On the Effective Prevention of TLS Man-In-The-Middle Attacks in Web Applications

Nikolaos Karapanos, Srdjan Capkun
Department of Computer Science, ETH Zurich
{*firstname.lastname*}*@inf.ethz.ch*

Abstract

In this paper we consider TLS MITM attacks in the context of web applications, where the attacker's goal is to impersonate the user to the legitimate server, and thus gain access to the user's online account. We describe in detail why the recently proposed TLS Channel ID-based client authentication, as well as client web authentication in general, cannot fully prevent such attacks.

We then leverage TLS Channel ID-based authentication and combine it with the concept of sender invariance to create a novel mechanism that we call SISCA: Server Invariance with Strong Client Authentication. SISCA resists user impersonation via TLS MITM attacks even if the attacker has obtained the private key of the legitimate server. We analyze our proposal and show how it can be integrated in today's web infrastructure.

1 Introduction

Web applications increasingly employ the TLS protocol to secure HTTP communication (i.e., HTTP over TLS, or HTTPS) between a user's browser and the web server. TLS enables users to securely access and interact with their online accounts, and protects, among other things, common user authentication credentials, such as passwords and HTTP cookies. Such credentials are considered weak, in the sense that they are transmitted over the network and are susceptible to theft unless protected by a secure connection.

Nevertheless, during TLS connection establishment, it is essential that the server's authenticity is verified. If an attacker successfully impersonates the server to the user, she is then able to steal the user's credentials and subsequently use them to impersonate the user to the legitimate server, thus gaining access to the user's account. This attack is known as TLS Man-In-The-Middle (MITM).

TLS server authentication is commonly achieved through the use of X.509 server certificates. A server certificate binds a public key to the identity of a server, designating that this server holds the corresponding private key. The browser accepts a certificate if it bears the signature of any Certificate Authority (CA) that is trusted by the browser. Nowadays, browsers are typically configured to trust hundreds of CAs.

An attacker can thus successfully impersonate a legitimate server to the browser by presenting a valid certificate for that server, as long as she has access to the corresponding private key. In previous years, quite a few incidents involving mis-issued certificates [1, 9, 51, 52, 54, 57] were made public.

In order to thwart such attacks, various proposals have emerged. Some proposals focus on enhancing the certificate authentication model. Their objective is to prevent an attacker possessing a mis-issued, yet valid certificate, from impersonating the server (e.g., [16, 37, 56]).

Other proposals focus on strengthening client authentication. Their goal is to prevent user credential theft or render it useless, even if the attacker manages to successfully impersonate the server to the user. One such prominent proposal is TLS Channel ID-based client authentication, introduced in 2012. TLS Channel IDs [4] are already experimentally supported in Google Chrome and are planned to be used in the second factor authentication standard U2F, proposed by the FIDO alliance [20, 21].

In this paper we show that TLS Channel ID-based approaches, as well as other web authentication approaches that focus solely on client authentication are vulnerable to an attack that we call *Man-In-The-Middle Script-In-The-Browser (MITM-SITB)* attack. This attack bypasses the protection offered by TLS Channel IDs by shipping malicious JavaScript to the user's browser within a TLS session with the attacker, and then using this JavaScript in a subsequent session to access the user's account on the legitimate server. Our attack is related to the Dynamic Pharming attack [34] that was introduced prior to Channel ID-based solutions; we discuss this further in Section 5. We also validate our attack through a proof of

concept implementation.

We therefore argue that effective TLS MITM attack mitigation in the context of web applications requires both client and server authentication[1]. Building on this observation, we propose a solution called *SISCA: Server Invariance with Strong Client Authentication*, that combines TLS Channel ID-based client authentication and the concept of sender invariance [14] to effectively defend against MITM-SITB, as well as against conventional MITM attacks.

In addition to addressing MITM attacks that are based on mis-issued certificates, our solution is also robust against MITM attacks where the attacker holds a private key of the legitimate server; this attack has so-far not been addressed in prior work.

In summary, in this work we analyze TLS MITM attacks whose goal is user impersonation and make the following contributions. (i) We show, by launching a MITM-SITB attack, that TLS Channel ID-based client authentication solutions do not fully prevent MITM attacks. (ii) We further argue that effective prevention of MITM-based impersonation attacks requires strong user authentication and (at least) server invariance. (iii) We propose a novel solution that prevents MITM-based user impersonation under a stronger attacker than previously considered. (iv) We implement a basic prototype of our solution.

The rest of the paper is organized as follows. Section 2 discusses TLS Channel ID-based protocols and shows how MITM attacks are possible using MITM-SITB. Section 3 discusses known solutions for addressing MITM attacks and introduces SISCA. Section 4 discusses how SISCA can be integrated with other web technologies. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2 Analysis: TLS Channel IDs and MITM Attacks

2.1 Attacker Model

The goal of the attacker in a MITM attack is usually to impersonate the user (victim) to the legitimate server and gain access to his online account. Alternatively, the attacker could aim to impersonate the server such that she serves the victim with fake content, or attempts to elicit information through social engineering. In this paper, we focus on the former scenario, where the adversary aims to access the victim's account on a target web server, such as a social networking or e-banking website. We consider a strong adversary, who is able to position herself suitably on the network and perform a TLS MITM attack between the user and the target web server. We distinguish between two different types of MITM[2] attackers; we show later why this distinction is important and which implications different MITM attackers have on the security of authentication solutions.

The *MITM+certificate* attacker holds a valid certificate for the domain of the target web server, binding the identity of the server to the public key, of which she holds the corresponding private key. The attacker, however, has no access to the private key of the target web server. This, for example, can happen if the attacker compromises a CA or is able to force a CA issue such a certificate. Such attacks have been reported in the recent years [1, 9, 51, 57].

The *MITM+key* attacker holds the private key of the legitimate server and can thus impersonate the server during a MITM attack. While we are not aware of public reports of incidents involving compromise of server keys, such attacks can arguably be very stealthy and remain unnoticed. Thus, they are well worth addressing [11, 30, 32, 35].

Even if the attacker does not hold a mis-issued certificated key or the private key of the server, a MITM attack can still succeed if the user is not careful in verifying the security indicators of the browser[3]. Such attacks have the same implications as the MITM+certificate attack and we don't detail them further.

We therefore assume that by performing a MITM attack or by some other means, the attacker is able to obtain the user's weak credentials, namely passwords and HTTP cookies, but is not able to compromise the user's browser or his devices (e.g., mobile phones).

2.2 TLS Channel IDs

TLS Channel IDs is a recent proposal for enhancing client authentication. It is a TLS extension, originally proposed in [13] with the name *Origin-Bound Certificates* (OBCs). A refined version is currently submitted as an IETF Internet-draft [4]. Currently, Channel IDs are experimentally supported by the Google Chrome browser.

In a nutshell, when the browser visits a TLS-enabled web server for the first time, it creates a new private/public key pair (on-the-fly and without any user interaction) and proves possession of the private key, during the TLS handshake. This TLS connection is subsequently identified by the corresponding public key, which is called the Channel ID. Upon subsequent TLS connections to the same web server, or more precisely, to the same web origin, the browser uses the same Channel ID. To protect

[1] We will show that for some types of attacks, server invariance suffices.

[2] We use the terms "TLS MITM" and "MITM" interchangeably.

[3] Throughout this paper, we use the term "browser" to refer to any "user agent" in general.

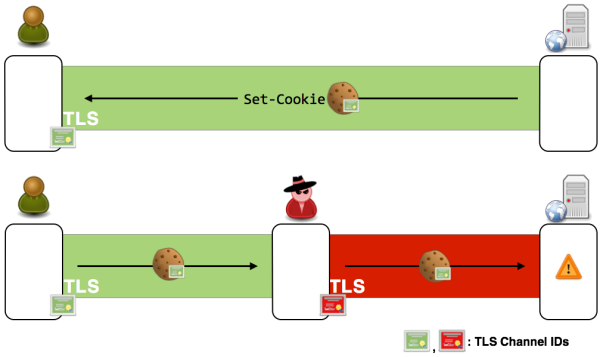


Figure 1: Binding authentication tokens, such as HTTP cookies, to the browser's Channel ID (shown in green). If an attacker manages to steal such a cookie, for example via a MITM attack, she is not able to use it to impersonate the user. This is because the attacker's TLS connection will have a different Channel ID (shown in red) than the connection initially established by the client.

the user privacy, the browser uses a different Channel ID to communicate with different web servers.

Channel IDs are not envisioned to be directly used by the web server to authenticate the user or the browser. They are instead used by the web server to identify the same browser across multiple TLS connections, as the browser will be using the same Channel ID for these connections. The web server can further bind authentication tokens, such as HTTP cookies, to a specific Channel ID, such that the token is considered valid only if it is presented over that particular Channel ID. For example, as proposed in [13], a web server may create a channel-bound cookie as follows: $\langle v, \text{HMAC}(k, v|cid)\rangle$, where v is the original cookie value, cid is client Channel ID and k is a secret key, only known to the web server, used for computing a message authentication code over the concatenation of v and cid.

