Building Anti-Phishing Browser Plug-Ins: An Experience Report

Thomas Raffetseder, Engin Kirda, and Christopher Kruegel
Secure Systems Lab, Technical University of Vienna
{tr,ek,chris}@seclab.tuwien.ac.at

Abstract

Phishing is an online identity theft that aims to steal sensitive information such as user names, passwords, and credit card numbers. Although phishing is a simple social engineering attack, it has proven to be surprisingly effective. Hence, the number of phishing scams is continuing to grow, and the costs of the resulting damages is increasing. Researchers as well as the IT industry have identified the urgent need for anti-phishing solutions and recently, a number of solutions to mitigate phishing attacks have been proposed. Several of these approaches are browser plug-ins.

In 2005, we implemented a Firefox anti-phishing browser plug-in called AntiPhish. After releasing AntiPhish, we decided to port it to the Microsoft Internet Explorer (IE) browser. Supporting IE was important because a majority of Internet users are accessing the web with this browser. Our initial expectation at the beginning of the project was that porting a browser plug-in that is written for Firefox to IE could not be too difficult; after all, browser plug-ins are conceptually similar. However, creating an anti-phishing browser plug-in for the IE proved to be much more challenging than expected. In this paper, we report on our experience in implementing anti-phishing (i.e., security) browser plug-ins and summarize five lessons we learned from our undertaking.

Keywords: Phishing, Security, Browser Helper Objects, .NET, Internet Explorer, Firefox

1 Introduction

Phishing is an online identity theft that aims to steal sensitive information such as user names, passwords, and credit card numbers. Identity and credential theft has been in the portfolio of criminals since computers became part of our lives. Today, online transactions worth billions of dollars are initiated every day, thus making this sector a prime target to criminals all over the world. A majority of phishing victims are online banking users whose credentials are stolen. These credentials are typically used to login into the online banking web site and to transfer money to the attacker's bank account. Although phishing is a simple social engineering attack, it has proven to be surprisingly effective. Hence, the number of phishing scams is continuing to grow, and the costs of the resulting damages is increasing. For example, the Anti-Phishing Working Group received over 35,000 unique phishing reports in November 2006, compared to around 7,000 in November of the year before [2].

Figure 1 shows a phishing email that targets the customers of a U.S. bank. The attackers have imitated the look and feel of the Bank of America online banking web site with the aim of fooling users into believing that the mail is authentic. The mail is asking users to update their personal information. Once a victim follows the link, she is prompted to enter her user ID, password, and ATM or check card number.

Phishing is a continuing problem and both academic researchers and the IT industry have identified the urgent need for anti-phishing solutions. Recently, a number of solutions to mitigate phishing attacks have been proposed. Several of these approaches are browser plug-ins.

In 2005, we implemented a Mozilla Firefox anti-phishing browser plug-in called AntiPhish [6]. AntiPhish is a component that is integrated into the web browser. It keeps track of a user's sensitive information (e.g., a password) and prevents this information from being passed to a web site that is not considered trusted (or safe). After releasing AntiPhish, we decided to implement a version of AntiPhish for the Microsoft Internet Explorer (IE) browser. Supporting IE was important because a majority of Internet users are using this browser. Our initial expectation at the beginning of the project was that porting a browser plug-in that is written for Mozilla Firefox to IE would be a straight-forward engineering exercise. After all, browser plug-ins are conceptually similar. However, creating an anti-phishing browser plug-in for the IE proved to be much more challenging than expected. In this paper, we describe our experience with implementing anti-phishing browser plug-ins for two different web browsers, and we present five lessons we learned during this process. As more and more

Third International Workshop on Software Engineering for Secure Systems (SESS'07)
0-7695-2952-6/07 $20.00 © 2007 IEEE

