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

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

Abstract

In this paper we consider TLS Man-In-The-Middle (MITM) attacks in the context of web applications, where the attacker is able to successfully impersonate the legitimate server to the user, with the goal of impersonating the user to the server and thus compromising the user's online account and data. We describe in detail why the recently proposed client authentication protocols based on TLS Channel IDs, as well as client web authentication in general, cannot fully prevent such attacks.

Nevertheless, we show that strong client authentication, such as Channel ID-based authentication, can be combined with the concept of server invariance, a weaker but easier to achieve property than server authentication, in order to protect against the considered attacks. We specifically leverage Channel ID-based authentication in combination with server invariance to create a novel mechanism that we call SISCA: Server Invariance with Strong Client Authentication. SISCA resists user impersonation via TLS MITM attacks, regardless of how the attacker is able to successfully achieve server impersonation. 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 cookies. Such credentials are considered *weak*; they are transmitted over the network and are susceptible to theft and abuse unless protected by TLS.

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. This way, the attacker gains access to the user's account and data which can be abused for a variety of purposes, such as spying on the user [18, 48]. 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 trusted Certificate Authority (CA). 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 holds the corresponding private key. In previous years, quite a few incidents involving mis-issued certificates [2, 11, 48, 49, 9] were made public. Even in the case where the attacker simply presents an invalid (e.g., self-signed) certificate not accepted by the browser, she will still succeed in her attack if the user defies the browser's security warning.

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., [20, 33, 36, 52]).

Other proposals focus on strengthening client authentication. *Strong* client authentication prevents user credential theft or renders it useless, even if the attacker can successfully impersonate the server to the user. One such prominent proposal is Channel ID-based client authentication, introduced in 2012. TLS Channel IDs [4] are experimentally supported in Google Chrome and are planned to be used in the second factor authentication standard U2F, proposed by the FIDO alliance [22].

In this paper we show that Channel ID-based approaches, as well as web authentication solutions that focus solely on client authentication are vulnerable to an attack that we call *Man-In-The-Middle-Script-In-The-Browser (MITM-SITB)*, and is similar to *dynamic pharming* [32] (see Section 4). This attack bypasses Channel ID-based defenses by shipping malicious JavaScript to the user's browser within a TLS connection with the attacker, and using this JavaScript in direct connections with the legitimate server to attack the user's account.

Nevertheless, we show that TLS MITM attacks where the attacker's goal is user impersonation can still be prevented by strong client authentication, such as Channel ID-based authentication, provided that it is combined with the concept of *server invariance*, that is, the requirement that the client keeps communicating with the same entity (either the legitimate server, or the attacker) across multiple connections intended for the same server. Server invariance is a weaker requirement than server authentication, but at the same time, it is easier to achieve as no initial trust is necessary. Building on this observation, we propose a solution called *SISCA: Server Invariance with Strong Client Authentication*, that combines Channel ID-based client authentication and server invariance.

SISCA can resist TLS MITM attacks that are based on mis-issued valid certificates, as well as invalid certificates, requiring no user involvement in the detection of the attack (i.e., no by-passable security warnings when server invariance violation occurs). SISCA also thwarts attackers that hold the private key of the legitimate server.

Contributions. 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 Channel ID-based client authentication solutions do not fully prevent TLS MITM attacks; *(ii)* we further argue that effective prevention of MITM-based user impersonation attacks requires strong user authentication and (at least) server invariance; *(iii)* we propose a novel solution that prevents MITM-based user impersonation based on the combination of strong client authentication and server invariance; *(iv)* we implement and evaluate a basic prototype of our solution.

2 Channel ID-based Authentication and MITM Attacks

2.1 Attacker Model and Goals

Attacker Goals. The attacker's goal in a MITM attack is typically to impersonate the user (victim) to the legitimate server (e.g., a social networking, webmail, or e-banking website) in order to compromise the user's online account and data. This is indeed the case where the attacker wishes for example to spy on the user [18, 48], or abuse his account for nefarious purposes, e.g., perform fraudulent financial transactions. Alternatively, the attacker could aim to only impersonate the server to the user (and not the user to the server) such that she serves the user with fake content (e.g., fake news). In this paper, we focus on the first, more impactful, scenario.

