Firm Grip Handshakes: a Tool for Bidirectional Vouching

Omer Berkman* Benny Pinkas† Moti Yung‡

Abstract

Clients trust servers over the Internet due to their trust in digital signatures of certification authorities (CAs) which comprise the Internet's trust infrastructure. Based on the recent DigiNotar attack and other attacks on CAs, we formulate here a very strong attack denoted "Certificate in The Middle" (CiTM) and propose a mitigation for this attack. The solution is embedded in a handshake protocol and makes it more robust: It adds to the usual aspect of "CA vouching" a client side vouching for the server "continuity of service," thus, allowing clients and server to detect past and future breaches of the trust infrastructure. We had simplicity, flexibility, and scalability in mind, solving the problem within the context of the protocol (with the underlying goal of embedding the solution in the TLS layer) with minor field changes, minimal cryptographic additions, no interaction with other protocol layers, and no added trusted parties.

1 Introduction

August 2011 presented a wakeup call for Internet security, when it was revealed that an unidentified attacker hacked into the computers of DigiNotar, a Dutch CA, and issued a fraudulent certificate for, among others, google.com. This certificate was subsequently used in a Man-in-the-Middle attack deployed against users in Iran. The certificate fooled browsers into assuming that they were encrypting data with a public key assigned to Google, where in fact they were using a public key chosen by the attacker. As a result, users' data, including their Gmail credentials such as their passwords and cookies, was revealed to the attacker.

The attack was possible since "trust" is assumed to operate "top down". Namely, the normal way for browsers to operate, based on the SSL/TLS protocol, is to trust any assertion made by a certificate authority (CA) that is trusted by the browser. In most cases this means that any CA whose key is preinstalled with the browser is trusted. The attack was identified since the Google certificate was "pinned" in the Chrome browser. Namely, its value was hardcoded in the browser, and the browser was not willing to accept certificates for Google (the browser's manufacturer) except from a very small number of CAs [10] (incidentally, this technology was deployed a mere two months before the attack). As a result, a suspicious user noticed the error and posted the rogue certificate on the web, and subsequently the certificate was identified as a fake one.

*Academic College of Tel Aviv Yaffo. Most of this work was done while visiting Google.

†Bar Ilan University. Most of this work was done while visiting Google. Partly supported by the SFEROT project funded by the European Research Council.

‡Google Inc. and Columbia University.

In the days that followed, it was revealed that the attackers issued hundreds of certificates for different high-importance web services, such as Google, Yahoo, Skype, Facebook, Microsoft Windows Update services and the anonymous communication system Tor, as well as some intelligence agencies. It was also revealed that DigiNotar knew of the breach more than a month before the rogue certificate became public, but has not notified anyone about it. Consequently, the root certificates of DigiNotar were revoked from all major browsers and the company went out of business [19, 1].

The goal of this work is to mitigate attacks like the one described here, by fundamentally changing the notion of trust over the web. This is done by adding a "bottom-up" component allowing clients to vouch for the server's trustworthiness: following the first server-client handshake, the client has a cross-session "firm grip" on the server. The goal is to add this property with relatively small efforts and modifications, and yet to achieve a much more robust authentication for the typical client-server web interactions (i.e., via a modified SSL/TLS protocol).

1.1 The Current Trust Infrastructure and Our Goals

The web's trust infrastructure based on its early days' need to bootstrap trust in an essentially unlimited number of sites from an initial trust in a limited number of CAs. The trust model is built top-down (reflecting on the CA infrastructure and X.509 standards). The core idea is that each browser ships with the public keys of a limited number of root CAs, and subsequently trusts a public key presented by a web site if it is accompanied by a certificate chain leading to the public key from one of the root CAs. Over the years, the number of CAs trusted by browsers became very large: more that 650 organizations, located in 52 countries, were identified as valid CAs for Mozilla or Microsoft browsers [7, 5]. It is likely that some of these CAs suffer from security vulnerabilities or are not managed properly. Indeed, earlier in 2011 an attacker obtained bogus certificates from Comodo, a major CA, but no actual attack using these certificate was identified in the wild [20].

The threat that the current infrastructure poses to users is very serious. It is sufficient for an attacker to forge certificates from a single "trusted CA" in order to apply Meet-in-the-Middle (MiTM) attacks to all web sites in all jurisdictions (with the help of phishing, malware or DNS contamination). This situation, allowing a very local exposure to potentially affect the web globally, is a colossal failure of the trust infrastructure from a risk managemnt perspective. [1]

Toward our goals, let us define the attack we aim to prevent, which we denote as the "certificate in the middle" (CiTM) attack. It is modeled after the DigiNotar incident, and is stronger than a man in the middle (MiTM) attack since it assumes that the attacker can both mount an active attack (e.g., by controlling the Domain Name System) *and* forge arbitrary valid certificate chains.