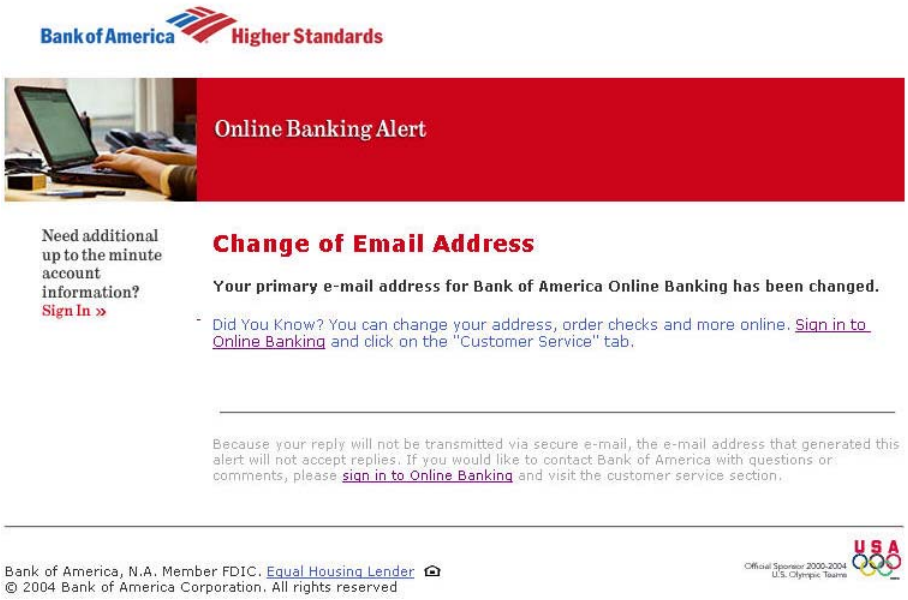

Figure 1. Phishing email targeting a U.S. bank.

browser plug-in-based security solutions are being created, we hope that our experience report will be useful for others, especially considering the fact that small mistakes can often lead to the complete compromise of a security solution.

The paper is structured as follows. The next section provides a brief overview of existing anti-phishing solutions. Section 3 presents techniques for writing Mozilla and Internet Explorer plug-ins. Section 4 discusses JavaScript attacks that a security plug-in needs to protect against. Section 5 describes our experience and the lessons-learned, while Section 6 concludes the paper.

2 Overview of Anti-Phishing Solutions

Several approaches have been proposed to protect users against phishing attacks. Some techniques attempt to prevent phishing mails from being delivered [11, 18], others blacklist malicious URLs [9], and yet others analyze web pages that a user visits [4]. All of solutions have advantages and disadvantages.

One of the main reasons why phishing attacks are possible is because mails can be spoofed easily. Clearly, it would be desirable to prevent malicious, spoofed mails from being delivered to users. Fortunately, it is possible to treat phishing mails as spam. However, although spam filters work quite well today, they cannot guarantee that all phishing mails are intercepted.

One anti-spam (and hence, anti-phishing) proposal is to authenticate the sender of an mail, thus preventing attackers from using hijacked mail addresses. Microsoft is implementing the so-called "Sender ID Framework" [11], and Yahoo is using it is own technique called "DomainKeys" [18]. Currently, Yahoo and other industry leaders are in the process of standardizing a technique called DKIM (DomainKeys Identified Mail), which is a direct descendant of DomainKeys [18, 14]. There were discussions on legal problems for open source software implementations due to patent issues. For example, Microsofts Royalty-Free Sender ID Patent License Agreement terms were a barrier to any Linux Debian package that wished to implement Sender ID or include Sender ID support [5]. However, Microsoft changed the licence terms of relevant patents to the "Microsoft Open Specification Promise," which is widely seen to be compatible to the GPL [12].

An approach that is analogous to the use of signatures in anti-virus products is to maintain a blacklist of sites that contain malicious content. Whenever the user tries to access a web site that has been blacklisted, the appropriate warnings are generated. Microsoft has chosen to use this approach (combined with some heuristics) in their phishing filter for the Internet Explorer 7 [9]. Verisign's approach to mitigate phishing attacks is to crawl millions of web pages to identify all web sites that closely imitate one of their customers' site. Once a web site is identified as a phishing site, the necessary steps are taken to shutdown the site as soon as possible. The problem with crawling and blacklisting proposals is that the anti-phishing organizations will find themselves in a race against attackers. This problem is analogous to the challenges faced by anti-virus and anti-spam companies. There is always a window of vulnerability

during which users are susceptible to attacks. Furthermore, blacklisting approaches are only as effective as the quality of the lists that are maintained.

