COUNTING SHEEP

Analysing online authentication security

Author:	M. (Marc) Sleegers, BA (851663203)
Mentor:	dr. ir. H.L. (Hugo) Jonker
Client:	dr. ir. H.L. (Hugo) Jonker
Coordinator:	dr. ir. H. (Harrie) Passier
Examiner 1:	prof. dr. T. (Tanja) Vos
Examiner 2:	prof. dr. M.C.J.D. (Marko) van Eekelen

T61327 - Afstudeerproject bachelor informatica
Open Universiteit Nederland, faculteit Informatica

Open Universiteit
www.ou.nl

Zwolle, 26 March 2017

“*Facts do not cease to exist because they are ignored.*”

Aldous Huxley

COUNTING SHEEP: ANALYSING ONLINE AUTHENTICATION SECURITY

THESIS

in the context of obtaining the title of

BACHELOR OF SCIENCE

in

INFORMATICA

by

Marc Sleegers, BA

Open Universiteit Nederland
Faculteit Informatica
Heerlen, Nederland
www.ou.nl

Zwolle, 26 March 2017

Abstract

Modern websites store an authentication cookie on the client computer when the login process succeeds. This cookie allows the end user to remain authenticated for the remainder of the session, forgoing the need to supply credentials to each following request. An attacker can steal this cookie in a session hijacking attack, effectively authenticating as the victim without needing the username and the password.

As such, it is vital that authentication cookies are used securely over an encrypted connection. Firesheep showed that websites typically protected the login process, but dropped down to an insecure connection immediately after in 2010. While this secured credentials, it left the authentication cookie exposed. Following this, Firesheep allowed any attacker to trivially hijack sessions on sites such as Google and Facebook. The websites in the spotlight quickly implemented a secure connection across the board, but it is unknown if others followed suit.

Analysing how widespread “faux” online authentication security still is first requires a way to identify domains that are vulnerable to session hijacking. We conclude that this type of faux web security can be identified by analysing the authentication cookies of a site. During initial testing we found that the problem still exists today, despite the internet ecosystem appearing to be more secure. Additionally, we found that mobile apps suffer from the same vulnerabilities. Following these results, we developed a tool, Shepherd, which analyses a given number of domains using a pre-emptive and generic approach. Using this tool, we were able to automate the entire login process for 4689 unique domains without the need for prior information such as credentials. We found that four out of five authenticated domains (3764) are indeed still vulnerable to session hijacking.

Project Context

The following four main products were delivered in context of this graduation project: this thesis, the included individual research (Chapter 3), the proof-of-concept software tool (described in Chapter 4) and several presentations including – most importantly – the final presentation during which the conclusions to the research question were presented.

In the context of the graduation process, the following applies:

- The graduate had 400 hours in total available time for this project.
- The thesis had to be written in English to better serve the research community and to best function as a solid base for a follow-up research paper on the same topic. The writing and / or publication of this paper is not within this project's contextual scope.
- The primary goal of this graduation program was writing this thesis. The developed proof-of-concept tool was a means to this end.

This research stems from the project application 'Firesheep' by dr. ir. H.L. (Hugo) Jonker. The initial goal of the contract was to design and to develop a plugin like Firesheep for the Firefox web browser under the Linux operating system. However, during initial domain research this goal was deemed redundant as the functionality of Firesheep was already realised under Linux by a (official, albeit basic) fork made by the program's original author.

Furthermore, from the initial interview with the project applicant followed that the client was mostly interested in the extent to which the 'secure login then insecure' problem still exists. The client envisioned a plugin (Firesheep) working under the Linux operating system as a means to that end. Hijacking sessions – the main purpose of Firesheep – was the methodology. Another goal was to use the prototype plugin for demonstrations on security. The project was therefore reinterpreted in consultation with the client to best match the primary research goal: measuring the extent to which faux authentication security still exists.

This study and its results are a direct contribution to the overarching research of dr. ir. H.L. (Hugo) Jonker in the context area 'Security and Privacy' conducted at the Open Universiteit Nederland. The central question of Jonker's research is: "*How to bring the security and privacy assurances of the real world to their digital life equivalents?*" The results of this study will be used to partially conclude the state of security and privacy assurances of the digital life in the context of online authentication. Additionally, the developed prototype tool will be used to explore related topics such as the state of authentication security of mobile apps and will be used for demonstrations about authentication security on conventions.

This thesis, the developed tool and the results serve as direct input for the follow-up paper by dr. ir H.L. (Hugo) Jonker, B. (Benjamin) Krumnow MSc and M. (Marc) Sleegers BA.

For ethical reasons (Chapter 1.5) and at behest of the Open Universiteit Nederland legal department, a non-disclosure agreement (NDA) was signed between the graduate Marc Sleegers, BA and the Open Universiteit Nederland which further outlines ethical guidelines.

Contents

Glossary

Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
CSRF	Cross-Site Request Forgery
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HSTS	HTTP Strict Transport Security
MITM	Man-In-The-Middle
MVC	Model-View-Controller
SSL	Secure Sockets Layer
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WPA2	Wi-Fi Protected Access 2
XSS	Cross-site scripting

Definitions

Faux	Appearing to be so.
GET	HTTP GET request. Request data from a server.
Man-In-The-Middle	Secretly relaying communication between two parties who believe they are directly communicating with each other.
POST	HTTP POST request. Send data to a server.
Replay	Sending an HTTP request again, possibly altering its contents.
Session hijacking	See Chapter 2.3.1 of this document.
Sidejacking	See ‘session hijacking’.
Sniffing	Reading (and capturing) data as it is transmitted over a network (e.g. HTTP requests).
Wi-Fi	Wireless local area networking with devices based on the IEEE 802.11 ‘radio frequency’ standard.

Abbreviations and definitions that are mentioned one time only are explained in-line.

Introduction

More and more of life is lived online or has an online counterpart. Facebook, for example, is the digital equivalent of the social circle and online banking replaces paper money transfer forms entirely. Websites that provide user interactions like this typically require the user to provide valid credentials to both identify and authenticate. Because of the way the underlying protocol (Hypertext Transfer Protocol or HTTP) works, authentication cookies are used to locally store the fact that the user was successfully authenticated. Securely handling these cookies is vital. If they are not protected properly, a malicious attacker can steal them and use them to impersonate someone else. An attacker does not need full credentials for this attack to succeed, as the session cookies are enough to fully authenticate when no other measures are taken. This attack is called a session hijack (or sidejack), which has been a known vulnerability since the early 1990s [1] [2].

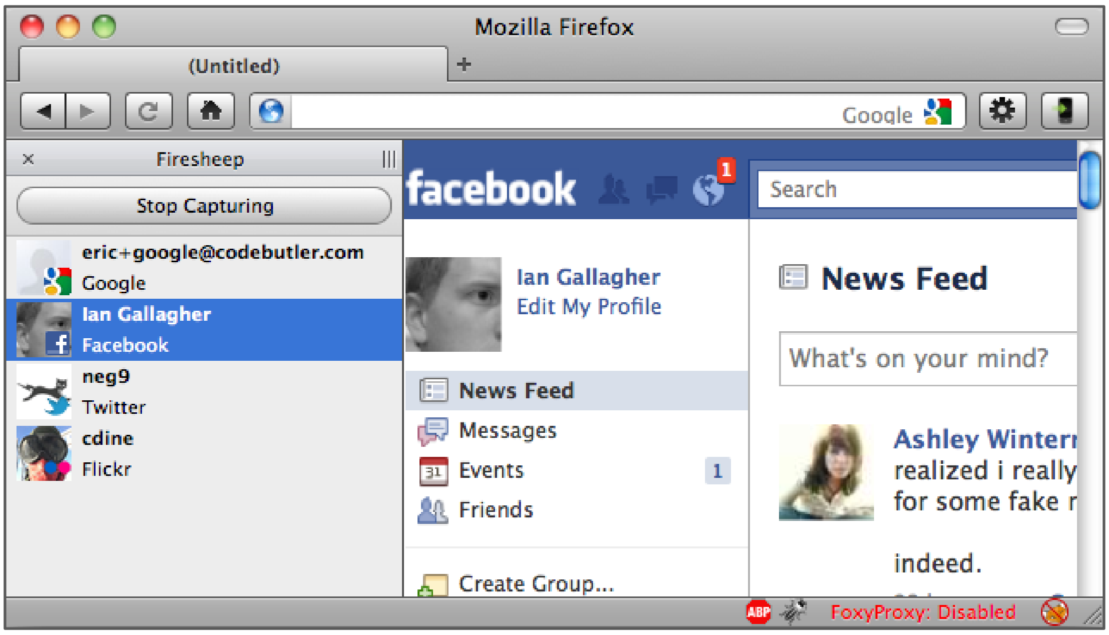


Figure 1: Firesheep in action at the Toorcon security conference in 2010 [3]

Security researcher Eric Butler notably demonstrated this type of attack when he introduced Firesheep in 2010 [3]. Firesheep is a plugin for Firefox used to trivialise sidejacking attacks over open wireless networks. Figure 1 shows Firesheep in action at the Toorcon security conference in 2010. The plugin specifically targeted major websites – including Amazon, Facebook and Twitter – which displayed 'secure login then insecure' behaviour. This means that the initial login is sent over a secure – costly – connection (HTTPS), but that authenticated visits drop to the unencrypted standard (HTTP) for the remainder of the session. As such, the credentials are sent in secret but the valuable session cookies are not.

Moreover, wireless networks typically lacked any type of security in 2010 [4] [5]. This was the cherry on top, as Firesheep exploits the combination of unprotected networks and insecure authentication cookies. Butler connected his laptop to a public network (e.g. in a train) that transmitted user requests as is. Using his easy-to-use tool Firesheep, he was able to monitor these requests and capture all insecure authentication cookies of supported websites. The results were disastrous: following the release of Firesheep, even non-technically inclined attackers were able to hijack sessions and to log in as someone else with a single click!

The uproar surrounding the release of Firesheep caused an upheaval about the priority of locking down the authentication process at the major websites targeted by Butler's tool. The companies responsible quickly implemented secure connections across the board [6] [7]. Those in the spotlight had their hand forced. The premise of this thesis is to investigate whether others promptly followed suit or are still in a state of faux authentication security.

1.1 Developments

In an increasingly digital world, it is more and more important that valuable data such as user accounts are kept safe. The security assurances of the real world must be brought to their digital life equivalents, but the internet as we know it is not the same as it was in 2010. The prevalence of mobile apps and the transition to 'HTTPS Everywhere', for example, are changing the playing field. The current state of web security across the board is unclear.

Only when the state of online authentication security is charted, can a solid attempt be made to make it as secure as meeting face-to-face. To conduct a practical exploration of this topic, a tool is needed to measure the current state of (faux) web security in the context of online authentication. Unlike Firesheep and similar programs which monitor available connections and use existing sessions retroactively, this tool needs to stand on its own and act pre-emptively. To this extent it is not necessary to act as a malicious man-in-the-middle, as providing valid credentials allows the necessary inspection of authentication cookies.

1.2 Goal

This study counts the sheep and measures how far online authentication security has really come since Butler released Firesheep in 2010. The research question is defined as follows:

How widespread is "faux" online authentication security still?

To answer this question, a custom proof-of-concept software tool was developed in order to measure the extent to which faux online authentication security is still a problem in 2017.

1.3 Contributions

While previous attempts have been made to follow in the footsteps of Butler's Firesheep, this is the first study that attempts to measure how widespread faux online authentication security still is using a pre-emptive approach. One major advantage of this approach is that it allows for analysis of fully secure connections (HTTPS) – an ability which previous tools (e.g. Firesheep) lacked. In Chapters 3.1 and 3.2 the fruit of this labour is described. This study resulted in the first quantitative set of data about this topic specifically.

To obtain this dataset, a prototype tool was developed that can automate the login process to pre-emptively analyse (i.e. without eavesdropping as a MITM) the security of online authentication. This tool is a contribution to existing scraping technology and serves as a modular, extendable framework for automating logins with scrapers. During initial (manual) local testing, more than a dozen vulnerable websites were identified. The detection of these vulnerabilities was handled responsibly, as described in Chapter 1.5 of this document. The fact that our tool can work on its own is a major advantage over similar studies (e.g. Newton) that use a reactive, non-generic approach. Newton, for example, is limited in both scope and in functionality because it requires user interaction for each unique domain.

Unlike Firesheep we do not intend to simplify session hijacking attacks for those who are less technologically inclined. Instead, our intention is for webmasters (or penetration testers) and not the attackers to use our tool to pinpoint issues before releasing a system or shortly thereafter. The ethical guidelines are described in detail in Chapter 1.5 of this document.

1.4 Project Scope

The security of online authentication is a very broad concept. This study focussed on analysing the online authentication security of websites only, in order to give a singular answer to the research question and to best match the project context. This excludes, for example, analysing the online authentication security of mobile apps which – while similar – is a self-contained topic and as such is reserved for further work. See also Chapter 4. While we initially wanted to analyse the Alexa top 1000000 domains, this number was soon scoped to the Alexa top 500000 due to the time constraints as discussed in the project context.

1.5 Ethics

On the surface, analysing a session cookie that is automatically stored on a computer while browsing the web may seem harmless. In reality, analysing such a cookie results in a conclusion about the state of (faux) authentication security for the relevant domain. This means that an attacker can conclude whether a website is vulnerable to session hijacking attacks by analysing the cookies. As such, automating this process for an arbitrary number of websites is a security risk. Using a program to list vulnerable domains results in a dataset that is very valuable to attackers with malicious intent. The dataset is like a roadmap to unlocked doors in this context.

Furthermore, to analyse the current state of (faux) web security it is not needed that the researchers know which specific domains are vulnerable to sidejacking. As such, we deem it best to avoid this knowledge.

To mitigate the security risk, several measures are taken during this study:

- Do not store any domain data related to authentication cookies.
- Tally the amount of vulnerable websites instead of listing them.
- Do not make the tool generally available.

Releasing the proof-of-concept tool as is results in a potential security risk. Because the developed proof-of-concept tool tallies vulnerable domains and has to visit a list of websites, an attacker only needs to insert a few print statements to obtain a list of potential targets. Therefore, we deem it not difficult enough for someone without technical knowledge to setup and to utilise the tool. We believe a malicious attacker targeting a specific domain already possesses the technical knowledge presented in this study. Our goal is neither to help these attackers find potential targets nor to create new attackers, taking into account the widespread use of the easy-to-install Firesheep. An open-source release is not ruled out, but researching what to release and which license to use is reserved for future work.

Vulnerable domains identified by testing or by manual labour were notified. To this extent, a letter was drafted (Appendix 4) which was sent to all domains identified as vulnerable. The supervisor of this thesis dr. ir. H.L. (Hugo) Jonker is in charge of this responsible disclosure.

1.6 Methods

First, the context of this study was defined in the research context. Second, a short literature study was conducted to conclude how faux web security can be identified. The conclusions were used to define the proof of problem. To this extent, a manual session hijacking attack was performed using the man-in-the-middle sniffing technique in a laboratory setting. Third, a short literature study was conducted in combination with technical testing to conclude how the developed proof-of-concept tool could automate the login process.

Following this research, the following approach was used to 'herd the sheep':

1. We identified test cases using existing techniques by analysing a handful of websites manually and performing manual session hijacking attacks to prove the problem.
2. We developed a proof-of-concept web scraper used to automatically login to websites using existing credentials. These credentials are sourced from the login provider BugMeNot, reducing the analysis problem other parties deemed too difficult to solve (Chapter 3.1.3) to its core: analysing authentication cookies.
3. Following local testing, we ran a limited field test to verify functionality.
4. Finally, we counted the sheep by analysing the Alexa top 500000 domains.

The methodology of this study was presented at the 2nd Cyber Security Workshop in the Netherlands on October 13th 2016 in the form of a poster presentation.

1.7 Overview

Chapter 2 will cover the research context of this study, which consists of a concise history of the session hijacking attack and related work in this field of research. In Chapter 3 the faux web security problem is defined. Using this knowledge, the existence of the problem is proved. Chapter 4 briefly covers the architectural design, the functionality and the implementation of the developed proof-of-concept tool. Additionally, the automation of the login process is covered in more detail. Following this, Chapter 5 covers the discussion of the research process. In Chapter 6 the conclusions to this study are presented. Finally, Chapter 7 covers future work for both the developed tool and the research.

Chapter 2

Research Context

In this chapter, the context of this study is described. Related research topics are described in chapters 2.1, 2.2 and 2.3 of this section. Respectively, the state of online authentication security before Firesheep, during Firesheep and beyond Firesheep is explained.

2.1 Before Firesheep

Security researcher Eric Butler put faux online authentication security (back) on the map when he released Firesheep in 2010. This problem, however, existed long before that.

2.1.1 HTTP

Hypertext Transfer Protocol (HTTP) is an application layer protocol intended for sending information across the internet [8]. HTTP is the foundation of the communication of data on the World Wide Web. HTTP is a stateless request-response protocol and functions in the client-server computing model [8]. This means that a client (e.g. a web browser) submits a request to a server (e.g. a web application).

```
GET http://www.marcsleegers.com/ HTTP/1.1
Host: marcsleegers.com
```

Figure 2: A very simple HTTP GET request

Figure 2 shows a simple example of a client request. The request is sent from the client to the server (host) in ASCII encoding (plain text). Additional information that is sent to the server includes the IP address of the client and possibly includes additional headers such as the user agent. The web server running on ‘http://www.marcsleegers.com/’ receives the request and replies.

```
HTTP/1.1 200 OK
Date: Sun, 1 January 2017 12:00:00 GMT
Accept-Ranges: bytes
Content-Type: text/html
Last-Modified: Sun, 1 January 2017 12:00:00 GMT
Server: Apache
Connection: close
```

Figure 3: Headers of a simple HTTP response

Figure 3 shows a basic HTTP response. The response is sent from the server to the client using the IP address the client added when submitting the initial request. The server replies with ‘text/html’ content, in this case the ‘index.html’ page. The status code ‘200’ indicates the request and the response were handled ‘OK’. The browser can now use the page it received to render the view for the user. Most of the other response headers are optional and additional headers such as the entity tag can be added.

As HTTP is a stateless protocol, the server is not required to retain any information about this transaction [8]. As such, the protocol inherently does not support sessions. This means that the next time the same user submits a request the server does not know it met the client before if no other measures are taken.

2.1.2 Cookies

Sometimes it is useful if the server remembered it interacted with a specific client before. One such example is managing a shopping cart. Another example is managing user authentication. Lou Montulli, while working at Netscape, thought the same in 1994 [9]. Montulli added his idea of state to the stateless HTTP in the form of 'cookies'. Web browsers have used cookies ever since to repeatedly and reliably identify themselves to servers. Cookies are, in a way, used as name tags to remind servers that previous interaction had happened. This state of remembrance in a continued interaction is called a session.

All session management mechanisms follow the same basic pattern, to illustrate [1] [4]:

1. The client submits a HTTP request.
2. The server starts session management, generating a unique session identifier.
3. The server replies to the client, setting the session cookie in the 'Set-Cookie' header.

The client now knows what session identifier the server uses to remember the client. The web browser adds this identifier to the 'Cookie' header when submitting subsequent requests, continuing the session. As such, the server may reply with a tailored response.

Initially, cookies were used as a way for end users to hold items in a virtual shopping cart as they navigate through an online store. Today, however, shopping carts are typically stored in a database on the server. Unique session identifier cookies – the process of which is outlined in the pattern above – are then used to match a user with rows in the database.

Other uses of cookies include user behaviour tracking, personalization and tailored advertisements [10]. Session management based on authentication is a specific use relevant to this study. When the client successfully authenticates to the server using valid credentials, the server remembers that particular session identifier as being authenticated. Subsequent requests can use the same session identifier to re-authenticate in favour of having to re-enter credentials every time a page is visited. This happens, to illustrate, when a user clicks a link on Facebook to send a message to one of their friends. In the context of this study, a session cookie is as valuable as user credentials as both are used to complete authentication.

As the use of cookies evolved, so did the cookies themselves. Cookies were initially used by supplying basic attributes such as a domain, a name and a value. For example, a cookie called 'session' could contain the session identifier token for a specific domain. Today, additional attributes (or 'flags') can optionally be set [1]:

- The ‘Expires’ attribute invalidates the cookie on a given date. This is useful over setting the ‘Max-Age’ flag, when a set of cookies should invalidate on a set date.
- The ‘Max-Age’ attribute limits the age of the cookie. This is useful over setting the ‘Expires’ flag when, for example, a specific session is only allowed to exist for a while.
- The ‘Domain’ attribute limits the scope of the cookie to the given domain.
- The ‘Path’ attribute limits the scope of the cookie within the scope of the domain.
- The ‘Secure’ attribute limits the use of the cookie to HTTPS.
- The ‘HttpOnly’ attribute blocks the cookie to, for example, scripting languages.

The usefulness of these cookie attributes, in particular the flags ‘Secure’ and ‘HttpOnly’, are discussed in detail in Chapter 3 of this document. Note that it is up to the client browser to implement this functionality as cookies are stateless. For example, it was technically possible for servers to set the ‘Secure’ flag of a cookie when sending a response over insecure HTTP until protocol update RFC 6265 (i.e. before Firefox 52 released in March 2017) [1].

Additional cookie attributes are suggested periodically. The attribute ‘SameSite’ is currently flagged as a work in progress after being suggested by a joint-venture between Google and Mozilla [11]. The intention of this flag is to limit automated requests (i.e. not initiated by the user) to the same domain, preventing request forgery [12]. The draft version of this flag is currently implemented in Chrome (51) and Opera (39) as of April 2016 [13]. The Firefox development team is still actively working on implementing the (draft) cookie attribute [14].

2.1.3 Session hijacking

If an attacker obtains the session (authentication) cookies of another user, the attacker can bypass authentication and, as such, impersonate the victim without the need of valid user credentials. We use the term session hijack (or sidejack) to denote this attack. Sidejacking is an attack that exploits the inherent vulnerability of cookies used to uniquely identify sessions [1] [2] [12] [13]. In such an attack, an attacker typically monitors unencrypted networks for traffic such as HTTP requests for visible session cookies. As HTTP requests and responses are transmitted in plain text, an attacker can view and copy the cookies contained in the HTTP headers. The attacker can then load the stolen cookies into their own web browser, effectively authenticating as the victim. If the attacker copies the intercepted request exactly, it is called a replayed request.

Because sidejacking exploits properties inherent to cookies and HTTP, this attack has been possible since the introduction of session management mechanisms that utilise cookies [4]. As such, session hijacking over HTTP using session cookies is theoretically possible today.

Initially, an attacker with network access would have to act as a man-in-the-middle – intercepting and replaying the entire connection – in order to steal cookie data. The availability of public, open, and often unencrypted wireless networks, however, dramatically increased the ease of the attack [4]. When connected such a network (e.g. in a train), an attacker is only required to be able to intercept Wi-Fi signals broadcast from the target device. This is called monitoring. Laptops typically have a special ‘monitoring mode’ to this day, in which the laptop listens to all the traffic on the wireless network it is connected to.

Unencrypted traffic, such as HTTP requests, can then easily be read by a malicious third-party – including cookies.

Sidejacking attacks were initially performed 'manually' as attackers analysed and captured HTTP requests for specific domains – typically fishing for user credentials or session cookies [4]. At the Black Hat security conference of 2007 security researcher Robert Graham demonstrated his tool Ferret and Hamster [14] [15]. Graham's tool simplified the capturing and replaying of HTTP requests on wireless networks. Ferret, the first half of the program, sniffs all data passing through a networking interface in monitoring mode and saves any HTTP cookies it finds to a local database. Hamster then allows the attacker to view all collected cookies and to inject them into their own web browser. While the tool automated the collection of data, 'manual labour' was still needed to separate the coal (traffic noise, e.g. tracker cookies) from the diamonds (sessions) – limiting its usefulness.

2.1.4 HTTPS

Because session (authentication) cookies can be used in place of user credentials in the context of this study, it is vital that they are protected from attackers. Hypertext Transfer Protocol Secure (HTTPS) is a protocol intended for securely sending information across the internet [16]. HTTPS combines communication using HTTP with a connection encrypted by Secure Sockets Layer or its successor Transport Layer Security. Netscape, which also introduced cookies, created HTTPS in 1994 as a way to protect sensitive transactions in corporate information systems. Configuring a server to use HTTPS comes with additional operating costs (e.g. in server load), because of the required encryption of traffic [4] [6].

