GlassTube

A Lightweight Approach to Web Application Integrity

Per A. Hallgren
Keyflow AB &
Chalmers University of Technology
Gothenburg, Sweden
per.zut@gmail.com

Daniel T. Mauritzson
Ericsson AB &
Chalmers University of Technology
Gothenburg, Sweden
daniel.mauritzson@gmail.com

Andrei Sabelfeld
Chalmers University of Technology
Gothenburg, Sweden
andrei@chalmers.se

Abstract

The HTTP and HTTPS protocols are the 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 prevents man-in-the-middle attacks and 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.

Categories and Subject Descriptors [*Security and privacy*]: Web application security

General Terms Security, integrity, man-in-the-middle-attacks

Keywords web application security, data integrity, lightweight enforcement, application-level security policies

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 Integrity in Web Applications

Information integrity is a vital security property in a variety of applications. In general, integrity is a versatile property. Indeed, security textbooks [16, 30] agree that it is hard to pin down the essence of integrity, and surveys [26, 32, 33], tutorials [18], and papers [4, 24] identify a range of integrity flavors.

PLAS'13, June 20, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2144-0/13/06...$15.00

In the setting of web applications, *data integrity* is particularly crucial. Data integrity, or simply integrity in the rest of this paper, requires that data sent over the network must be accurate and consistent with the intended message. This inherently means that information sent cannot be modified, and that the consignor is authentic. In contrast, *confidentiality* requires that sensitive information must not be leaked to an unauthorized party. *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.

The treatment of *sessions* is important for web application integrity. Sessions in web applications, intended to personalize user experience, are typically implemented by passing session identifiers via cookies between the server and the client. The cookies are sent with each request over the stateless HTTP protocol. A range of attacks such as *replay attacks* [35], *cross-site request forgery* [2], and *session fixation* [20] target stealing and abusing the session credentials in order to hijack sessions and impersonate users towards the server. The communication of session credentials in clear text has lead to the common misconception that *confidentiality* is needed in order to have a secure session towards a web server, in order to achieve *integrity*.

The only other standard alternative to HTTP for web applications is to use HTTPS, a web protocol that encrypts all communication using TLS/SSL. 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.

1.2 Importance of Integrity

Browsers indicate to users whenever they access a site over an unencrypted connection. However, many users associate encryption with security, and not with confidentiality. It is intuitive to most users that their connection to the bank should be encrypted. However, users are commonly far less aware about when the data they send and receive can be distorted by an unauthorized party. If integrity was maintained by default, we claim that the web would uphold the intuitive amount of security for a general web page.

Ubiquitous open wi-fi networks exacerbate the problem. Under an open configuration of a wi-fi network, as frequently used at 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. This type of vulnerability that can be exploited by script kiddies must be viewed as severe.

While it is essential that an attacker does not impersonate a user, hijacking sessions is extremely simple with many web popular services such as Wikipedia. This can be done using such popular tools as Wireshark for network monitoring and, for example, Opera to open a session with a user specified cookie. A successful attack can be mounted by non-experts in a matter of minutes. A man-in-the-middle can then freely modify any information sent between the server to the client. A victim service provider might suffer a damaged brand, reduced sales and general sabotage. A victim user might be infected with malware or provided with incorrect information.

We argue that integrity is vital for any web application, regardless of whether the application utilizes sessions. Clearly, there is a cost to maintain integrity, and in many cases the chances of a breach may not be so high as to weigh heavier than that cost. However, to neglect integrity implies both disregarding what a user sees on a site and disregarding what the server perceives that a user posts. To put it boldly, why communicate if you do not care if the content transferred between the two parties is arbitrarily modified?

1.3 Scenarios

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:

Public web site browsing. Open web sites such as Wikipedia allow users to manipulate public data. 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.

Provider or authority disruption. Even though in many cases a full-fledged man-in-the-middle attack may seem unlikely, they do exist outside the scope of an unprotected wi-fi. For example, a service provider has all the relevant tools and may have reason to interfere with their users traffic. Indeed, some ISPs are known to inject code into their user's web traffic on a regular basis [10].

1.4 Goals

Motivated by the above scenarios, our goal is to create an approach to integrity in web applications designed to authenticate both the client and the server towards each other. While we focus on the mutual authentication and integrity towards both the client and the server for each message within a session, we note that the authentication of *users* is an orthogonal issue [21], which we leave to the application. Both parties must be able to check that all packets originate from the other party in the conversation, and must also be able to verify the integrity of each message. Users and servers will thus be protected from session hijacking. 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 approach for integrity in web applications. It is thus important to support the approach with a proof-of-concept implementation, in order to evaluate programming overhead for the developer as well as indicative performance overhead.

The performance of an integrity approach for web applications is vital: if it does not relieve the server compared to HTTPS, there is no tangible reason to choose it over HTTPS. Hence, an important goal is performance boost compared to HTTPS, decreasing the cost per client. When bandwidth, database access, and/or other I/O form a bottleneck, the costs are still reduced in terms of computational resources on the server side. It is particularly interesting to study the overhead in an entire application, because optimizing the building blocks for cryptographic primitives does not necessarily mean overall performance improvement [31].

The final goal is a flexible framework that allows extensions where the degree of integrity can be tuned in an application-specific way. This is in contrast with application-agnostic HTTPS where the encryption method is set to null, as discussed in Section 6. GlassTube will function well for singular services, without requiring global support by browsers.

1.5 Contributions

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. The design will protect web applications against man-in-the-middle attacks where the attacker has complete control of the network. GlassTube guarantees protection against the following:

1. Modification of the data stream
2. Session hijacking
3. Reordering and replay attacks

GlassTube provides a general method for integrity in web applications and smartphone apps. GlassTube includes an initial setup phase, including a key exchange phase, where the server and client collaborate to establish a session key, to be later used for creating and verifying message authentication codes (MACs). The setup phase requires a connection which guarantees end-to-end integrity and the authenticity of the server; this 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 MACs 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.

