IT Systems Engineering
Universität Potsdam

Network Security in Practice

Web Service Security with TLS

by
Martin Kreichgauer, Daniel Taschik & Johan Uhle

Potsdam, March 2013

Supervisor

Feng Cheng, HPI

Contents

1. Introduction

In recent decades, the World Wide Web has gained exceptional importance in the life of people. For many, it became the main source of information, main means of communication with their friends and family and core part of their daily work. But since the web became this valuable asset, it also attracts malevolent actors. Therefore it is necessary to protect against these.

In this paper we give an overview of security on the World Wide Web and specifically treat the subject of security at the transport layer.

In Section 2 we introduce web security and motivate our concentration on the transport layer. We demonstrate attacking unencrypted web authentication by using the *Firesheep* tool.

In Section 3 we introduce the basics of the Transport Layer Security protocol (TLS).

In Section 4 we highlight some security issues around TLS, specifically using the tools *sslsniff* and *sslstrip*.

In Section 5 we provide *Best Practices* on how to securely set up TLS and mitigate potential attack vectors highlighted in the previous section.

2. Introduction to Web Service Security

Web service security deals with how to secure information on the World Wide Web [12]. In the following section, we will summarise the security goals for a typical web service.

In the World Wide Web we assume a client/server architecture. A client uses a web browser to requests a resource e.g. an html document or an image from a server. Each resource is identified by a unique resource identifier. The communication happens via a stateless application protocol, usually HTTP.

2.1. Security Goals

The following attributes, defined under the term *Information Security* [6], can be used as goals in web security.

Confidentiality Can unauthorized entities access information on the server or information transmitted between server and client?

Integrity Is the data transmitted between server and client unmodified?

Availablity Is the service on the server available to the client?

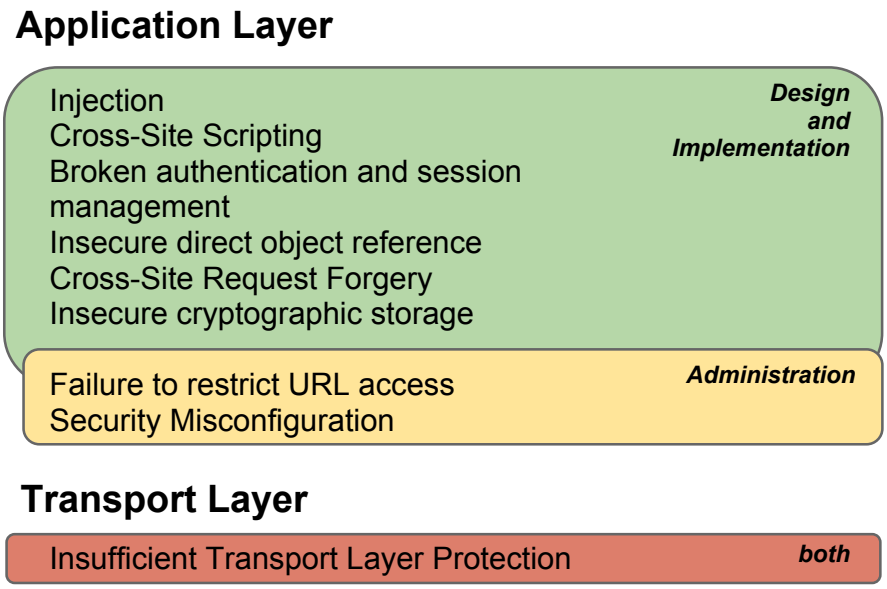

Figure 1: The OWASP Top 10 2010 loosely matched to the OSI model

2.2. Threat Model

The Open Web Application Security Project (OWASP [8]) provides a Top 10 classification of web security attack vectors. It mainly concentrates on the *Confidentiality* security goal. In Figure 1 we loosely mapped the OWASP Top Ten to the OSI model [7]. It can be seen that most attack vectors fall into the category of *Application design and implementation* as well as *Administration* on the *Application Layer*. Our assumption is that with the rise of higher-level web frameworks like Ruby on Rails or Django, attacks on the *Application Layer* will be less effective, since the frameworks come with built-in security features and sensitive defaults. We therefore focused our research on the *Transport Layer*.

2.3. Firesheep

Firesheep[1] is a Mozilla Firefox extension which was developed by Eric Butler in 2010 to demonstrate hijacking of HTTP session cookies. In [19] Butler presents the problem which arises if there is no end-to-end HTTPS connection. User session cookies are transferred unencrypted. If e.g. a victim is in an open wifi or an attacker is redirecting the victims traffic (e.g. arpspoofing) or has control of the network hardware, the attacker can easily scan for session cookies by capturing tcp packages and uses the cookies to login as the user itself.

Many websites only secure the login page with HTTPS. Once logged in via HTTPS they often redirect back to an unencrypted HTTP resource and are thus vulnerable to this attack. For websites which use end-to-end HTTPS and strict transport security this attack does not work.

[1] http://codebutler.com/firesheep/

Setup The tool requires at least Mozilla Firefox 3.6.12 in the 32bit version and the Winpcap package for Windows.

The Firefox extension has a list of websites for which it supports session hijacking by default. Among these are e.g. Amazon, Dropbox, Github, Google and Facebook. Some of these handlers did not work at the time when this paper was written. Examples for handlers working as of November 2012 can be found in Appendix A.

Execution To run a session hijacking attack it is necessary to capture traffic of the victim, e.g. by conducting an ARP spoofing attack[2] or running a packet capture on an unecrypted WiFi network. The attacker starts Mozilla Firefox with the Firesheep extension installed. In the preference pane, the attacker ensures that Firesheep captures traffic to port 80 on hosts on the WiFi. After setting up the appropriate handlers, the attacker can start capturing packages. When another user on the WiFi accesses a web page for which Firesheep has a working handler over plain HTTP, Firesheep captures their session cookie.

The Firesheep sidebar contains an entry representing the hijacked cookie. Clicking on that entry opens a new browser tab, requesting the target site with the hijacked session cookie. The server recognizes the session secret and serves the content for the logged in victim.

We have recorded a screencast of a Firesheep attack which can be viewed at [39].

3. Introduction to Transport Layer Security

Transport Layer Security (TLS) is an application-independent protocol that provides secure communication over the Internet. [16] In an initial handshake, the communicating peers can authenticate each other's identity using asymmetric cryptography, and negoatiate unique, secret keys. Communication is then symmetrically encrypted to provide secrecy, while Message Authentication Codes (MAC) ensure data integrity.

TLS is the standard protocol used today to protect sensitive Internet traffic, from social networking sites to e-commerce, online banking transactions and e-mail. It is usually implemented in libraries providing a socket-like API to the application developer. This makes it possible to use TLS with any protocol on the Application Layer, such as HTTP, POP3, SMTP or FTP. Our work focussed on the use of TLS in the web applications and its use of HTTP over TLS, more commonly known as HTTPS. [35]

In the rest of this section we will give an introduction to the Architecture of TLS, followed by a short overview of its history. We will also discuss different implementations and ways to deploy TLS.