Two academic browser-based client-side solutions have been proposed to mitigate phishing attacks [4, 15]. Both solutions are browser plug-ins that have been developed at Stanford university. PwdHash [15] is an Internet Explorer plug-in that transparently converts a user's password into a domain-specific password so that the user can safely reuse the same password on multiple web sites. Because the generated password is domain-specific, a password that is phished at the attacker's site cannot be used at the bank's site. Unfortunately, the proposed solution only works for passwords but it cannot protect sensitive information that is needed in unaltered form (e.g., credit card information or social security numbers). SpoofGuard [4] is a plug-in solution specifically developed to mitigate phishing attacks. The tool is symptom-based. That is, the plug-in looks for "phishing symptoms" such as similar sounding domain names and masked links in the web sites that are visited. Alerts are generated based on the number of symptoms that are detected.

Our anti-phishing solution, AntiPhish for the Mozilla Firefox browser, is similar to PwdHash and SpoofGuard in that it is a browser plug-in-based anti-phishing solution. One of its advantages is that it does not rely on a blacklist or any sort of symptom filter. Instead, we bind the sensitive information of a user (e.g., her password) to domain names. Then, we keep track of this sensitive information within the browser and generate warnings whenever it is typed into a form on a web site that is considered untrusted. Typically, a site is considered untrusted with respect to a piece of information when this information was not previously bound to the site. Because AntiPhish is user input-based, it can guarantee that sensitive information will not be transferred to a web site that is untrusted. For a more detailed description of AntiPhish, the reader is referred to [8].

3 Writing Browser Plug-ins

Different vendors are currently providing various mechanisms and APIs to enable third party software developers to integrate components into their browsers. Unfortunately, no standard exists, and the provided API functions that a plug-in can access differ between web browsers. In this section, we briefly discuss the implementation techniques for plug-ins for the two most popular browsers, Mozilla Firefox and the Internet Explorer.

3.1 Mozilla Firefox Extensions

The Mozilla Firefox browser offers a flexible and easy technique to write plug-ins. A so-called Firefox *extension* (i.e., plug-in) can access the resources within the browser and enhance the browser's functionality.

A Firefox extension is defined in an XPI (Cross-Platform Install) file that contains the plug-in source code and the corresponding installation information. XPI files are compatible with ZIP files. The installation information for the plug-in is defined in a file called "install.rdf" in the XPI archive.

The Firefox browser uses the Extension Manager (EM) to manage extensions. By looking at the "install.rdf" file in the XPI archive, it can obtain general information about the extension such as its name, the version of the browser that it is compatible with, and its unique identification number.

All Mozilla extensions are written in JavaScript. The JavaScript language [7] is widely used to enhance the client-side display of web pages. It was developed by Netscape as a light-weight scripting language with object-oriented capabilities and was later standardized by ECMA [1]. When an extension is executed, it is interpreted by the Mozilla JavaScript engine that is embedded within the browser. Because extensions are written in JavaScript and no native code is generated, all Mozilla extensions by definition are "open source". That is, it is possible to download any Mozilla extension and inspect its source code[1].

By writing an extension, the developer can react to and access events created by the browser. These events are typically created in response to user input or when a web page completes loading. For example, an extension can subscribe to key-press events to intercept all keys that are being pressed, or it can intercept all form data that is being submitted. Furthermore, the extension can also modify the contents of a page by accessing its Document Object Model (DOM) representation.

The Graphical User Interface (GUI) of an extension (if it has any) is created and defined using the XML User Interface Language (XUL). XUL is an eXtensible Markup Language (XML) dialect, which allows other XML standards such as XPath or XSLT to be used. The Gecko rendering engine interprets the XUL definition and displays the GUI within the browser.

A detailed overview of XUL can be found in [17], and the reader is referred to the Mozilla Developer Center web site [13] for detailed information about Mozilla extensions.