Definition 1.1 (CiTM attack)**.** *A "certificate in the middle" (CiTM) attack is an attack by an adversary with the following capabilities:*

- *It can eavesdrop on any communication channel (in particular, to communication between the client and server).*
- *It can change messages that are sent on any communication channel.*

[1] Indeed, it is very hard to come up with a set of policies defining a "normal" usage of CAs and certificates, and any such policy will result in many false warnings. We do not suggest to use such a policy, but rather use this situation to exemplify that the current trust model is very flawed and must be fixed.

- *It can generate arbitrary valid certificate chains. In particular, the adversary can generate a certificate which states that a certain public key (for which the adversary knows the corresponding private key) is the public key of a different party.*

Being polynomial-time bounded and deprived of access to secure private memories, the adversary cannot*, however, learn the private keys of specific parties. In particular it does not know the private keys of the server which it tries to attack.*

The current web trust infrastructure is completely vulnerable to CiTM attacks since its top-down approach assumes that all CAs are trusted, whereas the CiTM attacker is able to issue certificates in the name of trusted CAs.

It is easy to verify that due to the strong capabilities of the adversary, one cannot protect unfortunate users who have all their communication channels permanently controlled by a CiTM attacker. These users will always receive the same certificates forged by the attacker, whereas the server will always observe messages that will comply with whatever policy the server might have.

Have we formulated above an attack we cannot prevent? Are we at a loss here since we formulated such a strong adversary? We claim that, nevertheless, there is hope for protecting users very efficiently. This is so since not all users will be permanently under the control of the attacker. Some clients will use uncompromised connections before they are subject to a CiTM attack, whereas others might first be subject to the attack and then be able to connect to the server through a legitimate channel (say, when they travel outside of the affected country or if the attacker's infrastructure fails for a short period of time). Although not all users might fall into these categories, it is important to note that, as with the DigiNotar breach, even a single educated user who notifies the world about the attack is likely to lead to a complete revocation of the relevant CA. Furthermore, deploying a CiTM attack requires compromising a CA, and therefore even the most determined attackers cannot deploy the attack too many times. We thus argue that even alerting a limited number of users to the existence of the attack can severely diminish its utility to attackers.

1.2 Contributions

We propose a concept and a framework for handshake protocols, that extends an initial handshake and minimizes the "window of opportunity for an attack" that can be employed by the adversary. This type of limitation is a principle in designing robust and secure systems. Specifically, our framework uses a form of "chaining" between all protocol handshakes that are performed between a specific client-server pair (hence we call it a "firm grip handshake"). As a result, if a client establishes an uncompromised handshake before it is subject to a CiTM attack then the client identifies the attack at the moment that it is attempted due to lack of chaining. Alternatively, even if all client-server handshakes were subject to an attack, it is sufficient to have a single uncompromised handshake in the future in order to inform the client about the attack by breaking the chaining to the compromised certificate.

Regarding server side security, we cannot guarantee that the server identifies an attack: due to the stateless nature of web servers, the attacker can modify all communication from the client to look as if each interaction is the first interaction between the client and server, and therefore no chaining with a previous handshake exists. However, by forcing attackers to resort to such behavior we enable servers to identify attacks by examining connection statistics and attempting to use, say, off-line anomaly detection techniques on the server's logs; (for example, if an exceedingly large

proportion of connections from a certain location seem to be coming from new clients, e.g., new web browsers, then further examination of this phenomenon might be needed).

To highlight the new concepts and avoid implementation variations, we describe a high-level conceptual version of the protocol, rather than the specifics of embedding the protocol in existing TLS implementations (although we do discuss issues relevant to such an implementation). Our protocol is based on two main ideas:

- Extending the initial client-server handshake by having the server choose a key that will be used, together with the initial certificate chain, in order to MAC all future client-server handshakes. (The MAC value can be considered part of the reconstructable state and can be further hashed and MACed in existing fields, say in TLS).
- In order to improve scalability and retain the stateless-ness of web servers, the protocol stores the MAC key in the client side in encrypted and authenticated "sandwich cookies", or oreos, rather than storing it at the server.

1.2.1 Desired properties

We suggest the following protocol properties:

- Security requirements
  - *Client identifying future attacks.* If the initial client-server handshake is not affected by a CiTM attacker, then any future CiTM attack is identified by the client.
  - *Client identifying past attacks.* If the initial client-server handshake is compromised by a successful CiTM attack, then once the client performs a handshake with the server over an uncompromised channel, i.e., a channel whose contents cannot be changed by the attacker, the client identifies that an attack has previously occurred.
  - *Server identifying irregular behavior.* Any active attack will result in either the server identifying the attack, or it identifying a usage pattern which is different than normal.
- *Simplicity and flexibility.* Changes required for preventing CiTM attacks must be easily integrateable with existing protocols. The changes to the existing implementations of the protocol must be minimal.