3.1. Architecture

TLS is designed as a layered protocol. The lowest layer is called the *TLS Record Protocol*. It offers support for symmetric encryption and integrity checks of messages. It also encapsu-

[2] `https://en.wikipedia.org/wiki/ARP_spoofing`

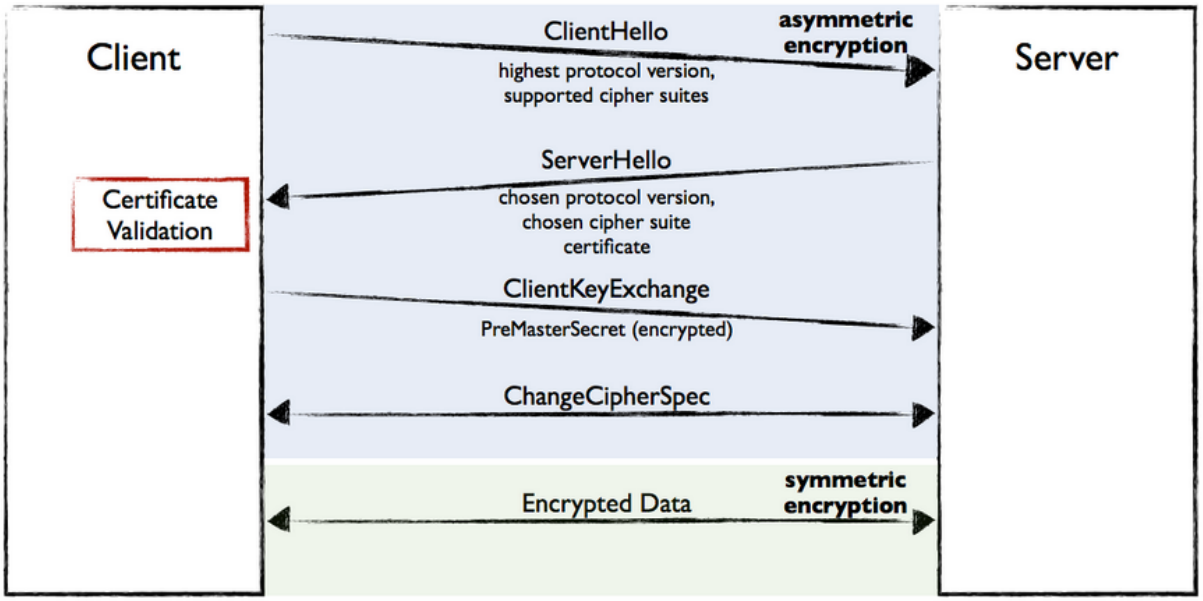

Figure 2: A simple TLS handshake

lates messages of higher-level protocols. One of these is the *TLS Handshake Protocol*. The TLS Handshake Protocol defines messages that allow client and server to authenticate each other using asymmetric public-key cryptography. Both sides also negotiate a cipher suite and a shared secret to symmetrically encrypt their messages.

Figure 2 shows the message flow of a simple TLS handshake where only the server authenticates itself to the client. After the client has opened a connection to the server, both sides send *Hello* messages. The client's Hello message contains a random number and lists of supported TLS protocol versions and cipher suites. A cipher suite is a tuple of cryptographic methods: a key exchange algorithm used in the handshake (e.g. RSA), a symmetric cipher to encrypt messages and a message authentication code (MAC) mechanism to verify message integrity. The server replies with a *Hello* message that also contains a random number and the protocol version and cipher suite it chose to use. After these parameters have been trasnmitted, the server sends a message containing a certificate from the X.509 public key infrastructure (PKI). A certificate contains identity information about the server, a public key and is signed by a trusted authority. The client uses this information to verify the server's identity. We will give more information about X.509 and the process of certificate verification in the following section. After successfully authenticating the server, the client generates the *PreMasterSecret*, a random secret key, encrypts it with the public key from the server certificate and sends that to the server in a *ClientKeyExchange* message. Both sides use the PreMasterSecret and the random numbers exchanged in the *Hello* messages to derive a *MasterSecret*, a 48-byte long completely unique secret, shared only between the two peers, from which all encryption keys for the session can be derived. Now both sides send a message from the *Change Cipher Spec*

protocol to indicate that the Record Protocol Layer will begin using the cryptographic ciphers negotiated in the handshake. All subsequent messages on the connection will be encrypted symmetrically. To conclude the handshake, both sides send *Finished* messages and switch to the *Application Data* protocol to start exchanging application layer data.

It is also possible to authenticate the client against the server, but in general, only one of both peers has to be authenticated. Web applications rarely use TLS for client authentication, but rely on application layer password-authentication instead.

3.1.1. X.509

The format of the certificates used for authenticating peers during the TLS handshake is defined in the *X.509* standard. X.509 is specified by the Standardization Sector of the International Telecommunications Union (ITU-T) and describes a public key infrastructure (PKI). In this PKI, unique public keys are associated with identity information using digital signatures. Such a combination of public key, identity information and signature is called a *certificate*.

Individuals and organizations can present their identity information and public key to a *Certificate Authority* (CA). If the CA can establish the correctness of the identity, e.g. by verifying corresponding legal documentation, it will provide a digital signature and issue the certificate. The digital signature in X.509 is a cryptographic hash of the information contained in the certificate, encrypted with the private key of the CA.

If a server wants to authenticate itself to a client, it sends its certificate. The client has a copy of the certificate of the CA that signed the certificate. It uses the public key from that certificate to decrypt the signature of the server's certificate and retrieve its hash. It then verifies that the hash matches the signed data from the certificate, namely its identity information and public key. If this check succeeds, the client can be certain that it was indeed the CA who issued the digital signature in the certificate and that neither the identity information nor the public key have been modified.

A client, like e.g. a web browser, has a set of CAs that it *trusts* by storing their certificates and accepting their signatures on server certificates. Such a certificate is called a *root certificate*. Not all server certificates are signed by root certificates. A root certificate can be used to sign another certificate, which is not stored in the client. Such a certificate is called an *intermediate certificate*.

Listing 1 is a textual representation of a certificate for `https://www.google.com`. [3] An ASN.1 specification of the exact format and different encodings can be found in [13].

Listing 1: X.509 certificate for www.google.com, retrieved on Jan 3, 2013

```
1 Certificate:
2     Data:
3         Version: 3 (0x2)
4         Serial Number:
5             50:26:32:6b:00:00:00:00:75:af
6         Signature Algorithm: sha1WithRSAEncryption
```

[3] The listing was generated with the `openssl` command-line tool, using `openssl x509 -in <file> -inform DER -text`. Longer text blobs such as keys were shortened.

```
7         Issuer: C=US, O=Google Inc, CN=Google Internet Authority
8         Validity
9             Not Before: Dec  6 08:55:15 2012 GMT
10            Not After : Jun  7 19:43:27 2013 GMT
11        Subject: C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.
              google.com
12        Subject Public Key Info:
13            Public Key Algorithm: rsaEncryption
14            RSA Public Key: (1024 bit)
15                Modulus (1024 bit):
16                    00:d5:5b:55:fc:c4:0f:9c:2c:27:0b:c1:32:d8:49: (...)
17                Exponent: 65537 (0x10001)
18        X509v3 extensions:
19            X509v3 Extended Key Usage:
20                TLS Web Server Authentication, TLS Web Client
                      Authentication
21            X509v3 Subject Key Identifier:
22                96:AB:DD:71:99:60:40:56:71:C4:FD:22:06:CE:ED:FD:4A: (...)
23            X509v3 Authority Key Identifier:
24                keyid:BF:C0:30:EB:F5:43:11:3E:67:BA:9E:91:FB:FC:6A: (...)
25            X509v3 CRL Distribution Points:
26                URI:http://www.gstatic.com/GoogleInternetAuthority/
                      GoogleInternetAuthority.crl
27            Authority Information Access:
28                CA Issuers - URI:http://www.gstatic.com/
                      GoogleInternetAuthority/GoogleInternetAuthority.crt
29            X509v3 Basic Constraints: critical
30                CA:FALSE
31            X509v3 Subject Alternative Name:
32                DNS:*.google.com, DNS:*.android.com, DNS:*.appengine.
                      google.com, (...)
33    Signature Algorithm: sha1WithRSAEncryption
34        bc:db:cd:b1:c6:27:f1:e1:d9:38:18:fa:ee:10:10:f4:ad:3f: (...)
```

3.1.2. Certificate Verification

In addition to the signature check we just described, several other steps have to be taken in order to verify an X.509 certificate in an SSL handshake.

