Attacking of SmartCard-based Banking Applications with JavaScript-based Rootkits (short paper)

Daniel Bußmeyer, Felix Gröbert, Jörg Schwenk, and Christoph Wegener

Horst Görtz Institute for IT Security
Chair for Network and Data Security
Ruhr-University Bochum
{daniel.bussmeyer,felix.groebert,joerg.schwenk,christoph.wegener}@rub.de

Abstract. Due to recent attacks on online banking systems and consequent soaring losses through fraud, different methods have been developed to ensure a secure connection between a bank and its customers. One method is the inclusion of smart card readers into these schemes, which come along with different benefits, e.g., convenience and costs, and endangerments, especially on the security side.
We give a review on a security concept and its implementation deployed as an online banking solution, which consists of a USB smart card reader and a customized browser. We propose a thread model and an attack vector exploiting the limited capabilities of the class one smart card reader. Furthermore a proof of concept malware is presented, which utilizes the primary vulnerability, i.e., class one reader, and otherwise supporting vulnerabilities, to show how transactions may be manipulated.

1 Introduction

Recent developments have shown that the endusers' environment must be considered as vulnerable and insecure. In order to provide secure banking, a lot of precautionary measures have been developed. Most aim to establish a secure communication channel, which supplies confidentiality, integrity and authenticity for transactions made between an enduser and the bank. A common method is the deployment of a class one, two, three or four smart card reader and supporting software at the enduser.

Invoking smart card readers from online banking software offers the possibility to encrypt and sign messages independent of the endusers' computer system. Thereby several differences between the smart card reader classes exist, which mainly vary in their equipped hardware:

Class 1. reader without pin pad
Class 2. reader with pin pad
Class 3. reader with pin pad and display
Class 4. reader with class 3 properties, RSA functionality and a bytecode VM

A class one reader thus only guarantees the lowest security level: The cryptographic primitives, e.g., signatures or encryption, are still executed on the host system which might be infected by malware. Besides the possibility to use a physical device for signing, the class one reader does not increase the attack complexity for a malware author, who infected an endusers machine.

Smart card readers of class two and higher, in contrast, offer a pin pad onboard. The pin code for the smart card entered into the reader does not leave the reader and thus is not readable by malware on the host system.

Furthermore class three readers are equipped with a display that is able to show the data to be signed or encrypted. This gives extra security to the signature data, because the user can actually see what he or she is about to sign. This follows the important principle *only sign what you see*.

Preceding security features can only be exceeded by the highest category of smart card reader classes, security class four. These readers also support RSA functionality and come with a virtual machine making it possible to run small software components. It is a clear advantage for the security of software to run in a virtual machine that cannot be accessed by the host system. Intercepting messages or manipulating procedures running in the virtual machine are very hard to conceal and expend a lot of effort to stay undetected.

In this paper we take a closer look on a deployed solution for internet banking on the basis of a USB device containing the banking software and a smart card reader class one. As shown before, it is difficult for malware to intercept and modify messages undetected, if an online banking system includes smart card readers. But if we consider the *man in the box* scenario, in which the host system is infected by a malware, a smart card reader satisfying only class one standards is not sufficient. As already mentioned before, readers of security class one do not offer the possibility of entering the pin on the device itself. The pin is always processed within the host system. Hence malware does not need to launch attacks on the reader's hardware since all important data is located in the machine's memory it is plugged into.
We discuss the prerequisites in Section 2, propose the attack vector and threat model for the scenario in Section 3.1 and present a proof of concept attack on the implementation in Section 3.2. We conclude in Section 6 on the basis of the related work in Section 4 and mitigation potentialities in Section 5.

2 Prerequisites

In this section we describe the audited solution and its intended usage. Furthermore we describe and discuss a typical usage environment for the application.

2.1 Target Solution

The targeted solution consists of a USB smart card reader including a USB flash drive. The flash drive comes with a customized Firefox web browser, which is run once the device has been inserted into a host system. The solution is used by a

large German, a Turkish, and a Swiss financial institution for internet banking. This paper focuses on the German version with a release date in spring 2009. When the USB device is inserted into a Windows-based machine the flash drive and the smart card reader are recognized by the operating system and (if enabled) autorun launches the application from the flash drive.

At first, the launcher application is started, which is responsible for the creation of other processes and temporary execution directories under `%TEMP%` in which the processes are executed. A dedicated application and a batch job ensures that if the enduser unplugs the device no temporary files, e.g., cookies or caches, of the internet banking session are left and then terminates the main application.

