HProxy: Client-side detection of SSL stripping attacks

Nick Nikiforakis, Yves Younan, and Wouter Joosen

IBBT-DistriNet
Katholieke Universiteit Leuven
Celestijnenlaan 200A B3001
Leuven, Belgium
{nick.nikiforakis,yves.younan,wouter.joosen}@cs.kuleuven.be

Abstract. In today's world wide web hundreds of thousands of companies use SSL to protect their customers' transactions from potential eavesdroppers. Recently, a new attack against the common usage of SSL surfaced, SSL stripping. The attack is based on the fact that users almost never request secure pages explicitly but rather rely on the servers, to redirect them to the appropriate secure version of a particular website. An attacker, after becoming man-in-the-middle can suppress such messages and provide the user with "stripped" versions of the requested website forcing him to communicate over an insecure channel. In this paper, we analyze the ways that SSL stripping can be used by attackers and present a countermeasure against such attacks. We leverage the browser's history to create a security profile for each visited website. Each profile contains information about the exact use of SSL at each website and all future connections to that site are validated against it. We show that SSL stripping attacks can be prevented with acceptable overhead and without support from web servers or trusted third parties.

Key words: MITM Detection, SSL Stripping, Browser Security

1 Introduction

In 1994 Netscape Communications released the first complete Secure Sockets Library (SSL) which allowed applications to exchange messages securely over the Internet [20]. This library uses cryptographic algorithms to encrypt and decrypt messages in order to prevent the logging and tampering of these messages by potential eavesdroppers. Today SSL is considered a requirement for companies who handle sensitive user data, such as bank account credentials and credit card numbers. According to a study by Netcraft[16], in January of 2009 the number of valid SSL certificates on the Internet reached one million, recording an average growth of 18,000 certificates per month. Due to its widespread usage, attackers have developed several attacks, mainly focusing in the forging of invalid SSL certificates and hoping that users will accept them.

Recently however a new attack has surfaced [13]. This technique is not based on any specific programming error but rather on the whole architecture and usage

of secure webpages. It is based on the observation that most users never explicitly request SSL protected websites, in the sense that they never type the `https` prefix in their browsers. The transition from cleartext pages to encrypted ones is done usually either through web server redirects, secure links, or secure target links of HTML forms. If an attacker can launch a man-in-the-middle (MITM) attack, he can suppress all such transitions by "stripping" these transitional links from the cleartext HTTP protocol or HTML webpages before forwarding these messages/webpages to the unsuspecting client. Due to stripping of all SSL information, all data that would originally be encrypted are now sent as cleartext by the user's browser providing the attacker with sensitive data such as user credentials to email accounts, bank accounts and credit card numbers used in online transactions.

In this paper, we explore the idea of using the browser's history as a detection mechanism. We design a client-side proxy which creates a unique profile for each secure website visited by the user. This profile contains information about the specific use of SSL in that website. Using this profile and a set of detection rules, our system can identify when the page has been maliciously altered by a MITM and block the connection with the attacker while notifying the user of an attacker's presence on the network. Our approach does not require server-side cooperation and it does not rely on third-party services.

The main contributions of this paper are:

- Analysis and extension of a new class of web attacks
- Development of a generic detection ruleset for potential attack vectors
- Implementation of a client-side proxy which protects end-users from such attacks

The rest of this paper is structured as follows. In Section 2, we describe how SSL stripping attacks work followed by the reasons which make these attacks widely effective in Section 3. In Section 4 we present the architecture and workings of HProxy. We discuss some difficulties and how we overcame them in Section 5. In Section 6 we present the evaluation of our approach followed by some implementation details in Section 7. Section 8 discusses the related work and we conclude in Section 9.

2 Anatomy of SSL stripping attacks

Once an attacker becomes MITM on a network, he can modify HTTP messages and HTML elements in order to trick the user's browser into establishing unencrypted connections. In the following two scenarios we present two successful attacks based on redirect suppression and target form re-writing. The first attack exploits HTTP protocol messages and the second attack rewrites parts of a cleartext HTML webpage.

2.1 Redirect Suppression

1. The attacker launches a successful MITM attack against a wireless network becoming the network's gateway. From this point on, all requests and responses from any host on the wireless network are inspected and potentially modified by him.
2. An unsuspecting user from this wireless network uses his browser and types in the URL bar, `mybank.com`. The browser crafts the appropriate HTTP message and forwards the message to the network's gateway.
3. The attacker inspects the message and realizes that the user is about to start a transaction with `mybank.com`. He forwards the message to MyBank's webserver.
4. `mybank.com` protects their entire website using SSL thus, the webserver responds with a 301 (Moved Message) to `https://www.mybank.com`.
5. The attacker intercepts the move message, and instead of forwarding it to the user, he establishes a secure connection with MyBank and after decrypting the resulting HTML, he forwards that to the user.
6. The user's browser receives cleartext HTML, as a response to his request and renders it. What the user now sees is an unencrypted version of MyBank's login page. The only thing that is missing is a subtle lock icon, which would be otherwise located somewhere on the browser window.
7. From this point on, all user-data are transmitted as cleartext to the attacker, where he tunnels them through his own encrypted connection. This results in completely functional but unencrypted web sessions.

