Residue Objects: A Challenge to Web Browser Security

Shuo Chen
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
shuochen@microsoft.com

Hong Chen*
Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
chen131@cs.purdue.edu

Manuel Caballero
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
v-mcab@microsoft.com

Abstract

A complex software system typically has a large number of objects in the memory, holding references to each other to implement an object model. Deciding when the objects should be alive/active is non-trivial, but the decisions can be security-critical. This is especially true for web browsers: if certain browser objects do not disappear when the new page is switched in, basic security properties can be compromised, such as visual integrity, document integrity and memory safety. We refer to these browser objects as *residue objects*. Serious security vulnerabilities due to residue objects have been sporadically discovered in leading browser products in the past, such as IE, Firefox and Safari. However, this class of vulnerabilities has not been studied in the research literature. Our work is motivated by two questions: (1) what are the challenges imposed by residue objects on the browser's logic correctness; (2) how prevalent can these vulnerabilities be in today's commodity browsers. As an example, we analyze the mechanisms for guarding residue objects in Internet Explorer (IE), and use an enumerative approach to expose and understand new vulnerabilities. Although only the native HTML engine is studied so far, we have already discovered five new vulnerabilities and reported them to IE developers (one of the vulnerabilities has been patched in a Microsoft security update). These vulnerabilities demonstrate a diversity of logic errors in the browser code. Moreover, our study empirically suggests that the actual prevalence of this type of vulnerabilities can be higher than what is perceived today. We also discuss how the browser industry should respond to this class of security problems.

Categories and Subject Descriptors D.4.6 **[Operating Systems]**: Security and Protection – invasive software; access controls

General Term Security

Keywords browser security; residue object; component object model (COM);

* Hong Chen worked on this project as a Microsoft Research intern.

EuroSys'10, April 13-16, 2010, Paris, France.

1. Introduction

Managing lifetimes for objects and deciding when they should be active are non-trivial in real-world software systems because there are many objects hosting and referencing each other at runtime. The mechanisms for managing objects can be security critical in some systems. This is especially true for web browsers. Web pages move in and out of a browser through navigations. After a web page is navigated away, the correct navigation semantics should simply be that "the old page is gone. Period." If certain objects in the old page do not disappear when the new page is switched in, the objects can become security hazards. Security properties at different levels can be compromised, such as visual integrity, document integrity and memory safety, as will be explained in Section 2.

The focus of this paper is on residue objects, i.e., *the objects in the old page that still reside in the memory after the navigation*. Residue objects are functionally useless. They are not intended to be used in any legitimate circumstance. Of course, a natural question is why the browser cannot just free them from the memory to eliminate these security hazards. It is because of a fundamental dilemma between the object-referencing ability of scripts, the garbage collection and the navigation mechanisms of modern browsers: i) using scripts, an object X in a page can be referenced by a data structure Y that is not within the page (of course, after the navigation of the page, Y is still a valid object because it does not belong to the navigated page); ii) on the other hand, because Y references to X, the garbage-collector cannot destroy X from the memory, otherwise Y would have a dangling reference and the memory safety is broken. Thus, object X has to reside in the memory and become a residue object. It is important to note that having residue objects in the memory is not due to any logic bug, but an expected situation that all browsers are facing, because the HTML specification allows an object to be referenced from outside its hosting page.

Therefore, every browser code has built-in guarding mechanisms to ensure that once an object becomes residual, it cannot be invoked or displayed. These mechanisms may seem easy to build because the policy appears to be simple and clear. However, we studied the SecurityFocus vulnerability repository [16] and the existing literature (see Section 3), and found attacks exploiting residue objects being reported sporadically

against leading browser products, including IE, Firefox and Safari. Despite the individual bug reports, residue objects have not been studied as a whole problem, and thus very little collective understanding is available. This paper is mainly motivated by two questions: (1) why the seemingly easy task of ensuring objects' inactivity turns out to be challenging in real browser implementations? (2) This class of vulnerabilities has not yet been prevalent in public vulnerability repositories, but is it because they are actually rare, or because this problem space has not been thoroughly examined by the security community?

We aim at answering the questions by finding concrete vulnerability instances in real browser code. As an example, we identify IE's basic mechanisms for guarding residue objects, then use an enumerative approach to generate different residue objects in order to expose new vulnerabilities in IE. We have found five new vulnerabilities and reported them to the IE team (one of them was patched in Microsoft February 2009 security update). The vulnerability instances show a diversity of logic errors, e.g., a residue window fails to be marked "dead"; a "dead" window remains visible or allows script execution in it; a reference is released prematurely; and an object is partially destructed but remains exposed to scripts. We will explain the subtle logic errors that browser developers should be aware of in guarding residue objects. Also, the discovered security bugs are an empirical evidence to show that the actual prevalence of these vulnerabilities may be considerably higher than what is perceived today.

We acknowledge that our analysis methodology in its current form is rather rudimentary. Although it is able to automatically track object states in the memory, it requires analysts' insights to specify security invariants, which are diverse as suggested by the vulnerabilities that we discovered so far. Rather than the analysis approach, the main value of this paper is to give an in-depth understanding of the browser residue object problem itself. We believe that the in-depth understanding can benefit the browser industry's on-going security efforts. In the past two years, browser vendors are undertaking rapid architectural enhancements, such as those on IE, Chrome, OP [6] and Gazelle [7]. We will discuss whether these efforts can help address the residue object problem fundamentally.

The rest sections are organized as follows: we explain security consequences of residue objects in Section 2, and study historical bugs in Section 3. Section 4 presents background knowledge for the later sections. IE's residue-object guarding mechanisms are described in Section 5. Our analysis methodology is detailed in Section 6. We present our findings in Section 7. Section 8 and 9 are discussions and related work. We conclude in Section 10.

2. Security Damages Caused by Residue Objects

A simple way to think about browsing is that a web page is a family of objects, and a browser is the house. During a navigation, the old page moves out, and the new one moves in. However, if residue objects can interfere with the browsing experience of the current page, some basic security properties become difficult to maintain. We list some possible damages, and will show vulnerability instances in details in Section 7.

Visual spoofing: The windowing mechanism of the browser is designed to ensure that objects are in appropriate visual contexts. Objects displayed in a browser tab are supposed to be the ones defined by the top-level document of the tab, or in descendent documents explicitly loaded by the top-level document. The origin of the top-level document is asserted by the address bar of the tab. Also, dialog boxes are displayed together with their opening tabs. If some out-of-context objects, neither defined in the top-level document nor in any descendant document, visually appear in the tab, the security goal of the windowing mechanism is defeated. Speaking about residue objects in particular, if an HTML object or a dialog box in the old page is present in the visual context of the new page, the user has no way to understand how to trust the page even when the address bar is examined. Figure 1 shows such a scenario: the logo of the IEEE, which is an object in the front tab before the tab is navigated to *http://www.acm.org*, remains in the tab. The logo is visually blended into the new page.