HTTPS provides both authentication and bidirectional encryption, which protects against man-in-the-middle and sidejacking attacks if properly implemented [17]. This means that the client and the server know they are communicating with each other and that the transmitted data has not been seen or altered by any third-party. As such, the privacy and the integrity of the messages are guaranteed. Does this mean sidejacking is impossible?

2.2 Firesheep

In an increasingly digital world, more and more information is stored on the internet. In 2010, for example, Facebook was already the most popular social network and millions of people shopped online in the Amazon web store. The developments in HTTPS and the increasing value of online data drove most top sites requiring authentication to implement HTTPS for the authentication of user credentials. As such, the username and the password required for the initial login are protected from session hijacking attacks if the secure connection is properly configured. These sites, however, typically displayed 'secure login then insecure' behaviour. This means that while the initial login is sent over a secure connection (HTTPS), following authenticated requests are sent over the unencrypted standard (HTTP) for the remainder of the session. This happens, because HTTPS is only implemented for the authentication process. As a result, the credentials (e.g. the password) are sent in secret but the valuable authentication cookies are not.

Eric Butler notably demonstrated this when he introduced Firesheep in 2010 [3]. Firesheep is a plugin for the Firefox web browser used to trivialise session hijacking attacks over open wireless networks. Firesheep specifically targeted major websites – including Amazon, Facebook and Twitter – which displayed 'secure login then insecure' behaviour. Firesheep was able to specifically target supported sites by means of pre-configured handlers. As such, adding additional handlers increased the number of supported sites.

Wireless networks typically lacked any type of security in 2010 [4]. This was the cherry on top for Butler, as Firesheep exploits the combination of unprotected networks and insecure authentication cookies. Butler monitored all HTTP requests on a public wireless network. Firesheep was able to capture all insecure authentication cookies of supported websites. The results were disastrous: following the release of Firesheep, even non-technically inclined attackers were able to hijack sessions and to log in as someone else with a single click!

Figure 1 (Chapter 1) shows Firesheep in action. Unlike Ferret and Hamster which functioned as a 'catch-all', Firesheep supported a few major websites out of the box by means of handlers. The lists of targets included Amazon, Facebook, Twitter and Google [3]. Additionally, Firesheep targeted security researchers specifically by also including HackerNews, GitHub and the website of Toorcon itself in the list of targets. All of these sites displayed 'secure login then insecure' behaviour in 2010 and, as such, were 'Firesheepable'.

This specific list of targets was both a blessing and a curse for Firesheep. On the one hand, it improved on the 'no-filter' implementation of Ferret and Hamster and fulfilled its goal as a sidejacking tool for a handful of websites. On the other hand, extending Firesheep beyond its initial list of targets required manual labour. Firesheep filtered all monitored HTTP data using handlers written in JavaScript. Additionally, using this handler, Firesheep would scrape the page of the relevant website for content about the user. Figure 1 (Chapter 1) shows the addition of the username and the user profile picture for the identified victims as a snippet.

Firesheep displayed any potential victims as snippets in the Firefox sidebar. A single click completed the session hijacking attack and authenticated the attacker as the victim. Because of its easy-of-use – even for non-technically inclined attackers – Firesheep quickly attracted the attention of the mainstream media [6] [18] [19] [20]. The uproar surrounding the release caused an upheaval about the priority of locking down the authentication process at the major websites supported by the plugin. The targeted companies responsible quickly implemented secure connections across the board [6] [7] [21].

Butler responded to the attention of the media with a statement on his blog [6]: "*Because of its simplicity, Firesheep has already succeeded in demonstrating the risks of insecure websites to a much wider audience than any previous tool, in a single day. Many companies make a business, not technical, decision to not implement security due to either perceived or actual costs. It is our opinion that turning a blind eye to customer privacy and security is never a good business, and we hope the people making these decisions will begin to agree.*"
Butler described Firesheep as more of a proof-of-concept tool than as a full fletched security auditing tool [6]. After releasing the source code, Butler discontinued the project. The tool quickly became outdated, being compatible with only 32-bit Windows and older versions of OS X in addition to requiring an old installation of Firefox 3.

2.3 Beyond Firesheep

Firesheep showed the importance of securely handling online authentication cookies. As a result, the companies the plugin specifically targeted were quick to solve the problem at hand. The internet as a whole, however, is way larger than a handful of major websites. Did the rest of the internet follow suit?

2.3.1 Modern day session hijacking

Over the years, several attempts have been made to follow in Firesheep's footsteps. In a first effort, a bare bones implementation of Firesheep was ported to work under the Linux operating system [22]. The CookieMonster project expanded on this by including a basic built-in tool for a man-in-the-middle attack using ARP spoofing, but otherwise suffered from the same problems as Ferret and Hamster with the catch-all approach [6].

The tools FaceNiff and DroidSheep have similar functionality to Firesheep, but leave the desktop environment in favour of the Android mobile operating system [4]. Just requiring root access to the phone, both apps significantly simplify the monitoring process. Both tools, however, have the same limits as Firesheep – supporting only a handful of sites and only analysing data sent over HTTP. Matthew Sullivan created Cookie Cadger as a 'spiritual successor' to Firesheep in an attempt to expand on requiring hard-coded support for a limited number of websites by making his tool domain agnostic [4]. Additionally, Sullivan made Cookie Cadger inherently cross-platform using the Java programming language combined with cross-platform libraries.

Technically, identifying and capturing HTTP requests and responses in Cookie Cadger works similarly to Firesheep. The attacker has to set up their wireless network card in monitoring promiscuous mode, enabling Cookie Cadger to sniff the networking interface. Cookie Cadger uses the cross-platform TShark binary, which is part of the Wireshark auditing suit, to obtain the raw HTTP data. Cookie Cadger expands on the catch-all implementation of Ferret and Hamster by automatically grouping requests by MAC-address and domain. While entire 'snapshots' can be replayed, however, manual analysis is still needed to identify session authentication cookies.

Having identified a session cookie, the attacker can store its name and characteristics in a local database. Essentially mimicking Firesheep's implementation of 'handlers', Cookie Cadger allows on-the-fly creation of handlers – also written in JavaScript – to simplify the analysis of authenticated requests from the same domain. Unlike Firesheep, however, Cookie Cadger does not need these handlers to function. As such, the tool is essentially a modern Ferret and Hamster with real-time filtering.

Finally, it is worth noting that performing session hijacking attacks 'manually' has been made significantly easier with the availability of security auditing tools such as Wireshark [23]. Before Wireshark, the standard application tcpdump was the default method for intercepting session cookies [4]. Using these utilities, concludes Sullivan, can be cumbersome to set up and requires extensive knowledge to filter out noisy data that is not needed to perform session hijacking attacks [4].

All of the described tools have the same malicious goal: capture an authentication cookie and use this cookie in a replay HTTP request in order to hijack a session. As all tools work reactively, none however, can be used to pre-emptively analyse the security of a domain in the context of the security session authentication cookies. Additionally, all of the described tools function only by intercepting HTTP traffic. As such, none of the tools include the analysis of secure traffic sent over HTTPS. Only when properly configured does HTTPS protect against session hijacking attacks [17].

2.3.2 The internet ecosystem in 2017

The real story is not the success of Firesheep, but the fact that the attack was still possible. As such, Butler stated that his tool is only truly successful when it no longer works at all [6]. Since the release of Firesheep in 2010, it appears that a lot of progress has been made to this extent. Popular websites (e.g. Facebook) announced site-wide SSL, WPA2 is now the de facto standard in Wi-Fi security and Google's push for HTTPS everywhere aims to ensure all websites use encrypted HTTPS across the board.

Sullivan described the internet ecosystem in 2010 as a 'perfect storm' for Firesheep to operate in [4]. Butler, too, explains the success his plugin as a combination of an increasingly digital society on the one hand and business (cost) versus security decisions on the other hand [6]. The internet ecosystem today, however, is not the same as it was in 2010. As of September 2016, 11.7% of the Alexa top one million websites use a secure connection as the default, meaning HTTPS is implemented across the board [24] [25]. The Let's Encrypts initiative reports that, as of June 2016, about 45% of page loads through the Firefox web browser use HTTPS [26].

Butler and Sullivan agree that most providers refuse to transition to fully encrypted connections, because of the additional cost and capacity (e.g. server load) that comes with encryption and decryption of traffic [4] [6]. Other reasons HTTPS is not as widely adopted include traffic shaping and traffic caching, both of which are ways to further reduce the operating cost of running a website [27]. From neither statistic can we conclude, however, if online authentication security is still faux as all login data of a website may in fact be served over HTTPS only. The different 'levels' of HTTPS implementations and the effects on online authentication security are described in detail in Chapter 3 of this document.

It is clear, regardless, that the amount of seemingly secure websites is increasing. One reason may be that Google recently decided to rank HTTP-only websites lower than those supporting HTTPS when serving search results [28] [29] [30]. While this is a direct incentive for companies to implement secure connections, this may in fact harm the overall state of web security. Webmasters may hastily or incorrectly implement SSL in order to meet the new standards and a such create a state of faux web security. In fact, Mozilla stated 'secure login then insecure' website behaviour still happens 'too frequently' just last year when the company pressed the importance of securing web forms when serving them over an otherwise secure connection [31] [32] [33]. This type of 'security' fails to protect the end users after authentication, leaving them vulnerable to programs such as Firesheep. In other words: faux web security or 'sheep' that must be herded.

Finally, a related problem is the online authentication security of mobile apps. While out of the scope of this study as described in Chapter 1.4, the prevalence of mobile apps is a game changer in the internet ecosystem when compared to the digital landscape 2010.

2.3.3 Related Work

This study references related work throughout this document. The scientific sources referenced and used as context are briefly outlined here. Additionally, this study references the implementation details of the protocols discussed (e.g. HTTPS). These implementations are described in the 'Request for Comments' (RFC) standard proposals and updates by the Internet Engineering Task Force (IETF). Specifically the RFC's 2616 (HTTP, 1991), 2818 (HTTPS), 6265 (HTTP State Management Mechanism), 6797 (HSTS) and 7231 (HTTP, 2014) are referenced widely throughout this document [1] [8] [16] [34] [35].

There is no scientific report discussing Firesheep specifically. Butler introduced the plugin in a blog post, stating Firesheep only exploits known vulnerabilities [3]. As such, Firesheep was not a research project per se. The tool is, however, widely referenced in the scientific literature described in this chapter and is often used as an example of what can go wrong if security is disregarded as an afterthought. Two similar projects, in contrast, are in fact the result from scientific research. Graham introduced Ferret and Hamster in 2007 [14] [15]. While its usefulness was limited (Chapter 2.1.3), the tool is regarded as the precursor to Firesheep. Because it builds upon earlier work by Graham and Butler, Sullivan's Cookie Cadger is described as the 'spiritual successor' to Firesheep and was introduced in 2012 [4].

The protocol that Firesheep and similar attacks exploit are heavily debated in scientific publications. The implementation details and the vulnerabilities of cookies are widely discussed in particular [12] [13]. Kristol discusses the privacy issues inherent to the use of cookies [9]. He argues that cookies turned into a political flashpoint (e.g. 'cookiewet', Chapter 3.2.1), while the vulnerabilities are a technical – not a social – problem to solve. Dacosta et al. conclude that the inherent security weaknesses of cookies are too challenging to overcome, instead opting for 'one-time cookies' as a more robust alternative [36]. They describe their alternative implementation as inherently resistant to session hijacking. Liu et al. suggest a stateful cookie protocol that handles authentication and integrity itself [37].

While the alternatives looked promising initially, they were widely criticised. Noubir, Raylan and De Ryck all argue that the security of the alternative protocols – and of similar options – require a full HTTPS configuration that cannot fail during the initial exchange of secrets and that assuming this is impossible [38] [39]. Stating that a system is only as strong as its weakest point and adding that alternative approaches are not robust to other attacks (e.g. phishing and Cross-Site Request Forgery), Noubir and Raylan conclude that secure session management cannot be built using alternative approaches to cookies.

Hodges and Steingruebl state that, as such, more research is needed to protect the current implementation of stateless cookies [40]. They conclude that there is an increasing need of coherent web security policy frameworks specifically. Thus, assuming cookies are indeed inherently insecure, session cookies must be protected by protecting their use. Directly related is session fixation: an attack in which an attacker forces the user's session identifier beforehand thus negating the need to capture it after authentication. Kolsek argues that this is a problem with the implementation of cookies, suggesting that web servers should always generate fresh session identifiers after a successful authentication [41]. Schrank et al. conclude that session fixation is a 'forgotten vulnerability' standing in the shadow of more pressing attacks (e.g. XSS) but that session fixation is indeed a severe attack [42].

Firesheep exploited the fact that many wireless networks typically lacked any type of security in 2010. Beck and Tews conclude, however, that a nearby attacker can possibly eavesdrop on a user's unencrypted internet traffic, such as HTTP, regardless of whether or not the local wireless network itself is secured. [43] Freely available wireless sniffing toolkits (e.g. Wi-Fi Pineapple) enable such attacks, greatly simplifying the scanning and interception of wireless traffic. A passive network attacker (e.g. sitting in a coffee shop) can then steal session cookies and possibly hijack the user's session. Barth and Jackson add that a proper configuration of HTTPS is a necessity in this case, as imperfect networks cannot be trusted by themselves [44]. They suggested that client browsers should force HTTPS as early as 2009, which can now be done with plugins such as HTTPS Everywhere. Byrd, too, states that wireless networks cannot be trusted to protect unencrypted HTTP traffic from outsiders, even if the network itself is secured [45]. He concludes that end users should treat wireless (secure) hotspots as hostile compromised networks in regards to unencrypted traffic.

The security of encrypted traffic (HTTPS) is widely discussed, too, however. Sheffer and Holz summarised the known attacks on HTTPS, concluding [46]: "*Attacks always get better; they never get worse.*" They state that attacks discussed (e.g. SSLStrip) circumvent the added security of encrypted traffic, allowing for exploitation of cookies (e.g. sidejacking) after a successful attack. Moller et al. specifically discussed the POODLE attack, which they state attacks effectively secure HTTPS connections by forcing a downgrade to an earlier – known to be insecure – version of the protocol [47]. Jesudoss et all. outline possible defences, concluding that the authentication of web applications can be attacked on several levels [48]. Ristic specifically mentions the security of session authentication cookies as a best practice, stating that improper configuration undermines a properly configured HTTPS connection [49]. In Chapter 3 of this document, the security of cookies is discussed in detail.

Mundada et al. indexed authentication security for a small number of domains by analysing authentication cookies in 2016 [50]. They concluded that the authentication of 29 out of their 45 test cases could be exploited. Their approach, however, is reactive and non-generic. The proposed analysis tool, Newton, requires that a user logs in using valid credentials. As such, it only analyses domains that an authenticated end user specifically visits. Additionally, the tool requires custom handlers for each domain it supports. The handler is used to attempt to find a user's username in the page source code as a means of concluding whether or not a user is successfully authenticated. Mundada concludes that this is a severe limitation, stating that not all sites show a user's username when authenticated. The main goal of their tool, however, is not widespread analysis as they instead target site developers.

This study has a direct topical relationship with the to-be-published research paper "*Man-in-the-middle attacks evolved… but our security models didn't*" within the research area 'Security and Privacy' of the Open Universiteit Nederland [51]. In this paper, Jonker et al. conclude that merely securing the contents of a message is not enough to secure the message. Likewise, it is not enough to secure the initial login when authenticated HTTP requests drop down to an insecure connection. Jonker et al. state that man-in-the-middle attacks are evolving rapidly, because of dated security protocols. Eric Butler notably demonstrated an attack exploiting this when he released Firesheep in 2010 [3].

Other research projects which Jonker supervises are related in the context of scraping the internet and analysing the collected data. Krumnow, firstly, investigates online price discrimination and uses scraping to collect his data [52]. Vlot researches the arms race between ad blockers and advertisers [53]. She asks if servers differentiate between manual and automated web requests. Drazner et al. research a related topic and ask which party is winning the arms race between ad blockers and advertisers [54].

Chapter 3

Research

This study counts the sheep and measures how far online authentication security has really come since the release of Firesheep in 2010. The research question is defined in Chapter 1.2:

How widespread is “faux” online authentication security still?

In the context of this study ‘faux authentication security’ is defined as an authentication process in which session authentication cookies are not transmitted exclusively over a secure channel (i.e. HTTPS). One example of such faux security is ‘secure login then insecure’ behaviour. To answer the research question, a custom proof-of-concept software tool was developed in order to measure the extent to which faux online authentication security is still a problem in 2017. Before this tool could be developed, however, it was first necessary to know how the problem can be measured. As such, the question of this chapter is as follows:

How can faux online authentication security be measured?

To answer this question, the following two sub-questions are defined:

1. **How can faux web security be identified?**
2. **Does faux online authentication security still exist today?**

These questions are answered in Chapter 3.1 and Chapter 3.2 respectively. The conclusions to this chapter were directly used as input for the proof-of-concept tool.

3.1 Detecting faux web security

To measure the extent to which faux online authentication security is still a problem in 2017, it is first necessary to define ‘faux web security’ in the context of this study. To this extent, the following research question is answered in Chapter 3.1:

How can faux web security be identified?

To answer this question, the definition of ‘faux web security’ is explained first. Afterwards, the three identified types of such faux security are described. From this, the method of measurement is distilled. Finally, the existing evaluations of (faux) web security are outlined.

3.1.1 Faux web security

Firesheep highlighted the ‘secure login then insecure’ problem specifically, but this is not the only type of faux web security. A full definition is needed to conclude how widespread such faux security still is. Extending on the single use case of Firesheep, three types of faux authentication security are distilled from the protocol behaviour in the context of this study.

Connection	Login (credentials)	Session	Cookies
I. HTTP → HTTP	INSECURE	INSECURE	INSECURE
II. HTTPS → HTTP	SECURE	INSECURE	INSECURE
III. HTTPS → HTTPS	SECURE	SECURE	SECURE?

Table 1: The three types of faux authentication security

Table 1 shows the three distinguished types of faux web security. This definition of faux authentication security is complete, because it encompasses all combinations of secure and insecure connections. The third type of faux authentication security is fully secure in theory, but from the research context follows that proper implementation is needed in practice.

Faux web security I: session hijacking over HTTP

The first type of faux web security is the base case in which both the initial and authenticated requests are transmitted using standard insecure HTTP. From the research context (Chapter 2) follows that the session hijacking attack exploits properties inherent to the combination of HTTP and session authentication cookies. As such, sidejacking has been possible since the introduction of session management mechanisms that utilise cookies and is still possible when using only HTTP traffic today [1] [4].

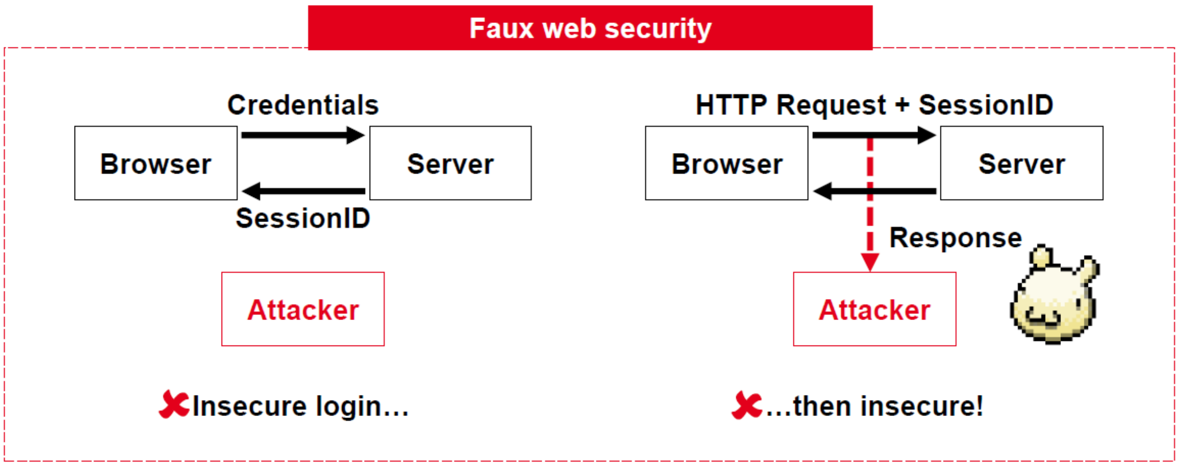


Figure 4: Session hijacking over HTTP explained

Figure 4 illustrates the session hijacking attack when only HTTP is used. The attacker, displayed in red, can capture the HTTP response sent by the web server during the initial login. This means the attacker can authenticate as the victim even if the victim never sends another request again. This also means that the attacker can sniff the user credentials, which is a totally different security issue – and the reason HTTPS was introduced in the first place. Using the captured credentials, the attacker is able to log in as the user without the need of a sniffed session – possibly locking the user out of their account. In the context of this study, however, the cookie is the most interesting as it is needed to perform the sidejacking attack.

Faux web security II: exploiting ‘secure login then insecure’ behaviour

The definition of the ‘secure login then insecure’ problem follows directly from the description of the Firesheep plugin for Firefox in the research context (Chapter 2).

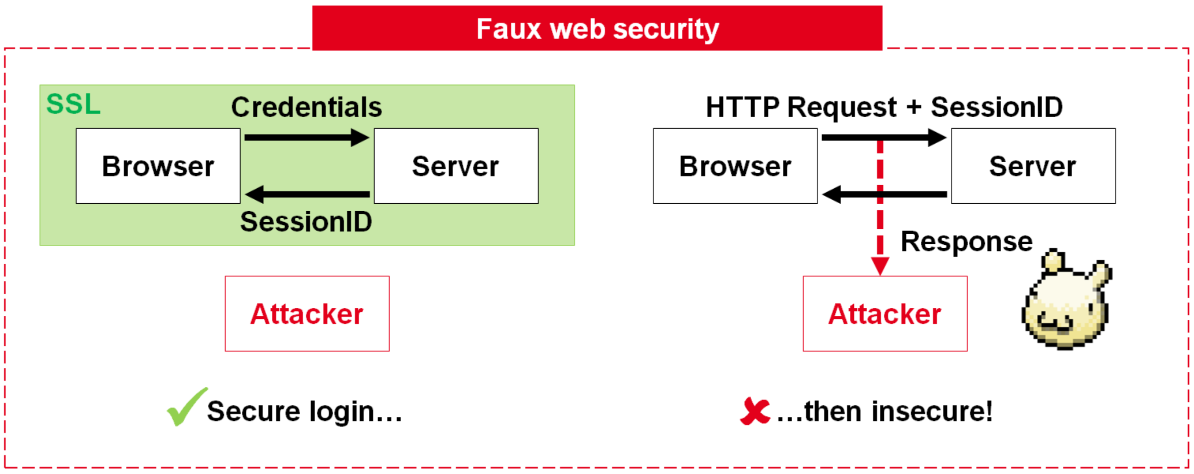


Figure 5: The 'secure login then insecure' problem explained

Figure 5 illustrates the 'secure login then insecure' behaviour websites typically displayed when security researcher Eric Butler released Firesheep in 2010. In short, a website protects the user credentials by transmitting the initial login over a secure connection (HTTPS). As Figure 5 shows, both the client request (Credentials) and the server response (SessionID) are encrypted. The attacker, again displayed in red, cannot capture either. Authenticated visits, however, drop down to the unencrypted standard (HTTP) for the remainder of the session.

As Figure 5 shows, the client sends an HTTP request to the server which includes the session authentication cookie. The attacker is able to sniff this traffic, thus capturing the cookie, as the request is transmitted in plain text for all to see. The following session hijacking attack authenticates the attacker, without the need to steal the credentials (e.g. the password).

Faux web security III: session hijacking over HTTPS

The third type of faux web security is seemingly the most secure. From the research context (Chapter 2) follows that the sites Firesheep targeted were quick to implement HTTPS across the board in an attempt to solve the 'secure login then insecure' problem. This, however, only protects the user from session hijacking attacks if HTTPS is properly configured [17].