3.2 Internet Explorer Browser Helper Objects

Developing a Browser Helper Object (BHO) on Microsoft Windows is the most popular technique to integrate a plug-in component into the Internet Explorer (IE). Using

[1] It is not surprising that the Mozilla foundation has chosen this open approach to creating plug-ins when one considers that its browser is open source.

BHOs, the developer has access to the event mechanism of IE and can also create user interface elements such as toolbars.

Browser Helper Objects and toolbars are binary objects that conform to the Component Object Model (COM). COM is a binary standard developed by Microsoft to support, among other things, a component-based software market [16]. Every COM object implements a set of interfaces, each of which is a well-defined contract that describes what functionality the object provides. The COM standard guarantees that the virtual tables of interfaces remain the same across compilers, allowing COM objects to be implemented and used by any language that supports calling functions through a table of function pointers. The `IUnknown` interface must be implemented by all COM objects. It contains the function `QueryInterface`, which allows one to query for the other interfaces that an object might implement.

A Browser Helper Object is in essence a simple COM object that implements the `IObjectWithSite` interface. Toolbar objects work in a way similar to BHOs but implement a few more interfaces and include a graphical component.

Whenever the Internet Explorer is started, it queries the registry key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects` and loads all object whose Class Identifiers (so-called CLSIDs) are stored there. By using the *IObjectWithSite* interface that the BHO implements, the IE browser provides the BHO a pointer to its *IUnknown* interface. The BHO can store this pointer and subsequently use it to query the IE for more specific interfaces (for example, *IWebBrowser2*, *IDispatch*, or *IConnectionPointContainer* [10], which can then be used to access user-triggered events and browser functions). Figure 2 shows a diagram from the Microsoft Developer Network (MSDN) documentation that depicts the Internet Explorer loading and initializing helper objects [10].

One of the main differences of Internet Explorer BHOs and Mozilla Firefox extensions is that BHOs run as *native* code within the browser process. Hence, they have unlimited access to *all* resources of the operating system (such as network sockets, files, and processes) that the Internet Explorer can access. As a result, BHOs are also often used by spyware authors. Using a BHO, it is easy to intercept all user interaction and store it somewhere on the user's computer.

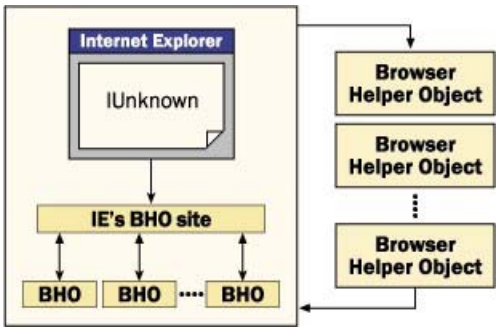

Figure 2. MSDN documentation that depicts the Internet Explorer loading and initializing helper objects.

4 JavaScript Attacks

A common and important task for most anti-phishing plug-ins is to check user input for the presence of sensitive information. The idea is to ensure that such information is only provided to trusted sites but never submitted to untrusted servers. A plug-in can easily carry out this task when the malicious phishing server only sends static HTML pages. In this case, sensitive information is only leaked when the user transmits her data to the attacker's site, for example, by submitting a web form. Clearly, the plug-in has the opportunity to intercept and cancel this operation before the form data is actually sent to the attacker.

Unfortunately, the situation gets more complicated when the attacker embeds malicious JavaScript into his phishing page. JavaScript is a powerful language that provides the attacker with a wide range of possibilities for bypassing monitoring plug-ins such as AntiPhish or SpoofGuard. For example, one approach is to use JavaScript hooks to listen for key-press events. Instead of waiting for the user to press a submit button to send the information, the attacker could intercept each key that is pressed and send the information character by character back to his server. Thus, before the plug-in can detect that sensitive information is entered, most of the data is already transmitted to the attacker.