2. The GlassTube Approach

GlassTube is designed to provide integrity over insecure connection, preventing manipulation of the data stream. This section specifies the GlassTube approach. The approach consists of two phases. The GlassTube Setup is the first phase. It maintains distribution of code and the key exchange. The second phase of the approach is the

GlassTube Data Transfer (GTDT), which ensures integrity between the web server and the client.

2.1 GlassTube Setup

GTDT requires that the code running on the client is not modified by an attacker, and that a session key is shared between the client and the application server, henceforth called the data site, which is secret except to the data site and the client. GlassTube can be initialized in any fashion that meets these requirements. This section describes some options to distribute code and how keys can be exchanged, independently from each other.

2.1.1 Client Code Distribution

Since web applications are not present on the client per default, the client-side code often needs to be sent at the beginning of the session. If an attacker can alter the client's code 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 the code's integrity can be guaranteed. Client code can be distributed either *statically* or *dynamically*. 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. In this case, for an end user to communicate with a web server running GlassTube, he or she needs 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 dynamic code consists of JavaScript, Flash, Silverlight, or Java applets embedded within the page. Dynamic code distribution for GlassTube must be done by a host which guarantees end-to-end integrity, and for which the client is able to verify the server's identity. This host is henceforth called the secure site. The secure site only communicates 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 are sent from the secure site.

2.1.2 Key Exchange

GTDT 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 create and verify MACs. A new session key is negotiated at the beginning of each session. It is not possible to reuse keys negotiated during TLS; JavaScript, and many server frameworks, does not provide access to transport layer information.

We propose the use of a known protocol, the Authenticated Diffie-Hellman protocol [12–14] (ADHE) to exchange keys, an authenticated version of the classical Diffie-Hellman key exchange [13]. The scheme requires that the client possess a public key belonging to the data site. The public key can be acquired either by distributing the key with the code or by fetching it from a secure site after code distribution.

The Diffie-Hellman protocol uses two domain parameters, a generator g and a prime number p. The domain parameters are not necessarily secret. The two parties within a Diffie- Hellman key exchange each generate 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 knows both X and Y, but neither of x and y, and thus cannot find Z [13]. The shared key will be called Z and *session key* interleaving throughout the paper.

Under the Authenticated Diffie-Hellman key exchange, the server's public key is signed using a DSA or RSA certificate. The client's public key can also be signed in the same manner, if client certificates can be distributed in a feasible manner. The client is thus able to verify the server's public key's integrity, preventing man-in-the-middle attacks [12, 14].

Random numbers are needed in the above key exchange. 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 by the secure site. It is possible for the server to compute the entire key as well as to simply generate a random number. It is preferred to let the client compute the keys instead, to relieve the server from the extra work. Random numbers sent from the secure site requires both integrity and confidentiality, as well as authentication of the secure site. In dynamic code distribution, the random numbers can be sent to the client embedded in the code. In that case, the code will need to be encrypted, as the numbers must remain secret. However, since the only practical protocol to securely distribute code is HTTPS, this would typically be done without any performance loss or modifications of the secure site.

2.2 GlassTube Data Transfer

Data transfers done with GTDT guarantees data integrity. The authenticity of the sender of each message is be verified, and messages are prevented from being replayed. This section details how this is accomplished.

2.2.1 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 valid. GlassTube addresses this by appending a unique identifier to each message, called *message identifier* or $I$for the remainder of this paper. I can be any entity using which the following two properties can be guaranteed:

- An attacker must not be able to forge messages by replaying them
- An adversary must not be able to reorder messages within a session

There are several common ways to guarantee freshness properties, each with their respective strengths and weaknesses. Timestamps are very common, as they are easy to implement and very efficient. As no additional information needs to be stored to verify if a timestamp is fresh, it is also a very scalable design. Timestamps does not, however, uphold either of the above properties. Unique nonces are also a common design, as they do not allow for a window in which messages can be replayed, and thus protect completely against eavesdroppers. However, nonces can be reordered if the attacker has sufficient access to the network. Sequence numbers provide desired security and can be efficiently implemented, making them a particularly suitable message identifier for GlassTube. Each party must for the duration of the session remember their own current sequence number, and which number is expected from the other party with the next message. When either party sends a message, they increment their own number by one. When receiving a message, the message identifier is verified by comparing the stored number with the received number; the received number must be equal to the stored number in order for the message identifier to be valid.

2.2.2 MAC

The data which is authenticated by the MAC in GlassTube includes all application-level information that defines the request or response, in the sense that if it is changed it will modify the data stream. A message should not be substitutable by a message which is not equivalent to the original. GlassTube must guarantee that the receiving end can verify the following based on the MAC, on application level:

1. The message has been delivered to the correct handler
2. No data used by the application has been modified, including headers values
3. This is the first time the message has been received
4. Messages are not being reordered

For a request, the defining data includes the complete URL (1), the request method (1), request parameters (2), and the message identifier (3, 4). For a response, defining data includes the response code (2), the response data (2), finally the message identifier (3, 4), and also the MAC for the request being responded to (1).

By including the URL in the MAC, the server can trivially verify which handler was intended to receive the request. For the client, there is no data in ordinary HTTP information which specifies a handler, as they are not apparent in traditional web applications. As a browser may execute multiple AJAX requests simultaneously, mapping responses to the correct handler is only possible by looking at port numbers, which is how the web browser determines the fact. GlassTube enables this to be done by the application by including the request's MAC in the response's MAC.

All header fields can not be included in the MAC, as content in header fields is frequently modified by intermediate hosts. Instead, the application can specify which headers it makes use of. Application headers included in the MAC are specified per request by the application using the X-GlassTube-Extra-Headers header field.