  Changes should be applied to only one layer of the communication stack (e.g., TLS) rather than several layers (e.g. TLS and the application which uses it). Ideally, they can be implemented without changing existing standards (e.g., consist of a few added fields in existing messages and data structures). No infrastructure changes should be required (e.g., no additional trusted parties are added).
- *Scalability.* The server need not keep a long-lived state for each client.
- *Efficiency.* The protocol requires only a few added crypto operations, preferably symmetric key operations.

The protocol does require the server to store a single short long-lived state. In Section 3.1 we describe how to support recovery from the server erasing this state, or from having this state being compromised.

1.3 Related Work

There have been several recent proposals for moving away from the current complete trust in certificate authorities, and they all deserve credit for worthy efforts to solve this problem. One such approach which is already deployed is the pinning of certificates in Chrome for the google.com domain [10], which proved to be successful in the DigiNotar attack. This approach, however, cannot scale since it requires to encode in the browser an entry for each supported domain.

In the origin-bound certificates [4] solution, the client browser generates a self-signed client certificate in its initial handshake with the server, and passes it to the server in all future handshakes. The server can then embed that certificate in the authentication processes, for example by storing it in a cookie in the application layer, and use it for authentication. This project has done remarkable work in implementing this solution as a TLS extension in the Chrome browser and in Google's web serving infrastructure. There are, however, some major differences between our solution and this project: Our protocol can be implemented in the TLS layer alone, and does not require integration with cookies served by the application layer. In addition, it identifies CiTM attacks even if they occur before the first uncompromised interaction between the client and the legitimate server, whereas such attacks against the origin-bound certificate solution remain undetected. A solution implemented in [2] is similar to origin-bound certificates but employs client's passwords available from any browser. It allows web applications to use secrets that they share with clients, in order to attest for the authenticity of their certificates.

The TACK Internet draft [14] describes a TLS extension that enables a server to assert the authenticity of its public key by signing it with a server "TACK key". TLS connections to a pinned hostname require the server to present a TACK containing a pinned TACK key and a corresponding signature of the TLS server's public key. Unlike our solution, TACK enables only the client, and not the server, to identify CiTM attacks. Another difference stems from the way TACK implements its goal of limiting the damage from transient attacks on servers. Unlike our approach (see Section 3.1), TACK tries to minimize the "window of attack opportunity" by using time. Specifically, it limits the duration in which a TACK key is pinned in the client to be the minimum between 30 days and the length of time in which this TACK key has been observed by the client. As a result, a user who, for example, connects to his bank quite rarely, say once every 31 days, is not offered any protection by TACK.

Another set of solutions is based on using third-party services for extending the current trust infrastructure. For example, it is suggested in [13] to form a public auditable repository of every publicly visible certificate. Each certificate issued must be accompanied by an audit proof, and servers must send these proofs along with the certificates to browsers, which will then check them. Unlike our suggestion, this solution adds a third party to the infrastructure. In addition, domain owners must regularly monitor the public logs to ensure that no rogue certificates were issued for their domain. This is a commendable project that will provide complete transparency of certificate usage. Yet, this project is somewhat orthogonal to our local protocol extension approach, and requires considerable global resources in order to be implemented and maintained. A project with similar goals is EFF's Sovereign Keys [6]. Another suggestion along these lines is to use DNSSEC to bound certificates to domain names [16]. Another third-party approach is to use a notary service, i.e. to ask a third-party observer whether it observes the same certificate as the client. This approach was taken in [18, 15, 11].

2 The Protocol

We give a high-level conceptual version of the protocol, which can be applied to any handshake protocol (where embedding it in TLS and other certificate-based protocols is a major goal). All that we assume is that the initial message in the protocol is sent from the client to the server, and that it is possible to add fields to protocol messages. This latter assumption can be justified by instantiating these additional fields in several ways, as we discuss in Section 2.1.

The protocol is based on two main ideas. The first idea is that in the initial handshake between the client and server, the server chooses a fresh key, denoted as the "cream" for reasons that will shortly become clear, and sends it to the client. (This key can be sent encrypted or in the clear; see discussion in Section 2.1.) Later, each handshake is accompanied with MACs, keyed with this key, of the messages seen by the parties in the handshake, as well as with hashes of the initial certificate chain that each of the parties had received. Therefore, an attacker who does not know the cream key will not be able to mount an active attack on future handshakes, even if it can issue a certificate of the server. Also, if the client receives an initial certificate chain different than the one sent by the server, then their hash values will be different and this fact will be identified in the first uncompromised communication.

The second idea is that the server need not store a state containing the cream key. Rather, it generates a sandwich cookie, or oreo, which contains an encrypted and authenticated envelope over the cream key and over a hash of the initial certificate chain. The oreo is stored at the client side and is sent by the client to the server in future handshakes. The server can then decrypt the oreo, verify its authenticity and use the resulting key for generating MACs and verifying MACs received from the client. We note that given the stateless nature of web servers the idea of keeping state at users has been suggested before (perhaps for the first time in [12]), and is typically used for providing the server with encrypted keys/states in cookies, for the server to restore keys and common state. We describe the protocol below. Its main steps are also depicted in Figure 1.