MITM Prevention. TLS Channel IDs are designed to resist both MITM+certificate and MITM+key attacker types [4, §6] (assuming TLS forward-secret connections for the latter type), due to the channel-binding property described above. An attacker that manages to steal a channel-bound cookie, e.g., through a MITM attack, cannot reuse it to impersonate the user to the web server, since she does not know the private key of the correct Channel ID. Figure 1 illustrates this concept.

2.3 Channel ID-Based Authentication: PhoneAuth and U2F

By *Channel ID-based authentication* we refer to the use of Channel IDs throughout the user authentication process. Specifically, when the user attempts to login to his online account for the first time from a particular browser, the web server requires that the user authenticates using a strong second factor authentication device, such as in PhoneAuth and FIDO U2F frameworks, described below, that leverages Channel IDs to prevent MITM attacks.

After successful initial authentication the server sets a channel-bound cookie to the user's browser. Subsequent user interaction with the server from that particular browser is protected by the channel-bound cookie, such that even if the attacker steals the cookie she cannot use it to impersonate the user (see Section 2.2). At this stage, the second factor device is not required for authenticating the user [10].

Whenever the channel-bound cookie is not present (e.g., it expired, the user deleted it, or the user tries to login from a new browser) or it is present but invalid (i.e, presented over an incorrect Channel ID), the server once again requires user authentication using the second factor device.

PhoneAuth. *PhoneAuth* is a user authentication framework, proposed in [11]. It leverages the user's smartphone in order to provide a second, strong authentication factor, and makes use of TLS Channel IDs in order to detect and prevent MITM attacks.

In brief, PhoneAuth works as follows. The user's mobile phone holds a private/public key pair, which it uses to enroll and generate a shared secret key with the server. After successful enrollment, when the user authenticates to the web server by providing his username and password, the server issues an encrypted and integrity-protected challenge, called *login ticket*, to the browser. The browser is then instructed, via JavaScript API calls, to interact with the user's phone over bluetooth. During this interaction the browser sends to the phone an *assertion request* which, among other things, contains the login ticket.

Upon receiving the assertion request, the phone performs a number of checks and, if they pass, creates an *identity assertion*, by signing the login ticket with its private key. The identity assertion is forwarded back to the web server, which verifies the phone's signature, and signs the user in. The server finally sets a cookie and binds it to the Channel ID of the user's browser (as described in Section 2.2).

U2F. *Universal 2nd Factor* (U2F) [24] is an initiative started by Google that aims to provide strong second factor web authentication. It is very similar to the PhoneAuth protocol, described above. The major difference is that, instead of using the user phone, U2F leverages a dedicated USB dongle. USB is a more ubiquitous interface for connecting the browser and the second factor device. A dedicated dongle can provide higher security assurances, given that mobile phone malware is becoming increasingly common [17, 18, 58], at the cost of

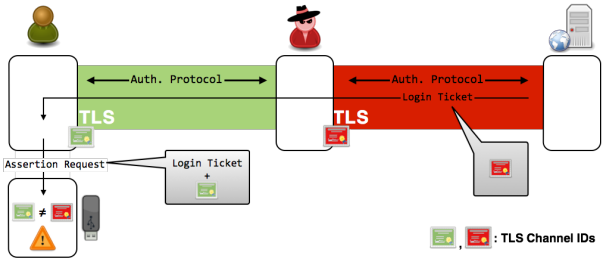


Figure 2: PhoneAuth/U2F. Leveraging TLS Channel IDs for detecting MITM attacks during initial user authentication.

an extra token that the user has to carry.

In the spring of 2013, Google joined *Fast IDentity Online Alliance* (FIDO) [20], which aims at providing an open set of standards for stronger online authentication. Many companies, like Microsoft, PayPal, MasterCard and RSA have also already joined the alliance, a fact that significantly increases the possibility of global adoption of standards and products proposed by FIDO in the near future. One of the proposed standards by FIDO is the Universal 2nd Factor (U2F) protocol. According to the draft specifications [21] the FIDO U2F protocol is similar to the Google U2F protocol and to PhoneAuth. Thus, the results of this paper could also be relevant for the design of FIDO U2F.

MITM Prevention. What is mostly of interest to us in PhoneAuth and U2F, is the approach they follow to address both types of MITM attackers presented in Section 2.1 [11, §3]. Their basic idea, illustrated in Figure 2, is to compare the TLS Channel ID of the user browser to the one seen by the web server.

Specifically, the web server includes the Channel ID that identifies the TLS connection with the browser in the login ticket. In other words, the web server binds the login ticket to that particular Channel ID. Furthermore, the browser adds its own Channel ID in the assertion request, which also contains the login ticket. The assertion request is forwarded to the second factor device, as previously described, and the device performs a number of checks. One of these checks is the comparison of the two Channel IDs contained in the assertion request. If the two Channel IDs are equal, this implies that the user browser is directly connected to the web server through TLS (because they share the same view of the connection), and thus there is no MITM attack taking place. On the other hand, if the two Channel IDs differ, it means that the web server is not directly connected to the user browser. Instead, as shown in Figure 2, there is an attacker in the middle that has established two TLS connections, one with the browser and one with the web server. Upon detecting such a mismatch, the device refuses to generate the identity assertion.

To the best of our knowledge, the notion of comparing the TLS sessions as seen by the client and the server in order to prevent MITM attacks, was originally proposed in the *TLS Session Aware User Authentication* (TLS-SA) scheme [43, 44]. Channel ID-based solutions, are built on top of this idea, but make important improvements in order to be more user-friendly, practical and deployable. In the following sections, we show that Channel ID-Based authentication protocols fail to fully solve the MITM problem, at least when applied to the context of web applications.

2.4 MITM Attack on Channel ID-Based Protocols

We show how the proposed Channel ID-based protocols still allow the attacker to successfully impersonate the user. This is due to the way web applications are run and interact with the servers, which differs from other internet client-server protocols like SMTP or IMAP over TLS. In particular, we make the following key observations:

- Web servers are allowed to push scripting code to the web browser, which the latter executes within the context of the web application (according to the rules defined by the same-origin policy [5]). In fact, client-side scripting, and especially JavaScript, is the foundation of dynamic, rich web applications that vastly improve user experience, and its presence is ubiquitous.
- A web browser is able to establish multiple TLS connections with the same server. In addition, a typical web application loads resources, such as images and scripts, from multiple domains (*cross-origin* network access [5]). Assuming that all communication is TLS-protected, this means that the browser needs to be able to establish TLS connections with multiple servers while loading a web page.

Given the above, there is a conceptually very simple attack that a MITM attacker can perform in order to bypass the security offered by Channel IDs. The attack can be realized by either MITM+certificate or MITM+key attackers. We assume that the user tries to access the target web server, say `www.example.com`. The attacker then proceeds as follows:

1. She intercepts a single TLS connection attempt made by the browser to `www.example.com`, and by presenting a valid server certificate, she successfully impersonates the legitimate server to the browser.
2. Through the established connection, the browser makes an HTTP request to the server. The attacker

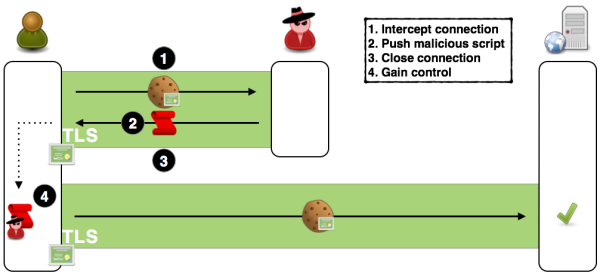


Figure 3: MITM-SITB attack on Channel ID-based authentication. The user has previously logged in on the target web server, `www.example.com`, and subsequent requests are protected with a channel-bound cookie. The attacker ships malicious JavaScript code to the browser, which is executed within the origin of `www.example.com` (shown by the dotted arrow).

replies with an HTTP response, which includes a malicious piece of JavaScript code. This script will execute within the origin of `www.example.com`.

3. The attacker closes the intercepted TLS connection. This forces the browser to initiate a new TLS connection in order to transmit subsequent requests, or use another existing one, if any (this behavior conforms with the HTTP specification [22]). At the same time, the attacker allows subsequent TLS connection attempts to pass through, without interfering with them. As a result, after the attacker closes that single intercepted connection, all other connections, existing and new, are directly established between the browser and the legitimate server.

