An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism

Shuo Chen
Microsoft Research
One Microsoft Way
Redmond, WA 98052
1-425-722-8238
shuochen@microsoft.com

David Ross
Security Technology Unit, Microsoft
One Microsoft Way
Redmond, WA 98052
1-425-705-2116
dross@microsoft.com

Yi-Min Wang
Microsoft Research
One Microsoft Way
Redmond, WA 98052
1-425-706-3467
ymwang@microsoft.com

ABSTRACT

Browsers' isolation mechanisms are critical to users' safety and privacy on the web. Achieving proper isolations, however, is very difficult. Historical data show that even for seemingly simple isolation policies, the current browser implementations are surprisingly error-prone. Isolation bugs have been exploited on most major browser products. This paper presents a focused study of browser isolation bugs and attacks. We found that because of the intrinsic complexity of browser components, it is impractical to exhaustively examine the browser implementation to eliminate these bugs. In this paper, we propose the script accenting *mechanism as a light-weight transparent defense to enhance the current domain isolation mechanism. The basic idea is to introduce domain-specific "accents" to scripts and HTML 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 attacks were defeated, and the proposed mechanism is fully transparent to existing web applications. The measurement about end-to-end browsing time did not show any noticeable slowdown. We also argue that accenting could be a primitive that is general enough for implementing other domain-isolation policies.*

Categories and Subject Descriptors

D.4.6 **[Operating Systems]**: Security and Protection – invasive software; C.2.0 **[Computer-Communication Networks]**: Security and Protection

General Terms: Security

Keywords

browser, domain isolation bug, accenting, same-origin policy

1. INTRODUCTION

Web browsers can render contents originated from different Internet domains. A major consideration of web security is the appropriate enforcement of the *same-origin principle*: although it has never been strictly defined, this principle can be loosely interpreted as "a script originated from one Internet domain should not be able to read, manipulate or infer the contents originated from another domain", which is essentially the *non-interference property* [9] in the web security context. Failures to enforce this principle result in severe security 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 malicious script can do almost anything that the victim user can do on the browser.

Same-origin-principle violations can be due to insufficient script-filtering of the web application on the server, or due to flaws in the browser domain-isolation mechanisms: 1) Script-filtering flaws are commonly referred to as cross-site scripting (or XSS) bugs [17]. By exploiting these bugs, malicious scripts from attacker websites can survive the filtering and later be executed in the same security context of the authentic web application. A wealth of work in the security literature addresses prevention and defense techniques against XSS attacks. We do not focus on XSS in this paper; 2) On the browser side, the same-origin-principle violations are due to the improper isolation of the contents from different domains, which is one of the biggest security problems faced by browser developers. Although at the policy-specification level, certain isolation policies still need to be standardized to support more browser functionalities while preserving security, we found that even for a well-specified policy, the implementation of the enforcement mechanism can be surprisingly hard and error-prone. For example, the most well-known isolation policy is the cross-frame same-origin policy, which states that scripts running inside a frame of *http://a.com* is not allowed to access objects inside a frame of *http://b.com*. Bugs in the enforcement mechanism of this seemingly simple policy have been discovered on major browsers, including Internet Explorer (IE), Firefox, Opera and Netscape Navigator [1][2][3].

Although browser vendors are aware of real-world attacks against browser isolation mechanisms, there is little work in the academic literature about this serious security problem. In order to better understand the problem space, we conducted a focused study of IE's domain-isolation bugs and real attacks discovered in the past. The study shows that browser's flaws in the isolation mechanism are due to many convoluted factors, including the navigation mechanism, function aliasing, excessive expressiveness of navigation methods, the semantics of user events and IE's interactions with other system components. The exploitations of these flaws, which we will explain in details, are highly heterogeneous, and thus it would be very challenging to exhaustively reason about every scenario that the isolation mechanism may encounter. Of course, the unsolved challenge suggests that the browser may have new bugs of this type discovered in the future, similar to the situation that we have with

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

buffer overrun bugs – individual bug patches would not solve the problem as a whole.