Next, the launcher application starts an update client which communicates with a server to download updates for the flash drive. The update mechanism might be an attack vector itself, for example by redirecting network traffic and emulating the update server to introduce malicious, infected software on the flash drive. We did not further analyze this vector, because the updates were signed and infecting the web browser is a more direct approach when assuming a *man in the box* threat model.

Third, the web browser is launched. The customized Firefox is the main application for the enduser and when it terminates, the launcher application conducts the removal of temporary files and then finally removes the flash drive.

The browser is based on Firefox 1.5.0.9 and is customized to interact with the USB smart card class one reader and includes a Java plugin. It also contains some code integrity checks:

- Every two seconds a thread checks whether the addresses of `SSL_*()`, `PR_*()` and other essential functions have changed.
- Every 42 seconds a thread checks if `isDebuggerPresent()` returns `true` and if any other Firefox extensions are installed.

Besides the primary, conceptual vulnerability of using a class one reader, we found several secondary vulnerabilities concerning the concrete implementation:

- Timing is done using `Sleep()` and `time()`. By hooking theses functions an adversary is able to disable the integrity checks of the application. In general, the reverse engineering process is not very complex as no code obfuscation is done. The code even contains debug strings and `OutputDebugString()` gives numerous error messages. A code protector, e.g., Themida, might increase the effort to analyze the application.
- Due to the class one reader a keylogger is capable of reading the smart card PIN using `GetAsyncKeyState()`. No countermeasures exist to prohibit this simple attack.
- The used Firefox version (1.5.0.9) and Java version are obsolete and may be attacked from the network using exploits already publicly known. This way the operating system can be compromised.
- The domain restriction for the user may be circumvented via the menu *Extras* → *Themes* → *Download Themes*. Originally the browser is restricted to the SSL server of the bank.

- The SSL implementation is flawed, so the Null-Byte-X.509-Attack [4] is possible.
- The certificate authorities are obsolete, so the MD5-Signing-Attack [5] is possible.
- No Certificate-Revocation-Lists are included.

2.2 Target Environment

A normal usage environment presumes a login, a password, and a PIN for the smart card. As the primary customers are corporate users, a main advantage of using a class one reader is the possibility of mass signing of transactions. Rather than signing each transaction separately by checking the display and entering a pin on a class three reader, it is possible to sign several hundreds of transactions with a small user effort. In our correspondence with one affected bank this feature outweighed the security disadvantages. Nevertheless we think this is a design decision with too much drawbacks on the security properties of the solution, which we explain in detail in the next section.

3 Proof of Concept Attack

In this section we determine the attack preconditions and present the proof of concept implementation.

3.1 Attack Vector and Threat Model

The adversary's goal is to trigger signed transactions without the user perceiving it. We demarcate the set of manipulations an adversary may introduce to only system-level software, commonly known as the *man in the box* threat model. Software which runs on the external device and on remote (banking) servers, can be considered well secured: the private key is generated with strong parameters and is properly saved on the smart card. The main attack assumption is that the security of the enduser's system is compromised by malware.
Thus, three technical attack vectors to manipulate the user interface arise:

- Operating system level: peripheral manipulation (keyboard, screen, etc.)
- Application level: browser manipulation
- Web application level: client-side JavaScript manipulation

The web application level attack vector poses less effort, but most flexible attack surface. JavaScript may be ported to other operating systems and the attack may be reconstructed for other commercial browser-based banking solutions.

The attack is divided into two stages: at first we modify the running application to include our malicious procedures in any webpage (see Figure 1). Secondly, we conduct the malicious manipulation of transaction using dynamically loaded JavaScript from external sources (see Figure 2).

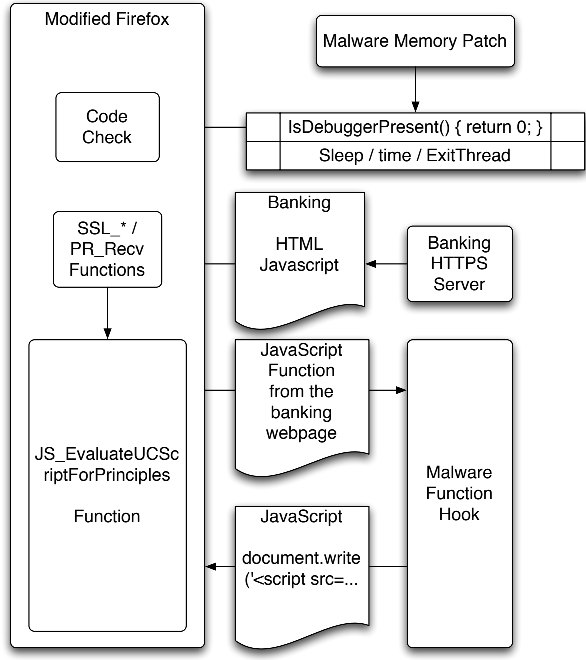

Fig. 1. Overview of stage one of the proof of concept attack.