The Basic Protocol

Long lived states: The server stores a long-lived server key, $\boldsymbol{sk}$. This is a symmetric key and is the only state that must be stored by the server. The client stores a "cream file", which is defined in the protocol below.

The protocol:

When initiating a connection to the server the client sends an additional bit, the $\boldsymbol{sbit}$, which states whether the client already stores a state for this server.
If $sbit = 0$ (no state) then

1. The server picks a random key, denoted as the "cream key", or $\boldsymbol{ck}$. This key is a symmetric key chosen using fresh randomness.

2. The server computes a hash of the certificate chain that it sends to the client. Denote this value as the server hash, $\boldsymbol{sh} = H(\text{certificate chain sent})$, where $H()$ is a collision intractable hash function, for example a function from the SHA family.

3. The server uses the long-lived server key sk to generate an encrypted and authenticated version of the cream key ck and of the server hash sh. The result is denoted as the "sandwich cookie", or **oreo** (which contains a cream filling in it).

4. During the handshake, or immediately afterwards, the server sends the key ck and the oreo to the client. (See discussion in Section 2.1 on how these values can be sent, and whether they should be encrypted or not.)

5. The client receives the cream key ck. It also computes a hash of the certificate chain that it received. Namely, computes a client hash $\boldsymbol{ch} = H(\text{certificate-chain received})$ using the same function $H()$ as the server. The values ck and ch will never be sent in the clear by the client.

   The client stores a state for the server in an entry which includes the cream key ck, the client hash ch and the oreo, and is stored in a special cream file. If all went well, the oreo contains a server hash sh that is identical to the client hash ch. (See discussion in Section 2.1 on storing this cream file.)

If $sbit = 1$ (namely, the client connects to a server for which it already has an entry in its cream file) then

1. The client sends the oreo to the server. This is done during the handshake or immediately afterwards. (See discussion in Section 2.1 on how this data can be sent.)

2. At the end of the handshake the client sends to the server a MAC, keyed with the cream key ck, of the concatenation of the string "client view" to all messages seen by the client in the handshake (messages both sent and received by the client). It also sends ch, the hash of the certificate chain received by the client in the initial handshake. (See discussion in Section 2.1 on why this value is not included in the MAC.) We assume that the "view" (of client/ server) is different in each interaction, and thus serves as a mechanism that prevents "replay attacks."

3. The server decrypts the oreo and learns the cream key ck and the server hash sh of the certificate chain sent by the server in the initial handshake. If the check of the authenticity of the oreo fails then the server aborts and notifies its operator.

   Otherwise, the server uses ck to check the MAC received from the client (namely, compute a MAC keyed by ck of the string "client view" concatenated to the messages seen by the server, and compare it to the MAC received from the client). It also compares the received ch value to its own sh value. If any of these checks fails then the server aborts and notifies its operator.

4. The server sends to the client a MAC, keyed with the cream key ck, of the string "server view" containing the concatenation of all messages seen by the server in the handshake. It also sends to the client the server hash sh. The client checks the MAC using the cream key ck and compares the received sh value to the ch value (recall that ck and ch are stored in the client's cream file). If any of these checks fails then the client aborts and notifies the user. (In a way similar to the message that is presented when certificate pinning is used and the wrong certificate is received.)

Note that if the client has an entry for the server in its oreo file, then the client always expects to receive a MAC at the end of the protocol, assuming (for now) that the MAC key is always available at the server.