Reordering request parameters or header fields will produce a different MAC. Thus, the order of these are explicitly defined by the GlassTube protocol, rather than by the order in which they are received by the application. Both header fields and request parameters in HTTP are constructed as key-value pairs, and both are henceforth called message variables. A response cannot contain request parameters, and message variables in a response will thus only consist of header fields. We denote a message variable as $V_i = (k_i, v_i)$, for it's key k_i and value v_i. The union of all message variables are sorted alphabetically, resulting in a list ω, in order to deterministically determine the order on an arbitrary platform. The list of alphabetically ordered message variables ω are concatenated as follows while preparing the MAC, resulting in the string τ:

$$\tau = cat(k_0, "=", v_0) \tag{1}$$

$$\tau = cat(\tau, cat(k_i, "=", v_i, "\&")) \qquad \forall i \in [0 \,..\, |\omega|] \tag{2}$$

Where $cat()$ performs string concatenation of all input parameters, with respect to order, and returns the result, while not modifying the parameters. The algorithm will result in a string which looks very much like GET request parameters.

As HTTP is an asymmetric protocol, the MAC is constructed slightly differently by both parties. The server will compute the MAC using a keyed-hash function Θ, τ as computed above, the session key Z, the message identifier I, the response body $body$, and the response code rc as:

$$MAC = \Theta(Z, concat(rc, \tau, body, req\_mac, I)) \tag{3}$$

The client computes the MAC using τ, the message identifier I, the request method $method$, and the URL url as:

$$MAC = \Theta(Z, concat(method, url, \tau, I)) \tag{4}$$

Message variables, the response body, and the response code are sent by the web application as usual.The MAC and the message identifier are included in each request and each response using two header fields, for the message identifier the header field is called X-GlassTube-Message-Identifier and for the MAC it is called X-GlassTube-MAC.

The HMAC-SHA1 MAC algorithm is particularly suitable for use with GlassTube as it is efficient and secure [28]. Cipher suite negotiations are common in similar protocols, but are not feasible in GlassTube web application, or smartphone app. In both cases, the client and the server are parts of a specifically designed system. The smartphone app is designed to work with a specific web service as a back-end, and the web app is sent by the web service by the beginning of the session. Thus, the programmer will decide ciphers as they construct the service. The browser plugin is more complicated, as it can be used towards several, different web services. In order for it to conform to the GlassTube protocol, browser plugins are restricted to HMAC-SHA1. Servers must announce GlassTube support to browser plugins unless they support HMAC-SHA1. Currently, only HMAC-SHA1 is supported by the GlassTube libraries.

2.2.3 Verifying a Message

Upon receiving a message, the recipient first calculates the MAC for the message, as described in Section 2.2.2. In the case that the MAC does not match the received MAC, 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 The procedure is illustrated in Figure 1.

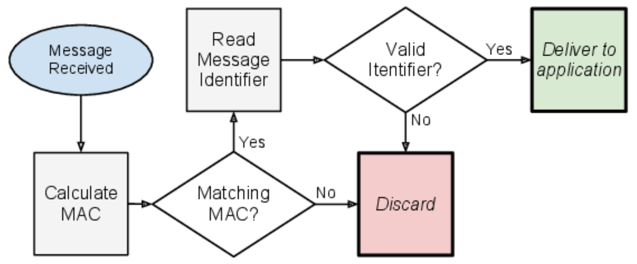

Figure 1. The verification process

Whenever the server discards a message, it must send an error message to the client, indicating what went wrong, so that a proper error message may be displayed to the end user. The error message is authenticated with GlassTube, as a normal message. Whenever the client discards a message, it must restart the session. The client is always able to address problems gracefully. If the client's message is faulty, it will receive an error message from the server, and resets the session. If the server's message is faulty (including if an error message is faulty, in the case that both the server's and the client's messages are being altered), the client resets the session.

3. GlassTube Instances

GlassTube offers a range of setup and deployment choices. This section outlines three different GlassTube instances to illustrate practical applications of the concepts presented in the previous section. Note that although all examples describe how a service may support GlassTube, the service may still choose to support HTTPS and HTTP for some pages at the site, in parallel with GlassTube.

However, it is important that the programmer is well aware that any HTTP transfers within a GlassTube session may void the integrity of messages later in the session, if used carelessly. For instance, many web applications depend on third parties to collect statistical data about how users traverse their sites. One such example is Google Analytics. Google Analytics is set up with a very short code snippet, which fetches another JavaScript file from Google's servers. It is vital that all transfers are all done using HTTPS, even when initiated by an included library or plugin. Otherwise, the site will be susceptible to man in the middle attacks. In the case of Google Analytics the request is done with SSL, and there is no need to worry.

3.1 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 2.

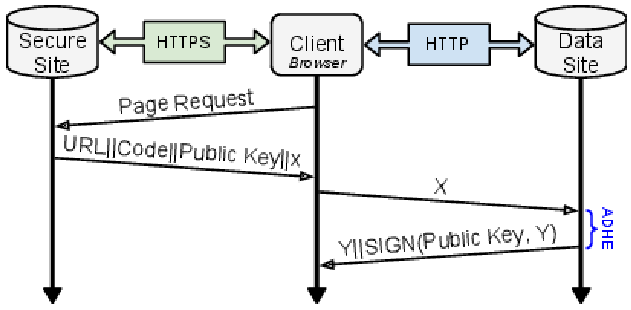

Figure 2. GlassTube for a generic web application

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 private key x to be used in the ADHE key exchange. While sending the entire user interface from the data server might be somewhat cheaper, the user will experience some flicker and extra loading time, and thus sending a first view from the secure site will create a much smoother rendering of the page. When the client has interpreted the received code it will initiate a key exchange towards the data site using AJAX, after which it will have established a GlassTube session towards the server. Following requests should start to populate the rich web application, as the user navigates the already loaded UI, while never refreshing the web page, as that would incur another key exchange. The integrity of each message during GTDT will be guaranteed by GlassTube, and sensitive - though not secret - data can be sent confidently.

The developer sets up two sites, the secure site running HTTPS and the data site servicing HTTP. The secure site and data site may 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 may import a GlassTube library and mark individual pages to use GlassTube. As the secure site and the data site makes use of different protocols (HTTPS and HTTP, respectively), they are separate origins and separated by the same origin policy [3]. 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) [37]. A typical client code needs few modifications in order to work with GlassTube.

