CHALMERS

GlassTube

A Lightweight Approach to Web Application Integrity

Per A. Hallgren
Daniel T. Mauritzson

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2012

Master's Thesis 2012

GlassTube: A Lightweight Approach to Web Application Integrity

DANIEL MAURITZSON
PER HALLGREN

©DANIEL MAURITZSON, May 2012.
©PER HALLGREN, May 2012.

Examiner: ANDREI SABELFELD

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Goteborg
Sweden
Telephone + 46 (0)31-772 1000
Department of Computer Science and Engineering
Goteborg, Sweden May 2012

Abstract

The HTTP and HTTPS protocols are the main corner stones of the modern web. From a security point of view, they offer an all-or-nothing choice to web applications: either no security guarantees with HTTP or both confidentiality and integrity with HTTPS. However, in many scenarios confidentiality is not necessary and even undesired, while integrity is essential to prevent attackers from compromising the data stream.

We propose GlassTube, a lightweight approach to web application integrity. GlassTube guarantees integrity at application level, without resorting to the heavyweight HTTPS protocol. GlassTube provides a general method for integrity in web applications and smartphone apps. GlassTube is easily deployed in the form of a library on the server side, and offers flexible deployment options on the client side: from dynamic code distribution, which requires no modification of the browser, to browser plugin and smartphone app, which allow smooth key predistribution. The results of a case study with a web-based chat indicate a boost in the performance compared to HTTPS, achieved with no optimization efforts.

Acknowledgments

The creation of the GlassTube protocol has been performed as a Master Thesis work in Computer Science and Engineering at Chalmers University of Technology. The work was performed at OmegaPoint Göteborg AB. We wish to thank:

- *Andrei Sabelfeld* for being our examiner, for truly committing to the project and caring about our progress.
- *Peter Magnusson* for supervising the thesis work. Thank you for all your input and feedback, your knowledge and competence has been invaluable.
- *OmegaPoint Göteborg AB* thank you for a friendly and inspiring work environment. Thank you the coffee, without it the project would surely have failed!

Contents

1

Introduction

With the overwhelming expansion of the world wide web and increasing reliance on it by the society, the security of web applications is a crucial challenge to be addressed.

1.1 Background

Data integrity, or simply integrity in the rest of this report, is a basic security requirement for web applications; data sent over the network must not be compromised. Integrity is particularly important for *sessions* in web applications. Intended to personalize user experience, sessions are typically implemented by passing session identifiers via cookies [1]. The cookies are sent with each request over the stateless HTTP protocol. A range of attacks such as *replay attacks* [2, p. 40,471], *cross-site request forgery* [3], and *session fixation* [4] target stealing and abusing the session credentials in order to hijack sessions and impersonate users towards the server.

Passive attackers are able to eavesdrop on the network and reuse any obtained sensitive information such as session tokens to impersonate the client for the server, and vice versa. *Active attackers* pose additional challenges for integrity as they are able to suppress and modify messages in transit and mount fully-fledged man in the middle attacks. Ubiquitous open wi-fi networks further exacerbate the problem. Under an open configuration of a wi-fi network, as frequently used in hotels, airports, and restaurants, the traffic between the user's device and base station is unprotected. Open wi-fi networks are susceptible to both passive and active attackers. This creates an ideal scenario for session hijacking attacks, as popularized by tools such as Firesheep [5], a Firefox extension to impersonate users logged on to social networks such as Facebook.

The only other standard alternative to HTTP for web applications is to use HTTPS, a web protocol that encrypts all communication using TLS/SSL [6]. TLS/SSL provide encryption of all data traffic at the transport layer, relying on asymmetric cryptography for key exchange, symmetric encryption for confidentiality, and *message authentication codes* (MAC) for message integrity. Achieving both confidentiality and integrity comes at a price of performance on both the sending and receiving ends.

From a security point of view HTTP and HTTPS offer an all-or-nothing choice to web applications; either no security guarantees with HTTP or both confidentiality and integrity with HTTPS at the price of performance. However, in many scenarios confidentiality is not necessary and even undesired, while integrity is essential to prevent attackers from compromising the data stream. Example scenarios include:

- *Intranet traffic.* In a corporate environment, it is important to maintain integrity of Intranet web traffic. At the same time, confidentiality is not desired because encrypted traffic is an obstacle for network logging and intrusion detection.

- *Public web site browsing.* Many web resorces allow users to manipulate public data, such as Wikipedia. Data confidentiality is not needed, while attempts of malicious modification of content and impersonating users need to be thwarted.

- *Open source projects.* Large volumes of data are transferred for publishing and downloading open source software projects. Since the data is public from the outset, confidentiality is not necessary. Integrity is however a must to prevent malicious modification of the code.

1.2 Goal

Motivated by the above scenarios, our goal is to create a protocol designed to authenticate both the client and the server towards each other. Both parties should be able to check that all packets originate from the other party in the conversation, and should also be able to verify the integrity of each message. A key goal is to protect against man in the middle attacks, thwarting any attempts of modifying the data stream by the adversary. We aim at specifying a general yet practical protocol for integrity in web applications. It is thus important to support the protocol with a proof-of-concept implementation, in order to evaluate programming overhead for the developer as well as indicative performance overhead.

1.3 Method

With the goals above in mind, we propose GlassTube, a lightweight approach to web application integrity. GlassTube guarantees integrity at application level, without resorting to the heavyweight HTTPS protocol. GlassTube provides a general method for integrity in web applications and smartphone apps. The protocol includes an initial setup part, including a key exchange phase, where the server and client collaborate to establish a session key, to be later used for signing and signature verification. The setup part requires an encrypted connection, which can be accomplished with the help of HTTPS. Once set up, the following messages in the session are sent over HTTP, with integrity assured by GlassTube signatures on per-message level.

GlassTube is easily deployed in the form of a library on the server side, and, as mentioned above, it offers flexible deployment options on the client side: from dynamic code distribution, which requires no modification of the browser, to browser plugin and smartphone app, which allows smooth key predistribution.

To evaluate GlassTube in practice, we have implemented a simple web chat service that uses GlassTube as library for integrity. The chat service requires minimal efforts from the developer to enable secure GlassTube sessions. Further, our experiments indicate a boost in performance compared to HTTPS, achieved even when no optimization efforts were made.

GlassTube opens up new possibilities for web application security. Application-level support implies flexibility in customizing the level of cryptographic protection suitable for different applications. It also opens up new avenues for application-specific confidentiality, where only selected information is encrypted, useful when the bulk of communicated data is public.

1.4 Delimitations

While we focus on the mutual authentication of the client and the server and integrity of the data stream, we note that the authentication of *users* is an orthogonal issue, which we leave to the application.

1.5 Disposition

The rest of the report is organized as follows. Chapter 3 presents the GlassTube protocol. Chapter 4 demonstrates the generality of the protocol by overviewing protocol instances. Chapter 5 focuses on the security of the protocol. Chapter 6 describes the case study of a web chat. Chapter 7 discusses related work, offers concluding remarks and gives an outlook into the future of the GlassTube protocol.

2

Theory

THIS CHAPTER DESCRIBES THE TECHNOLOGIES used in the later parts of the report. The chapter gives the reader a sufficient basic knowledge to understand later topics. If the reader is already comfortably familiar with a topic, the section describing it can confidently be skipped.

2.1 Cryptography

Cryptographic functions are vital in creating a secure communications protocol. In this section all cryptographic concepts, their use cases and their currently known strengths and weaknesses, are described. This section should help the reader evaluate the security strength of the concepts described later in the report.

2.1.1 Cryptographic Strength

The cryptographic strength of a cryptographic function is measured in bits, by how much effort it takes for an attacker to actually break it. In practice, this is equivalent to how many guesses an attacker has to do before he discovers the key. If there are only eight possible keys it takes eight guesses for an attacker - in the worst case - to find the key. The cryptographic strength of the cryptographic function is thus three, since three bits can store all possible keys ($2^3 = 8$).

Often, a cryptographic function is said to be weakened as per some new research. This means that the cryptographic function does not properly make use of it's key space. The attacker can somehow determine what keys are more probable - or rule out certain keys - and thus does not have to guess at the entire key space. In the example above, this could perhaps mean that the keys 111 and 000 are never used, and that we thus only have a cryptographic strength of $2^{\log(6)} \approx 2^{2.584}$.[2, p. 59-62]

As per Moore's law, computational power is constantly increasing. This means that the time to execute a successful attack against any cryptographic function is steadily decreasing, and NIST has recently (January 2011) increased their recommendation for the number of bits of security to be used in a cryptographic scheme, in order for it to be classified as secure, from 80 to 112 [7].

