Browser Extension-Based Man-in-the-Browser Attacks against Ajax Applications with Countermeasures

Sampsa Rauti Ville Leppänen

Abstract: *As the web pages today rely on Ajax and JavaScript, a larger attack surface becomes available. This paper presents in detail several different man-in-the-browser attacks against Ajax applications. We implemented browser extensions for Mozilla Firefox to demonstrate these attacks and their effectiveness. Some countermeasures to mitigate the problem are also considered. We conclude that man-in-the-browser attacks are a serious threat to online applications and there are only partial countermeasures to alleviate the problem.*

Key words: *Man-in-the-browser, Ajax, Browser extensions*

1 INTRODUCTION

Man-in-the-browser is a Trojan horse that infects a web browser and has the ability to tamper with the contents of web pages and transactions. For example, the Trojan can wait that the user goes to a predefined web page and logs in. When the user inputs sensitive data, it silently modifies the data on submit without the user noticing anything suspicious. If a confirmation is shown to user after the transaction, the Trojan replaces the modified data with the correct values given by the user. The Trojan can also steal important data, but here we concentrate on malware that alters data.

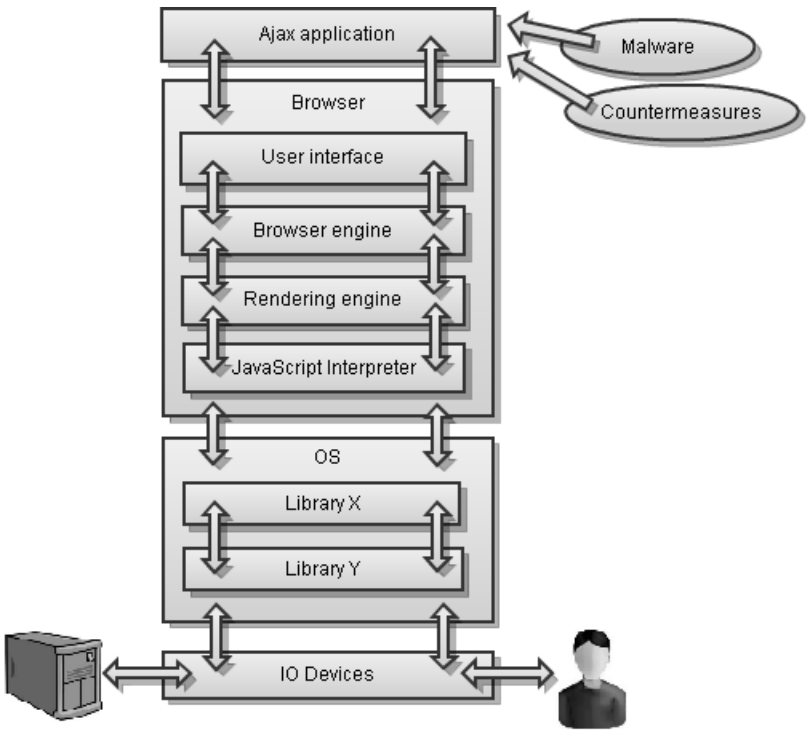

Figure 1. *Path of data from user to server and different levels involved. In this paper, we mainly limit our attention to Ajax application level.*

The fact that data was altered before it was sent to the server is completely invisible to the user. Neither the user nor the server can detect anything unusual, because the malware acts between them and delivers fallacious information to both parties. In this sense, a man-in-the-browser attack bears resemblance to the traditional man-in-the-middle attack. Man-in-the-browser, however, is able to bypass all the modern authentication mechanisms like username/password pair, PIN/iTAN pair and SmartCard, because it acts in the application logic and UI, not at the authentication level or in secure communication channel.

As financial institutions have moved online, the amount of Trojans intercepting and manipulating user input has increased rapidly in recent years [7]. Because the customer's computer is the weakest link in the chain, attackers usually choose the client side as their target rather than trying to penetrate servers.

There are many possible ways to realize a man-in-the-browser attack, but in this paper we concentrate on Ajax application level, as shown in Figure 1. This kind of attack can be carried out by implementing a malicious browser extension, which can be installed on user's computer by using a security hole or social engineering. This unfortunately opens many possibilities for a malevolent adversary. As the web pages today rely heavily on JavaScript and Ajax technology, a wide range of attack methods becomes available.