2.2 Target form re-writing

This attack is quite similar to the redirect suppression attack except for a significant detail. Target form re-writing is an attack against websites which operate mainly over HTTP and they only protect parts of their webpages, such as a login form and any subsequent pages for logged-in users. The way this is constructed in HTML is that while the main page is transfered over HTTP, the target URL of a specific form has an HTTPS prefix. When the user clicks the "submit" button, the browser recognizes the secure protocol and attempts to establish an SSL connection with the target web server. This is disastrous for an attacker because, even though he controls all local network connections, he has no realistic way of presenting a valid SSL certificate for the secure handshake of the requested web server. The attacker thus, will have to present a self-signed certificate resulting in multiple warnings which the user must accept before proceeding with the connection. In order to avoid this pitfall, the attacker strips all secure form links and replaces them with cleartext versions. So, a form with a target of `https://www.example.com/login.php` becomes `http://www.example.com/login.php` (note the missing `S` from the protocol). The browser has no way of knowing that the original link had a secure target and thus sends the user's credentials over an unencrypted channel. In the same way as before, the attacker uses these credentials in his own valid SSL connection and later forwards to the user the resulting HTML page.

3 Effectiveness of the attack

In this section we would like to stress the severity of the SSL attacks described in Section 2. We argue that the two main reasons which make SSL stripping such an effective attack are: a) the wide applicability of it in modern networks and b) the way that feedback works on browser software.

3.1 Applicability

When eavesdropping attacks were first introduced, they targeted hubbed networks since hubs transmit all packets to all connected hosts, leaving each host to choose the packets that are addressed for itself and disregard the rest. The attacker simply configured his network card to read all packets (promiscuous mode) and had immediate access to all the information coming in and out of the hubbed network. Once hubs started being replaced by switches, this attack was no longer feasible since switches forwarded packets only to the hosts that were intended to receive them (using their MAC addresses as a filter). Attackers had to resort to helper techniques (such as ARP flooding, which filled-up the switch's memory forcing it to start transmitting everything to everyone to keep the network functioning) in order for their eavesdropping attacks to be effective [15].

Today however, due to the widespread use of wireless network connections, attackers have access to hundreds of thousands of wireless networks ranging from home and hotel networks to airport and business networks. Wireless networks are by definition hubbed networks since the transport medium is "air". Even secure wireless networks (WEP/WPA2) are susceptible to MITM attacks as long as the attacker can find the encryption key (trivial for WEP [25] not so trivial for WPA2).

The ramifications become even greater when we consider that wireless networks are not restricted to laptops anymore due to the market penetration of hand held devices which use them to connect to the Internet. More and more people use these kind of devices to perform sensitive operations from public wireless networks without suspecting that a potential attacker could be eavesdropping their transactions.

3.2 Software feedback

The second main reason that makes this attack effective is that it doesn't produce negative feedback. Computer users have been unconsciously trained for years that the absence of warning messages and popups means that all operations were successful and nothing unexpected happened. This holds true also for security critical operations where users trust that a webpage is secure as long as the browser remains "silent".

In the scenario where an attacker tries to present to a web browser a self-signed, expired or otherwise illegal certificate, the browser presents a number of dialogues to the user which inform him of the problems and advise him not to

proceed with his request. Modern browsers (such as Firefox) have the user click many times on a number of different dialogues before allowing him to proceed. Many users, understand that it is best to trust their browser's warnings, especially if they are working from an unfamiliar network (such as a hotel network), even if they end up not doing so [22].

In the SSL stripping attack however, the browser is never presented with any illegal SSL certificates since the attacker strips the whole SSL connection before it reaches the victim. With no warning dialogues, the user has little to no visual cues that something has gone wrong. In the case of SSL-only websites (websites that operate solely under the HTTPS protocol) the only visual cue that such an attack generates is the absence of lock icon somewhere on the browser's window (something that the attacker can compensate for by changing the .favico icon of the website to a padlock). In partly-protected websites, where the attacker strips the SSL protocol from links and login forms, there are no visual cues and the only way for a user to spot the attack is to manually inspect the source code and identify the parts that have been changed.

4 Automatic Detection of SSL stripping

In this section we describe our approach that automatically detects the existence of a MITM attacker conducting an SSL stripping attack on a network. The main strength of MITM attacks is the fact that the attacker has complete control of all data coming in and going out of a network. Any client-side technique trying to detect an attacker's presence must never rely solely on data received by the current network connection.

4.1 Core Functionality

Our approach is based on browser history. The observation that lead to this work is that while a MITM attacker has at some point in time, complete control of all traffic on a network, he did not always have this control. We assume that users mainly use secure networks, such as WPA2-protected wireless networks or properly configured switched networks and use insecure networks only circumstantially. Regular browsing of SSL-enabled websites from these secure locations can provide us with enough data to create a profile of what is expected in a particular webpage and what is not.

Our client-side detection tool, History Proxy (HProxy), is trained with the requests and responses of websites that the user regularly visits and builds a profile for each one. It is important to point out that HProxy creates a profile based on the security characteristics of a website and not based on the website's content, enabling it to operate correctly on static as well as most dynamic websites.

HProxy uses the profile of a website, the current browser request and response along with a detection ruleset to identify when a page is maliciously modified by a MITM conducting an SSL stripping attack. The detection ruleset is straightforward and will be explained in detail in Section 4.3.

4.2 Architecture of HProxy

The architecture of HProxy comprises of the detection ruleset and a number of components which utilize and enforce it - Fig. 1. The main components are: a webpage analyzer, which analyzes and identifies the requests initiated from the browser along with the server responses, a MITM Identifier which checks requests and responses against the detection ruleset to decide whether a page is safe or not and lastly a taint module which tries to prevent the leakage of private information even if the MITM-identifier incorrectly tags a page as safe.