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 an ordinary image tag, such as `<img src="image.jpg" />` is rendered. The browser will fetch the image $image.jpg$ from the server, without resorting to AJAX and JavaScript - indeed, JavaScript is completely oblivious regarding all data transfers which it does not initiate itself. The same is true for `<script>` and `<iframe>` tags, which means that if these tags are used by in the application to reference a resource at the server, they may break GlassTube. However, a programmer may make use the data URI scheme [25], 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 fully transparent to the end user with this setup. 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 pre-distributed code and public key.

3.2 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 3. Upon connecting to a web site, the plugin announces that it can handle GlassTube sessions, and the secure site will respond appropriately.

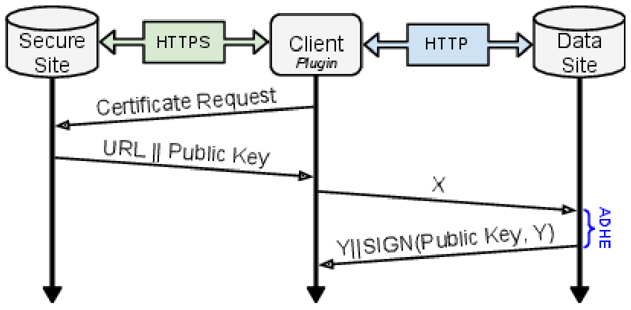

Figure 3. GlassTube as a browser plugin

A generic plugin removes the need for individual application providers to maintain and distribute plugins, and the end user only needs to take one action to be able to use any number of GlassTube sites. Benefits of using a plugin instead of dynamic code is potentially better client performance and the possibility to handle HTML-initiated requests. Note that due to the lack of a cipher suite exchange, the browser plugin will not be able to connect to services not supporting HMAC-SHA1.

An alternative approach is to store the public key along with the secure site's SSL certificate and the signature of the public key at the data site; the signature is created with the certificate. The browser plugin can then initiate the session directly towards the data site, and thus removing the reliance on HTTPS. The browser plugin verifies the data site's authenticity of the public key, and thus the data site, by verifying the signature using the certificate and the certificate towards the browsers installed root certificates.

3.3 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. 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 is dedicated to one web site, thus an end user has to install one app per site.

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

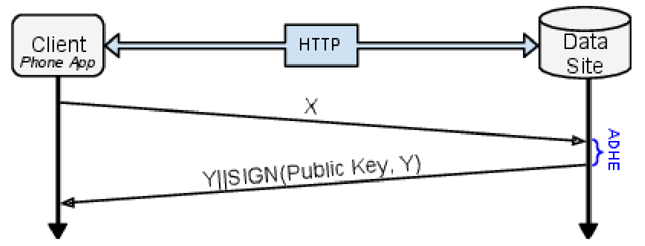

Figure 4. GlassTube as used by a smartphone app

app for key exchange and GTDT can be accessed from a library, thus very little extra work needs to be done. The data site is set up as in Section 3.1.

3.4 Notes on Scalability

The nature of GlassTube has some implications on large and complex systems, this section will briefly discuss how load balancers and cache servers integrate with GlassTube. As GlassTube enforces end-to-end integrity for the span of a session, the client will experience it as though a man-in-the-middle attach has occurred if a load balancer changes application server, and the application servers of the service does not somehow share GlassTube session keys.

Intermediate cache servers will not be able to cache any GlassTube requests, as each request is made unique by the message identifier. This fact should prevent caching completely, but misconfigured caches may erroneously cache GlassTube requests. If this occurs, the client will experience it as though an attacker is replaying old responses, and reset the session.

4. Security Considerations

This section discusses the security of GlassTube, and is divided into three parts. The first part details the different parts of the GlassTube approach, highlighting each part of a GlassTube session. The second part covers GlassTube's security guarantees. The third part pursues general security considerations.

4.1 The GlassTube Approach

This section covers the security within each stage of the GlassTube Setup, and the security of the GlassTube Data Transfer.

4.1.1 Client Code Distribution

To ensure that messages are handled correctly, it is important that the code running on the client is not modified by an attacker. Two options were presented in Section 2.1.1, 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.

For dynamic code distribution, the service is responsible for delivering the code to the client. As man in the middle attack can be attempted at any point during the transmission, end-to-end integrity must be guaranteed, and the client must be able to verify the server's identity. 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.

4.1.2 Key Exchange

Authenticated Diffie-Hellman is a canonical way of exchanging keys in a secure manner, as it a part of TLS. Diffie-Hellman guarantees that only the communicating parties can compute the session key; eavesdropping is ineffective. Diffie-Hellman is, however, susceptible to man-in-the-middle attacks, where the attacker exchanges one key with the server and one with the client, as the identity of the server is not verified by the client. ADHE successfully thwarts man-in-the-middle attacks, as it uses a previously known server certificate, with which the client verifies the server's identity.

An attentive reader might notice that the server does not have any previous knowledge about the client, and thus a man-in-the-middle may seem possible by impersonating the client. However, this requires either that the impersonated client does not verify the server's signature in the key exchange, or that the adversary does not alter the server's Diffie-Hellman public key. In the latter case, the adversary can not interfere in the following session without being discover, and in the former the impersonated client is faulty.

4.1.3 GlassTube Data Transfer

The MAC of each message is the most vital part of GlassTube, as it ensures integrity of every message. By including the request parameters and the URL in the MAC, GlassTube asserts that the client will know if intended data is delivered to the intended service on the intended server. MACs are constructed in an asymmetric manner for requests and response, as responses and requests contain different information. Note that an adversary can not confuse the protocol by sending requests to the client or responses to the server, as such packages are discarded already by the browser or web server, and are not accepted by the application.