2.1.1.1 Entropy

Entropy is a collection of random data, and is used to create random numbers. Random numbers are usually provided by using a deterministic random bit generator (DRBG), often called a pseudo-random number generator (PRNG). A DRBG is initialized with a bit string, called a seed, from which it can generate an arbitrary number of numbers seemingly random. A DRBG is considered secure if an attacker is unable to guess any numbers that are generated, without knowing the seed, regardless of whether he or she can observe output from the DRBG. The security strength of a secure DRBG is the same as the number of bits of entropy provided with the seed. If there are 2^{80} possible seeds with equal probability, the security strength of the DRBG is 2^{80} since it will have only that number of random number sequences. [8]

2.1.2 Hash Algorithms

A cryptographic hash function, in this report simply called hash function or hash algorithm, is a deterministic procedure that given data of any amount produces a bit-string of a fixed size. A hash function should be considered a one-way function; it should be impossible to construct the original data using just the output from the hash function.

2.1.2.1 MD5

MD5 is one of the fastest and most commonly used hash algorithms, but since its publication by Ronald L. Rivest in 1992 [9] a number of weaknesses have been discovered. Collision attacks on MD5 which only needs an order of $2^{20.96}$ computations have been found [10], which is indeed far less than 2^{112}. Despite collision attacks, MD5 can be securely used for HMAC, as theoretical Preimage attacks against it only have lowered the security to 123.4 bits from the original 128 bits of security [11].

However, it is recommend not to use MD5 in new implementations, as weaknesses have been found. IETF stated in RFC6151 from March 2011 *"MD5 is no longer acceptable where collision resistance is required such as digital signatures. It is not urgent to stop using MD5 in other ways, such as HMAC-MD5; however, since MD5 must not be used for digital signatures, new protocol designs should not employ HMAC-MD5."* [12].

2.1.2.2 SHA-1

The SHA-1 hash algorithms was published in 1995 by NIST and has 160-bits of security [13]. It has since its publication become one of the most popular hash functions. But as with MD5, weaknesses have been found for SHA-1. IETF stated in RFC6194 that *"It must be noted that NIST has recommended that SHA-1 not be used for generating digital signatures after December 31, 2010, and has specified that it not be used for generating digital signatures by U.S. federal government agencies "for the protection of sensitive, but unclassified information" after December 31, 2013"* [14].

These weaknesses are in regard to collision-attacks, there have been some work done by John Kelsey and Bruce Schneier in regards to preimage-attacks but the data sizes required is not practical [15]. According to NIST [16] it is still safe to use SHA-1 for HMAC's as there are no indications that collision attacks against SHA-1 affects HMAC-SHA-1 in any way.[14]

2.1.3 Signature Schemes

To ensure that a sent message have arrived without being tampered with, and thus ensuring data integrity, a signature scheme can be utilized. A signature scheme usually consist of three parts: a secret key, a signing algorithm and a verification algorithm. The key is used to enforce security and is used in both the signing and the verification algorithm. It should be infeasible for an attacker to construct a valid signature without knowing the secret key. This section should help the reader evaluate the different schemes that are discussed in the report.

2.1.3.1 HMAC

HMAC is an abbreviation for Keyed-Hash Message Authentication Code, and is standardized by NIST [17]. An HMAC applied to a message enables the receiver to authenticate the source and verify the message's integrity. The sender supplies the HMAC with the message and a key shared only by he sender and the receiver, and sends the produced MAC along with the message. The receiver can apply the HMAC to the message with the shared key, which should compute the accompanying MAC. If the sent MAC does not match the computed one, the message's authenticity is void [17]. Equation 2.1 shows how an HMAC is computed.

$$
\begin{aligned}
H &= \text{the hash function} \\
K &= \textit{key} \\
B &= \text{block input size of } H \\
\textit{opad} &= 56_{16} \text{ repeated } B \text{ times} \\
\textit{ipad} &= 3c_{16} \text{ repeated } B \text{ times} \\
b &= 00_{16} \text{ repeated } B - |H(K)| \text{ times} \\
a &= 00_{16} \text{ repeated } B - |K| \text{ times} \\
K_0 &= \begin{cases} H(K) \quad || \quad a & |K| > B \\ K \quad || \quad b & |K| \leq B \end{cases} \\
\mathrm{HMAC}(K, m) &= \mathrm{H}((K_0 \oplus \textit{opad}) \;||\; \mathrm{H}((K_0 \oplus \textit{ipad}) \;||\; m))
\end{aligned}
$$

Note that even if the hash function is applied twice, the outer will always be fed an input of fixed size $(B + |H(x)|)$. Thus, the running time of a HMAC is quickly dominated by the running time of the innermost hash function since the message size is not fixed.

NIST [16] supplies HMAC-SHA1 as one of it's recommended HMAC algorithms, and several currently emerging technologies such as OAuth2 [18] have chosen to support HMAC-SHA1. As mentioned previously in this chapter HMAC-MD5 is no longer to be supported in newer protocols, making HMAC-SHA1 the fastest secure HMAC algorithm as of the time of writing.

2.1.4 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange is an algorithm with which two parties may agree on a common secret while keeping it secret from a potential eavesdropper. The protocol makes use of the fact that the discrete logarithm problem is known to be hard, while exponentiation is computationally easy. [19]

The protocol uses two domain parameters, namely a generator g and a prime number p. These does not have to be keep a secret for the key exchange to be safe. The two parties within a Diffie-Hellman key exchange each generates a random number to be their private key x and y, respectively. They then create their public key as $X = g^x \bmod p$ and $Y = g^y \bmod p$, which they send to the other party. Both parties can now compute their shared key $Z = g^{(xy)} \bmod p = X^y \bmod p = Y^x \bmod p$. An eavesdropper will know both X and Y, but neither of x and y, and can thus never find Z. [20]

2.2 Web Technologies

This report will have a major focus on web technologies, and as such the reader is expected to have a general idea of how a web application works. A few more advanced terms are explained in this section.

2.2.1 AJAX

AJAX stands for Asynchronous JavaScript and XML. In a conventional HTTP request to a web server an HTML document is delivered to the client, the client reads the HTML document and builds a DOM object from which it renders graphics. With an AJAX call the HTML information is retrieved directly in the JavaScript instead in your browser. This means that information can be sent and received without having to reload the page, that is what the Asynchronous part of the name stands for. [21]

2.2.2 Data URI Scheme

Also called the Data URL Scheme, allows for programmers to embed the binary data of an element instead of referencing to its location. This scheme can be used to lessen the number of HTTP requests needed to populate a HTML document, by replacing i.e.

```
<IMG SRC="http://example.com/larry.gif" ALT="Larry">
```

With the following data URL

```
<IMG SRC="data:image/gif;base64,R0lGODdhMAAwAPAAAAAAAP
     ///ywAAAAAMAAwAAAC8IyPqcvt3wCcDkiLc7C0qwyGHhSWpjQ
     u5yqmCYsapyuvUUlvONmOZtfzgFzByTB10QgxOR0TqBQejhRN
     zOfkVJ+5YiUqrXF5Y5lKh/DeuNcP5yLWGsEbtLiOSpa/TPg7J
     pJHxyendzWTBfX0cxOnKPjgBzi4diinWGdkF8kjdfnycQZXZe
     YGejmJlZeGl9i2icVqaNVailT6F5iJ90m6mvuTS4OK05M0vDk
     0Q4XUtwvKOzrcd3iq9uisF81M1OIcR7lEewwcLp7tuNNkM3uN
     na3F2JQFo97Vriy/Xl4/f1cf5VWzXyym7PHhhx4dbgYKAAA7"
     ALT="Larry">
```

However, for larger images this becomes impractical as a way for the programmer to embed images into his or her markup. [22]

2.2.3 Client Side Storage

Client side storage was introduces in HTML 5 through the *Storage* interface [23] to enable data to be stored locally on the client. Each top-level browser context, e.g. a tab, has a pair of *Storage* objects for each origin. Each pair consists of a *sessionStorage* object which will be cleared at the end of each session, and a *localStorage* object which is only cleared for security purposes or when request by the user. Traditional HTTP state management [1] make use of cookies, which are sent over the network and may introduce security risks.

2.2.4 Cross-origin Resource Sharing

It is common for web applications to load content, i.e. a JavaScript library, from other domains through a content distribution network [24]. This behavior can be used by malicious web site operators to disturb or abuse traffic on a benign site, which has lead to the fact that browsers need to isolate content retrieved from different sites through the Origin HTTP header [25]. In order to still allow specific content to be loaded across sites, developers can specify allowed sites through the header field Access-Control-Allow-Origin [26].

2.3 Attacks

The report will discuss different attacks and present ways to counter them. This section should give the reader good insight into what different attacks consist of, and how they may be applied by an attacker to a web application.