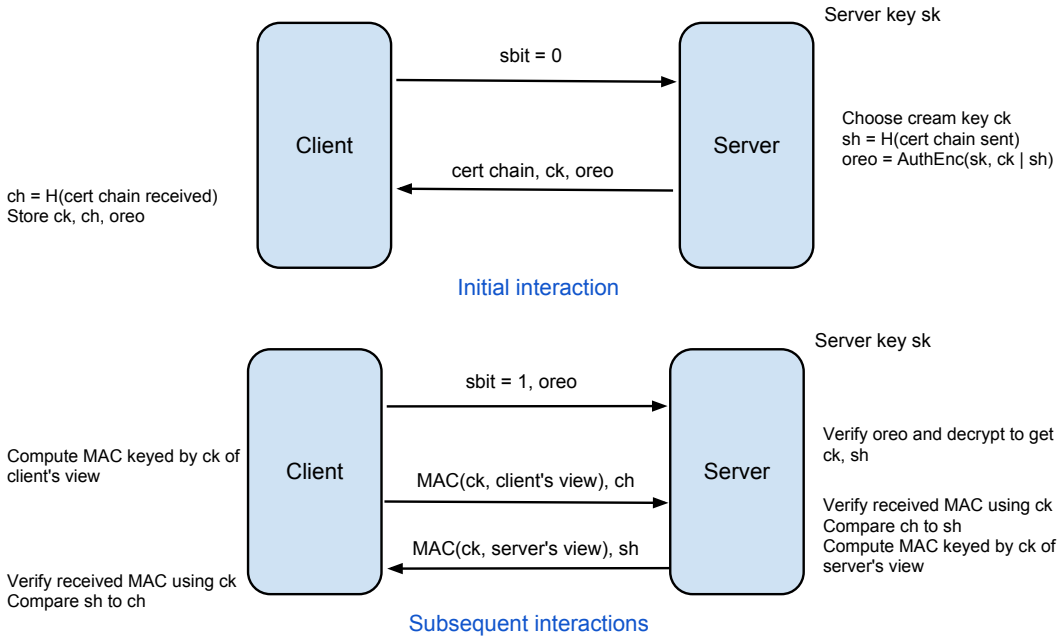

Figure 1: The basic protocol.

2.1 Comments

Sending ck from server to client. In the initial handshake the server chooses a cream key ck and sends it to the client. It is possible to send this value encrypted or unencrypted. The advantage in sending ck encrypted is that this prevents an adversary from learning the value even if the adversary is able to eavesdrop on the initial handshake. The disadvantage in encrypting ck is that encryption can only be performed after the two parties agree on a common key. Therefore an encrypted ck can only be communicated in the last message sent by the server in the handshake protocol (the server-finished message in TLS), or immediately after the handshake. This might complicate the implementation, as well as its integration with the existing protocol and the applications using it.

Note that the only advantage in encrypting ck is against an adversary that at the time of the initial interaction has capabilities which allow it to eavesdrop on the first handshake, but do not enable it to forge a server certificate or mount an active attack. Yet at a later time the adversary has the full capabilities required for mouting a full CiTM attack. The decision about whether to use encryption depends on whether one expects to encounter adversaries with this set of capabilities.

Sending separately the hash of the certificate chain. The last steps of the protocol have each party send to the other party a MAC of its respective view, as well as a hash of the initial certificate chain. A more natural way of implementing this step would have been to send just a single value equal to the MAC of both the view and the initial certificate chain. Namely, have, say, the client, send $\mathrm{MAC}_{ck}(\text{client}'\text{s view} \mid ch)$. This, however, prevents us from proving security

based on standard assumptions[2] and therefore we resulted to sending the hash of the certificate separately.

Storing the cream file. For each server the client must store an entry which contains the cream key *ck*, the client hash *ch*, and the oreo that the client received from the server. The client might store additional information that could be useful for forensic examination in case that a CiTM attack is identified, such as the time of the initial handshake and the certificate chain received in it.

This data can be stored in a special "cream file". This file is similar to a cookie file except for one important difference: the contents of the cream file are never sent outside of the client, except for the oreo data which is sent to the relevant server. Similar to a cookie file, the cream file contains sensitive private information that reveals, for example, which sites were visited by the user. Therefore it must be possible to apply to the cream file the same privacy controls as for cookie files, for example the option of deleting entries for specific sites (at the cost of losing CiTM protection for connections to these sites).

Changing to a new CA. The protocol does not prevent the server from having a new CA sign its certificates (say, because that CA gives the owner of the server a better financial offer). Future interactions with the client will be made using certificates signed by the new CA, but the oreo will not change, and handshakes will be MACed using the original cream key *ck*, and will include the hash of the certificate chain used in the initial client-server interaction.

Using existing TLS modes Following the initial handshake which establishes the cream key *ck*, which then servers as a shared key between client and server, the protocol could be modified in many ways. Two, perhaps interesting, variants are to use *ck* as the pre-shared-key in [8] (using, say, the RSA-PSK key exchange) or as the password in [17].

Correctness. Let us state an easy and yet important claim about the well-functioning of the protocol if no active attack is used.

Claim 2.1. *If no change is made to the messages exchanged between the client and server, then the new protocol provides the same functionality as the original handshake protocol.*

Proof. If no changes are made to the messages then, in particular, the certificate chain sent by the server is identical to the certificate chain received by the client, and therefore the server hash *sh* is equal to the client hash *ch*. In addition, both parties have identical values for the key *ck*, and also the MAC values received are equal to those that are sent. Therefore all checks made by the parties are successful and the original protocol is allowed to run in its entirety. □

2.2 Embedding the Protocol Data in Existing Protocols.

The new protocol requires sending additional fields between the client and server, namely *sbit*, oreo, MAC and *ch* from the client to the server, and *ck*, oreo, MAC and *sh* from the server to the client.

[2]The problem is with the case of an adversary that controls the initial communication with the server. That adversary can send the client an arbitrary MAC key *ck* and has to make sure that MACs computed with this key, of the certificate chain that it sent, are equal to the MACs that the server expects to receive. This seems as a very hard task, which is indeed impossible if we assume the MAC to be computed by a random oracle. However, the security definition of MACs in the standard model assumes that the adversary has no information about the key that is used, and this is not the case with this attack. Therefore we cannot prove the security of this protocol variant in the standard model.