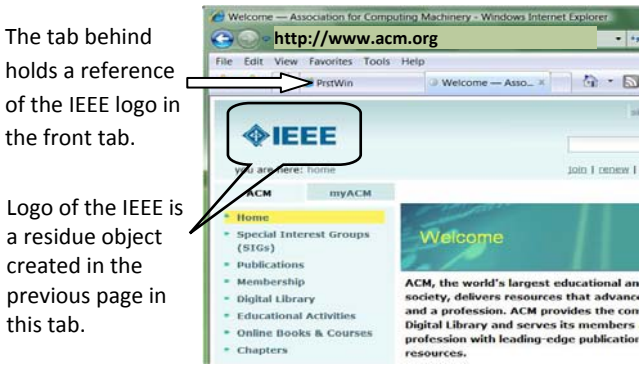


Figure 1: IEEE logo resides on the ACM homepage

Involuntary navigation: a page visited in the past should no longer be able to navigate the browser. We refer to the violation of this common expectation as *involuntary navigation*. Involuntary navigation is dangerous – essentially, clicking a hyperlink in an honest page can navigate the browser to a random website. Figure 2 shows an attacker scenario: if the user ever visits the attacker's page, there will be a residue inline-frame (a.k.a. iframe) hidden in the browser. The iframe can survive all navigations, and thus the script in it is always running. The browser is then navigated to the real PayPal login page.

After 8 seconds, the script navigates the browser to a fake re-login page to phish the user's password. Therefore, in this involuntary navigation scenario, what the user perceives is exactly as if the login page voluntarily navigated to the re-login page. Essentially, once the browser has ever visited an untrusted page, future navigations can be controlled by the script residing in the browser even after the untrusted page is navigated away.

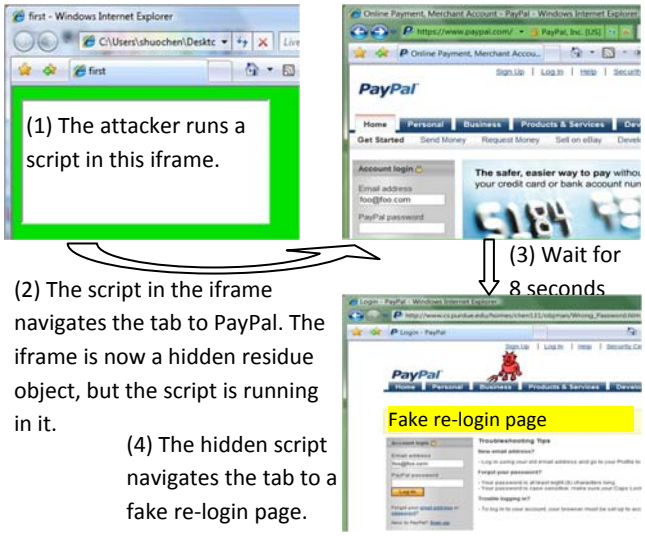


Figure 2: Involuntarily navigate to a fake login page

Cross-domain access: the basic access control policy on web documents is known as the same-origin policy [15], which prevents scripts from *a.com* from accessing a page from *b.com*. However, it is possible that a reference to an object in the old page from *a.com* is still valid after the new page from *b.com* is switched in – a scenario similar to the old family still having certain types of accesses to an object inside the house after the new family moves in. The existence of the residue objects makes the same-origin-policy difficult to implement securely.

Memory corruption: Being a classic vulnerability category, memory corruption is of course a serious security concern. Once the memory safety is violated, the potential damage is difficult to estimate because the control flow at the binary executable level can be diverted, and sometimes security-critical data can be overwritten. Residue objects impose additional complexity to maintaining memory safety because when a page is navigated away, internal objects may be partially destroyed, and thus become inconsistent, which result in dangling references, etc. If malicious code can access residue objects, it is likely that the dangling references are exposed. We will show two memory corruption bugs found by us, one of which is able to corrupt the EIP register.

3. Historical Bugs in Major Browsers

Instances of vulnerabilities related to residue objects have been documented individually in public sources, such as SecurityFocus bug repository [16] and academic papers [2][3], although they were not categorized in a single class in the existing literature. These bugs exist in various browsers – IE, Firefox and Safari. Though the quantity of these bugs documented in the literature does not seem big yet, it is because the overall problem has not been systematically studied. We will show later that when a browser is under a focused examination, many new bugs are uncovered.

Table 1 lists some documented bugs. The right-most column describes the consequences. Note that "a script persisting across navigations" can result in visual spoofing (if the script creates visible objects) and involuntary navigation (if the script navigates the browser), as we described earlier.

Table 1: Historical Bugs in IE, Firefox and Safari
(Further details in the appendix)

Source	Date	Browser	Consequence
SecurityFocus: ID 5196	2002-07	IE	Cross-domain access
SecurityFocus: ID 5841	2002-10	IE	Cross-domain access
SecurityFocus: ID 6028	2002-10	IE	Cross-domain access
SecurityFocus: ID 13230	2005-04	Firefox	A script can persist across navigations.
SecurityFocus: ID 17671	2006-04	Firefox	Memory corruption
SecurityFocus: ID 24286	2007-06	Firefox	A script can persist across navigations.
Section 4.2 of paper [3]	2007	IE	Cross-domain access
Section 4.2 of paper [2]	2009	Safari	Cross-domain access

All the published exploits share a common four-step pattern: step 1 – a web page is loaded into the browser; step 2 – an object inside the page is held by a reference; step 3 – the page is navigated away or unloaded, making the object a residue object; step 4 – the reference is used to invoke the residue object. For example, Table 2 shows one of the exploits. It first loads *empty.html* from the same website into the browser, and holds the inner frame by a global variable `ref`, then navigates the frame to a different website. Unexpectedly, the DOM of the new page can be accessed through `ref` – a cross domain attack.

Table 2: Exploit of Vulnerability 5196 in SecurityFocus

```
<object id="data" data="empty.html" type="text/html">
</object>                                   //step1: load a page
var  obj = document.getElementById("data")
var ref = obj.object;                       //step2: hold an inner object
ref.location.href = "http://targetSite.com";    //step3: navigate
setTimeout("read(ref.cookie)",500);   //step4: invoke ref. Attack!
```

Although all exploits follow the four-step pattern of "creating, holding, navigating and invoking", the exact cause of each bug is not clear unless we examine the internals of the browsers. In the next section, we will present some basic knowledge about garbage collection and object referencing. The knowledge is a basis for understanding the object management in browser code, and the challenges to ensure its effectiveness.

4. Background Knowledge of Garbage Collection and Browser Objects

This section covers some very basic concepts about garbage-collection, reference counting, COM object and ActiveX object. Readers familiar with these concepts can read through the section quickly.

4.1. Garbage collection algorithms

A real-world software system, such as a browser, consists of a large number objects, cross-referencing each other to form the data structures that are necessary for complex functionalities. Object X no longer needing to access object Y does not implies that object X can destruct object Y from the memory, because object Z may access it later. Coordinating object destructions among different code modules in an ad-hoc manner is error-prone and not scalable in a complex system. This is the reason for garbage collection algorithms.