3.2 Implementation

The implementation of stage one is done using the Immunity Debugger [1] and its Python-based scripting interface. Although a real world rootkit would certainly not use a debugger to hook and modify the relevant functions, we choose this technique because we only wanted to demonstrate the general feasibility.

To circumvent the code integrity checks and to manipulate the JavaScript processing functions, we first patch the `IsDebuggerPresent()` function to remain undetected. We therefore let it return zero for any call, as displayed in Figure 1.

```
1 function = imm.getAddress("kernel32.IsDebuggerPresent")
2 imm.writeMemory(function, imm.Assemble("xor eax, eax\n ret"))
```

Listing 1.1. IsDebuggerPresent always returns zero.

To introduce the malicious JavaScript code, we hook the Firefox function `JS_EvaluateUCScriptForPrinciples()` from `js3250.dll`, which processes any JavaScript found in the originally legitimate HTML received via SSL from the banking server. We then modify the legitimate JavaScript to include a remote script loaded from the adversary's page.

[1] For details see: `http://www.immunityinc.com/products-immdbg.shtml`

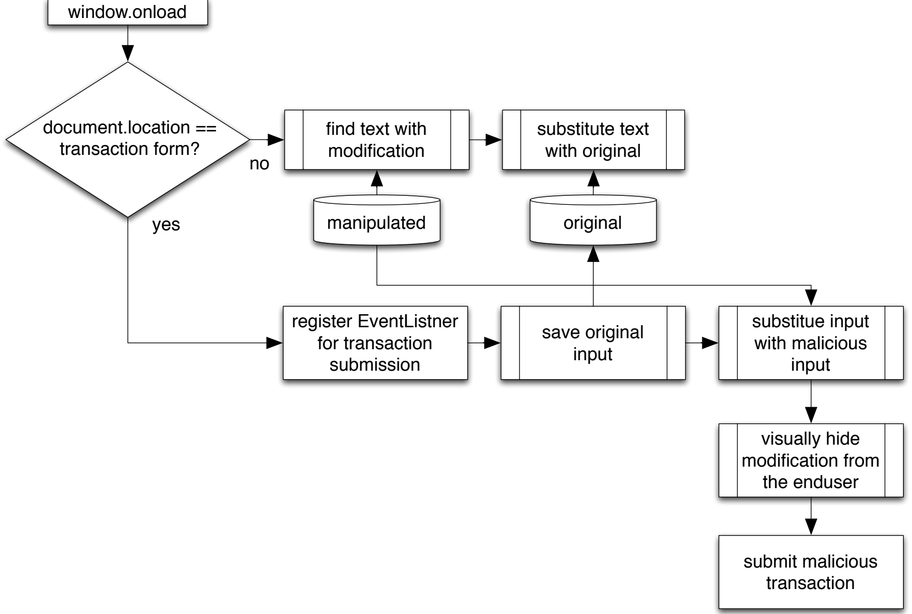

Fig. 2. Overview of stage two of the proof of concept attack.

```
1 document.write(
2   '<script src="http://example.com/malicious.js"></script>')
```

Listing 1.2. Injection of malicious JavaScript.

This way every page viewed in the secure SSL banking context includes malicious and remote JavaScript rootkit which may modify every in- and output on the webpage.

The second stage, displayed in Figure 2, is implemented in the loaded file `malicious.js`. First, when the `window.onload` event fires, the `document.location` is checked whether we are on a banking webpage where the user enters a new transaction destination. If this is the case an event handler is registered for the submit button which in addition to the transfer of specific crucial content of the banking transactions also modifies these values, e.g., the destination account or the amount of money to be transferred. To graphically hide the replacement after the user has clicked the submit button, the text color changes to white. The original input by the user is saved to a cookie for later use. The form is then submitted to the banking server, which expects a transaction to the manipulated account.

