment of byte-code, which (1) gets the `window` object `O`, and looks up the name "document" under `O` to get the object referred to as `O1`; (2) looks up the name "body" under the object `O1` to get the object `O2`, which is the `body` object. Note that the mapping from a name to an actual DOM object is not necessarily injective, i.e., there can be different names mapped to the same object. In the example in Section 4.4, the font object could be referenced as `Frame1.document.body.children(3)` or `window.event.srcElement`. From the perspective of the script runtime, the execution paths of these two references are unrelated.

We obtained the call graph using various name querying scenarios, including the queries of objects, attributes and methods, as well as the aliases of them. Because IE uses the COM programming model [6], the browser objects are implemented as *dispatches*, each represented by a dispatch ID. Obtaining the dispatch ID is a necessary step before a script can do anything to the object. In the script runtime, a single-entrant chokepoint for name querying scenarios is function `InvokeByName`, which maps an object name string to a dispatch-ID. However, the script runtime does not have the knowledge about the dispatch ID table, so the name query is passed into the HTML engine (mshtml.dll), where another single-entrant chokepoint `GetDispatchID` performs the actual lookup.

Having the above knowledge, it is obvious how to implement our mechanism: (1) the accenting should happen at function `InvokeByName` using the key of the script; (2) the de-accenting should happen at function `GetDispatchID` using the key of the frame hosting the DOM. This reflects the principle of object ownership – every object is owned by the frame that hosts its DOM, regardless of how the object is referenced. In addition, because calling `GetDispatchID` is the only way to look up the dispatch ID table, our design reflects the principle of self defense – The de-accenting operation at function `GetDispatchID` is fully responsible for preventing any script from accessing the object, if the script runtime does not know the accented name of the object.

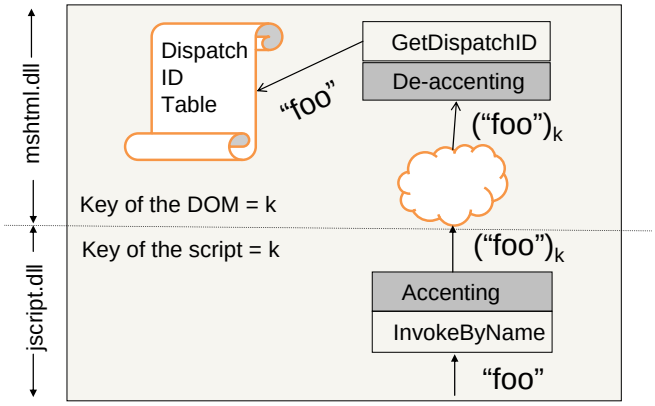


Figure 8: Accenting and De-Accenting of Name Queries

6. Evaluations

The script accenting mechanism can be implemented on the current version of IE (version 7) and the version shipped with Windows XP RTM (version 6). Currently, we choose IE version 6 as the platform to prototype the technique because most known cross-frame bugs have been patched individually in IE version 7. In this section, we evaluate the effectiveness of our defense against real attacks in the past. Because the script accenting is a generic technique, we believe that it will also be effective against cross-frame attacks discovered in the future. This section also presents the evaluation results about the transparency and the performance overhead of our mechanism.

6.1 Protection

We now revisit the attack scenarios discussed in Section 4 and demonstrate how the script accenting mechanism can defeat all these attacks. Also, these examples support our argument that the correct implementation of the accenting/de-accenting operations is significantly more robust than that of the current frame isolation mechanism. The latter is to enforce a global property about how the information is propagated in the system, but the former is about local properties of the information sender and the receiver.

Attack 1 Revisited. As shown in Figure 2, the attack is to exploit a path that causes Windows Explorer to send a piece of script supplied by the malicious frame to the victim frame. It is very hard for

IE developers to anticipate that Windows Explorer, which is a component outside IE, can be used to relay the javascript-URL between two IE frames.

The same attack was launched against our IE executable with the script accenting in place. When the script executed `open("file:javascript:doEvil","frame2")`, we observed that the function `InvokeNavigation` gets the URL argument `file:javascript:doEvil` (see Figure 7 for the call graph), which was not accented because the URL is not a javascript-URL. The URL is then passed to Windows Explorer, corresponding to the cloud of complicated navigation logic in Figure 7. Windows Explorer removed the "`file:`" prefix and handled it as a javascript-URL, so it passed the URL `javascript:doEvil` to `frame2`, which is the receiver frame. Before the compilation of the string `doEvil`, the accent key of `frame2` is used to de-accent the string. Because no accenting operation had been performed on `doEvil` in the sender frame, the de-accenting operation makes it illegible for the compilation, and thus the attack is thwarted.