The prevalence of browser isolation bugs in all major browser products naturally calls for a comprehensive defense technique. However, two practical constraints need to be considered when we design the defense technique: 1) *the technique must be transparent to existing browser functionalities*. A large volume of web applications have been developed based on existing browser functionalities. It would be a significant deployment hurdle if a defense mechanism broke these applications; 2) *we have only limited understanding of the browser implementation*. Browser products have very large code bases. The comprehensiveness of the defense should only rely on the correct understanding of a small subset of the source code, and thus should be straightforward.

In this paper, we propose a light-weight transparent defense technique with the consideration of the above practical constraints. 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. To implement this, 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 interfere if they have different accent keys.

The concept of script accenting provides a higher assurance for the implementation of the browser isolation mechanism. 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 on IE. The evaluation showed that all known cross-frame attacks 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 3.16% worst-case performance overhead, but the measurement of the end-to-end browsing time did not show any noticeable slowdown.

The rest of the paper is organized as follows: Section 2 discusses related work. We briefly introduce the basics of IE's domain-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 experimental 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 [14]. 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 [11]. Because XSS attacks are due to the failures of script filtering, Xu et al proposed using taint tracking to detect the attacks [15]. The attacks discussed in this paper are a different type of attacks. They exploit flaws in the browser isolation mechanism, not the input-filtering bugs on the web applications.

Interesting research has also led to the proposals of new 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* [16] and *JSONRequest* [7] are the 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 specified the same-origin policies for browser caches and visited links [4][8][10]. In this paper, we do not discuss how to specify better policies, but focus on how to correctly and securely implement those that are well-specified.

The isolation bugs that we discuss are at the HTML and Javascript level, rather than the OS process level. One can imagine that if the IE process is compromised by buffer overrun or other binary code attacks, then the malicious binary code can directly access pages from different domains without exploiting any domain-isolation bugs at the HTML/Javascript level. These attacks can be thwarted by OS-level process isolations. For example, Tahoma is a web browsing system based on virtual machine monitor (VMM) [6]. It implements a browser operating system (BOS) to guarantee that each web application runs inside its own virtual machine. Therefore, even when a browser instance in a web application is compromised, it cannot interfere with other web applications. It should be noted that HTML/Javascript isolation and OS-level process isolation are very much orthogonal in today's browsers because no matter how the underlying processes are isolated, the HTML/Javascript semantics require the capabilities of cross-domain navigations, frame-hosting, event capturing, etc, which are the source of isolation bugs discussed in this paper.

3. THE BASIC MECHANISM FOR DOMAIN ISOLATION OF IE

This section gives a short introduction of IE's basic isolation mechanism – the *frame-based isolation*. In IE, each HTML document is hosted in a frame (or an inline frame)[1]. A browser window is the top-level frame, which hosts the top-level document that may contain other frames. IE implements a security mechanism to guarantee that scripts from one frame can access documents in another frame **if and only if** the two frames are from the same domain.

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

[1] The security aspect of inline frames is very similar to that of regular frames. In the rest of this paper, the term "frame" refers to regular frames and inline frames.

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.

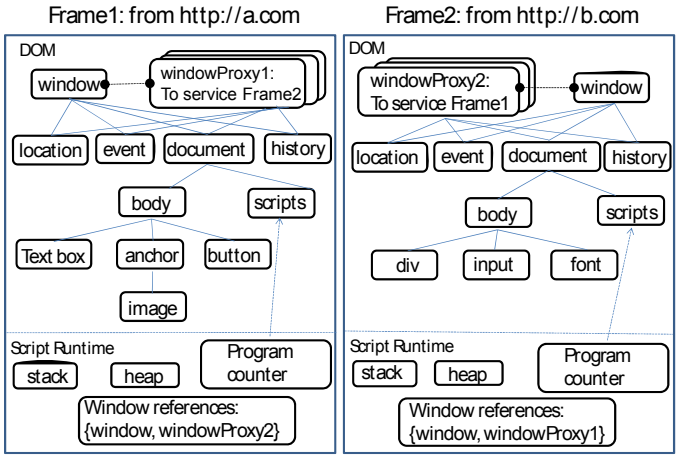

Figure 1: Cross-Frame References and the Isolation Between `Frame1` and `Frame2`

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 instruction "LoadMember baseObj,nameString" 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 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. Therefore it is 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. Conceptually the window proxy is a clone of 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[2] 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.