2.3.1 Man-In-The-Middle Attack

In a Man-In-The-Middle (MITM) attack an attacker has control over a part of the network used by a client and server TCP connection, and therefore have the capability to split the connection in two parts: client - attacker and attacker - server. See figure 2.1.

If the data sent between the sever and client is completely unprotected then it is easy for the attacker to change the content before passing it along and the client/server is none the wiser, this is for example possible to do when using the HTTP protocol. However even if the session between the client and server are protected there are ways to get around it. The attacker can initiate two SSL connections, one with the client and one with the

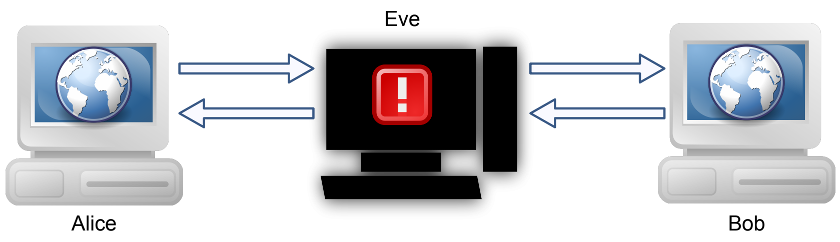

Figure 2.1: All messages between Alice and Bob goes through Eve, meaning she can manipulate these at will.

server. But unless the attacker have a certificate that is signed by a trusted Certificate Authority, the client will be likely to receive a warning from the browser but of course the user can choose to ignore the warning and proceed to send data to the attacker [27].

2.3.2 Replay Attack

In its essence a replay attack consists of passively capturing data to later replay it either to the receiver or the sender. A successful replay attack can at the minimum disrupt services with messages that appear to be genuine but are not, but more severe damage can also be done. An example of this is a money transfer, if an attacker could replay this transfer the victim might lose a lot of money [2, p. 40,471]. See figure 2.2 for an example of an simple replay attack.

2.3.3 Denial of Service Attack

A denial of service aims to cripple or completely shutdown a service, this can be anything from a single machine to an entire network. It can be performed in many different ways: disabling the service in question physically, overloading it with messages so that "real" traffic can't get through or exploiting a weakness in the software to cause errors that prevents other actions [2, p. 40-41]. See figure 2.3.

2.3.4 Brute Force

When doing a brute force attack it is not interesting to know how the specific algorithm used works, all that is interesting is the bit

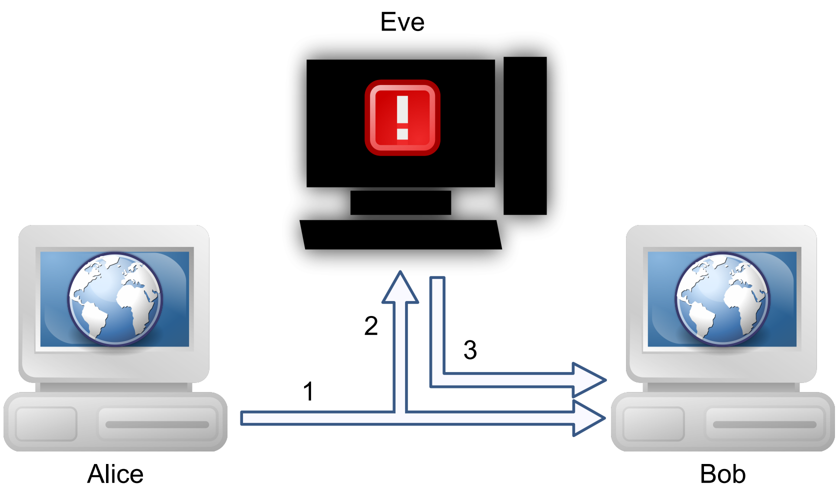

Figure 2.2: 1.Alice Sends a message to Bob 2. Eve captures the message 3. Eve later replay message to Bob

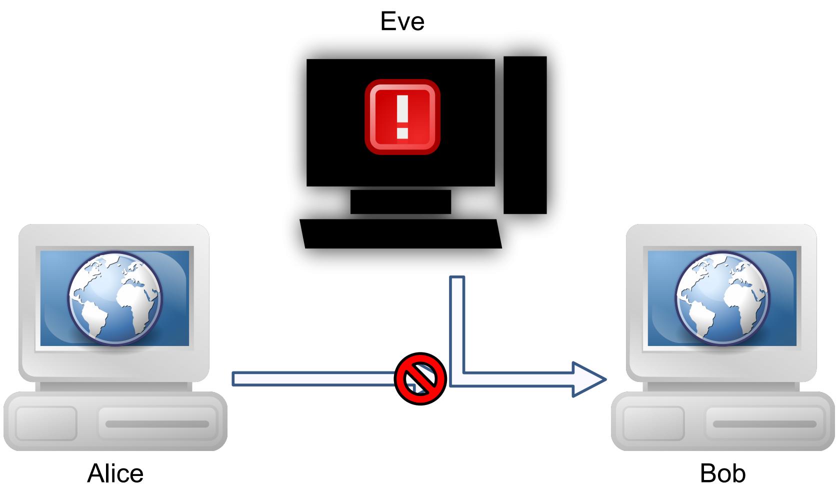

Figure 2.3: Alice Sends a message to Bob but Eve disrupts Bob so that the message doesn't reach him.

length. Brute force means that the attacker tries every possible key until a match is found such that encrypted text is intelligible or that a signature can be created for new messages. Usually the key is in focus, because once the key have been calculated all messages encrypted with this key can be decrypted and messages constructed by the attacker can be encrypted, the same goes for signatures. On average about half of all keys have to be tried before finding the right one [2, p. 62].

2.3.5 Attacking an HMAC

The security of an HMAC function is dependent on the key and the underlying hash function. An attack on an HMAC can focus on either finding the key using brute force or finding a collision in the hash function.

The brute force attack on an n-bit key is an attack against the compression function but if a message of size b-bits, the b is the number of bits in a block, is used it is then the equivalent of attacking the hash function. This attack require work in the size of 2^n[2, p. 403].

To find a collision the so called collision resistant attack is used, which builds on the birthday paradox, and means that the attacker is looking for two messages $M1$ and $M2$ such that they produce the same hash value $H(M1) = H(M2)$. In essence this means that an attack like this will take $2^{m/2}$ attempts for a message with a m-bit hash value [2, p. 362].

2.3.6 Cross-site Scripting

Cross-site scripting is an attack/vulnerability where the attacker injects malicious scripts or code or both to an otherwise trusted website. And the idea is then that a user on the website will do something, like use a search function, click on a link, fill in a form or something else and in the reply from the trusted server will then contain the malicious code. The clients browser have no way of knowing if the code that it receives is malicious or not and will execute it in either way, and this also mean that the malicious code have access to cookies, session-storage or any other information that is accessible to JavaScript.

Cross-site scripting is a widespread problem and unless user input are checked, sanitized or validated on a website it is a target for this kind of attack [28].

2.3.7 Cross-site Request Forgery

An cross site request forgery(CSRF) attack is when an attacker forces an end user to execute unwanted actions on a web application where the user currently is authenticated. A user may be forced to execute actions of the attacker's choosing, which could be anything from logging out to sending an email to making a purchase. A successful CSRF attack can compromise an end user's data and functionality in the case of a normal user, but if the target is an administrator the entire web application can be compromised. [3]

2.4 Google Web Toolkit

Google Web Toolkit (GWT) is a set of open source tools developed and maintained by Google, made for developing web application front-end using Java code. Except for a few native JavaScript libraries everything is Java source. An entire application can more or less be completely coded in Java and then during compile time GWT will convert the application to JavaScript, ready to deploy. The generated JavaScript code comes tailored to work for most of the popular browsers today, meaning you don't have to do this manually.[29]

3

The GlassTube Protocol

GlassTube is designed to provide integrity over insecure connection, preventing manipulation of the data stream. This chapter specifies the GlassTube protocol. The protocol consists of two steps. GlassTube Setup (GTS) is the first part of the protocol. It maintains distribution of code and the key exchange. The second part of the protocol is the GlassTube Integrity Protocol (GTIP), which ensures integrity between the web server and the client.

3.1 GlassTube Setup

GTIP requires code to be distributed to the client and that a session key is shared between the client and the application server, henceforth called the data site. GlassTube can be initialized in any fashion that securely meets these requirements. This section describes some options to distribute code and how keys are exchanged, independently from each other.

3.1.1 Client Code Distribution

Since web applications are not present on the client per default, the client-side code needs to be distributed. If an attack can be successful at this stage, following packets must be considered compromised as e.g. script injection at this stage can change how transfers are made in the future. It is therefore vital that code distribution is done securely. Client code can either be statically or dynamically distributed. Statically-distributed code is previously present at the client (e.g. a browser plugin). Dynamically-distributed code is sent when the web application is accessed by the web browser. Examples of both are presented in the following paragraphs.

