PROTECTING FINANCIAL INSTITUTIONS FROM MAN-IN-THE-BROWSER ATTACKS

Xinran Wang & Yao Zhao
Shape Security Inc., USA

Email {xinran, yzhao}@shapesecurity.com

ABSTRACT

Banking malware is one of the most serious threats to both end-users and financial institutions. It is reported that over 1,400 financial institutions have been targeted by attackers using banking trojans and the top 15 targeted financial institutions were attacked by more than 50 per cent of the trojans in 2013. One major tactic of banking malware is the use of man-in-the-browser attacks (web injection attacks). In fact, almost all modern banking malware uses this tactic. In this paper, we first explain how banking malware conducts credential stealing and automatic transactions with man-in-the-browser attacks, and we analyse several web injection scripts from prevalent banking malware families. Then we present our survey of existing techniques against these malware families, as well as their limitations. Next, inspired by the observation that banking malware's web injection is based on a certain context of the target web pages, we propose HoneyWeb, an application layer system to protect financial institutions from web injection attacks.

The HoneyWeb system works as an HTTP reverse proxy in front of protected web servers, and injects fake context into the target page, according to the malware's web injection configuration. The fake context traps the banking malware's web injection scripts in an invisible HTTP element. An alert is also triggered when injection happens, so the system detects the ongoing attacks. More importantly, it prevents credential stealing as the web injection scripts are injected into invisible decoy elements.

1. INTRODUCTION

Banking malware is one of the most serious threats to both end-users and financial institutions. It is reported that over 1,400 financial institutions have been targeted by attackers using banking trojans and that the top 15 targeted financial institutions were targeted by more than 50 per cent of the trojans in 2013 [1].

Man-in-the-browser (MitB) attacks are one of the main techniques used by prevalent banking malware such as Zeus, Gameover and SpyEye. A classic goal of a MitB attack is stealing credentials – not only usernames and passwords, but also other sensitive personal information such as social security numbers and PIN numbers. Generally, banking malware uses web injection techniques to get bank customers to type in their sensitive personal information when they are browsing legitimate web pages. This kind of attack is much more powerful than phishing. Recently, criminals have taken a further step to use Automatic Transaction Systems [2] to automatically and stealthily make transactions to steal money from bank customers. The sophisticated attacks even hide the real balance of the bank accounts, so that the victim doesn't know the attack is happening.

Existing solutions to mitigate man-in-the-browser attacks fall into two categories: detection and prevention. Web tripwire [3] and Zarathustra [4] detect if any unexpected content appears in the HTML text or the DOM (Document Object Model) of the browser. But one disadvantage of this type of approach is that the adversaries can upgrade their MitB tools to not only inject content, but also remove or disable detection scripts. Web page obfuscation [5] and polymorphism [6] can be used to stop the malicious content injection, or stop the automatic transactions.

In this paper, we propose HoneyWeb, which is a combination of both detection and prevention ideas. HoneyWeb uses existing obfuscation and polymorphism techniques to prevent web injection attacks. At the same time, using the philosophy of Honeynet, HoneyWeb itself injects fake content (called the honey object) into web pages in order to trap malicious web injection.

HoneyWeb has the ability to detect the compromise of a customer's machine with an extremely low false positive rate. This detection allows banks to notify victims and advise them to clean up the malware, change their credentials, etc.

This paper is organized as follows: we provide some background information in Section 2, and survey related work in Section 3. In Section 4 we describe the details of the HoneyWeb system. Then we discuss our future work and conclude the paper in Section 5.

2. BACKGROUND

2.1 From keylogging to form grabbing

Keyogging is a common method for banking malware to steal credentials. Keyloggers capture every key typed into a system. But key log data can be messy and the technique misses any data the user inputs without using the keyboard. For example, keyloggers may miss sensitive data that a user copies and pastes into a form or selects via an options dropdown provided by autocomplete.

Some banks use a virtual keyboard for the password entry, which does not trigger keystrokes either. To overcome this, banking malware such as SpyEye and Zeus record screenshots at regular intervals or upon each mouse click in order to defeat the virtual keyboard.

Form grabbing retrieves authorization and login credentials from a web data form by intercepting the HTTP POST data before the data passes through encryption routines [7]. This method is more effective than keylogger software because it acquires the user's credentials even if they are inputted using a virtual keyboard, autofill, or copy and paste. Form grabbing provides much cleaner, better structured data based on its variable names, such as username and password.

SpyEye implements form grabbing by hooking HttpSendRequestA and HttpSendRequestW to intercept content-bearing HTTP requests (usually POST requests) made by *Internet Explorer*-based browsers [8].