Validity period Lines 8–10 show the validity period of the certificate. A client verifying the certificate has to make sure it still is within this period, i.e. the current date is between *Not Before* and *Not After*.

Certification path validation If a given server has only been signed by an intermediate certificate, there has to be a chain of signing certificates that leads up to a known root certificate. All of the certificates in the chain have to be valid. And, most importantly, every certificate in the chain must have the boolean *CA* field in the *Basic Constraints* section (cp. line 29–30) set to

true. The purpose of this field is to prevent owners of ordinary server certificates from issuing new certificates that are then accepted by clients. CAs therefore must set it to *false* when issuing server certificates. Without this check, active Man-in-the-Middle attacks become possible: Clients would suddenly accept a certificate for *www.yahoo.com* signed by the server certificate of *google.com* because there is a chain of certificates leading up to a trusted Root CA. There have been numerous instances where the test was not implemented, as we will explain in section 4.3.

Host name matching The *Subject* section on line 11 contains the identity associated with this certificate. When using HTTP over TLS, it is critical to make sure that the identity presented on the certificate corresponds to the identity that was expected when the user's web browser made the HTTP request. In HTTPS this is done by comparing the hostname for the server, which can be obtained from the request URI, with the information in the server certificate. There are basically two ways to accomplish this. The deprecated way is to compare against the *Common Name* from the *Subject* field (cp. line 11). Since X.509 was originally designed for use in the X.500 directory services, Subject is an X.500 Distinguished Name (DN) and not really intended to be used with DNS names. Instead, CAs are encouraged to use the *Subject Alternative Name* extension from X.509 v3 instead. As you can see in line 32 of the listing, it contains a sequence of DNS names. It is also possible to specify wildcard names.

Certificate revocation Certificates are intended to be valid for the entire period specified in the Validity section. Occasionally, however, they have to be revoked earlier, e.g. because the corresponding private key was stolen or otherwise compromised. There are two ways to accomplish this. The first is called Certificate Revocation List (CRL). The *CRL Distribution Points* section of the X.509 certificate allows to specify lists of mechanisms to retrieve CRLs combined with on optional reason for the revocation (e.g. compromised private keys). In our example certificate from listing 1 one CRL is given in the form of a URI. A client would download the CRL from that URI and check whether it contains an entry for the certificate it is trying to verify. The RFC for X.509v3 also contains an ASN.1 specification of the CRL format. The second mechanism is the *Online Certificate Status Protocol* (OCSP). In OSCP clients can send requests about the status of a certificate to an OSCP Responder. The reponder will reply with a digitally signed message that indicates whether the status is valid, invalid or unkown.

3.2. History

TLS is based on an earlier security protocol called Secure Sockets Layer (SSL). SSL was originially developed by Netscape. SSL v2 was the first version that Netscape released to the public in 1994. It was later revised several times because it contained numerous severe security flaws. These ultimately led to the release of SSL v3 in 1996, which was essentially a redesign and was not backwards-compatible to SSL v2.

TLS, unlike SSL, is standardized by the Internet Engineering Task Force (IETF) and published in RFCs. The first release, TLS 1.0, took place in 1999 as RFC 2246. [14] TLS 1.1 followed in 2006 to fix a cryptographic vulnerability. [15] The corresponding attack called BEAST is explained in section 4.4. The most recent version, TLS 1.2, changes the pseudo-random function

mentioned in the previous section: While TLS up to version 1.1 relied on a combination of the MD5 and SHA-1 hash functions, TLS 1.2 uses the more secure SHA-256. [16]

In addition to the specifications for each version of TLS, several other RFCs refine and extend the standard. For example, RFC 6176 removes the possibility to renegotiate the use of SSL 2.0 in TLS 1.2.

3.3. Implementations

Developers who want to secure communication of their applications have a number of possibilities depending on the application architecture and environment. Developers of web applications need to configure their web server to serve applications over HTTPS. Since modern web browsers all implement HTTPS and abstractions hide it from application developers, not much has to be done for the client-side implementation of an application. We give a short overview of best practices for server- and client-side HTTPS in section 5. It is noteworthy that as of early 2013, only few browsers support TLS 1.1, and virtually none TLS 1.2.

Developers of applications that use their own application layer protocol instead of HTTP can choose from many available TLS implementations. The two biggest differentiating factors between implementations are licensing model and platform and operating system support. Three widely used free and open-source implementations with support for several large operating systems are *OpenSSL*[4], *GnuTLS*[5] and *Network Security Services (NSS)*[6]. All are available for at least POSIX and Windows platforms. GnuTLS was created as a GPL-compatible alternative to OpenSSL, which is licensed under the Apache License 1.0. NSS evolved from the security code in Netscape Navigator and is still the TLS library used in Mozilla Firefox and Google Chrome. An extensive comparison of different implementations can be found at [2].

Applications that communicate over HTTP but are not web applications, like e.g. many Smartphone applications or command-line clients for HTTP APIs typically use HTTP client libraries that wrap one of the common SSL implementations. Examples include *libcurl*[7], *Apache HttpClient*[8], or the Python module *urllib*[9]. A recent paper by Georgiev et al. has shown that such applications often have certificate validation flaws, mainly caused by confusing APIs and a lack of understanding by application developers. [21]

4. Security of TLS

4.1. Introduction

Even though designed as a secure protocol, TLS has vectors for attacks. A model for this has been introduced by Ivan Ristić, who created a map of the SSL Threat Model [37]. In our

[4] https://www.openssl.org/
[5] http://www.gnutls.org/
[6] http://www.mozilla.org/projects/security/pki/nss/
[7] http://curl.haxx.se/libcurl/
[8] https://hc.apache.org/httpcomponents-client-ga/
[9] http://docs.python.org/2/library/urllib.html

considerations, we limit ourselves to four categories:

1. server-side and configuration issues
2. client-side implementation issues
3. protocol-level and cryptographic issues
4. authenticity framework

In the following, we will show some attack vectors around these categories in the context of web security. We do not try to create a complete listing of threats here, but rather highlight some popular issues. Many of the issues were identified in publications like the OWASP Top 10 [8] or IT-Grundschutz-Baustein B 5.21 Webanwendungen[10]. In our research, we mainly concentrated on the first two categories.

4.2. Server-side Configuration Issues

Unprotected Resources The most obvious security issue is to omit encryption when transmitting sensitive data, because an attacker might be able to eavesdrop on the connection as demonstrated with Firesheep in section 2.3. It is to be noted that the encryption has to be in place for all sensitive data transmissions. For instance in the Firesheep example, the username and password data was protected by using TLS on the login flow, but the session cookie was not protected afterwards and thus opened a vector for attack.

Mixed content When a webpage is transmitted over a secure connection, it is necessary that all linked resources are transmitted over a secure connection as well. If just one element in the page is transmitted over an unsecured connection, an attacker might be able to tamper with that content and compromise all content on the webpage, e.g. by injecting JavaScript or changing an image to misleading content for the user. Current Browsers display a mixed content warning but don't abort the connection. An example of mixed content [33] and the browser response can be seen in Figure 3.

Insecure cookies A standard way in web applications to keep state with a user between requests is by setting cookies in the web browser. A common state to keep is the authentication of a logged-in user. Usually, a secret token authenticating the user is saved in a cookie. On each request, the cookie with the token is transmitted to the server, where after evaluation e.g. access to restricted resources is granted. Cookies in unencrypted connections are prone to theft via eavesdropping, as demonstrated earlier with Firesheep in section 2.3. Thus it is recommended to transmit sensitive cookie information only over a secure connection, by setting the *Secure* flag on a cookie. If this flag is not set, an attacker might be able to force the client to an unencrypted connection, where the cookie would be visible in an eavesdropping attack.