Attack 2 Revisited. Attack 2 exploits the function aliasing to confuse `Frame1` about which frame really initiates the "`location.assign`" call (see Figure 3). Because of the function aliasing, the timer for delayed execution, and the navigation happening in the meanwhile, the execution path leading to the attack is highly convoluted.

When the attack was launched against our IE executable, steps (1) – (3) of the attack are unaffected by the script accenting mechanism. At step (4), despite the confusion caused by the aliasing of `location.assign`, our principle of script ownership is straightforward to conform to – the string `doEvil` was supplied by the script running in `Frame2`, so it was accented using the key of http://evil. This accented version of the string `doEvil` was then de-accented using the key of http://payroll at the receiver frame `Frame1`, which failed the compilation.

Attack 3 Revisited. In Attack 3, because of the confusing navigation statement, the cross-frame check is erroneously performed to examine if `frame2` can navigate `frame1` to a javascript-URL. This is a wrong check because `frame0`, not `frame2`, is the real initiator of the navigation.

When the attack was replayed on our IE executable, there was no confusion about the accenting policy. `Frame0` supplied the javascript-URL, so `Frame0`'s key, corresponding to http://evil, was used in the accenting operation. When this URL is received by `Frame1`, it was de-accented using the key of http://payroll, and thus the attack was not effective.

Attack 4 Revisited. Attack 4 exploits the semantics of user events. The script in `Frame0` can reference to the DOM objects in `frame1` through `event.srcElement`, and therefore does not need to pass the cross-frame check performed by the window proxy between `frame0` and `frame1`.

Our IE executable defeated this attack because of the accenting of object name queries. The script in `frame0` was able to reference to `event.srcElement`, which is an alias of an object in `frame1`. However, because of the mismatch between the DOM key and the script key (see Figure 8), the script cannot access to any attribute/method/sub-object of the object. Therefore, merely obtaining the cross-frame object reference is useless. This is similar to the situation in a C program where a pointer references to a memory location that is not readable, writable or executable, and any dereference of the pointer results in a memory fault.

Attack 5 and Attack 6 Revisited. Attack 5 is due to the script merging problem. It was defeated by our defense in the same way as we described above. Attack 6 exploits a race condition to cross the isolation boundary of the *XML* object. Although it is not a cross-frame attack, we believe that the object name accenting mechanism would be effective against it, if the mechanism was implemented for the *XML* object. In IE, the frame object is not the only object subject to the same-origin policy. The *XML* object and the *XMLHTTPRequest* object, for example, are also protected by the policy. They should not be

accessed by any script running in another domain. Because the domain IDs of these objects are not necessarily identical to the domain ID of the frame hosting them, our implementation needs to be extended to accent the names of their internal sub-objects using the keys of their own domains. We plan to implement the accenting mechanism for these non-frame objects as the immediate next step.

6.2 Transparency

Although our technique is to offer the protection for the browser, it is also important that the technique is fully transparent to existing web applications. It would be a significant deployment hurdle if the mechanism was not transparent to current browser features and cause existing web applications to work improperly.

An advantage of the script accenting mechanism is that it does not affect the internal logic of the browser components, but only the arguments passed at their interfaces. Because of the clear interfaces, it is easy to achieve the full transparency. Our IE executable has been tested on many web applications. Table 1 shows a number of representative examples. We intentionally selected the web applications with rich user interaction capabilities in order to test the transparency of the mechanism. We observed that all these applications run properly in our IE executable.

Table 1: Representative Web Applications

Name (site)	Description of the Web Application
Virtual Earth (map.live.com)	Microsoft's map service. The features include the road map, the satellite map, the bird eye view, and the driving direction planner. It supports rich user interaction capabilities, including zooming the map, drag-and-drop, and gadget moving, etc.
Google Map (map.google.com)	Google's map service. The features include the road map, the satellite map and the driving direction planner. It supports rich user interaction capabilities.
Citi Bank (citi.com)	An online banking application. The features include user authentication, electronic bank statement and other banking services.
Hotmail (hotmail.com)	A popular web-based email system.
CNN (cnn.com)	A popular news page. Because it contains many browser features, the IE development team uses the CNN page as an important test case.
Netflix (Netflix.com)	A popular movie-rental application. The page is user-specific.
YouOS (youos.com)	A web operating system. It provides the user a unix/linux-style operating system inside the browser. It supports very rich user interaction capabilities.
Outlook Web Access (https://mail.microsoft.com)	A web-based email system. It provides the user interface of Microsoft Outlook in the browser. The user interaction capabilities of Outlook Web Access are similar to those of Microsoft Outlook.
Slashdot (Slashdot.com)	A popular technology-related news website. It is similar to a blogging site.