2.2 Web injection

Keylogging and form grabbing are passive ways to steal credentials, while a man-in-the-browser attack (also known as web injection) is a proactive way to steal credentials. For example, MitB can steal additional credentials which may not be requested by banks, such social security number (SSN) and PIN. MitB is a technique in which malware hooks into the browser and manipulates data before it is displayed. A simple MitB attack scenario is described as follows: a user attempts to log into a banking website. Banking malware intercepts the request, then injects a form or extra fields such as SSN or PIN into the response. The victim unknowingly submits the sensitive information to the attacker. As a MitB attack happens at the presentation layer, there are no obvious indications of malicious activity. The domain is legitimate and the security certificate has not been tampered with, which all adds credibility to attacker's requests and can end up fooling the user.

Web injection for both SpyEye and Zeus is implemented as a WebInject configuration file. A WebInject file is a text file which contains JavaScript and HTML code. The file allows the banking malware to target financial institutions and inject specific code into victims' browsers so they can modify the web pages the users access in real time. Banking malware equipped with a WebInject file can easily make deceptive forms or fields that ask victims for specific credentials (e.g. SSN and PINs).

Figure 1 shows an example of WebInject configuration. The 'set_url' parameter sets the attack target; the 'data_before' parameter describes the bank web data to search for before the injection; the 'data_inject' parameter is the actual script that will be injected. The example in Figure 1 shows that the code snippet will be injected into any URL that contains 'https://www.bankofexample.com/login.html', that it will be injected after the data in 'data_before', and the code itself takes the form of additional fields in the form requesting 'SSN'.

```
set_url http://www.bankofexample.com/login.html GP
data_before
name="password"*</tr>
data_end
data_inject
<tr><td>SSN:</td><td><input type="text" name="ssn" id="ssn" /></td></tr>
data_end
data_after
data_end
```

Figure 1: A simple web injection configuration.

2.4 Automatic Transaction System

Unlike traditional WebInject files that inject extra forms or fields to steal victims' credentials, a sophisticated web injection called ATS (Automatic Transaction System) can automatically execute transactions in the background [2]. It checks account balances and performs wire transfers using the victim's credentials without alerting them. ATS is invisible. ATS also changes account balances and hides illegitimate transactions. As long as a system remains infected with an ATS, its user will not be able to see the illegitimate transactions made from his accounts.

This essentially makes online banking fraud automatic, because cybercriminals no longer need user intervention to obtain money.

Figure 2 shows an example of code injected into a WebInject file. It calls a remote file that contains the JavaScript or HTML code that will perform the injection. Figure 3 shows the actual JavaScript code that performs the wire transfer.

```
set_url *bankofexample*main.html* GP
data_before
<body>
data_end
data_inject
<script>
document.write('<scr'+'ipt type="text/javascript" src="http://botnet-cnc-
server/ATS/plugin.js"></sc'+'ri'+'pt><iframe id="gozilla" onload="gozilla();" src="
/transfer" scrolling="no" frameborder="0" style="border:none; overflow:hidden; width:
0px; height:0px;" allowTransparency="true"></iframe>');
</script>
data_end
data_after
data_end
```

Figure 2: ATS web injection configuration.

```
function gozilla() {
    // ...
     var gozi = frames[0].document.getElementById("TransferForm");
     gozi.ctl03$InternalOrExternalPayment[0].checked = false;
     gozi.ctl03$InternalOrExternalPayment[1].checked = true;
     gozi.ctl03_txtExternalPaymentAccount.value = destination_account;
     gozi.ctl03_txtAmt.value = AZ_chunks;
     gozi.ctl03_btnTransfer.click();
     // ...
}
```

Figure 3: An ATS JavaScript performing wire transfers.

3. RELATED WORK

There are several other research projects that are closely related to our work.

3.1 Web page inspection

Reis *et al.* proposed 'web tripwire' [9]. A web tripwire uses JavaScript code to detect textual changes in an HTTP web page, with the ability to report any changes both to the user and to the publisher. This JavaScript code runs in the user's browser and compares the page the user receives what it is expected to be. This technique has been suggested as a countermeasure [3] to detect banking malware's web injection. However, web tripwire is not secure: adversaries could remove the web tripwire if they wish to avoid detection.

3.2 Web injection fingerprint extraction

Bosatelli proposed 'Zarathustra', an automated system that detects the activity of banking trojans that perform web injection on the client side [4]. Zarathustra extracts the DOM differences by first rendering a banking website's page multiple times in an instrumented browser running on distinct and clean virtual machines. This builds a model of legitimate differences (e.g. due to ads, A/B testing, cookies, load balancing, anti-caching mechanisms, etc.). Zarathustra repeats the same

procedure on an infected machine and extracts and generalizes the differences called 'fingerprints'. The fingerprints are generated on dedicated machines, which operate offline, without any interaction with real clients. The system has the advantage of requiring no reverse engineering effort: the only requirement is a binary sample of the malware to infect the controlled machine, which is used to identify differences in web pages generated by the malware's web injection techniques.