[10] `https://www.bsi.bund.de/DE/Themen/ITGrundschutz/itgrundschutz_node.html`

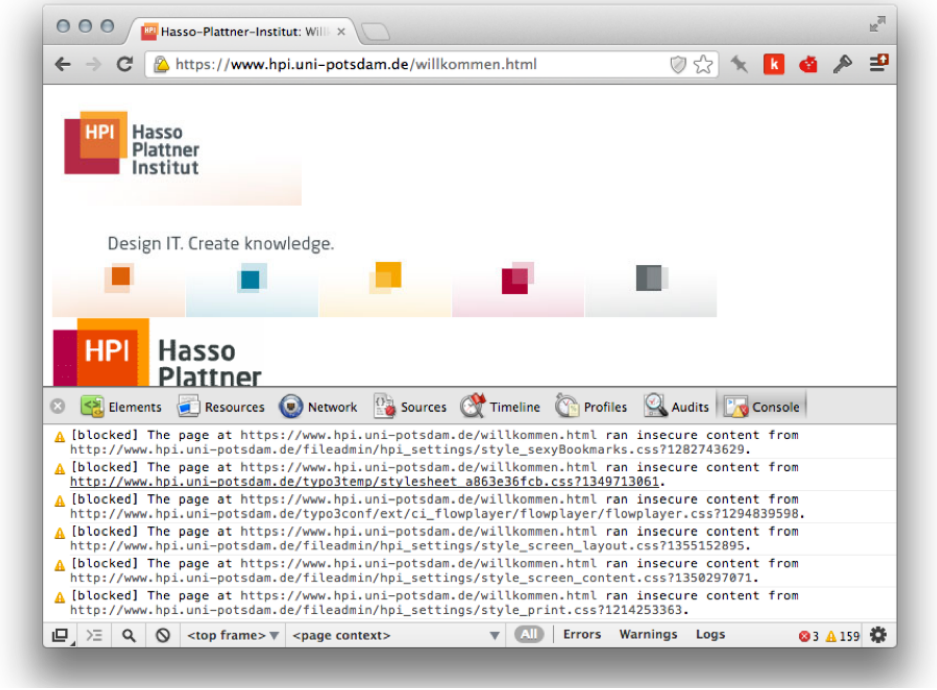

Figure 3: Mixed content on `https://www.hpi.uni-potsdam.de/willkommen.html` in Chrome Version 23.0.1271.95

An example of the correct configuration of a secure cookie on *google.com* can be seen in Figure 4.

Weak protocol versions and ciphers or key exchanges During the TLS Handshake, the client and server negotiate which protocol version, cipher and key exchange to use. Some of these versions are considered weak and stronger versions should be used. We will treat some attacks on weaker versions in section 4.4. Which versions to use is a trade-off between enforcing security by preferring stronger versions and allowing legacy-compatibility to clients with weaker versions. Since the negotation part of the handshake is unencrypted, an attacker performing a man-in-the-middle attack (MITM) can degrade the connection to the weakest versions allowed by both server and client. Thus the connection to a server is only as secure as the weakest version allowed. We will cover which versions to choose in Section 5.

Vulnerable libraries When using SSL, developers will usually not implement the protocol themselves but rather rely on existing libraries. These libraries will issue updates from time to time, sometimes including fixes for security vulnerabilities. As an example, OpenSSL is one of the most popular libraries for TLS, for which in 2012, 5 security advisories and 15 versions including bug fixes and security fixes were released. [3]

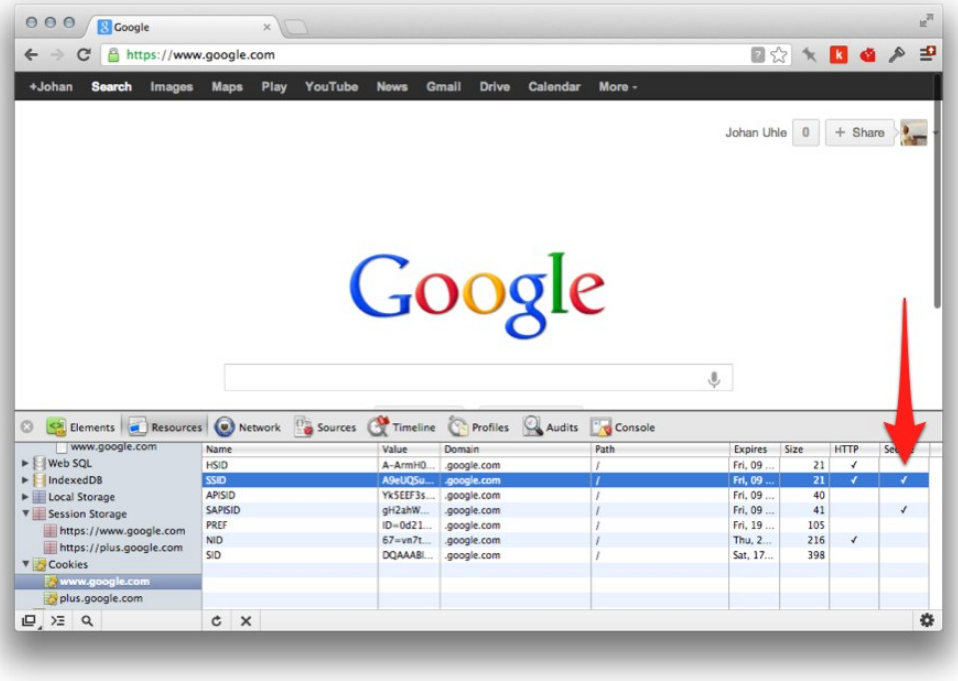

Figure 4: Correctly configured secure cookie on google.com in Chrome Version 23.0.1271.95

4.2.1. Circumventing Encryption with sslstrip

Introduction In 2009 at the BlackHat DC Conference, researcher Moxie Marlinspike [32] introduced an attack on the communication between a web services and a browser. [26] He demoed the attack with the tool sslstrip. [31]

As explained earlier, many web services do not secure all resources they offer, but only secure access to some resources. Typically resources that are secured involve authentication and the transmission of sensitive data.

Given this separation, there is a moment for the browser to switch from loading unsecured resources to secured resources. This happens either 1) with a redirect (HTTP status 301 or 302) or 2) a hyper link to the secured resource (typically <a href="https://[..]>"). Both are sent over a plaintext HTTP connection.

This is the point where sslstrip acts by using a man-in-the-middle attack. Whenever the server uses one of the aforementioned means to switch to a HTTPS connection, sslstrip replaces these towards the client to keep a plaintext HTTP connection, but proxies the requests to HTTPS towards the server. Thus the attacker is able to read the transmitted data, but for the server the connection appears regularly encrypted. In the next section, we will describe the attack in detail.