4. From that point on, the attacker has gained full control over the web application with respect to this particular user and can perform arbitrary malicious requests to the target server, for example through the use of the `XMLHttpRequest` object [3]. Such a request could modify the user's account, or extract sensitive user information. In the latter case, the malicious code can upload the extracted data to an attacker-controlled server. As another example, if the web application is Ajax-based, the attacker can perform *Prototype Hijacking* [47]. This allows her to eavesdrop and modify on-the-fly all the HTTP requests made through `XMLHttpRequest`.

In summary, the MITM attacker manages to "transfer" herself (via the malicious script) within the user's browser, and continue her attack from there. We call this attack *Man-In-The-Middle-Script-In-The-Browser*, or *MITM-SITB* for short.

Figure 3 shows how the attack works in the case when TLS Channel IDs are used, the user has already logged

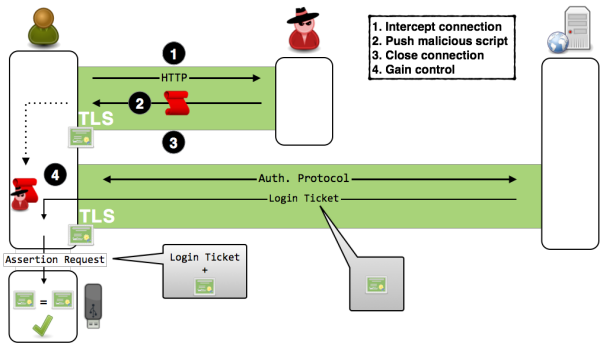


Figure 4: MITM-SITB attack on PhoneAuth/U2F authentication protocols.

in on `www.example.com` in the past, and the server has set a channel-bound cookie in the user's browser. The attacker pushes malicious JavaScript code to the browser by intercepting a TLS connection to `www.example.com`. Once this happens, the attacker terminates the intercepted connection. This forces the browser to establish a new TLS connection for subsequent HTTP requests. This new connection will be established with the legitimate server, i.e., the attacker will not hijack it. This is important, because it ensures that subsequent requests are transmitted to the legitimate server over the correct channel ID. As a result, the attacker's injected script is able to submit arbitrary HTTP requests to `www.example.com`; these requests will be approved by the server since they will carry the channel-bound cookie, which authenticates the user, over the correct Channel ID.

Figure 4 shows a high level description of the MITM attack against PhoneAuth or U2F. Here, we consider the case when the user authenticates to the server for the first time. Like before, in this attack, the attacker intercepts a TLS connection, pushes her code to the user's browser, and terminates the connection. The attacker thus ensures that the browser authenticates to the web server over a direct connection (not through the attacker), but with the attacker's code running in the browser. This way, the view of the TLS channel will be the same for both the browser and the server, and the Channel ID comparison made by the second factor device, will pass successfully.

From the above attack description there are various details that remain unclear. For example, which TLS connection the attacker should intercept, whether to "hit and run" or attempt to persist as much as possible, etc. Depending on the scenario, there are various alternatives, which are mostly implementation decisions. For example, assuming that the attacker wishes to be as stealthy as possible, she can choose the following strategy. She intercepts the *very first* TLS connection, i.e, the one that the browser initiates once it is directed to `www.example.com`. Depending on the situation, the

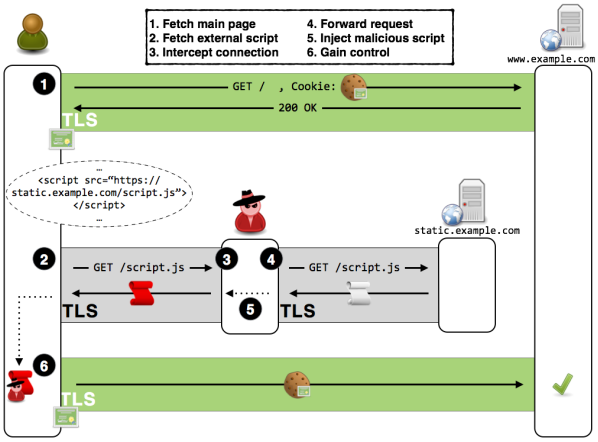


Figure 5: MITM-SITB attack on Channel ID-based authentication, when cross-origin communication is involved. Channel IDs for `static.example.com` are of no use (there is nothing to bind). Hence, they are not shown in the picture (the respective TLS connection is colored in light grey).

attacker's HTTP response could contain the expected HTML document of the website's starting page, together with the appropriately injected malicious script, or it could only contain the malicious script, which will take care of loading the starting page in the browser. Then, as described before, the attacker closes this first connection and subsequent communication (malicious or not) takes place through a direct connection to the legitimate server.

The Cross-Origin Communication Case. Visiting a single web page typically involves cross-origin communication with different domains in the background. Consider, for example, a typical network optimization technique, which is to have the browser load the static resources of the website, such as images, style sheets and scripts, from so-called *cookieless domains* (e.g, Google websites usually load static resources from `gstatic.com` [25]). Those domains, as their name suggests, do not set any cookies, in order to minimize network latency. As a matter of fact, on such domains, the concept of client authentication does not even apply at all, as they are just used to serve static resources, which anyone, including the attacker, can access. Hence in those cases, the attacker can perform a conventional MITM attack against a cookieless domain, and inject its malicious code, at the moment when the target web server requests a legitimate JavaScript file from that domain.

Figure 5 illustrates the attack sequence. The attacker lets all communication to `www.example.com` (the main web server) pass through. Initially, the browser connects to `www.example.com` in order to load some page. The returned HTML document includes a JavaScript file from the cookieless domain `static.example.com`. Assuming that the resource is not cached, or the cached file has expired, the browser initiates a TLS connection to that domain, which is intercepted by the attacker. The attacker fetches the script, and before forwarding it to the browser, she injects her malicious code. This script is then executed within the origin of `www.example.com` and, like before, the attacker gains complete control of the web application.

2.5 Proof of Concept Attack

We validate our attack against TLS Channel IDs through a proof of concept implementation. We use two Apache TLS-enabled servers, one for the attacker and one for the legitimate server, and an interception proxy that can selectively forward TLS connections to either server. The legitimate server uses a patched OpenSSL version that supports Channel IDs and leverages them for creating channel-bound cookies. We use Google Chrome as the user's browser, since it supports Channel IDs, and make sure the it accepts the certificates of both servers. We are then able to inject JavaScript code to the user's browser from the attacker's server and make HTTP requests that are accepted and processed by the legitimate server.

2.6 Scope and Implications of the Attack

The MITM-SITB attack presented in Section 2.4 is not specific to TLS Channel ID-based client authentication protocols. In fact, it applies to *any* web client authentication method. This attack demonstrates that, in the context of web applications, it does not seem possible to prevent TLS MITM attacks via client authentication alone, no matter how "strong" the latter is.

We provide the following informal reasoning for the above claim. Client authentication alone does not prevent an attacker from impersonating the target web server. This allows her to intercept a server-authenticated (i.e., TLS) connection and ship malicious JavaScript code to the user's browser. The browser, treating the attacker's code as trusted (since it came through a server-authenticated connection), executes it within the origin of the target server. The attacker finally accesses the user's account through requests initiated by her code, and transmitted over another, direct connection between the browser and the legitimate server.

As a result, schemes such as traditional TLS client authentication [12], TLS Session Aware User Authentication [43, 44], and Browser-based Mutual Authentication [23], are all still susceptible to TLS MITM attacks. The attacker succeeds in impersonating the user to the web server and thus accessing his account (recall that this is the attacker's goal).

On the other hand, the Phoolproof phishing prevention system [48] is able to resist MITM+certificate attacks, due to its use of a server certificate pinning approach (see Section 3.1). However, this is true only if certificate pinning applies to every TLS connection that the browser establishes with the target web origin, as well as cross-origin communication that imports JavaScript.

We stress that, even if MITM-SITB is similar to existing attacks, it is *not* the same as a *Man-In-the-Browser* (MITB) attack [46, 53]. The latter implies that the attacker is able to take full control of the browser by exploiting some vulnerability, or installing a malicious browser plugin. In MITM-SITB, the attacker runs normal JavaScript code within the target web origin and only within the boundaries established by the JavaScript execution environment. Therefore, no browser exploitation is required. Similarly, this attack is *not* the same as *Cross-Site-Scripting* (XSS) [45, 55] either. Namely, there is no vulnerability in the pages served by the target web server that the attacker exploits in order to inject her code. Nevertheless, the end result of arbitrary client-side code execution within the target web origin is the same.

	Attacker Types	
Defenses	MITM+certificate	MITM+key
Strong Client Authentication	✗ (MITM-SITB)	✗ (MITM-SITB)
Enhanced Cert. Verification	✓	✗ (Conventional MITM, or MITM-SITB)
SISCA (our proposal)	✓	✓
Attacker's goal: User impersonation (via MITM attacks)		

Table 1: Overview on the prevention of TLS MITM attacks in the context of web applications.