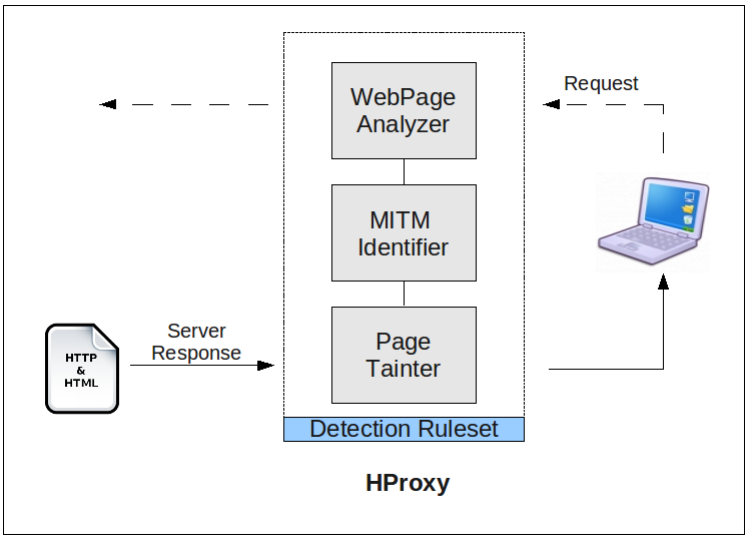

Fig. 1. Architecture of HProxy

Webpage analyzer The webpage analyzer is the component responsible of identifying all the critical parts of a webpage. The critical parts of a webpage are the parts that a MITM attacker can insert or alter in order to steal credentials from the end users and are the following:

- JavaScript blocks
- HTTP forms and their targets
- `Iframe` tags
- HTTP Moved messages

The Webpage Analyzer identifies all of the above data structures, along with their attributes and records them in the page's current profile. If a particular page is visited for the first time then this current profile is registered in the

profile database, effectively becoming the page's original profile, and the page is forwarded to the user. If not, then the current profile will be checked against the page's original profile by the MITM Identifier. Why these structures are dangerous will be described in detail in Section 4.3.

MITM Identifier The MITM Identifier component encapsulates almost all the detecting capabilities of HProxy (except of the taint component which will be discussed later). It uses the page's current profile as created by the Webpage Analyzer against the page's original profile. In order to make a decision whether a page is altered by an attacker or not, the MITM Identifier utilizes the detection ruleset of HProxy. This ruleset consists of rules for every sensitive data structure that was previously mentioned. Each rule contains the dangerous modifications that can appear in each page, using the page's original profile as a base. Any modifications detected by the Webpage Analyzer that are identifiable by this ruleset are considered a sign of an SSL stripping attack and thus the page is not forwarded to the user.

PageTainter Even though we have strived to create a ruleset which will be able to detect all malicious modifications we deliberately decided to allow content changes when we cannot decisively classify them as an attack. In order to compensate for these potentially false negatives, HProxy contains a module called PageTainter. The purpose of PageTainter is to enable HProxy to stop in time the leakage of private user data, even when the MITM Identifier module wrongly tags a malicious page as "safe". For HProxy to stop the leakage of private data, it must first be able to identify what private data is. In order to do this, PageTainter modifies each webpage that contains a secure login form (identifiable by the password-type HTML element) and adds a JavaScript routine which sends the password from it to HProxy once the user types it in. This password is recorded in HProxy in a domain,password tuple[1]. In addition to that, it taints all forms with an extra hidden field which contains location information so that we can later identify which page initiated a GET or a POST request. For each request that initiates from the browser, the PageTainter module, using the hidden domain field checks for the presence of the stored password in the outgoing data. If the page is legitimate, the domain's password will never appear in the HTTP data because it is exchanged only over SSL. A detection of it signifies the fact that an attacker's successful modification passed through our MITM Identifier and is now sending out the password. In this case, HProxy does not allow the connection to be established and informs the user of the attack. To make sure that an attacker will not obfuscate the password beyond recognition by the PageTainter, our detection ruleset has very strict JavaScript rules which will be explained in the next section.

[1] HProxy runs on the same physical host as the browser(s) that it protects thus there are no privacy issues with the stored passwords

4.3 Detection Ruleset

Using the description of SSL-stripping attacks as a base, we studied and recorded all possible HTML and HTTP elements that could be misused by a MITM attacker. This study resulted in a set of pragmatic rules which essentially describe dangerous transitions from the original webpage (as recorded by HProxy) to all future instances of it. A transition can be either an addition of one or more HTML/HTTP elements by the attacker to the original webpage or the modification of existing ones.

The detection ruleset consists of dangerous modifications for every class of sensitive data structures. Each page that comes from the network is checked against each class of rules before it is handed back to the user. In the rest of this section we present the rules for each class of sensitive structures.

HTTP Moved Messages The HTTP protocol has a variety of protocol messages of which the "moved" messages can be misused in an SSL stripping attack since their suppression can lead to unencrypted sessions (as shown in the example attack in Section 2.1). The main rule for this class of messages states that, if the original page profile contains a move message from an `HTTP` to an `HTTPS` page, then any other behavior is potentially dangerous. Given an original request of `HTTP GET` for `domain_a` and an original response stored in the profile database of `MOVED` to `HTTPS domain_a/page_a`, we list all the possible modifications and whether they are allowed by our ruleset, in the following table.