The most straightforward way would have been to change existing protocols, such as TLS, in order to support these new fields. However, changing TLS would be a lengthy process and it is preferable to be able to communicate the new fields without changing the original protocol. This goal is aided by the fact the new fields are not very long: if we assume a symmetric key to be 10-16 bytes long, the output of $H()$ to be 20 bytes long, and a MAC value to be 8-12 bytes long (shorter than a key, since attacking it requires an online attack), then the length of the oreo would be 18-28 bytes, and the length of the hash values ch, sh would be 20 bytes.

We describe here how the additional fields can be embedded over TLS messages. In particular, we describe how superflous certificate and implicit sending of values, can be used to communication the required information with minimal or no changes to the messages that are sent in the protocol.

Client to server communication. With regard to TLS, the oreo and ch value sent by the client can be embedded in either the client-hello or client-finished messages. The MAC sent by the client must be sent in the client-finished message, or after the handshake is over. The client-hello message contains an "extra data" field which can be used for sending arbitrarily long data [3]. In addition, the client-random field of the client-hello message contains a 28 byte long random string, part of which can be used to send data. The client-finished message contains a client-certificate field, which can be used in order to send a certificate that contains the MAC and the oreo (and which does not need not be signed by a CA since it will be otherwise ignored by the server).

Superflous certificates. In the first handshake the server needs to send ck and the oreo to the client in the server-hello or server-finished message. A natural solution is to embed these values in a new field of a certificate sent by the server. An unfortunate disadvantage of this approach is that CAs are known to charge significant amounts of money for adding new fields to certificates. A workaround is to use "superflous certificates", based on an undocumented feature of TLS.[3] This method works in the following way. The server server-hello message sent by the server contains a set of certificates. Ideally these certificates should form a certificate chain to a CA trusted by the client, but it is often the case that the server sends a set of certificates, only some of which form this chain. (This happens, for example, if instead of removing old certificates the server just adds new certificates to the set that it is sending to clients.) As a result, all major browsers accept server-hello messages in which only a subset of the certificates sent by the server form a chain, and the other certificates are superflous. Therefore, if in our protocol the server wishes to send additional information to the client, it can encode this information in a new certificate and add it to the certificate set sent to the client. It is not required to have this certificate signed by any CA since the client will identify a different valid certificate chain leading to a trusted CA. Note that the entire server-hello message is authenticated in the MAC sent in the server-finished message, keyed by the key agreed upon in the handshake protocol (here we refer the MAC sent as part of the TLS protocol, not the MAC in our protocol). Therefore an attacker cannot change this set of certificates or add new ones. In effect, the result is that by using superflous certificates, servers can add arbitrary authenticated information to TLS handshakes.

Using implicit MACs and hash values. The last steps of the protocol have each of the parties send to the other party a MAC of its view and a hash of the initial certificate. The other party checks these messages by computing its own version of these values and comparing it to the the

[3] See references to this method in `http://www.ietf.org/mail-archive/web/tls/current/msg08820.html`, `http://tools.ietf.org/agenda/81/slides/tls-2.pdf`, and `http://code.google.com/p/certificate-transparency/source/browse/src/client/ct.cc?r=103ff6cd41788fb51d37c3362632f639759ef4a7`.

values that it receives. Therefore, there is no need to send these values but only to make sure that both parties agree on them. Note also that TLS already requires each party to send, as its last message, a hash of all the messages it sent and received in the handshake protocol. These messages are denoted in TLS as the client-finished/server-finished messages, respectively.

We can therefore make the following change to the protocol: the client (and similarly the server) does not send the MAC and hash as required by our protocol but instead computes the last hash in the client-finished message as if it has sent these values. The server, which knows which values it expects to receive as the MAC and hash, uses these values to verify the client-finished message. The effective result is that each party can verify that the other party could have sent the correct MAC and hash values, but this is accomplished without sending any value expect for normal TLS fields.

3 Security and Extensions

We show that the client can identify CiTM attacks as long as it runs a single uncompromised handshake with the server. We also claim that although the server might not be explicitly warned about these attacks, it is likely to have sufficient information to implicitly suspect the presence of the attacks.

Intuitively, if the initial handshake is uncompromised then the client and server share the cream key *ck* which is unknown to any attacker, and which is used for MACing future handshakes. Therefore no active attack can be applied to a future handshake. If, on the hand, an attacker mounts a successful CiTM attack on the initial handshake, and since we assume that the attacker does not know the secret keys of the server, then the attacker must have used a certificate chain different than any certificate chain used by the server. As a result, whenever the client performs a handshake with the original server they do not agree on the hash values, and the client informs its user about this discrepancy.