In order to see how effective and easily achievable these attacks would be, we implemented several test extensions for Mozilla Firefox. The detailed analysis of four different attack patterns and some proposed countermeasures form the main contributions of this paper.

It turns out that the browser extensions we built are very powerful and building this kind of malicious programs is fairly easy. Because of this, the man-in-the-browser attacks are a real problem with only partial countermeasures available.

Section 2 presents the state of related work. Section 3 contains an introduction to Ajax and browser extensions. In Section 4, we introduce various ways to apply man-in-the-browser attacks against Ajax applications by using browser extensions. Section 5 proposes countermeasures for the attacks, and Section 6 concludes the paper and discusses future work.

2 RELATED WORK

Despite the threat it poses, the man-in-the-browser problem has received relatively little scientific attention. The attack was first introduced by Augusto Paes de Barros [6], and Philipp Gühring later presented a more detailed description of the problem and possible countermeasures [4]. Dougan and Curran recently published a comprehensive review on man-in-the-browser attacks and several related Trojans [2].

Several solutions to the man-in-the-browser problem have been suggested. Among the most effective ones are out-of-band authentication [8] and a separate device with keyboard and display that digitally signs user's transaction. Out-of-band authentication usually makes use of SMS and can be defeated by a Trojan targeting user's mobile phone. A device for signing user's transactions may be fairly impractical for the user. Both of these approaches also partially take Internet banking off the Internet.

One further solution is to try to track down unusual patterns in user behaviour in order to tell apart the malware and the users performing transactions [3]. However, for the man-in-the-browser Trojans that only change a few transaction details, and let the user take care of everything else, there is not much unusual behaviour to be detected. Trojans that navigate through several pages or quickly fill in forms on their own may fall victim to behavioural analysis, but a simple content tampering Trojan is hard to spot with this method.

Traditional anti-virus technology has also proved quite ineffective against man-in-the-browser Trojans. Since the smart Trojans continuously keep mutating, only a minority of them will be found using anti-virus programs [10].

There has not been much research on malicious browser extensions. Mike Ter Louw et al. [9] propose a security solution that employs integrity checking and runtime monitoring to defend against malware extensions in Mozilla Firefox. While this is a good security practice, it requires some changes in the browser itself which still have not been implemented by Mozilla. As our work proves, it is still very easy to build and run harmful Firefox extensions.

3 AJAX AND BROWSER EXTENSIONS

Ajax (Asynchronous JavaScript and XML) is a group of features and techniques to make web sites and applications more interactive. The browser exchanges data with the server on the background, without full page reloads. Ajax combines the use of HTML, CSS, JavaScript and DOM and XMLHttpRequest object.

XMLHttpRequest is used to communicate with the server asynchronously. XML or JSON, for example, can be used as a data interchange format. Using JavaScript and DOM (Document Object Model) manipulation, new information can be dynamically added to the application based on the received server responses. HTML and CSS are used to modify the mark-up and appearance of the web page.

Browser extensions are powerful enhancements extending browser's core functionality. For the purposes of this article, the term extension means a browser extension for Mozilla Firefox, currently the most popular browser with 36.6 % usage [11]. Because these extensions are implemented in JavaScript, all Ajax techniques are available as well. Moreover, Firefox extensions are not sandboxed but are run with full privileges in the browser. As a result, an extension can freely modify any web page using DOM, write to the file system or tamper with the browser's HTTP requests, for example. It can even hide itself from the extension list or inject itself into another extension [9].

Virtually all functionality in the browser is available for extensions through XPCOM (Cross Platform Component Object Model). A special layer of this component object model, XPConnect, can be used to reflect XPCOM library interfaces to JavaScript so that browser extensions can make use of them.

4 MAN-IN-THE-BROWSER ATTACKS BASED ON BROWSER EXTENSIONS

Inside the browser, there are several layers where man-in-the-browser attacks may take place. We now present four different attack patterns, with a special emphasis on Ajax applications. We tested these attacks by implementing several Firefox extensions. Some attacks include code samples, but the complete source codes of extensions are not presented in order to avoid distributing harmful code.

It is also worth noting that these extensions are somewhat simple compared to real-world malware. These programs can usually perform several other functions along with altering the data given by the user. For instance, Zeus, one of the most popular pieces of malware performing the man-in-the-browser attack, can grab all kinds of data from user's computer and communicate with the headquarters for further instructions [2]. However, the data modifying part in many malevolent programs is very similar to the methods we present here.