Current Response	**Modification**	**Allowed?**
MOVED HTTPS domain_a/page_a	None	Yes
MOVED HTTPS domain_a/page_b	Changed page	Yes
MOVED HTTP domain_a/page_a	Non-SSL protocol	No
MOVED HTTP domain_b/page_a	Changed domain	No
MOVED HTTPS domain_b/page_a	Changed domain	No
OK `<html>....</html>`	HTML instead of MOVED	No

This ruleset derives from the observation that the developers of a website may decide to create new webpages or rename existing ones, but they will not suddenly stop providing HTTPS nor export their secure service to another domain. For websites that operate entirely using SSL, this is the only class of rules that will be applied to them as they will operate securely over HTTPS once the `MOVE` message has been correctly processed.

The rest of the ruleset is there to protect websites that are partly protected by SSL. Such websites use SSL only for their login forms and possibly for the subsequent pages that result after a successful login. The transition from unprotected to protected pages (within the same website) is done usually through a HTTPS form target or through a HTTPS link.

JavaScript JavaScript is a powerful, flexible and descriptive language that is legitimately used in almost all modern websites to make the user experience

better and to offload servers of common tasks that can be executed on the client-side. All these features of JavaScript, including the fact that it is enabled by default in all major browsers make it an ideal target for attackers. Attackers can and have been using JavaScript for a multitude of attacks ranging from Cross-site Scripting [11] to Heap Spraying attacks [19]. For the purpose of stealing credentials, JavaScript can be used to read parts of the webpage (such as a typed-in username and password) and send it out to the attacker.

JavaScript can be categorized as either inline or external. Inline JavaScript, is written inline an HTML webpage, e.g. `<html><script>...</script> </html>`. External JavaScript, is written in separate files, present on a webserver that are being included in an HTML page using a special tag, e.g. `<html><script src="http://domain1/js_file.js"> </html>`. Unfortunately for users, both categories of JavaScript can be misused by a MITM. If an attacker adds inline JavaScript in a webpage before forwarding it to the user, the browser has no easy way of discerning which JavaScript parts were legitimately present in the original page and which were later added by the attacker. Also, the attacker can reply to a legitimate external JavaScript request with malicious code since he already has full control over the network and can thus masquerade himself as the webserver.

Because of the nature of JavaScript, HProxy has no realistic way of discerning between original and "added" JavaScript except through the use of whitelisting. The first time that a page which contains an HTTPS form is visited all JavaScript code (internal and external) is identified and recorded in the page's profile. If in a future request of that specific webpage, new or modified JavaScript is identified then the page is tagged as unsafe and it is not forwarded to the user. HProxy's initial whitelisting mechanism involved string comparisons of JavaScript blocks between page loads of the same website. Unfortunately though, the practice of simple whitelisting can lead to false positives. A way around these false positives is through the use of a JavaScript preprocessor. This preprocessor can distinguish between the JavaScript parts that have been legitimately changed by the web server and the parts which have been added or modified by an attacker. We expand HProxy to include such a preprocessor and we explore this notion in detail later on, in Section 5.

Iframe tags can be as dangerous as JavaScript. An attacker can add extra `iframe` tags in order to overlay fake login forms over the real ones [7] or reply with malicious content to legitimate `iframe` requests. Our detection ruleset for `iframe` tags states that no such tags are allowed in pages where an SSL login form is present. The only time an `iframe` tag is allowed is when the original profile of a website states that the login form itself is coded inside the `iframe`.

HTTP Forms can be altered by a MITM attacker so as to prevent the user's browser from establishing an encrypted session with a web server, as was demonstrated in Section 2.2. Additionally, extra forms can also be used by an attacker

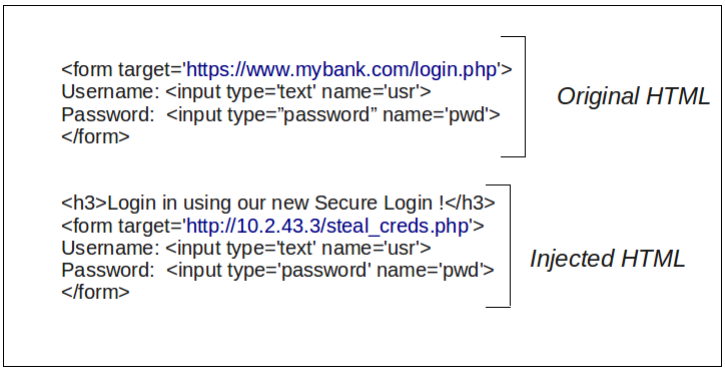

Fig. 2. Example of an injected HTML form by a MITM attacker

as a way of stealing private information. The set of rules for this class of sensitive data structures is similar to the HTTP Move class ruleset. The previously mentioned Webpage analyzer, records every form, target and protocol for each page that an SSL login form is identified. The ruleset contains the dangerous form modifications that could leak private user credentials. The main rules are applied on the following characteristics:

- **Absence of forms** - The profile for each website maintains information about the number of forms in each page, whether they are login forms and which forms have secure target URLs. Once a missing form is detected, HProxy reads the profile to see the type of the missing form. If the missing form was a secure login form then HProxy tags this as an attack and drops the request. If the missing form was a plain HTTP form (such as a `Search` form) then HProxy allows the page to proceed.
- **New forms** - New forms can be introduced in a webpage either by web designers (who wish to add functionality to a specific page) or by an attacker who tries to lure the user into typing his credentials in the wrong form - Fig 2. If the new form is not a login form then it is an allowed deviation from the page's profile. If the new form is a login-form it is only allowed if the target of the form is secure and in the same domain as the original SSL login form of the page. Even so, there is a chance that a MITM can convince a user to submit his credentials through a non-login form. In these cases, PageTainter will identify the user's password in outgoing data and drop the request before it reaches the attacker.
- **Modified forms** - In this case, an attacker can modify a secure form into an insecure form. Based on the same observation from HTTP moved messages, HProxy does not allow a modified form to be forwarded to the browser if it detects: `(a)` a security downgrade in a login form (the original had an HTTPS target whereas the current one has an HTTP target); or `(b)` a domain change in the target URL

4.4 Redirect Suppression Revisited

In Section 2.1 we presented one of the most common SSL stripping attacks against browsers, namely redirect suppression. The MITM suppressed the HTTP `Moved` messages and provided the user with an unencrypted version of an originally encrypted website. In this section we repeat the attack but this time, the user is running the HProxy tool. Steps 1-5 are the same with the earlier example but are repeated here for the sake of completeness.