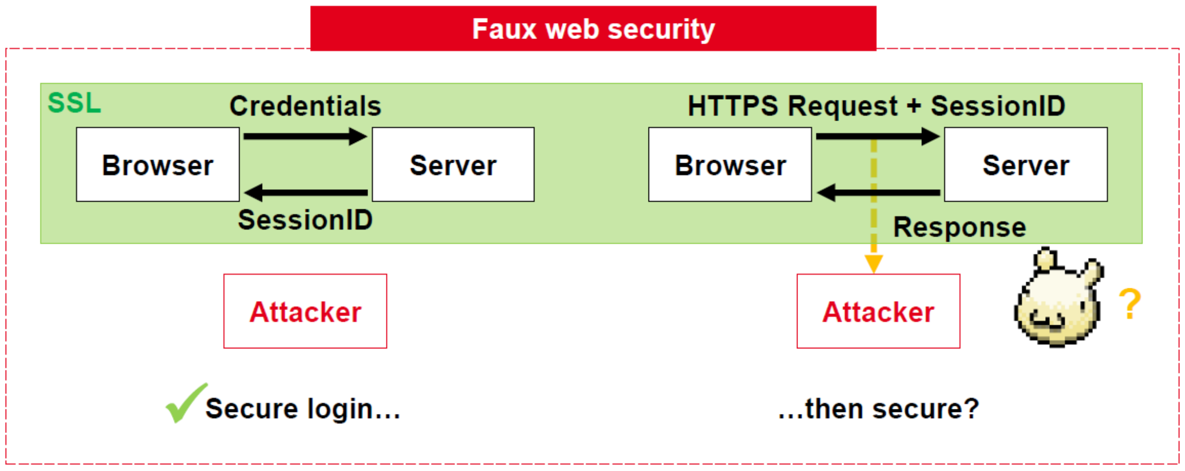


Figure 6: Session hijacking over HTTPS explained

Figure 6 illustrates the session hijacking attack when a secure connection is used across the board, but HTTPS is configured incorrectly. Unlike Figure 4 and Figure 5, which show the sidejacking attack is definitely possible at some part in the process, Figure 6 shows the attack might be possible – or might not be. Depending on the configuration of HTTPS, the site can be vulnerable to the session hijacking attack during some parts of the process but can be effectively invulnerable to this type of attack in other parts. While this scenario is seemingly more secure than 'secure login then insecure', the security is faux. Only when the session cookie itself is properly secured is authentication secure.

Bar the possibility of attacks on the integrity of the secure HTTPS connection, improperly configuring the security of the session authentication cookie(s) can lead to vulnerability to session hijacking attacks. All cookies are – by default – not protocol specific [1] [9]. This means that a cookie set on a website requested over HTTPS (which is secure), will also be available to the same website served over HTTP (which is insecure) if no other measures are taken to enhance security. This means that, if the user requests a website over HTTP, the situation changes from the one illustrated in Figure 6 to the one illustrated in Figure 5 – exposing the session cookie even if a secure HTTPS connection is used across the board.

From the above follows that if an active network attacker were to trigger the user to connect to the site via HTTP, the improperly secured session cookie will be transmitted in plain text. As such, the attacker is then able to capture the session authentication cookie by reducing the attack to exploit the 'secure login then insecure' problem illustrated in Figure 5. This is not a direct problem of SSL/TLS, but it does affect its practical application nevertheless.

To prevent this from happening, the 'Secure' cookie flag was introduced [1] [49]. This attribute limits the scope of the cookie to secure connections (HTTPS). A web server can specify the 'Secure' flag when setting the cookie when sending an HTTP response to the received HTTP request, much like the server can set the name and the value. Unlike the name and the value, however, the 'Secure' flag is optional. When a cookie has the 'Secure' attribute set, the client will only include the cookie in a request if the request is transmitted over HTTPS. Thus, setting the 'Secure' flag for a session authentication cookie prevents the cookie from leaking when the website is requested over insecure HTTP. Improperly protecting important cookies by omitting the 'Secure' flag is one of the main threats defined in the SSL Threat model of 2016 [55].

Although seemingly adequate for protecting session authentication cookies, the 'Secure' flag only protects the confidentiality of the cookie [1]. An active network attacker can overwrite the flag from an insecure channel, disrupting the integrity. This applies when, for example, the subdomain 'foo.marcsleegers.com' sets a secure authentication cookie for the domain '.marcsleegers.com' over a secure HTTPS connection. The domain attribute is loosely defined (LDD) [1]. If the subdomain 'bar.marcsleegers.com' sets the same cookie for the same domain over an insecure HTTP connection, the cookie with the 'Secure' flag set will be overwritten. If the attacker figures this out, the insecure cookie can be manually set as being 'secure' – thus enabling access to the secure subdomain because the cookie has a loosely defined domain. This theoretical edge case was first illustrated by The Internet Engineering Task Force (IETF) to explain the weak integrity property of cookies and has no practical value in the context of this study [1].

Setting the ‘Secure’ attribute is not a failsafe approach to protect the session authentication cookie(s). Cookies are – again, by default – readable by scripting languages (e.g. JavaScript) in the browser. This means that if an attacker manages to run a script in the browser of the victim, the attacker can steal all cookies from the target domain. The malicious script is able to read all local cookies stored on the victim’s computer and is able to send these cookies to another computer controller by the attacker. This is called cross-site scripting (XSS) [56] [57]. Cross-site scripting is one of the main threats defined in the SSL Threat model of 2016 [55].

The ‘HttpOnly’ cookie flag is used to prevent cross-site scripting attacks [1] [57]. This attribute limits the scope of the cookie to HTTP requests, meaning it is inaccessible to scripting languages running in the browser. The existence of the attribute specifically instructs the client to omit the cookie when providing access to cookies through an API that does not use HTTP. This means that scripting languages such as JavaScript are not able to read the local cookie thus effectively preventing a cross-site scripting attack. A related flag is ‘SameSite’, which is currently flagged as a work in progress after being suggested by a joint venture between Google and Mozilla [11]. The intention of this flag is to limit automated requests (i.e. not initiated by the user) to the same domain, preventing cross-site scripting and request forgery (CSRF) attacks.

Both the ‘HttpOnly’ and the ‘Secure’ cookie flags limit the scope of the cookie. Arguably both attributes must be set – when relevant – to properly protect important cookies such as session authentication cookies [1] [49] [58] [59]. In the case where the ‘HttpOnly’ flag is set and the ‘Secure’ Flag is omitted, the cookie can be sniffed by performing a downgrading attack. In the case where the ‘Secure’ flag is set and the ‘HttpOnly’ flag is omitted, the cookie can be captured by performing an XSS attack if the website is vulnerable to XSS. In the context of this study, however, the ‘Secure’ flag is most relevant as omitting this flag directly leads to vulnerability to session hijacking attacks such as Firesheep.

3.1.2 Measuring faux web security

In this chapter, the three types of faux web security were described. In summary, the three distinguished types (Table 1) are as follows – ordered from least to most ‘secure’:

- HTTP login to HTTP.
- HTTPS login to HTTP: ‘secure login then insecure’.
- HTTPS login to HTTPS, when the secure connection is not configured properly.

In the context of this study a website is considered vulnerable to session hijacking attacks such as Firesheep if any of the described types of faux web security is displayed. As such, faux web security is measurable over a range of websites by concluding which – if any – type of faux web security is displayed by each site. This, of course, only applies if a user is actually able to authenticate to a website. Sites without logins (e.g. static pages) do not display faux web security in this context by definition as there are not authenticated sessions to steal.

In each of the three types of faux web security the session authentication cookies are used in exactly the same way. As such, the measuring problem can be greatly reduced from detecting which type of faux web security is displayed to analysing the ‘Secure’ flag of the session authentication cookies. In the first type of faux web security, cookies are not secured by definition as no secure connection is used. In the second type of faux web security, cookies are also not secured by definition: the fact that the ‘Secure’ flag is omitted is the reason the ‘secure login then insecure’ behaviour is displayed. In the third type of faux web security, finally, the value of the ‘Secure’ flag of the cookies separates the properly configured sites from the vulnerable. As such, whether or not the ‘Secure’ flag is set on the session authentication cookies determines if a site is vulnerable to sidejacking attacks.

The simplicity of the method of measurement comes from the simplicity of the problem. Security researcher Eric Butler stated that Firesheep was so successful, in fact, because of the simplicity of the problem [7]. Regarding the first and the second type of faux web security, Butler – and later Sullivan – argued that companies made business decisions (cost) to not implement HTTPS across the board in 2010 [4] [6]. While cost was a valid concern in 2010, this is no longer a solid argument with the existence of free HTTPS certificates (e.g. Let’s Encrypt) [60]. Furthermore, there is – and never was – a valid argument that defends the third type of faux web security as the solution is as simple as the definition of the problem. As such, Sullivan states that sites might as well not implement HTTPS across the board at all if the ‘Secure’ flag is not set on the session authentication cookies [4].

3.1.3 Existing evaluations of (faux) web security

In the research context (Chapter 2) several tools and techniques were described that make the process of manually hijacking a session more efficient. As all described tools work reactively, however, none can be used to pre-emptively analyse the security of the cookies of a large number of domains. Additionally, all of the described tools function only by intercepting HTTP traffic. As such, none of the tools allow for the analysis of seemingly secure traffic sent over HTTPS.

Webmasters can internally audit the security of their servers by e.g. analysing the configuration of SSL/TLS. Internal audits require knowledge only available within the walls of a company. This information is private and acquiring and analysing this information for a large number of domains is neither feasibly nor necessary in the context of this study.

External tools exist that allow anyone to audit the configuration of SSL/TLS of any website from the outside. The leading tool is SSL Labs’s SSL Server Test [61]. SSL Labs is a non-commercial research effort backed by Qualys, Thrustworthy Internet Movement and OWASP. Qualys intents SSL Labs to be a collection of documents and thoughts related to SSL in an attempt to understand how SSL is used. Additionally, Quelys intents these documents to be used as reference material when setting up HTTPS.

Anyone can use the SSL Lab’s SSL Server Test tool to audit the security of any website externally. The tool analyses both the certificate chain (e.g. cipher strength) and the server configuration (e.g. protocol details) of a domain by requesting it with all existing web browsers. Both desktop and mobile browsers are used. Additionally, SSL Server Test checks if the domain is vulnerable to a number of known attacks (e.g. BEAST and XSS) specifically. The test results in a detailed rapport and an overall rating, ranging from ‘F’ to ‘A+’.

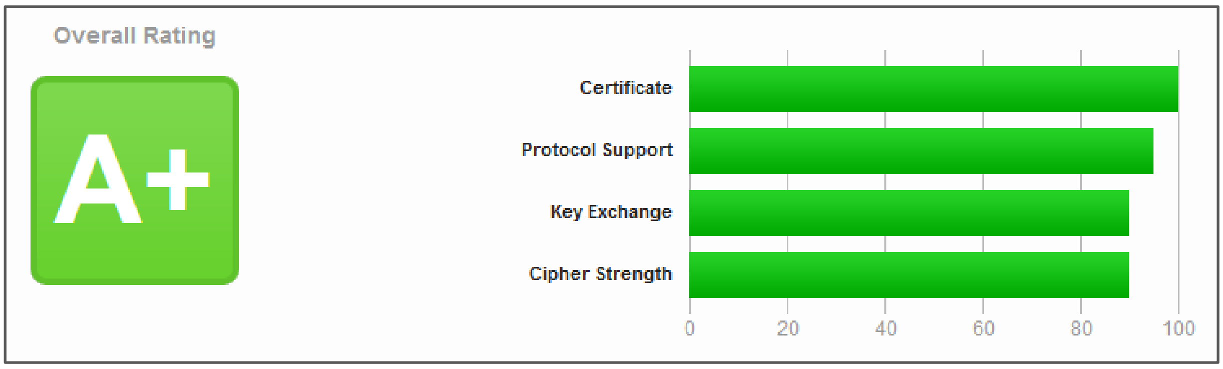


Figure 7: SSL Labs score for one of the test cases, redacted for ethical reasons

Figure 7 shows the results for one of the test cases. The domain scored the highest overall rating possible, which means the server configured SSL properly and is effectively secure.

The Thrustworthy Internet Movement (TIM) aggregates the results from the SSL Lab's SSL Server Test tool in SSL Pulse, the leading survey of SSL implementation of the most popular websites [62]. After experimenting with an assessment of all public sites served over HTTPS, TIM now uses a smaller list of about 200.000 sites based on the Alexa top sites as the test case [63]. The researchers state that this allows the publication of results more often in addition to allowing a more thorough analysis for each domain, resulting in a more relevant conclusion that excludes abandoned sites. SSL Pulse is updated monthly.

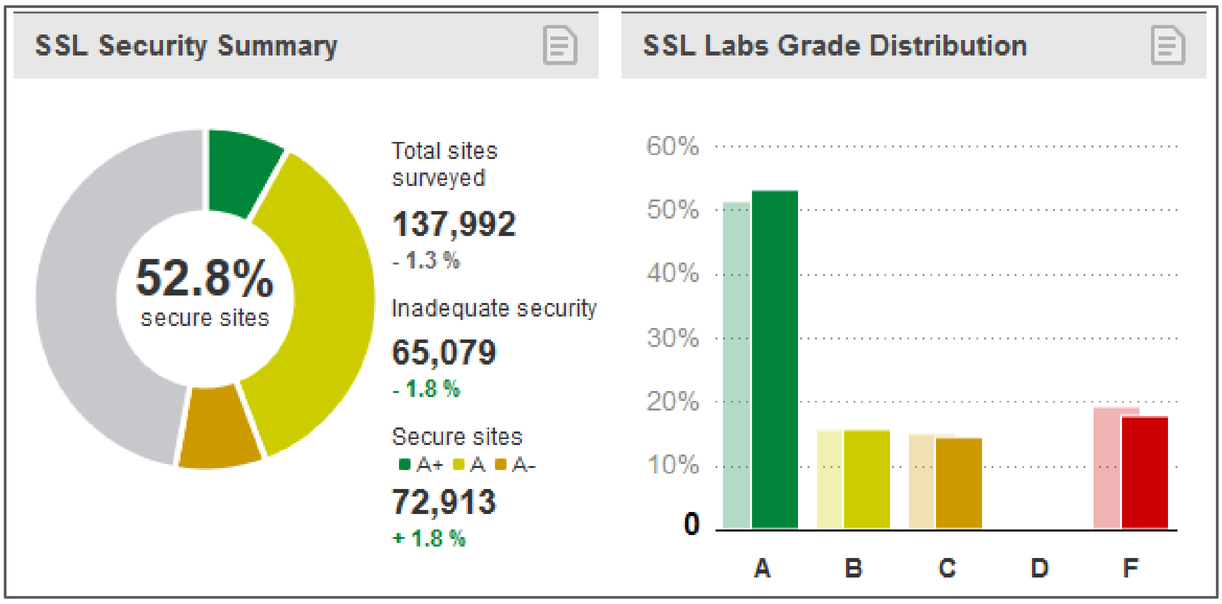


Figure 8: SSL Pulse results for February 2017 [62]

Figure 8 shows that SSL Pulse reports that 52.8% of all tested sites are secure as of February 2017, up 1.8% from January 2017. TIM considers a website effectively secure if it scores at least an 'A' in the SSL Lab's SSL Server Test tool. In the bar graph displayed in Figure 8 (right) the scores for 'A' and 'A+' are consolidated. Additionally, SSL Pulse reports that 10.9% of sites surveyed support HTTP Strict Transport Security (HSTS) [62].

While it may appear that 52.8% of all tested sites are invulnerable to session hijacking attacks, this conclusion is false. The SSL Server Test tool does not take the security of session cookies into account at all [64]. The SSL Server Rating Guide – the full version of which (white paper) is available on request from Quelys – states that detection of insecure cookies is non-trivial to perform in an automated session and that the SSL Server Test tool does not attempt to do it [64]. Additionally, section 4.4 of the full section of the white paper specifically states: *"To remove any doubt that might exist about the seriousness of the above-mentioned issues, we will state that any application that incorrectly implements session token propagation should be awarded a zero score."*

This acknowledgement is nowhere to be found in the (detailed) rapport that results from a test run by the SSL Server Test Tool. Furthermore, the potential drastic impact of this on the scores of domains is not mentioned at all in the documentation of TIM's SSL Pulse. In contrast, the SSL Pulse methodology states that the single selection dataset (SSL Lab's SSL Server Test) gives a clear indicator of where the current state of SSL is heading [62]. Additionally, TIM specifically states that the methodology used allows for analysis of application-layer issues that may subvert SSL security. The fact that a session hijacking attack exploits the properties inherent to HTTP (in the application layer) is not noted.

While (external) evaluations of HTTPS configurations exist, these tools are clearly not usable to conclude the current state of faux web security. The fact that these tools score the effective security of websites without taking the security of session cookies into account may in fact increase that state of faux web security. A webmaster could, for example, falsely conclude that a server is secure because it was awarded the score 'A+' while that server in truth should be awarded a zero score.

3.1.4 Conclusion to Chapter 3.1

A website can display faux web security in one of three ways: serving the entire site over unencrypted HTTP, serving the initial login over secure HTTPS then dropping down to HTTP and serving the entire site over 'secure' HTTPS that is incorrectly configured. Firesheep highlighted the second type, named 'secure login then insecure' behaviour in the context of this study. Whether a given domain displays any one type of faux web security can be measured by analysing the cookies for the existence of the 'Secure' flag. If this attribute is not set, the cookie can be sniffed and can possible be used in a session hijacking attack.

While existing evaluations of (faux) web security exist, they cannot be used to analyse the security of the session authentication cookies for a given domain as these tools do not take the existence of the 'Secure' flag into account at all. Using existing techniques, a web administrator can falsely conclude that his server is effectively secure. Strengthening the validity of our methodology, the SSL Server Rating Guide states that any application that incorrectly implements session cookies should be rated as totally insecure regardless of the results from existing evaluations. Additionally, omitting the 'Secure' flag is one of the main threats defined in the SSL Threat model. Prior research does accurately conclude that the number of domains that are served over HTTPS is increasing over time. This does not mean, however, that the number of sites invulnerable to sidejacking increases in the same ratio.

In conclusion, faux web security can be identified by analysing the session authentication cookies and checking these cookies for the existence of the 'Secure' attribute.

3.2 Proof of problem

To measure the extent to which faux online authentication security is still a problem in2017, it is first necessary to prove the existence of the problem. The purpose of proving the existence of the problem is to prove the validity of the developed proof-of-concept tool.

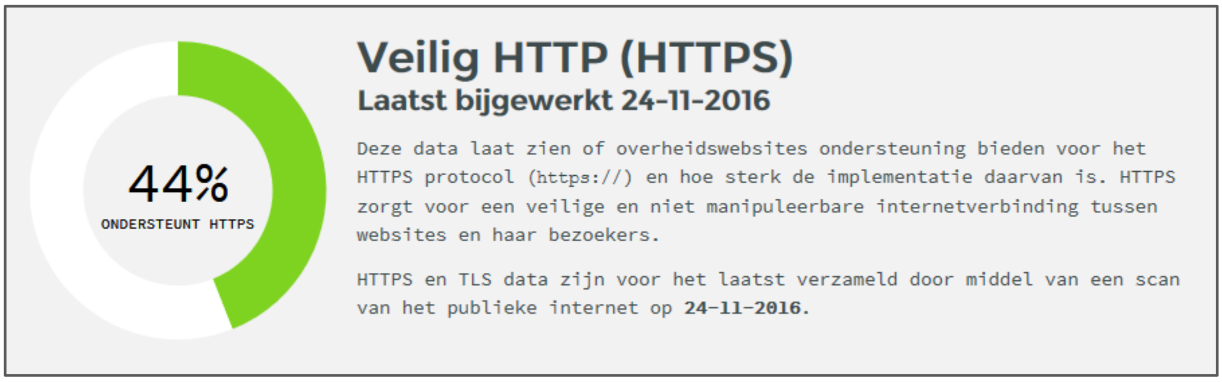
Testing can only prove the presence of bugs, not the absence. As such, test cases are needed to verify the functionality of the tool. Additionally, proving the existence of the problem proofs the validity of the method of measurement by bridging the gap between the theoretical 'can break' and the practical 'is broken'. Finally, proving the problem was used as an initial test to indicate the feasibility of the then to-be-developed prototype tool and, as such, indicate the projected value of this study. To this extent, the following research question is answered in Chapter 3.2:

Does faux online authentication security still exist today?

To answer this question, related work is first described as an extension to the existing evaluations of (faux) web security described in the previous section. Following this, the existence of all three identified types of faux online authentication security is proved 'manually'. Following the base cases, the methods to increase the security of a secure connection (HTTPS) are outlined to strengthen the importance of properly combatting session hijacking attacks at the root of the problem. Afterwards, the process of identifying the test cases is explained. Finally, the results regarding mobile apps and web frameworks that we found when manually performing sidejacking attacks are briefly described.

3.2.1 Faux web security at the Dutch government

Research by the Open State Foundation (OSF) concluded that the majority of websites owned by the Dutch government do not implement HTTPS at all just last year [65].



Veilig HTTP (HTTPS)

Laatst bijgewerkt 24-11-2016

Deze data laat zien of overheidswebsites ondersteuning bieden voor het HTTPS protocol (https://) en hoe sterk de implementatie daarvan is. HTTPS zorgt voor een veilige en niet manipuleerbare internetverbinding tussen websites en haar bezoekers.

HTTPS en TLS data zijn voor het laatst verzameld door middel van een scan van het publieke internet op **24-11-2016**.

Figure 9: HTTPS implementation at the Dutch government as of November 2016

Figure 9 shows that just 44% of all sites in use by the Dutch government support HTTPS as of December 2016, 5.9% lower than the global average as reported by the Thrustworthy Internet Movement (TIM) in the same month [65] [66]. The results are directly comparable, because OSF too classifies the security of the testes sites by just grabbing the census data from SSL Lab's SSL Server Test tool [67]. OSF uses a web crawler based on Pulse by 18F, an office within the General Services Administration (GSA), to rapport the findings [68]. The original Pulse also just looks at the scores as reported by SSL Lab's SSL Server Test tool. As such, the same remarks can be made about the conclusions of this research in context of the current state of faux online authentication security as in Chapter 3.1.3 of this document.

Unlike the research done by TIM – which cautiously interprets the conclusions as 'effective security' – OSF, however, specifically states that using a HTTPS connection guarantees that communication is not intercepted thus guaranteeing its integrity [67]. This is a hasty conclusion as of the analysed sites that use authentication, the number of which is unknown, the integrity of the connection is not at all guaranteed if the session authentication cookies are not secured properly. As such, the number of secure sites that are in use by the Dutch government may in fact be less than the reported 44%.

Following the press release of OSF regarding their conclusions, news network Nederlandse Omroep Stichting (NOS) and digital rights foundation Bits of Freedom (BoF) put the results in context [69]. The organisations thought it was strange that the government does not lead by example and thought it was even stranger that government sites specifically intended to pioneer internet security did not configure HTTPS correctly. As a result, the Dutch government intends to make HTTPS mandatory on government site through the Wet Generieke Digitale Infrastructuur in late 2017 [70]. Whether or not this legislation will mention the security of session authentication cookies is still unknown.

A related Dutch legislation is the so called 'cookiewet' [71] [72]. The original definition of this law stated that websites are required to disclose any use of cookies to the user by means of an opt-in system. The goal is to best guarantee the end user privacy. The latest revision of this legislation, however, requires websites to only mention third-party tracking cookies [71]. Any first-party tracking cookies or 'essential' cookies (e.g. authentication cookies) can be used without any disclosure whatsoever. As such, it is possible that a user may falsely think his login is secure because the use of authentication cookies over HTTP is not disclosed.