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

4. A Study of Real-World Attacks Against IE

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 a window proxy, (2) the window proxy has the correct domain-IDs of the accessor frame and the accessee frame. We studied the Microsoft Security Vulnerability database, and found that all discovered frame-isolation bugs are because of the invalidity of these assumptions. There are unexpected 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, function aliasing in the script runtime, excessive expressiveness of frame navigations, and the semantics of user events. In this section, four real attacks are discussed to show that it is very hard to for browser developers to exhaustively reason about all possible execution scenarios.

In these examples, we assume the user's critical information is stored on the website *http://payroll*, and the malicious website visited by the user is *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 damage. In the following discussion, the attacker's goal is to execute doEvil in the context of *http://payroll*.

4.1 Exploiting the Interactions between IE and Windows Explorer

IE and Windows Explorer[3] 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",

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

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

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 [13]. 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, since the javascript-URL is passed from the Windows Explorer, `Frame2` receives the script as if it was from the local machine, not from the Internet, which bypasses the same-origin check.

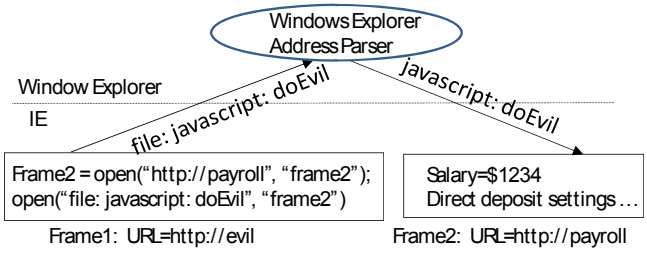

Figure 2: Illustration of Attack 1

4.2 Exploiting Function Aliasing

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, executing 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 expires, `location.assign('javascript:doEvil')` is executed in `Frame2`. Because of the aliasing, the statement really means "`frame1.location.assign('javascript:doEvil')`". Despite that 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. As a result, `doEvil` is merged into the scripts of the *http://payroll* DOM, and get executed.

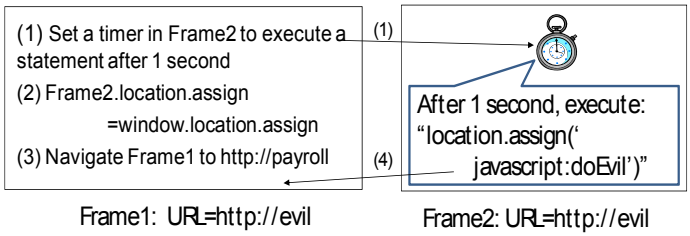

Figure 3: Illustration of Attack 2

4.3 Exploiting 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 above, `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 is the initiator of the navigation, `Frame0` or `Frame2`? In the unpatched versions of IE, `Frame2` is considered the initiator, because the `open` method being called is a member of `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.

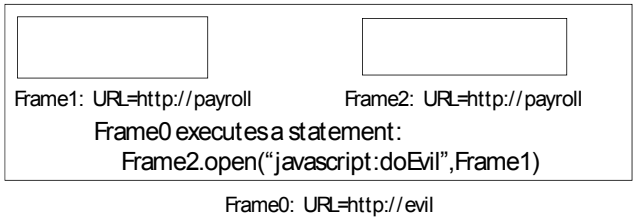

Figure 4: Illustration of Attack 3

4.4 Exploiting 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.

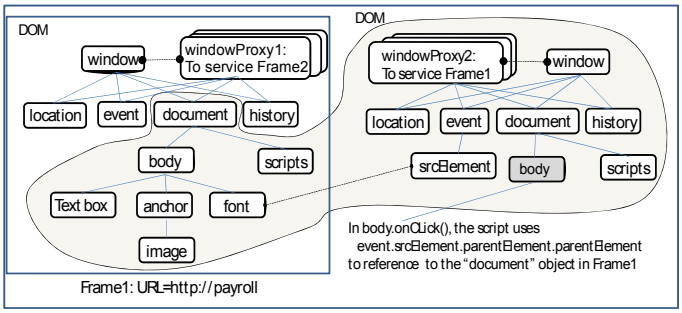

Figure 5: Illustration of Attack 4

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 script `doEvil` can be executed in `Frame1` by the assignment `F1Doc.scripts(0).text = doEvil`.