In systems built in C++, a commonly used algorithm is reference-counting, a.k.a. refcounting. Each object keeps a refcount to indicate how many of its references have been passed out to other data structures, a.k.a. reference holders. When a data structure obtains a reference of the object, it increases the refcount of the object by 1. This operation is known as "AddRef". When the reference is no longer needed, the holder calls the "Release" operation to decrease the refcount by 1, indicating that this holder will not use the reference in the future. When the refcount becomes 0, the object is automatically freed from the memory. Refcounting is in fact a contract between holders and objects. The holders never directly free the objects, but only release the references.

Languages such as Java and C# treat objects and references as first-class citizens. The language runtimes usually have built-in garbage collectors, whose algorithms are based on object reachability – starting from persistent data structures, the object reference graph is traversed to decide which objects are reachable. All unreachable objects are freed. The main advantage of reachability-based collection is the elimination of memory leaks due to circular references.

4.2. Objects in IE and Firefox

Microsoft's Component Object Model (a.k.a. COM) is a programming framework that support runtime loading, hosting and componentizing code modules [1]. Internet Explorer (IE) uses COM to build browser's internal components and objects. Cross-Platform-COM, a.k.a. XPCOM, is a framework developed by Mozilla [10]. Firefox uses XPCOM. COM and XPCOM are very similar. At the most basic level, every object defines methods: `AddRef` and `Release` for refcounting.

An *ActiveX object* is a COM object that contains a number of properties, sub-objects and methods, each known as a "dispatch", identified by a "dispatch ID". Every ActiveX object has the method `Invoke` to apply an operation on the object according to a dispatch ID.

The ActiveX technology is generic enough to build objects of different scales. Not only HTML elements and Javascript variables, but also the entire HTML engine and the Javascript engine, are built as ActiveX objects. Moreover, both the HTML engine and Javascript engine themselves can host ActiveX objects in their address spaces. The mechanism in HTML engine is to use the <object> tag. For Javascript engine, it is through "`new ActiveXObject(ID)`". Because the HTML engine itself is an ActiveX object, it can be hosted inside a Javascript engine or another HTML engine.

Similarly, Mozilla's technology for building scriptable XPCOM objects is XPConnect [11]. As an object hosting mechanism, XPCOM has some similarities with COM: an XPCOM object can be hosted in the Javascript engine by calling `Components.classes(ID)`; Mozilla's HTML engine also supports <object> to import XPCOM objects.

5. Understanding the Mechanisms for Guarding Residue Objects

We stated earlier that residue objects are due to a fundamental dilemma between the scripting capability, the garbage collection and the navigation, and thus every browser needs to build mechanisms for guarding residue objects. Historically, making these mechanisms secure has been challenging, as shown by the reported vulnerabilities in IE, Firefox and Safari in various sources.

We choose IE as a concrete browser implementation to understand the challenges of guarding residue objects. Through the study of the source code, we get basic understanding about the guarding mechanisms at the C++/COM level. The object management in the browser is based on the object-capability model [12]: it tries to ensure that a component can access an object only if it has a reference to the object. The key mechanisms are below.

5.1. CWindow and CDocument objects

<*Window*> and <*document*> are standard HTML elements. The relationship between <document> and <window> is easy to understand – a document object is hosted in a window object. Upon a navigation, the old document is gone, but the window persists. At the C++ level, they are represented by the COM classes *CWindow* and *CDocument* (The initial character "C" indicates a class name). Each HTML file being rendered by the browser is instantiated as a CDocument object, representing an HTML document. Each CDocument object is hosted in a CWindow object. A tab, a frame, an iframe or a dialog box is a CWindow object.

The protections for <window> and <document> elements are critical in the browser logic. The access control on HTML DOM (document object model) structures is enforced in per-document-granularity: the

browser needs to block any access to a document from a different domain or any access to a document that has been navigated away. Being the fence surrounding the document, the window object needs to ensure many critical security properties. The window has its unique security challenges: i) it serves as a persistent object in the navigation; ii) it is directly exposed to arbitrary scripts because even pages from a different domain can legitimately hold a reference of the window; iii) display management and mouse-event handling rely on the security of windowing.

5.2. CWinProxy – the proxy class of CWindow

In an object-capability system, "proxy objects" are used to impose security checks. This idea is analogous to a gift card as a "proxy object" of cash. Gift card and cash have the same functionality – making payments, but the card issuer can impose access policies or revoke it, while cash has an unconditional purchasing power.

In browser implementations, cross-window security checks are necessary because the HTML specification allows the <window> element to be referenced from a different domain.[1] In such a scenario, if the script in the different domain could hold the CWindow object of the <window> element, the script would be able to unconditionally invoke all functionalities of the CWindow, with no respect to the same-origin-policy. To address this problem, CWinProxy is defined as the proxy object of CWindow so that the reference from the script to a CWindow object is indirect. CWinProxy implements the same interfaces as CWindow. However, when a method of a CWinProxy object is invoked, it always performs the same-origin-check, then invokes the actual CWindow object to fulfill the actual functionality.

IE follows some general guidelines about CWindow and CWinProxy: a reference of a CWindow object should only be passed to an object inside its own window. When an object outside the window requests a reference of the window, IE passes a CWinProxy object out.

5.3. Validities of CWindow, CWinProxy and CDocument

Every CWindow/CWinProxy/CDocument object is associated with a Boolean flag to indicate whether the object is valid for access. When a CWindow/CWinProxy /CDocument object is initially created, it is marked as valid. During the complex procedure of navigation, some objects can be marked as invalid, based on a fairly convoluted logic that may be error-prone. Any attempt to invoke an invalid object should get an `access_denied` error, and the operation is aborted.

[1] A script of *a.com* is allowed to execute *ref=window.open ("http://b.com")*. In this case, *ref* is a reference to a <window> element in a different domain. However, accessing *ref.document* is disallowed.

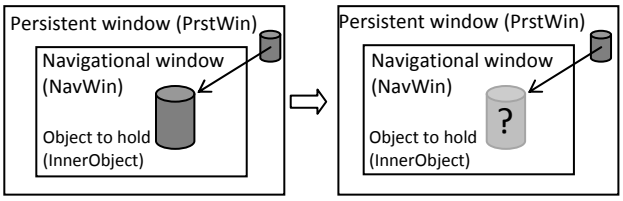
Controlling and checking object validities are the key mechanism to prevent residue objects from being invoked. The general guideline is that i) objects should be invalidated when their hosting pages have been navigated away or unloaded; ii) the validity of an object should be checked when the object is accessed. At the first glance, this guideline seems straightforward to follow. However, in the next two sections, we will see in the reality why it is complicated to guard residue objects.

6. Our Enumerative Approach for Studying Residue Objects

In this section, we describe our approach of examining the logic correctness of the guarding mechanisms, which is to generate different residue objects, and observe the browser's memory states. Specifically, it consists of two steps: (1) at the HTML/Javascript level, we use a tactic to enumerate different scenarios for object-creation, cross-referencing and navigation; (2) at the C++ level, we augment the browser to log critical function calls. Then, an analysis tool is built to derive the reference relationships and the object validity states from the logs. This level of insight enables us to see the internal states of the browser, and often gives new leads to the exposures of real vulnerabilities.