3 Addressing TLS MITM Attacks

As we have shown in Section 2, client authentication is not sufficient to prevent MITM attacks that lead to user impersonation in web applications. Instead, we argue that in order to prevent such attacks, the client must be able to identify the content coming from a legitimate server.

In this section we discuss possible ways of effectively addressing TLS MITM attacks (including MITM-SITB). First, we briefly review existing techniques that address forged server certificates and thus can be applied to defend against MITM+certificate attacks. Then, we introduce a new approach, orthogonal to existing proposals, that is designed to resist both MITM+certificate and MITM+key attacks.

3.1 Existing Solutions

MITM+certificate attacks are feasible mainly due to the fact that web browsers blindly trust hundreds of CAs to sign certificates for *any* domain [27, 50] . A way to improve the security of the CA trust model is therefore to reduce the level of trust placed in the CAs. In recent years various proposals have emerged that follow this idea and perform *enhanced certificate verification*. These proposals are mostly based on two techniques: pinning and multi-path probing. We briefly mention some of the existing proposals below. We refer the interested reader to [8], for a thorough survey on existing solutions.

Pinning enables a web server to instruct browsers to accept only a specific set of certificates when establishing TLS connections to that server. Example solutions include HTTP [16] and TLS [39] extensions. Multi-path probing increases assurance about the legitimacy of the certificate by consulting (several) external sources. Prominent proposals include Perspectives [56], Convergence [38], DoubleCheck [2] and Certificate Transparency [37]. Besides pinning and multi-path solutions, several hybrid solutions have also emerged, including DNS-Based Authentication of Named Entities (DANE) [31] and Sovereign Keys (SKs) [15].

All the aforementioned techniques thwart MITM+certificate attacks, since they prevent the attacker from impersonating the legitimate server through the use of mis-issued certificates. In the presence of such solutions, the browser would not establish a connection or execute code from the rogue server. Moreover, these techniques can be combined with strong client authentication to protect the user in the best possible way, i.e., prevent user impersonation not only via MITM attacks, but also other attack vectors, such as phishing.

3.2 Our Proposal: SISCA

Each of the existing techniques mentioned above has its own strengths and weaknesses (discussed in detail in [8]). Moreover, to the best of our knowledge, none of the existing techniques were designed to address the MITM+key attacker. These techniques assume that the certificate used by the attacker is different from the legitimate one, which is not the case for MITM+key attacks.

In this section we present a solution, called *Server Invariance with Strong Client Authentication (SISCA)*, that mitigates both MITM+certificate and MITM+key attacks, such that the attacker cannot impersonate the user

to the legitimate server. Our solution is based on the combination of strong client authentication with the concept of sender invariance [14]. Table 1 shows a high level overview of existing techniques and SISCA, with respect to preventing user impersonation through TLS MITM attacks. SISCA is able to resist MITM+key attacks and, as we discuss in Section 3.2.7, also offers advantages for preventing MITM+certificate attacks, compared to existing proposals.

3.2.1 Main Concept

Our solution stems from the following observation. In the context of web applications, a MITM attacker can use two approaches to mount a user impersonation attack:

1. The conventional MITM attack, through which the attacker compromises user's credentials and uses them for impersonation. This attack can be effectively prevented with strong authentication e.g., using Channel ID-based protocols.

2. The MITM-SITB attack, presented in Section 2.4. As discussed in Section 2.6, client authentication alone cannot prevent this attack.

For the second attack to be successful, the user's browser needs to communicate with two different entities, namely the attacker and the target web server. Communicating with the attacker is, of course, necessary for injecting the attacker's script to the browser through the intercepted TLS connection. In addition, communication with the target server is essential, because this is how the attacker, accesses the user's account through her script.

As a result, we can detect and prevent this attack by making sure that the browser communicates only with one entity, either the legitimate server or the attacker, but *not* with both, during the same *browsing session* (a browsing session is terminated when the user closes the browser). In other words, we need to enforce *server invariance*. When combined with strong client authentication solutions (e.g., based on Channel IDs), which eliminates the first attack option, this technique manages to effectively thwart MITM attacks. The details of our solution follow below.

3.2.2 Assumptions

In our solution we assume the following. First, strong client authentication, which prevents the traditional way of implementing MITM attacks (see Figures 1 and 2), is in place. Specifically, we assume that *TLS Channel ID-based client authentication* is deployed. As mentioned before, Channel IDs are already experimentally supported in Google Chrome. Moreover, FIDO U2F, according to its draft specifications, also leverages Channel

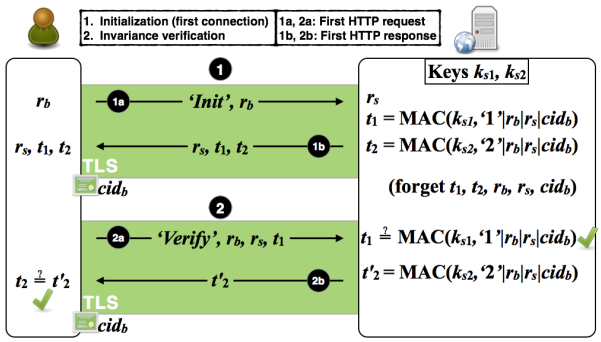


Figure 6: Basic SISCA protocol.

IDs similarly to Google U2F and PhoneAuth protocols, so it is likely that TLS Channel ID-based client authentication will become available in the foreseeable future.

Second, we assume that the legitimate web servers, that implement SISCA, support *TLS with forward secrecy* by default [30, 32, 35]. As we discuss below this is required for preventing MITM+key attacks (not relevant for MITM+certificate attacks). Moreover, we assume that TLS is secure and cannot be broken by attacks, such as those surveyed in [8].

We finally assume that the MITM+key attacker does not gain persistent presence on the target web server's side. As we discuss later, this enables SISCA to resist server key compromise (i.e, MITM+key attackers) through frequent rotation of the server secrets that are used in SISCA (Section 3.2.5). We also note that if an attacker would gain persistent control over the target server, she would most likely not need to resort to MITM attacks to access the users' accounts.

3.2.3 Basic Protocol

We begin describing how SISCA works, using `www.example.com` as our running example. We follow a structural approach, meaning that we start with a basic version of our protocol, described in this section. Then, in subsequent sections, we incrementally add features.

The protocol is implemented in the application layer, over established TLS sessions, via a new HTTP header, named `X-Server-Inv`, which is used for transmitting the protocol messages. For the protocol to be secure, on the client side this header is controlled solely by the browser. It cannot be created or accessed programmatically via scripts (similar to cookie-related headers [3]).

Figure 6 illustrates the protocol, assuming no attack. Prior to the protocol execution, the server (`www.example.com`) generates two keys k_{s1}, k_{s2}, which are not disclosed to the client or to the attacker. As we discuss in section 3.2.5, these keys can be frequently rotated, and this is why SISCA can also resist MITM+key attack-

ers (assuming no persistent server compromise). Moreover, the server and client deploy TLS Channel ID-based authentication. Each TLS session will therefore have a Channel ID cid_b that is created by the client's browser. The protocol consists of the following two phases.

Initialization. The first phase, called *initialization*, occurs after the browser establishes a TLS connection to `www.example.com`, for the *first time* in a browsing session (upper connection in Figure 6). The browser picks a random number r_b. It then sends $\langle \text{`}Init\text{'}, r_b \rangle$ to the server (`*Init*' is a string constant), within the *first* HTTP request[4] over that connection. Upon receiving this message, the server chooses a random number r_s and computes the following message authentication tags:

$$t_1 = \text{MAC}(k_{s1}, \text{`1'}|r_b|r_s|cid_b) \quad (1)$$

$$t_2 = \text{MAC}(k_{s2}, \text{`2'}|r_b|r_s|cid_b) \quad (2)$$

where '1' and '2' are strings constants. Notice that the server binds the computed tags to the Channel ID of the browser cid_b. r_b, r_s and the MAC tags will be used in subsequent TLS connections for the verification of server invariance.

Finally, the server sends $\langle r_s, t_1, t_2 \rangle$ to the browser within its first HTTP response. The browser stores $\langle r_b, r_s, t_1, t_2 \rangle$, while the server does not store any client-specific information. At this point, the initialization phase is complete. Subsequent HTTP requests and responses over that particular TLS connection do not include an `X-Server-Inv` header.