5. DESIGN/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 enumerate and test all these unexpected scenarios because too many code modules are involved, including the scripting engine, the HTML engine, the navigation mechanism, the event handling mechanism, and even non-browser components. 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 *script accenting* as a defense technique. The technique takes advantage of the fact that the browser executable has a clean interface between the component responsible for the DOM (e.g., the HTML engine *mshtml.dll* in IE) and the component responsible for the Javascript execution (e.g., the Javascript engine *jscript.dll* in IE). Because by definition the domain-isolation 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. Accenting is the operation to tightly bind the accent key with the script. This section describes our design and implementation of the accenting mechanism on IE.

5.1 The Primitive Operation of Accenting

A possible implementation of the accenting operation could be to attach the accent key with the script when the script is generated, and to propagate the accent key with the script in memory copies. This mechanism is often referred to as "tainting". Usually, tainting is implemented as a system-wide infrastructure at the hardware architecture level or in the virtual machine. We, however, want to implement the accenting mechanism completely in the browser, where it is not practical to track all memory operations because of the complexity of the source code. Furthermore, Attack 1 in Section 4.1 is an example to show that the script can even travel to a non-browser component, so tainting-based implementation is not possible for us.

XOR-based randomizations are frequently used in security defenses. Our current implementation also uses XOR as the primitive operation for accenting: we generate a 32-bit random number (four-bytes) as the accent key for each domain. The primitive operation to accent a text 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. This accenting operation has two clear advantages: (1) it guarantees that the accented script or any portion of the script is illegible to other domains, regardless of how the script travels; (2) the operation does not increase the length of the script, so the existing browser code can correctly handle the accented script without the possibility of buffer overflow. This is important for the transparency of our mechanism.

5.2 Accent Key Generation and Assignments

We keep a lookup table in the HTML engine (*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.

The browser provides support for a frame to change its domain during the page rendering and the execution of its scrip[4]. 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 support this feature, we redo the key generation/assignment operations when the `document.domain` attribute is changed. Note that in the domain isolation mechanism shown in Figure 1, *http://map.live.com* and *http://live.com* are two different domains once the domain changing operation is done, so each one has its own accent key. In other words, when a frame is from *http://live.com* or from *http://map.live.com* with its `document.domain` set to *http://live.com*, its accent key is the one correspondent to *http://live.com*; if a frame is from *http://map.live.com* without its `document.domain` being set, its accent key is the one correspondent to *http://map.live.com*.

5.3 Script Ownership and Object Ownership

Section 4 shows 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

[4] 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*.

guarantee that every script executes and travels in its accented form. In particular, we conform to two simple rules.

Rule 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 rule 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.

Rule of Object Ownership: The rule 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.

5.4 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(ScrSrc)`, `setTimeout(ScrSrc,…)` and `setInterval(ScrSrc,…)`, where `ScrSrc` is the text string of the script source code, or (2) navigating the frame to a Javascript-URL. The format of the Javascript URL is "`javascript:ScrSrc`", where `ScrSrc` 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.assign(…)`", "`location.replace(…)`", 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.