3.2.2 Hijacking a session: HTTP to HTTP

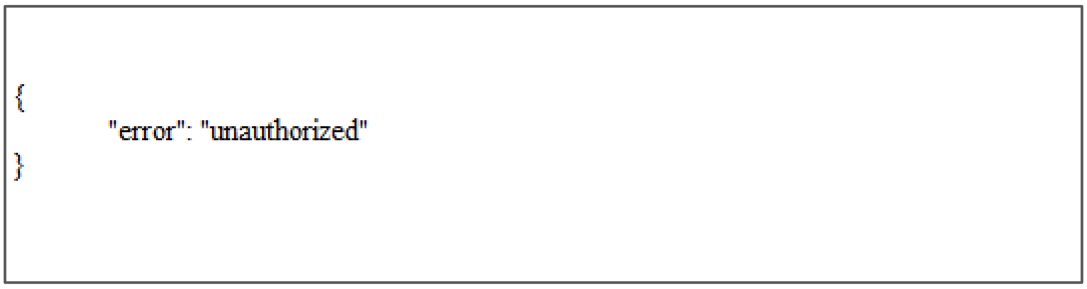
From the research context (Chapter 2) follows that session hijacking has been a known vulnerability since the introduction of cookies in 1994. Before the release of Ferret and Hamster, however, sidejacking attacks were initially performed 'manually'. To prove the existence of the problem in its most basic form – and to prove the inherent connection between sidejacking and cookies sent over HTTP – we performed a manual sidejacking attack in a laboratory setting.

The laboratory setting consists of a web server that sends session cookies without a secure connection (HTTPS) to allow authentication after the initial login process has concluded. This is the default pattern websites used before the introduction of HTTPS in 1994 and this is, therefore, still the pattern websites use when a secure connection is not supported.

A Flask web server was set up on the domain 'api.marcsleegers.com' using Python 3.5 on the back-end. Bar the default configuration of this server on '/', the endpoint '/sheep' is used to prove the existence of the problem. The behaviour of the endpoint is defined as follows:

```
@app.route(“/sheep”, methods=[“GET”])
def index_sheep():
  if verify_session():
    data = {“status”: “OK”,
            “message”: “Logged in as ‘{}’”.format(escape(session[“username”]))}
    return jsonify(**data)
  abort(403)
```
Figure 10: Basic HTTP session management using a session cookie for authentication

Figure 10 shows the basic HTTP session management of the test server. The decorator ‘*@app.route*’ invokes the function ‘*index_sheep()*’ when the server receives a HTTP GET request on the endpoint ‘/sheep’. The function ‘*verify_session()*’ first checks if the request is authenticated by looking for the session cookie and afterwards validates this cookie if one is supplied. If the session cookie exists and the cookie is valid, the request is authenticated. In this case, the username for the authenticated user is returned. If either test fails, the unauthorized HTTP error 403 is returned.



```
{
        "error": "unauthorized"
}
```

Figure 11: The web view the user sees when not authenticated (HTTP 401)

Figure 11 shows the web view the user sees if they are not logged in. This is also the base view from the perspective of the attacked: the view before the session hijacking attack.

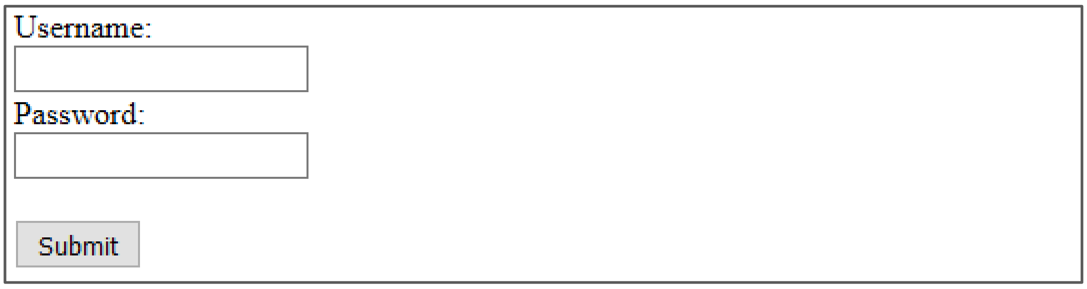
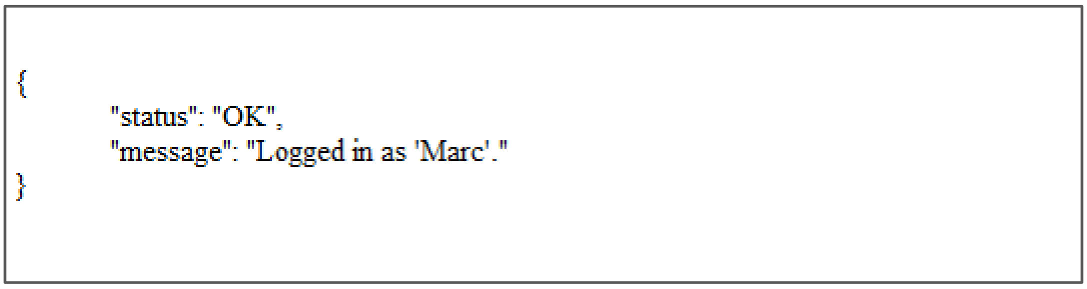


Figure 12: The web view the user sees when visiting the login page unauthenticated

The endpoint ‘/sheep/login’ is used to authenticate the user if no session exists. To this extent, the user is presented with a basic web form. Figure 12 shows this web view. The function ‘*login()*’ validates the user supplied credentials when the form is submitted to the server with a HTTP POST request. If the supplied credentials are invalid, the unauthorized HTTP error 403 is returned. If the supplied credentials are valid, the user is redirected to the endpoint ‘/sheep’. More importantly, the HTTP response sets the cookie ‘session’ on the client by including it in the ‘Set-Cookie’ header. To this extent the server uses the default implementation of the ‘Flask.session’ class which sets the value of the cookie to the contents of the session data dictionary, which in this case only contains the username, and signs this value cryptographically using the default secret key [73]. This means that the user could look at the cookie, but not modify it, unless they know the secret key used for signing. The same applies to the attacker.

If either test fails, the function looks for user credentials and attempts authentication if credentials are supplied. If credentials are supplied and they are valid, the request is authenticated. Additionally, the HTTP response containing the requested data then sets the session cookie ‘session’ to uniquely identify the authenticated session.

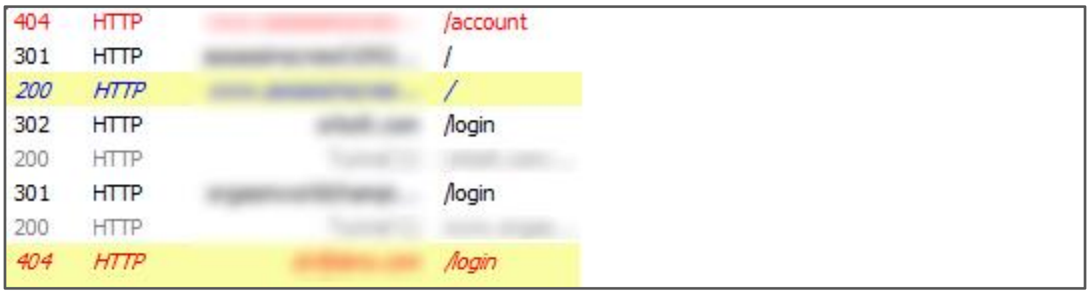


```
{
        "status": "OK",
        "message": "Logged in as 'Marc'."
}
```

Figure 13: The web view the user sees when authenticated (HTTP 200)

Figure 13 shows the web view the user sees after the login process has concluded successfully. The goal of the malicious attacker is to view this screen using a session hijacking attack – thus not needing the credentials the user supplied during the initial login.

To perform the sidejacking attack, the first step is to capture the HTTP traffic of the victim. The initial attack in the laboratory setting was performed in such a way that it mimics Firesheep. To this extent, all HTTP traffic from a controlled victim device (iPhone 6s) was captured on an insecure wireless network using a laptop set up in promiscuous mode. The traffic was logged using the tool Wireshark. While wireless networks are now typically more secure than they were in 2010 (WPA2), Beck and Tews conclude that a nearby attacker can still eavesdrop on a user’s unencrypted internet traffic, such as HTTP, regardless of whether or not the local wireless network itself is secured. [43] Barth and Jackson add that networks cannot be trusted by themselves [44]. Byrd, too, states that wireless networks cannot be trusted to protect unencrypted HTTP traffic from outsiders, even if the network itself is secured [45]. He concludes that end users should treat (secure) wireless hotspots as hostile compromised networks in regards to unencrypted traffic.



404	HTTP	[redacted]	/account
301	HTTP	[redacted]	/
200	HTTP	[redacted]	/
302	HTTP	[redacted]	/login
200	HTTP	[redacted]	[redacted]
301	HTTP	[redacted]	/login
200	HTTP	[redacted]	[redacted]
404	HTTP	[redacted]	/login

Figure 14: Some HTTP traffic sent by and sent to the controlled victim device

Figure 14 shows how the attacker sniffs all HTTP traffic. As follows from the research context, however, sniffing HTTP traffic as shown in Figure 14 can be cumbersome to set up and requires continuously – manually – filtering noisy data that is not needed in the context of this test. Using Sullivan's Cookie Cadger, which was also tested, provided no significant benefit as still only a fraction of the captured traffic was useful for testing.

To speed up testing, the laboratory setting was simplified after having proved the existence of the problem by mimicking the configuration of Firesheep. To this extent, the network monitoring tool Fiddler was used [74]. Fiddler is an HTTP proxy application and, in the context of this study, replaces the wireless network and the promiscuous mode laptop. Fiddler is set up as a proxy, connecting the victim device (iPhone 6s) to the internet. This means that all HTTP traffic coming from the phone was captured by Fiddler – effectively functioning as a man-in-the-middle – and logged before being routed. This significantly sped up testing, as Fiddler includes extensive tools for filtering and grouping noisy HTTP traffic. The end result, however, is exactly the same as in the original laboratory setting.

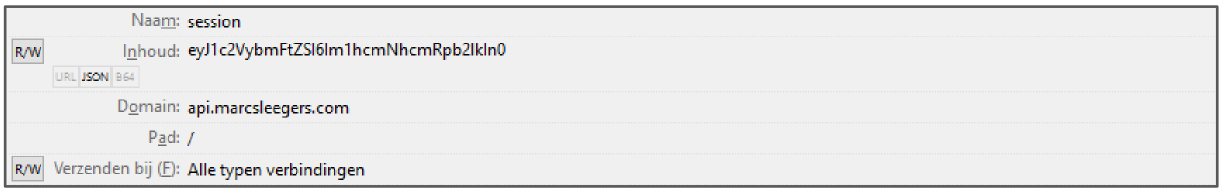
After the user logged in to the test web server running on 'api.marcsleegers.com/sheep', authenticated requests are sent over HTTP. This should mean that the attacker is able to intercept the session cookie included in the request headers. For this test, the attacker sniffed all HTTP traffic sent wirelessly using a laptop set up in promiscuous mode. Note that in this case, HTTP login to HTTP, the user credentials are also exposed as is. We ignore this for testing purposes, as we intent to prove authentication by performing a session hijack.

```
GET http://api.marcsleegers.com/sheep HTTP/1.1
Host: api.marcsleegers.com
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 10_2_1 like Mac OS X)
        AppleWebKit/602.4.6 (KHTML, like Gecko) Version/10.0 Mobile
Accept: text/html
Referer: http://api.marcsleegers.com/sheep/login
Cookie: session=eyJ1c2VybmFtZSI6Im1hcmNhcmRpb2lkIn0
Connection: keep-alive
```

Figure 15: A sniffed HTTP GET request sent from the controlled victim device

Figure 15 shows the relevant information of a HTTP request the attacker was able to intercept. The request is sent from the victim iPhone 6s and follows the automated redirection from the login page which the web server instructed the client to do. From the information presented in Figure 15 follows that the attacker was able to intercept the cookie named 'session'. Additional HTTP headers, such as cookies set for tracking services, are not shown in Figure 15. Now that the attacker has intercepted the session cookie, the attacker can replay it by injecting it into a controlled browser.

To best mimick the functionality of Firesheep, the Firefox browser is used on a machine the attacker controls – in this case a stand-alone desktop computer without a promiscuous networking card and without a connection to Fiddler in later stages of testing. The attacker is initially presented with the unauthorized view as shown in Figure 11. Now, the attacker injects the captured session cookie into the Firefox cookie store – in this case using the Firefox plugin Cookies Manager+ which supports the saving and loading of cookie sets [75]. It is also possible for the attacker to inject the cookie when sending an HTTP request to the server, but adding the cookie to the browser cookie store easily allows subsequent requests.



	Naam:	session
R/W	Inhoud:	eyJ1c2VybmFtZSI6Im1hcmNhcmRpb2lkIn0
URL JSON B64		
	Domain:	api.marcsleegers.com
	Pad:	/
R/W	Verzenden bij (F):	Alle typen verbindingen

Figure 16: The cookie store of the attacker's browser after session injection

Figure 16 shows the cookie store belonging to the attacker's browser after injecting the session cookie captured from the victim iPhone 6s. After having modified the cookie store, the attacker sends another HTTP request to the domain endpoint (e.g. by simply pressing 'F5') – thus adding all data for this domain contained in the cookie store to the request. Following this request, the attacker is presented with the web view as shown in Figure 13: the attacker hijacked the session of the victim successfully and can now browse the website masquerading as the victim. Note that the attacker does not know the user credentials. Also note that the victim does not know the sidejacking attack is happening as the attacker is merely reading freely broadcasted data.

3.2.3 Hijacking a session: HTTPS to HTTP

From the research context (Chapter 2) follows that the companies targeted by Firesheep quickly acknowledged the importance of securing the credentials of their users. To this extent, their websites secured initial authentication by sending it over a secure connection (HTTPS) – thus securing the supplied credentials – but typically ignored the security of the remainder of the session. As such, authenticated requests dropped down to the unencrypted standard (HTTP). In the context of this study this is called 'secure login then insecure' behaviour.

The proof of this case of faux web security follows directly from the proof presented in Chapter 3.2.1 that proved the existence of the base case. While the initial login is protected – unlike the base case where we ignored this for testing purposes – authenticated requests are handled in exactly the same way. As such, the attacker can perform the session hijacking attack in exactly the same way. By capturing the session cookie and injecting it in a controlled browser, the attacker is able to authenticate. To this extent it does not matter that the initial login is secured. Security researcher Eric Butler notably showed the severity of the 'secure login then insecure' problem with the release of his tool Firesheep in 2010 [3].

3.2.4 Hijacking a session: HTTPS to HTTPS

From the research context and the conclusions presented in Chapter 2 follows that HTTPS is only secure when it is properly configured. To prove the existence of this case of faux web security, two proofs are needed. First, it is necessary to find a domain that implements HTTPS across the board, but has improperly configured HTTPS by not setting the 'Secure' flag on the session authentication cookie. Second, it is necessary to prove that such an insecure cookie can be captured by the attacker as in this case sniffing requires an extra step as – by default – all traffic is sent over a secure HTTPS connection. This second proof takes this type of faux web security from 'can be broken' to 'is broken' by combining theory and practice.

As only a single domain was needed to prove the existence of this case of faux web security, we used a brute force approach to find an example. To this extent all HTTP traffic was logged using Fiddler while browsing the web with a clean-slate (meaning no initial cookies were set) Firefox web browser. First, the major sites identified by Firesheep (e.g. Amazon) were visited. Second, the distinct domains in the Firefox history of the researcher were visited in descending order to best mimic 'regular browsing'. Only the sites that implemented HTTPS across the board were tallied for this test.

For each site, the base domain (e.g. 'marcsleegers.com') was visited first. If applicable, the user authentication process was then started meaning the researchers logged in to the site with existing credentials. Finally, the cookies were analysed using the Firefox plugin Cookies Manager+ [75]. The logged HTTP traffic was useless, as expected, because all traffic – including the authentication cookies – was sent over a secure HTTPS connection. Note that it was technically possible for servers to set the 'Secure' flag of a cookie when sending a response over insecure HTTP until protocol update RFC 6265 (i.e. before Firefox 52 released in March 2017) [1]. No such cases were found and the protocol update appears to be semantic as this only happens when a server is deliberately configured to both reply to a secure request with an insecure response and to send a secure cookie with this response.

Within a time frame of 30 minutes four potential victims were identified as being potentially vulnerable. This means the sites implemented HTTPS across the board, but incorrectly configured their session authentication cookies as the 'Secure' flag was missing. The list of potential victims is confidential and therefore not included in this chapter, but is known by the supervisor of this study dr. ir. H.L. (Hugo) Jonker.

The existence of insecure authentication cookies did not prove the existence of the problem in total, however. To prove vulnerability to a session hijacking attack, the attacker has to be able to capture these insecure authentication cookies and use them to authenticate. This means the attacker has to 'force' the victim to explicitly request (part of) the seemingly secure domain using insecure HTTP.

In one of the test cases, this was done automatically by the web server. While the session cookie was only useful on the subdomain 'secure.', the cookie's domain was loosely defined. This means that the cookie is sent with every request on the entire domain and is not restricted to the subdomain. An image file within the page was requested over using basic HTTP, which means the cookie was added to this insecure request.

In the other test cases, the attacker had to interfere. For this, there are several possibilities. The first is based on the case of the insecurely requested image as described above. To this extent an email was drafted which includes an image stored on the target domain. In the URL 'https://' was replaced with 'http://' if applicable. When the victim opens this email, the image is automatically requested using insecure HTTP. It does not matter if the web server redirects this request to 'https://' as the request has already been made and – thus – the cookies have already been sent over insecure HTTP.

This approach is similar to how 'tracking pixels' work – usually for advertisement purposes [10] [76] [77]. An image cropped to 1x1 pixel dimensions is added to an email – typically a newsletter – as a way to track the user. When the user opens the email, the image is requested. As advertisement id cookies are typically defined loosely, they are sent with every request. As such, the sender of the email knows when a specific user has opened the sent newsletter. Email clients can block this behaviour by blocking requests to images, but the major email clients do not block image requests by default [78]. So while image tracking has legitimate uses, it can also be used to assist session hijacking attacks. The major drawback of this approach is that the attacker has to both know the email address of his victim while being connected to the same network – essentially rendering large scale attacks ineffective. The 'SameSite' cookie flag was introduced specifically to protect against these requests forgery attacks. The draft version of this cookie flag is currently implemented Chrome (51) and Opera (39) as of April 2016 [13]. The Firefox development team is still actively working on implementing the (draft) cookie attribute [14].

A related approach is agnostic to the type of request, which means it uses any type of HTTP request and not just image requests. For this approach to work, the user just has to browse to e.g. 'http://www.marcsleegers.com/' explicitly. This can happen, for example, when the user manually enters a URL in the browser or visits a bookmark saved with 'http://' as prefix. This approach can also be targeted by emailing the user a link which they have to click on, again requiring the attacker to have additional knowledge about the victim.

A final approach worth mentioning is cross-sit scripting (XSS). This is a technique that takes advantage of websites that allow it's users to post HTML and JavaScript content [56]. Doing so, an attacker can force the victim's browser to send the cookies for the target domain to a website the attacker controls. Whether or not the 'Secure' flag is set on the authentication cookie is not important, as cross-site scripting circumvents that entirely. This attack can be mitigated with another flag: 'HttpOnly'. This flag indicates that the cookie will not be accessible to client-side scripting languages.

3.2.5 Securing HTTPS

Two major techniques are available to help protect the user when a website displays the type of faux web security described in the previous section. The first is the tool HTTPS Everywhere [79]. This plugin for the major web browsers rewrites any requests to sites known to have HTTPS implemented to use HTTPS. This intents to solve, for example, websites that default to unencrypted HTTP or fill encrypted pages with links that go back to the unencrypted site. This solution is not fool proof, as it requires the user to manually install a plugin. Additionally, the plugin does not catch all insecure requests as the identified test case that requested an image insecurely was still vulnerable after HTTPS Everywhere had been enabled because the image was still requested over unencrypted HTTP.

Another technique – requiring no action from the end user – is HTTP Strict Transport Security (HSTS) [34] [80]. HSTS is a security policy introduced in 2012 intended to protect websites against protocol downgrade attacks, such as explicitly requesting part of a site (e.g. an image) over insecure HTTP. One such downgrade attack is POODLE, which exploits the web browser behaviour of falling back to the outdated SSL 3.0 when the more recent TSL was unavailable [46] [47]. The current implementation of HSTS Is based on original work by security researchers Collin Jackson and Adam Barth [44]. They identified downgrade attacks as a way to compromise browsing sessions back in 2009 – one year before the release of Firesheep. Hodges and Steingruebl also wrote about HSTS in 2010, naming it ‘an extensible approach to the proliferating problem of web based malware and attacks’ [40].

The preference of the web server using HSTS to use a secure connection is communicated to the client in the first HTTP response by adding the header ‘Strict-Transport-Security’. The header specifies a period of time and – as such – asks the client web browser to only send future requests over a secure HTTPS connection. Like the plugin HTTPS Everywhere, the web browser then rewrites all HTTP requests to the domain to use the secure HTTPS before actually accessing the server. The Internet Engineering Task Force (IETF) states that a potential man-in-the-middle attacker has greatly reduced ability to intercept traffic when the HSTS security policy is used [34]. Hodges specifically names Firesheep as one of the things that could have been prevented had the HSTS policy been implemented [40].

HSTS is not flawless, however. By definition, the first HTTP request remains unprotected from sniffing as the client is – at that time – unaware that the server prefers a secure connection as this has not yet been communicated. Because of the time limited approach of the policy, it is also vulnerable to attacks that shift the victim’s computer time [34]. Additionally, HSTS is not a remedy to two other classes of threats: phishing and malware.

Finally, HSTS cannot prevent attacks against SSL/TSL itself (e.g. SSLStrip) as HSTS only works when a secure connection is available. As such, attacks on HTTPS such as BEAST and CRIME are still possible. The CRIME attack specifically targets session authentication cookies [46]. This exploit is still a problem today.

The goal of this chapter is not to argue the security of HTTPS, but to point out that even correctly configured HTTPS is not a failsafe approach to prevent session hijacking attacks. In the light of all possibilities, however, it is vital that HTTPS is correctly implemented on the server. There is no reason to discard the ‘Secure’ flag for authentication cookies, for example, if the cookie is never intended to be send over insecure HTTP.

3.2.6 Identifying test cases

The identified test cases follow directly from the vulnerable sites observed when the existence of the problem was proven in the previous sections of this chapter. More than a dozen vulnerable websites were identified during this process. Of each identified vulnerable case the domain, the date, the vulnerable cookie(s) and the displayed type of faux web security were noted. The list of identified cases is confidential and therefore not included in this chapter, but is known by the supervisor of this study dr. ir. H.L. (Hugo) Jonker.

Without naming and shaming individual websites, we can state that all three identified types of faux web security are represented in the set of test cases. It is worth mentioning that – on some test cases – secure third-party authentication mechanisms could be circumvented. This means that a server outsourced the login process to a third-party system (e.g. ‘Login with Google’) which is secure by itself, but failed to secure their end of the deal. Additionally, it is worth mentioning that one of the test cases is part of a ‘network’ of websites. Capturing the vulnerable session authentication cookie(s) on one site in this network, meant that we could authenticate to other sites in this network using this cookie. In one of the cases the other site was secure (i.e. not displaying faux web security) when analysed by itself.

As described in Chapter 1.5 of this document, the list of identified test cases was handled responsibly. To this extent, a responsible disclosure letter (Appendix 4) was drafted and sent to all domains we identified as being vulnerable to session hijacking attacks. The supervisor of this thesis dr. ir. H.L. (Hugo) Jonker is in charge of this responsible disclosure. To prevent responsible disclosure from taking up too much time and because the test set represented all identified three types of faux web security, we stopped manually performing session hijacking attacks after initial testing concluded. This means the test set of vulnerable domains did not increase further during the remainder of this research. In Chapter 1.5 the consequence for the proof-of-concept tool is described: after local testing, vulnerable domains are (unrecognisably) tallied instead of (recognisably) listed.

During the time period this research was conducted, a handful of the identified test cases fixed the displayed type of faux web security – effectively becoming invulnerable to session hijacking attacks as performed in the context of this study.