6.1. Tactic for Generating Residue Objects

The vulnerability reports in Section 3 suggest the steps for generating residue objects: we need a "navigational window" (NavWin), which contains an object InnerObject. We need a persistent window (PrstWin) from which InnerObject is referenced. Both windows are in the same domain to allow the cross-window reference. NavWin is then navigated to another page, InnerObject will become a residue object. This tactic is shown in Figure 3.



Make a reference from PrstWin to the object, and navigate NavWin

Figure 3: The tactic for generating residue objects

The nesting relation of PrstWin and NavWin is not essential, but for the convenience of our test case enumeration. In actual exploits, the relation between them can be flexible – PrstWin can also be a sibling of NavWin, or the two windows can be in different tabs. Of course PrstWin cannot be a child of NavWin, otherwise it will not be persistent after NavWin is navigated away. This tactic is shown as an HTML/Javascript template in Table 3.

The template defines Javascript variables `NavWin` and `ref` as global variables. The template contains three code snippets, denoted as `snippets #1–#3`, corresponding to steps 1-3 discussed in Section 3: NavWin-Creation, Object-Referencing, and NavWin-Navigation. We studied the HTML functionalities supported by the browser, and found several different mechanisms to realize each step. By combining these mechanisms, we produce the set of test cases.

Table 3: HTML/Javascript Template for PrstWin.html

```
<script>  var NavWin, ref;
</script>

*** snippet #1: create the navigational window
(NavWin) to host a page named "page1.html".
Assign this window to the jscript variable
"NavWin".

<script>
function GetInnerObject() {
  *** snippet #2: return an object inside navWin
                  as InnerObject
}
function DoNavigation() {
  *** snippet #3: specify the navigation
                  behavior for navWin
}

setTimeout("ref=GetInnerObject();",time1);
setTimeout("DoNavigation()",time2);
</script>
```

6.1.1. Creation of NavWin

In snippet #1, there are different ways to create NavWin. Table 4 shows some mechanisms.

Table 4: Some mechanisms for loading an HTML page

Loading a page by iframe (in the current HTML engine):

```
<iframe id=NavWinElem src="page1.html">
</iframe>
<script> NavWin=NavWinElem.contentWindow;
</script>
```

Loading a page by hosting an HTML engine as an ActiveX object in the current HTML engine:

```
<object data="page1.html" type="text/html"
id=obj> </object>
<script> NavWin=obj.object.parentWindow;
</script>
```

Loading a page by hosting a HTML engine as an ActiveX object in the current Javascript engine:

```
obj=new ActiveXObject('htmlFile');
NavWin=obj.parentWindow;
NavWin.location="page1.html";
```

For example: (1) The most straightforward mechanism is to define an iframe (or a frame) to load a page; (2) We described earlier that both the HTML engine and the Javascript engine can host ActiveX objects. Because the HTML engine itself is an ActiveX object, a window can be hosted by loading an HTML engine as an ActiveX object in PrstWin. In this scenario, NavWin and PrstWin are in different HTML engines; (3) Similarly, Javascript engine can also host an ActiveX object, which is through "new ActiveXObject()". For example, the ID of the ActiveX object for the HTML engine is "htmlFile". Using this method, an HTML engine is created in the address space of the Javascript engine.

In addition to HTML, other types of web documents, such as XML, Adobe Flash and Microsoft SilverLight, can be hosted by loading the corresponding runtimes as ActiveX objects. Currently, we have been focused on the native HTML only, and have not explored the mechanisms of loading other content types.

6.1.2. Referencing the inner object and navigating NavWin

Once NavWin is created, snippet #2 is to return an object (including properties and methods) in NavWin as an inner object to be held from PrstWin.

In snippet #3, there are various mechanisms to navigate NavWin to *page2.html*, such as: `open ("page2.html","NavWin")`, `NavWin.location ="page2.html"`, `NavWin.Navigate("page2.html")` and other navigation statements. Their internal execution paths are slightly different, which may affect the behaviors of page unloading and garbage-collection.

The Javascript code in Table 3 specifies that at time1, a reference is made from PrstWin to obtain the inner object by "`ref=GetInnerObject()`", then the navigation happens at time2.

6.2. Examining browser's memory states

The goal of each test case is to leave residue COM objects in the memory. In order to examine the exact reference relations of the COM objects, their validities and their refcounts, we added monitoring code in the browser to log critical function calls, and built an analysis tool to derive the states of COM objects from the log.

6.2.1. Event logging

For the simplicity of presentation, we refer to method calls as "events". We log all events of construction, destruction, addRef, release, validation and invalidation of CWindow, CWinProxy and CDocument objects. For every event, the log entry contains the method name, the object, its address, as well as important attributes of the object, e.g., the CWindow object associated to each CWinProxy object, etc. Because COM references do not contain the explicit information about these holders (i.e., objects holding the references), we also need to log the call stack for every event to help identify the holder. By logging all these events, we obtain the entire history of the state changes about CWindow/CWinProxy/CDocument objects when a test case runs through the browser.

6.2.2. Filtering out AddRef/Release pairs

We are interested in the objects that hold the references of residue objects after the navigation, but at the

logging time, it is too early to know which object will become residue objects, so we have to log the references to every CWindow/CWinProxy/CDocument. The numbers of log entries are very large even for very simple test cases. For example, we sampled a number of logs, and found that the average number of entries is 1329.

Fortunately, many events are AddRefs and Releases, which can be canceled out if we can match them in pairs based on their holders. The matching is technically non-trivial because an object is usually referenced by multiple holders, but the individual COM references do not identify their holders. To tackle this problem, we need to assign an identity to each reference. Since a reference in C++ is simply a pointer stored in a memory location, the memory location can be used as the identity, which we refer to as the *holder-mem* of the reference. An AddRef log entry and a Release log entry are a pair if they have the same *holder-mem*. Logging the *holder-mem* information requires adding code in the call sites of AddRef/Release. We have added the logging code in 76 such call sites throughout the browser code, and are able to match almost all the AddRef/Release pairs, and massively reduce the numbers of events in the logs.

6.2.3. Constructing object charts to examine the reference states

After filtering out the AddRef/Release pairs, the AddRef entries remaining in the log are those not released at the time when the logging ends. We developed a tool to construct an object chart based on the filtered log. Table 5 gives a simple test case to illustrate the reference graph obtained by the tool. In this test case, the inner object returned by `GetInnerObject` is the document in NavWin. We then navigate the NavWin from *page1.html* to *page2.html*, both being blank HTML pages.

Table 5: A simple test case

```
<iframe id=NavWinElem src="page1.html">
</iframe>
<script>
NavWin=NavWinElem.contentWindow;
function GetInnerObject() {
   return   NavWin.document;
}
function DoNavigation() {
  NavWin.navigate("page2.html");
}
... the rest of the code in Table 3 ...
```

When this test case runs on the browser, we obtain the object chart in Figure 4. Each object is shown by its class name and its address. The events of this object are shown along the time axis, with `C` for creation, `D` for destruction, `x` for invalidation and `v` for validation. The destructed objects are grayed out. Each CWinProxy entry also displays the address of the CWindow object that it represents.