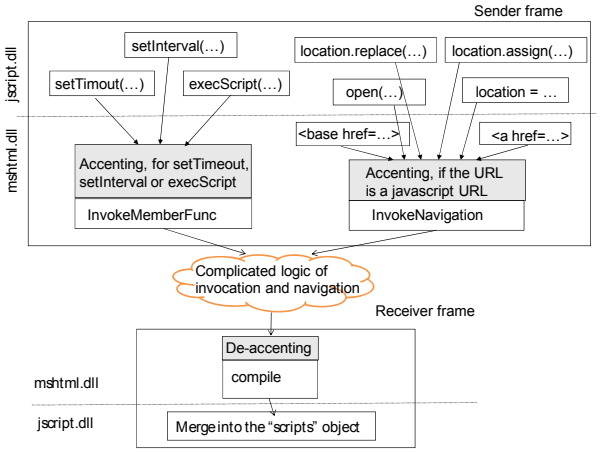

Figure 6: Accenting/De-Accenting Script Source Code

For each invocation or navigation scenario, we obtained a call path. These paths form a call graph shown in Figure 6. We observed that internally a common function called by `execScript`, `setTimeout` and `setInterval` is `InvokeMemberFunc`, and a common function called for all Javascript URL navigations is `InvokeNavigation`. Therefore, we insert the accenting operation before `InvokeMemberFunc` and `InvokeNavigation`. At these two functions, it is straightforward to conform to the rule 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 in the receiver frame is much simpler. 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 entry function of the compilation, which is an ideal location to perform the de-accenting operation, i.e., removing the accent from the script by applying an XOR using the accent key of the receiver.

As we discussed in Section 4, exploitable bugs have been discovered in the past in the complicated logic that implements cross-frame invocation and navigation, which we represent as a cloud. A significant advantage of our design is that we do not need to understand this complicated logic. The security of our mechanism only relies on the fact that any script needs to be compiled by the function `compile` before it is executed. Note that although we believe `InvokeMemberFunc` and `InvokeNavigation` are able to comprehensively perform accenting in all script-sending scenarios, the security does not rely on the comprehensiveness – hypothetically, if there was an unexpected scenario to send a script without being accented, it would fail the compilation in the receiver frame. In other words, the incomprehensiveness would not cause a security compromise, but a compatibility problem only cause a fail-stop, but not a security bug. Of course, fail-stop is also undesirable as it causes application incompatibilities. Section 6.3 will show that we have not found any incompatibility when we tested our mechanism against real applications.

5.5 Accenting the Object Name Queries to Defeat DOM Merging Attacks

Real-world 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 can be referenced either by "`Frame1.document.body.children(3)`" or by "`window.event.srcElement`". From the perspective of the script runtime, the execution paths of these two references are unrelated.

To obtain the call graph of name querying, we studied various name querying scenarios, including the queries of objects as well as the aliases of them. Because IE uses the COM programming model [5], 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, the interface function for name querying is `InvokeByName`, which is responsible for mapping 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 the function `GetDispatchID` performs the actual lookup.

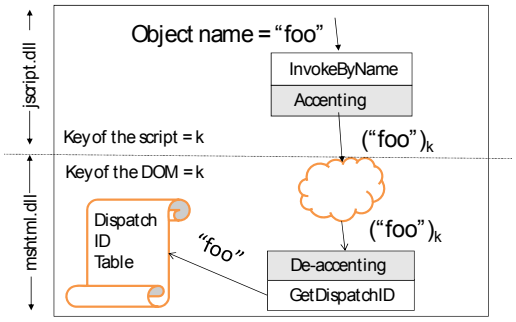

Figure 7: Accenting/De-Accenting of Name Queries

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 frame of the script; (2) the de-accenting should happen at function `GetDispatchID` using the key of the frame hosting the DOM (Figure 7). This reflects the rule of object ownership – every object is owned by the frame that hosts its DOM, regardless of how the object is referenced. In this design, the security only relies on the fact that every object-reference needs to call `GetDispatchID` to obtain the dispatch-ID, which we believe is a simple assertion to make based on the browser implementation. We do not need any assumption about the code paths of object-name querying, which are difficult to exhaustively enumerate.

Note that in IE's implementation, the object names in DOM trees and scripts are represented using wide-characters (two bytes per character). In a rare situation when the object name of the cross-DOM reference has only a wide-character, the strength of the XOR operation is weaker than usual, because the probability of a successful random guess is $1/(256^2)$, not $1/(256^4)$. A straightforward solution is to pad a wide-character, e.g., "0x0001", to the original one-character object name before applying the accenting operation. After the de-accenting, the padding character "0x0001" should be removed from the name string.