Invariance Verification. The second phase, called *invariance verification*, takes place upon every subsequent TLS connection to `www.example.com`, which occurs within the same browsing session (lower connection in Figure 6). Like in the first phase, the protocol messages are exchanged within the *first* HTTP request/response pair. The browser sends $\langle \text{`}Verify\text{'}, r_b, r_s, t_1 \rangle$ to the server, as part of the first request. After receiving the request, and *before* processing it, the server first checks if

$$t_1 \stackrel{?}{=} \text{MAC}(k_{s1}, \text{`1'}|r_b|r_s|cid_b) \quad (3)$$

. Here, cid_b corresponds to the Channel ID of the TLS session within which the protocol is being executed, which, if under attack, might differ from the Channel ID that was used in the initialization phase. If the check passes, the server computes

$$t_2' = \text{MAC}(k_{s2}, \text{`2'}|r_b|r_s|cid_b) \quad (4)$$

, processes the received request, and passes $\langle t_2' \rangle$ within the HTTP response back to the browser. Finally, the browser checks if $t_2' \stackrel{?}{=} t_2$. Assuming that the check succeeds, the server and the browser conclude that they are not under a MITM attack.

Analysis When Under Attack. Figure 7 illustrates how the protocol detects and prevents MITM attacks. Recall that, due to the usage of TLS Channel ID-based client authentication, the attacker cannot perform the attack in the traditional way (Figures 1 and 2) – the attacker's TLS sessions will have a different Channel ID than the client's and will thus be rejected. Instead, she has to execute the MITM-SITB attack, i.e., by shipping malicious JavaScript code to the browser.

In Figure 7 we illustrate two possible attack scenarios and we show why the attacker fails in both. In Figure 7a the attacker intercepts the verification phase of SISCA. Since the attacker didn't participate in the initialization phase of the protocol, she does not know the correct MAC response t_2 to the client's challenge. Moreover, since she does not have access to k_{s2}, she cannot calculate the correct t_2 either (Equation (4)). As a result, the user's browser rejects the attacker's response and terminates the session, notifying the user that a MITM attack was detected. Even if the attacker attempts to push a malicious script in her response, it will not get a chance of being executed.

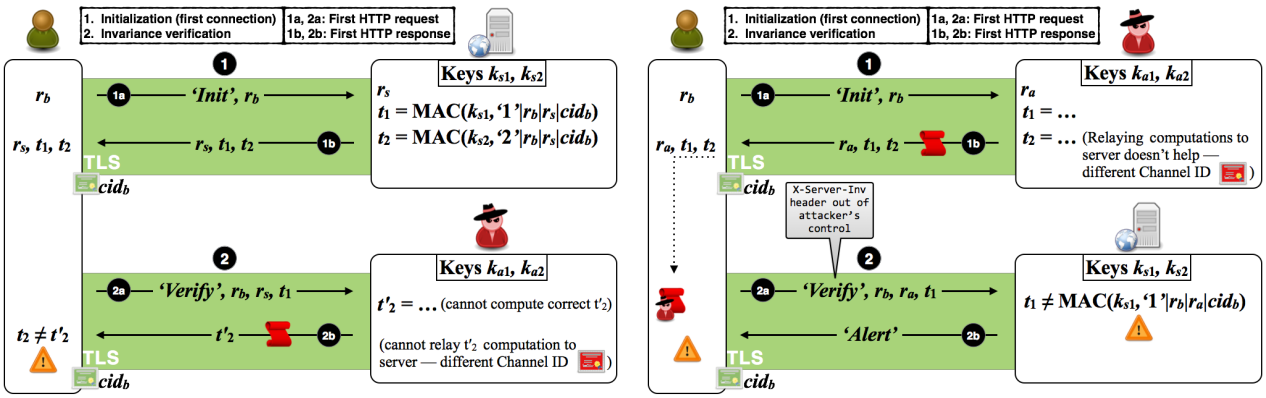
In the second scenario, depicted in Figure 7b, the attacker intercepts the first TLS connection to `www.example.com`. She executes the initialization phase with the browser and injects her script, which is executed within the web origin of `www.example.com`. To successfully complete her attack, the attacker needs to let a subsequent TLS connection reach the legitimate server, and access the user's account via that connection.

After the browser establishes a connection with the legitimate server, the two of them execute the invariance verification phase, as part of the first HTTP request/response pair. The server, before processing the HTTP request (which might as well be malicious), checks whether Condition (3) is true. Since the attacker does not have access to key k_{s1}, she could not have computed the correct t_1 (Equation (1)). Thus, during the initialization phase, she sends a t_1 value to the browser that is not the correct one. Consequently, Condition (3) will not be satisfied. In this case the server does not process the request, and instead notifies the browser by sending an empty HTTP response containing $\langle \text{`}Alert\text{'} \rangle$ in the `X-Server-Inv` header. This indicates violation of the server invariance and the browser aborts the session.

We remark that in the second scenario, it is the legitimate server that checks server invariance, detects the ongoing MITM attack and notifies the browser. This is important in order to prevent any malicious requests from being accepted and processed by the server.

We conclude our analysis, with a few remarks that are

[4]Note that this is a request that browser would anyway submit, i.e. required for loading the web page. It is not an extra request.



(a) Attacker intercepts invariance verification phase.

(b) Attacker intercepts initialization phase.

Figure 7: Resilience of SISCA to MITM (and SITB) attacks.

relevant for both of the scenarios described above. First, note that the attacker cannot relay any of the necessary MAC computations to the legitimate server. In other words, she cannot manipulate the server to compute for her the values needed for cheating in the protocol. This is because the server binds all its computations to the channel ID of the client with whom it communicates (the attacker's channel ID will be different from the user's).

Second, note that the protocol is secure so long as the attacker cannot "open" already established TLS connections between the browser and the legitimate server (i.e, connections that she chose not to intercept). If she could do that, she would be able to extract the correct values of both t_1 and t_2 and successfully cheat. Recall that, the MITM+key attacker holds the private key of the legitimate server. Therefore, in order to prevent such an attacker of eavesdropping on already established TLS connections, it is essential that these connections have TLS forward secrecy enabled.

Third, the attacker can choose not to reply at all, when executing the protocol with the user. This essentially leads to a Denial of Service (DoS) attack. However, such attacks can already be achieved even by attackers less powerful that those considered here. That is, attackers that cannot perform TLS MITM attacks, but can block network traffic between the browser and the server.

Different Origins. The SISCA protocol execution is guided by the same-origin policy [5]. In particular, SISCA is executed separately (i.e, different protocol instances) for web pages and documents that belong to different origins. For example, assume that the user navigates to `www.example.com` for the first time in the current browsing session. Then, a new instance of SISCA will be created for this origin, i.e., initialization phase will be executed on the first TLS connection. If the user subsequently clicks on links pointing to `www.example.com`, and this triggers the creation of new TLS connections by the browser, then for those connections, the browser will execute the invariance verification phase of the SISCA instance corresponding to `www.example.com` (same origin). If the user clicks on a link pointing to another website (different origin), say `www.another.com`, or opens a new tab and navigates to that site, then another new instance of SISCA will be created and used for the loading of documents from that origin (assuming that this is the first visit to `www.another.com` in that browsing session).

3.2.4 Cross-Origin Communication

Until now we assumed that accessing the web pages of the target server `www.example.com` involves communication only with that domain, i.e, web origin. However, this is not a realistic scenario in today's web applications. Many websites perform cross-origin requests (e.g., to load resources), either to subdomains, or even different top level domains. SISCA can accommodate for such scenarios so long as all the accessed domains belong to, and are administered by the *same entity*, such that the required SISCA keys, k_{s1} and k_{s2}, can be distributed across all the relevant servers.

Therefore, for cross-origin communication, the browser uses the SISCA instance corresponding to the initiating origin. For example, assume that a page loaded from `www.example.com` performs a cross-origin request to `static.example.com`. Then, the browser will create a TLS connection `static.example.com` and will perform the invariance verification phase of the SISCA protocol instance corresponding to `www.example.com`.

Different Channel IDs. The basic protocol we described in the previous Section also works in the cross-origin communication scenario, provided that the TLS Channel ID used by the browser is the *same*. The Channel ID specification draft already recommends us-

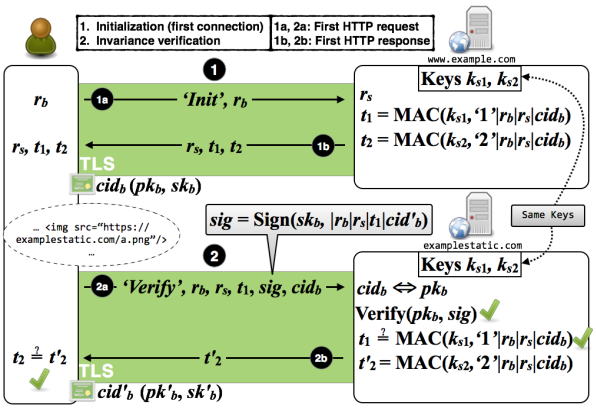


Figure 8: SISCA adapted for cross-origin communication (both origins belong to the same entity), when the browser uses a different Channel ID for each origin. In this example `www.example.com` performs a cross-origin request to `examplestatic.com`.

ing the same Channel ID for servers within the same top-level domain [4] (to account for top-level domain channel-bound cookies). This, for example, means that the browser should use the same Channel ID for `www.example.com` and `static.example.com`. Nevertheless, for privacy reasons, the specification recommends using different Channel IDs for different top-level domains. In such a case, SISCA has to account for the usage of different Channel IDs across domains.