Attacker Model. We adopt the attacker model considered by Channel IDs [4]. The adversary is able to position herself suitably on the network and perform a TLS MITM attack between the user and the target web server. In other words, the attacker is able to successfully impersonate the server to the user. We distinguish between two types of MITM[1] attackers.

The *MITM+certificate* attacker holds *(i)* 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 [2, 11, 48, 9]. Moreover, in this category we also consider a weaker attacker that only holds *(ii)* an *invalid* (e.g., self-signed) certificate. In this case, the attacker will still succeed in impersonating the server to the user if the latter ignores the security warnings of the browser[2], which is a common phenomenon [51].

The *MITM+key* attacker holds the *private key of the legitimate server*. While we are not aware of publicized incidents involving server key compromise, such attacks are feasible, as the Heartbleed vulnerability in OpenSSL has shown [1], and can be very stealthy, remaining unnoticed. Thus, they are well worth addressing [28, 30, 35].

From the above it follows that the attacker is able to obtain the user's weak credentials, namely passwords and HTTP cookies. She is not, however, able to compromise the user's browser or his devices (e.g., mobile phones).

2.2 TLS Channel IDs

Channel IDs is a recent proposal for strengthening client authentication. It is a TLS extension, originally proposed in [15] as *Origin-Bound Certificates* (OBCs). A refined version has been submitted as an IETF Internet-Draft [4]. Currently, Channel IDs are experimentally supported by Google's Chrome browser and Google servers.

In brief, 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. This enables the web server to identify the same browser across multiple TLS connections.

2.2.1 Channel ID-Based Authentication

By *Channel ID-based authentication* we refer to the use of Channel IDs *throughout* the user authentication process, designed to thwart both types of MITM attackers presented in Section 2.1 [4, §6], [13, §3].

---

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

[2] We use the term "browser" to refer to any "user agent" in general.

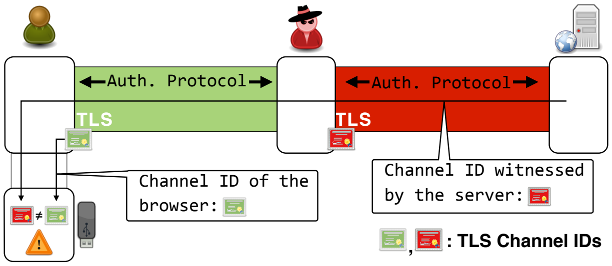


Figure 1: PhoneAuth/U2F. Leveraging Channel IDs to secure the initial login against MITM attacks.

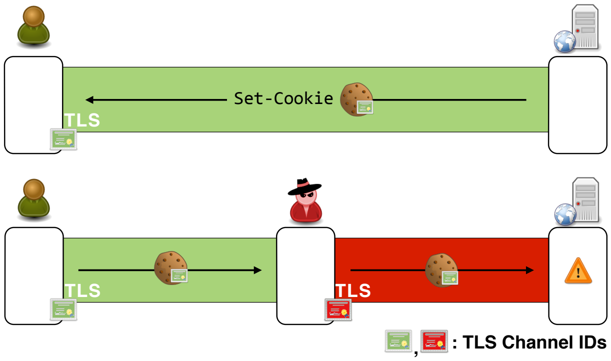


Figure 2: Binding authentication tokens (e.g., cookies) to the browser's Channel ID (green). A MITM attacker who steals such a cookie, cannot use it to impersonate the user, since the attacker has a different Channel ID (red).

Initial Login. 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, as in *PhoneAuth* [13] and *FIDO Universal 2nd Factor (U2F)* [22] protocols. These protocols leverage Channel IDs to secure the login process against MITM attacks. In brief, as part of the authentication protocol, the second factor device compares the Channel ID of the browser to the Channel ID of the TLS connection that the server witnesses. If they are equal, then the 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 Channel IDs differ, then the server is not directly connected to the user's browser. Instead, as shown in Figure 1, there is an attacker in the middle, and the device aborts the authentication protocol, stopping the attack.