5.6 Other Ways of Implementing the Script Accenting Mechanism

The basic idea of accenting is to introduce domain-specific versions of scripts and HTML object names. As a concrete implementation, we use the XOR operation to bind a domain-specific secret with the string being accented. This is by no means the only way to implement the accenting mechanism. Our XOR operation is conceptually equivalent to tagging a domain ID to the string. A possible alternative implementation is to use a hash value of the domain name as the accent key K, and a string S can be accented as "K#S". When it is deaccented, K is compared with the accent key of the destination frame, and removed if identical. Nevertheless, this scheme might have some concerns: (1) the string S still travels inside the browser in its plain text form. If the attack has a way to reference to it, it can be precisely corrupted; (2) this scheme requires extra bytes for the accented string. Reallocating larger buffers is not always easy when we work on legacy code. It may cause compatibility problems, and requires source code understanding in order to free these buffers correctly.

However, except these potential concerns, we believe that "accenting" is a high-level idea which may have several valid implementations.

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), because their isolation mechanisms have no significant difference. 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 this type of attacks discovered in the future. This section also presents the evaluation results about the transparency and the performance overhead of our defense 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-based isolation mechanism. While the latter attempts to enforce a global property about how information is propagated in the system, the former focuses on the more tractable task of enforcing local properties at a few instrumentation locations.

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 6 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 6. 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 initiated the "`location.assign`" call (see Figure 3). Because of 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 rule 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`, and failed to be compiled.

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 7), 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.

6.2 Impossibility of XOR Probing Attacks

Because our current implementation uses XOR ($\oplus$) as the primitive operation for accenting, the security relies on the invisibility of the accent keys to the attacker's script. Hypothetically, if the attacker's script had the knowledge about $k_{atk} \oplus k_{vtm}$, where k_{atk} is the accent key of the attacker frame and k_{vtm} is the accent key of the victim frame, then the attacker can send the script "doEvil $\oplus$ ($k_{atk} \oplus k_{vtm}$)" to the victim frame, which will be accented and then deaccented to "doEvil". Therefore, a caveat of XOR-based security approach is that the attacker might have smart methods to efficiently guess the value of ($k_{atk} \oplus k_{vtm}$).

Remember that the accent keys are four-byte words. The attacker could guess the first two bytes of ($k_{atk} \oplus k_{vtm}$) and send the script ("//xx" $\oplus$ ($k_{atk} \oplus k_{vtm}$)) to the victim frame. If the guess is correct, the script will be compiled correctly because "//" denote a comment line in javascript. If the guess is incorrect, a syntax error will be generated by the victim frame. If the attacker can catch the syntax errors, he/she can successfully guess the first two bytes in 65536 attempts. Then he/she can probe the third byte by using ";//x" in 256 attempts, and the fourth byte by using ";;//" in another 256 attempts.

Although the above probing attack seems plausible at the first glance, it is not effective for two reasons. First, we observe that scripts in IE are always represented using wide-characters, which means the string "//" is already four-byte long. It requires 256^4 attempts to guess. More fundamentally, even for a browser not using the wide-character representation, the attack still lacks an important prerequisite – there is no way for the attacker frame to detect a syntax error in the victim frame, because the two frames are in different domains. In other words, for the probing attack to succeed, the attacker frame already needs the capability to communicate with the victim frame (e.g., through the `onerror` method of the victim frame), but such a prerequisite is exactly the domain-isolation violation that the attacker tries to achieve. This is a cyclical cause-and-effect situation. Therefore, the XOR-probing is not a real concern of the accenting mechanism.

Another issue related to XOR probing is the potential weakness in accenting an object name that has only one wide-character. We have discussed this in the last paragraph in Section 5.5: we need to pad another wide-character so that the object name is four-byte long.

6.3 Application Compatibility

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 is not transparent to current browser features and causes web applications to malfunction.

