A browser-based distributed system for the detection of HTTPS stripping attacks against web pages

Marco Prandini and Marco Ramilli

Università di Bologna, DEIS, Viale del Risorgimento 2, 40136 Bologna, Italy
{marco.prandini,marco,ramilli}@unibo.it

Abstract. *HTTPS stripping* attacks leverage a combination of weak configuration choices to trick users into providing sensitive data through hijacked connections. Here we present a browser extension that helps web users to detect this kind of integrity and authenticity breaches, by extracting relevant features from the browsed pages and comparing them to reference values coming from different sorts of trusted sources. The rationale behind the extension is discussed and its effectiveness is demonstrated with some quantitative results, gathered on the prototype that has been implemented for Mozilla Firefox.

Keywords: HTTPS stripping, Peer-to-peer, Browser plugin

1 Introduction

Stealing sensitive data from users is one of the most common targets pursued by attackers on the Web. There are many ways to lure users into providing their data over the wrong connection, leading to the attacker's server instead of the legitimate one. Eventually, the widespread usage of HTTPS seemed like the ultimate weapon against this kind of hijacking. However, the very success of HTTPS backfired as many high-traffic websites staggered under the computational load associated with serving every page through an encrypted connection. This lead some sites to adopt a trade-off solution, foreseeing the usage of HTTPS only for the connections involving the transmission of sensitive data. However, the lack of integrity protection for the page containing the link for the submission opens a crack that an attacker can leverage to compromise the whole transaction. This paper illustrates a method for solving this problem based on a browser extension. In the following, section 2 details the attack; section 3 outlines the design principles of the proposed countermeasure; section 4 describes the extension implementation as a Mozilla Firefox plugin; finally section 5 draws conclusions.

2 Analysis of the Attack

Let's assume the common scenario in which a user on a client host (CH) wants to establish a secure transaction with a Web server on a server host (SH). Given that CH and SH must exchange data on the network, a Man In The Middle (MITM) attack is possible if the attacker host (ATH), by means of skillful manipulation of network devices, becomes a gateway for the traffic stream. The attacker intercepts the traffic from the source and

forwards it to the destination (and vice versa), preserving the illusion of *CH* and *SH* of being connected through an unaltered channel, but at the same time being able to modify messages and insert new ones. While this is not a completely trivial feat, there are sound reasons to worry about this possibility, if the attacker is on the same network of the victim but also if he is in a remote location, due to the insecure default configuration of many home access routers [5,2]. An attack to the professionally-managed infrastructure on the server side is less likely to succeed.

Any kind of MITM would fail if the very first page of the visited site is served on HTTPS (and the user checks it actually is!), because, with some exceptions [1], nobody can circumvent the cryptographic authentication and impersonate the real server. However, the initial page is usually the one responsible for a significant part of a web site traffic, and often is the starting point for a navigation through sections of the site that do not need protection. Thus, to avoid paying the high price associated with serving the first page on HTTPS, many sites use plain HTTP. Then, if the page contains a form for the user to provide identification data, the submission of the form is protected by pointing it to a HTTPS link, reassuring the user about the security of the process by means of graphical cues or textual explanations (Fig. 1).

(c)

```
<form action="https://online.XXXXXX.com/auth/pwd.tb" method="post" id="loginForm"
    name="loginForm" onSubmit="return submitOLBLoginCheckUserName();">
   <label for="userid" class="loginlabel" style="margin-right:13px;">User ID:</label>
   <a href="/XXX/online-banking/forgot-user-id.html"
      onClick="NewWindow(this.href,'product','450','200','no');return false;"
      style="text-decoration:underline;">Forgot User ID?</a>
   <div style="padding-bottom:7px; margin-top:3px;">
   <input id="usernamefield" name="UserName" type="text" class="logininput" size="10"
         maxlength="40" tabindex="1" autocomplete="off" />
   </div><div style="padding-bottom:10px;">
   <a href="/XXX/online-banking/enter-password.html"
      onClick="NewWindow(this.href,'product','575','575','no');
      return false;" style="text-decoration:underline;">Where do I enter my password?</a>
   </div><div>
   <button name="input" type="submit" value="Go" class="logonButton"
         title="Logon to Secure XXXX OnLine Banking" tabindex="3"></button>
   </div><div class="alertMsg" style="padding-top:9px;padding-bottom:1px;">
   <a href="/XXX/about/privacyandsecurity/onlinebankinglogin.html"
      onClick="NewWindow(this.href,'product','660','340','no');return false;"
      style="text-decoration:underline;">About our secure logon</a>
   </div>
</form>
```

Fig. 1. Screenshot of the login box on the home page of a bank. Notice (a) that the page is served on HTTP, (b) the graphics suggesting a secure login process, and (c) the underlying HTML code, which sends data on HTTPS, that is, as long as a MITM attack does not modify it.