For a request, the attacker may try to forge the URL, the request parameters, or the message identifier. If the request parameters or the message identifiers are modified, the message will trivially be discarded by the GlassTube service. If an attacker modifies the URL, the request may be delivered to a different web application, in which case the client will reject the response. If the packet is delivered to the same application but using a different URL, the application will detect that the MAC is invalid, and will discard the packet.

With each response, the response code and data are included in the MAC along with the message identifier. The data and message identifiers are authenticated, and only handled by the client application, meaning that any modification will invalidate the MAC. Modifying the response code can have more complicated consequences, as the response code is interpreted by the web browser, but none of these cause anything else than truncation. By modifying the port numbers in a response, an adversary may cause a request to be delivered to the wrong response handler at the client. By embedding the request's MAC in the response's MAC, such attacks are prevented, as the client will be able to identify the request is handled by the intended request handler.

4.2 Security Guarantees

This section will describe how GlassTube protects against 1) Modification of the data stream, 2) Session Hijacking & 3) Reordering and replay attacks.

4.2.1 Modification of the Data Stream

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. GTDT will protect a correctly set up session during data transfer, as described above.

An active attacker may during a full man in the middle attack modify the entire packet, which includes TCP and IP headers. However, as both the client and the server verifies information on the application layer, nothing can be achieved except for truncation, if the attacker modifies the lower levels - assuming that no data from the lower layers are used by the application. By verifying each request using a secure MAC while keeping the session key secret, there can be no modification of the data stream during a GlassTube session.

4.2.2 Session Hijacking

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 2.1.1. Both of these methods are considered secure, see Section 4.1.1. Once a session is set up an attacker can not impersonate the client, as the session is protected by each message being authenticated with MAC. If an attacker, Eve, would attempt to use the session cookie of another user, Bob, the server would at first glance think that Eve is actually Bob, thus using Bob's session key to verify the MAC. The verification would fail, as Eve does not know Bob's session key, and cannot create a correct MAC.

Note that both server and client may relay messages on to a third party, outside of the protocol. This does not violate any of the guarantees made by GlassTube, and there is no feasible way to prevent such behaviour. Once a session is set up, both parties have expressed explicit trust for the other.

4.2.3 Modifying HTTP headers

An active attacker can add, remove and modify HTTP headers. If the application uses HTTP headers, they must be included in the MAC. 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. Thus, all malicious modifications of HTTP headers in a request is easily detected, and analysis of the server becomes simple. Any modified headers will either not be acted upon, or be included in the MAC. If headers are modified in responses, analysis becomes more complex, as we may not presume that the programmer is in control of the inherent behavior of the web browser.

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 MAC will be void. An attacker can use the Location header [15, 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 4.1.3, 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. Lastly, an attacker may attempt to change caching behaviour at the client. This, however, will not have any affect on the application, as each GlassTube request is unique and not cacheable.

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.

4.2.4 Replay and Reordering Attacks

GTDT uses message identifiers, as described in Section 2.2.1, to prevent both replay and reordering attacks. As the message identifier is unique for every message, the recipient will only accept a specific message identifier once. Thus, replay attacks are not possible during a GlassTube session. Reordering is made impossible by sequence numbers being sent in strictly increasing order.

4.3 General Security Considerations

The subsection will discuss general security issues, which are not confined within the limits of web applications integrity.

4.3.1 Entropy

JavaScript currently has no support for generating cryptographically secure random numbers. Unless a third party library is used, only `Math.random()` is available, which is not cryptographically secure. Although adding cryptographically strong random number generation to JavaScript API is only a matter of time [36], 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 cannot guarantee cryptographically secure random numbers. In this case, the secure site generates the random numbers and sends them securely to the client. This is the choice taken in our GlassTube instances.

4.3.2 User Authentication

Recall that user authentication is an orthogonal issue [21] left to the application. 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.

4.3.3 Denial of Service

GlassTube does not have a large performance impact on the secure site, as this site will always perform 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. However, it is very likely that the work done by the application itself will by far exceed that of any method for maintaining integrity.

5. Case Study

This section 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.

5.1 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 [17] (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 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 5.2. The JavaScript client is developed to assert that the user experience is not noticeably affected by GlassTube.

5.1.1 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 is capable of serving both the Java client and the web client respectively. The Java client uses static code distribution, while the JavaScript client makes use of dynamic code distribution. To the Java client, we thus only need to send the data site's public key, while for the web client there's also a need to serve JavaScript and the client's private key for the Diffie-Hellman key exchange. As the secure site only takes part in the setup phase no further data is handled by this server.

To make development at the data site easy, a GlassTubeServlet was created. It extends the HTTPServlet and adds GlassTube specific functions for the key exchange and for constructing and verifying MACs. The GlassTubeServlet demands that the first message from a client contains the client's public key for the Diffie-Hellman key exchange, from which the servlet computes the session key. The data site then signs its public key and sends the key as well as the signature to the client, concluding the key exchange. All servlets extending GlassTubeServlet on the data site are now ready to use GTDT. All information required for GTDT is stored in the server's session storage.

5.1.2 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 and constructing and verifying MACs are thus coded in Java, while data transfers during GTDT 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 the key exchange by computing the client's public key with the server-generated private key, embedded in the code. The client will also have the data site's DSA public key, which the client uses to verify the server's response. When the key exchange is complete, the web application is ready to be used by the end user.

5.1.3 Java Client

The Java Client is able to make use of Java's standard API to provide all needed cryptographic functions. It fetches the data site's public key and a server-generated private key from the secure site, to emulate a web client, after which the Java client commences the key exchange towards the data site. When keys have been exchanged, GTDT is ready to be used.

5.2 Benchmark

This section details the results of a series of tests conducted to verify how GlassTube performs in relation to HTTP and HTTPS. 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 the end user. As each GlassTube message must be uniquely identifiable, lest the protocol be vulnerable to replay attacks, intermediate caches will unfortunately not be able to help boost the performance of GlassTube. This is true also for HTTPS, while HTTP can obtain significant boosts by caching.