If we then land on a webpage which summarizes the transaction, or otherwise displays content, we have to make sure that the manipulated transaction is substituted with the original transaction. We therefore evaluate a regular ex-

pression on all text fields, which replaces the manipulated transaction values with the saved original values from the submission above. This way the enduser never sees the manipulated transaction, because the transaction is only shown in the document object model over which the JavaScript rootkit has full control.

The advantages of implementing the main malicious routines using JavaScript is that the adversary may dynamically react on the content of the target victim site. This is enabled by the dynamic loading of the JavaScript each time a website is rendered and eliminates the need for a malware update function.

4 Related Work

Several approaches exist, e.g. [7], to leverage USB devices towards secure transaction signature devices.

Vulnerabilities of transaction signature software has been shown before [3,6,2]. The main attack against class one readers is the keylogger attack, although the modification of signature text has been demonstrated for common banking software.

Several frameworks exist to modify a running application to inject our malicious payload into the webpage. Although we also leverage code modification techniques to inject JavaScript code, the main rootkit functionality is implemented in JavaScript. Besides [1] this is one of the first attacks on internet software using JavaScript as a main rootkit environment.

5 Mitigation

As already mentioned, the usage of a class one smart card device is a conceptual problem. By using this hardware device further security features are useless.

Because the enduser has no extra control over data signed with his or her private key, all malware attacks can take place on his or her computer. Due to a missing display on the USB device or the possibility of entering the PIN code at the smart card reader itself, the enduser has to rely on the data shown on his or her screen. An additional check whether the signed and actually shown data differ is not a valid option.

All security features that smart cards are capable of, do not apply to the concept considered here, because all software is running on the enduser's system. A good approach to provide more security within this concept is the usage of an at least class two smart card reader. When integrating this hardware, the smart card's PIN does not leave the reader. A smart card reader class three, in contrast, is one step ahead by showing the data to be signed or encrypted on an extra display. In this case the user is able to actually see the data and track the corresponding data flow.

If strong security requirements are defined, as for example for online banking software, concepts have to be considered thoroughly. Just adding extra hardware to the solution neither makes it more secure nor bulletproof. Only a well designed

scheme has good chances to win the arms-race between upcoming advancing malware attacks and a feasible usability.

6 Conclusion

In this paper we have shown how a class one smart card reader implementation can be attacked under the man in the box model. Although this has been common knowledge for years and attacks on these systems have occurred in the past, there are still commercially available products building their security concept on the basis of a class one reader. To conduct the proof of vulnerability, we used a JavaScript-based attack vector and could successfully manipulate the content of the financial transaction without being visible for the user. In summary it turns out that a class one security model is not sufficient when transaction security is demanded. Whenever transaction authenticity is required, at least a class three reader has to be used which allows a check of the transaction details on a trusted display. But it is also clear that some application scenarios exist where a class one reader is the only choice as a result of the business constraints. In this case a classic threat model analysis has to show which transaction mechanism satisfies the required process security.

References

1. B. Adida, A. Barth, U. Berkeley, and C. Jackson. Rootkits for JavaScript Environments. *3rd USENIX Workshop on Offensive Technologies (WOOT'09).*
2. S. Drimer, S. Murdoch, and R. Anderson. Thinking inside the box: system-level failures of tamper proofing. In *IEEE Symposium on Security and Privacy, 2008. SP 2008*, pages 281–295, 2008.
3. H. Langweg, H. Langweg, and E. Snekkenes. A Classification of Malicious Software Attacks. In *Proceedings of 23rd IEEE International Performance, Computing, and Communications Conference*, 2004.
4. M. Marlinspike. More Tricks For Defeating SSL In Practice. *Blackhat USA*, 2009.
5. A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. de Weger. MD5 considered harmful today: Creating a rogue CA certificate (December 2008) 25th Chaos Communications Congress. *Berlin, Germany.*
6. A. Spalka, A. Cremers, and H. Langweg. The Fairy Tale of »What You See Is What You Sign«. Trojan Horse Attacks on Software for Digital Signatures. In *Proceedings of the IFIP WG*, volume 9, 2001.
7. T. Weigold, T. Kramp, R. Hermann, F. Horing, P. Buhler, and M. Baentsch. The Zurich Trusted Information Channel–An Efficient Defence against Man-in-the-Middle and Malicious Software Attacks. *Lecture Notes in Computer Science*, 4968:75–91, 2008.