A note about the cream key ck. Our analysis here assumes that the cream key *ck* is sent encrypted in the initial client-server handshake. As is discussed in Section 2.1, this value can be sent either encrypted or unencrypted. The same analysis holds even if *ck* is sent unencrypted, as long as we assume that the attacker does not eavesdrop on that initial handshake.

The claims about security are based on assuming that some connections are made over "uncompromised channels", which we now define.

Definition 3.1 (Uncompromised channel)**.** *A channel is* uncompromised *if an adversary can eavesdrop on communication carried out over the channel but cannot change it. (Namely, the adversary is only a passive eavesdropper.)*

We first state and prove two claims about the client identifying CiTM attacks if it has an uncompromised connection with the server.

Claim 3.1. *Assuming that the cryptographic primitives used in the protocol are secure, then if the initial client-server handshake is uncompromised, the client will identify any future CiTM attack.*

Proof. (Sketch) Since the initial handshake was uncompromised, the client received in it certificate chain and an oreo identical to the ones sent to it by the server, and therefore the client hash *ch* is equal to the server hash *sh*, and the oreo contains valid encrypted and authenticated values of *ck* and of *sh*.

Consider what happens before the first active attack attempt by the adversary. The adversary might have eavesdropped on the initial handshake and learned the certificate chain as well as an encryption of *ck* and the encrypted and authenticated oreo. (These encryptions are done using keys which are indistinguishable from random by the attacker.) The adversary might have also eavesdropped on subsequent handshakes which contained copies of the same oreo, and MACs keyed by *ck*. The adversary might have eavesdropped on communications of other clients, but these used key values which were independent of those used by the attacked client, except for the oreos which are all encrypted by the same server key *sk*. Standard cryptographic arguments can therefore show that if the adversary can learn anything about the cream key *ck* then either the encryption functions or the MAC are insecure. Since we assume these to be secure we conclude that the adversary cannot distinguish *ck* from a random string.

Now, consider the first handshake in which the adversary aims to change any of the messages. The adversary must send to the client a MAC of the new transcript of messages sent and received by the client, keyed by *ck*. Since part of these messages contain randomness chosen by the client, it holds with overwhelming probability that the transcript of messages in the current handshake is not identical to any of the transcripts of previous handshakes. Therefore the adversary must forge a MAC with a key it has no information about. A secure MAC algorithm ensures the adversary's failure, except with negligible probability. □

As for CiTM attacks on the initial handshake, note that we need only consider attacks in which the attacker sends a certificate chain different than the certificate chain sent by the server to the client, since using a certificate chain of the original server requires the attacker to complete the initial handshake with a MAC based on a key encrypted with the public key of the server, which the attacker cannot decrypt. We are now ready to state our second claim.

Claim 3.2. *If the initial handshake is compromised by a CiTM attack, and the attacker sends in it a certificate chain that is different than the one used by the server, then the client will identify the attack when it first connects to the server through an uncompromised channel.*

Proof. (Sketch) The client receives an oreo from the attacker in the initial handshake. Then, in the client's first handshake with the real server it sends it this same oreo. The server checks the authenticity of the oreo, and therefore aborts the protocol unless the oreo was generated by the server itself. (It is possible that the oreo is not rejected, if the attacker sends to a client a valid oreo that a different client received from the same server.) The oreo contains a server hash *sh* of a certificate chain sent by the server. That certificate chain is different than the one received by the client, and as a result *sh* is different than the client hash *ch*, since both are computed by applying a collision intractable hash function to the certificate chain seen by the server and client, respectively. The last step of the handshake requires the server to send *sh* to the client, which then compares it to *ch* and aborts, since the two are different. □

Attack identification by the server. In many cases it is sufficient to alert the client alone to the fact that a CiTM is taking place, since, as in the DigiNotar case, clients can use alternative communication channels to notify the rest of the world about the attack. Still, it is preferable to let the server learn in realtime that an attack is taking place.

The server checks the MAC sent to it by the client at the end of the handshake, and this check rejects any changes that an attacker injected into the messages. There is, however, an easy way for the attacker to prevent the server from identifying the attack, by setting the *sbit* to 0 and

pretending that this handshake is the first handshake made by the client. In this case the server assumes that there is no oreo stored at the client, and it does not expect to receive a MAC at the end of the handshake (and therefore the attacker can drop this MAC from the messages transferred to the server).

This attack works in principle, but it should be possible for the server to use other signals in order to identify that an attack is taking place. Normally, we can assume that clients will set the *sbit* to 0 whenever they first connect to a server (say, when a user switches to a new web browser and connects to a site it frequented in the past). However, this event should not happen too often. The server could therefore gather relevant data and analyze it, perhaps using machine learning techniques, in order to identify suspicious signals. These signals could include, for example, a large percentage of users from the same physical location or country who all seem to be using a new browser. Or a specific user who, in every new connection, seems to be connecting from a new browser. Overall, we expect that large scale or persistant CiTM attacks will be identified by the server as well as by the client.