3.2.7 Mobile apps

While the online authentication security of mobile apps is not within the scope of this study as described in Chapter 1.4, some preliminary results were found when analysing raw sniffed HTTP traffic during local testing. In exactly the same way as with websites, we were able to capture session authentication cookies sent from mobile apps. We did not attempt to inject the captured cookies into the cookie store of another mobile phone, as that process is drastically different from injecting a cookie in the cookie store of a (desktop) browser. We were, however, able to inject session authentication cookies captured from sniffing mobile app traffic into a browser on a desktop computer. In some cases, we were able to successfully authenticate a fresh browser session using these cookies.

Following this preliminary work, we can conclude that some domains share the same (signature) of cookies between the mobile app and the website. Further research is needed to conclude what this means for the current state of faux online authentication security.

3.2.8 Web frameworks

Likewise, the online authentication security of web frameworks (e.g. WordPress) specifically is not within the scope of this study. While these websites are analysed, the fact that they may belong to a 'family' of websites running the same web framework is not taken into account in this study to prevent deviation from the problem at hand.

During local testing, however, we quickly discovered that (session authentication) cookies often use the same signature as other websites using the same web framework on the back-end. This conclusion is backed by Sullivan [4]. During development of Cookie Cadger, he discovered the same property of the signature of these cookies. Following this, Sullivan was able to successfully pinpoint which web framework – if any – a server was running based on the signature of the (session authentication) cookies. Following this, it may be possible for an attacker to use specific attacks he knows a website is vulnerable to because he knows which attacks the underlying framework is vulnerable to.

Additionally, we discovered that web frameworks – by default – do not set the 'Secure' flag on (session authentication) cookies [81]. This is to be expected, however, as these web frameworks can also be used to power a website that does not use a secure connection at all. Following this, it is up to the webmaster to properly implement secure cookies – sometimes by configuring the admin panel (WordPress) and in other cases by coding it themselves (WordPress and others). It is worth noting that WordPress specifically attempts to automatically set the 'Secure' flag by checking if the current request is secured by SSL. In all cases we briefly observed, however, both the function that sets the authentication cookie(s) and the function that checks for a secure connection can be overridden both manually and by plugins [81]. As such, installing a plugin means that the configuration of the session authentication cookies will be reset and in the worst case re-implemented to not set the 'Secure' flag. With the prevalence of websites using a web framework, this may be one of the reasons the 'secure login then insecure' behaviour is displayed, but we cannot conclude this based on these preliminary results.

Further research is needed on both topics to conclude what this means for the current state of faux online authentication security (and the cause) or the security of web frameworks.

3.2.9 Conclusion to Chapter 3.2

Recent research by the Open State Foundation concluded that the majority of the sites in use by the Dutch government do not implement HTTPS. The conclusions, however, are based on the same existing evaluations as described in Chapter 3.1.3 of this document. It is therefore not guaranteed that the reportedly secure sites are in fact invulnerable to session hijacking attacks. Unlike related research by the Trustworthy Internet Movement which cautiously concludes 'effective security', the Open State Foundation relates the implementation of HTTPS to the guarantee of integrity because communication cannot be intercepted. We know this is false, as sidejacking attacks can indeed be performed if the session authentication cookies are not secured properly.

Bridging the gap between the theoretical 'can break' and the practical 'is broken', the existence of all three types of faux web security as described in Chapter 3.1 is proven in a laboratory setting. Hijacking a session for a site entirely served over unencrypted HTTP is proven by simply sniffing the plain text HTTP traffic. That 'secure login then insecure' behaviour indicates the existence of the problem is proven by sniffing the plain text HTTP traffic after the initial login. The possibility of hijacking a session for a site that implements HTTPS across the board is proven by forcing a protocol downgrade, for example by injecting an image served over insecure HTTP as a man-in-the-middle on a wireless network.

There are techniques to further increase the security of a site that is only served over secure HTTPS, such as the HTTPS Everywhere plugin and the HSTS security policy. These band aids are not fool-proof, however, and do not protect against attacks on SSL/TLS itself. As such, there is no reason to forego setting the 'Secure' flag on cookies that are only intended to be transmitted over a secure connection such as session authentication cookies.

That faux online authentication security still exists today follows directly from the result of the laboratory test. The result is a set of test cases containing all three identified types of faux web security. As such, the problem exists both in theory and in practice. During local testing, the problem was also shown to exist in both mobile apps and web frameworks. How widespread the existence of the problem is in both contexts is reserved for future work.

3.3 Conclusion

Chapter 3.1.4 concludes how faux web security can be identified. In summary, there are three types of faux web security: sites served over unencrypted HTTP only, sites securing the initial login then dropping down to HTTP and sites incorrectly using HTTP across the board. Whether a domain displays any one type of faux web security can be measured by analysing the session authentication cookie(s) for the existence of the 'Secure' flag.

Following this, Chapter 3.2.9 concludes that faux online authentication security does indeed still exist today. The theoretical case is proven by relating the coverage of HTTPS to the problems inherent to the use of the protocol in the context of session hijacking. The theoretical case is proven by hijacking a session for all three types of faux web security.

In conclusion, faux online authentication security can be measured by analysing which type of faux web security – if any – a domain displays by analysing the authentication cookies.

Chapter 4

Prototype

This study counts the sheep and measures how far online authentication security has really come since Butler released Firesheep in 2010. The research question is defined as follows:

How widespread is "faux" online authentication security still?

To answer this question, a custom proof-of-concept software tool was developed in order to measure the extent to which faux online authentication security is still a problem in 2017. In Chapter 3 is described how faux online authentication security can be measured to provide the necessary input for this chapter. As such, the conclusions of Chapter 3 were directly used during the development of the proof-of-concept software tool. The results of the tool are directly used to answer the main research question of this study.

Only the most interesting and most relevant topics and design decisions are described in this chapter. Other topics are discussed briefly where applicable. A more thorough explanation of the developed proof-of-concept tool is available as a design document. This design document, a brief manual, the documentation and the source code of the developed prototype system are available from the supervisor of this thesis dr. ir. H.L. (Hugo) Jonker.

First, the functionality of the developed proof-of-concept tool is described. Afterwards, the effects of the ethical guidelines (Chapter 1.5) on the development process are outlined. Following this, the design of the prototype is illustrated. Then, the automation of the login process is discussed in depth. Finally, following a brief explanation of the packaging and the testing of the tool, the analysing process is described.

4.1 Functionality

The functionality of the proof-of-concept software tool is defined as follows: the tool must be able to automatically analyse the online authentication security of end users for a given number of domains. Following local testing, the 'given number' of domains is to be scaled up. To prevent the same problem that limited Firesheep (specific handlers for each domain) from occurring, the tool must be made generic. As such, the login process for any given domain must be automated given valid user credentials. These credentials are to be sourced from the BugMeNot database initially. Finally, the tool must run on multiple platforms.

Chapter 1.4 describes what is within the scope of this project thematically. To best match the project context (i.e. the total number of available hours), the scope of the development process was also limited. For example, there are a number of possible ways to obtain valid user credentials for testing purposes. Three examples are manual registration, sourcing from the BugMeNot database and automating the registration process. To be able to best answer the research question of this thesis, we decided that it is best to focus on a single source for user credentials. Our reasoning is that the tool is more useful if one of the implementations works well than if a few implementations work slightly less well. Another area in which the development process was scoped is the different ways the login process can be automated. Chapter 4.4 details the choices we made. Following this, the tool had to be made both generic and modular to simplify extending the tool during future work later on.

Taking into account the ethical guideline of this study as outlined in Chapter 1.5, the intended audience of the prototype tool consists of security researchers familiar with the topics discussed. As such, usability of the tool for non-technical end users is not within the scope of this study. Additional security measures that limit or scope the functionality of the developed proof-of-concept tool are outlined in Chapter 1.5 of this document.

4.2 The effect of ethics on the development of the prototype system

The development process of the prototype tool generally spanned two major iterations.

The first iteration followed the feasibility study described in Chapter 3.2 as the proof of problem. The goal of this first iteration was to automate the 'manual' actions performed during initial testing extended with the functionality as described in Chapter 4.1 of this section. This means the first iteration resulted in a system that is able to automate the login process using credentials sourced from the BugMeNot database. The status and the results are displayed to the end user of the tool on a per domain basis. If the results are 'successful' (i.e. the domain is marked as vulnerable to sidejacking) the end user can use the tool to replay the captured cookies. Replaying the cookies is defined as using the cookies to authenticate, thus proving that the attack works and the domain is indeed vulnerable.

The second iteration follows the release of the minimum viable product of the first. The goal of this second iteration was to develop a tool that can measure the extent to which faux online authentication security is still a problem in 2017. As such, the results of the tool directly answer the main research question of this study. The main difference between the first and the second iteration, bar scaling, is that the second iteration takes into account the ethical guidelines as outlined in Chapter 1.5 of this document. This means that the tool no longer displays the status and the results to the end user on a per domain basis and no longer allows the end user to replay captured sessions. Instead, the final proof-of-concept tool decouples the results (i.e. if a domain is vulnerable) from the actual domain and tallies the results. Only the end results are stored internally and, as such, only the end results are presented to the end user of the developed tool.

4.3 Design

This section briefly describes the functional choices made during the development process. As stated in the introduction of this chapter, a more thorough explanation of the developed proof-of-concept tool is available as a design document.

4.3.1 Language

From the project context and the stated functionality of the tool as outlined in Chapter 4.1 follows that the tool must run on multiple platforms. From the functionality of the tool also follows that the tool clearly consists of a few different parts: the tool is part user-interface, part web-scraper, part HTTP traffic analyser and part session replay software. The choice of language must represent all these different parts.

Also taking into account the need for rapid prototyping (i.e. the problem had to be proven fast) and the project context (i.e. the total number of available hours) a language was needed that could quickly bridge the gap between testing script and proof-of-concept tool. Additionally, the tool must be easily extendable during future work.

Finally, we considered the fact that scraping a given number of domains is a potentially computing intensive task. As such, we deeded it important that the code base could easily be shared between and run on multiple systems. From the project planning (not included in this document) follows that a (Linux) server is available via the service PythonAnywhere.

In conclusion, we decided to develop the proof-of-concept tool using the Python (3.5) language. This language ticks all the boxes, allowed us to offload intensive tasks to a remote always running environment and comes with a set of (third-party) libraries that match the intended functionality of the tool.

The (third-party) libraries used include ‘PyQt5’, ‘Requests’, ‘aiohttp’, ‘Selenium’, ‘PhantomJS’, ‘Beautiful Soup 4’ and ‘lxml’. These libraries are described in the separate design document, in which their use within the context of this tool is also outlined.

4.3.2 Architecture

The developed proof-of-concept tool is a separate system, meaning the tool does not run within the runtime of another program (e.g. like a plugin such as Firesheep). This means the main window class (*ShepherdMainWindow(QMainWindow)*) is the main point of entry. The tool, however, also supports environments that do not support user interface layers such as the remote environment running on the (Linux) server available via the service PythonAnywhere (Chapter 4.3.1). As such, the tool has two additional points of entry in the headless worker thread class (*ShepherdWorker(QThread)*) and as a stand-alone script.

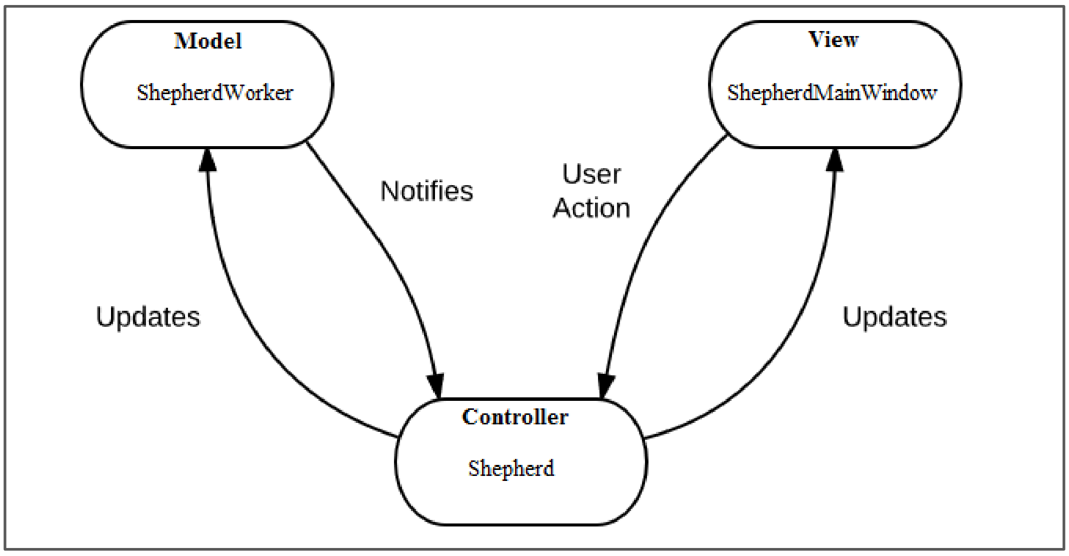


Figure 17: The main components of the prototype (MVC)

The tool is developed using the Model-View-Controller (MVC) software architectural pattern as illustrated in Figure 17. The view is the interface layer and consists only of the class '*ShepherdMainWindow(QMainWindow)*'. User interaction (e.g. clicking a button) signals the controller. The controller consists only of the singleton class '*Shepherd(QObject)*' and bridges the gap between the model and the view. The model is the domain layer, the point of entry of which is the class '*ShepherdWorker(QThread)*'. This layer expresses the behaviour of the application and manages the rules and the login. Because HTTP requests and responses are blocking actions or – in the case of an asynchronous implementation – awaited, the model is set up in a separate thread. This means the program keeps functioning (e.g. the user can press the button labelled 'STOP') while the analysis is in progress. The communication between the controller thread and the model thread is set up using thread signals (class '*pyqtSignal()*') and thread slots (decorator '*@pyqtSlot*'). If a signal is connected to a slot then the slot is called when the signal is emitted. For example, the controller observes the signal '*signal_results*' which communicates the analysis results for a single domain. Upon emit, the connected slot is called – in this case redirecting the signal arguments to a local function.

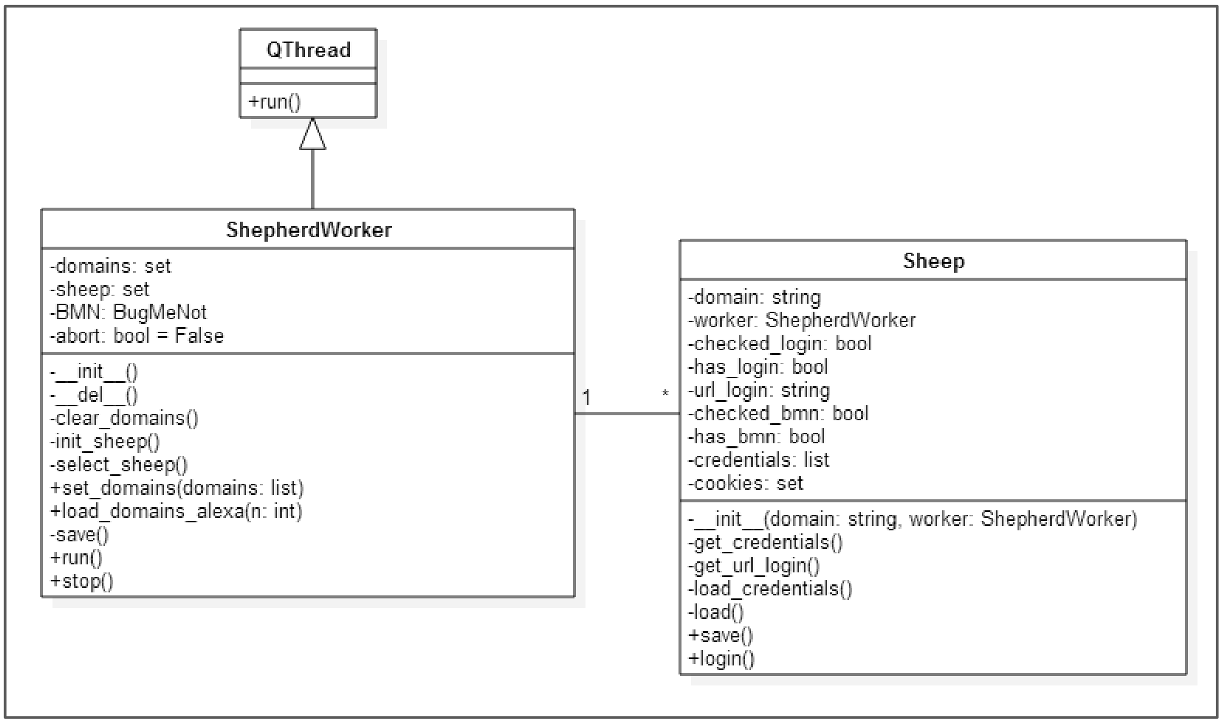


Figure 18: The (simplified) class diagram illustrating the domain layer

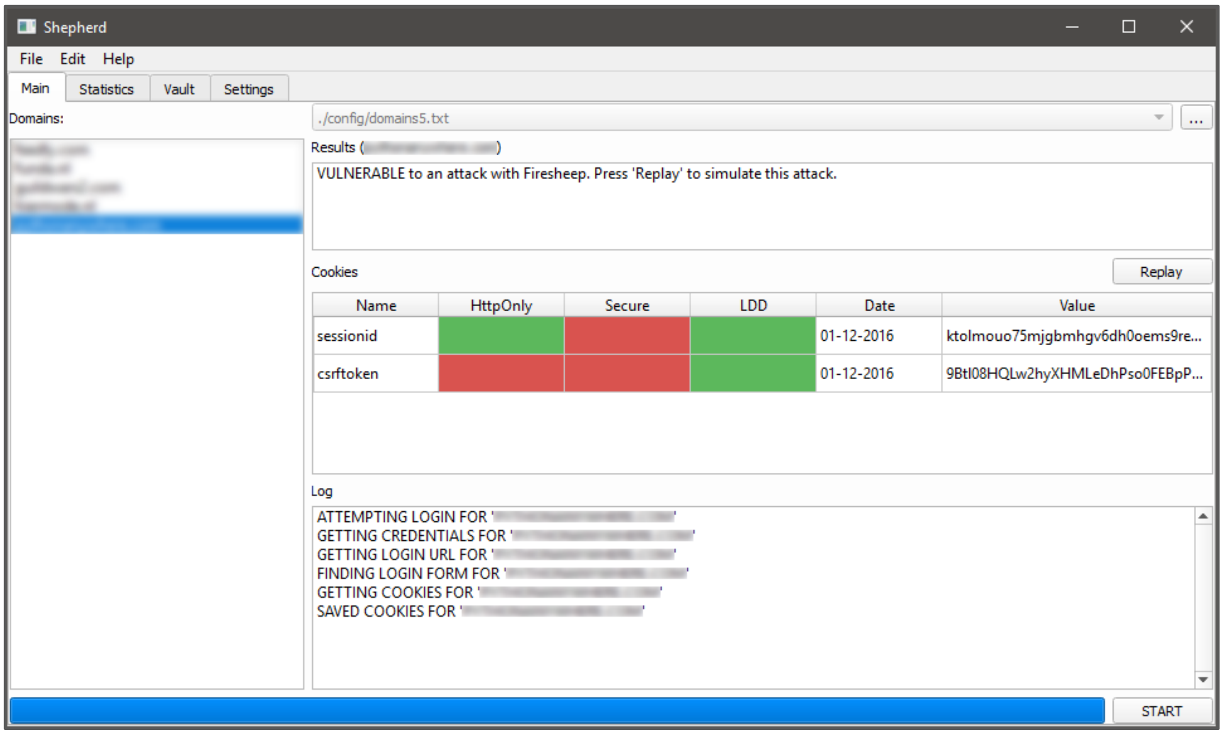
Figure 18 shows a simplified illustration of the domain layer. Note that trivial methods (e.g. getters and setters) as well as signals and slots are not displayed. The threaded class '*ShepherdWorker(QThread)*' is responsible for analysing a number of domains. These domains are, shuffled and in an arbitrary order, stored in a set. This way, we cannot directly relate the tallied results to a single domain – even for very small sample sets. Each domain is represented as an instance of '*Sheep()*', which stores all domain data in memory.

Each sheep is saved to and loaded from the database layer, which is not displayed in Figure 18. The database layer consists of the connection between the model and the SQLLite database. SQLLite is described as a replacement for 'f*open()*', which matches the caching functionality the database is used for in the context of this study. Note that the end results of the analysis are not stored in the database as per the ethical reasons described in Chapter 1.5 of this document. The results are, tallied without signature, saved as comma-separated values (CSV) for later use. The database layer is described in the separate design document.

Also not displayed in Figure 17 and in Figure 18 are utility classes, for which we also refer to the separate design document. These classes include functionality for BugMeNot, Alexa, the database, finding the login page, filling the login form, extensions to interface widgets, tests, custom exceptions and alternative implementations of the login process (e.g. Selenium).

4.3.3 Interface

The end user interface of the developed proof-of-concept tool is drastically different between the two major iterations. This is to be expected, because both iterations have different functional goals as outlined in Chapter 4.1 of this document.



Results ([illegible])

VULNERABLE to an attack with Firesheep. Press 'Replay' to simulate this attack.

Cookies

Name	HttpOnly	Secure	LDD	Date	Value
sessionid	🟩	🟥	🟩	01-12-2016	ktolmouo75mjgbmhgv6dh0oems9re...
csrftoken	🟥	🟥	🟩	01-12-2016	9Btl08HQLw2hyXHMLeDhPso0FEBpP...

Log

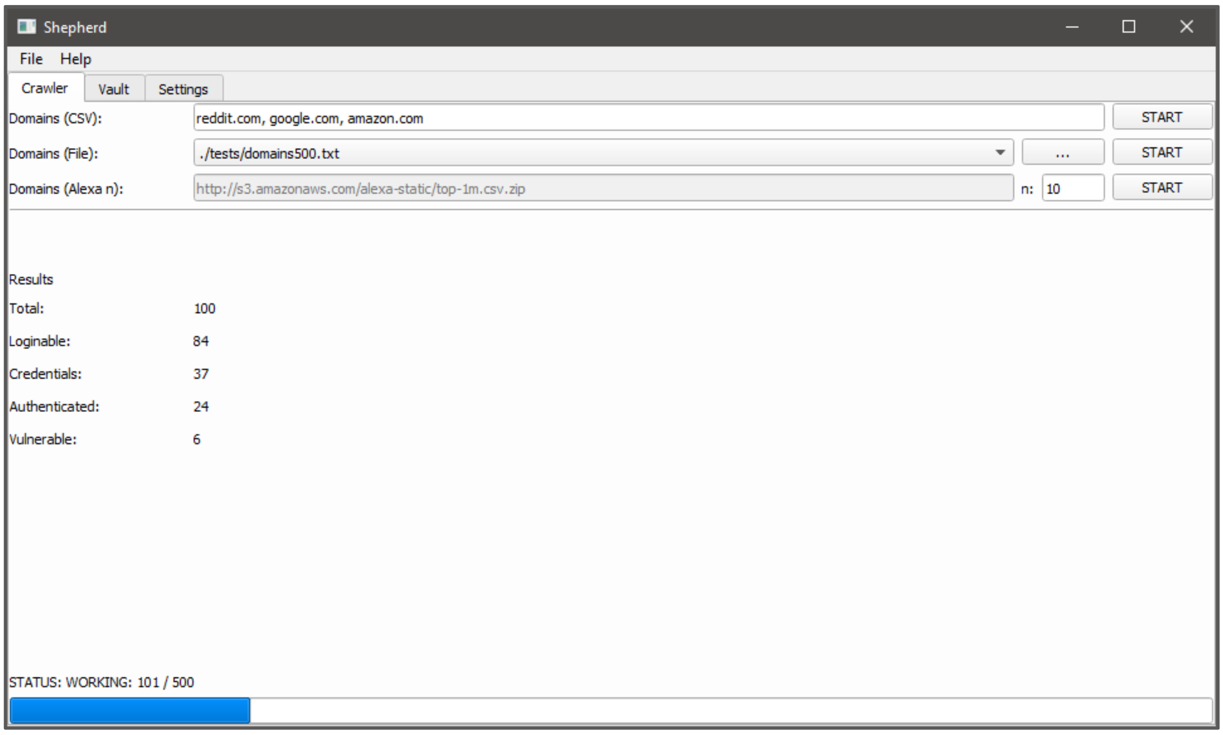
```
ATTEMPTING LOGIN FOR '[illegible]'
GETTING CREDENTIALS FOR '[illegible]'
GETTING LOGIN URL FOR '[illegible]'
FINDING LOGIN FORM FOR '[illegible]'
GETTING COOKIES FOR '[illegible]'
SAVED COOKIES FOR '[illegible]'
```

Figure 19: Shepherd proof-of-concept iteration 1 main view

Figure 19 shows the user interface of the first iteration of the proof-of-concept tool. The list titled 'Domains:' on the left displays a subset of the test cases used as described in Chapter 3.2.6 of this document. Note that all domains and identifiable results are blurred to comply with the ethical guidelines as outlined in Chapter 1.5 of this document.