In addition to the popular web applications, we conducted another test to verify that our mechanism is fully transparent to legitimate cross-frame communications: the attacks discussed earlier are interesting and convoluted scenarios to accomplish illegitimate cross-frame communications. In our transparency test, each attack scenario was converted into a legitimate cross-frame access scenario by loading all frames with pages from the domain http://payroll[6]. Therefore, each previous attack script became a script containing convoluted but legitimate cross-frame accesses. We observed that all these scripts ran successfully, and all cross-frame accesses happened as expected. This is a strong evidence that the script accenting mechanism does not affect the communications conformant to the same-origin policy, and thus is fully transparent to legitimate web applications.

6.3 Performance

As described previously, the accenting mechanism is performed in two situations: (1) *When a frame sends a script to another frame*. The performance overhead incurred by our code is negligible in this situation because it simply applies an XOR primitive on every 4-byte word in a string. This is

[6] For Attack 1, the URL prefix "file:javascript:" needs to be modified to "javascript:" to conform to the same-origin policy.

insignificant compared to the runtime overhead for the sending, receiving, compiling and merging of the script. (2) *When a script queries the name of a DOM object*. Name querying happens frequently during the execution of a script. We perform an accenting operation and a de-accenting operation for every query, which may incur noticeable performance overhead. Intuitively, the overhead should not be significant because every name query is made through a deep stack of function calls from jscript.dll to mshtml.dll, which is already a non-trivial operation. To quantitatively measure the upper bound of the performance overhead, we ran the following script to access `window.document.body.innerText` for 400,000 times. The observed performance overhead is 3.16%.

Table 2: Stress Test to Measure Performance Overhead of Name Querying

`for (i=0;i<400000;i++)`
`  x=window.document.body.innerText`
	`Original IE executable: 17.812 seconds`
	`Our IE executable: 18.093 seconds`
	`Performance overhead: 3.16%`

Note that this 3.16% performance overhead is the worst-case result, because the test is a stress test that does nothing but querying names. To estimate how the performance overhead affects users' browsing experience, we measured the page initialization time of popular websites. The initialization time includes the page downloading and the execution of the main script on the page. The measurement is made by subscribing a time recording function to the `BeforeNavigate` and the `NavigateComplete` events of the browser [13]. For each page, we measured 50 times. The result is shown in Figure 9, where we see the standard deviations much larger than the differences between the average numbers for the original IE executable and our IE executable. We believe that the differences are caused by network conditions, and the script accenting mechanism has almost no effect on user's browsing experience.

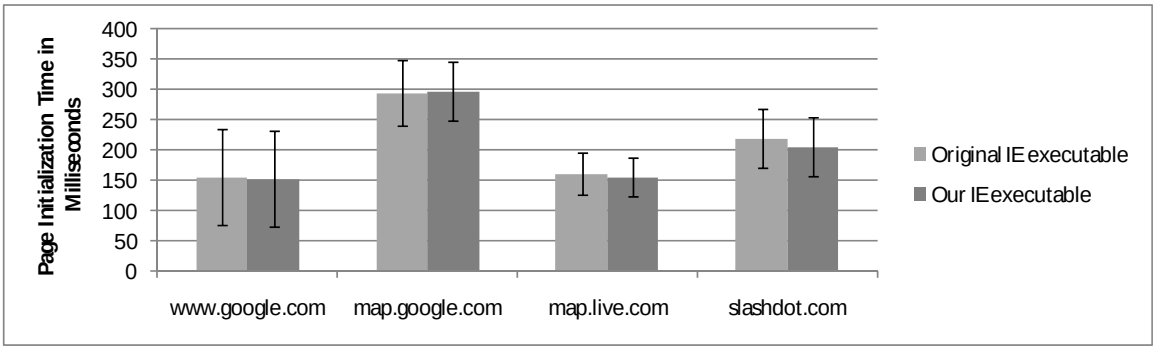


Figure 9: Page Initialization Times With and Without Script Accenting

7. Conclusions and Future Work

The browsers' isolation mechanisms are critical to users' safety and privacy on the web. Achieving proper isolations, however, is very difficult. At the policy level, research effort is still needed to refine the policies to guarantee isolations and permit legitimate browser functionalities. At the implementation level, historical data show that even for well-defined isolation policies, the current enforcement mechanisms can be surprisingly error-prone. As an example, the implementations of the cross-frame policy enforcement have been exploited on most major browser products.