Static distribution of client code refers to the installation and use of browser plugins or applications for smartphones. For an end user to communicate with a web server running GlassTube he or she would have to acquire and install additional software prior to making the first request.

Dynamic distribution is the most common type for web applications, as it is used by almost every page on the web. Typically it consists of JavaScript, Flash, Silverlight, or Java applets embedded within the web page.

Dynamic code distribution for GlassTube must be done over a link with at least the security equivalent to that of a host servicing HTTPS, called the secure site. The secure site will only communicate with the client during the setup of a GlassTube session.

If code is dynamically distributed, the goal is that the client is reinitialized as rarely as possible, to boost performance by limiting the reliance on HTTPS. Instead, all content can be fetched with AJAX from the data site, and only the bare bones of the application should be sent from the secure site.

3.1.2 Key Exchange

GTIP relies on a session key to be shared between the client and the data site, in order for both of them to be able to sign messages and verify their integrity. The goal of key exchange is to establish such a session key.

The key exchange scheme needs to be secure; there are several ways to accomplish this. We propose two key exchange schemes, one being Authenticated Diffie-Hellman [30], while the other being GlassTube's own key exchange scheme, the GlassTube Key Exchange (GTKE), which have better performance then Authenticated Diffie-Hellman. Both key exchange schemes require that the client possess a public key belonging to the data site. The public key can be acquired either by distributing it with the code or by fetching it from a secure site after code distribution.

Random numbers are used in both of the key exchange schemes. It is possible that the client does not have enough entropy to provide secure random number generation. In that case, the client can be supplied with random numbers as well as code, in dynamic code distribution.

GTKE is accomplished by letting each party generate a separate part of the session key K, both at random. The client constructs a pair (K_1, C). Where K_1 is the first part of the session key, and $C = \mathrm{E}(K_1)$ is encrypted using the data site's asymmetric public key. The client sends the encrypted key C to the data site, after which the data site can compute $K_1 = D(C)$. The data site then generates the second part of the session key, K_2, and computes the final session key $K = K_1 \oplus K_2$. The data site signs K_2 with K, creating the signature $SIG = \mathrm{SIGN}(K, K_2)$ and sends $K_2 || SIG$ to the client. The client uses the received second part of the key K_2 to compute $K = K_1 \oplus K_2$, and must finally verify the data site's authenticity by comparing if SIG is equal to $\mathrm{SIGN}(K, K_2)$.

3.2 GlassTube Integrity Protocol

Data transfers done with GTIP guarantees that data integrity is kept; the authenticity of the sender of each message can be verified, prevents messages from being replayed. Details on how this is accomplished are described in the rest of this section.

3.2.1 Message Identifier

In order to prevent replay attacks it must be possible to distinguish each request from other requests containing the same data. The receiver must be able to determine whether a message identifier is invalid or if it is to be accepted. The protocol addresses this by appending a unique identifier to each submission. This identifier can be anything from a sequence number or a timestamp to a nonce. Sequence numbers provides good security and can be efficiently implemented and are therefore the recommend message identifier for GlassTube.

3.2.2 Signature

The authenticated data includes all information that defines the request or response. All data that if changed would modify the data stream must be signed, together with the message identifier. The complete URL and all parameters have to be signed for requests, for responses the response code have to be signed. And for both of them the message identifier have to be signed.

Reordering the request parameters will produce a different signature. The union of all request parameters are sorted alphabetically, in order to deterministically determine the order on an arbitrary platform. The request parameters consist of key-value pairs which are concatenated as follows:

```
$PARAMS := [$KEY1=$VAL1[&$KEY2=$VAL2[...]]]
```

The parameters must be encoded as they will be sent in the actual HTTP request by the browser. The string to be signed is then constructed as:

```
$MESSAGE   := if(client)
                  $URL?$PARAMS:$MSG_IDENT
              else
                  $RESP_CODE:$RESP_DATA:$MSG_IDENT
$SIGNATURE := SIGN($SESSION_KEY, $MESSAGE)
```

A GlassTube signature is computed on the GlassTube message using the session key. HMAC-SHA1 is the recommended method for signing GlassTube packets as it is efficient and secure. The signature and the message identifier are included in each request and each response.

3.2.3 Verifying a Request

The steps needed to verify a signed message are depicted in Figure 3.1. The figure gives an overview of the process, detailed in the following paragraph.

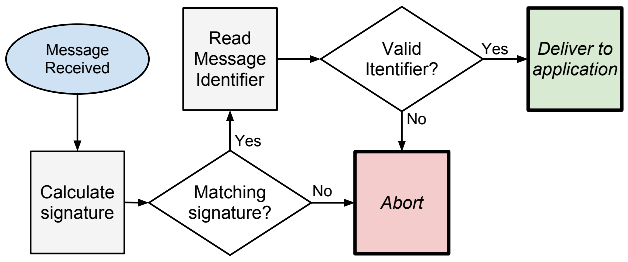

Figure 3.1: A flow diagram that outlines the verification process

Upon receiving a message, the recipient must first calculate the signature, as described in Section 3.2.2. In the case that the signature does not match the received signature, the message is discarded. Otherwise, the message identifier must be verified. If the message identifier is valid, the message is accepted and sent to the application, and if it is not, it is discarded. Whenever a client discards a message, it must reset the session. If a server discards a message, it must send a signed error message to the client, indicating what went wrong.

4

Protocol Instances

The GlassTube protocol offers a wide range of setup and deployment choices. This chapter outlines three different instances that all use the GlassTube protocol, in order to help the reader see the practical applications of the concepts presented in the Chapter 3.

4.1 Web Application

This section describes a setup for a web application which uses dynamic code distribution and server-side random number generation, illustrated in Figure 4.1.

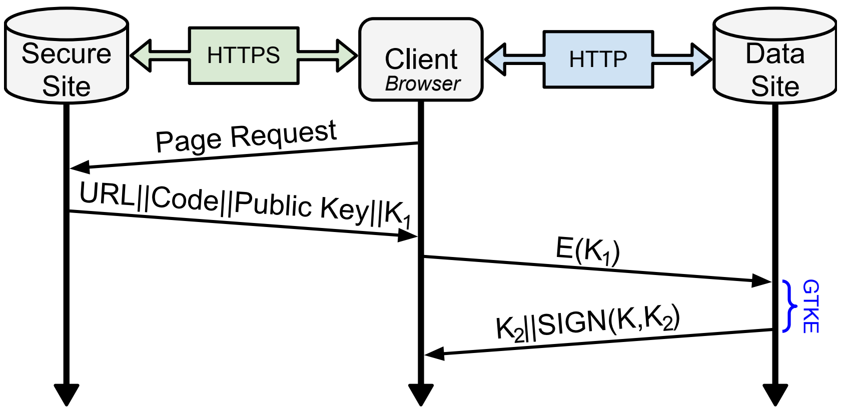

Figure 4.1: A GlassTube setup to be used with a web browser with no additional software installed.

To start the session the client sends a `Page Request` to the secure site. The response from the secure site consists of four parts: a `URL` to the data site, `Code` that consist of GlassTube functionality and the web page's static elements and layout, the data site's `Public Key` and the client's part of the session key K_1.

When the client has interpreted the received code it will initiate GTKE towards the data site using AJAX, after which it will have established a GlassTube session. If the data site receives a regular, non-GlassTube page request, it would typically redirect the client to the secure site to force it to set up a GlassTube session.

The developer sets up two sites, the secure site running HTTPS and the data site servicing HTTP. Note that the secure site and data site can be hosted by the same machine, but may as well be distributed among different machines for load balancing. To add GlassTube functionality to the data site the developer would typically import a GlassTube library and mark individual pages to use GlassTube. As the secure site and the data site make use of different protocols, they are separate origins and separated by the same origin policy [25]. The data site must therefore explicitly allow the secure site to make Cross-Origin requests to it, making use of Cross-Origin Resource Sharing (CORS) [26]. A typical client code needs few modifications in order to work with GlassTube, for an example see Section 6.1.2.

In a web application, it is common that data transfers are initiated when HTML is loaded on a page, making the browser fetch additional content. This happens when e.g. `<img src="image.jpg" />` is rendered. The browser will fetch the image *image.jpg* from the server, without resorting to AJAX and JavaScript. The same is true for `<script>` and `<iframe>` tags, which means that if these tags are used by in the application, they may break GlassTube. However, a programmer may make use the data URL scheme [22], which embeds the binary data of an element instead of referencing to its location, to achieve the same functionality that can be achieved with the traditional URL scheme.