Figure 8 depicts how the protocol works in such a scenario. The browser navigates to `www.example.com`, and starts a new SISCA instance for that origin. The browser uses Channel ID cid_b (with public key pk_b, and private key sk_b). Afterwards, the page loaded from `www.example.com` performs a cross-origin request to `examplestatic.com`, which is controlled by the same entity. Nevertheless, since it corresponds to a different top-level domain, the browser uses a different Channel ID, say cid'_b (with pk'_b, sk'_b being the corresponding public/private key pair). In this case, although the initialization phase of SISCA was executed using cid_b, the invariance verification phase will have to be executed over a TLS connection with Channel ID cid'_b.

As Figure 8 shows, the browser needs to tell the server `examplestatic.com` to use cid_b instead of cid'_b, but do so in a secure way. This can be achieved by having the browser endorse cid'_b, by signing it with sk_b, and thus proving to the server that it owns the private keys of both Channel IDs cid_b and cid'_b. As shown in the figure, the browser extends the '*Verify*' message by appending cid_b and a signature over cid'_b (i.e., the Channel ID of that TLS connection) and the rest of the message parameters using sk_b. The server, before processing the request, verifies the signature on cid'_b using the supplied cid_b (i.e., pk_b). If it passes, then the server uses Channel ID cid_b for the subsequent steps of the invariance verification phase, which remain unchanged.

This modified version of the SISCA protocol adds additional network and processing overhead due to the transfer and use of public-key cryptographic material. Channel IDs use elliptic curve cryptography [4], thus the overhead is not expected to be significant. Moreover, this can be further minimized, if the browser uses the same Channel ID, not only for subdomains, but also for domains that belong to the same entity. Although we do not elaborate on this idea here, this could be heuristically determined by the browser, based on which domains are involved in the execution of the same SISCA instance.

Overlapping Cross-Origin Access. Browsers typically send multiple HTTP requests over the same network connection (persistent connections [22]). Due to the existence of cross-origin communication, a TLS connection to a particular domain, say `static.example.com`, can be used by the browser to transmit cross-origin requests to `static.example.com` made by different initiating origins. For example, the browser uses the same TLS connection to `static.example.com`, to transmit, first, a request originating from a document belonging to `www.example.com` and then, a request originating from a document belonging to `shop.example.com` (we still assume that all three domains belong to the same entity). In this case, the TLS connection to `static.example.com` has to be verified using SISCA for both initiating domains, independently.

The browser executes the invariance verification phase with the SISCA instance of `www.example.com`, upon establishing the TLS connection to `static.example.com` and sending the first HTTP request, originating from `www.example.com`. Subsequently, when the browser wants to reuse this connection to send a cross-origin request from `shop.example.com`, it once again executes the invariance verification phase, only this time with the SISCA instance of `shop.example.com`. This takes place upon transmitting the first HTTP request, which originates from `shop.example.com`.

Origin Change. A web page is allowed to change its own origin (effective origin) to a suffix of its domain, by programmatically setting the value of `document.domain` [41]. This allows two pages belonging to different subdomains, but presumably to the same entity, to set their origin to a common value and enable interaction between them[5]. For example, a page from `www.example.com` and a page from `shop.example.com` can both set their origin to `example.com`. In such a case, the attacker can, for example, attack the user account at `shop.example.com`, by

[5] Both pages have to explicitly set `document.domain`.

intercepting the first connection to `www.example.com` (or any other `example.com` subdomain).

To prevent such an attack, the browser has to verify that server invariance holds across each pair of origins, that change their effective origin to a common value before allowing any interaction between them. Each origin has its own SISCA instance established, and we must ensure that both SISCA instances were initialized with the same remote entity. This can be achieved by running the invariance verification phase of both instances over the same TLS connection (established to either origin). The browser can reuse an already established and verified connection with one origin, and just verify the connection with the SISCA instance of the other origin. If no such connection exists at that time, then the browser can create a new one to either origin and execute the invariance verification phase of both SISCA instances. If there is no actual HTTP request to be sent at that time, the browser can make use the HTTP `OPTIONS` request.

Partial Support. Some websites may not support our solution. Moreover, other websites might opt for partial support. For example, a domain implements SISCA but still needs to perform cross-origin requests to a domain, called *incompatible*, that either does not support SISCA, or supports it, but belongs to a 3rd party, i.e., it has different SISCA keys (we discuss on the security of such design choices at the end of this section).

Partial support can be achieved by allowing exceptions. If a particular domain does not support SISCA (including legacy servers that are not aware of SISCA at all), then it can just ignore the `X-Server-Inv` header of the request during the initialization phase, and reply without including any SISCA-related information. This will be received by the browser as an *exception claim*. Moreover, if a domain supports SISCA but performs cross-origin communication with one or more incompatible domains, then it can append an *exception list* in its response, during the initialization phase, designating the incompatible domains. However, if the attacker intercepts the initialization phase of the protocol, then she could perform a *protocol downgrade attack*, by skipping the inclusion of the `X-Server-Inv` header, or providing false exception messages or exception lists, in her response.

To prevent this attack, SISCA protocol messages should include (in the `X-Server-Inv` header) the origin that corresponds to the corresponding SISCA instance. This can help distinguish between genuine and attacker-produced exception claims in the following way. Upon subsequent TLS connections, the browser, instead of executing the invariance verification phase, sends a notification of the exception claim it received during SISCA initialization. The receiving server can leverage the supplied origin information to check if the exception claim is valid and act accordingly. This assumes that each domain is aware of the domains that is compatible to execute SISCA with (i.e, domains with which it shares the same SISCA keys), which is not difficult to implement.

3rd Party Content Inclusion. As mentioned above, a domain, say `www.example.com`, implementing SISCA can still perform cross-origin requests to incompatible 3rd party domains as long as it designates those domains as exceptions for the protocol. This of course means that TLS connections to those domains will not be protected by SISCA, and could be MITM-ed by the attacker to perform a user impersonation attack on `www.example.com`. This is indeed the case if `www.example.com` includes active content [40] (in particular, JavaScript and CSS) from those domains. Embedding JavaScript from 3rd party sites is generally not recommended, and usually there are ways of avoiding it [42]. Furthermore, depending on the use case, it may be possible to use iframes to include active 3rd party content instead of directly embedding it within the target origin, in order to mitigate the risk (the `sandbox` attribute can help even further).

On the other hand, the embedding of passive content only, such as images, does not give the attacker the ability to execute her code within the target origin. Hence, with respect to user impersonation prevention, such embeddings are considered safe and do not undermine the security offered by SISCA.

3.2.5 Key Rotation

In SISCA, the server has a pair of secret keys, k_{s1} and k_{s2}. To resist key compromise (i.e, MITM+key attackers), these keys, unlike the server's private key, can be easily rotated. This is because the SISCA keys need not undergo any certification process, and can thus be rotated frequently, e.g, weekly, daily or even hourly. The more frequent the rotation the smaller is a MITM+key attacker's window of opportunity to successfully mount MITM attacks.

The key transition, of course, has to be performed such that it does not break the execution of active browser SISCA instances that rely on the previous keys. On a high level, one way of achieving this, is to have the server keep previous keys for a certain period of time (i.e, allow partial overlap of keys). This allows browsers with active SISCA instances that rely on the previous keys to securely transition to new protocol parameters, i.e, t_1 and t_2, computed using the new server SISCA keys.

For domains served by a single machine, this is only a matter of implementing the corresponding functionality in the web server software (e.g., Apache). For multiple domains controlled by the same entity and served by multiple machines, located in the same data center or even in different data centers across the world, arguably more effort is required in order to distribute the ever-changing

keys and keep the machines in sync. Nevertheless, a similar mechanism is needed for enabling TLS forward secrecy while supporting TLS session tickets [36]. According to Twitter's official blog [32], Twitter engineers have implemented such a key distribution mechanism.

3.2.6 Resource Caching

Caching of static resources, such as scripts and images, helps reduce web page loading times as well as server resource consumption. However, the way caching is currently implemented [22, 26] can give a MITM attacker the opportunity to subvert SISCA.

In a nutshell, during one browsing session, the attacker intercepts all TLS connections and ensures that a maliciously modified, legitimate script that is required by the target web server is cached by the browser. Then, during a second browsing session, the attacker lets all connections pass through. When the legitimate web page asks for the inclusion of the aforementioned script, the browser will load it from cache, essentially enabling the execution of the attacker's malicious code. The attacker will thus be able to access the target web server.