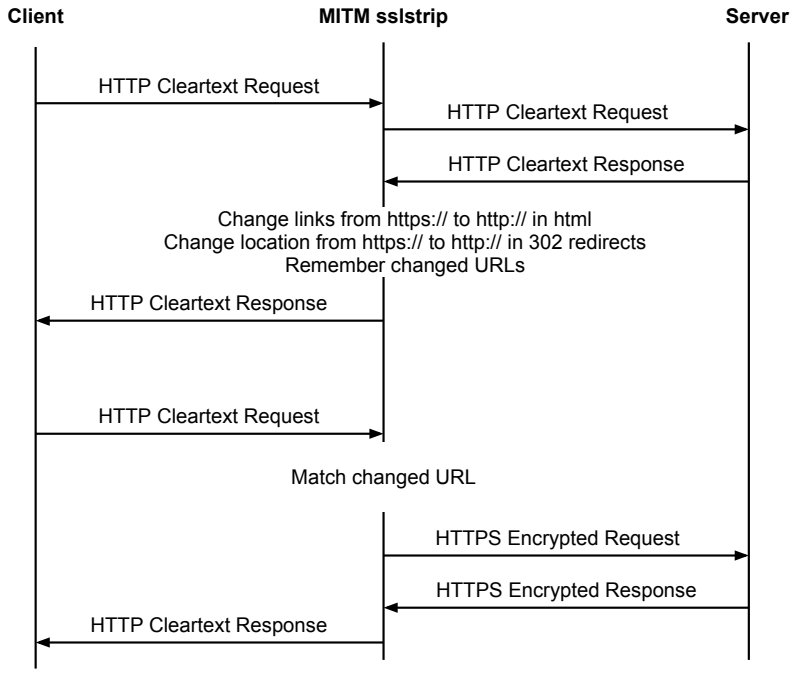

Figure 5: sslstrip attack

Anatomy of the attack The first step for the attacker is to perform a MITM attack against the target's connection to the server. Usually an attacker intercepts the whole traffic between the target and the Internet. A common way for this is ARP spoofing[11].

Once the MITM proxy is set-up, sslstrip will parse the HTTP plaintext responses from the server, looking for redirects or hyperlinks that change the plaintext HTTP connection to an encrypted HTTPS connection. Once such a means is found, sslstrip will change it back to a plaintext format (thus https:// becomes http://), and send it changed to the client, keeping track of all resources that have been changed this way.

When the target now requests one of the changed resources, sslstrip will proxy the plaintext request as a secured request to the server. When it receives the secured response from the server, it proxies the response back to the target in plaintext. In the response, it strips the HTTPS links and redirects again, just as depicted before.

The plaintext requests and responses can now be logged by the attacker and searched for sensitive data like account credentials or session cookies.

We demonstrated the attack against an account login on *www.google.com*. The attacker used sslstrip 0.9 and arpspoof on Ubuntu 10.4.3. The target was running Internet Explorer 10.0.9200.16384 on Windows 8. A screencast of the attack execution is available at [42].

Limitations of the exploit When attacked, the connection from the target's browser to the sslstrip MITM proxy is not encrypted. Browsers indicate this by omitting cues that the con-

[11] `https://en.wikipedia.org/wiki/ARP_spoofing`

nection is secured. Common cues in the address bar include

- displaying the url starting with https://
- displaying a lock symbol
- displaying the name of the website's organization (if an Extended Validation certificate[12] is used)

With these cues missing, a user is able to determine that a connection is not secure. But as demonstrated by [44], users typically do not notice the ways browsers display (or not display) the state of the security of the connection. Thus, it can be assumed that users will not notice the unsecured state of the connection and continue assuming that the connection is secure.

The attack only works if it can intercept the cross-over from a plaintext to a secured connection. For this to work, the client must use plaintext connections to start with. Usually, this is fulfilled whenever the user enters a website address in the browser's address field manually.

An effective counter-measure to this type of attack is the protocol HSTS. It is further explained in section 5.1.5.

4.3. Client-side Implementation Issues

4.3.1. Overview

When using SSL as a client, many issues arise from the incorrect use of SSL libraries or bugs in said libraries. This was shown for example by Georgiev et al. [21] and Fahl et al. [20]. Next we will show one of these attacks stemming from an insufficient client implementation.

4.3.2. A MITM attack on an HTTPS Connection

Introduction Performing a MITM attack on a TLS-secured connection usually does not yield results for an attacker. The packets are encrypted and the attacker usually is not able to decrypt them themselves in reasonable time (with the exception of cryptographic attacks as covered in section 4.4). The only thing the attacker can do is proxy the packets between server and target. But what if the attacker would be able to establish an encrypted connection in both directions, one encrypted connection to the target and another encrypted connection to the server. In this case, the connection data would be available in cleartext to the attacker. In this chapter we will show the anatomy of such an attack with the example of broken hostname validation in the client for null-prefix certificates.

Null-Prefix Attacks using sslsniff To perform a man-in-the-middle attack against HTTPS connections we first need to intercept the traffic between the victim and the Internet, as shown before in Section 4.2.1.

[12] https://en.wikipedia.org/wiki/Extended_Validation_Certificate

The next step is to establish HTTPS connections to both, the client and the server. The connection between attacker and server is trivial, since to the server the attacker is just another client connecting. The connection between attacker and target is the interesting point. To encrypt this connection, the attacker has to use their own private certificate. This certificate has to pass the client's certificate validation (as explained before in section 3.1.2). We will now discuss how to obtain a certificate the target trusts.

Let's look at each certificate verification criteria. Acquiring any certificate that is neither expired nor revoked is easy. Getting one that also has an intact chain of trust is difficult. There are two major ways:

The first one is to install an own CA certificate on the target's machine. Ways to achieve this are social engineering to trick the target into installing the attacker's root certificate or the attacker breaking into the target's machine to install the certificate themselves. Both ways are impractical since they are time consuming and work differently (if at all) for each target.

The second one is to get a certificate signed by a CA the target trusts. Given that there is a number of CAs that every browser trusts today, the attacker will have to go to one of these CAs to acquire a certificate. During the issuing process, the CAs are bound to check that the certificate applicant is permitted to register a certificate for said domain. Tricking a CA to issuing a fraudulent certificate (e.g. by social engineering or compromising their infrastructure) is likely possible but infeasible for our purposes.

Certificate with a NULL character in the common name When a client validates a certificate, they have to match the currently accessed domain with the common name field in the certificate. For example, if the client accesses *www.dropbox.com* and the certificate has the field *CN=*.dropbox.com*, the check is valid. On the other hand, if the certificate's common name field is e.g. *CN=*.attacker-website.com* the validation fails.

But in 2009 at the BlackHat US 09 conference, Moxie Marlinspike published a vulnerability of this check in some TLS client implementations [27].

These use the *strcmp()* function to compare the current domain and the allowed common name from the certificate. The common name attribute from a X.509 certificate is to be interpreted as a pascal string, but *strcmp()* interprets the string as a C string, which are terminated by a NULL byte (*\x00*). So if the attacker generates a certificate with a common name like *CN=*.dropbox.com\x00.attacker-website.com*, the client will only compare the string until the NULL character and omit the later part of the string in the comparison. I.e., in our example it assumes the certificate is valid for **.dropbox.com*. An attacker is able to acquire a certificate with such a common name from a CA since a NULL character is interpreted as a normal character in the subdomain.

Executing the Attack with sslsniff *sslsniff* [30] is a tool that was originally introduced by Moxie Marlinspike in 2002 [32] as a proof-of-concept for a similar security vulnerability involving the check of the Basic Constraint flag in X.509 certificates. [25] The tool allows a MITM attacker to switch certificates on a secured connection. This can be targeted to specific domains only, whereas the rest of the traffic is left untouched. In our specific case, we can use the NULL character certificate to intercept a secure connection to *www.dropbox.com*.