GlassTube is completely transparent to the end user. Users access a GlassTube web page as any other web page, e.g. using a bookmark or following a link. A benefit of using this setup is that the application provider does not have to create, maintain and distribute a separate software for the client. The setup requires a secure site servicing HTTPS, and will thus have a slightly bigger impact on the server than a client with completely predistributed code and public key.

4.2 Generic Browser Plugin

Another way to set up a GlassTube session is by utilizing a generic browser plugin; generic in the sense that it is not bound to a certain web application or domain. This setup is outlined in Figure 4.2. Upon connecting to a web site, the plugin will announce that it can handle GlassTube sessions, through e.g. a custom header field. The secure site will see if a new client is running a compatible plugin, and if it is respond with the `URL` to the data site and its `Public Key`. The plugin will initiate GTKE with the data site, and once a GlassTube session has been established it will load the initial user interface from the data site.

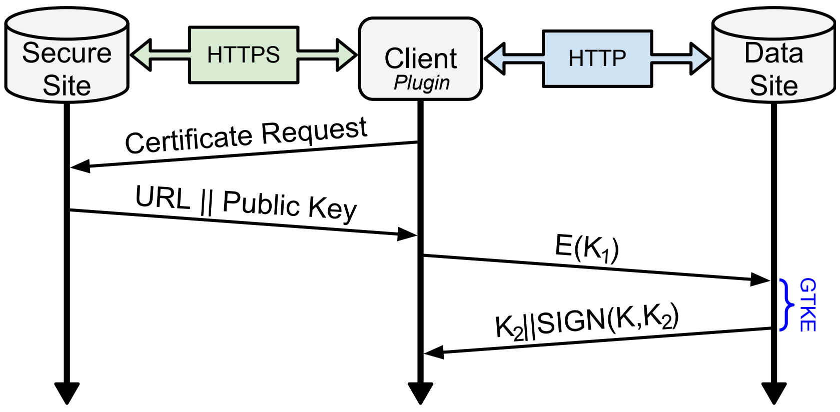

Figure 4.2: GlassTube protocol instance where a generic plugin which fetches the data site's public key for each new session.

By creating a generic plugin the need for individual application providers to maintain and distribute it is avoided, and the end user only needs to do one additional action for any number of GlassTube sites. The public key for every accessed site must be fetched from a secure site. Benefits of using a plugin instead of dynamic code is potentially better client performance and the possibility to sign HTML initiated requests.

The work needed from the developer to implement GlassTube in this setup is very similar to the work required in Section 4.1. The implementation difference between the two methods is that with this approach, no effort related to the client code, and CORS does not have to be used.

4.3 Smartphone App

Many web applications might find it desirable to release an app that lets the users browse and edit the information, this setup is illustrated in Figure 4.3. It differs from the other two in the sense that no secure site is used, instead the data site's public key comes shipped with the app. This means that the overhead of first connecting to a secure site is gone, which gives a performance boost. As most apps it will be dedicated to one web site, thus an end user has to install one app per site.

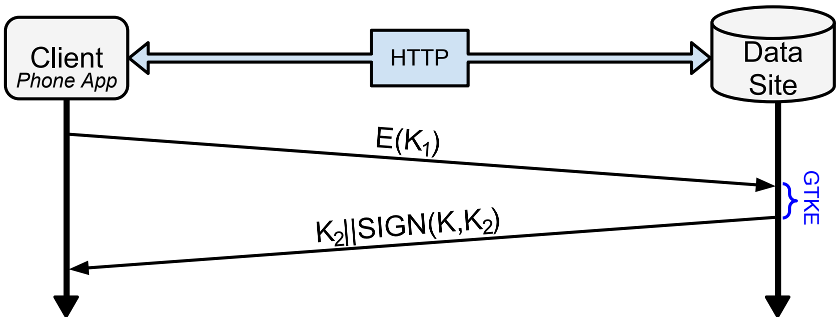

Figure 4.3: GlassTube as used by a smartphone app tailored to work only with one web application.

For this setup the developer creates and releases an app. No secure site needs to be deployed, as the public key for the data site is shipped together with the app. Any functionality needed in the app for key exchange and GTIP can be accessed from a library, thus very little extra work needs to be done. The data site is set up as in Section 4.1.

5

Security Considerations

This chapter covers the security within each stage of the GlassTube Setup, and assesses the security of the GlassTube Integrity Protocol. Different attacks and the respective countermeasures are discussed.

5.1 Client Code Distribution

To ensure that requests are signed and verified correctly, it is important that the code running on the client is not modified by an attacker. This means that the code has to be distributed in a secure way. Two options were presented in Section 3.1.1, one where the code is fetched from a dedicated code server, dynamically, and one where the code is available via a plugin or application on the client, statically. For both mechanisms, trusted third parties are used to securely distribute initial code.

When code is sent at the beginning of each session, it needs to be done over a secure channel. The channel must be end-to-end between the server and client, as man in the middle attacks can be attempted at any point during the transmission. HTTP over SSL offers the required security and is a typical choice as it is broadly supported.

Static code distribution is an effective way to decrease the risk of a man in the middle attack against the client code, since the code is not sent over the network at the start of each session. It is still vital that the code is securely distributed to the client. The common distribution channels for smart-phone apps and browser plugins provide secure downloads and verifies the publisher. This means that most use cases of statically distributed code can be assumed to be secure.

GlassTube relies on HTTPS in many of the concepts presented, for which there is a known man in the middle attack where the attacker relies on users to disregard when the browser warns about incorrect certificates [31]. This can lead to insecure dynamically distributed code or forged certificates, which will break GlassTube.

5.2 Key Exchange

Recall that there are two key exchange mechanisms to support the GlassTube protocol, authenticated Diffie-Hellman and GTKE. Authenticated Diffie-Hellman is a canonical way of exchanging keys in a secure manner, as it a part of TLS versions 1.0 [32], 1.1 [33] and 1.2 [30].

GTKE is a novel contribution of this report. It makes use of asymmetric encryption to share a session key between the two parties, and is detailed in Section 3.1.2. The secrecy of K_1 is guaranteed, as long as a sufficiently secure encryption is used. K_2 is added by the server, and is sent in clear text to the client. K_2 is the only part of the key that is publicly known, but a potential attacker must also know K_1 in order to find K. An attacker can try to recover the K_1 by attacking the HMAC algorithm, but this is redundant since the attacker will know K without finding K_1 if he or she attacks the HMAC algorithm.

K_2 is present in GTKE in order to prevent an attacker from replaying a key exchange followed by replaying selected messages from the session which followed. The two parties share K_1, that could be used as a session key, before K_2 is generated. However, if the server does not generate some addition to the key, an eavesdropper can replay an entire session. The attacker will not be able to sign any new messages, but can send messages that he or she has listened to (possibly with restricted order because of sequence numbering), since the attacker knows that the server will use the same session key as in the original session. With the addition of K_2, each new client will have a new session key, and sessions can not be replayed.

5.3 GlassTube Integrity Protocol

The signature of each message is the most vital part of GlassTube, as it is the entity which enables the integrity of a message to be verified. The GlassTube signature must be created using a secure signature algorithm, and relies on the fact that session keys are kept secret.

By including the request parameters and the URL in the signature, GlassTube asserts that the client will know if intended data is delivered to the intended service on the intended server. If an attacker modifies the URL, it may be delivered to a different web application, in which case the client will reject the response since the signature will be invalid. If the packet is delivered to the same application but using a different URL, the application will detect that the signature is invalid, and will discard the packet. If the service is not running GlassTube, for example if it is a public page on the same service, or simply another service not running GlassTube, the packet might not be discarded by the server. However, neither will it be signed correctly, which will make the client discard the response. Any modifications to the request parameters will trivially invalidate the signature.

With each response, the response code and response data are signed. An attacker will therefore fail to forge new data, as it will invalidate the signature. Modifying the response code can have more complicated consequences, as it is interpreted by the web browser. There are several response codes that omit an entity body, such as 204 No Content [34, p. 59]. If an adversary modifies the packet to use an empty response, there will be no signature, and the client will therefore discard the packet and reinitialize the GlassTube session. The response code 304 Not Modified [34, p. 62] may be inserted by the attacker to make the browser use a cache, but as GlassTube messages include message identifiers they cannot be cached. This leaves browsers in a state inconsistent with the specification. The browser may try to resend the request, and if so, it will be detected as a replay attack by the server, and the session will be terminated. Some response codes, as 301 Moved Permanently [34, p. 61], are intended to redirect the client, such responses will cause the session to be terminated and are further described in Section 5.7.3. Thus, editing the response code will terminate the session, possibly after an extra round-trip.

5.4 Entropy

When creating random numbers to be used in a cryptographic function, it is important that the random generator used is supplied with enough entropy to be secure [8]. Not all programing languages have generators suited for this. JavaScript [35] currently has no support for secure random number generation. If no third party library is used only `Math.random()` is available in JavaScript, which is not guaranteed to give cryptographic security. However, adding cryptographically strong random number generation to JavaScript API is only a matter of time [36].