To prevent the above attack, we need to change the way caching is performed for active content that would enable this attack (JavaScript and CSS files). We need to make sure that the browser always communicates with the server in order to verify that the cached version is the most recent and also the correct one (i.e, not maliciously modified). Thus, caching of such files should be performed only using Entity Tags (ETags) [22], but in a more rigorous way than specified in the current HTTP specification. In particular, if a web server wishes to instruct a browser to cache a JavaScript or CSS file, the server should use an `ETag` header which always contains a cryptographic hash of the file. The browser, before using, and caching the file should *verify* that the supplied hash is correct. Subsequently, before the browser uses the cached version of the file, it first verifies (using the `If-None-Match` header, as before) that the local version matches the version of the server.

3.2.7 Comparison with Existing Techniques

Besides being able to resist MITM+key attacks, SISCA also offers advantages for preventing MITM+certificate attacks. Compared to multi-path probing solutions, SISCA does not rely on any trusted third parties. Since SISCA is built on top of Channel ID-based authentication, it has to assume that no MITM attack takes place during user enrollment. Nevertheless, after this step, no "blind" trust is required when the user uses a new or clean browser, contrary to most pinning solutions (except preloaded pins). Finally, SISCA is scalable since it can be deployed independently by web providers.

The main disadvantage of SISCA is that it only protect against MITM attackers whose goal is to impersonate the user to the server. It does not protects against attackers whose objective is to provide fake content to the user or try to elicit user information, through social engineering techniques. In contrast, the existing techniques can protect against such attacks (when considering a MITM+certificate attacker). As a result, a good strategy would be to combine SISCA with existing techniques. This would provide a higher level of protection against MITM+certificate attackers, as well as prevent MITM+key attackers from accessing the users' accounts on the legitimate server.

3.3 Prototype SISCA Implementation

We created a proof of concept implementation of the basic SISCA protocol, with additional support for cross-origin communication, when the same Channel ID is used. On the server side we use Apache 2.4.7 with OpenSSL 1.0.1f, patched for Channel ID support. SISCA is implemented as an Apache module and consists of 313 lines of C code. On the client side we implement SISCA by modifying the source code of Chromium 35.0.1849.0 (252194) and the WebKit (Blink) engine. We make a total of 319 line modifications (insertions/deletions) in existing files and we add 6 new files consisting of 418 lines of C++ code.

We use base64 encoding for binary data transmission. When using 128-bit random values (r_b and r_s) and HMAC-SHA256 (i.e, 256-bit tags, t_1 and t_2), the client's lengthiest message is 114 bytes long (excluding the origin of the SISCA instance that has to be sent as well). The server's lengthiest message is 132 bytes long.

Finally, we tested our implementation and verified that it successfully detects and blocks our proof of concept MITM-SITB attack.

4 Discussion

Interaction With Other Web Technologies

SPDY. SPDY protocol [6] multiplexes concurrent HTTP requests over the same TLS connection to improve network performance. In order for SISCA to be compatible with the general SPDY functionality, the browser must ensure that, before the SISCA protocol is completed successfully (i.e, the first request/response pair is exchanged), no further requests should be pushed to the SPDY connection.

Furthermore, SPDY offers a feature, called IP Pooling, that allows, under certain circumstances, HTTP sessions from the same browser to different domains (web

origins) to be multiplexed over the same connection. Version 3 of SPDY is compatible with TLS Channel IDs (recall that different Channel IDs may need to be presented to different origins, but now there is only one TLS connection). SISCA is also compatible with IP Pooling, provided that the browser keeps track and manages the multiplexed HTTP sessions independently, with respect to the execution of the SISCA protocol.

WebSocket. SISCA is compatible with the WebSocket protocol [19], when the latter is executed over TLS. This, of course assumes that (i) Channel IDs are used for the WebSocket TLS connections, (ii) the SISCA protocol is executed during the WebSocket handshake (i.e, first request/response pair), and (iii) JavaScript is not be able to manipulate the `X-Server-Inv` header.

Web Storage. Web Storage [29] is an HTML5 feature that allows a web application to store data locally in the browser. SISCA does not prevent a MITM attacker from accessing information stored in `window.localStorage` (permanent storage), so non sensitive information should be stored there. It protects `code.sessionStorage` which stores data for one session (data is lost when the tab is closed).

Offline Web Applications. HTML5 offers Offline Web Applications [28] which allow a website to create an offline version, stored locally in the browser through caching. As with regular file caching discussed in Section 3.2.6, this feature can be leveraged by the attacker to bypass SISCA. Making this feature secure requires the introduction of design concepts similar to what we proposed for regular caching.

Other Client-Side Technologies. The attacker might attempt to leverage various active client-side technologies besides JavaScript, such as Flash, Java and Silverlight. Such technologies allow the attacker to create direct TLS connections to the legitimate server. Some of the APIs offered by those technologies also allow the attacker to forge and arbitrarily manipulate HTTP headers, including cookie-related headers or the `X-Server-Inv` header. However, provided that TLS Channel IDs and SISCA are not integrated with these technologies[6], the attacker will not be able to impersonate the user and access his account on the legitimate server.

5 Related Work

A significant amount of research in the past years surrounds the security of the TLS protocol, in the context of web applications (i.e, HTTPS), as well as web server and client authentication. A comprehensive overview is provided in [8], which, among others, surveys existing primitives that enhance the CA trust model to more effectively address MITM attacks. Some techniques focus on enhancing the server certificate verification process, while others enhance client authentication in order to specifically deter user impersonation attacks.

The use of server impersonation for the compromise of the user's account by serving the attacker's script to the victim's browser was first introduced in [34]. In this attack, called Dynamic Pharming, the attacker exploits DNS rebinding vulnerabilities in browsers, by dynamically manipulating DNS records for the target web server in order to force the user's browser to connect either to the attacker (to inject her script) or to the legitimate server (to access the user's account via her script).

MITM-SITB is very similar to Dynamic Pharming in that it leverages server impersonation to serve the script to the victim's browser. However, since MITM-SITB attacks are designed against Channel ID-based solutions where the server's certificate is miss-issued or its key is compromised, the attack can be even simpler and does not need to rely on DNS manipulation. Instead, it can leverage any form of MITM where the attacker controls the communication to the client (e.g., an attacker sitting on a backbone) and rely only on the behavior of the browser to re-establish a connection (with the legitimate server) once the attacker closes the connection within which she injected her script to the browser.

Since dynamic pharming attacks were published before the introduction of Channel ID-based solutions, their implications to these solutions were not discussed in [34]. Nevertheless, we note that dynamic pharming can equally be used to successfully attack Channel ID-based solutions. Recently, the possibility of leveraging script injection via server impersonation against TLS Client Certificate Authentication was also briefly discussed in [49]. However, none of the prior works discuss or implement these types of attacks on Channel ID-based solutions.

To prevent dynamic pharming, the authors of [34] proposed the locked same-origin policy, which refines the concept of origin by including the public key of the server. Locked same-origin policy per se does not prevent a MITM attacker from stealing weak user credentials (password, cookies) and using them to impersonate the user. A strong client authentication solution should be used in conjunction, as with SISCA. Moreover, locked same-origin policy does not address cross-origin active content inclusion and does not resist MITM+key attacks. We note that SISCA prevents both dynamic pharming attacks and other similar forms of MITM attacks such as MITM-SITB, including under the assumption of MITM+key attackers.

The current Channel ID specification [4] was recently found to be vulnerable to a new class of attacks

[6]This, for example, means that a TLS connection created by such an API will have to create and use its own Channel IDs, and that the browser will not execute SISCA over those connections

known as "triple handshake attacks" [7], that affect TLS client authentication in general. A MITM+certificate or MITM+key attacker can exploit a protocol design flaw during TLS session resumption in order to successfully trick the legitimate server into believing that the attacker holds the private key that corresponds to the user browser's Channel ID. This flaw allows the attacker to bypass the protection offered by Channel IDs and successfully mount a conventional MITM attack in order to impersonate the user to the server. A mitigation is proposed in [7] that has already been implemented in the version of Chromium that we used in this work. SISCA assumes that Channel IDs work as expected and hence eliminating triple handshake attacks is essential for its security. However, the mitigation proposed in [7] is focused on ensuring that TLS Channel IDs work indeed as expected and therefore does not prevent MITM-SITB attacks.

The risks involved when a web page imports active content, such as JavaScript, that can be intercepted and modified by an attacker are discussed in [33]. SISCA can protect such inclusions (either from the same origin or cross-origin) as long as the involved domains belong to the same entity and thus share the same SISCA keys.

The concept of sender invariance was formally defined in [14]. SISCA is inspired by this notion, assuming that the server's authenticity cannot be established via conventional server certificate verification and requiring that the browser always communicates with the same entity (either the legitimate server or the attacker, but not with both) during the same browsing session.