The button labelled 'START' on the bottom right of the interface asks the controller to signal the model to start analysing all listed domains. The blue progress bar on the bottom of the interface shows the total progress that the domain thread has made. During the time the tool is analysing the domains, most user interface actions (e.g. selecting and loading another file and the button labelled 'Replay') are greyed out. During – and after – the analysation thread is running the user can click on a domain on the left side of the screen at any time. Doing so immediately updates the main section of this screen to show all status messages and results, if any, for the selected domain. At any time, the user can press the start button, which is labelled 'STOP' during analysation, to immediately stop analysing domains. At any time, the user can copy and paste any label text or column text by using the keyboard commands specific to the operating system.

After the model is done analysing or after analysing has been stopped, the user can click on the button labelled 'Replay' for any domain. Doing so opens a Chromium browser window (running on Selenium) as a sub process. All identified cookies are injected into this fresh browser session. If the tool marked the domain as vulnerable to sidejacking and authentication was successful, the user is then logged in without the need for credentials.



Domains (CSV): reddit.com, google.com, amazon.com
Domains (File): ./tests/domains500.txt
Domains (Alexa n): http://s3.amazonaws.com/alexa-static/top-1m.csv.zip n: 10

Results

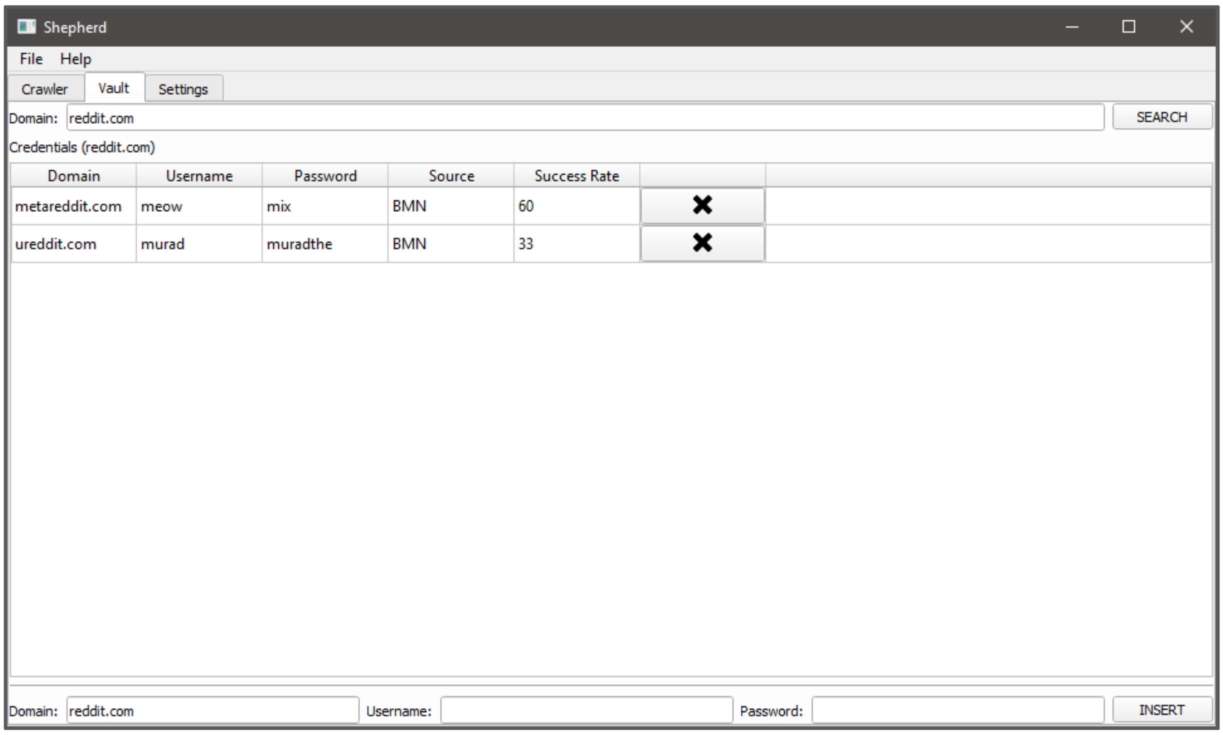
Total:	100
Loginable:	84
Credentials:	37
Authenticated:	24
Vulnerable:	6

STATUS: WORKING: 101 / 500

Figure 20: Shepherd proof-of-concept iteration 2 main view

Figure 20 shows the user interface of the second iteration of the proof-of-concept tool. The domain/details view/subview combination is replaced entirely with a domain agnostic window displaying the current statistics. This user interface shows that the goal of the program changes between the first and the second main iterations. The white space to the right of the statistic view is left blank on purpose. While visualising the live data is not within the scope of this study, real-time graphics (e.g. graphs) can be added here in future work.

The top part of Figure 20 shows the three different types of user input: comma separated values (CSV), file and Alexa n. If the user enters invalid input, a blocking error message is shown stating the error. The buttons labelled 'START' on the top right of the interface ask the controller to signal the model to start analysing all user supplied domains. As in the first iteration, most user input is greyed out when the crawler is running its analysis. Note that the domains are analysed in a non-specific order, meaning the first n-x results will not be the same if two tests are run with n=100 consecutively. After the crawler is done analysing or after the system has been stopped manually, the results are saved to disk (CSV).



Domain	Username	Password	Source	Success Rate	
metareddit.com	meow	mix	BMN	60	✖
ureddit.com	murad	muradthe	BMN	33	✖

Figure 21: Shepherd proof-of-concept iteration 2 vault view

Figure 21 shows the 'Vault' tab of the user interface of the second iteration of the proof-of-concept tool. This tab holds a fairly straightforward CRUD-application (Create, Read, Update and Delete). The user can enter a domain in the search bar labelled 'Domain:' in the top part of the screen. Pressing the button labelled 'SEARCH' will query the database for any credentials found in the database listing the domain, the username, the password, the source and the success rate in the table below. Pressing the button labelled 'X' deletes the credential from the database. Pressing the button labelled 'INSERT' add the user supplied credential to the database with the source 'MAN' for 'MANUAL' and a success rate of 100. Inserting a new domain automatically queries the database for this domain as if the user searched for this domain manually. Likewise, searching for a domain automatically populates the domain on the bottom of the screen to allow for quick-insertion of new credentials.

4.4 Automating the login process

To automatically analyse the session authentication cookies for a given number of domains pre-emptively, the login process had to be automated generically to obtain these cookies.

Automating the login process consists of the following four steps, in order:

1. Getting authentication credentials.
2. Finding the login page.
3. Finding the necessary login data.
4. Attempting the login.

This chapter describes these four steps in order.

4.4.1 Getting authentication credentials

There are several possible ways to obtain (valid) user credentials. Before development, we identified the following possibilities:

- Manual registration.
- Sourcing credentials from the BugMeNot database.
- Sourcing credentials using Amazon Mechanical Turk.
- Automating the registration process.

Manual registration is the process of manually registering to a domain as a new user. This method was used during initial testing, but was deemed not feasible for a larger number of domains (i.e. Alexa top n) because of the costly required amount of manual labour (time).

BugMeNot is an internet service that provides credentials (i.e. usernames and passwords) to let internet users bypass registration on websites [85]. The service is a response to the fact that many internet users find required registration to be an annoyance [83]. BugMeNot returns a list of known credentials when the end user supplies a domain. After being pressured to shut down, the creators behind BugMeNot implemented an opt-out system for sites behind a paywall. The site itself is ranked 16081 in the top Alexa sites as of February 2017 [84]. There are no major, legal alternatives to BugMeNot. The sites listed by the Alexa index as 'related' serve illegal content (e.g. stolen credentials to payed accounts), are often related to adult themes or were marked as being malicious by Firefox during research.

Amazon Mechanical Turk is an online marketplace for work intended to give business and developers access to an on-demand, scalable workforce [85]. Given clear instructions, called HITs, a Mechanical Turk Worker (which is a real person) can earn real money by completing these HITs to the satisfaction of the Mechanical Turk Requester. Amazon boasts that paying customers can get thousands of HITs completed in minutes. In the context of this study, this means that workers can be payed from a monetary research fund to manually register new user accounts to a large number of domains.

Automating the registration process is the process of automating the process of manually registering to a domain as a new user, much like this chapter describes automating the login process. Automatically registering to a given domain and making this process generic is a research project on its own, however. A sample problem that could occur is automatically solving CAPTCHA's. We deemed this out of the scope of this study, given the availability of other options and taking into account the project context (i.e. the total number of available hours). A related problem is regulatory: for now, we deemed it not ethical to create 'spam accounts' for research purposes as we did not consult all 'n' end user agreements.

In conclusion, we decided to source the BugMeNot database to obtain valid authentication credentials as this option is free and does not require a significant amount of work – hours which could be spend better elsewhere. The other options are reserved for future work.

Regarding the technical implementation of this module, one thing is worth noting. During initial testing, this module was developed to use an asynchronous approach when sourcing credentials from BugMeNot. As BugMeNot does not have a public facing database, the credentials were initially sourced by – for each domain – asynchronously requesting the URL ‘bugmenot.com/view/DOMAIN’ where ‘DOMAIN’ is replaced with the name of the domain. The server running on ‘bugmenot.com’ soon started throttling the asynchronous HTTP requests thus limiting the usefulness of this approach. Following this setback, we switched to a synchronous approach and instructed the crawler to wait between each request. Doing so we successfully circumvented the request throttling at the cost of increasing runtime. Whichever request approach we used, any listed credentials contained within the returned page source were found by parsing and traversing the HTML node tree.

4.4.2 Finding the login page

The page returned when requesting the base domain (e.g. ‘marcsleegers.com’) is often not the login page. In the context of this study, the ‘login page’ is the page that contains the login form an end user can use to submit his or her credentials. By pressing a button (e.g. labelled ‘Submit’) an end user attempts to login. To automate this login process, the login page must be found – if any – for a given domain.

Existing web crawlers and scrapers (e.g. Scrapy) assume that the given base URL is the login page [86]. Finding the login page, we found, is seemingly straightforward. All of the identified test cases that are ‘loginable’ (i.e. have a login page) link to the login page from the start page if the session is not currently authenticated. This means that given a domain (e.g. ‘marcsleegers.com’) an end user can browse to the login page by clicking a link (e.g. labelled ‘sign in’). Not providing an easy way for an end user to log in results in a bad user experience. As such, we assume that the following holds true for all domains in the context of this study: if a domain is ‘loginable’, the login page is the start page or can be found as a link on the start page.

Regarding the technical implementation of this module, it is worth noting that the search queries used to find the login page within the parsed HTML node tree are fully customisable. The current implementation is scoped to only search for phrases containing Western Latin character sets (e.g. the Dutch ‘inloggen’ and the English ‘sign-in’). This functionality can be extended (e.g. supporting translations) in future work.

4.4.3 Finding the necessary login data

Given a domain, its login page and (valid or invalid) credentials an end user can enter the credentials on the login form presented on the login page and attempt authentication. To automate this step of the login process, we identified the following two possibilities:

- Mimic the POST HTTP request that is send when a login form is submitted.
- Masquerade as an end user client browser.

When an end user attempts authentication using a login form the client browser sends a HTTP POST request to the server containing the supplied credentials. This process is described in detail in Chapter 3, including authenticated requests using session cookies. The login form, however, possibly contains hidden data and an action redirect URL that possibly differs from the login page [87]. An example of a piece of hidden data is the CSRF token, intended to stop cross-site requests [88]. When the end user clicks the submit button, this hidden data is automatically attached to the HTTP POST request by the client browser to validate the authentication attempt. The authentication attempt is invalid – regardless of the validity of the credentials – if the required hidden data is not attached to the request. As such, this hidden data is necessary when mimicking a manual login attempt and must be found in the page source.

The other possibility, masquerading as being an end user client browser, does not require extracting hidden form values in the page source code as the client browser handles submitting the form. Selenium is one tool to automate browsing the web by instantiation an actual client web browser [89]. Unlike a regular end user web browser which is driven by user input (e.g. mouse clicks), Selenium is driven by code input. As such, this step of the login process can be automated by ordering an instance of Selenium to browse to the login page and subsequently fill in and submit the login form.

We decided to mimic the HTTP POST requests that are sent when a login form is submitted to automate this step of the login process. The main reason we decided to go with this option is the effectiveness. HTTP requests can be batched or be sent asynchronously, unlike a Selenium instance which is blocking. This means the crawler cannot perform any other actions while waiting on a response from Selenium. Other browser automation tools suffer from the same problem in the context of this study. Headless automation tools (i.e. without a user interface), for example PhantomJS, are also blocking [90].

One way to combat this drawback is be to instantiate multiple instances of Selenium. We deemed this solution not scalable to a larger amount of domains (e.g. Alexa top n) because of the overhead. Sending, for example, a thousand HTTP requests at the same time directly translates to opening a thousand instances of Selenium. The impact on performance is effectively the same as opening a thousand instances of Firefox as an end user. The added overhead is disproportionate and unnecessary as a basic HTTP POST request fulfils the same goal. Furthermore, any additional functionality that comes with browser automation tools is not needed when performing this step of the automation of the login process. Session replay functionality as used in the first iteration of the tool (Chapter 4.3.3) can easily be added using blocking tools when the session has been authenticated using a bare bones request.

Custom crawler and scraping code was developed to locate and to extract any possible hidden data that is needed to successfully submit a login form. This code is based on the 'loginform' code of the open-source web crawler Scrapy, which scores a set of forms based on the form items (e.g. a field named 'username') [91]. The final algorithm used in the proof-of-concept tool locates all forms, if any, in the HTML source code of a given login page by building an element node tree from the source code. All forms in the tree are then scored and ranked based on the algorithm provided by the Scrapy code. The form items in the highest scoring form are cross referenced with the results from the Scrapy algorithm. As such, the login form is known as is the intended function (e.g. the field for the username) for each form item on the form. Additionally, the algorithm returns any additional form values extracted from the '<form>' HTML tag such as hidden tokens and an action redirect URL.

4.4.4 Attempting the login

Given a domain (known), (valid or invalid) credentials (step 1), the login page for the domain (step 2) and the login form data (step 3) the prototype crawler can attempt to login. Attempting the login is done by sending a HTTP POST request to the URL in the form's 'action' attribute and supplying both the credentials and any hidden data in the request payload in a fresh session (i.e. the session has no pre-existing cookies). Afterwards, the authenticated session cookie store is analysed for containing cookies marked as 'Secure'.

The attempted login is not a successful login by default. One way the attempted login can fail is if the supplied credentials are invalid. As such, the crawler needs to check whether or not the login attempt succeeded. Theoretically, we are allowed to assume that the login attempt succeeded if the server responds with the HTTP status code 200 ('OK') [8] [92] [35]. Other HTTP status codes (e.g. 401 ('Unauthorized') or 403 ('Forbidden')) are available if the authentication attempt is valid but the authentication did not succeed. Unauthorized means that the request is successful (i.e. all required form data was sent), but the following authentication failed because the supplied credentials are invalid. Forbidden means the session – regardless of authentication status – is not allowed to view the response, e.g. if a logged in user tries to view administrator content.

Initial testing, however, concluded that not all websites follow standard procedure by definition. For example, one test case responded to the login attempt with the HTTP status code 200 effectively stating the attempt was successful. Upon inspection of the returned HTML source code however, we discovered that a modified login page was returned. On this modified page, the text 'invalid username/password' was labelled in red just below the login form. So while we initially assumed the login attempt had succeeded, the login attempt had in fact failed. Scrapy and other existing web crawlers do not have functionality that solves this problem, instead assuming the supplied credentials are valid or the behaviour of an invalid login attempt is known [86]. This is a valid design choice if Scrapy is used to, for example, scrape a specific website. In this unique case of mass web scraping, however, neither the validity of the credentials or the behaviour of a given website is known as one of the goals of the proof-of-concept tool is to be both generic (e.g. no handlers) and modular.

To combat this behaviour and to, as such, decrease the number of false positives reported, the functionality of the crawler was altered in a later iteration. To this extent, we assume that the following holds true for all domains in the context of this study (as it did for the test cases): if the session is authenticated with a session identifier, the session cannot authenticate with a different session identifier at the same time. This is true by definition (see Chapter 2.1.1), because session authentication cookies have a unique name. This is why a session hijacking attack – in which the *value* of the session cookie with the same *name* is replaced – works in the first place. An end user is typically shown this by, for example, replacing the text 'log in' on a webpage with the text 'log out'.

Technically, the proof-of-concept crawler uses this property of cookies to check if the login attempt had succeeded by performing step 3 of the login process again. This means that, given the session, the crawler attempts to locate the login form again (Chapter 4.4.3). If no login form can be found when it was found before, the attempted login was successful. A negative side-effect of this technique is that this requires an additional HTTP GET request. Given a large number of domains, the amount of requests made increases by the number of domains. As a result, the runtime of the tool increases.

Other techniques were attempted to limit the amount of extra HTTP GET requests, but these options were not as successful. Comparing the HTML source code was unsuccessful, and introduced false negatives, because of dynamic content. Comparing page titles or locating usernames was unsuccessful, because of static titles (e.g. always displaying 'Company' and not 'Company | Account'), concluded also by Mundada et al. [50]. Finally, comparing raw cookie data was unsuccessful because no values were changed in some test cases even when the login attempt was successful. This last case is a problematic and faulty implementation of HTTP cookies. Without changing the value of the session cookies (e.g. managing state on the server) allows for session fixation attacks [42] [41]. This type of attack is not within the scope of this study and vulnerability to session fixation was therefore not tallied.

To showcase the modularity of the developed proof-of-concept tool to the client, the (bare bones) automation of the 4 steps of the login process was also implemented using browser automation tools distributed as a separate script. Development on both a headless approach (PhantomJS) and an approach using a user interface (Selenium) was started. Extending and integrating these modules in the proof-of-concept tool is reserved for future work.

4.5 Packaging

The developed proof-of-concept tool is distributed as a Python 3.5 package. Python 3.5 is an interpreted language, which means the project directory does not need to be built. Python is included in the Linux operating system by default. Users of other operating systems can easily download Python from 'https://www.python.org/downloads/release/python-353/' or compile it from source by following the same link.

The proof-of-concept tool requires a handful of third-party dependencies to run as described in Chapter 4.3.1 of this document. During development, these dependencies were automatically managed by the integrated development environment (IDE) in the file 'requirements.txt'. For the end user, two custom batch scripts have been developed to simplify distribution of the developed prototype tool. Running the included script 'INSTALL.bat' sources all required dependencies listed in 'requirements.txt' from the Python Packaging Index (PIP) and installs these dependencies to the local Python package library. The included script 'RUN.bat' starts the proof-of-concept program using the Python 3.5 interpreter in the main point of entry (GUI).

Note that the Alexa top 500000 contains a large amount of adult websites. We advise to run the tool in a virtual machine or to ask the IT-department for permission to use the tool.

To achieve a uniform coding style the Style Guide for Python Code (PEP8) was followed where applicable [93]. Only the third party library 'PyQt5' does not follow this coding style, because the coding style inherent to Qt is followed. As such, we decided to all interface code conform to this style to achieve a uniform coding style in the interface layer too. All public facing methods and classes are documented with docstrings conforming to PEP8.

A more thorough description of the packaging of the tool and a more detailed user manual are available in the separate design document and the separate user manual.

4.6 Testing

Worth mentioning here are the integration tests, which tested the combination of the individual software modules as a group. The content of the test set is described in Chapter 3.2.6 as the domains that were manually identified as being either vulnerable or invulnerable to sidejacking attacks during local testing. Also stated in Chapter 3.2.6 is that the size of the test set did not increase as per the ethical guidelines as described in Chapter 1.5 of this document. Without naming and shaming individual websites, we can state that all three identified types of faux web security are represented in the set of test cases. The developed proof-of-concept tool passed all integration tests, in which the results of the tool were compared with the manually obtained results for each domain in the test set.

The interface layer and the packaging of the tool were tested during the system tests, in which the tool was deployed and used on different computing environments. The use case for each of these tests was to analyse a given subset of the Alexa top n domains, effectively computing the results for the analysis in parallel (Chapter 4.7.3).

All test- and documentation methods used (e.g. the validity tests) are described in the separate project planning document.

4.7 Analysing (faux) online authentication security

First, the set of domains to be analysed is explored. Second, the results of the analysis are described. Afterwards, the runtime of the program is discussed. The final entire analysis was done in the months February and March of 2017.

4.7.1 The set of domains to be analysed

Following local testing, the number of domains to be analysed was incrementally scaled up to the Alexa top 500000 domains. We were soon able to conclude, however, that the coverage of the BugMeNot dataset did not perfectly match the testing dataset. This means that the BugMeNot database we scraped did not contains credentials for all of the domains in the testing dataset, averaging a coverage of about 10% during initial testing. As such, we decided to preload the dataset using a tailored script that only performs the first step of the login process (Chapter 4.4). This resulted in a database containing 92783 credentials for 46548 unique domains. As such, the sourced credentials cover 9,3% of the Alexa top 500000.

Preloading the database with credentials significantly reduced the runtime of the following steps of the automation of the entire login process, as the tool could only attempt to login to about 10% of the Alexa top 500000. This also means that we are no longer able to directly conclude the ratio of sites that have a login page (i.e. are not static) and that we have sourced credentials for. We believe, however, that the amount of 'loginable' domains is significantly higher than 10% of the Alexa top 500000. One argument is that the current dataset contains exactly 51634 unique domains for which the crawler has checked the existence of a login page. This disparity does not necessarily mean that the remainder of the domains do not have a login page. As outlined in Chapter 4.4.2, step 2 of the login process was scoped to include only Western Latin character sets in the context of this study.

Group, by ranking	Number of domains
First 100	36
First 1000	80
First 10000	2921
First 100000	18985
First 200000	28467
First 300000	35358
First 400000	40960
Alexa top 500000	46485

Table 2: Number of domains (n=500000) grouped by ranking

Another argument could have been the relationship between the ranking of a domain in the Alexa top n and the (lack of) availability of credentials in the BugMeNot database. The reasoning behind this argument is the opt-out system described in Chapter 4.4.1, which we initially thought would skew the dataset in favour of domains with a lower ranking. Table 2, however, shows this is not true at all. The data shows that the number of domains with at least one credential is fairly even distributed over the Alexa top 500000 domains. Note that we are not able to say anything about the relationship between the number of *valid* credentials and the ranking of the domain, because of the methodology of this study. As described in Chapter 1.5, the crawler only tallies the end results for ethical reasons. This means that the uniquely identifying domain name is decoupled from the results.

To reason about the validity of the credentials before performing the actual analysis of the entire login process, the success rate as provided by BugMeNot was used. Next to the credentials for a given domain, BugMeNot also provides a user defined success rate on a scale of 0 to 100.

Group, by success rate	Number of domains
=100	13220
>=75	18840
>=50	28756
>=25	35938
>0	38755

Table 3: Number of domains (n=500000) grouped by credential success rate

Table 3 shows the numbers of unique domains with a least one credential grouped and sorted by success rate. From the data follows that 13220 domains have perfectly working credentials and 7793 domains only have credentials that do not work at all according to the users of BugMeNot. Anyone, however, is able to vote on the validity of a credential on the BugMeNot website without registration. Manual inspection of the database showed that some of the credentials with a provided success rate of 0 indeed appeared to be invalid, in some cases consisting of an advertisement text for the username whereas an email address was expected. Other cases, however, appeared to be regular credentials.