In any case, GlassTube does not depend on the ability of the client to generate random numbers. The client must not be used to generate random numbers if it can not guarantee cryptographically secure random numbers. In this case, the secure site generates the random numbers and sends them embedded within the code.

5.5 User Authentication

The GlassTube protocol does not provide user authentication, but leaves it to the application to authenticate each user. By design, GlassTube does not offer confidentiality, and it is therefore important that the application does not send authentication data in clear text. If the user gives the attacker enough information to authenticate as the user, there is often no need for integrity.

5.6 Replay Attacks

GTIP makes use of message identifiers, as described in Section 3.2.1, to prevent replay attacks. If the message identifier is unique for every message, the recipient will only accept a specific message identifier once. Thus if message identifiers are unique, no messages can be replayed. It is possible to use a timestamp as a message identifier, which is not unique as the recipient only verifies that the message is fresh, if the service can accept that replays are possible within a short timeframe after the message has been sent. Different message identifiers thus provide different levels of prevention against replay attacks. By using the recommended message identifier, sequence numbers, all replay attacks are prevented.

5.7 Man In The Middle

A correctly set up GlassTube protects against most aspects of a man in the middle attack, except for when the attacker delays or completely removes packets from the stream, for which it is unfeasible to create a solution.

An adversary cannot masquerade as the data site because each key exchange scheme uses public keys. The public keys are either distributed with the code, or fetched from the secure site, as detailed in Section 3.1.1. Both of these methods are considered secure, see Section 5.1.

An active attacker may during a full man in the middle attack modify the entire packet, which includes HTTP headers, TCP and IP. The rest of this section discusses in depth what data can be modified in order to try to nullify the integrity of the data stream, and how GlassTube can prevent some of these attack.

5.7.1 Network Layer

GTIP does not sign any information from the network layer. This means that if an application is dependent on information from the network layer, GlassTube can not guarantee the integrity of the data stream for that application. A common usage of information from the network layer is geolocation, where the application uses the IP address to find the geographic location of the user.

5.7.2 Transport Layer

By changing the TCP sequence numbers of packets in a GlassTube stream, an attacker can make sure that they are delivered to the application layer out of order [37]. This can be countered if sequence numbers are used as message identifiers, as it assures that packets arrive in order. Since message identifiers are signed, the attacker can not modify these to make packets arrive out of order, as detailed in section 5.6. There are numerous ways the attacker can disrupt the data stream by modifying data in the TCP layer, such as truncating the session by sending a FIN [38] packet.

5.7.3 Application Layer

An active attacker can add, remove and moify HTTP headers, but since GlassTube forbids the application to depend on any supplied headers the result of a request should always be the same regardless of any HTTP headers. Any inherent behavior of the web server is considered to be the responsibility of the programmer; the configuration of the web server is part of the application. However, since browsers and intermediate hosts, such as firewalls and proxies, may react to different header fields, consideration and care is still needed.

It is possible for an attacker to modify HTTP headers in such a way that the message is interpreted differently by the browser. However, if an attacker forces the browser to modify a message, the signature will be void. An attacker can use the Location header [34, p. 135] to force the browser to execute a new request towards another URL. Since the browser sends the *same* request to a different URL, this message will be treated in the same manner as a message with a modified URL, described in Section 5.3, and will lead to that the either the request is discarded by the server or by the client, depending on where the new URL points.

When a benign intermediate host, such as firewalls and proxies, modifies header fields, a GlassTube session may be terminated or unable to commence. Thus, there may be false negatives over certain links, making GlassTube malfunction. These may completely prevent a client from reaching the service, but does not lead to a breach of integrity.

5.8 Denial of Service

Both the secure site and data site can be attacked independently. Denying the user one service will have the same effect as denying both, since neither will function without the other.

GlassTube does not have a large performance impact on the secure site, as this site will always perform only a fixed number of operations for every client; the extra workload caused by every separate GlassTube client will not increase with the number of clients.

Each session between a client and the data site requires that the data site stores information. The time to access this information will increase with each client, and thus each new session does not just add its own workload but does also affect the workload for all other sessions. This in turn means that the data site is more vulnerable to a resource exhaustion attack.

6

Case Study

THE FOLLOWING CHAPTER presents a working prototype of GlassTube, and investigates how it performs relatively to HTTP and HTTPS. The web application used in the study is a simple chat that allows the users to login, post messages, read messages, and logout.

6.1 GlassTube Implementation

This section covers a server implementation of GlassTube using Java and two separate clients implemented in Java and JavaScript. Java is chosen as the backbone for both the server and clients, using Google Web Toolkit [29] (GWT) to generate JavaScript as needed. Standard Java libraries are used whenever possible as they provide reasonable performance and are easy to use. The chosen implementation strategies are dynamic code distribution and server-side random number generation for the JavaScript client, with static code distribution and client-side random number generation for the Java client. A secure site and a data site are set up, but no actions have to be taken to prepare a web browser to use the JavaScript client.

The Java client is developed in order to benchmark the server, see Section 6.2. The JavaScript client is developed to assert that the user experience is not noticeably affected by GlassTube.

6.1.1 Server

Both servers are implemented with Java servlets using standard Java libraries for cryptographic functions as well as web application functionality. The implementation of the server-side part of GlassTube consists of 203 lines of Java, 66 at the secure site and 137 at the data site.

The secure site consist of two servlets. The first servlet only delivers the data site's public key. The second delivers static elements, JavaScript, K_1 as well as the data site's public key. This separation of functionality is because the Java client only needs the public key, while a browser needs to be served with a normal web page. K_1 is generated using Java's SecureRandom and the data site's public key is hardcoded to the servlet. As it is only part of the setup phase no further data is handled by this server.

For the data site a GlassTubeServlet is created. It extends the HTTPServlet but adds GlassTube specific functions for key exchange and signing and verification of messages. The GlassTube-Servlet demands that the first message from a client contains C. K_2 is generated using SecureRandom, and $K = D(C) \oplus K_2$ is calculated. The response $K_2||SIGN(K,K_2)$ is sent to the client, concluding the GTKE. All servlets extending GlassTubeServlet on the data site are now ready to use GTIP. All information required for GTIP this is stored in the server's session storage.

6.1.2 JavaScript Client

The JavaScript client uses GWT to convert all cryptographic functions from Java to JavaScript. The jQuery JavaScript library is used to provide smooth access to AJAX and different user interface functionality. Functions for exchanging keys, signing and verifying are thus coded in Java, while data transfers during GTPI are managed in native JavaScript using jQuery. The Java code is 75 lines long, and the JavaScript functionality needed is 14 lines long. This excludes the cryptographic functions (HMAC and SHA-1) that were needed to be imported because javax.security is unavailable to GWT.

Upon initialization, the JavaScript generated by GWT initiates GTKE using the public key embedded in the code, together with the first part of the session key K_1, which is computed by the secure site. When the key exchange is complete, the web application is ready to be used by the end user.

GlassTube's presence is only noted when signing and verifying messages. To make an AJAX request with jQuery, the programmer writes for example `$.ajax(url, { data: parameters })`. Using our implementation the programmer has to add the line `$.extend(parameters, gt.sign(url, parameters))` prior to making the AJAX call. The extend function simply appends the information needed in GTIP, to be sent along with the regular AJAX call. To the programmer, GlassTube does not incur a large amount of extra work.

6.1.3 Java Client

The Java Client is able to make use of Java's standard API to provide all needed cryptographic functions. The Java client has access to Java's SecureRandom, which is cryptographically secure, and K_1 is therefore generated locally by the Java client. It fetches the data site's public key from the secure site, after which it commences GTKE towards the data site. When GTKE is completed, GTIP is ready to be used.

6.2 Benchmark

This section details the results of a series of tests conducted to verify how GlassTube performs in relation to SSL.

The first benchmark measures how well the server performs, and compares the average number of successful requests per second for the different techniques. The second benchmark compares the response time as experienced by a web client without a plugin to boost the performance of GlassTube.

6.2.1 Server benchmark

The benchmark is done against a simple chat application by the name of SimpleChat. The implementation of SimpleChat is very simple, using static fields to maintain the state of the web application. The servlets that make use of SimpleChat simply passes along parameters from the request to SimpleChat. The only implementation differences between the instance for HTTP and HTTPS, and the instance running GlassTube, is the servlets base class. The servlets in the GlassTube instance are a subclasses of GlassTube-Servlet instead of HttpServlet. Each access to any of SimpleChat's functions is counted as a successful request by a client.