Subsequent Logins. Upon successful initial authentication the server sets a cookie to the user's browser, and binds it to the Channel ID of the browser. As proposed in [15], a server may create a *channel-bound cookie* as follows: $\langle v, \mathrm{HMAC}(k, v|cid)\rangle$, where v is the original cookie value, cid is the browser Channel ID and k is a secret key, only known to the server, used for computing a MAC over the concatenation of v and cid. The channel-bound cookie is considered valid only if it is presented over that particular Channel ID. Therefore, subsequent interaction with the server from that particular browser is protected by the channel-bound cookie. An attacker that manages to steal a channel-bound cookie, e.g., through a MITM attack, cannot use it to impersonate the user to the web server, since she does not know the private key of the correct Channel ID. Figure 2 illustrates this concept. Note that at this stage, the second factor device is not required for authenticating the user [12].

2.3 MITM Attack on Channel ID-Based Authentication

We show how Channel ID-based authentication still allows a MITM 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 (e.g., IMAP over TLS).

In particular, web servers are allowed to push scripting code to the browser, which the latter executes within the security 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.

Moreover, a browser can 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 establish TLS connections with multiple servers while loading a web page.

Given the above, there is a conceptually simple attack that a MITM+certificate or MITM+key attacker can perform, which bypasses the security offered by Channel IDs. 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 certificate (or invalid with the user ignoring the browser's warning), 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 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 con-

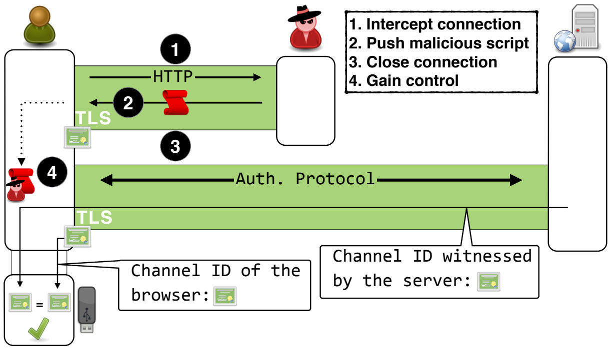


Figure 3: MITM-SITB attack on Channel ID-based PhoneAuth/U2F, used for the initial login. The attacker's JavaScript code is executed within the origin of the target server (shown by the dotted arrow).

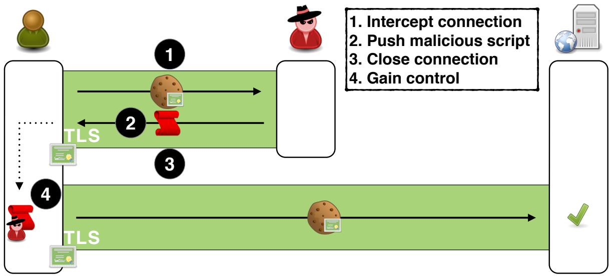


Figure 4: MITM-SITB attack on Channel ID-based authentication after the initial login, where requests are protected with a channel-bound cookie.

nection in order to transmit subsequent requests, or use another existing one, if any (this behavior conforms with the HTTP specification [23]). At the same time, the attacker allows subsequent TLS connection attempts to pass through, without interfering with them. As a result, once the attacker closes that single intercepted connection, all other connections, existing and new, are directly established between the browser and the legitimate server.

4. The attacker gains full control over the user's session in that particular web application. Her script has unrestricted access over the web documents belonging to `www.example.com` and can monitor all the client-side activity of the web application. Moreover, she can issue arbitrary malicious requests to the target server using the `XMLHttpRequest` object [3], in order to perform a desired action or extract sensitive user information. The malicious code can upload any extracted data to an attacker-controlled server. As another example, if the web application is Ajax-based, the attacker can perform *Prototype Hijacking* [46]. This allows her to eavesdrop and modify on-the-fly all the HTTP requests made through `XMLHttpRequest`.

In summary, the MITM attacker "transfers" herself (via the malicious script) within the user's browser, and continues her attack from there. We call this attack *Man-In-The-Middle-Script-In-The-Browser* (*MITM-SITB*).

Figure 3 illustrates the MITM-SITB attack in the case when the user is about to initially authenticate to `www.example.com` using PhoneAuth or U2F. The attacker intercepts a TLS connection, pushes her JavaScript code to the user's browser, and terminates the connection. The browser then establishes a new TLS connection for subsequent communication, only this time with the legitimate server; the attacker will not hijack it. This ensures that the user authentication is performed over a direct connection between the browser and the server, but with the attacker's code running in the browser. The view of the TLS channel will be the same for the browser and the server, and the Channel ID comparison made by the second factor device will pass.