Listing 1 shows JavaScript code that implements the aforementioned attack. In Lines 18 and 23, the attacker has defined a function *keysEvent()* that is invoked every time a "key up" event is generated (i.e., a pressed key is released). In the example code, a dialog box is displayed (Line 10) that contains the values of the user name and password fields. In a real attack, the attacker would send this information to his server. Of course, JavaScript does not allow network connections to be opened. Note that the newer Ajax technology allows connections to be opened using XMLHttpRequest objects, but with the constraint that the "domain of the URL request destination must be the same as the one that serves up the page containing the script [3]." However, the attacker can easily leak out the information by modifying the URL of a (hidden) image embedded in the page so that the web browser requests this image from a web server under the attacker's control. Of course, arbitrary informa-

tion can be embedded into this URL. For example, if the key "z" is pressed, the URL of an embedded image could be set to *http://evil.example/log?key=z*.

Instead of hooking key strokes, the attacker could also leverage the JavaScript *setTimeout()* method. Using this method as a trigger, the attacker can periodically capture the current values of all form fields on the page.

```
<html>
<head>
<title>JavaScript Attack</title>
</head>

<body>
<script type="text/javascript">
function keysEvent()
{
  alert("username: " +
    document.testform.username.value
    + "\npassword: " +
    document.testform.pwd.value);
}
</script>
<form name="testform">
  Username:
  <input type="text" name="username"
    onKeyUp="keysEvent()">
  <br>
  Password:
  <input type="password" name="pwd"
    onKeyUp="keysEvent()">
</form>
</body>
</html>
```

Listing 1. Intercepting key strokes using JavaScript.

Obviously, the easiest solution to mitigate JavaScript attacks is to deactivate JavaScript on pages that contain forms. This approach, unfortunately, is not feasible because many web sites make use of JavaScript to submit forms. Furthermore, JavaScript is also often used for client-side input validation purposes.

The solution we used in the Mozilla Firefox version of AntiPhish was to deactivate JavaScript every time the focus is put on a form element (e.g., an input field), and to reactivate it whenever the focus is lost. Using this technique, we can ensure that an attacker cannot capture key strokes or launch timing attacks before AntiPhish can examine any input for sensitive data.

5 Lessons Learned

After finishing a prototype of AntiPhish for Mozilla Firefox in 2005, we started to implement an Internet Explorer (IE) version. We expected to have an IE prototype in a short period of time. As discussed in Section 3, although plug-ins for various browser platforms are implemented slightly different, they are conceptually similar. If a developer is familiar with browser events (e.g., page not found event, document loaded event, etc.), porting a plug-in from one browser to another should not be difficult. Unfortunately, in real life, creating an anti-phishing browser plug-in for the IE proved to be much more difficult than expected.

In this section, we describe five lessons we learned during the development of AntiPhishIE, the AntiPhish solution for the Internet Explorer. We discuss the problems we encountered and the workarounds we developed.

Lesson 1: JavaScript cannot be deactivated temporarily on IE

As discussed in Section 4, JavaScript attacks pose a serious threat to anti-phishing plug-ins. On Mozilla Firefox, we were able to temporarily disable the JavaScript engine by appropriately setting a boolean flag in its configuration. Every time the boolean value was modified, the changes became effective immediately. The idea is to intercept user interface focus events and to disable the JavaScript engine each time the focus is put on a form element. In this way, we disable the JavaScript whenever a user is typing data into a form, making it impossible for an attacker to steal data while it is entered. Note that this approach also effectively mitigates timer-based Javascript attacks with which the attacker can steal the data entered into a form. In this attack, the attacker might use a Javascript timeout function that is automatically invoked after a specified amount of time has elapsed. Because Javascript is deactivated while the user is entering information into the form and because AntiPhish checks if sensitive information is being typed into an untrusted domain after each key press, such a timer-based attack cannot succeed.