3.3 Web page obfuscation

As shown in Figures 1 and 2, Zeus and SpyEye use web injection configuration files to perform web injection. The location of injected code (context) is described in the 'data_before' or 'data_after' parameters of configuration files. Mador *et al.* [5] proposed a method to obfuscate the context and thus prevent banking malware's web injection. They encrypt the web page content in JavaScript and only decrypt when the web page is loaded in a browser. The obfuscation method was originally used in exploit kits by cybercrimals to avoid detection. It is now used to confuse banking malware and prevent web injection.

Once banking malware is aware of the obfuscation, it can perform deobfuscation. However, security researchers have responded by making the variable name of the decrypted function polymorphic so that banking malware cannot detect the obfuscation.

4. HONEYWEB SYSTEM

In this section, we describe the details of the HoneyWeb system, which combines prevention and detection of MitB attacks.

4.1 Overview of HoneyWeb

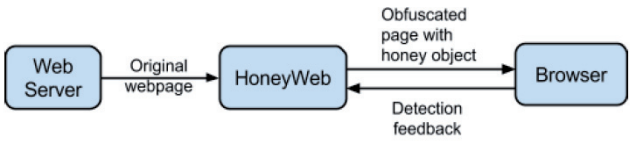

Figure 4: HoneyWeb deployment.

HoneyWeb works as a reverse proxy, which is transparent to both web server and browsers (as well as the end-users behind the browsers). Figure 5 shows the overall function of the HoneyWeb system. We define the honey object as some HTML, CSS or JavaScript code that is injected by HoneyWeb. An important feature of the honey object is that it is 'invisible' to the human eye when the browser renders the web page.

When a user visits a protected web page, HoneyWeb takes the original content of the requested URL and rewrites it with three basic changes (see Figure 5 as an example):

1. It obfuscates the original content so that banking malware fails to inject its malicious content into the original target. In the example shown in Figure 5, the target form is obfuscated, and for example, we can use techniques introduced in [5] to change HTML clear text to JavaScript code.

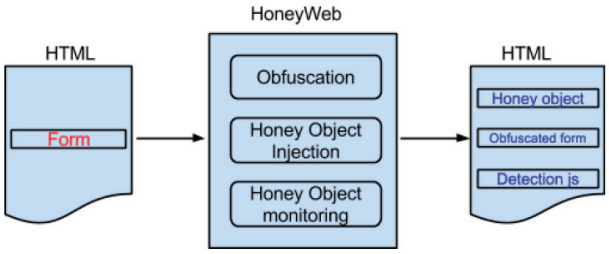

Figure 5: HoneyWeb modules.

2. It adds a honey object to the web page, so that banking malware will match the fake content and inject their malicious content there. Note that the honey object will be invisible to the user when the web page is rendered by a real browser.
3. It adds a piece of JavaScript code to monitor the honey object. If anything malicious is injected into the honey object, it reports a detection result back to the HoneyWeb system.

4.2 Details of HoneyWeb

In this section, we describe the details of the three modules of the HoneyWeb system.

4.2.1 Obfuscation module

The obfuscation of HTML and JavaScript has been well studied in literature, e.g. [5]. HoneyWeb can use any existing obfuscation technique to prevent the target code (e.g. form) being found by banking malware (e.g. using a regular expression). As a reverse proxy, the procedure can be summarized as three steps:

1. Given the web page URL, HoneyWeb loads the obfuscation configuration, which might be as simple as a regular expression.
2. HoneyWeb matches the content in the web page using the obfuscation configuration.
3. The matched content is replaced with a piece of JavaScript code that generates the same content.

4.2.2 Honey object injection

As mentioned previously, the honey object is the fake content that is injected by HoneyWeb to be matched by the banking malware's injection rules such as exact matching or regular expressions. It seems to be quite simple to inject honey content that will satisfy the requirements, however, in practice there are a couple of problems to overcome.

Invisibility

The honey object must be invisible to a real human user. To achieve this, HoneyWeb places the fake content inside a div or iframe that is invisible, by setting the proper CSS style (e.g. display:None).

No interference

Injecting new content into a web page may interfere with the existing content, especially JavaScript.

For example, assume the banking malware looks for the string '<input name='password' id='password' type='password'>' and inserts a line to request a PIN number below. A simple honey object may look like the code shown in Figure 6.

```
<div style="display:none">
<input name='password' id='password' type='password'>
</div>
```

Figure 6: Honey object example.