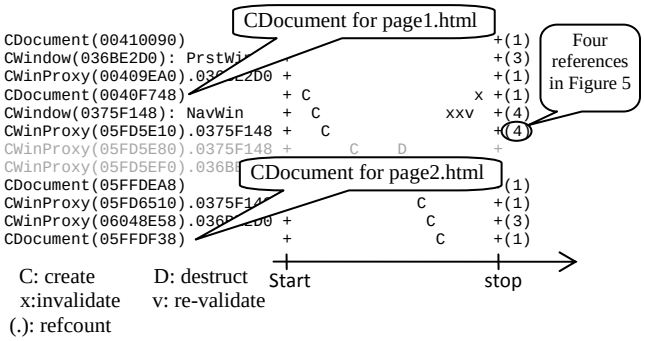


Figure 4: The object chart of the simple test case

The rightmost column in the chart shows the refcounts when the test case is finished. For example, the refcount of the CWindow representing NavWin is 4. The call stacks corresponding to the four references can be viewed by opening this particular entry. By examining the call stacks, we are able to see which data structures are holding references of this object. Figure 5 illustrates the four references of NavWin being held by various data structures when the test case finishes, including: (1) a reference from a window proxy held in PrstWin. It corresponds to the Javascript variable "`NavWin`" in Table 5; (2) a reference from a window proxy known as the "trusted proxy". The trusted proxy is used by NavWin's own objects to access NavWin; (3) a reference from the document inside NavWin corresponding to the `parentWindow` property of the document; (4) a reference from an object of the type CWebOC[2]. The insights about which objects are holding references of the target objects are valuable for us to see all potential access paths to the target objects.

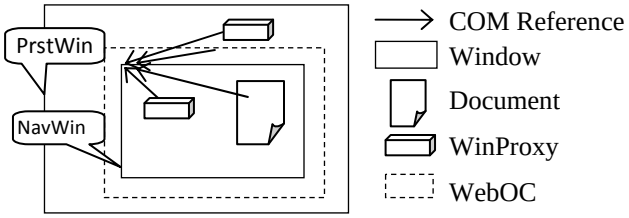


Figure 5: The references to the CWindow of NavWin, corresponding to the circled number in Figure 4

6.2.4. Limitations of the Analysis Approach

Currently, the analysis approach is still rudimentary, as its only goal is to identify residue objects, record their states and expose the reference relationships, which can be done automatically. However, as stated earlier, leaving residue objects in the memory does not necessarily imply security violations. It requires the analyst's insights to

[2] Because a browser can load non-HTML documents, such as those in PDF or MS Word, a generic window type is needed, which is CWebOC in IE. The functionality of WebOC is out of the scope of the discussion here. We simply show that every reference holder is revealed by the analysis approach.

specify the conditions to examine if the residue objects cause security consequences. In the next section, we explain real vulnerability cases and their violated conditions. Currently we are not clear about the complete set of these conditions as they appear to be very diverse. In this sense, we acknowledge that our understanding of the overall problem is still limited, and so is the analysis approach.

7. Recognizing the Challenges of the Residue Object Guarding Logic

This section explains the five new vulnerabilities found in our analysis. These vulnerability instances reveal many unexpected pitfalls to topple the effectiveness of the guarding mechanisms in Section 5. They also suggest that the prevalence of vulnerabilities caused by residue objects is probably much higher than what the existing literature recognized, thus the problem deserves more investigations.

7.1. Pitfall #1: "Invalidated" ≠ "Invisible"

We begin with a test case that is slightly more complex than the one in Section 6.2.3: we place an iframe in *page1.html*, which is loaded by NavWin. This iframe, namely InnerWin, is returned as the object to be held by PrstWin. Figure 6 shows the scenario and some entries in the object chart of this scenario.

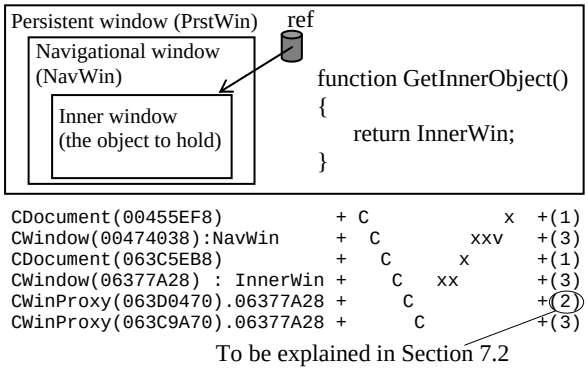


Figure 6: Holding an iframe as the inner object

We observe that InnerWin is invalidated, but is still in the memory. As stated in Section 2, visual spoofing is one of the potential consequences of residue objects. When an object is physically in the memory, even if it is marked as invalid, there is still the possibility of being visible, because the validity is just a Boolean flag of the object for access control, which does not necessarily imply its visibility.

The security of display can be independent from the access control. For example, in the above case, the iframe InnerWin visually disappears when NavWin is navigated. It is a secure behavior. However, besides iframe, there are other objects implemented using CWindow, such as modalDialog, modelessDialog and popup. ModalDialog and modelessDialog have their own window borders, so they are less of a visual spoofing concern. The popup object[3] is particularly interesting: it is usually used to implement tool tips in the hosting window, and does not have its own window border. We found that when a popup object created inside NavWin is held by PrstWin, it will remain visible, despite the invalidation, after the navigation of NavWin. Having this observation, we construct the visual spoofing scenario previously shown in Figure 1: we host PrstWin and NavWin in different tabs, create and hold a popup object to show the IEEE logo in NavWin, then navigate NavWin to *http://www.acm.org*. The logo is thus visually blended into the ACM page.

7.2. Pitfall #2: confusion due to the polymorphism of a pointer

Polymorphism is an important concept in the object-oriented programming: objects of concrete classes can be referenced by pointers of their abstract base class, and thus the detailed differences between the concrete classes are ignored. We stated earlier that a CWinProxy object is essentially a fake CWindow object: they share a common base class. In the browser code, an object of CWinProxy or CWindow is often referenced by a pointer of the base class, so the actual type of the object is not clear in the source code. We found a vulnerability apparently due to the confusion of such a pointer.

The issue surfaces when we hold a method of the inner window and navigate the middle window, and try to invoke it in the future. The `GetInnerObject` method is as follows.

```
function GetInnerObject() {
   return InnerWin.setTimeout;
}
```

The object chart of this scenario is similar to the one in Figure 6: the CWindow for InnerWin is invalid, while the two CWinProxy objects for InnerWin are still valid. The CWinProxy object from PrstWin to InnerWin has the refcount 2 (shown in the small circle in Figure 6). One of the two references is made in the constructor of CWinProxy and the other is made from a `CFuncPointer` object highlighted in the call stack below:

```
Stack top -> CWinProxy::AddRef
             CFuncPointer::AddRef   <- Call by a polymorphic pointer
             CAttrArray::Set
             VAR::InvokeByName
             CScriptRuntime::Run
```

- The real type of the polymorphic pointer is CWinProxy, as suggested by the call stack.