1. The attacker launches a successful MITM attack against a wireless network becoming the network's gateway. From this point on, all requests and responses from any host on the wireless network are inspected and potentially modified by him.
2. An unsuspecting user from this wireless network uses his browser and types in the URL bar, `mybank.com`. The browser crafts the appropriate HTTP message and forwards the message to the network's gateway.
3. The attacker inspects the message and realizes that the user is about to start a transaction with `mybank.com`. He forwards the message to MyBank's webserver.
4. `mybank.com` protects their entire website using SSL thus, the webserver responds with a 301 (Moved Message) to `https://www.mybank.com`.
5. The attacker intercepts the move message, and instead of forwarding it to the user, he establishes a secure connection with MyBank and after decrypting the resulting HTML, he forwards that to the user.
6. HProxy receives the response from the "server" and inspects it. HProxy's trained profile for MyBank states that `mybank.com` is an SSL protected website and when the user requests the website using HTTP, the server redirects him to the HTTPS version of it. This time however HProxy identifies the response as cleartext HTML which is not acceptable according to its detection ruleset.
7. HProxy drops the request and notifies the user about the presence of a MITM on the local network along with specific details.

5 Discussion

By analyzing the JavaScript code generated by the top visited websites (as reported by Alexa [24]) we discovered that the dynamic nature of today's Internet doesn't stop in dynamically generated HTML. Many top websites provide different JavaScript code blocks each time they are visited, even when the visits are seconds apart. This means that a simple whitelisting of JavaScript based on string comparison would result in enough false positives to render HProxy unusable. In this section we discuss two techniques that can greatly reduce these false positives: JavaScript preprocessing and Signed JavaScript. The final version of HProxy includes a JavaScript Preprocessor while Signed JavaScript can be used in the future to completely eliminate false positives. We also describe a different way of identifying a MITM by inspecting client requests and the potential problems of that approach.

5.1 JavaScript Preprocessing

Most of the JavaScript blocks, even the ones that constantly change, follow a specific structure that can be tracked along page loads. By comparing internal and external JavaScript along two consecutive page loads of a specific webpage, we can discover the static and the dynamic parts of that code. E.g., The JavaScript code in two consecutive loads of Twitter's login page differs only in the contents of a specific variable - Fig. 3

We leverage this re-occurring structure to design a JavaScript preprocessor that greatly reduces false positives. When a website is visited for the first time through HProxy, the Webpage Analyzer (Section 4.2) makes a duplicate request and compares the JavaScript blocks from the original response and the duplicate one. If the blocks are different it then creates a template of the parts that didn't change and records the place and length of the dynamic parts. This information is stored in the Web pages profile and all future visits of that website will be validated against this template. This enables us, to discern between normal dynamic behavior of a website and JavaScript that was maliciously added by a MITM in order to steal the user's credentials. Although a JavaScript preprocessing that would work on an interpretation level would possibly be able to produce zero false positives we believe that the overhead of such an approach would be prohibitively high and thus we did not research that direction.

5.2 Signed JavaScript

Signed JavaScript (SJS) is JavaScript that has been signed by the web server using a valid certificate such as the one used in HTTPS communications. SJS can provide among other features (such as access to restricted JavaScript functions) the guarantee that the script the browser parses has not been modified since it was sent by the Web server [17]. This integrity assurance can be used by HProxy to whitelist unconditionally all JavaScript code blocks that are signed. The downside of this technique is that it requires both server and client-side support[2].

5.3 Inspecting Client Requests vs. Server Responses

It is evident that trying to secure JavaScript at the client-side can be a tedious and error-prone process. A different approach of detecting a MITM which may at first appear more appealing is to analyze the client-side requests for anomalous behavior rather than the server-side responses to client-side requests. In such a case, the resulting system would inspect the requests (both secure and insecure) of the browser and compare them to the requests done in the past. A security downgrade of a request, (e.g. the browser is currently trying to communicate to website `X` using an unencrypted channel whereas it always used to communicate over a secure channel), would be a sign of a MITM operating on the network

[2] At the time of this writing, only Mozilla Firefox appears to support SJS.

```
page.controller_name = 'SessionsController';
page.action_name = 'new';
twttr.form_authenticity_token =
'bcf48ddc78846bea1db1f357300d3e4ad174e2ee';


  page.controller_name = 'SessionsController';
  page.action_name = 'new';
  twttr.form_authenticity_token =
  '644bb1da2eaf04ef5983b7b36d38f411d962856a';
```

Fig. 3. Portion of the JavaScript code present in two consecutive page loads of the login page of Twitter. The underlined part is the part that changes with each page load

and the request would be dropped. In such a system, JavaScript whitelisting would not be an issue since HProxy would only inspect the outgoing requests, regardless of their origin (HTML or JavaScript).