Table 1: Representative Web Applications

App	Description of the Web Application
Virtual Earth	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 interactions, including zooming the map, drag-and-drop, and gadget moving, etc.
Google Map	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	An online banking application. The features include user authentication, electronic bank statement and other banking services.
Hotmail	A popular web-based email system.
CNN	A popular news page which contains many browser features,
Netflix	A popular movie-rental application. The page is user-specific.
YouOS	A web operating system. It provides the user a unix/linux-style operating system inside the browser. It supports rich user interactions.
Outlook Web Access	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	A popular technology-related news website. It is similar to a blogging site.

As stated earlier, if the accenting was not performed comprehensively in all legitimate execution paths, normal browser functionalities would be broken because scripts could not be correctly deaccented and compiled. To verify the transparency of our implementation, our modified 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.

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*. 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 good evidence that the script accenting mechanism does not affect communications conforming to the same-origin policy.

6.4 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 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 measure the upper bound of the performance overhead, we queried `window.document.body.innerText` for 400,000 times. The execution times for the original IE and our modified IE are 17.812 seconds and 18.374 seconds, respectively. The observed performance overhead is 3.16%.

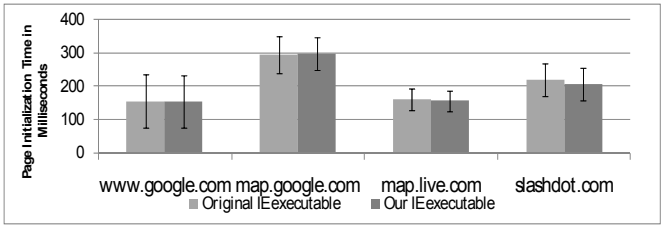

Figure 8: Page Initialization Times With and Without Script Accenting

Note that this 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 the end-to-end browsing time, 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 [12]. For each page, we measured 50 times. The result is shown in Figure 8, 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.

7. CONCLUSIONS

Browsers' isolation mechanisms are critical to users' safety and privacy on the web. Achieving proper isolations, however, has proven to be difficult. Historical data show that even for well-defined isolation policies, the current enforcement mechanisms can be surprisingly error-prone. Browser isolation bugs have been exploited on most major browser products. To the best of our knowledge, this is the first focused academic study of real-world browser isolation bugs.

We analyzed the implementation of IE's domain-isolation mechanism and the previously reported attacks. The analysis showed that the attack scenarios involve complicated HTML/script behaviors in 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 these attacks. A prototype has been implemented on IE. The evaluation showed that all known 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 3.16% worse-case performance overhead, the measurement of end-to-end browsing time 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 can be a general idea to apply in other isolation mechanisms. For example, even within a frame, the browser needs to enforce domain isolation for *XML* objects and *XMLHTTPRequest* objects, whose domains may be different from the domain of the frame. In addition to its current implementation on IE, we think the same idea can be applied on other browsers if they have well-defined interfaces between their HTML engines and script engines. More broadly, non-browser platforms need to enforce domain isolation as well. For example, CLR is the runtime environment for the .NET framework [18] and *Application Domain* is a security infrastructure in CLR. We speculate that the idea of accenting might also be applicable to platforms like CLR.

ACKNOWLEDGEMENTS

We thank our colleagues Emre Kiciman and Helen J. Wang for valuable discussions and suggestions. Anonymous reviewers provided insightful feedbacks to help improve the quality of the paper. We also thank our shepherd Dan Boneh for instructing us towards the final version. Jose Meseguer helped us better understand the concept of non-interference.

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

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

[6] Richard S. Cox, Jacob G. Hansen, Steven D. Gribble and Henry M. Levy: "A Safety-Oriented Platform for Web Applications," IEEE Symposium on Security and Privacy, 2006

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

[8] E. W. Felten and M. A. Schneider, “Timing attacks on web privacy,” in Proc.ACM Conference on Computer and Communications Security, 2000

[9] J. A. Goguen and J. Meseguer, “Security policies and security models,” in Proc. 1982 IEEE Symposium on Security and Privacy

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

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

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

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

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

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

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

[17] *Cross-site scripting*. http://en.wikipedia.org/wiki/Cross _site_scripting

[18] Common Language Runtime (CLR). MSDN Online. http://msdn2.microsoft.com/en-us/netframework/aa497 266.aspx