4.1 Modifying payload

Mozilla's XPCOM interfaces allow extensions to observe and modify all HTTP requests the browser makes and all responses it receives. We can listen to notifications that are sent by Firefox each time when a request or response takes place.

```
var httpRequestObserver = {

  observe: function(aSubject, aTopic, aData) {

    if (aTopic == "http-on-modify-request") { // HTTP request is being made
      var httpChannel = aSubject.QueryInterface(Components.interfaces.nsIHttpChannel);

      if(httpChannel.requestMethod == "POST") { // POST request
        // Get the HTTP channel and the upload channel associated with it
        var uploadChannel = httpChannel.QueryInterface(Components.interfaces.nsIUploadChannel);
        var uploadChannelStream = uploadChannel.uploadStream;
        uploadChannelStream.QueryInterface(Components.interfaces.nsISeekableStream)
                           .seek(Components.interfaces.nsISeekableStream.NS_SEEK_SET, 0);

        // Read the POST data from upload channel's stream
        var stream = Components.classes["@mozilla.org/binaryinputstream;1"]
                               .createInstance(Components.interfaces.nsIBinaryInputStream);
        stream.setInputStream(uploadChannelStream);
        var postBytes = stream.readByteArray(stream.available());
        var poststr = String.fromCharCode.apply(null, postBytes);

        // Modify POST string here

        // Set a new upload stream with modified payload and resume request
      }

    }

  }

};

var observerService = Components.classes["@mozilla.org/observer-service;1"]
                                .getService(Components.interfaces.nsIObserverService);
observerService.addObserver(httpRequestObserver, "http-on-modify-request", false);
```

Figure 1: Part of the JavaScript code for a payload modifying extension

The source code example in Figure 1 shows an implementation of nsIObserver, an interface used to listen to notifications. This listener is added to an observer service in order to receive notifications regarding a specific topic. This is done on the last line in the sample code. In this case, we are interested in http-on-modify-request topic, because we want to intercept outgoing traffic.

The observe function receives and handles all notifications. Associated with an arriving notification is an HTTP channel that implements nsIHttpChannel interface. This interface, in turn, includes an upload channel, nsIUploadChannel. Using an upload stream of this channel, the request's POST data can be obtained. This data can now be modified and put back in the HTTP channel. When everything is done, the request is resumed with altered data. Altering data and preparations for continuing the request are left out from the code sample.

Similarly, all HTTP responses can be captured by listening notifications with http-on-examine-response topic. A response can be read and modified via nsITraceableChannel interface [5].

Using nsIObserver interface to listen notifications, the attacker can capture both traditional HTTP traffic and asynchronous data interchange of XMLHttpRequest objects. Moreover, the adversary does not necessarily even have to know the structure of the application he targets very well when he uses this method. Performing blind string replacements on outgoing and incoming data is sufficient in many cases. Even regular expressions could be used. For example, substrings matching a given account number format could be replaced with the attacker's own account number.

The downside of this attack type is that compared to other methods, it is harder to implement and somewhat error-prone.

4.2 Modifying Ajax transmission mechanism

There are no classes in JavaScript. Instead, objects are created based on prototypes. As a result, modifying the payload of any Ajax request is possible by overriding XMLHttpRequest's prototype implementation, a method formerly presented in [1]. An attacker can modify XMLHttpRequest's prototype by adding malicious code as a part of send function:

```
XMLHttpRequest.prototype.originalSend = XMLHttpRequest.prototype.send;

var evilSend = function(data) {
  // Modify the data here
  this.originalSend(data);
};

XMLHttpRequest.prototype.send = evilSend;
```

All XMLHttpRequest objects now use this new implementation of send function, which silently modifies the data and then passes it to the original send function each time a request is sent.

XMLHttpRequest's responseText, which contains the server response, is a read-only property, so we can not edit it directly. However, we can replace XMLHttpRequest's onreadystatechange callback function:

```
XMLHttpRequest.prototype.onreadystatechange = handler;
```

This function can be used to handle responses from server and simulate application's normal behaviour. To achieve this, however, the attacker has to understand the inner workings of the application well. This is possible in Ajax applications, because all code on the client side is available for the attacker.