The developed tool was used to run a small test including only a handful of randomly selected domains and credentials that were reported to be invalid. The crawler was able to use reportedly 'invalid' credentials to successfully authenticate to 15 out of 100 domains in this small test. Again, we were unable to link a specific domain to these successes because the domain name is decoupled from the results because of ethical reasons as described in Chapter 1.5 of this document. Following these results, we decided to ignore the reported success rates and include all data in the analysis as some credentials were indeed valid.

4.7.2 Results of the analysis

Final testing was performed on 46548 unique domains, distributed over the Alexa top 500000 as illustrated in Table 3. The crawler was able to successfully authenticate to 4689 domains. This is 10,1% of the entire test set and 0,94% of the Alexa top 500000. Of these domains, the crawler identified 3764 domains as vulnerable to session hijacking attacks. This means that 80.1% of sites that the crawler was able to authenticate to are vulnerable to sidejacking. This also means that 0,75% of all websites in the Alexa top 500000 are vulnerable to attacks such as Firesheep given the methodology used in this study.

While the coverage of the Alexa top 500000 is low (both in the size of the test set and in the number of results) it is indeed alarming that 80.1% of successful authentications resulted in the detection of vulnerability to session hijacking attacks. Note that the results are heavily skewed towards domains that serve their web content using a Western Latin character set, exist in the BugMeNot database and did not bother to opt out of the BugMeNot database. As such we cannot directly extrapolate this ratio of successful authentications and detected vulnerabilities to the entire set of Alexa top 500000 domains.

From these results we cannot directly conclude that 89,9% of sourced credentials are invalid. Another reason could be that the crawler was unable to locate the login form because of non-Western Latin characters. The crawler marked an additional 19253 unique domains as having credentials, while not having a login page. This means that the crawler was unable to find the login page or that the domain does in fact not have a login page. In this last case, the credentials provided by BugMeNot could be advertisement text instead of credentials (see chapter 4.7.1). Assuming that this is not the case and the credentials provided by BugMeNot are indeed valid credentials, an additional 19253 domains can be authenticated to by extending the proof-of-concept tool. Exactly 10554 of these domains end in '.com', which means the domains are international and could be of any language. The other domains are localised websites, of which Russian Federation sites (non-Western Latin characters) span the crown totalling 826 unique domains ending in '.ru'. Of the entire Alexa top 500000 exactly 244396 (48.9%) unique domains end in '.com' and 23402 (4.7%) unique domains end in '.ru' specifically; suggesting that future work extending the tool can indeed be valuable.

Other reasons authentication failed include invalid credentials, the server detecting the crawler as non-human thus providing a (re)CAPTCHA to solve, two-factor authentication. As scoped earlier in this chapter, all of these are problems reserved for future research.

4.7.3 Runtime of the analysis

Excluding preloading the dataset, the runtime of the entire analysis (n=46485) was about 19 hours. The availability of credentials for the Alexa top 500000 was preloaded using 4 computing environments in parallel, resulting in a real runtime of about 1 day and a total runtime of about 4 days. Note that the runtime is variable, as the delay between each HTTP request and its response is not set.

A performance bottleneck exists with foreign domains. Chinese and Russian domains, for example, respond significantly slower to requests than Western domains. In some cases, this resulted in the total analysis of a foreign domain taking over 30 seconds to complete whereas analysing a typical Western domain resulted in a runtime of less than 1 second. To combat this, we first tried setting a (more) strict timeout, which obviously resulted in the crawler discarding a subset of results as having timed out. Given enough time, however, the crawler was in fact able to complete authentication to slow foreign domains. As such, we decided to wait for and to tally these slow results in order to best answer the research question. We also tried excluding foreign domains (filtering out '.ch' and '.ru'.), but again decided against this for the same reason: we wanted to tally the results.

The maximum possible runtime occurs when each HTTP request made replies just before timing out. The default HTTP timeout set in the tool by the dependency 'Requests' is 9 seconds. Including the sourcing of credentials, a maximum of 4 HTTP requests are made for each domain. Three HTTP GET requests are used to obtain the login data (credentials, login page and login form data) and one HTTP POST request is performed to attempt authentication. This is also the minimum number of HTTP requests needed, given no data exists. Following this, the theoretical maximum runtime of the analysis is 464.9 hours or 19 days for the current dataset (n=46485) and about 200 days for the Alexa top 500000.

Chapter 5

Discussion

While the results of this study are both promising and alarming, the success rate (i.e. number of successful authentications) must be increased to be able to conclude how widespread faux online authentication security still is with more certainty. Because (the methodology of) this study was heavily scoped in multiple parts, extending this research results in an increase of the success rate of the developed prototype tool.

Nevertheless, the fact that this study was scoped meant that certain choices were made over others. While we think the arguments presented for choosing the BugMeNot database over other sources of credentials are still sound, exploring the usefulness of the BugMeNot database earlier could have projected the lessened impact (n=46548) of the approach. That said, without actually using all credentials to authenticate it is impossible to verify whether or not a credential is (still) valid. As such, the results from a more extensive preliminary study would still end up the same (n=4689).

In hindsight, it might have been better to focus on sourcing a large number of credentials first. This could have been done, for example, by first conducting a study about the automation of the registration progress. That said, only one option could be chosen taking into account the project context (i.e. a single graduate with a total of 400 hours available). Afterwards, even if a large number of credentials were obtained, the usefulness of the database would have been very limited. Without knowing if – and if so, how – the credentials can be used to automate the login process in a generic fashion, indicating the projected value of such a study would have hard. Therefore, this study can also be viewed as a means to project the value of a study focused on sourcing a large number of credentials.

That said, the pre-emptive and generic approach of our tool already overs significant advantages over similar studies that use a reactive and non-generic approach. The tool Newton by Mundada et al., which has a similar goal, requires user interaction and custom handlers for each unique domain. Additionally, their approach to validating authentication success (comparing usernames) did not prove fruitful. As such, their final test case consisted of 45 unique domains. Extending this set by a single domain already requires development of an extra handler in addition to monitoring user interaction reactively – in which the availability of valid credentials is assumed. In contrast, our proof-of-concept tool was already able to analyse 4689 domains – each of which required no specific attention.

Our thoughts about increasing the success rate are combined with our other recommendations for future work in Chapter 7 of this document.

Chapter 6

Conclusion

This study counted the sheep and measured how far online authentication security has really come since the release of Firesheep in 2010. The following research question was explored:

How widespread is "faux" online authentication security still?

To answer this question, a custom proof-of-concept tool was developed in order to measure the extent to which faux online security is still a problem in 2017. Before developing this tool, a theoretical literature study and a practical laboratory test showed that faux online authentication security is indeed still a problem and that its existence can be measured.

A website can display faux web security in one of three ways: serving the entire site over unencrypted HTTP, serving the initial login over secure HTTPS then dropping down to HTTP and serving the entire site over 'secure' HTTPS that is incorrectly configured. Firesheep highlighted the second type, dubbed 'secure login then insecure' behaviour. Whether a domain is vulnerable can be measured by analysing the cookies. If 'Secure' attribute is not set, the cookie can be sniffed and can theoretically be used in a session hijacking attack.

While existing evaluations of web security exist, they cannot be used to analyse the security of the session authentication cookies of a given domain, as these tools do not take the existence of the 'Secure' flag into account at all. Using existing techniques, a web administrator can falsely conclude that his server is effectively secure. Strengthening the validity of our methodology, the SSL Server Rating Guide states that any application that incorrectly implements session cookies should be rated as totally insecure regardless of the results from existing evaluations. Additionally, omitting the 'Secure' flag is one of the main threats defined in the SSL Threat model. Prior research does accurately conclude that the number of domains that are served over HTTPS is increasing over time. This does not mean, however, that the number of sites invulnerable to sidejacking increases in the same ratio.

Bridging the gap between the theoretical 'can break' and the practical 'is broken', the laboratory test showed that faux online authentication security is indeed still a problem that can be exploited in practice today. Local testing proved the existence of and the practical validity of a session hijacking attack for all three types of faux web security. Hijacking a session for a site entirely served over unencrypted HTTP is proven by simply sniffing the plain text HTTP traffic. That 'secure login then insecure' behaviour indicates the existence of the problem is proven by sniffing the plain text HTTP traffic after the initial login. The possibility of hijacking a session for a site that implements HTTPS across the board is proven by forcing a protocol downgrade, for example by injecting an image served over insecure HTTP as a man-in-the-middle on a wireless network. Faux online authentication security was also briefly shown to exist in both mobile applications and in web frameworks.

Following the conclusions that the faux online authentication security is still a problem and that the existence of this problem can be measured, a proof-of-concept tool was developed to help answer the main research question of this study. In the first iteration of the prototype system, the possibility of automated analysis of vulnerability to session hijacking was shown for a limited number of domains. The crawler automates the sourcing of user credentials, finding the login page and the required login form data and the actual authentication process. Following a successful login, the validity of the methodology was proven by the replay functionality of the tool – logging into accounts without credentials.

Following local testing, the Alexa top 500000 domains were analysed initially using the second iteration of the proof-of-concept tool. To this extent, the BugMeNot database was used as a means to source a large number of user credentials. Unfortunately, the sourced data did not cover the Alexa top 500000 entirely. Using credentials sourced from BugMeNot, the state of the authentication security of 46548 unique domains was analysed. While only about 9.3% of the initial domains were analysed, the domains checked were distributed fairly even over the Alexa top 500000.

The entire dataset, including the preloading of credentials for all 500000 domains in parallel, was analysed in just below 50 hours. During this time, the crawler was able to successfully authenticate to 4689 unique domains identifying 3764 unique domains as being vulnerable to a session hijacking attack. This does not necessarily mean that 89.9% of the sourced credentials are invalid. Other reasons could be that the crawler was unable to locate the login form (non-Western Latin characters) or that the server detects the tool as a non-human and provides a CAPTCHA. Both problems are not within the scope of this study.

While the coverage of the Alexa top 500000 is low (both in the size of the test and in the number of results) it is indeed alarming that 80.1% of successful authentications resulted in the detection of vulnerability to a session hijacking attack. As the results are heavily skewed towards domains that serve their web content using a Western Latin character set and that did not bother to opt out of the BugMeNot database, this result cannot be directly extrapolated to the Alexa top 500000 domains. From these results alone, however, we can already conclude that faux online authentication security is indeed still a widespread problem in 2017.

This study and its results are a direct contribution to the overarching research of dr. ir. H.L. (Hugo) Jonker in the context area ‘Security and Privacy’ conducted at the Open Universiteit Nederland. The results of this study will be used to partially conclude the state of security and privacy assurances of the digital life in the context of online authentication. Additionally, the developed prototype tool will be used to explore related topics such as the state of authentication security of mobile apps and will be used for demonstrations about authentication security on conventions. Finally, this thesis, the developed tool and the results serve as direct input for the follow-up paper by dr. ir H.L. (Hugo) Jonker, B. (Benjamin) Krumnow MSc and M. (Marc) Sleegers BA.

Chapter 7

Future work

Several interesting and related topics were discovered during this study. Some of the questions popped up early in the research phase and were deemed not within the scope of this project, taking into account the project context (i.e. the total number of available hours). Other questions popped up during the development of the proof-of-concept tool, from the discussion of this study or following the conclusion to this study.

Two related problems are faux online authentication security in both mobile apps and in web frameworks. During this study, vulnerabilities to session hijacking attacks were discovered in both contexts. Faux online authentication security in apps is especially related to this study, as the discovery of the problem seems to be identical. Using sniffed sessions in an app controlled by the attacker, however, is different from the methodology used during this study. Future research can conclude how captured sessions can be used to perform session hijacking attacks on mobile apps both in theory and in practice. Related, future research can conclude how widespread the existence of the problem is in mobile apps in the same fashion as this study did for websites. Regarding web frameworks, future research can follow up Sullivan's remarks about fingerprinting. An interesting question is if targeted sidejacking attacks can be trivialised based on the cookie fingerprint of a web framework.

This study aimed to analyse the Alexa top 500000 domains. To this extent, the BugMeNot database was chosen as the source of credentials – sparingly supplemented by manual registration during local testing. The resulting set of credentials unfortunately does not cover the entire Alexa top 500000 domains. As such, future work can explore alternative ways to source (valid) user credentials. During this study, the following three possible alternatives were identified but deemed out of scope: (extensive) manual registration, sourcing credentials using Amazon Mechanical Turk and automating the registration process. The last option in particular is extremely suited for future research. Related problems are the automated circumvention of CAPTCHA's, complying with regulations against spam accounts and validating new accounts that require activation (e.g. by clicking a link in an email). Future work in these contexts requires both a theoretical study as a possible extension to the modular proof-of-concept tool as developed during this project.

Increasing the 'n' in 'Alexa top n' can be done in other ways than increasing the amount of available credentials. Future work can, for example, analyse the long tail of the web. A related problem is exploring if there is a difference between the number of vulnerable domains in the Alexa top and lesser used domains. An alternative approach is analysing a localised top n, e.g. the 500000 top domains in The Netherlands. Future work on web frameworks relates to this, as web frameworks are typically used by mom and pop stores.

The number of analysed domains can possibly be increased further by exploring attacks on SSL/TSL itself. Attackers potentially use any means necessary to increase the effectiveness ratio of their attacks, so exploring which – and if so, how – attacks increase the effectiveness of session hijacking attacks. Examples include SSLStrip, WEP-cracking and POODLE.

From the technical implementation of the proof-of-concept tool follows an interesting question about the density of 'loginable' sites. To scope this study, we chose to only analyse whether or not a site is 'loginable' if a credential was available as this resulted directly in useful data for the project. It can be useful to know, however, how many sites in a list of domains (e.g. the Alexa top 500000) have a login page and are subject to analysis. A related topic is increasing the number of identifiable domains with a login page by extending the parsing of the HTML node tree to support other character sets (e.g. translations) as the current implementation only functions for sites using Western Latin character sets.

Finally, two technical recommendations follow directly from the implementation of the prototype tool. The first recommendation is increasing the ease of use of the tool, for example by adding real-time visualisations of the analysed data to the existing white space. The second recommendation is exploring how the automation of the login process can be extended and if this increases the number of results. The two bare bones implementations in Selenium and PhantomJS, for example, can be extended and included in the modular tool.

This thesis serves as a basis for a follow-up paper on the same topic by dr. ir. H.L. (Hugo) Jonker, B. (Benjamin) Krumnow MSc and the author of this document. This paper is intended to be submitted to both ESORICS'17 and CCS'17. Regarding this follow-up project, future research may be needed on the licence of the proof of concept tool. A related problem is whether or not it is sound to release the tool as an open-source project as is, taking into account the response to and the widespread usage of the Firesheep plugin for Firefox.

Bibliography

1. **Barth, Adam.** HTTP State Management Mechanism. *IETF.* [Online] 2011. https://tools.ietf.org/html/rfc6265. RFC 6265.

2. **Infosec Institute.** Session Hijacking Cheat Sheet. *Infosec Institute.* [Online] 20 January 2015. http://resources.infosecinstitute.com/session-hijacking-cheat-sheet/.

3. **Butler, Eric.** Firesheep. *codebutler.* [Online] 24 October 2010. http://codebutler.com/firesheep.

4. **Sullivan, Matthew.** *Cookie Cadger: An Auditing Tool for Wi-Fi or Wired Ethernet Connections.* Ames : Iowa State University, 2012.

5. **Murphy, Katie.** New Hacking Tools Pose Bigger Threats to Wi-Fi Users. *New York Times.* [Online] 16 February 2011. http://www.nytimes.com/2011/02/17/technology/personaltech/17basics.html?_r=0.

6. **Butler, Eric.** Firesheep, a day later. *codebutler.* [Online] 26 October 2010. http://codebutler.com/firesheep-a-day-later.

7. **Butler, Eric en Gallagher, Ian.** Firesheep, three weeks later: Fallout. *codebutler.* [Online] 18 November 2010. http://codebutler.com/firesheep-three-weeks-later-fallout.

8. **Fielding, Roy T, et al.** Hypertext Transfer Protocol -- HTTP/1.1. *IETF.* [Online] 1999. https://tools.ietf.org/html/rfc2616. RFC 2616.

9. *HTTP Cookies: Standards, privacy, and politics.* **Kristol, David.** 2, sl : ACM, 2001, ACM Transactions on Internet Technology (TOIT), Vol. 1.

10. **Smith, Richard.** The Web Bug FAQ. *EFF.* [Online] 11 November 1999. https://w2.eff.org/Privacy/Marketing/web_bug.html.

11. **Walker, J, West, M en Goodwin, M.** SameDomain Cookie Flag. *IETF.* [Online] 6 April 2016. https://tools.ietf.org/html/draft-west-first-party-cookies-07. Suggested update to RFC6265.

12. **Helme, Scott.** Cross-Site Request Forgery is dead! *Scott Helme's Security Blog.* [Online] 20 February 2017. https://scotthelme.co.uk/csrf-is-dead/.

13. **Google.** 'SameSite' cookie attribute. *Chrome Platform Status.* [Online] 29 March 2016. https://www.chromestatus.com/feature/4672634709082112.

14. **Goodwin, Mark.** Bug 795346 - Add SameSite support for cookies. *Bugzilla@Mozilla.* [Online] 13 March 2017. https://bugzilla.mozilla.org/show_bug.cgi?id=795346.

15. **Stuttard, Dafydd en Pinto, Marcus.** *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws.* sl : Wiley, 2011. 1118026470 / 978-1118026472 .

16. **Whitaker, Andrew en Newman, Daniel.** *Penetration Testing and Network Defense.* sl : CISCO, 2005. 978-1587052088.

17. **Graham, Robert.** SideJacking with Hamster. *Errata Security.* [Online] 5 August 2007. http://blog.erratasec.com/2007/08/sidejacking-with-hamster_05.html.

18. **Ou, George.** Hamster plus Hotspot equals Web 2.0 meltdown! *ZDNet.* [Online] 2 August 2007. http://www.zdnet.com/article/hamster-plus-hotspot-equals-web-2-0-meltdown/.

19. **Rescorla, Eric.** HTTP Over TLS. *IETF.* [Online] 2000. https://tools.ietf.org/html/rfc2818. RFC 2818.

20. **Palmer, Chris en Zhu, Yan.** How to Deploy HTTPS Correctly. *EFF.* [Online] 9 February 2017. https://www.eff.org/https-everywhere/deploying-https.

21. **Rusli, Evelyn.** Lazy Hackers Unite: Firesheep Boasts +104,000 Downloads In 24 Hours. *TechCrunch.* [Online] 25 October 2010. https://techcrunch.com/2010/10/25/lazy-hackers-twitter-firesheep-boasts-100000-downloads-faceboo/.

22. **Atwood, Jeff.** Breaking the Web's Cookie Jar. *Coding Horror.* [Online] 13 November 2010. https://blog.codinghorror.com/breaking-the-webs-cookie-jar/.

23. **Gibson, Steve.** *Security Now! with Steve Gibson: Firesheep.* [Podcast] 2010. Episode 272.

24. **Craver, Thom.** Goodbye, Keyword Data: Google Moves Entirely to Secure Search. *Search Engine Watch.* [Online] 23 September 2013. https://searchenginewatch.com/sew/news/2296351/goodbye-keyword-data-google-moves-entirely-to-secure-search.

25. **Bower, Jeff.** Installing Firesheep on Ubuntu. *ebower.* [Online] 5 June 2012. https://www.ebower.com/docs/ubuntu-firesheep/ubuntu-firesheep.pdf.

26. Wireshark. [Online] https://www.wireshark.org/.

27. **Hunt, Troy.** HTTPS adoption has reached the tipping point. [Online] 30 January 2017. https://www.troyhunt.com/https-adoption-has-reached-the-tipping-point/.

28. HTTPS usage statistics on top websites. *STATOPERATOR.* [Online] https://statoperator.com/research/https-usage-statistics-on-top-websites/.

29. **Aas, Josh.** Progress Towards 100% HTTPS, June 2016. *Let's Encrypt.* [Online] 22 June 2017. https://letsencrypt.org/2016/06/22/https-progress-june-2016.html.

30. **Gilbertson, Scott.** HTTPS is more secure, so why isn't the web using it? [Online] 3 March 2011. https://arstechnica.com/business/2011/03/https-is-more-secure-so-why-isnt-the-web-using-it/.

31. **Bahaji, Zineb en Illyes, Gary.** HTTPS as a ranking signal. *Google Webmaster Central Blog.* [Online] 6 August 2014. https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html.

32. **Schechter, Emily.** Moving towards a more secure web. *Google Security Blog.* [Online] 8 September 2016. https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html.

33. **Digital India.** How To Rank Better in 2017? Start with SSL Certificate. *Digital India.* [Online] 20 March 2017. https://www.https.in/blog/how-to-rank-better-in-2017-start-with-ssl-certificate/.

34. **Dolanjski, Peter en Vyas, Tanvi.** Communicating the Dangers of Non-Secure HTTP. *Mozilla Security Blog.* [Online] 20 January 2017. https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/.

35. **Vyas, Tanja.** Login Forms over HTTPS, Please. *Mozilla Hacks Blog.* [Online] 28 January 2016. https://hacks.mozilla.org/2016/01/login-forms-over-https-please/.

36. **Vyas, Tanvi.** No More Passwords over HTTP, Please! *Mozilla Blog.* [Online] 28 January 2016. https://blog.mozilla.org/tanvi/2016/01/28/no-more-passwords-over-http-please/.

37. **Hodges, Jeff en Barth, Adam.** HTTP Strict Transport Security (HSTS). [Online] 2012. https://tools.ietf.org/html/rfc6797. RFC 6797.

38. **Fielding, Roy.** Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. *IETF.* [Online] 2014. https://tools.ietf.org/html/rfc7231. RFC 7231.

39. **Dacosta, Italo, Chakradeo, Saurabh en Amahad, Mustaque.** *One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens.* Georgia : Georgia Institute of Technology, 2012.

40. **Liu, Alex, et al.** *A Secure Cookie Protocol.* sl : IEEE, 2005.

41. **Noubir, Guevara en Raynal, Michel.** *Networked Systems.* Marrakech : Springer, 2014. 978-3-319-09581-3.

42. **De Ryck, Philippe, et al.** *Primer on Client-Side Web Security.* 2014 : Springer. 978-3-319-12226-7.

43. *The Need for Coherent Web Security Policy Framework(s).* **Hodges, Jeff en Steinguebl, Andy.** sl : Proceedings of the Web 2.0 Security and Privacy 2010 Workshop, 2010.

44. **Kolsek, Mitja.** *Session fixation vulnerability in web-based applications.* sl : Acros Security, 2002.

45. **Schrank, Michael, et al.** *Session Fixation - the Forgotten Vulnerability?* sl : Sicherheit, 2010.

46. *Practical attacks against WEP and WPA.* **Tews, Erik en Beck, Martin.** Zurich : WiSec '09 Proceedings of the second ACM conference on wireless network security, 2009. 978-1-60558-460-7.

47. *ForceHTTPS: Protecting High-Security Web Sites from Network Attacks.* **Barth, Adam, Jackson, Collin.** sl : Proceedings of the 17th International World Wide Web Conference, 2008.

48. **Byrd, Christopher.** *Unsafe at Any SSID: Wireless Hotspot (In)Security.* sl : ISSA, 2011.

49. **Sheffer, Y en Holz, R.** *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS).* sl : IETF, 2015.

50. **Moller, Bodo, Duong, Thai en Kotowicz, Krzysztof.** *This POODLE Bites: Exploiting The SSL 3.0 Fallback.* sl : Google, 2014.

51. *A Survey On Authentication Attacks And Countermeasures In A Distributed Environment.* **Jesudoss, A en Subramaniam, N.** 1, sl : Indian Journal of Computer Science and Engineering (IJCSE), 2014, Vol. 5. 0976-5166.

52. **Ristic, Ivan.** *SSL/TLS Deployment Best Practices.* sl : Qualys, 2014.