To test the server performance using the three different techniques, a benchmarking tool has been written in Java as to not be constrained by the performance of JavaScript. The tool supports three different types of clients, HTTP, HTTPS and GlassTube. All clients make use of Apache's `DefaultHttpClient` [39], and the GlassTube client makes use of the GlassTube Java client, described in Section 6.1.3.

The benchmarking tool is able to spawn a user-specified number of threads, each opening an individual and independent (on application level) connection to the server. Each thread logs in to SimpleChat and starts sending chat posts. The Java client is capable of both sending posts as quickly as possible and with a delay between each. The later is more similar to that of an actual user scenario, since the first sends many hundred requests per second. The size of the data sent is also configurable.

Figure 6.1 plots the result of a benchmark conducted towards GlassTube, HTTPS and HTTP. The graph plots successful messages per second on the Y-axis, and the tests are carried out with an increasing number of clients for each test. The clients are configured to log in with a delay of between 0 and 100 milliseconds, and once logged in they will proceed to send 2000 messages of 4096 bytes at an interval between 10 and 300 milliseconds. The rate at which the clients send messages is labeled as Max in Figure 6.1, as it is the maximal throughput of a server with unlimited processing power and a network without latency.

The benchmarks were carried out towards a server running Tomcat 6, configured with 200 threads for both HTTP and HTTPS connectors, with the java runtime allowed 1.5G of ram. The server machine is an HP Compaq dc7600 with a 3.00 GHz 64 bit processor with two cores and 2GB of RAM. The client machine was an HP Compaq 6730b with a 2.4 GHz 64 bit processor with two cores and 4GB of RAM.

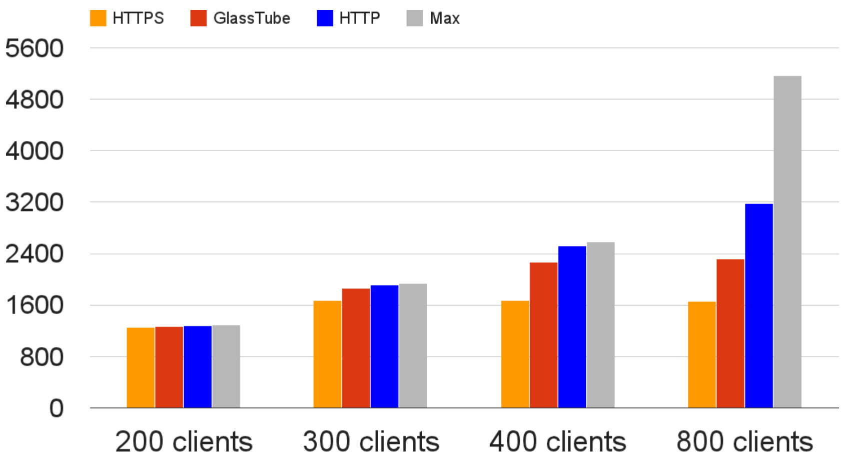

Figure 6.1: Shows different number of clients on the X-axis, and successful messages per second on the Y-axis. Very short delays are in place, and each post is 4096 Bytes large.

Figure 6.1 clearly indicates what the throughput limit for each protocol is. HTTP will follow closely to the maximal limit until just about 3200 packets per second is reached, while HTTPS can handle almost 1680 packets per second and GlassTube can process around 2300 packets per second. The number of requests that can be served by GlassTube in this setting is 39.6% higher than that of HTTPS, at 800 clients.

6.2.2 Client benchmark

The client was benchmarked using Google Chrome, comparing the time for an AJAX call to be prepared, sent, received, and interpreted. The application used was SimpleChat, and each chat message posted was timestamped. Any timestamps in JavaScript inherently includes any time spent preparing and verifying HTTPS details, and thus all GlassTube computations are included in the timestamps as well.

The average of 20 samples was 8.4 ms for HTTPS, 10.2 ms for GlassTube, and 9.15 ms for HTTP. This shows clearly that the protocol used has little, if any at all, implication on the user experience for an application such as SimpleChat, as plain HTTP does not perform strictly better than the other two.

7

Discussion

This chapter first discusses how GlassTube differs from similiar protocols, after which it offers concluding remarks and summarizes the report, and lastly gives an outlook into the future of the GlassTube protocol.

7.1 Related Work

There has been several attempts to create an alternative to HTTPS for web applications, as the redundancy of confidentiality is apparant with applications such as Wikipedia. This section will put these in relation to GlassTube.

Adida presents SessionLock [40], a mechanism to protect a web session from eavesdropping. SessionLock uses HMAC to prevent eavesdroppers from simply reusing the session cookie to authenticate themselves. SessionLock does not prevent against active attacks, but will prevent session hijacking and thus incapacitate tools such as Firesheep. However, an active attacker can easily alter the client's behavior by modifying a response to contain different JavaScript, which could then be used to either leak the session key or make use of the compromised client to construct signed messages.

Dacosta et al. suggest One-Time Cookies [41] (OTC) as an alternative to using session cookies for authentication. OTC protects the session by sending a session key, encrypted, which is also used to sign the message together with each request. The stateless protocol is inspired by Kerberos, leading to a scalable design. However, the server responses are not signed, and thus the protocol is vulnerable to man in the middle attacks, in the same manner as SessionLock.

Singh et al. propose HTTPi [42], as an alternative to HTTPS that guarantees end-to-end integrity. They achieve convincing performance results by focusing on utilizing web caching. HTTPi shares much of the motivation with our approach when arguing that integrity without confidentiality is often desired. However, it occupies a somewhat different point in the design space. HTTPi is a direct alternative to HTTP and HTTPS, with possibilities for access control across HTTPS, HTTPi, and HTTP content. Similarly to HTTP and HTTPS, HTTPi relies on the support of the browser. In contrast, GlassTube is a lightweight approach that focuses on application-level support for integrity. GlassTube does not require browser modification. Being a customizable library, GlassTube features flexibility for supporting application-specific policies.

Choi and Gouda describe an integrity protocol for web applications, named similarly to the above protocol, HTTPI [43]. HTTPI is designed to allows intermediate cache servers to function, while still maintaining integrity. However, the protocol lacks protection from replay attacks, and it requires a plugin to function. Cache servers can be a great performance boost for web applications, which is most desirable. However, the choice of MD5 for hashing makes collision attacks feasible, leading to inferring the hash of the content, and hence opening for man in the middle attacks.

In the paper App Isolation [44] Chen et al. presents a means to tackle the problem of cross-site attacks that can occur while accessing multiple websites simultaneously in the same browser, such as cross-site request forgery. To address this problem they isolate browser sessions from each other. GlassTube provides the same level of protection when using JavaScript or a smartphone app without any additional efforts, as each browser window will have a local and protected session key, which cannot be accessed by other windows. When utilizing a browser plugin it is up to the implementation of the plugin to provide this separation. Efforts such as App Isolation thus becomes redundant if GlassTube is employed.

The tools like SIF [45] and SWIFT [46], based on Jif [47], allows the programmer to enforce powerful policies for confidentiality and integrity in web applications. The programmer labels data resources in the source program with fine-grained policies using Jif, an extension of Java with security types. The source program is compiled against these policies into a web application where the policies are tracked by a combination of compile-time and run-time enforcement. The ability to enforce fine-grained policies is an attractive feature of this line of work. At the same time, the enforcement is rather heavyweight, given that the programmer is required to use Jif as the programming language.

7.2 Conclusion

This section will present conclusions drawn from the material presented in this report.

We have proposed GlassTube, a lightweight approach to web application integrity. Such an approach is vital when confidentiality is not needed or undesired and when application-specific integrity policies are in place. GlassTube is compatible with several secure setup options with and without modified client. Upon successful setup, GlassTube guarantees per-message integrity, preventing a man in the middle attack from inferring changes to data between the client and the server, without being detected. GlassTube assures mutual authentication between client and server. As is common, the authentication of the user to the application is left to the application.

We have demonstrated that the deployment of GlassTube is lightweight, both in the web application setting and in the scenario of smartphone apps. Little effort is required of the developer to use the GlassTube library. GlassTube is fully transparent for the end user. The benchmarks from the case study show that GlassTube reduces the load compared to HTTPS. The performance results are encouraging, given that no optimization efforts were made.

GlassTube provides a solid foundation for future implementations both refining security policies and optimizing performance, so that it can be efficiently implemented and easily deployed in existing applications.

7.3 Future Work

This section suggests future work to complement GlassTube and bring a full-featured integrity protocol for web-applications closer to a state where it can be deployed with such ease that it a clear alternative to HTTPS for applications where confidentiality is unwanted.

An important avenue for future work is lifting the restriction GlassTube puts on the developer regarding HTTP headers: they can be used but are not protected by GlassTube. It will increase the usefulness of GlassTube if support for a selected few standard headers is added, since they are sometimes used by developers. To be able to add this support, future work is focused on an in-depth study of the standard headers.