While this approach looks promising it produces more problems than it solves since it has no good way of discerning the nature of new outgoing requests. Consider the scenario where an attacker adds a JavaScript routine which copies the password from the correct form, encrypts it and sends it out using an AJAX request to a new domain. The system would not be able to find a previous outgoing request to match the current request by, and would have to either drop the request (also dropping legitimate new requests - false positives) or let it pass (false negatives). Also, in partly SSL-protected pages, where the client communicates with the same website using both encrypted and unencrypted channels, the MITM could force the browser to send private information over the wrong channel which would again result in leaking credentials.

For these reasons, we decided that a combination of inspecting server responses, preprocessing JavaScript and tracking private data (through the PageTainter - 4.2) would be more effective than inspecting client requests and thus we did not implement such a system.

6 Evaluation

In this section we provide a security evaluation, the number of false positives and the performance overhead of our approach.

6.1 Security Evaluation

HProxy can protect the end-user against the attacks described in [13] as well as a number of new techniques that could be used to steal user credentials in the

context of SSL stripping attacks. It can protect the user from credential stealing through redirect suppression, insecure forms, JavaScript methods and injected `iframe` tags.

In order to test the actual effectiveness of our prototype we created a network setup with two clients and a wireless Access Point(AP) with Internet connection. One client was the legitimate user and the other one the MITM, both running the latest version of Ubuntu Linux. From the MITM machine we enabled IP forwarding and we used the `arpspoof` (part of the `dsniff` suite [6]) to position ourselves between the victim machine and the AP. We then run `sslstrip` [21], a tool which strips the SSL links from incoming traffic, creates SSL tunnels with the legitimate websites and captures sensitive data typed by the user. We started browsing the web from the victim machine and we observed that pages which normally are protected through SSL (like GMail and Paypal) were now appearing over HTTP, without any browser warnings whatsoever. Any data typed in fields of those pages were successfully eavesdropped by the MITM host.

We reset the experiment, enabled HProxy and started browsing the web. We browsed through a number of common websites so that HProxy could create a profile for each one of them. We then repeated the procedure of becoming MITM and run `sslstrip`. Through the victim client, we started visiting all the previously "stripped" websites. This time however, HProxy detected all malicious changes done by `sslstrip` and warned the user of the presence of a MITM attacker on the network.

6.2 False Positives

A false positive, is an alert that an Intrusion Detection System (IDS) issues when it detects an attack, that in reality did not happen. When HProxy parses a page, it can occasionally reach to the conclusion that the page was modified by an attacker even if the page was legitimately modified by the web server. These false conclusions can confuse the user as well as undermine his trust of the tool. Most of HProxy's false positives can be generated by its JavaScript rules, as explained in section 4.3.

In order to make these occasions as rare as possible we decided to monitor JavaScript blocks only in pages that contain (or originally contained) secure login forms. This decision does not undermine the overall security of HProxy since in the context of SSL Stripping attacks, JavaScript can only be used to steal credentials as they are typed-in by the user in a secure form. In addition to that, we developed a JavaScript Preprocessor, as explained in Section 5.1 which generates a template of each website and a list of expected JavaScript changes.

To measure the amount of false-positives, we compiled a list of 100 websites that contain login pages and we programmed Firefox using `ChickenFoot` [4] to automatically visit them three consecutive times. Firefox's incoming and outgoing traffic was inspected by HProxy which in turn decided whether the page was secure or not. The first time the page was visited, HProxy created a profile for it, which it used for the next two times. Due to our lab secure network settings, any attack reported by HProxy was a false positive.

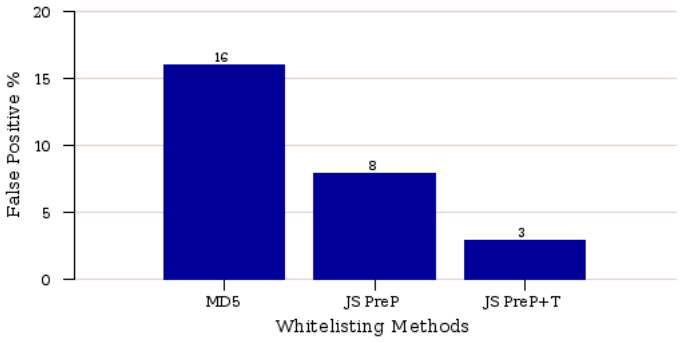

Fig. 4. False-positive ratio of HProxy using three different methods of whitelisting JavaScript

In Fig. 4 we present the ratio of HProxy's false-positives using three methods of whitelisting JavaScript. The first method that we used is simply gathering all the JavaScript blocks of a webpage and computing their MD5 checksum. If the JavaScript blocks between two page loads differ, then their checksums will also be different. In the second method, we use the JavaScript preprocessor with a strict template, where the changes detected by the preprocessor must be in the precise place and of precise length as the ones originally recorded. Finally we use the same preprocessor but this time we include a "tolerance factor" of 10 characters, where the position and length of changes may vary up to 10 characters (less that 1% of the total length of JavaScript code for most websites).

Using the last method as the whitelisting method of choice, HProxy can handle almost all JavaScript changes successfully. The false-positives are created by webpages which produce JavaScript blocks of different length each time that they are visited. The websites that contain such pages are always the same and can thus be added to a list of unprotected pages.