4.3 Modifying DOM tree

Another easy way to alter user input is to use JavaScript to manipulate elements in a web document's DOM tree. Assume the author of the malicious extension wants to secretly change value of a text field that will be sent to the server. To achieve this, following steps can be used:

1) Listen to the click events in the document body. If a text field is clicked, it is immediately made invisible using CSS style definitions and its value is changed according to the choice of the attacker.
2) A new, identical text field is put in the place of the original, now invisible text field. The focus is set on the new element.
3) User types the input in the new text field without noticing anything suspicious.
4) When the data is submitted, the value of the original text field is sent to the server and the new text field filled by the user is saved for later use.

When the modified text is sent back from the server, the attacker has to replace it with the original value to fool the user. In Ajax applications, there is no full page load the malware could take advantage of. However, to display the data submitted by user, the application has to edit the DOM tree structure. As a consequence, the attacker can listen to DOMNodeInserted event, for example, and immediately replace the new element's innerHTML when it is inserted.

Modifying DOM tree is a relatively easy attack method and this is probably why some existing Trojans use this approach.

4.4 Modifying Ajax application functionality

This method can be seen as a special case of modifying DOM tree. However, in this scenario we do not alter any visible elements in the tree. Instead, we can replace and add script elements to modify Ajax application's functionality.

We can either add our malicious code to existing functions or replace functions altogether with our own implementation. DOM replaceChild, removeChild and appendChild methods can be used to achieve this. For example, it is possible to check where XMLHttpRequestObject's send function is called and replace the data given as an argument to this function. We simply make a string replacement to the code in the original script element and replace this old element with a new script element containing the modified content.

If we want to overwrite an entire function, as we might want to do to the function that handles Ajax responses, we can simply append a new script tag containing our own implementation. If there are two function declarations in JavaScript with the same name, the first one is overwritten.

5 COUNTERMEASURES

There is no silver bullet against malware. Man-in-the-browser Trojans have demonstrated ability to defeat practically all existing security measures. In this section, we propose several techniques to mitigate man-in-the-browser attacks in the Ajax application's JavaScript code itself. No changes to the browser are necessary.

As all the JavaScript functionality in an Ajax application can be modified, these countermeasures can be defeated as well. Trojans are getting smarter all the time and many of them already contain functionality to defeat the preventive actions aimed against them. However, the methods we present here still mitigate the problem in many cases and make attacking the application considerably harder.

5.1 Modifying payload

An apparent security solution against payload modification is to encrypt data before it is sent to the server. This way, no simple HTTP request listener can read or modify data.

The attacker can no longer replace the data directly in the HTTP channel. Instead, he has to take a look at the implementation of the Ajax application and find new ways to attack it. The obvious downside of encryption is that we need to implement it on the client side, where the attacker can read its source code.

5.2 Modifying Ajax transmission mechanism

Encryption also works against data modification in an XMLHttpRequest object. The situation is similar to payload modification with XPCOM. Malicious code looking for some specific string or regular expression in the data will fail because of encryption. Even if it alters the data, encrypted data gets corrupted and will most likely not pass validity tests on the server side.

Because modifying the Ajax transmission mechanism is based on overriding XMLHttpRequest's prototype, we could also replace it with a secure implementation before sending the data. Moreover, we could periodically ensure that the responses are indeed processed by the correct handler function.

5.3 Modifying DOM tree

Encrypting data will not work against DOM tree modification because the attacker can modify data before the encryption process takes place. Therefore, we have to find new approaches to thwart this attack.

Elements can be made harder to access by randomizing elements' id attributes or removing the id altogether. Simple Trojans using getElementById will fail with this approach. Elements' indices and levels in the tree can also be varied arbitrarily.

Ajax applications often temporarily hide widgets, but altering the value of an invisible text field is suspicious. We could monitor value changes in hidden elements, as this is rarely part of applications' normal functionality.

5.4 Modifying Ajax application functionality

Again, if we use random function names and vary their implementation, it is harder for the attacker to modify the application's functionality without breaking it. The server could even switch parts of the application on the fly by sending new implementations for specific functions.

5.5 Supplementary countermeasures against the attacker

When encryption is used as a countermeasure, the adversary might attack the encryption function itself. After all, nothing is safe on the client side. To make this a little bit harder, we could randomize the name and implementation of the encryption function. We can create many different implementations for the same function by varying control structures, changing the order of statements and adding all kinds of complexity to make the code harder to read.