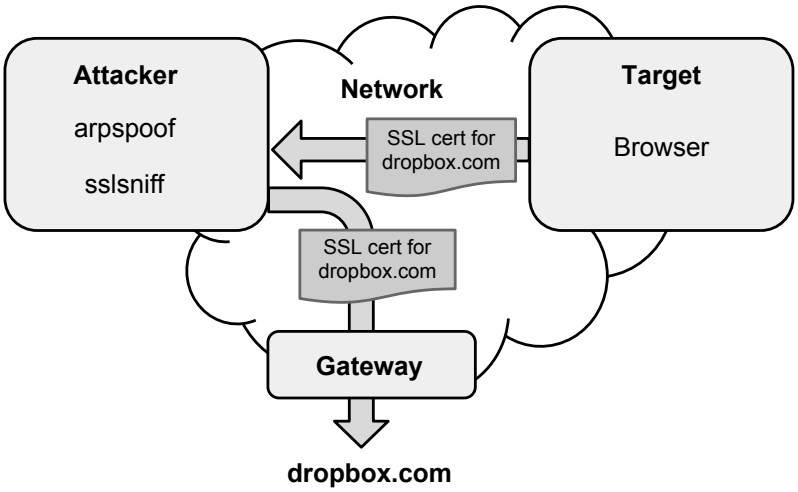

Figure 6: Overview of the attack with sslsniff

In our tests, we successfully executed the attack using sslsniff 0.8 on Ubuntu 10.4.3 and arpspoof on the attacker machine. The target was running Mozilla Firefox 2.0.0.20 on Windows 8 and accessed *www.drobox.com*. Since this test was only for academic purposes, we simplified the test setup by not acquiring a real certificate, but instead generated our own CA and NULL character certificate.

A screencast of the attack is available at [41]. An overview over the setup of the attack is visible in Figure 6.

4.4. Cryptographic attacks

As explained in section 3, TLS makes use of several cryptographic functions. These are used in various steps of implementing the protocol and users can choose which exact function to use in each step. Some of these functions have proven vulnerabilities and we will now introduce some attack vectors based on such weaknesses.

MD5 collisions MD5 is a cryptographic hash function. In TLS, a hash function is used to validate the chain of trust for X.509 certificates (as described in section 3.1.1). It has been shown by Wang et al. that MD5 is prone to collisions. [43] In [38] Stevens et al. showed that by exploiting an MD5 hash collision, it it possible to create a rogue CA certificate that is trusted by browsers. In response to the attack, CAs do not issue certificates with MD5 signatures anymore [13], but rather use other functions preferably from the SHA family.

BEAST To encrypt data payload, TLS uses a chosen cipher. When using a CBC cipher, TLS can be prone to the BEAST attack. [17] It exploits a vulnerability shown by Brad [10] [11] that sensitive information in a CBC encrypted stream can be extracted via brute force guesses

[13] http://www.win.tue.nl/hashclash/rogue-ca/#sec71

when knowing the initialization vector. Among the proposed counter measures are enforcing a stream cipher like RC4 on the server, updating to TLS 1.1 (which might not be supported by clients) or prepending each encrypted packet with an empty packet, which forces a new random initialization vector (which breaks some TLS implementations). [45] [5]

CRIME The CRIME attack [18] is similar to the BEAST attack, in that sensitive information can be derived from an encrypted connection. Here, the exploited characteristic is information leakage if compression is used. [23] As a counter measure, most browsers have disabled compression for headers, when using TLS.[14]

4.5. Authenticity Framework

While the vast majority of security issues can be fixed by revising parts of the implementations or even the protocol specification itself, a major problem arises from the role CAs play in the authenticity of the system. Security incidents like the compromise of the DigiNotar CA demonstrated the issues of the Certification Authority system when attackers were able to issue fraudulent certificates for Google services which were later used to conduct man-in-the-middle attacks on Iranian Internet users. [1]

Moxie Marlinspike has written about the flaws and is also the author of the proposed solutions which we describe in this section. [28] [29] According to him, the issue can be best described as a lack of *Trust Agility*: In TLS, trust is largely based on the inclusion of an organization's certificate in the Root CA store. The decision to include a certificate is difficult to reverse because doing so has the potential to disrupt many services relying on the inclusion of a certain root certificate. Similarly, if an individual user were to explicitly mistrust one Certification Authority, they would lose trust to all services and websites that have used that CA to establish their authenticity.

4.5.1. Convergence

Convergence is a system that is aimed at replacing the current Certificate Authority model with one that offers trust agility. It was first described by Marlinspike in [29]. In Convergence, a website's authenticity is not based on a single CA but rather on several *Notaries*. Clients verify a server's certificate with every one of the notaries. In the case of a disagreement, the user can choose whether a single notary's trust suffices, the majority decision is trusted or consensus between all parties is required.

4.5.2. Public Key Pinning

While Convergence is basically a separate authenticity infrastructure, there are also proposals to address the problem with authenticity differently. One of these is Public Key Pinning.

[14] e.g. `https://chromiumcodereview.appspot.com/10825183`

Public Key Pinning refers to a method to detect fraudulent certificates by trusting only certificates containing a public key which has been explicitly white-listed (*pinned*) for the site that it authenticates. Google Chrome implements Public Key Pinning for most Google services. [24] Since Google maintains the list of pinned keys manually, this approach does not scale for the entire web. Two mechanisms that are more scalable have been proposed lately.

TACK TACK (*Trust Assertions for Public Keys*) is an implementation of Public Key Pinning at the TLS level. [9] It uses a separate key pair, called the *TACK signing key* (TSK). Upon request, a complying server sends an TLS extension message containing a *TACK*. A TACK contains copies of all TLS public keys, signed by the TSK private key, the TSK public key and a set of metadata. Clients keep a copy of the TACK. When they communicate with the same site again, they can check whether the public key the server uses is valid by checking if it is signed in the previously stored TACK. Servers can also revoke public keys by publishing a newer version of the TACK that does not contain that key.

Public Key Pinning Extension for HTTP There is another approach that implements public key pinning at the HTTP level. [4] It proposes for servers to include a *Public-Key-Pins* HTTP header in their response. This header contains a list of pins, which are pairs of cryptographic hashes of a public key and a max-age directive. As in TACK the client stores a copy of the pins and compares them against the keys used in later connections.

5. Best Practices for TLS

In this section, we discuss best practices for the usage of TLS in the context of HTTP. We will take into account server- and client-side configuration. Failure to properly configure TLS on a server might e.g. lead to a possible loss of confidentiality. Many active attacks, such as man-in-the-middle, also exploit the carelessness or ignorance of end-users. The following section explains best practices for both sides.

5.1. Server Configuration

A comprehensive summary of best practices for a TLS server configuration can be found in [36]. In the following sections we will highlight the most important aspects.

5.1.1. Private Key length

When using RSA, it is important to use private keys of at least 2048 bit to be protected against prime factoring attacks in the coming years.

5.1.2. Supported TLS versions

When a user agents connects to a web server via HTTPS, both parties negotiate the cryptographic protocol they are going to use. As explained in Section 3.1, it is up to the server to choose the exact cipher suite, and it usually uses the strongest protocol both parties support.

Despite having been released in 2008, the latest version of TLS, 1.2, is still not supported by most browsers so far. On the other hand, SSLv3 has been around for more than 16 years and is thus widely supported by all implementations and libraries. However, SSLv3 uses the MD5 cryptographic hash function which is known to be vulnerable to collision attacks. It also uses weaker cipher suites than later versions of TLS. It is thus not advisable to use SSLv3.

For greater backwards-compatibility, TLS allows to downgrade the protocol to a lower version. This potentially allows for an active MITM attacker to force the connection to more insecure protocol version that makes it easier to compromise the communication. Downgrading to SSLv2 has thus already been prohibited in [40]. It is also advisable to not offer SSLv3 for the same reason, unless required for compatibility with old clients. This leaves TLS 1.0 as the recommended lowest protocol version a server should offer.

5.1.3. Cipher suites

Cipher suites specify which ciphers are used to communicate securely between server and client. They are negotiated at the handshake. Every security protocol defines a set of key exchange algorithms, hash functions, authentication and encryption methods. It is not trivial to choose the right cipher suite. Configuring a server with the right cipher suits depends on the used TLS implementation.