The `CFuncPointer` object is the COM object representing a method of an HTML element, such as `window.setTimeout`. The class definition of

[3] The term "popup" is overloaded. Some people refer to a newly created window as a popup, e.g., as in the term "popup blocker". In this paper, it refers to the specific object created by createPopup(), often used as a tool tip inside the hosting window.

`CFuncPointer` contains a method ID and a polymorphic pointer of the base class. However, the call stack clearly shows that `setTimeout` is associated with the CWinProxy object of the inner window, not the CWindow object of it. This is a key insight that cannot be obtained at the Javascript/HTML level. Because the CWinProxy object is still valid, `setTimeout` can still be invoked to schedule a delayed-execution in the invalidated inner window. Earlier in Figure 2, we showed an involuntary navigation attack against PayPal.com homepage based on this bug: a few seconds after the browser lands on the PayPal homepage, a script in the invalid (hidden) iframe navigates the browser to a fake page.

The unique aspect of this vulnerability is that despite the residue object being invalidated and a validity check being performed, the logic is still flawed because the check is performed on a wrong object, presumably due to the confusion of the polymorphic pointer in `CFuncPointer`.

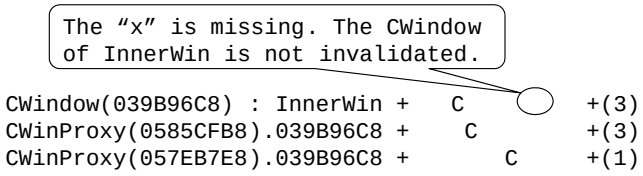
7.3. Pitfall #3: cross-engine invalidation

The residue object guarding logic becomes more challenging when different windows are in different HTML engines. We found a vulnerability in which the navigation does not trigger the necessary invalidation of a window in another HTML engine.

The test case is similar to Figure 6, except that the inner window is not an iframe inside NavWin, but a window in a new HTML engine hosted in NavWin. More specifically, NavWin contains the following <object> element. The window created by the <object>, referred to as the inner window, is returned by `GetInnerObject()`.

```
<object    id=obj    data="inner.html"
type="text/html">  </object>
function GetInnerObject() {
     InnerWin = obj.object.parentWindow;
     return InnerWin;
}
```

The inner window is in fact the top window of the new HTML engine created by <object>. We made an interesting observation about the CWindow object of the inner window: the "x", indicating the invalidation event, is missing in the object chart. The fact that the inner window is in a separate HTML engine complicates the invalidation logic and makes it error-prone. This error allows us to set a timer in the inner window for delayed execution of scripts, or display a dialog box or a popup object in NavWin's visual context.



This vulnerability shows that because the navigation can be started from one HTML engine instance and the target page can be hosted by another, cross-engine communication is needed to invalidate the page properly.

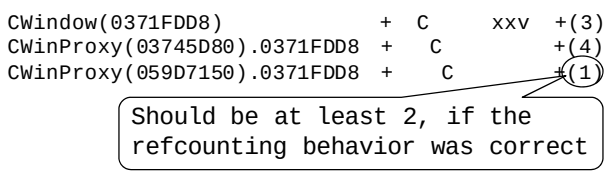
7.4. Pitfall #4: Erroneous refcounting

In the above scenarios, we see that because the residue objects are not securely guarded, they can later show their appearance in unexpected circumstances. On the other hand, when the browser frees the residue objects from the memory prematurely, there are possibilities of leaving dangling references in the hands of attack scripts, which result in memory safety bugs. We found such a vulnerability by observing the object chart.

In the scenario discussed in Section 7.2, PrstWin holds a reference of a method of the inner window. In the test case below, the inner object is a method of NavWin instead.

```
function GetInnerObject() {
//We hold NavWin's method, not InnerWin's
       return NavWin.setTimeout;
}
```

The object chart below exposes a fairly obvious problem: the refcount of the CWinProxy object of NavWin should be at least 2, because the constructor of CWinProxy sets the refcount one, and the CFuncPointer object of *setTimeout* also holds a reference of NavWin, as explained in Section 7.2. However, the object chart of this case shows the refcount 1. We searched in the event log and observed that the CFuncPointer object releases the reference during the navigation of NavWin, despite the holding of the method in variable `ref`. Obviously there is an error in refcounting.



Once this insight is obtained, causing the memory safety violation is easy: we navigate PrstWin to a blank page, thus NavWin is removed from PrstWin. This removal causes the last reference of the problematic CWinProxy to be released, and thus free the CWinProxy object from the memory. Since the variable `ref` still holds a reference to `NavWin.setTimeout`, invoking this method will trigger the dangling reference of the non-existent CWinProxy object. We observe that all the four bytes of the EIP register are corrupted and the browser's control flow is diverted randomly. A heap spray attack [13] can potentially cause a malicious binary code to run. After we reported this bug, Microsoft patched it in the February 2009 security hot fix.

7.5. Pitfall #5: Partially destroyed data structures inside valid objects

For an object that is not invalidated, it is important to ensure that none of its internal data structures is destroyed. Otherwise the script holding the object can cause memory violations by performing operations on this valid-but-partially-destroyed object.

We found a memory corruption exploit using a test case discussed in Section 7.3. The bug shown in Section 7.3 is because the CWindow object of InnerWin is not invalidated in a cross-engine situation, so that the residue InnerWin is fully functional after the navigation. In the memory corruption scenario, however, we found that as long as InnerWin contains a script in its child window, some internal data structures of InnerWin are destroyed after the navigation of NavWin, although the validity of InnerWin is not revoked. We observe that any write operation to the document in InnerWin results in a memory violation.

7.6. Summary of the discovered vulnerabilities

The discovered vulnerabilities indicate the challenges in actual browser implementations. Although the high-level policy seems simple – residue objects should be invalidated during navigations, and invalidated objects should not be invoked, mapping this policy into a concrete implementation is not straightforward. The decisions about three different properties of an object need to be correlated: (1) its existence, which is governed by the garbage-collection algorithm; (2) its validity, which is governed by the navigation logic; (3) its structural integrity, which is about the existences and the validities of its descendant objects. As shown in this section, making these decisions in a real browser implementation can be very subtle errors, and error-prone, e.g., in the cross-engine scenarios or due to polymorphism of pointers.

We can classify the discovered vulnerabilities in this section, as well as the vulnerabilities previous reported in Section 3, into two severity levels: (1) the memory corruption bugs are more traditional. If exploited successfully, they allow the malicious websites to run arbitrary binary code outside the browser sandbox, effectively compromising the browser machine; (2) other types of attacks do not directly compromise the browser machine, but can fake information of a trusted website, lure users to surrender passwords or expose sensitive online profiles to malicious websites. Today, as people increasingly rely on web applications to real-world tasks, these web-based compromises also impose realistic threats.