However, the attacker is left free to become a MITM between *CH* and *SH* during the first, unprotected exchange of information. He can intercept the initial request/response between *CH* and *SH*, substituting HTTP for HTTPS in every link of the returned page before serving it to *CH*. When the browser on *CH* requests additional contents linked from the page, or submits a form, it actually makes a HTTP connection to ATH, where the attacker can read every byte in plaintext. The attacker then relays every request to *SH* using the correct protocol specified in the original page, to be sure of complying with the configuration of *SH*, and sends the decrypted response back to the browser; possibly, a favicon representing a secure lock is also injected (or crafted into the page), giving a false perception of a secure connection to the client.

The detailed implementation of this attack is described in [4].

3 The proposed countermeasure

All the browsers come with a default setting to alert users about to submit information over an insecure channel. This is a very effective countermeasure against the described attack. Unfortunately, web pages that submit user-provided, harmless information over an insecure channel are in the millions. Thus most users, after the first few false alarms, disable this check [7].

The proposed approach is to treat web pages like any other kind of potentially malicious content, subjecting them to the analysis of a security module very similar to anti-malware software, and comparing the content of the page against suitable information patterns to try and detect if a MITM has modified it. There are two key issues related to this approach, namely choosing a method to extract sensible page features and providing users with the reference features representing authentic pages.

The first issue arises because, nowadays, the vast majority of web pages are dynamically generated. They almost invariably include sections that change each time they are served. It is necessary to characterize a page by extracting only the invariant parts, but making sure that they represent all the contents whose integrity needs to be checked. The result should be a fingerprint of the page, a hash value that can be reliably computed each time the same page is visited and compared to a reference value computed over the authentic page. Then, the second issue comes into play. It is necessary to define how to provide the reference value to every user who is visiting a page in a trusted way.

Regarding the second issue, we envisaged three possible scenarios.

Local database. In principle, each user can build a local database containing the reference values for the pages of his interest. While this method has the undeniable advantage of placing the user in full control of the database, it exhibits a significant drawback: the user must be absolutely sure that he is safe from the MITM attack when he computes the reference value.

Trusted online repository. If the users are willing to place their trust upon a third party of some sort, for example a directory, such a system can act as the authoritative source for computing and distributing reference values. This approach suffers from the usual drawbacks associated with putting a central entity in charge of essential functions: the entity itself becomes a very valuable target for attackers, who would be highly rewarded by a successful compromise of its database or even a simpler DoS attack.

Peer exchange. At any given time, a web page is viewed by a set of clients. The more popular the page, the more interesting target it makes for an attacker, and the larger the set. Under the assumption that mass compromise of clients is unlikely, it is possible to share the reference values between every client through a peer-to-peer network, and to choose the most frequent value associated with a given URL as the correct one.

4 Prototype

We implemented the described solution as a browser plugin which can warn the user of a possible attack. The extension's architecture provides an easy means of porting the code on many different platforms, simply changing the browser-specific interface to the core logic, written in Java. As of now, the SecureExtension (SecExt) plugin is available

for Mozilla Firefox, chosen for being the most widespread open source browser, at `http://code.google.com/p/secureext/downloads/list`, and a usage demo can be viewed at `http://www.youtube.com/user/SecExt`. The plugin architecture is modeled around the the three basic functions outlined in the general description: page characterization, page evaluation, and information sharing. The following paragraphs describe the detail of each phase.

4.1 Page characterization

Web pages are usually composed of many different sections, including parts that are dynamically generated and thus differ each time the page is loaded. Trying to characterize a page by simply computing its hash with a message digest algorithm over its whole content would certainly fail to yield a sensible reference value. It would never be the same even if the page is authentic.

The process we devised for proper characterization starts by observing that, for our purposes, the only important kind of content is the set of links possibly pointing to the submission target of the login form, of other form collecting sensitive data from the user, or possibly opening such a form in a separate but closely related space (iframe, pop-up window, etc.). Every bit of the page which is not a URL is then discarded.

The characterization procedure then removes the parameters (i.e. anything following a "?" character, if present, that could make the same page look different each time it is loaded) from each URL. Their removal does not affect the reliability of attack detection, since the attacker aims simply at changing "https" into "http". Actually, the URL cleaning could be pushed even further by removing everything but the protocol, host and port elements of the URL, to deal with sites that use "/" instead of "?" to have dynamic pages indexed by search engines, but we need further testing to decide whether the (rather small) increase in generality is worth the loss of captured information or not.

Finally, the string originated by the concatenation of the cleaned URLs is given as the input of a message digest algorithm, whose compact and fixed-size output is well suited to summarize the page characteristics.