OpenSSL for example offers a variety of cipher suites. This diversity is deliberate: If one mechanism turns out to be vulnerable, another mechanism can be used to offer secure communication. A trivial example for insecure cipher suites are NULL suites, which offer no encryption at all. Also cipher suites relying on Cipher Block Chaining (CBC) should be avoided because they are vulnerable to the BEAST attack as discussed in Section 4.4.

Supported cipher suites are configured using cipher lists. A cipher list is a colon-separated list of cipher strings. [34] offers detailed information about the usage of OpenSSL cipher suites.

Listing 2: nginx cipher suite configuration for TLS 1.0/1.1/1.2

```
1 ssl_ciphers HIGH:RC4:+HIGH+TLSv1:!aNULL:!eNULL:!MD5;
```

Listing 2 shows a secure configuration example. It only allows encryption cipher suites with a key length of at least 128bit. In addition, it allows cipher suites using RC4 and disables all algorithms that offer no encryption rely on the MD5 cryptographic hash function as a MAC.

5.1.4. Redirects

As demonstrated in Section 2.3, it is not enough to secure only the login or the post of the login credentials. The best way to achieve a permanent HTTPS connection is to redirect any

request arriving on HTTP port 80 to the HTTPs port 443. This can be easily done by the rewrite module of Apache web server or simple rewrite line in a nginx configuration server container. Listing 3 and 4 show exemplary how a permanent HTTPS connection can be achieved.

Listing 3: Apache global rewrite from HTTP to HTTPS

```
RewriteEngine   on
RewriteCond     %{SERVER_PORT} ^80$
RewriteRule     ^(.*)$ https://%{SERVER_NAME}$1 [L,R]
RewriteLog      "/var/log/apache2/rewrite.log"
RewriteLogLevel 0
```

Listing 4: nginx global rewrite from HTTP to HTTPS

```
server {
    listen       80;
    server_name  _;
    rewrite ^(.*) https://$host$1 permanent;
}
```

5.1.5. HTTP Strict Transport Security

Even when TLS redirect have been enabled on all resources, users are still vulnerable to SSL stripping attacks as shown in Section 4.2.1 because the initial redirect can be tampered with by an active MITM attacker.

HTTP Strict Transport Security (HSTS) has been specified in RFC 6797 [22] and addresses this issue. It gives websites the capability to state themselves as being only accessible via HTTPS. Every resource of that webpage must be transmitted via a secure connection. This mitigates the risk of passive network attackers (e.g. session hijacking), active network attackers (e.g. MITM attacks) and faulty website implementations or deployment.

When an HSTS supporting web browser visits an HSTS enabled website for the first time, it stores that HSTS is enabled for this website. Whenever the user visits that website again, the browser enforces a secure connection: All resources on the website have to be transmitted via HTTPS. The browser will block access to the site if that is not the case.

Additionally, issues with the certificate, like a failing chain-of-trust validation, also lead to the browser terminating the connection. That prevents end-users from accidentally ignoring security warnings about forged certificates.

HSTS is currently supported by the following browsers: Google Chrome (Version 4.0.211.0)[15], Mozilla Firefox (Version 4)[16], Opera (Version Presto 2.10)[17]. Apple Safari and Microsoft Internet Explorer do not support HSTS as of November 2012.

HSTS uses the security principle of Trust-on-First-Use (TOFU). Some browsers like Google Chrome and Mozilla Firefox come with a preloaded list of known HSTS enabled websites.

[15] http://lists.w3.org/Archives/Public/public-webapps/2009JulSep/1148.html
[16] https://blog.mozilla.org/security/2010/08/27/http-strict-transport-security/
[17] http://www.opera.com/docs/specs/presto2.10/#m210-244

Chrome's list of known websites can be found here [18].

Enabling HSTS on the server The web server enables HSTS for a website by setting the Strict-Transport-Security HTTP response header field. This header field contains one required and one optional argument:

- max-age (required)
- includeSubdomains (optional)

The max-age directive specifies for how long a user agent will consider the contacted host as an HSTS enabled host. Its number is given in seconds. To increase security the value should be set to a very high value like 6 months or more. If the value is "0" it signals the browser to remove the host from the list of known HSTS hosts.

IncludeSubdomains is optional and if present declares that in addition to the host itself, all subdomains of the requested host will be handled as HSTS enabled as well.

All major web servers support modification of HTTP headers and hence support HSTS. Listings 5 and 6 show how HSTS can be enabled in Apache web server as well as in nginx web server.

Listing 5: enable HSTS in Apache

```
1 Header always set Strict-Transport-Security "max-age=32140800;␣
    includeSubDomains"
```

Listing 6: enable HSTS in nginx

```
1 add_header Strict-Transport-Security "max-age=32140800;includeSubdomains";
```

5.2. Blackbox Testing

A useful means to verify secure HTTPS configuration is blackbox testing. One of the available black box testing tools is the Qualys SSL Server Test[19]. It analyses the server certificate and chain-of-trust, the supported protocols and cipher suites. It returns a rating from A to F, that indicates the security of the configuration, and explicitly lists all flaws present in the server configuration.

5.3. Client-side Considerations

When implementing TLS in a client-side application, developers can choose from one of the implementations previously listed in Section 3.3. Generally it is important to choose an up-to-date version of a well-supported, widely used library and be aware of vulnerabilities and security patches. Since TLS is inherently complex, most of the libraries are not trivial to use.

[18] http://dev.chromium.org/sts
[19] https://www.ssllabs.com/ssltest/

It is thus essential to be familiar with proper and secure use of the library. As previously mentioned, it has been demonstrated that security issues due to misuse of TLS libraries are widespread. [21] Developers should especially make sure that proper certificate validation takes place since some implementations either don't implement it fully or, e.g., require users to provide an implementation for host name verification.

Lastly, public key pinning, i.e. restricting the server's public keys to a set that is whitelisted in the client application, can drastically improve security, since it prevents active MITM attacks.

6. Conclusion

The World Wide Web developed tremendously over the past two decades. Its widespread usage in almost every field like private, business or scientific use, requires solid and reliable security. Without a decent security e.g. online banking, e-commerce, dealing with personal data, secure communication and many more can not work in the Internet. Fraud and crime could not be prevented at all, no guarantees for the privacy of a user could be made. But assuring proper web security is very hard and requires the responsibility of many. Server administrators, developers and end-users are all in charge to assure the secure usage of the aforementioned services and applications.

In this paper, we gave an introduction to the motivation and goals behind web security. We demonstrated how to trivially access unprotected communication using Firesheep. We presented two practical weaknesses of TLS: one caused by a weak server configuration and another by a faulty client implementation. We discussed shortcomings of the current Certificate Authority system. Furthermore we took a look at protocol-level and cryptographic weaknesses. Finally, we offered best practices for a secure server configuration and client-side implementation of TLS.

In this paper we have shown that web security is a complex topic. While not claiming to be complete, we hope to give developers insight on how to implement TLS correctly and encourage people to be skeptical about end-user security in HTTPS.