6.3 Performance

To measure the performance overhead of our HProxy prototype, we used a list of the top 500 global websites [24] and we programmed Firefox to visit them ten times each while measuring how much time each page needed to fully load. In order to avoid network inconsistencies we downloaded a copy of each website and browse them locally using a web server that we setup on the same machine that Firefox was running. All caching mechanisms of Firefox were disabled and we were clearing the Linux memory cache between experiments. We repeated the experiment three times and in Fig. 5 we present the average load time of Firefox when it run:`(a)` without a proxy `(b)` using a proxy that just forwarded requests to and from Firefox and `(c)` using HProxy. Hproxy shows an overhead of 33% when compared with the forwarding proxy and 51% when compared with Firefox

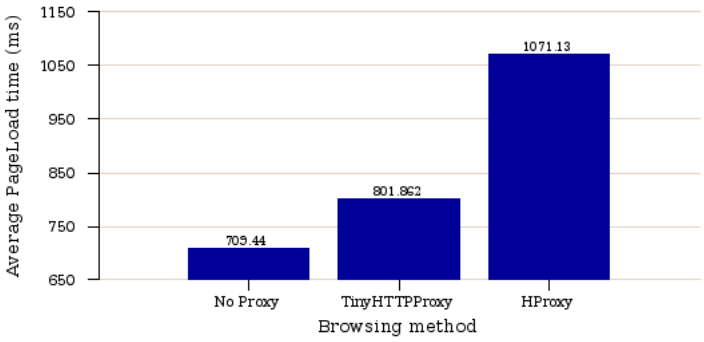

Fig. 5. Average load time of the top 500 websites of the Internet when accessed locally without a proxy, with a simple forwarding proxy(TinyHTTPProxy) and with HProxy

directly accessing the web pages. While this overhead appears substantial, it is important to remember that even the 51% overhead is actually an overhead of 0.41 seconds of time. Firefox starts rendering received content, long before each page fully loads. This means that the user can start "consuming" the content of each page without having to wait for all objects to be downloaded. Given this behavior, we believe that the added delay of HProxy is only minutely, if at all, perceived by the user during normal web browsing.

7 Implementation

We implemented a prototype version of HProxy using Python. We used an already implemented Python proxy, TinyHTTPProxy [10] and we built on top of it to add the various detection mechanisms that were described in earlier sections. We chose to implement HProxy as a stand-alone application and not as a browser plugin because we wanted to test parts of its functionality (such as the AJAX functions emmited by the PageTainter module) with multiple browsers. HProxy runs on the same physical machine as the browser(s) that it protects. A proxy running on a different machine could potentially be used by multiple users to improve caching but that would allow a MITM to impersonate HProxy and steal user credentials. The Webpage Analyzer and the PageTainter modules use the BeautifulSoup HTML parser [2] to recognize forms, JavaScript and `iframe` tags. For the HTTP `Moved` messages we wrote our own parser using regular expressions.

The reason why we chose Python instead of another programming language is because Python's features make it ideal for fast prototyping. We believe however, that if HProxy gets re-implemented using a compiled language or if it becomes part of a browser (as an extension or as part of the browser's code) the overhead of its use will be much lower than the one we measured in Section 6.3.

8 Related work

To the best of our knowledge, this paper is the first academic countermeasure which is specifically geared towards SSL stripping attacks. Previous studies mainly focus on the detection of a MITM attacker especially on wireless networks. While a number of these studies detect a wider range of attacks than our approach, it is important to point out that most of them require either specific hardware or knowledge of the network that surpasses the average user's session. This effectively means that unless the techniques are employed before-hand by the administrators of the network they can be of little to no use to the connecting clients. On the other hand HProxy is a client-side tool which protects users from SSL stripping attacks without requiring any support from the wireless network infrastructure.

A number of studies use the information already existing in the 802.11 protocol to identify attackers that try to impersonate legitimate wireless nodes by changing their MAC address. The authors of [9, 26] use the sequence number field of MAC frames as a heuristic for detecting nodes who try to mimic existing MAC addresses. The sequence number is incremented by the node every time that a frame is sent. They create an intrusion detection system which identifies attackers by monitoring invalid, duplicate or dis-proportionally large sequence numbers. Martinez et al. [14] suggest the use of a dedicated passive Wireless Intrusion Detection System (WIDS) which identifies attackers by logging and measuring the time interval between beacon frames. Beacon frames that were broadcasted before the expiration of the last beacon frame (as announced by the AP) are a sign of an impersonation attack. In the same manner, Laroche et. al [12] present a WIDS which uses information such as sequence numbers and fragment numbers, to identify layer-2 attacks. Genetic algorithms are executed against these datasets in an effort to identify impersonating nodes. Unfortunately, their IDS requires training on labeled data sets making it impractical for fast fluctuating wireless networks such as the ones deployed in hotels and airports where wireless nodes are constantly added and removed.

Other researchers have focused more on the physical characteristics of wireless networks and how they relate to intrusion detection. Chen et. al [3] as well as Sheng et al. [18] use the Received Signal Strength (RSS) of a wireless access point as a way to differentiate between the legitimate access point(s) and an attacker masquerading as one. In both studies, multiple passive gathering devices are used to record the RSS and the data gathered is analyzed using cluster algorithms and Gaussian models. Similarly Suski et al. [23] use special wireless hardware monitors to create and monitor an "RF Fingerprint" based on the inherent emission features of each wireless node. While the detection rates of such studies are quite high, unfortunately their approaches are inherently tied to a significant increase in setup costs (in time, hardware or both) making them unattractive for everyday deployment environments.

Moving up to the top layer of the OSI model, several studies have shown that security systems lack usability and that users accept dialogues and warnings without really understanding the security implications of their actions [1, 5, 8, 22].