6 Conclusion

In this paper we discussed the requirements to effectively preventing TLS MITM attacks in the context of web applications, when the attacker's goal is to impersonate the user to the legitimate server and access his account. We showed that the recently proposed TLS Channel ID-based authentication cannot prevent such attacks. We provided informal reasoning why such attacks can only be thwarted by addressing the weaknesses of the server authentication process. We proposed SISCA, a novel technique that combines Channel ID-based client authentication with the notion of sender invariance. SISCA is able to resist MITM attacks, when the attacker holds a mis-issued certified key, or the key of the legitimate server. We showed how our solution can be integrated in today's web infrastructure, along with existing defenses in order to further enhance the security against MITM attacks.

References

[1] ADKINS, H. An update on attempted man-in-the-middle attacks. http://googleonlinesecurity.blogspot.ch/2011/08/update-on-attempted-man-in-middle.html, 2011.

[2] ALICHERRY, M., AND KEROMYTIS, A. D. DoubleCheck: Multi-path verification against man-in-the-middle attacks. In *ISCC, 2009*.

[3] AUBOURG, J., SONG, J., STEEN, H. R. M., AND VAN KESTEREN, A. XMLHttpRequest (W3C Working Draft). http://www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/, Dec. 2012.

[4] BALFANZ, D., AND HAMILTON, R. Transport Layer Security (TLS) Channel IDs, v01 (IETF Internet-Draf). http://tools.ietf.org/html/draft-balfanz-tls-channelid-01, June 2013.

[5] BARTH, A. The web origin concept (RFC 6454). http://tools.ietf.org/html/rfc6454, Dec. 2011.

[6] BELSHE, M., AND PEON, R. SPDY protocol (IETF Internet-Draf). http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00, Feb. 2012.

[7] BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., PIRONTI, A., AND STRUB, P.-Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. Submitted for publication.

[8] CLARK, J., AND VAN OORSCHOT, P. C. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE SP (Oakland), 2013*.

[9] COATES, M. Revoking trust in two TurkTrust certificates. https://blog.mozilla.org/security/2013/01/03/revoking-trust-in-two-turktrust-certficates/, 2013.

[10] CZESKIS, A., AND BALFANZ, D. Protected login. In *USEC, 2012*.

[11] CZESKIS, A., DIETZ, M., KOHNO, T., WALLACH, D., AND BALFANZ, D. Strengthening user authentication through opportunistic cryptographic identity assertions. In *CCS, 2012*.

[12] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) protocol, version 1.2 (RFC 5246). http://tools.ietf.org/html/rfc5246, Aug. 2008.

[13] DIETZ, M., CZESKIS, A., BALFANZ, D., AND WALLACH, D. S. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX Security, 2012*.

[14] DRIELSMA, P. H., MÖDERSHEIM, S., VIGANÒ, L., AND BASIN, D. Formalizing and analyzing sender invariance. In *FAST, 2006*.

[15] ECKERSLEY, P. The Sovereign Keys project. https://www.eff.org/sovereign-keys, Nov. 2011.

[16] EVANS, C., PALMER, C., AND SLEEVI, R. Public key pinning extension for HTTP (IETF Internet-Draf). http://tools.ietf.org/html/draft-ietf-websec-key-pinning-09, Nov. 2013.

[17] F-SECURE. Mobile threat report Q3 2013. http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf, 2013.

[18] FELT, A. P., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *SPSM, 2011*.

[19] FETTE, I., AND MELNIKOV, A. The WebSocket protocol (RFC 6455). http://tools.ietf.org/html/rfc6455, Dec. 2011.

[20] FIDO ALLIANCE. http://fidoalliance.org/.

[21] FIDO ALLIANCE. Universal 2nd Factor (U2F) overview, Version 1.0 (review draft). http://fidoalliance.org/specs/fido-u2f-overview-v1.0-rd-20140209.pdf, 2014.

[22] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616). http://tools.ietf.org/html/rfc2616, June 1999.

[23] GAJEK, S., MANULIS, M., SADEGHI, A.-R., AND SCHWENK, J. Provably secure browser-based user-aware mutual authentication over TLS. In *ASIACCS, 2008*.

[24] GOOGLE. Universal 2nd Factor (U2F). https://sites.google.com/site/oauthgoog/gnubby.

[25] GOOGLE DEVELOPERS. Minimize request overhead. https://developers.google.com/speed/docs/best-practices/request.

[26] GOOGLE DEVELOPERS. Optimize caching. https://developers.google.com/speed/docs/best-practices/caching.

[27] HAYES, J. M. The problem with multiple roots in web browsers-certificate masquerading. In *WETICE, 1998*.

[28] HICKSON, I. Offline web applications (HTML 5 working draft). http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html.

[29] HICKSON, I. Web storage (W3C Recommendation). http://www.w3.org/TR/webstorage/, July 2013.

[30] HIGGINS, P. Pushing for perfect forward secrecy, an important web privacy protection. https://www.eff.org/deeplinks/2013/08/pushing-perfect-forward-secrecy-important-web-privacy-protection, 2013.

[31] HOFFMAN, P., AND SCHLYTER, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) protocol: TLSA (RFC 6698). http://tools.ietf.org/html/rfc6698, Aug. 2012.

[32] HOFFMAN-ANDREWS, J. Forward secrecy at Twitter. https://blog.twitter.com/2013/forward-secrecy-at-twitter-0, 2013.

[33] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *Web 2.0 Security and Privacy, 2008*.

[34] KARLOF, C., SHANKAR, U., TYGAR, J. D., AND WAGNER, D. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS, 2007*.

[35] LANGLEY, A. Protecting data for the long term with forward secrecy. http://googleonlinesecurity.blogspot.ch/2011/11/protecting-data-for-long-term-with.html, 2011.

[36] LANGLEY, A. How to botch TLS forward secrecy. https://www.imperialviolet.org/2013/06/27/botchingpfs.html, 2013.

[37] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency (RFC 6992). http://tools.ietf.org/html/rfc6962, June 2013.

[38] MARLINSPIKE, M. Convergence. http://convergence.io/, 2011.

[39] MARLINSPIKE, M., AND PERRIN, T. Trust Assertions for Certificate Keys (TACK) (IETF Internet-Draft). http://tack.io/draft.html, Jan. 2013.

[40] MOZILLA DEVELOPER NETWORK. Mixed content. https://developer.mozilla.org/en-US/docs/Security/MixedContent.

[41] MOZILLA DEVELOPER NETWORK. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript.

[42] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote Javascript inclusions. In *CCS, 2012*.

[43] OPPLIGER, R., HAUSER, R., AND BASIN, D. SSL/TLS session-aware user authentication - Or how to effectively thwart the man-in-the-middle. *Computer Communications 29*, 12 (Aug. 2006), 2238–2246.

[44] OPPLIGER, R., HAUSER, R., AND BASIN, D. SSL/TLS session-aware user authentication revisited. *Computers & Security 27*, 3-4 (2008), 64–70.

[45] OWASP. Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[46] OWASP. Man-in-the-browser attack. https://www.owasp.org/index.php/Man-in-the-browser_attack.

[47] PAOLA, S. D., AND FEDON, G. Subverting Ajax. 23rd Chaos Communication Congress, 2006.

[48] PARNO, B., KUO, C., AND PERRIG, A. Phoolproof phishing prevention. In *Financial Cryptography, 2006*.

[49] PARSOVS, A. Practical issues with TLS client certificate authentication. In *NDSS, 2014*.

[50] PERLMAN, R. An overview of PKI trust models. *Network, IEEE 13*, 6 (1999), 38–43.

[51] SCHNEIER, B. New NSA leak shows MITM attacks against major Internet services. https://www.schneier.com/blog/archives/2013/09/new_nsa_leak_sh.html, 2013.

[52] SOGHOIAN, C., AND STAMM, S. Certified lies: Detecting and defeating government interception attacks against SSL. In *Financial Cryptography, 2011*.

[53] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your botnet is my botnet: Analysis of a botnet takeover. In *CCS, 2009*.

[54] TRUSTWARE SPIDERLABS. Clarifying The Trustwave CA policy update. http://blog.spiderlabs.com/2012/02/clarifying-the-trustwave-ca-policy-update.html, 2012.

[55] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRÜGEL, C., AND VIGNA, G. Cross Site Scripting prevention with dynamic data tainting and static analysis. In *NDSS,2007*.

[56] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference, 2008*.

[57] WOOD, M. Fraudulent certificates issued by Comodo, is it time to rethink who we trust? http://nakedsecurity.sophos.com/2011/03/24/fraudulent-certificates-issued-by-comodo-is-it-time-to-rethink-who-we-trust/, 2011.

[58] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *IEEE SP (Oakland), 2012*.