We analyzed the implementation of IE's cross-frame policy enforcement and the cross-frame attacks reported to Microsoft. The analysis showed that the attack scenarios involve the navigation mechanism, the function aliasing, the excessive expressiveness of navigation methods, the semantics of user events and IE's interactions with other system components, which are very difficult to anticipate by the developers.

In this paper, we proposed the script accenting technique as a light-weight transparent defense against the cross-frame attacks. A prototype has been implemented on IE. The evaluation showed that all known cross-frame attacks were defeated because of the mismatch of the accents of the accessor frame and the

accessee frame. We also showed that the mechanism is fully transparent to existing web applications. Despite a 3.6% worse-case performance overhead, the end-to-end measurement about user's browsing experience did not show any noticeable slowdown.

The basic idea of the accenting is that the origin identities can be piggybacked on communications at the interfaces between different system components without affecting their internal logic. This idea is applicable to other isolation mechanisms. Our immediate next step is to implement the accenting technique for the *XML* object and the *XMLHTTPRequest* object, whose isolation policies are similar to the cross-frame policy that we discussed in the paper.

Acknowledgements:

The author thanks David Ross at the Microsoft Security Technology Unit for providing technical insights to the cross-frame vulnerabilities. Emre Kiciman, Ben Livshits, Madan Musuvathi, Helen Wang and Yi-Min Wang at Microsoft Research offered precious comments on this work.

References:

[1] Firefox Cross-Frame Vulnerabilities. Security Focus Vulnerability Database. Bug IDs: 10877, 11177, 12465, 12884, 13231, 20042. http://www.securityfocus.com/bid

[2] Opera Cross-Frame Vulnerabilities. Security Focus Vulnerability Database. Bug IDs: 3553, 4745, 6754, 8887, 10763. http://www.securityfocus.com/bid

[3] Netscape Navigator Cross-Frame Vulnerabilities. Security Focus Vulnerability Database. Bug IDs: 11177, 13231. http://www.securityfocus.com/bid

[4] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. "Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks," 10th ACM Conference on Computer and Communication Security (CCS), October 2003.

[5] A. Clover. Css visited pages disclosure, 2002. http://seclists.org/lists/bugtraq/2002/Feb /0271.html.

[6] Don Box. Essential COM. ISBN 0-201-63446-5. Addison Wesley.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. "PointGuard: Protecting pointers from buffer overflow vulnerabilities," the 12th USENIX Security Symposium. Washington, DC, August 2003.

[8] Douglas Crockford. "JSONRequest," http://www.json.org/JSONRequest.html

[9] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In ACM Conference on Computer and Communications Security, 2000

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In Proc. 1982 IEEE Symposium on Security and Privacy

[11] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. "Protecting Browser State from Web Privacy Attacks," the 15th ACM World Wide Web Conference, Edinburgh, Scotland, 2006.

[12] Martin Johns. "SessionSafe: Implementing XSS Immune Session Handling," 11th European Symposium on Research in Computer Security, Hamburg, Germany, September, 2006

[13] MSDN Online. http://msdn.microsoft.com

[14] The "Javascript:" Protocol. http://www.webreference.com/js/column35/protocol.html

[15] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. "Countering Code-Injection Attacks With Instruction-Set Randomization," 10th ACM International Conference on Computer and Communications Security (CCS). October 2003.

[16] Benjamin Livshits and Monica S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis," Usenix Security Symposium, Baltimore, Maryland, August 2005.

[17] Wei Xu, Sandeep Bhatkar and R. Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," 15th USENIX Security Symposium, Vancouver, BC, Canada, July 2006.

[18] The XMLHttpRequest Object. W3C Working Draft 27 September 2006. http://www.w3.org/TR/ XMLHttpRequest/

[19] *Cross-site scripting*. http://en.wikipedia.org/wiki/Cross_site_scripting