We expected the IE to work the same way. Unfortunately, we discovered that this was not the case. Instead, it is only possible to turn off JavaScript on a system-wide basis. To achieve this, the registry settings for IE have to be modified, and the new settings become active only after restarting the browser. Because we were not able to temporarily disable the JavaScript engine of IE, we had to implement a workaround. The idea was to intercept all keyboard events before they reach the JavaScript engine. In other words, we aimed to disable the delivery of events to the JavaScript engine while the input focus was on a form element. In this fashion, no JavaScript code is triggered while the user fills

out a web form. The implementation is not trivial and uses Win32 API functions to implement keyboard hooks. More precisely, the plug-in uses the *SetWindowsHookEx()*, *UnhookWindowsHookEx()* and *CallNextHookEx()* functions to insert itself between the Windows messaging system and the JavaScript engine. Whenever the user is typing input, no events are delivered to the JavaScript engine. While this could interfere with legitimate JavaScript code, it is necessary to prevent the attacks mentioned in the previous section.

Unfortunately, although intercepting key events at the operating system level mitigates JavaScript key logging attacks, timer-based attacks are still possible. We did not manage to find an elegant solution around this problem. In our current prototype, AntiPhishIE inspects the JavaScript code in the HTML page and displays an alert whenever it detects that the JavaScript *setTimeout()* command is being used. In this case, a dialog is displayed to the user that informs of possible security issues. This could lead to false alarms because there might be sites that legitimately use the JavaScript *setTimeout()* function.

Lesson 2: Using .NET for BHOs has disadvantages

Microsoft recently introduced the .NET framework, a new development and runtime environment where machine-independent byte-code is executed by a virtual machine. Multiple programming languages can be used with the .NET framework (e.g., VB.NET, C#, C++, etc.). The idea is that programs in any of these languages are compiled into a Common Intermediate Language (CIL), which is hardware-neutral byte-code. The bottom layer of the architecture consists of the Common Language Runtime (CLR). The CLR is a virtual machine that interprets and executes the .NET byte-code. Programs that are written in the common intermediate language are referred to as *managed code*.

Using managed code when developing an application has many advantages. Managed code runs entirely inside a sandbox. Hence, it cannot make calls outside of the .NET framework and this permits applications built with managed code to run safer. We implemented AntiPhishIE as a managed .NET application, assuming that using .NET would allow us to build the prototype faster and easier.

Unfortunately, the downside of using managed code was that fact that it was significantly more difficult to use low-level Windows functions. In particular, it was difficult to invoke the Windows API functions that implement the interception of keyboard events (as discussed in the previous lesson). One problem was that these API functions were not declared in the .NET *System.Runtime.InteropServices* namespace and had to be redefined as managed C# interfaces. This task is tedious and error-prone, as it requires the programmer to provide a mapping between native Win32 data types and .NET types. We learned that implementing BHOs in .NET is not always as straightforward as one thinks, especially when native Windows support is required.

Lesson 3: BHOs provide more flexibility than Mozilla extensions, but this comes at a price

One advantage of BHOs in comparison to Mozilla extensions is that native Windows code may be used within a BHO. A BHO, in essence, is an "independent" Windows thread that is running in the same address space as the Internet Explorer. Hence, the programmer can create processes, files, and network connections or invoke existing legacy code. While this flexibility is certainly an advantage, the downside is that more care needs to be taken when writing (or integrating) native code. In particular, when using native code (e.g., C or C++) in a BHO, one needs to ensure robust input validation and memory management. In comparison, when writing Mozilla extensions, such considerations are not necessary because JavaScript is an interpreted language[2]. Thus, according to our experience, coding Mozilla extensions is easier in the sense that it is less likely to make programming mistakes that crash the browser.

Lesson 4: Mozilla extensions are more tedious to maintain

Once a BHO is created, one can be quite certain that it will work as expected also in future releases of the Internet Explorer. Mozilla extensions, in comparison, need to define the versions of Mozilla browsers that the extension supports. The idea of this feature is to prevent extensions from not working properly when newer versions of Mozilla are released. If the extension detects that the current version of Mozilla is not supported, a warning is displayed and the extension is deactivated. Thus, we have to release a new installation (i.e., XPI) package every time a new Mozilla version is released. This overhead can be reduced by defining a yet non-existing, future version of Firefox that the extension claims supports. Of course, the downside of this approach is that the extension might not work properly when installed because of modifications in the browser APIs.