However, the honey object code in Figure 6 introduces an element with id name 'password', which also appears in the obfuscated code. This means there will be two inputs with the same ID 'password' in the DOM. As indicated in the HTML specification, it leads to undefined behaviour when JavaScript calls document.getElementById("password"). In general, we'd like to avoid such interference being introduced by honey content.

HoneyWeb has a couple of solutions to deal with different situations. Here we list some of them:

- Put the honey object into HTML comments. This way, the honey object can still be matched and located by banking malware, but the honey object means nothing to the DOM.
- Use JavaScript to avoid duplicated IDs. For example, we inject the honey object before the corresponding obfuscated code. Then we can use JavaScript to locate the elements in the honey object, and change the ID dynamically. For example, the code below avoids duplication of IDs via a line of JavaScript.

```
<div id="honeydiv" style="display:none">
<input name='password' id='password' type='password'>
</div>
<script type="text/javascript">document.getElementById
('password').id="xxxx"</script>
```

Figure 7: Example of changing the element ID at runtime.

4.2.3 Honey object monitoring

The purpose of the honey object is to passively detect when bank customers' computers are compromised. The monitoring component detects the compromise and sends this information to the bank.

HoneyWeb injects a piece of JavaScript at random location in the original web page. The JavaScript code does the following work on the browser side:

- It schedules the malware detection code to run when the whole page is loaded and every few seconds periodically.
- The detection code reads the static content of the honey object (i.e. via object.innerHTML) and DOM elements of the honey object.
- If any injection into the honey object is detected, a synchronized data transfer (i.e. Ajax POSTs) is used to send an alert. Meanwhile, the code may also alert the end-user about the compromise by popping up a message.

HoneyWeb collects both compromise alerts, as well as the login information (e.g. username and password), if possible. Next, HoneyWeb may send all the compromise information to the bank. The bank can inform its customers according to the login information via other communication methods such as email and phone calls.

5. CONCLUSION AND FUTURE WORK

In this paper, we describe HoneyWeb, a system that prevents web injection attacks by banking malware, while also retaining the ability to detect the compromise of a machine by the malware.

Similar to HoneyNet that attracts malicious traffic, HoneyWeb uses invisible fake contents to attract malicious injected web content, and then detects the injection with very few or no false positives. This advantage allows banks to cooperate with their customers to remove the future lost cost by the compromise.

Currently, HoneyWeb relies on known malware signatures to determine which part of web content to be obfuscated, and then to inject invisible trapping contents. An improved system may combine automatic signature extraction systems such as Zarathustra [10] and HoneyWeb. This fully automated system can extract malware signature first, update obfuscation and honey object injection module automatically, and then finally alert on the compromise.

REFERENCES

[1] Doherty, S.; Krysiuk, P.; Wueest, C. The State of Financial Trojans 2013, Security Response White Papers, Symantec.

[2] Kharouni, L. Automating Online Banking Fraud. Automatic Transfer System: The Latest Cybercrime Toolkit Feature. http://www.trendmicro.com/cloudcontent/us/pdfs/securityintelligence/whitepapers/wp_automating_online_banking_fraud.pdf.

[3] Barnett, R.; Grossman, J. Web Application Defender's Cookbook: Battling Hackers and Protecting Users.

[4] Bosatelli, F.; Zarathustra: Detecting Banking Trojans via Automatic, Platformindependent WebInjects Extraction, https://www.politesi.polimi.it/handle/10589/78343, 2013.

[5] Mador, Z.; Barnett, R. An Arms Race: Using Banking Trojan and Exploit Kit Tactics for Defense, RSA Conference, 2014.

[6] Wang, X.; Kohno, T.; Blakley, B. Polymorphism as a Defense for Automated Attack of Websites, Applied Cryptography and Network Security Lecture Notes in Computer Science, 2014.

[7] Capturing Online Passwords and Antivirus. Web log post. Business Information Technology Services, 24 July 2013.

[8] IOActive, Inc. Reversal and Analysis of Zeus and SpyEye Banking Trojans. http://www.ioactive.com/pdfs/ZeusSpyEyeBankingTrojanAnalysis.pdf.

[9] Reis, C.; Gribble, S.; Kohno, Y.; Weaver, N. Detecting InFlight Page Changes with Web Tripwires, NSDI, 2008.

[10] Criscione, C.; Bosatelli, F.; Zanero, S,; Maggi, F. Zarathustra: Extracting WebInject Signatures from Banking Trojans, 20th Annual International Conference on Privacy, Security and Trust, 2014.