Xia et al. [27] try to combat MITM attacks by developing a system which tries to give as much information to the user as possible when invalid certificates are encountered or when a password is about to be transmitted over an unencrypted connection. Due to the nature of SSL stripping attacks, the attacker does not have to present an invalid certificate in order to successfully eavesdrop the user, thus the part of their approach that deals with invalid certificates is ineffective against it. The part that deals with the un-encrypted transmission of a password can be of some use but can be easily circumvented using JavaScript or `iframe` tags as shown in Section 4.3.

9 Conclusion

Hundreds of thousands of websites rely on SSL daily to protect their customers' traffic from eavesdroppers. Recently though, a new kind of attack against the usage of the SSL protocol surfaced: SSL stripping. The power of such an attack is mainly due the fact that it produces no negative feedback, something that users have been unconsciously trained to search for as an indicator of a page's "insecurity".

In this paper we demonstrated that SSL stripping attacks are a realistic threat and presented a countermeasure that protects against them. This countermeasure, called HProxy, leverages the browser's history to create security profiles for each website. These profiles contain information about the use of SSL and every future load of that website is validated against that profile. Our prototype implementation of HProxy accurately detected all SSL stripping attacks with very few false positives. Our evaluation of HProxy showed that it can be used with acceptable overhead and without requiring server side support or trusted third parties to secure users against this type of attack.

10 Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven.

References

1. Hazim Almuhimedi, Amit Bhan, Dhruv Mohindra, and Joshua Sunshine. Toward Web Browsers that Make or Break Trust. In *Symposium Of Usable Privacy and Security (SOUPS)*, 2008.
2. BeautifulSoup Parser. `http://www.crummy.com/software/BeautifulSoup/`.
3. Yingying Chen, Wade Trappe, and Richard P. Martin. Detecting and Localizing Wireless Spoofing Attacks. In *in Proceedings of the Fourth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (IEEE SECON 2007)*, San Diego,CA, USA, 2007.

4. Chickenfoot for Firefox: Rewrite the Web. http://groups.csail.mit.edu/uid/chickenfoot/faq.html.
5. Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590, New York, NY, USA, 2006. ACM.
6. dsniff. http://monkey.org/~dugsong/dsniff/.
7. Manuel Egele, Marco Balduzzi, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A Solution for the Automated Detection of Clickjacking Attacks. In *Proceedings of ASIACCS, Beijing, China, April 2010*.
8. Batya Friedman, David Hurley, Daniel C. Howe, Edward Felten, and Helen Nissenbaum. Users' conceptions of web security: a comparative study. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 746–747, New York, NY, USA, 2002. ACM.
9. Fanglu Guo and Tzi cker Chiueh. Sequence number-based mac address spoof detection. In *RAID*, pages 309–329, 2005.
10. Suzuki Hisao. Tiny HTTP Proxy in Python. http://www.okisoft.co.jp/esc/python/proxy/.
11. Amit Klein. Cross Site Scripting Explained, Sanctum White Paper, 2002.
12. Patrick LaRoche and A. Nur Zincir-Heywood. Genetic Programming Based WiFi Data Link Layer Attack Detection. In *CNSR '06: Proceedings of the 4th Annual Communication Networks and Services Research Conference*, pages 285–292, Washington, DC, USA, 2006. IEEE Computer Society.
13. Moxie Marlinspike. New Tricks for Defeating SSL in Practice. In *Proceedings of BlackHat 2009*, DC, 2009.
14. Asier Martínez, Urko Zurutuza, Roberto Uribeetxeberria, Miguel Fernández, Jesus Lizarraga, Ainhoa Serna, and naki Vélez, I˙ Beacon Frame Spoofing Attack Detection in IEEE 802.11 Networks. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 520–525, Washington, DC, USA, 2008. IEEE Computer Society.
15. Corey Nachreiner. Anatomy of an ARP Poisoning Attack. http://www.watchguard.com/infocenter/editorial/135324.asp.
16. NetCraft. One Million SSL Sites on the Web. http://news.netcraft.com/archives/2009/02/01/one_million_ssl_sites_on_the_web.html.
17. Jesse Ruderman. JavaScript Security: Signed Scripts. http://www.mozilla.org/projects/security/components/signed-scripts.html.
18. Yong Sheng, Keren Tan, Guanling Chen, David Kotz, and Andrew Campbell. Detecting 802.11 MAC Layer Spoofing Using Received Signal Strength. In *Proceedings of INFOCOM 2008*, pages 1768 – 1776, 2008.
19. Alexander Sotirov. Heap Feng Shui in Javascript. In *Proceedings of BlackHat Europe 2007*, 2007.
20. The SSL Protocol. http://www.webstart.com/jed/papers/HRM/references/ssl.html.
21. Moxie Marlinspike's sslstrip. http://www.thoughtcrime.org/software/sslstrip/.
22. Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of Usenix Security*, 2009.
23. W.C. Suski, M.A. Temple, M.J. Mendenhall, and R.F. Mills. Using Spectral Fingerprints to Improve Wireless Network Security. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5, 30 2008-Dec. 4 2008.

24. Alexa Top 500 Global Sites. `http://www.alexa.com/topsites`.
25. Jesse R. Walker, Submission Page Jesse Walker, and Intel Corporation. Unsafe at any key size; An analysis of the WEP encapsulation, 2000.
26. Joshua Wright. Detecting Wireless LAN MAC Address Spoofing, 2003.
27. Haidong Xia and José Carlos Brustoloni. Hardening Web browsers against man-in-the-middle and eavesdropping attacks. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 489–498, New York, NY, USA, 2005. ACM.