A. Firesheep Handlers

Listing 7: Facebook Firesheep Handler

```
register({
  name: 'Facebook',
  url: 'https://www.facebook.com/home.php',
  domains: [ 'facebook.com' ],
  sessionCookieNames: [ 'datr', 'c_user', 'lu', 'xs' ],

  processPacket: function () {
    var cookies = this.firstPacket.cookies;
    this.sessionId = cookies.c_user + cookies.xs;
  },

  identifyUser: function () {
    var resp = this.httpGet(this.siteUrl);
    this.userName = resp.body.querySelector('.headerTinymanName').
        innerHTML;
    this.userAvatar = resp.body.querySelector('.headerTinymanPhoto').src;
  }
});
```

Listing 8: Stack Overflow Firesheep Handler

```
register({
  name: 'Stack Overflow',
  url: 'http://stackoverflow.com/',
  domains: [ 'stackoverflow.com' ],
  sessionCookieNames: [ 'usr', '__utmz', '__utma' , '__qca' ],

  identifyUser: function () {
    var resp = this.httpGet(this.siteUrl);
    this.userName = resp.body.querySelectorAll('a')[3].textContent;
  }
});
```

Listing 9: Flickr Firesheep Handler

```
register({
  name: 'Flickr',
  url: 'http://www.flickr.com/me',
  domains: [ 'flickr.com' ],
  sessionCookieNames: [ 'cookie_session' ],

  identifyUser: function () {
    var resp = this.httpGet(this.siteUrl);
    var path = resp.request.channel.URI.path;
    this.userName = path.split('/')[2];
    this.userAvatar = resp.body.querySelector('.Buddy_img').src;
  }
});
```

References

[1] DigiNotar, .
URL https://en.wikipedia.org/wiki/DigiNotar.

[2] Comparison of TLS implementations, .
URL https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations.

[3] OpenSSL News.
URL http://openssl.org/news/.

[4] Public Key Pinning Extension for HTTP, 2011.
URL https://tools.ietf.org/html/draft-ietf-websec-key-pinning-01.

[5] Rizzo/Duong chosen plaintext attack (BEAST) on SSL/TLS 1.0, 2011.
URL https://bugzilla.mozilla.org/show_bug.cgi?id=665814.

[6] Information Security, 2012.
URL https://en.wikipedia.org/wiki/Information_security.

[7] OSI model, 2012.
URL https://en.wikipedia.org/wiki/OSI_model.

[8] Open Web Application Security Project, 2012.
URL https://owasp.org/.

[9] Trust Assertions for Certificate Keys draft-perrin-tls-tack-02.txt, 2013.
URL http://tack.io/draft.html.

[10] G. V. Bard.
Vulnerability of SSL to Chosen-Plaintext Attack.
2004.

[11] G. V. Bard.
A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL.
2006.

[12] T. Berners-Lee.
Information Management: A Proposal.
Technical report, CERN, 1989.
URL http://www.w3.org/History/1989/proposal.html.

[13] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk.
Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.
RFC 5280 (Proposed Standard), May 2008.
URL http://www.ietf.org/rfc/rfc5280.txt.

[14] T. Dierks and C. Allen.
The TLS Protocol Version 1.0.
RFC 2246 (Proposed Standard), Jan. 1999.
URL `http://www.ietf.org/rfc/rfc2246.txt`.
Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[15] T. Dierks and E. Rescorla.
The Transport Layer Security (TLS) Protocol Version 1.1.
RFC 4346 (Proposed Standard), Apr. 2006.
URL `http://www.ietf.org/rfc/rfc4346.txt`.
Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176.

[16] T. Dierks and E. Rescorla.
The Transport Layer Security (TLS) Protocol Version 1.2.
RFC 5246 (Proposed Standard), Aug. 2008.
URL `http://www.ietf.org/rfc/rfc5246.txt`.
Updated by RFCs 5746, 5878, 6176.

[17] T. Duong and J. Rizzo.
Here Come The XOR Ninjas.
2011.

[18] T. Duong and J. Rizzo.
The CRIME attack.
Ekoparty, 2012.

[19] Eric Butler.
Start protecting user privacy instead of pretending to, 2010.
URL `http://codebutler.github.com/firesheep/tc12/`.

[20] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben.
Why Eve and Mallory Love Android : An Analysis of Android SSL (In) Security Categories and Subject Descriptors.
pages 50–61, 2012.

[21] M. Georgiev, S. Jana, and D. Boneh.
The Most Dangerous Code in the World : Validating SSL Certificates in Non-Browser Software.
pages 38–49, 2012.

[22] J. Hodges, C. Jackson, and A. Barth.
HTTP Strict Transport Security (HSTS).
RFC 6797 (Proposed Standard), Nov. 2012.
URL `http://www.ietf.org/rfc/rfc6797.txt`.

[23] J. Kelsey.
Compression and Information Leakage of Plaintext.
2002.

[24] A. Langley.
Public key pinning, 2011.
URL http://www.imperialviolet.org/2011/05/04/pinning.html.

[25] M. Marlinspike.
Internet Explorer SSL Vulnerability, 2002.
URL http://www.thoughtcrime.org/ie-ssl-chain.txt.

[26] M. Marlinspike.
New Tricks For Defeating SSL In Practice.
BlackHat DC, 2009.
URL https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf.

[27] M. Marlinspike.
More Tricks For Defeating SSL In Practice.
BlackHat USA, 2009.
URL https://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf.

[28] M. Marlinspike.
SSL And The Future Of Authenticity, 2011.
URL http://www.thoughtcrime.org/blog/ssl-and-the-future-of-authenticity/.

[29] M. Marlinspike.
BlackHat USA 2011: SSL And The Future Of Authenticity, 2011.
URL https://www.youtube.com/watch?v=Z7Wl2FW2TcA.

[30] M. Marlinspike.
sslsniff 0.8, 2011.
URL http://www.thoughtcrime.org/software/sslsniff/.

[31] M. Marlinspike.
sslstrip 0.9, 2011.
URL http://www.thoughtcrime.org/software/sslstrip/.

[32] M. Marlinspike.
thoughtcrime.org, 2012.
URL http://www.thoughtcrime.org/.

[33] Mozilla Foundation.
MixedContent, 2012.
URL https://developer.mozilla.org/en-US/docs/Security/MixedContent.

[34] OpenSSL.
OpenSSL Project - Ciphers, 2013.
URL http://www.openssl.org/docs/apps/ciphers.html#CIPH.

[35] E. Rescorla.
HTTP Over TLS.

RFC 2818 (Informational), May 2000.
URL http://www.ietf.org/rfc/rfc2818.txt.
Updated by RFC 5785.

[36] I. Ristić.
SSL/TLS Deployment Best Practices.
URL https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices_1.0.pdf.

[37] I. Ristić.
SSL Threat Model, 2009.
URL http://blog.ivanristic.com/2009/09/ssl-threat-model.html.

[38] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger.
MD5 considered harmful today - Creating a rogue CA certificate, 2008.
URL http://www.win.tue.nl/hashclash/rogue-ca/.

[39] D. Taschik.
Hijacking a Facebook session using Firesheep, 2012.
URL http://youtu.be/ftDw7NiWaAA.

[40] S. Turner and T. Polk.
Prohibiting Secure Sockets Layer (SSL) Version 2.0.
RFC 6176 (Proposed Standard), Mar. 2011.
URL http://www.ietf.org/rfc/rfc6176.txt.

[41] J. Uhle.
MITM attack on Dropbox on Firefox 2.0 with sslsniff and Null character certificate, 2012.
URL http://www.youtube.com/watch?v=Q4QhEbVr96k.

[42] J. Uhle.
Using sslstrip to obtain Google account credentials via MITM, 2012.
URL http://www.youtube.com/watch?v=x-EwHKrfUGc.

[43] X. Wang and H. Yu.
How to Break MD5 and Other Hash Functions.
2006.

[44] T. Whalen.
Gathering Evidence : Use of Visual Security Cues in Web Browsers.
pages 137–144, 2002.

[45] T. Zoller.
The BEAST summary - TLS, CBC, Countermeasures.
URL http://blog.zoller.lu/2011/09/beast-summary-tls-cbc-countermeasures.html.