Lesson 5: IE passes control to BHOs after scripts have been invoked

Both BHOs and Mozilla extensions have similar event notification mechanisms when a web page has been loaded. However, one interesting difference that we noticed was

[2]Note that the JavaScript engine itself may have a buffer overflow problem. However, such errors are less common.

that Firefox delivers control to an extension *before* the JavaScript on the page is executed, whereas the IE delivers control to the BHO *after* the JavaScript is executed.

This behavior might be problematic for security plug-ins that need to control and intercept the execution of the JavaScript code in a page. One possible remedy is to deactivate JavaScript globally as soon as the BHO is started. This approach, however, may have adverse effects on some web pages, which may not behave as expected once the JavaScript is restarted.

6 Conclusion

Phishing is an online identity theft that aims to steal sensitive information such as user names, passwords, and credit card numbers. Although a simple attack, phishing has become a large problem for organizations that are doing online business. The number of phishing scams are continuously growing, and the cost of the resulting damage is increasing. Researchers as well as the IT industry have identified the urgent need for anti-phishing solutions and recently, a number of solutions to mitigate phishing attacks have been proposed. Some of these anti-phishing solutions are browser-based and have been implemented as plug-ins.

In many scientific papers, the authors present a small prototype and claim that their tool can be extended or ported easily to run on any platform that is actually used in the real-world. In fact, often, such a task is only considered an engineering exercise. In 2005, we developed an anti-phishing browser plug-in for Mozilla Firefox. Before porting this plug-in to the Internet Explorer, we expected such an engineering exercise. Unfortunately, the task proved to be more difficult than expected. In this paper, we described our experiences when implementing security browser plug-ins for Mozilla and Internet Explorer. We hope that the difficulties we faced, the lessons we learned, and the experiences we reported in this paper will be useful for other developers who are in the same situation.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grants P-18764, P-18157, and P-18368.

References

[1] ECMA-262, ECMAScript language specification, 1999.

[2] Anti-Phishing Working Group. Phishing Activity Trends Report. `http://www.antiphishing.org/`, 2005.

[3] Apple Inc. Dynamic HTML and XML: The XMLHttpRequest Object. `http://developer.apple.com/internet/webcontent/xmlhttpreq.html`, 2007.

[4] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. Mitchell. Client-side defense against web-based identity theft. In *11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[5] Debian. DEPLOY: Debian project unable to deploy Sender ID. `http://www.debian.org`, 2006.

[6] Engin Kirda, Christopher Kruegel. Protecting Users Against Phishing Attacks with AntiPhish. `http://www.seclab.tuwien.ac.at/software/antiphish/`, 2006.

[7] D. Flanagan. *JavaScript: The Definitive Guide*. December 2001. 4th Edition.

[8] E. Kirda and C. Kruegel. Protecting Users against Phishing Attacks. *The Computer Journal, Oxford University Press*, 2006.

[9] Microsoft. Anti-Phishing Technologies. `http://www.microsoft.com`, 2005.

[10] Microsoft. Browser Helper Objects: The Browser the Way You Want It. `http://msdn.microsoft.com`, 2005.

[11] Microsoft. Sender ID Framework Overview. `http://www.microsoft.com`, 2005.

[12] Microsoft. Microsoft Open Specification Promise. `http://www.microsoft.com/interop/osp/default.mspx`, 2007.

[13] Mozilla Foundation. Mozilla Developer Center: Extensions. `http://developer.mozilla.org`, 2006.

[14] Mutual Internet Practices Association. DomainKeys Identified Mail (DKIM). `http://www.dkim.org/`, 2007.

[15] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. Stronger Password Authentication Using Browser Extensions. In *14th Usenix Security Symposium*, 2005.

[16] S. Willliams and C. Kindel. The Component Object Model: A Technical Overview. Microsoft Technical Report, October 1994.

[17] XULPlanet. XULPlanet. `http://www.xulplanet.com`, 2005.

[18] Yahoo. Yahoo! Anti-Spam Resource Center. `http://antispam.yahoo.com`, 2006.