Another direction of future work is focused on the truncation attacks in the TCP layer. When a user performs a number of actions in sequence, the adversary might cause unexpected results by dropping the last packets. A promising way to combat this is to implement application level transactions. This means that if an adversary tries to truncate the data stream it will be detected, and changes made by previous request during the transaction will roll back.

GlassTube will have the potential to further enhance flexibility over HTTPS-based applications if encryption is supported. This would enable the programmer to specify application-specific confidentiality and integrity policies. We conjecture that sending a few packets encrypted with GlassTube while already having a GlassTube session negotiated is more efficient than setting up a new HTTPS connection for these transfers.

Another improvement that GlassTube can benefit from is freeing the programmer from using the binary data of each image, instead of its path. A similar improvement can be also made for dynamically loaded frames and scripts. This can be accomplished by having GlassTube deployed as a proxy or a module in the web server, similarly to the technique by Lekies et al. [**?**].

Bibliography

[1] A. Barth, HTTP State Management Mechanism, RFC 6265 (Proposed Standard) (Apr. 2011).
URL `http://www.ietf.org/rfc/rfc6265.txt`

[2] W. Stallings, Cryptography and Network Security, 5th Edition, Pearson Education, 2011.

[3] Aung Khant, A Most-Neglected Fact about Cross Site Request Forgery, `http://yehg.net/lab/pr0js/articles/A_Most-Neglected_Fact_About_CSRF.pdf?1334750354` (Aug. 2010).

[4] Mitja Kolšek, Session fixation vulnerability in web-based applications, `http://www.acros.si/papers/session_fixation.pdf`, accessed: 2012-05-02.

[5] E. Butler, Firesheep, `http://codebutler.com/firesheep`, accessed: 2012-05-02.

[6] E. Rescorla, HTTP Over TLS, RFC 2818 (Informational), updated by RFC 5785 (May 2000).
URL `http://www.ietf.org/rfc/rfc2818.txt`

[7] National Institute of Standards and Technology, Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, SP 800-131A (Jan. 2011).

[8] National Institute of Standards and Technology, Recommendation for Random Number Generations Using Deterministic Random Bit Generations, SP 800-90A (Jan. 2012).

[9] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321 (Informational), updated by RFC 6151 (Apr. 1992).
URL `http://www.ietf.org/rfc/rfc1321.txt`

[10] T. Xie, D. Feng, How to find weak input differences for md5 collision attacks, Cryptology ePrint Archive, Report 2009/223 (2009).
URL `http://eprint.iacr.org/2009/223`

[11] Sasaki, Yu and Aoki, Kazumaro, Finding Preimages in Full MD5 Faster Than Exhaustive Search, in: Joux, Antoine (Ed.), Advances in Cryptology - EUROCRYPT 2009, Vol. 5479, Springer Berlin / Heidelberg, 2009, pp. 134–152.
URL `http://dx.doi.org/10.1007/978-3-642-01001-9_8`

[12] S. Turner, L. Chen, Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, RFC 6151 (Informational) (Mar. 2011).
URL `http://www.ietf.org/rfc/rfc6151.txt`

[13] National Institute of Standards and Technology, Secure Hash Standard, FIPS 180-1 (Apr. 1995).

[14] T. Polk, L. Chen, S. Turner, P. Hoffman, Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms, RFC 6194 (Informational) (Mar. 2011).
URL `http://www.ietf.org/rfc/rfc6194.txt`

[15] J. Kelsey, B. Schneier, Second preimages on n-bit hash functions for much less than 2^n work, Cryptology ePrint Archive, Report 2004/304, `http://eprint.iacr.org/` (2004).

[16] National Institute of Standards and Technology, Cryptographic Algorithm Object Registration, `http://csrc.nist.gov/groups/ST/crypto_apps_infra/csor/algorithms.html` (Feb. 2011).

[17] National Institute of Standards and Technology, The Keyed-Hash Message Authentication Code (HMAC), FIPS 198-1 (Jul. 2008).

[18] Internet Engineering Task Force, HTTP Authentication: MAC Access Authentication, draft-ietf-oauth-v2-http-mac-00 (May 2011).

[19] W. Diffie, M. E. Hellman, New directions in cryptography (1976).

[20] E. Rescorla, Diffie-Hellman Key Agreement Method, RFC 2631 (Proposed Standard) (Jun. 1999).
URL `http://www.ietf.org/rfc/rfc2631.txt`

[21] Getting Started, `https://developer.mozilla.org/en/AJAX/Getting_Started`, accessed: 2012-04-18 (Mar. 2012).

[22] L. Masinter, The "data" URL scheme, RFC 2397 (Proposed Standard) (Aug. 1998).
URL `http://www.ietf.org/rfc/rfc2397.txt`

[23] Ian Hickson, Cross-Origin Resource Sharing, `http://www.w3.org/TR/2011/CR-webstorage-20111208/` (Dec. 2011).

[24] A. Barbir, B. Cain, R. Nair, O. Spatscheck, Known Content Network (CN) Request-Routing Mechanisms, RFC 3568 (Informational) (Jul. 2003).
URL `http://www.ietf.org/rfc/rfc3568.txt`

[25] A. Barth, The Web Origin Concept, RFC 6454 (Proposed Standard) (Dec. 2011).
URL `http://www.ietf.org/rfc/rfc6454.txt`

[26] World Wide Web Consortium, Cross-Origin Resource Sharing, `http://www.w3.org/TR/2012/WD-cors-20120403/` (Apr. 2012).

[27] Open Web Application Security Project, Man-in-the-middle attack, `https://www.owasp.org/index.php/Man-in-the-middle_attack`, accessed: 2012-04-10 (Apr. 2009).

[28] Open Web Application Security Project, Cross-site Scripting, `https://www.owasp.org/index.php/Cross-site_scripting`, accessed: 2012-04-10 (Aug. 2011).

[29] Google Web Toolkit, https://developers.google.com/web-toolkit/, accessed: 2012-04-19.

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

[31] Burkholder, P., SSL Man-in-the-Middle Attacks, http://www.sans.org/rr/whitepapers/threats/480.php, accessed: 2012-04-11 (Feb. 2002).

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

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

[34] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, RFC 2616 (Draft Standard), updated by RFCs 2817, 5785, 6266 (Jun. 1999).
URL http://www.ietf.org/rfc/rfc2616.txt

[35] ECMA International, ECMAScript Language Specification, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (Jun. 2011).

[36] W3C Web Cryptography Working Group, Group charter, http://www.w3.org/2011/11/webcryptography-charter.html.

[37] S. Bellovin, Defending Against Sequence Number Attacks, RFC 1948 (Informational), obsoleted by RFC 6528 (May 1996).
URL http://www.ietf.org/rfc/rfc1948.txt

[38] J. Postel, Transmission Control Protocol, RFC 793 (Standard), updated by RFCs 1122, 3168, 6093, 6528 (Sep. 1981). URL `http://www.ietf.org/rfc/rfc793.txt`

[39] DefaultHTTPClient, `http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/impl/client/DefaultHttpClient.html`, accessed: 2012-04-19.

[40] B. Adida, Sessionlock: securing web sessions against eavesdropping, in: Proceedings of the 17th international conference on World Wide Web, WWW '08, ACM, New York, NY, USA, 2008, pp. 517–524.
URL `http://doi.acm.org/10.1145/1367497.1367568`

[41] I. Dacosta, S. Chakradeo, M. Ahamad, P. Traynor, One-time cookies: Preventing session hijacking attacks with stateless authentication tokens, `http://smartech.gatech.edu/handle/1853/42609`.

[42] K. Singh, H. Wang, A. Moshchuk, C. Jackson, W. Lee, Practical end-to-end web content integrity, in: Proceedings of the 21st international conference on World Wide Web, ACM, 2012, pp. 659–668.

[43] T. Choi, M. Gouda, HTTPI: An HTTP with Integrity, in: Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on, 2011, pp. 1 –6.

[44] E. Y. Chen, J. Bau, C. Reis, A. Barth, C. Jackson, App isolation: get the security of multiple browsers with just one, in: Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, ACM, New York, NY, USA, 2011, pp. 227–238.
URL `http://doi.acm.org/10.1145/2046707.2046734`

[45] S. Chong, K. Vikram, A. C. Myers, Sif: Enforcing confidentiality and integrity in web applications, in: Proc. USENIX Security Symposium, 2007, pp. 1–16.

[46] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, X. Zheng, Building secure web applications with automatic partitioning, Commun. ACM 52 (2) (2009) 79–87.
URL `http://doi.acm.org/10.1145/1461928.1461949`

[47] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, N. Nystrom, Jif: Java information flow, software release. Located at `http://www.cs.cornell.edu/jif` (Jul. 2001).