5.2.1 Server Benchmark

The benchmark is done against a very simple chat application called SimpleChat, using static fields to maintain the state of the web application. Each access to any of SimpleChat's functions is counted as a successful request by a client. A benchmark using each of the three different techniques, HTTP, HTTPS and GlassTube is performed. During the HTTP benchmark only the data site will receive traffic, during the HTTPS benchmark only the secure site will receive traffic, and during the GlassTube benchmark both will receive traffic. Therefore, the same machine is hosting both the secure site and the data site in order for the same computational resources to be available during all of the benchmarks.

The benchmarks were carried out towards a server running Tomcat 6, using standard configuration. The server machine used in the server benchmark is a Packard Bell Dot M/A-NCD/711, with a 1.2 GHz 64 bit processor with a single core, and 2GB of RAM. The SSL connection towards the server was for both clients using 256 bit AES in CBC mode, with SHA1 for message authentication and ECDHE_RSA as the key exchange mechanism.

Figure 5 plots the result of a benchmark conducted towards HTTP, HTTPS and GlassTube. 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 messages of 4096 bytes at an interval of between 10 and 300 milliseconds. The Max label in Figure 5 shows the maximal throughput for a server with unlimited processing power and a network without latency using the given client configuration.

The chosen configuration gives measurements with very little focus on how setting up new connections within the different protocols perform. However, it is very common that a user sends a multitude of requests to a server. Loading www.facebook.com, the authors observe 144 requests within the first 3 seconds at the time of writing. At the very least, a site will contain a script file, a style sheet and a couple of images besides the markup. Given that the user will navigate to a couple of pages within the site, the number of requests used during data transfer quickly grows to make the setup less prominent, when comparing the performance of the different protocols.

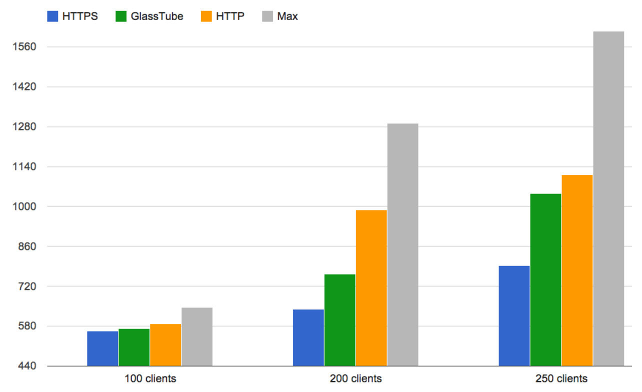

Figure 5. Benchmarks for performance

Figure 5 shows what the throughput limit for each setup is. HTTP will follow closely to the maximal limit until just about 1000 packets per second is reached, and reaches it's limit at around 1150 packets per second, while HTTPS can handle about 800 packets per second and GlassTube can process around 950 packets per second. The number of requests that can be served by GlassTube in this setting is at 250 clients 32% higher than that of HTTPS and 6% lower than HTTP, while HTTP can serve 41% more clients than HTTPS.

It is clearly visible that there is a performance gain in the above scenario when using GlassTube instead of HTTPS. GlassTube is also very close to the performance of HTTP at 250 clients, which is notable. While this does not mean that GlassTube will provide a performance boost as compared to HTTPS for an arbitrary service, it proves that it is possible to find cases where it is profitable to use application-level integrity enforcement, instead of resorting to lower-level techniques. Be it GlassTube, a derivate or later version of GlassTube, or a completely different protocol.

5.2.2 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. All GlassTube computations are included in the timestamps. The machine used in the client benchmark was a 13" MacBook Pro, running Intel i5 2.4 GHz CPU, and 4GB of RAM.

The average of 100 samples was 7.42 ms for HTTP, 9.9 ms for GlassTube, and 11.61 ms for HTTPS. These numbers are tiny, and the delay is not noticeable by a user. The results are consistent with the results on the server side, and shows that GlassTube can be implemented efficiently in JavaScript, as well as in Java.

6. Related Work

We share much of the motivation with work on application-level integrity described below. However, our work offers substantial added values: protection against active attackers and fine-grained application level integrity support.

SSL supports both null encryption and null authentication, both in which the respective security property is neglected. When null encryption is used, SSL functions as an integrity-only protocol. However, neither null encryption or authentication is allowed by any major browsers[22]. IPsec supports end-to-end integrity, using Authentication Header (AH) in Transport Mode. However, when AH is used in transport mode, the origin IP address is included in the signature[29]. This means that AH cannot be used in combination with NAT, and thus is not useful for most practical practical applications. We emphasize that in contrast to GlassTube, both SSL with null encryption and IPsec AH transport are protocol-level, lacking flexibility for expressing application-specific policies.

Adida presents SessionLock [1], 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 prevents session hijacking and thus incapacitates tools such as Firesheep. However, an active attacker can easily alter the client's behavior by modifying a response to contain different JavaScript, which can then be used to either leak the session key or make use of the compromised client to construct signed messages.

BetterAuth [21] by Johns et al. describes a non-regressive approach to authentication, secure by default. BetterAuth is focused upon authentication, which does not handle integrity in its abstract sense. Orthogonal to GlassTube, the focus is on user authentication rather than application integrity. A common case with open services is to use authenticators specific for every application that uses the service, through e.g. OAuth [19], and to not only authenticate the user.

Dacosta et al. suggest One-Time Cookies [11] (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 [34], 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, HTTPi 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 [7]. HTTPI is designed to allow 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.

Chen et al. present App Isolation [6] against cross-site attacks that occur while accessing multiple websites simultaneously in the same browser, such as cross-site request forgery. 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 has 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 [8] and SWIFT [9] allow 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 [27], 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. At the same time, the enforcement is rather heavyweight: the programmer is required to use Jif as the programming language.

7. Conclusion and Future Work

We now summarize the results and give an outlook onto future work.

7.1 Summary

We have proposed GlassTube, a lightweight approach to web application integrity. Such an approach is vital when confidentiality is not needed or even 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 for each message within a session. As is common, the authentication of the user to the application is left to the application [21].