A page can include code from separate sources, for example by means of `iframe` commands. The process can handle this possibility very easily: SecExt considers each piece of HTML code that can be referenced by a URL as an independent "page". Let's suppose that a separate piece of code is included by the main page to handle user login. If the main page is served on HTTP, the attacker will target the link pointing to the included code, and the attack will be recognized as a modification to the main page. If the main page is secured by HTTPS, but the included code is vulnerable to the stripping attack instead, the latter will be independently characterized and a successful attack against it will be explicitly reported.

4.2 Page evaluation

Each time the user loads a page in the browser, the SecExt plugin computes its hash value according to the illustrated algorithm, then looks for records regarding the page in the database (whose construction is detailed in the next section 4.3). The query can yield different outcomes.

- No records are found for the page's URL. No check can be made about the integrity status of the page. It is possible to envisage a plugin enhancement warning the user trying to submit data on HTTP from this kind of unverifiable pages. The evaluation of the consequences in terms of usability are under investigation.
- The hash of the current page matches the value most frequently associated with its URL in the database. SecExt deduces that most likely the browsed page has not been compromised through an HTTPS stripping attack.
- The hash of the current page does not match the value most frequently associated with its URL in the database. The current page then has a different content from the version most commonly seen in different times or places. The plugin alerts the user by visualizing a warning message on the screen. Before the user can interact with the browsed page he needs to confirm the warning message. Then it is up to the user browsing the page or not, possibly after in-deep verification of the underlying code.

4.3 Information sharing

SecExt can build the database of hash values by composition of two different partial sources: a local database, containing only hashes computed by the local system, and a global database, which is itself a collation of the local databases shared by other users over a P2P network. The sum of these parts allows SecExt to leverage both local knowledge, possibly gathered in a controlled environment where the user can confidently assume to be safe from MITM attacks, and the same kind of knowledge gathered by users who run SecExt as well. In the latter case, we claim that a large enough user base will lead to the population of a global database containing a striking majority of hash values computed over pages which have not been tampered with.

The P2P network run in SecExt is based upon a Java implementation of the Chord protocol [6], chosen for this first prototype for its simplicity. The Chord daemon runs in a background process to keep the communication with peers active independently of the plugin activations. Chord exploits a distributed hash table to store key-value pairs by assigning keys to different computers (known as "nodes"); a node will store the values for all the keys for which it is responsible. Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key. In simpler terms, Chord lets the connected nodes to collectively build a virtual shared folder. Every peer shares its local database as a file, placed in the virtual folder, named by a unique node identifier. The file can get actually copied on other peers when they come online and search for new resources. The virtual global database that is the collation of all the local databases is then materially represented by a highly available collection of files, and the load to access it is spread among the peers.

4.4 Experimental validation

We tested the SecExt plugin effectiveness in a lab environment. The results, which cannot be detailed here for space constraints, showed satisfactory detection rates and a limited amount of false positives. An accurate judgment of our solution, however, must wait until some limitations regarding the security of the P2P exchange are solved and a real-world, wider testing campaign can be rolled out.

5 Conclusions and future work

We surveyed a large set of websites belonging mainly to financial institutions, which are particularly interesting for fraudsters looking for user credentials to steal, and found a significant fraction of them vulnerable to the HTTPS stripping attack. Since users cannot force webmasters to fix the problem where it should be fixed, we proposed a client-side, anti-malware-style approach to the detection of the attack. It leverages the distributed knowledge of a potentially large community of users to identify modified pages even if the user has never visited them before, exploiting peer-to-peer architectures to spread knowledge of the reference values representing unaltered pages without resorting to a trusted third party. We implemented the countermeasure as a plugin for Mozilla Firefox, and verified the practical feasibility and correctness of all its basic principles. The plugin was able to correctly characterize the pages used for testing, taking into account all the relevant data for evaluating its integrity but avoiding to include variable parts that could trigger false positives. Currently, we are working to achieve higher communications efficiency and better handling of updates through finer granularity, whereas for this first prototype we implemented the knowledge sharing as a distribution of the whole reference values database on the P2P network. We are also extending SecExt towards a more comprehensive architecture, to be able to easily "hook" different code-analysis modules into the core logic, timely adding new detection capabilities when new threats appear.

References

1. R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.
2. C. Heffner. How to hack millions of routers. In *Black Hat Conference 2010*, 2010.
3. N. Nikiforakis, Y. Younan, and W. Joosen. Hproxy: Client-side detection of ssl stripping attacks. In C. Kreibich and M. Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 200–218. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14215-4_12.
4. M. Prandini, M. Ramilli, W. Cerroni, and F. Callegati. Splitting the https stream to attack secure web connections. *IEEE Security and Privacy*, 8:80–84, November 2010.
5. S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by pharming. In S. Qing, H. Imai, and G. Wang, editors, *Information and Communications Security*, volume 4861 of *Lecture Notes in Computer Science*, pages 495–506. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-77048-0_38.
6. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.
7. J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: an empirical study of ssl warning effectiveness. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 399–416, Berkeley, CA, USA, 2009. USENIX Association.