8. Possible Responses to the Residue Object Problem

We have seen that the correct logic for guarding residue objects is challenging in the real browser implementations. In this section we discuss potential opportunities that browser vendors may take in response to the problem. Especially, in the past two years, the browser industry and the research community have been undertaking some significant efforts attempting to fundamentally improve browser security in general. It is worth to discuss some of these efforts in the specific context of the residue object problem.

8.1. Writing browser code in languages with automatic garbage collectors?

Researchers have been undertaking the effort of implementing browsers using more sophisticated languages. For example, the HTML parser, the Javascript engine and the browser kernel of the OP browser are written in Java for stronger isolations [6]. The runtimes of many contemporary languages, such as Java virtual machines and the Common Language Runtime (CLR) for .NET, have built-in garbage collectors that are based on object-reachability. They do not require developers to do refcounting, and thus eliminate the memory-corruption bugs (if the language runtimes are correctly implemented).

The bugs at the DOM access control level and the visual representation level, however, cannot be addressed by having a more sophisticated garbage collector. When exploiting these bugs, the attack script always holds at least one reference to the object in order to avoid garbage collection, because the bottom line of any garbage-collector is that it cannot free such a object – if the garbage-collection algorithm is based on refcounting, the refcount of the object is at least 1, and thus it is not collectable; if the algorithm is based on object reachability, the object is reachable from a persistent data structure, and thus not collectable. In other words, the bugs above the memory level are a problem orthogonal to the choice of the underlying garbage collection algorithms.

8.2. Enhancing the browser's architecture for security?

A recent direction of browser architectural enhancements is to host web contents in different processes. We argue that the effectiveness of this general approach needs to be qualified to avoid overstatements. It is clear that when the attacker can execute binary code in a browser process (e.g., through a buffer overrun bug), the process boundary can prevent the process from directly overwriting the memory of other processes. This is a significant advantage over the "monolithic process" architecture.

However, a residue object bug enables the attack script to access an object through a path of references obtained from the browser. In such a scenario, whether each reference crosses the process boundary is unimportant: in fact, COM programs observe very few barriers of the process boundary, because of the DCOM architecture [16]. A COM object can be easily created in another process, and anyone holding its reference can invoke it without

knowing that it is a cross-process invocation. For example, IE8 deploys a multi-process architecture to render different windows. We tested IE8 using the five exploits that we found in IE7: three of them still work against IE8; the exploit in Section 7.4 has been blocked specifically in response to our bug reporting; the other exploit (in Section 7.2) does not seem to work against IE8 because of an access policy of DOM is changed. These test results suggest that if the object reference policies remain unchanged, merely placing different windows in different processes will not address the residue object problem. Instead, the key question to investigate is when and why some references exist in the subtle corner-case situations, and how to specify precisely the DOM access policies to restrict the references but still permit legitimate functionalities.

Chrome also deploys a multi-process architecture so that different windows can be placed in different processes. Four browsing models are implemented in Chrome: monolithic, process-per-browsing-instance, process-per-site-instance and process-per-site, each having a specific policy for hosting windows in different processes. The architecture is mainly designed for robustness and responsiveness of the browser [14]. Similar to IE8, we argue that such process-based isolations are not fine-grained enough to address the residue object problem that is mainly about COM-level references and accesses.

It is encouraging that the research community is looking into more profound security enhancements. Browser prototypes, such as OP [6] and Gazelle [7], demonstrate a number of endeavors which may address the residue object problem. OP and Gazelle not only deploy multi-process architectures, but also spend much effort on the policies on top of the architectures. For example, Gazelle defines how to treat subdomains, manage display regions and potentially sacrifice a small degree of compatibility to change some access control policies. In the development process of the OP browser, formal methods were used to check the implementation against the same-origin-policy and certain display properties. We believe that these steps toward a more secure object model will be the key efforts to address the problem that we discuss in this paper. Of course, OP and Gazelle are not full-fledged browsers yet. With more functionalities supported, dealing with the logic complexity and the compatibility will become more challenging.

8.3. Systematic testing of browser security logic?

Besides the direction taken by OP and Gazelle, we believe that systematic testing is another promising approach to address the problem.

The analysis approach presented in this paper gives a sketch of a testing methodology. In order to expose logic bugs, we need to enumerate the mechanisms for creating documents, holding objects and navigating windows. In a modern browser, there are several object models, such as the HTML DOM, the XML DOM, the Flash DOM, etc. They all need to be examined. When a test case runs on the browser, we need to log and analyze critical events of refcounting, validity change and visual disappearance. These events contain the crucial information to expose the class of bugs, according to our experience.

9. Related Work

Browser security is becoming an important research area. A number of recent papers discuss specific classes of browser vulnerabilities, such as DNS-rebinding [8], dynamic pharming [9] and cross-domain attacks [3]. A recent paper describes a class of vulnerabilities called Javascript capability leaks [2], which is a type of cross-domain bugs in browsers. Some techniques for defeating cross-domain attacks are proposed, e.g., script accenting [3] and Javascript reference leak detection [2]. Because a subset of residue object bugs result in cross-domain accesses, these defense techniques are good mitigations for such bugs. However, they are not designed to mitigate other types of residue object bugs.

Residue objects are the objects that are functionally useless but stay in the memory. In a broad sense, research about excessive data lifetime is related to our work. Chow et al found that many privacy-sensitive data, such as Windows logon password, remain in user or kernel memory for indefinite periods after their memory blocks are returned back to the system [5]. This is a privacy concern because if the system is compromised at the binary-executable level, the attacker can dump the memory to get the data. Chow et al proposed a secure deallocation mechanism to zero the data blocks within a short period of time after they are freed [4].

10. Conclusions and Future Work

Residue object is a problem that all browsers need to face, because it is due to a fundamental dilemma between the basic mechanisms in browsers: the scripting capability of HTML, the garbage collection and the navigation. When not properly guarded, residue objects can cause violations of basic security properties, such as visual spoofing, involuntary navigation, cross-domain access and memory corruption. We show that IE, Firefox and Safari had this type of vulnerabilities publicly reported in the past.

We conducted a focused study about IE's mechanisms for guarding residue objects. Our analysis approach is to examine the browser's logic by enumerating different residue objects and deriving object states at the memory level. Using this approach, we discovered five new vulnerabilities, one of which was patched by Microsoft in a hot-fix. More importantly, this study gives answers to our initial motivating questions: (1) we use concrete examples to show why the seemingly simple guarding mechanisms are difficult to implement securely, and where some of the pitfalls are; (2) we empirically show that the actual

prevalence of this type of vulnerabilities can be significant, if the browsers are under a focused examination.

We argue that the problem deserves more efforts from the industry due to the non-trivial logic involved. Since the browser industry today is undertaking serious efforts to fundamentally improve the overall security of browser products, it is necessary to recognize the residue object problem so that it can be fundamentally addressed in such efforts. By formulating the problem and showing concrete instances, this work gives the initial endeavor toward the exploration in this problem space.

For the future work, we plan to study non-HTML document types, such as XML, Adobe Flash and Microsoft SliverLight. These runtimes have their own document object models, their own same-origin policies, and the interoperability with HTML/Javascript. We believe that there are also subtle logic errors of this type.