Figure 4 shows how the attack works in the case when the user has already logged in on `www.example.com` in the past, and the server has set a channel-bound cookie in the user's browser. Like before, the attacker ships malicious JavaScript code to the browser by intercepting a TLS connection to `www.example.com`. She then terminates the intercepted connection. This forces the browser to establish a new TLS connection, which is not intercepted by the attacker. This ensures that any subsequent requests, either legitimate or malicious (issued by the attacker's script) are accepted by the legitimate server, since they will carry the channel-bound cookie, which authenticates the user, over the correct Channel ID.

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 persist as much as possible, etc. Depending on the scenario, there are various alternatives, which are mostly implementation decisions. The attacker can for example 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 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. For exam-

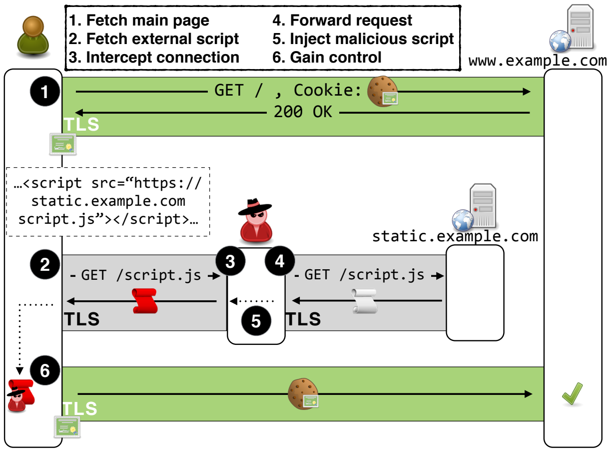


Figure 5: MITM-SITB attack on Channel ID-based authentication leveraging cross-origin communication. Channel IDs for `static.example.com` are of no use.

ple a typical network optimization technique 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` [24]). These domains, as their name suggests, do not set any cookies, so as to minimize network latency. As a matter of fact, on such domains, client authentication does not 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 her malicious code at the moment when the target web server requests a legitimate JavaScript file from that domain (Figure 5).

2.4 Proof of Concept Attack

We validate our attack against Channel IDs through a proof of concept implementation. We use two Apache TLS-enabled servers (one for the attacker, 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 ensure that 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 issue HTTP requests that are accepted and processed by the legitimate server.

2.5 Scope and Implications of the Attack

The MITM-SITB attack presented in Section 2.3 is not specific to 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.

We provide the following informal reasoning for the above claim. Client authentication does not prevent an attacker from impersonating the legitimate server. This allows her to intercept a server-authenticated (i.e., TLS) connection and ship her JavaScript code to the user's browser. The browser, treating the attacker's code as trusted (as it came through a server-authenticated connection), executes it within the target server's origin. The attacker 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 [14] and TLS Session Aware User Authentication [42, 43] are still susceptible to TLS MITM attacks. The attacker succeeds in impersonating the user to the web server and thus compromising his account.

3 Addressing TLS MITM Attacks

As shown in Section 2, strong client authentication alone is not sufficient to prevent MITM attacks that lead to user impersonation in web applications. So, how can we effectively prevent such attacks? In this section we show that there are two orthogonal solutions; *(i)* the known solution of preventing the attacker from impersonating the legitimate server at all, i.e., ensuring correct server authentication; *(ii)* our novel approach of combining strong client authentication with server invariance.

3.1 Prevent Server Impersonation

The known and straightforward solution to the problem at hand is to prevent the attacker from impersonating the server in the first place. This way, the attacker can neither steal weak user credentials in order to mount a conventional MITM attack, nor ship malicious Javascript in order to mount a MITM-SITB attack. Note that in this case, strong client authentication (e.g., Channel ID-based) is not necessary for preventing MITM attacks (it is, however, still useful for preventing other attacks, such as phishing and server password database compromise).

The solutions that try to prevent server impersonation, essentially address the issue of forged server certificates (and thus defeating MITM+certificate attacks), by performing *enhanced certificate verification*. Such solutions are mainly based on *pinning* [20, 38], *multi-path probing* [33, 36, 37, 52] and hybrid approaches [19, 29] (a thorough survey can be found in [10]).