3.1 Recovery of Server Keys

The basic protocol assumes that the server key sk is always accessible by the server and is never learned by an attacker. The protocol states that once the client stores an entry for a server in its cream file, it always expects to receive MACs from that server. Therefore, if the server loses its server key sk and is unable to decrypt oreos, it will not be able to communicate with the client. Furthermore, if sk becomes known to an adversary then that adversary will be able to compute valid MACs on future handshakes and apply CiTM attacks.

An extension to the protocol must therefore enable the server to recover from losses or compromises of the server key sk. This functionality is supported using a "recovery key" prk. This key is part of a public key-pair of secret/public keys (srk, prk) which are used for signining and for signature verification, respectively. An important property is that the private key srk can be easily kept offline until the time that it is needed (i.e., until the unlikely case that the server key sk is lost or is compromised). The key srk can be stored disconnected from the network, or even be stored in a non-electronic format, in order to minimize the chances of it being compromised.

The protocol is changed so that in the initial handshake the server sends the public key prk to the client, which stores it in its cream file. Afterwards, the protocol continues as usual except for the following change: the client sends an *sbit* equal to 1, but it agrees to receive server answers corresponding to $sbit = 0$ if those answers and the entire handshake are signed with the secret key srk corresponding to the key prk in the server's entry in the cream file. Namely, the client accepts a change of the handshake by the server to an initial client-server handshake, as long as this change is signed by srk. Note that this feature can be used for periodic update of keys, as well as for recovery from key compromise.

Acknowledgments: We would like to thank Úlfar Erlingsson, Adam Langley and Cem Paya for valuable discussions.

References

[1] Arthur, C.: Rogue web certificate could have been used to attack iran dissidents. http://www.guardian.co.uk/technology/2011/aug/30/faked-web-certificate-iran-dissidents (Aug 2011)

[2] Dacosta, I., Ahamad, M., Traynor, P.: Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties. In: Foresti et al. [9], pp. 199–216

[3] Dierks, T., Allen, C.: The TLS protocol version 1.0. RFC-2246 (1999)

[4] Dietz, M., Czeskis, A., Balfanz, D., Wallach, D.S.: Origin-bound certificates: a fresh approach to strong client authentication for the web. In: USENIX Security. Berkeley, CA, USA (2012)

[5] Eckersley, P., Burns, J.: An observatory for the SSLiverse. https://www.eff.org/files/DefconSSLiverse.pdf (2010)

[6] EFF: The Sovereign Keys project, `https://www.eff.org/sovereign-keys`

[7] EFF: The EFF SSL observastory. https://www.eff.org/observatory (2010)

[8] Eronen, P., Tschofenig, H.: Pre-shared key ciphersuites for transport layer security (TLS). RFC-4279 (2005)

[9] Foresti, S., Yung, M., Martinelli, F. (eds.): ESORICS 2012, Pisa, Italy., LNCS, vol. 7459. Springer (2012)

[10] Google: New chromium security features. http://blog.chromium.org/2011/06/new-chromium-security-features-june.html (Jun 2011)

[11] Holz, R., Riedmaier, T., Kammenhuber, N., Carle, G.: X.509 forensics: Detecting and localising the SSL/TLS men-in-the-middle. In: Foresti et al. [9], pp. 217–234

[12] Janson, P., Tsudik, G., Yung, M.: Scalability and flexibility in authentication services: The kryptoknight approach. Annual Joint Conference of the IEEE Computer and Communications Societies (1997)

[13] Laurie, B., Langley, A.: Certificate authority transparency and auditability. http://www.links.org/files/CertificateAuthorityTransparencyandAuditability.pdf (2011)

[14] Marlinspike, M., Perrin, T.: Trust assertions for certificate keys. draft-perrin-tls-tack-00.txt (2012)

[15] Marlinspike, M.: Convergence, `http://convergence.io`

[16] Osterweil, E., Kaliski, B., Larson, M., McPherson, D.: Reducing the X.509 attack surface with DNSSEC's DANE. In: SATIN: Securing and Trusting Internet Names (March 2012)

[17] Taylor, D., Wu, T., Mavrogiannopoulos, N., Perrin, T.: Using the secure remote password (SRP) protocol for TLS authentication. RFC-5054 (2007)

[18] Wendlandt, D., Andersen, D.G., Perrig, A.: *Perspectives:* improving SSH-style host authentication with multi-path probing. In: Isaacs, R., Zhou, Y. (eds.) USENIX Annual Technical Conference. pp. 321–334. USENIX Association (2008)

[19] Wikipedia: DigiNotar — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/DigiNotar (2012)

[20] Zetter, K.: Hack obtains 9 bogus certificates for prominent websites; traced to Iran. http://www.wired.com/threatlevel/2011/03/comodo-compromise/ (2011)