If the attacker has enough time, he can always decipher the obfuscated source code. However, we can limit this time frame by providing different implementations for certain functions, like the encryption function, in each session. These implementations can also be changed dynamically during execution of the application to give the adversary less time to analyze them.

All the countermeasures listed in this section increase the application's complexity. However, this kind of preventive functionality could be packaged in its own library and the server side can obfuscate the code automatically. Moreover, in the frameworks like Google Web Toolkit where the programmer only writes Java and the application's JavaScript code is automatically generated, there is no problem with JavaScript to begin with. This way, the security measures will not make the programmer's life too complicated.

6 CONCLUSIONS AND FUTURE WORK

Man-in-the-browser attacks pose a serious threat for many online services. This problem is enhanced by the powerful browser extensions and growing attack surface of rich internet applications.

We have seen that it is relatively easy to manipulate user inputs and server responses with many different techniques. Because they are not sandboxed at all, Firefox extensions are perhaps too powerful. While it is true that many benign extensions need strong privileges, techniques to restrict functionality of malicious extensions should be implemented. This problem is not only present in Firefox but in several other browsers, like IE, as well. It is important to note, however, that the browser extensions are not the only way to realize man-in-the-browser attack. The browsers have other vulnerable points as well.

There are only partial countermeasures to mitigate man-in-the-browser attack, as all existing preventive technologies have their weaknesses. The techniques proposed in this paper have flaws as well, because they are implemented on the target site in JavaScript which can be overwritten by the attacker. However, as discussed previously, it may be

pretty difficult and time-consuming to make sense of obfuscated code that is a little different every time. The dynamically changing implementations limiting attacker's time frame make this job a whole lot harder.

Future work will focus on the countermeasures against man-in-the-browser attacks and practical implementations of these methods.

REFERENCES

[1] Di Paola S., Fedon G. Subverting Ajax. *In Proceedings of 23rd CCC Conference*, 2006.

[2] Dougan, T. & Curran K. Man in the Browser Attacks. *International Journal of Ambient Computing and Intelligence*, vol. 4, no. 1, pp. 29-39, March 2012.

[3] Entrust (2010, Mar.). Defeating Man-in-the-Browser. How to Prevent the Latest Malware Attacks against Consumer & Corporate Banking. 2010. [Online]. Available: http://download.entrust.com/resources/download.cfm/24002/

[4] Gühring, P. (2006). Concepts against Man-in-the-Browser Attacks. [Online]. Available: http://www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf

[5] Odvarko, J. (2008). nsITraceableChannel, Intercept HTTP Traffic. [Online]. Available: http://www.softwareishard.com/blog/firebug/nsitraceablechannel-intercept-http-traffic/

[6] Paes de Barros, A. (2005). O futuro dos backdoors, o prior dos mundos. [Online]. Available: http://www.paesdebarros.com.br/backdoors.pdf

[7] RSA (2011). Making Sense of Man-in-the-Browser Attacks. Threat Analysis and Mitigation for Financial Institutions. [Online]. Available: http://www.rsa.com/products/consumer/whitepapers/10459_MITB_WP_0611.pdf

[8] SafeNet (2010). Man-in-the-Browser. Understanding Man-in-the-Browser Attacks and Addressing the Problem. [Online]. Available: http://ru.safenet-inc.com/uploadedFiles/About_SafeNet/Resource_Library/Resource_Items/White_Papers_-_SFDC_Protected_EDP/Man%20in%20the%20Browser%20Security%20Guide.pdf

[9] Ter Louw, M., Lim J. S., Venkatakrishnan, V. N. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, vol. 4, no. 3, pp. 179-195, August 2008.

[10] Trusteer (2009, Sep.). Measuring the in-the-wild effectiveness of Antivirus against Zeus. [Online]. Available: http://www.trusteer.com/files/Zeus_and_Antivirus.pdf

[11] W3Schools. Browser statistics. [Online]. Available: http://www.w3schools.com/browsers/browsers_stats.asp

ABOUT THE AUTHORS

PhD student Sampsa Rauti, Department of Information Technology, FIN-20014 University of Turku, FINLAND. E-mail: Sampsa.Rauti@utu.fi.

Prof. Ville Leppänen, Department of Information Technology, FIN-20014 University of Turku, FINLAND. E-mail: Ville.Leppanen@utu.fi.