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.

7.2 Future Work

An important direction for future work is focused on detecting truncation attacks. 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.

GlassTube could further enhance flexibility over HTTPS-based applications if encryption could be supported by GlassTube. This will 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. [23]. Future work includes studying the performance implications.

Acknowledgments

Thanks are due to Cedric Fournet, Martin Johns, and Frank Piessens for the fruitful feedback. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

[1] Ben Adida. SessionLock: securing web sessions against eavesdropping. In *Proc. International Conference on World Wide Web (WWW)*, pages 517–524, 2008.

[2] 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, August 2010.

[3] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

[4] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *Proc. International Conference on Information Systems Security*, LNCS, December 2010.

[5] Eric Butler. Firesheep. http://codebutler.com/firesheep.

[6] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin 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, pages 227–238, New York, NY, USA, 2011. ACM.

[7] Taehwan Choi and M.G. Gouda. HTTPI: An HTTP with Integrity. In *Proc. Computer Communications and Networks (ICCCN)*, pages 1–6, August 2011.

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

[9] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Building secure web applications with automatic partitioning. *Commun. ACM*, 52(2):79–87, February 2009.

[10] C. Chung, A. Kasyanov, J. Livingood, N. Mody, and B. Van Lieu. Comcast's Web Notification System Design. RFC 6108 (Informational), February 2011.

[11] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. http://smartech.gatech.edu/handle/1853/42609.

[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[13] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. on Information Theory*, 22(6):644–654, November 1976.

[14] W. Diffie, P. C. Van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[16] D. Gollmann. *Computer Security (2nd Edition)*. Wiley, 2006.

[17] Google. Google Web Toolkit. https://developers.google.com/web-toolkit/.

[18] J. Guttman. Invited tutorial: Integrity. Presentation at the Dagstuhl Seminar on Mobility, Ubiquity and Security, February 2007. http://www.dagstuhl.de/07091/. Slides at http://web.cs.wpi.edu/~guttman/.

[19] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), April 2010.

[20] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1531–1537, New York, NY, USA, 2011. ACM.

[21] Martin Johns, Sebastian Lekies, and Walter Tighzert. Betterauth: Web authentication revisited. In *28th Annual Computer Security Applications Conference (ACSAC 2012)*, 2012.

[22] Kenji Urushima. SSL/TLS Supported Cipher Suites. http://www9.atwiki.jp/kurushima/pub/pkimisc/SSLTLS_CipherSuite_Support_Table_.html, March 2010.

[23] Sebastian Lekies, Walter Tighzert, and Martin Johns. Towards stateless, client-side driven cross-site request forgery protection for web applications. In *5th conference on "Sicherheit, Schutz und Zuverlässigkeit" (GI Sicherheit 2012)*, 2012.

[24] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST'03)*, 2003.

[25] L. Masinter. The "data" URL scheme. RFC 2397 (Proposed Standard), August 1998.

[26] T. Mayfield, J. E. Roskos, S. R. Welke, J. M. Boone, and C. W. McDonald. Integrity in automated information systems. Technical Report P-2316, Institute for Defense Analyses, 1991.

[27] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[28] National Institute of Standards and Technology. Cryptographic Algorithm Object Registration. http://csrc.nist.gov/groups/ST/crypto_apps_infra/csor/algorithms.html, February 2011.

[29] B. Noble, G. Nguyen, M. Satyanarayanan, and R. Katz. Mobile Network Tracing. RFC 2041 (Informational), October 1996.

[30] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing (4th Edition)*. Prentice Hall, 2006.

[31] M. Roe. Performance of protocols. In *Security Protocols*, volume 1796 of *LNCS*, pages 147–152, 2000.

[32] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[33] Ravi S. Sandhu. On five definitions of data integrity. In *Proceedings of the IFIP WG11.3 Working Conference on Database Security VII*, pages 257–267, 1994.

[34] K. Singh, H.J. Wang, A. Moshchuk, C. Jackson, and W. Lee. Practical end-to-end web content integrity. In *Proceedings of the 21st international conference on World Wide Web*, pages 659–668. ACM, 2012.

[35] William Stallings. *Cryptography and Network Security*. Pearson Education, fifth edition, 2011.

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

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

A. HTTP Header Fields

A.1 Volatile Headers

Headers listed in this section are frequently changed by intermediate hosts. Thus, including them in the MAC will often cripple an application. Potential attack vectors are considered for each header field.

An adversary can modify headers with the intent to make either browsers or server-side caches to present a cached version instead of a fresh response. This, however, will not be possible since each GlassTube message includes a message identifier which makes all requests unique and not cacheable. These include the fields Cache-Control [15, p. 108], Date[15, p. 124], Age[15, p. 106], If-Modified-Since[15, p. 130], If-Unmodified-Since[15, p. 134], If-Range[15, p. 133], Vary[15, p. 145], Expires[15, p. 127] and Last-Modified[15, p. 134].

A number of header fields are used mainly to keep track of different properties of intermediate hosts. Modification of such may lead to truncation of the data stream, but none modifies the application data in any way. They include the Via [15, p. 116], Warning [15, p. 145] and Connection [15, p. 146] header fields. Other header fields used in relation to proxies and intermediate hosts are Max-Forwards [15, p. 136], Proxy-Authenticate [15, p. 137] and Proxy-Authorization [15, p. 137]. The Max-Forwards header field can never have any other effect on a stream than truncation, as it only specifies how many times a packet can be forwarded by intermediate hosts. The Proxy-Authenticate response header is used by proxies to require the requester to supply a Proxy-Authorization request header. The first is sent out by proxies, and the latter is used to authenticate the end user towards a proxy. If a request or response causes caches or proxies to modify the message, the MAC will become invalid, and the session will be reset.

A.2 Application Headers

Headers mentioned in this section should be safe to include in the MAC, while leaving them out of the MAC will not allow an adversary to modify the data stream if they are not used by the application. Many of these headers are not modifiable form JavaScript, but they are still consider mutable as the adversary will have full access to modify messages sent over the network. If any header is used by the application, it must be included in the MAC.