Further down the road, we plan to extend the study on other browsers. We did a preliminary study about the residue-object-guarding code of Firefox, and identified the objects representing window and document, as well as the flag indicating validity. The detailed mechanisms in Firefox are different from IE, e.g., in Firefox, writing HTML texts and scripts to a residue document object (`nsHTMLDocument`) seems legitimate, but scripts should not be allowed to run in residue documents. These mechanisms involve window reference holdings and validity flag settings, etc. They also need to be thoroughly examined because they seem as non-trivial as the ones in IE.

Acknowledgements:

We thank Emre Kiciman and Zhenbin Xu for valuable technical discussions. We also thank anonymous reviewers and our shepherd Ashvin Goel for insightful comments.

References:

[1] Don Box, “Essential COM,” ISBN 0-201-63446-5, Addison-Wesley 1998

[2] Adam Barth, Joel Weinberger, and Dawn Song. “Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense,” In Proc. of the 18th USENIX Security Symposium, 2009

[3] Shuo Chen, David Ross, Yi-Min Wang, “An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism,” in ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Oct-Nov 2007.

[4] Jim Chow, Ben Pfaff, Tal Garfinkel, Mendel Rosenblum, “Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation.” In Proceedings of the 14th USENIX Security Symposium, August 2005.

[5] Jim Chow, Ben Pfaff, Tal Garfinkel, Mendel Rosenblum, “Understanding data lifetime via whole system simulation.” In Proc. of the 12th USENIX Security Symposium, 2004.

[6] Chris Grier, Shuo Tang, and Samuel T. King, “Secure web browsing with the OP web browser”, Proceedings of the 2008 IEEE Symposium on Security and Privacy, May 2008.

[7] Chris Grier, Helen J. Wang, Alexander Moshchuk, Samuel T. King, Piali Choudhury, Herman Venter.”The Multi-Principal OS Construction of the Gazelle Web Browser,” Proceedings of the 18th USENIX Security Symposium, Montreal, Canada, August 2009.

[8] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh, “Protecting Browsers from DNS Rebinding Attacks,” the Fourteenth ACM Conference on Computer and Communications Security (CCS 2007), November 2007.

[9] Chris Karlof, Umesh Shankar, J.D. Tygar, and David Wagner, “Dynamic Pharming Attacks and Locked Same-origin Policies for Web Browsers,” the Fourteenth ACM Conference on Computer and Communications Security (CCS 2007), November 2007.

[10] Mozilla, “XPCOM”, http://www.mozilla.org/projects/xpcom/index.html

[11] Mozilla, “XPConnect (Scriptable Components)”, http://www.mozilla.org/scriptable/

[12] Object Capability Model. http://c2.com/cgi/wiki? ObjectCapabilityModel

[13] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. “Nozzle: A Defense Against Heap-spraying Code Injection Attacks,” Microsoft Research Technical Report MSR-TR-2008-176, November 2008.

[14] Charles Reis, Steven D. Gribble. “Isolating Web Programs in Modern Browser Architectures,” Eurosys 2009. Nuremberg, Germany, April 2009.

[15] Jesse Ruderman. “The Same Origin Policy,” http://www.mozilla.org/projects/security/components/same-origin.html

[16] Markus Horstmann and M. Kirtland, “DCOM Architecture,” http://msdn.microsoft.com/en-us/library/ms809311.aspx

[17] SecurityFocus Vulnerability Repository. http://www.securityfocus.com/bid

[18] Net Applications. “Browser Market Share of March 2009” http://marketshare.hitslink.com/browser-market-share.aspx?qprid=0

Appendix: Bug Reports in SecurityFocus.com

A1: Bugs in IE

BugID: 5196 Date: 2002-07-10

Bug Title: Microsoft Internet Explorer OBJECT Tag Same Origin Policy Violation Vulnerability

URL: http://www.securityfocus.com/bid/5196

Description snippet: … Malicious script code may **obtain a legitimate reference** to an embedded object containing a web page from the same domain. This script may **then change the location** of the embedded object to a sensitive page, and maintain the reference to the object. This provides **full access to the DOM** of the embedded page....

Exploit code snippet:

```
<object id="data" data="empty.html" type="text/html">
</object>
var  obj = document.getElementById("data")          //step1
var ref = obj.object;                               //step2
ref.location.href = "http://targetSite.com";        //step3
setTimeout("read(ref.cookie)",5000);                //step4
```

BugID: 5841 Date: 2002-10-01

Bug Title: Microsoft Internet Explorer Document Reference Zone Bypass Vulnerability

URL: http://www.securityfocus.com/bid/5841

Description snippet: … to execute script code in the context of other domains … **access to a document object is attempted through a saved reference** ...

Exploit code snippet:

```
open(location+"2").blur();                  //step1
f = opener.location.assign;                 //step2
opener.location="res:";                     //step3
...
f("javascript:c1.Click();c2.Click();");     //step4
```

BugID: 6028 Date: 2002-10-22

Bug Title: Multiple Microsoft Internet Explorer Cached Objects Zone Bypass Vulnerability

URL: http://www.securityfocus.com/bid/6028

Description snippet: … **creating a reference** to several methods of the target child window… then **have the child window open a website in a different domain** ...

Exploit code snippet:

```
var oWin=open("blank.html","victim");                  //step1
[Cache line here]          //"cache" means "make a ref" – step2
location.href="http://google.com";                     //step3
setTimeout(
      function () {[Exploit line(s) here]},3000        //step4
);
```

A2. Bugs in Firefox

BugID: 13230

Bug Title: Mozilla Suite And Firefox Global Scope Pollution Cross-Site Scripting Vulnerability

Date: 2005-04-16

URL: http://www.securityfocus.com/bid/13230

Description snippet: A remote cross-site scripting vulnerability affects Mozilla Suite and Mozilla Firefox **because the software fails to properly clear stored parameters.**

As you browse from site to site each new page should start with a clean slate. "shutdown" reports a technique that pollutes the global scope of **a window in a way that persists from page to page**.…

Exploit code snippet: Not disclosed to SecurityFocus

BugID: 17671

Bug Title: Mozilla Firefox iframe.contentWindow.focus Deleted Object Reference Vulnerability

Date: 2006-04-24

URL: http://www.securityfocus.com/bid/17671

As the title suggested, this is due to a deleted object.

Exploit code snippet:

```
var ifr;
ifr = document.createElement("iframe");     //step1 & step2
htmlarea.appendChild(ifr);
var doc = ifr.contentWindow.document;
doc.write("<iframe src=' '>");          //step3: unload the object
ifr.contentWindow.focus()               //step4: invoke the object
```

BugID: 24286

Bug Title: Mozilla Firefox About:Blank IFrame Cross Domain Information Disclosure Vulnerability

Date: 2007-06-04

URL: http://www.securityfocus.com/bid/24286

Description snippet: Mozilla Firefox is prone to a cross-domain information-disclosure vulnerability because **scripts may persist across navigations**.....

Exploit code snippet: Not disclosed to SecurityFocus