53. **Mundada, Yogesh, et al.** *Half-Baked Cookies: Client Authentication on the Modern Web.* sl : Georgia Tech, 2016.

54. **Jonker, Hugo, Mauw, Sjouke, Trujillo-Rasau, Rolando.** *Man-in-the-middle attacks evolved... but our security models didn't.* Heerlen : Open Universiteit Nederland, 2017.

55. **Krumnow, Benjamin.** *Tracking online data.* To be published.

56. **Vlot, Gabry.** *Determining the effectiveness of webscraping.* To be published.

57. **Drazner, Clayton en Duza, Nicola.** *Block-assessor.* To be published.

58. **Labs, SSL.** *SSL Threat Model.* sl : Qualys, 2016.

59. **OWASP.** Cross-site scripting (XSS). *OWASP.* [Online] 6 April 2016. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

60. **Czagan, Dawid.** Securing Cookies with HttpOnly and Secure Flags. [Online] 6 March 2014. http://resources.infosecinstitute.com/securing-cookies-httponly-secure-flags/.

61. **Atwoord, Jeff.** Protecting Your Cookies: HttpOnly. *Coding Horror.* [Online] 28 August 2008. https://blog.codinghorror.com/protecting-your-cookies-httponly/.

62. **Pastor, Adrian.** Why HttpOnly Won't Protect You. *GNUCITIZEN.* [Online] 12 April 2007. http://www.gnucitizen.org/blog/why-httponly-wont-protect-you/.

63. Let's Encrypt. [Online] https://letsencrypt.org/.

64. SSL Server Test. [Online] https://www.ssllabs.com/ssltest/.

65. SSL Pulse. [Online] https://www.trustworthyinternet.org/ssl-pulse/.

66. **Ristic, Bylvan.** SSL Pulse - To Make SSL More Secure and Pervasive. *Trustworthy Internet Movement.* [Online] 25 April 2012. https://www.trustworthyinternet.org/blog/2012/4/25/ssl-pulse-to-make-ssl-more-secure-and-pervasive/.

67. **SSL Labs.** *SSL Server Rating Guide.* sl : Qualys, 2016.

68. **Open State Foundation.** *Many Dutch government websites not secure.* sl : Open State Foundation, 2016.

69. Pulse. [Online] https://pulse.openstate.eu/https/domains/.

70. Pulse Guidance. [Online] https://pulse.openstate.eu/https/guidance/.

71. **Mill, Eric.** Tracking the U.S. government's progress on moving to HTTPS. *General Services Administration.* [Online] 4 January 2017. https://18f.gsa.gov/2017/01/04/tracking-the-us-governments-progress-on-moving-https/.

72. **Schellevis, Joost.** 'Veel overheidssites hebben geen beveiligde verbinding'. *NOS.* [Online] 1 December 2016. http://nos.nl/artikel/2145883-veel-overheidssites-hebben-geen-beveiligde-verbinding.html.

73. **Van Voorst, Sander.** Plasterk: HTTPS bij overheidssites alleen nodig bij gevoelige gegevens. *Tweakers.* [Online] 9 January 2017. https://tweakers.net/nieuws/119857/plasterk-https-bij-overheidswebsites-alleen-nodig-bij-gevoelige-gegevens.html.

74. **Justitia.** Justitia. *Justitia: Cookiewet.* [Online] http://www.justitia.nl/cookiewet.html.

75. **Dutch government legislation.** Telecommunicatiewet. [Online] 10 March 2017. http://wetten.overheid.nl/BWBR0009950/2017-03-10.

76. Flask Documentation. [Online] http://flask.pocoo.org/docs/0.12/quickstart/#sessions.

77. Fiddler . [Online] http://www.telerik.com/fiddler.

78. Cookies Manager+. [Online] https://addons.mozilla.org/en-US/firefox/addon/cookies-manager-plus/?src=api.

79. **Olsen, Stefanie.** Nearly undetectable tracking device raises concern. *CNET.* [Online] 2 January 2002. https://www.cnet.com/news/nearly-undetectable-tracking-device-raises-concern/.

80. *Hidden surveillance by web sites: web bugs in contemporary use.* **Martin, David, Wu, Hailin en Alsaid, Adil.** 12, New York : ACM, 2003, Vol. 36.

81. **Klein, Matt.** How to Block Third-Party Cookies in Every Web Browser. *How-To Geek.* [Online] 2 April 2016. https://www.howtogeek.com/241006/how-to-block-third-party-cookies-in-every-web-browser/.

82. **EFF.** HTTPS Everywhere. *EFF: HTTPS Everywhere.* [Online] https://www.eff.org/https-everywhere.

83. **Hodges, Jeff, Jackson, Collin en Barth, Adam.** Threat Model. *IETF.* [Online] 2012. https://tools.ietf.org/html/rfc6797#section-2.3. RFC 6797.

84. Wordress Codex: Reference for set_auth_cookie. [Online] https://codex.wordpress.org/Function_Reference/wp_set_auth_cookie.

85. BugMeNoti. [Online] http://bugmenot.com/.

86. **Metz, Rachel.** We Don't Need No Stinkin' Login. *Wired.* [Online] 7 July 2004. https://www.wired.com/2004/07/we-dont-need-no-stinkin-login/.

87. Alexa Traffic Statistics. [Online] http://www.alexa.com/siteinfo/bugmenot.com.

88. Amazon Mechanical Turk. [Online] https://www.mturk.com/mturk/welcome.

89. Scrapy. [Online] https://scrapy.org/.

90. **Hoffman, Pablo.** How To Fill Login Forms Automatically. *Scraping Hub.* [Online] 26 October 2012. https://blog.scrapinghub.com/2012/10/26/filling-login-forms-automatically/.

91. **OWASP.** Cross-Site Request Forgery (CSRF. *OSAWP.* [Online] 2 November 2016. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

92. SeleniumHQ. [Online] http://www.seleniumhq.org/.

93. PhantomJS. [Online] http://phantomjs.org/.

94. Loginform Documentation. [Online] https://github.com/scrapy/loginform.

95. HTTP Status Codes. *REST API Tutorial.* [Online] 23 May 2016. http://www.restapitutorial.com/httpstatuscodes.html.

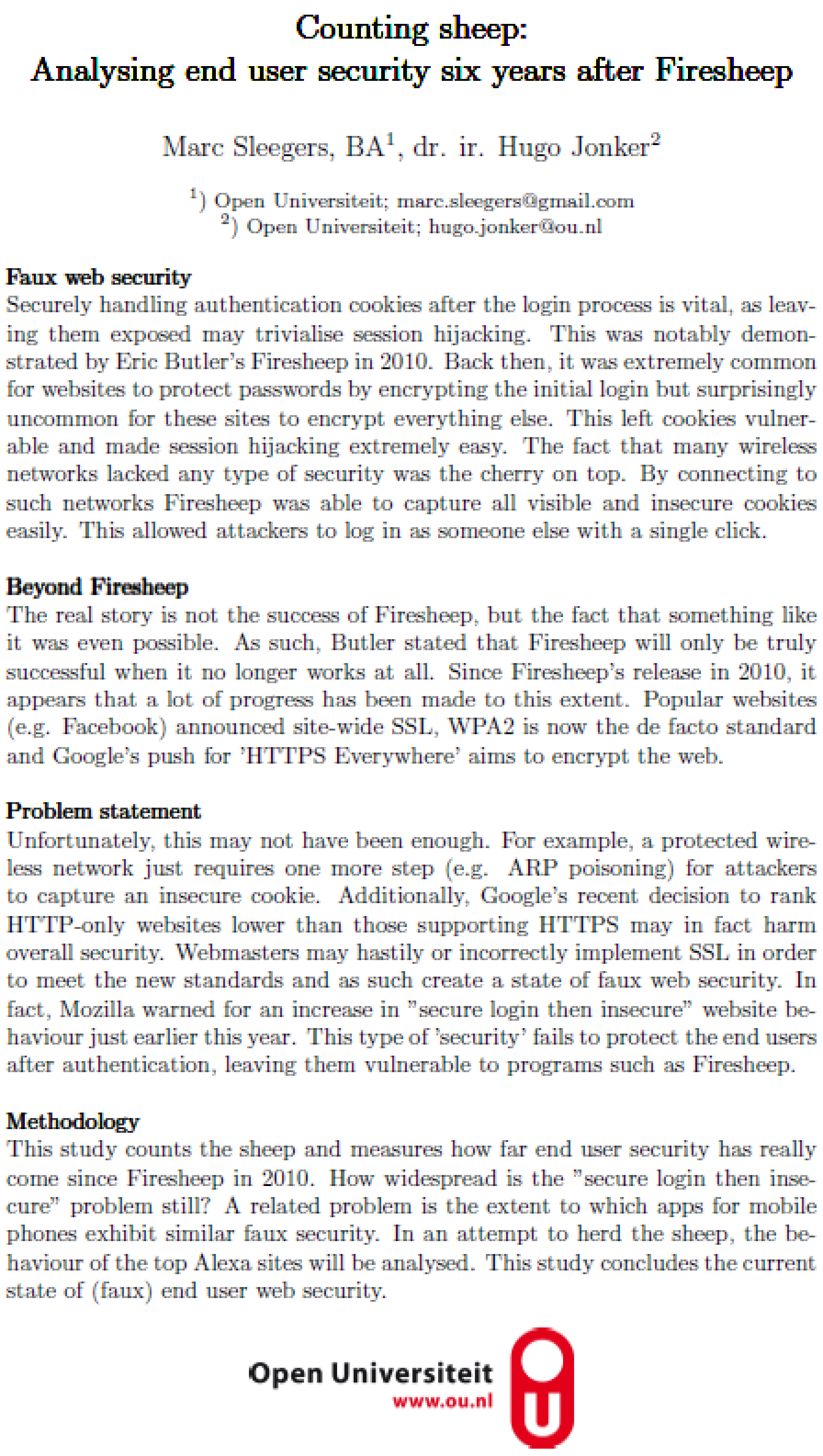
96. **Van Rossum, Guido.** PEP8 -- Style Guide for Python Code. [Online] 5 July 2001. https://www.python.org/dev/peps/pep-0008/. PEP8.

97. **J, Walker en Goodwin, M.** SameDomain Cookie Flag. *IETF.* [Online] 6 April 2016. https://tools.ietf.org/html/draft-west-first-party-cookies-07. Suggested update to RFC6265.

Appendix 1: List of figures

Appendix 2: List of tables

Appendix 3: Abstract for 2nd CSW in the Netherlands



Counting sheep: Analysing end user security six years after Firesheep

Marc Sleegers, BA[1], dr. ir. Hugo Jonker[2]

[1]) Open Universiteit; marc.sleegers@gmail.com
[2]) Open Universiteit; hugo.jonker@ou.nl

Faux web security
Securely handling authentication cookies after the login process is vital, as leaving them exposed may trivialise session hijacking. This was notably demonstrated by Eric Butler's Firesheep in 2010. Back then, it was extremely common for websites to protect passwords by encrypting the initial login but surprisingly uncommon for these sites to encrypt everything else. This left cookies vulnerable and made session hijacking extremely easy. The fact that many wireless networks lacked any type of security was the cherry on top. By connecting to such networks Firesheep was able to capture all visible and insecure cookies easily. This allowed attackers to log in as someone else with a single click.

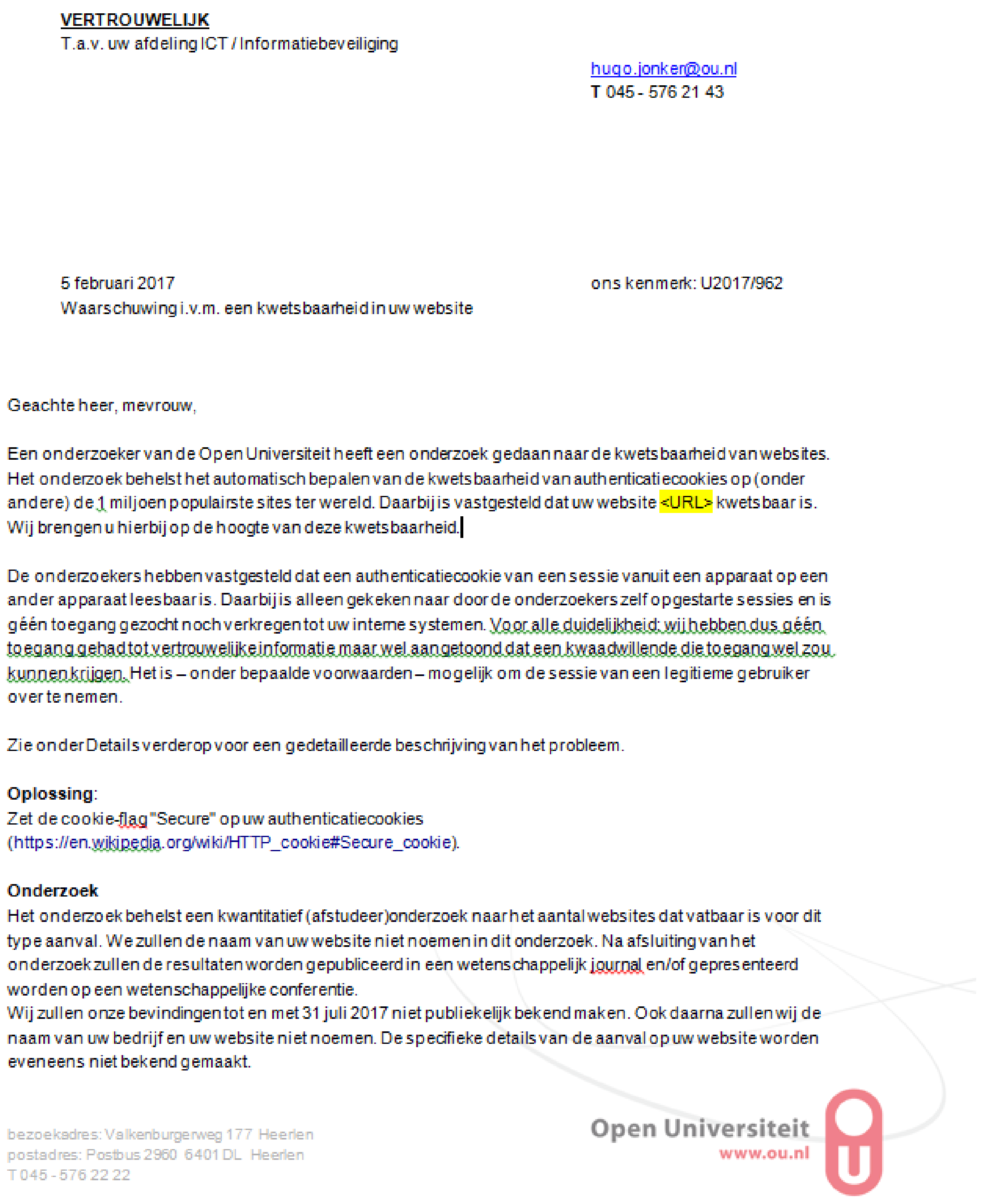
Beyond Firesheep
The real story is not the success of Firesheep, but the fact that something like it was even possible. As such, Butler stated that Firesheep will only be truly successful when it no longer works at all. Since Firesheep's release in 2010, it appears that a lot of progress has been made to this extent. Popular websites (e.g. Facebook) announced site-wide SSL, WPA2 is now the de facto standard and Google's push for 'HTTPS Everywhere' aims to encrypt the web.

Problem statement
Unfortunately, this may not have been enough. For example, a protected wireless network just requires one more step (e.g. ARP poisoning) for attackers to capture an insecure cookie. Additionally, Google's recent decision to rank HTTP-only websites lower than those supporting HTTPS may in fact harm overall security. Webmasters may hastily or incorrectly implement SSL in order to meet the new standards and as such create a state of faux web security. In fact, Mozilla warned for an increase in "secure login then insecure" website behaviour just earlier this year. This type of 'security' fails to protect the end users after authentication, leaving them vulnerable to programs such as Firesheep.

Methodology
This study counts the sheep and measures how far end user security has really come since Firesheep in 2010. How widespread is the "secure login then insecure" problem still? A related problem is the extent to which apps for mobile phones exhibit similar faux security. In an attempt to herd the sheep, the behaviour of the top Alexa sites will be analysed. This study concludes the current state of (faux) end user web security.

Open Universiteit
www.ou.nl

Appendix 4: Responsible Disclosure



VERTROUWELIJK
T.a.v. uw afdeling ICT / Informatiebeveiliging

hugo.jonker@ou.nl
T 045 - 576 21 43

5 februari 2017 ons kenmerk: U2017/962
Waarschuwing i.v.m. een kwetsbaarheid in uw website

Geachte heer, mevrouw,

Een onderzoeker van de Open Universiteit heeft een onderzoek gedaan naar de kwetsbaarheid van websites. Het onderzoek behelst het automatisch bepalen van de kwetsbaarheid van authenticatiecookies op (onder andere) de 1 miljoen populairste sites ter wereld. Daarbij is vastgesteld dat uw website <URL> kwetsbaar is. Wij brengen u hierbij op de hoogte van deze kwetsbaarheid.

De onderzoekers hebben vastgesteld dat een authenticatiecookie van een sessie vanuit een apparaat op een ander apparaat leesbaar is. Daarbij is alleen gekeken naar door de onderzoekers zelf opgestarte sessies en is géén toegang gezocht noch verkregen tot uw interne systemen. Voor alle duidelijkheid: wij hebben dus géén toegang gehad tot vertrouwelijke informatie maar wel aangetoond dat een kwaadwillende die toegang wel zou kunnen krijgen. Het is – onder bepaalde voorwaarden – mogelijk om de sessie van een legitieme gebruiker over te nemen.

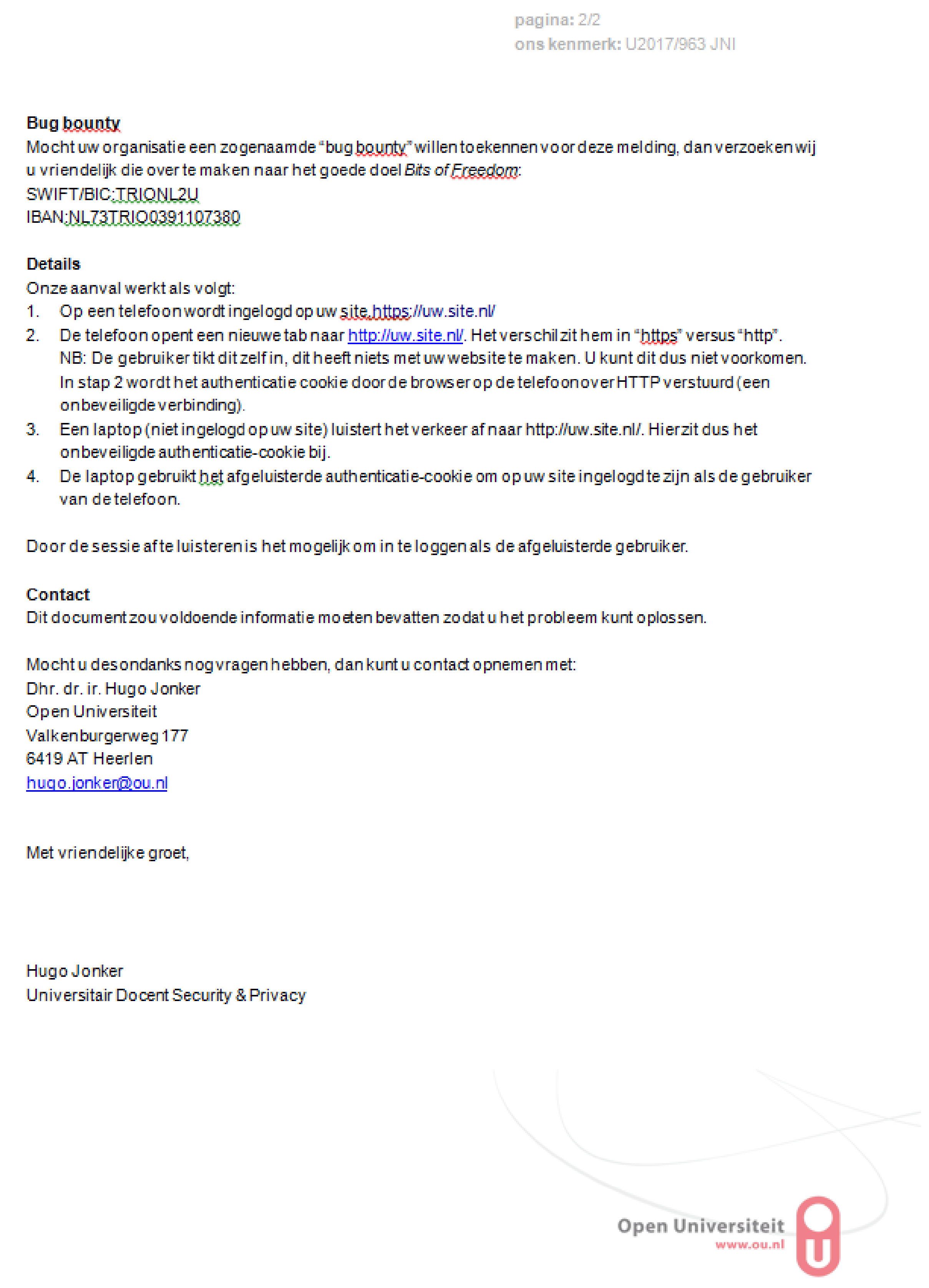
Zie onder Details verderop voor een gedetailleerde beschrijving van het probleem.

Oplossing:
Zet de cookie-flag "Secure" op uw authenticatiecookies (https://en.wikipedia.org/wiki/HTTP_cookie#Secure_cookie).

Onderzoek
Het onderzoek behelst een kwantitatief (afstudeer)onderzoek naar het aantal websites dat vatbaar is voor dit type aanval. We zullen de naam van uw website niet noemen in dit onderzoek. Na afsluiting van het onderzoek zullen de resultaten worden gepubliceerd in een wetenschappelijk journal en/of gepresenteerd worden op een wetenschappelijke conferentie.
Wij zullen onze bevindingen tot en met 31 juli 2017 niet publiekelijk bekend maken. Ook daarna zullen wij de naam van uw bedrijf en uw website niet noemen. De specifieke details van de aanval op uw website worden eveneens niet bekend gemaakt.

bezoekadres: Valkenburgerweg 177 Heerlen
postadres: Postbus 2960 6401 DL Heerlen
T 045 - 576 22 22

Open Universiteit
www.ou.nl



Bug bounty
Mocht uw organisatie een zogenaamde "bug bounty" willen toekennen voor deze melding, dan verzoeken wij u vriendelijk die over te maken naar het goede doel *Bits of Freedom*:
SWIFT/BIC: TRIONL2U
IBAN: NL73TRIO0391107380

Details
Onze aanval werkt als volgt:
1. Op een telefoon wordt ingelogd op uw site https://uw.site.nl/
2. De telefoon opent een nieuwe tab naar http://uw.site.nl/. Het verschil zit hem in "https" versus "http". NB: De gebruiker tikt dit zelf in, dit heeft niets met uw website te maken. U kunt dit dus niet voorkomen. In stap 2 wordt het authenticatie cookie door de browser op de telefoon over HTTP verstuurd (een onbeveiligde verbinding).
3. Een laptop (niet ingelogd op uw site) luistert het verkeer af naar http://uw.site.nl/. Hier zit dus het onbeveiligde authenticatie-cookie bij.
4. De laptop gebruikt het afgeluisterde authenticatie-cookie om op uw site ingelogd te zijn als de gebruiker van de telefoon.

Door de sessie af te luisteren is het mogelijk om in te loggen als de afgeluisterde gebruiker.

Contact
Dit document zou voldoende informatie moeten bevatten zodat u het probleem kunt oplossen.

Mocht u desondanks nog vragen hebben, dan kunt u contact opnemen met:
Dhr. dr. ir. Hugo Jonker
Open Universiteit
Valkenburgerweg 177
6419 AT Heerlen
hugo.jonker@ou.nl

Met vriendelijke groet,

Hugo Jonker
Universitair Docent Security & Privacy