The Retry-After response-header field [15, p. 140] is used to inform the client of how long a resource is expected to be unavailable, or for how long the client should stall before following a redirect. The data stream can not be modified in any way through the use of this parameter, it may however cause truncation of the stream.

The Content-MD5 header field [15, p. 121] is the MD5-signature of the entity body. If the Content-MD5 header field is modified, without the entity body being modified, the receiving party must decline the request. If both the Content-MD5 header field and the entity body is modified, in such a way that the header field is the MD5 hash of the entity body, the GlassTube MAC will be incorrect for the received entity body, and the request will be declined. This header field is completely redundant in a GlassTube session and should not be used.

The Range request header field [15, p. 138] is used to specify what parts of (which range of bytes) an entity is wanted by the application. This can be useful when retrieving a large entity. The server may decide to disregard a range request. Whenever a certain range of bytes is sent in a response, the exact range returned is specified in the Content-Range entity header field. If the Range request header field is modified, the client will need to send additional requests in order to retrieve the entire entity. If the Content-Range response header field is modified, the received bytes will be interpreted as a different byte range than the original, which will cause the MAC to be invalid, as the response body will be interpreted differently by the client than by the server.

The Accept-Ranges response-header [15, p. 105] is used to advertised to the client what, if any, byte ranges the client can request. Byte ranges may still be requested regardless of whether Accept-Ranges have been received by the client or not. Modifications of this field will lead to performance losses at worst, by introducing extra round trips. The same effect can be accomplished by the adversary by simply dropping packets.

The WWW-Authenticate response header field [15, p. 150] includes a challenge to enable the user to authenticate him- or herself. If this header field is modified, the user will not be able to be authenticated, and the result is the same as stream truncation. If a response contains a WWW-Authenticate header field, the client should respond with a request containing the Authorization header field [15, p. 107]. The Authorization header field is used to authenticate the user. However, since the Authorization field would be sent in cleartext using GlassTube, WWW-Authenticate and Authorization must never be used during a GlassTube session as it could allow an adversary to impersonate the user. Note however, that this does not violate any of GlassTube's guarantees, as the user's session is still has valid integrity.

The Allow header field [15, p. 106] is used to specify which request methods are supported for the requested resource. An attacker can set not allowed methods to be advertised as being allowed, however, they will not be served by the origin server. If the adversary changes allowed request methods to be disallowed, the data stream will be truncated the. However, nothing except for truncation can be accomplished.

The Host header field [15, p. 128] identifies the resource being requested, on a host running web services for multiple domains it controls which site is served. Modifications to the Host header field was discussed in section 4.1.3. Any modifications to this field can only lead to truncation of the data stream.

The From header field [15, p. 128] is used to identify the person responsible for a request, but only for logging purposes and never for authorization. The From header can thus never affect a current session in any way.

The Location response-header [15, p. 135] is used to redirect the client to another resource together with a response code in the range 300-399. If a GlassTube request is responded by a redirect, it will have the same effect as if the original request is modified so that it is delivered to a different service, as the browser will resend the exact same request to another host and/or URL. Signatures will not match on the target machine, and a GlassTube service will thus discard the packet. A service not running GlassTube will respond, but without adding a MAC to the response, which means that the client will discard the message. Redirects must not be used within a GlassTube session, as the client can not successfully follow them. Attacks that modify this header are void, as even benign usage leads to declined messages.

The Pragma header field [15, p. 135] is used solely to supply implementation-specific directives, to any machine along the message path. As only the active client and intended service can construct a correct MAC for a modified packet, modifications by any other party are not interesting. The only standardized value of the Pragma header field is $no-cache$, which will not cause any issues with GlassTube (removing the header may cause truncation of the data stream).

The header fields Accept [15, p. 100], Accept-Charset [15, p. 102], Accept-Encoding [15, p. 102], Accept-Language [15, p. 104]), Expected [15, p. 126] and the TE [15, p. 142], controls in what formats a response is accepted. An adversary can thus try to change the response by changing these headers, if the application reacts to them.

The two conditional request-header fields If-Match [15, p. 129] and If-None-Match [15, p. 132] are used to control whether a request is served, i.e. if a PUT is applied or if a GET is returned. If-Match and If-None-Match match against entity tags according to known entities. Entities are introduced by the ETag [15, p. 126] response-header field.

The Referer field (misspelled in the standard [15, p. 139]) may be used by the application to keep track of from which resource the request-URI was obtained.

The User-Agent [15, p. 145] field is often used to tailor user experience, as different browsers use different rendering engines and in some cases the size of the users screen can be guessed from the user agent, as is the case with smartphones. The Server response header [15, p. 141] informs the client what server software is serving the request.

The header fields Content-Language [15, p. 118], Content-Length [15, p. 119], Content-Location [15, p. 120], Content-Encoding [15, p. 118] and Content-Type [15, p. 124] are used to inform the receiving party about how the entity body is formatted. If these headers are modified by an attacker, a web browser may transform the data, e.g. by only reading the number of bytes specified in the Content-Length header field from the entity. If this occurs the MAC will not match - if the application can interpret the result at all.

The Transfer-Encoding header field [15, p. 143] describes what codings have been applied to the message during transfer. This coding applies to the message, and not to the entity - as is the case with Content-Encoding, and it is thus applied by the web server or web browser, before it reaches the application. If an attacker modifies the Transfer-Encoding header field, MAC will not match on enclosed data, and it will be decoded incorrectly.

The Trailer header field [15, p. 143] describes what header fields are present in the trailer of a chunked message. If this header field is modified by an attacker, it may change how the data reassembled. If it is, the MAC within the data will either not match the data, or will be indistinguishable.

The Upgrade header [15, p. 143] is used to change the application-layer protocol, to e.g. change from HTTP/1.0 to HTTP/1.1, or from HTTP/1.1 to FTP/1.0. A GlassTube service must decline any upgrade requests.