3.2 Our Proposal: SISCA

3.2.1 Main Concept

The fact that strong client authentication alone cannot effectively prevent MITM attacks in web applications,

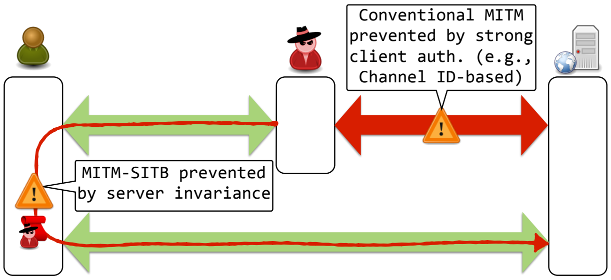


Figure 6: TLS MITM attacks in web applications can be thwarted by combining strong client authentication with server invariance.

raises the following question. Is there a way to somehow still benefit from strong client authentication with respect to addressing MITM attacks?

To answer, we make the following observation. In the context of web applications, a MITM attacker can perform user impersonation via two approaches:

1. The *conventional MITM* attack, in which the attacker compromises the user's credentials and uses them for impersonation. This attack can be effectively prevented by strong client authentication e.g., using Channel ID-based protocols (Figures 1, 2).
2. The *MITM-SITB* attack, presented in Section 2.3 (Figures 3, 4, 5). As discussed in Section 2.5, client authentication alone cannot prevent this attack.

For the MITM-SITB 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, so that the attacker accesses the user's account and data, through her script.

As a result, we can detect and prevent MITM-SITB by making sure that the browser communicates only with one *entity*, either the legitimate server or the attacker, but *not* with both, during a *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 (e.g., Channel ID-based), which stops the conventional MITM approach, this technique manages to effectively thwart MITM attacks. Figure 6 illustrates the concept.

In the remaining section we present a novel solution, called *Server Invariance with Strong Client Authentication (SISCA)*, which stems from the above result. SISCA is able to resist MITM+certificate attacks, offering advantages compared to existing solutions that focus at preventing server impersonation (see Section 3.2.9), as well as MITM+key attacks under the assumption that the attacker does not persistently compromise the server (see Section 3.2.2). The details of our solution follow below.

3.2.2 Design Goals and Assumptions

In SISCA we seek to satisfy the following requirements: *(i)* incremental deployment, *(ii)* scalability, *(iii)* minimal overhead (processing, network, maintenance), *(iv)* account for cross-origin communication, assuming that the involved origins belong to, and are administered by the *same entity*, *(v)* mitigation of MITM+key attacks (besides MITM+certificate attacks).

We make the following assumptions. First, strong client authentication, which prevents the conventional way of implementing MITM attacks (Figures 1, 2) is in place. Specifically, we assume that SISCA-enabled servers implement *Channel ID-based client authentication*. As mentioned before, Channel IDs are already experimentally supported in Google Chrome. Moreover, FIDO U2F leverages Channel IDs, as metiononed in Section 2.2.1, so it is likely that Channel ID-based authentication will become available in the foreseeable future.

Second, we assume that SISCA-enabled servers support *TLS with forward secrecy* by default [28, 30, 35]. As we discuss below, this is only required for preventing MITM+key attacks (not relevant for MITM+certificate attacks). Moreover, we assume that TLS is secure and cannot be broken by cryptographic attacks, such as those surveyed in [10].

We finally assume that the MITM+key attacker does not persistently compromise the target web server. 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 (see Section 3.2.8). We also note that if an attacker gained persistent control over the target server, she would probably not need to resort to MITM attacks to compromise the users' accounts, but at the same time she would increase the probability of being detected.

3.2.3 Server Invariance Versus Authentication

As stated above, our goal is to combine strong client authentication with server invariance. Invariance is a *weaker* property than authentication, but it is also *easier* to achieve, as no a priori trust is required. In contrast, authentication requires some form of initial trust so that the client can correctly authenticate the server [17].

Consequently, we stress the following very important difference. Server authentication (and solutions that try to enforce it, like those mentioned in Section 3.1) implies that every single TLS connection will be established with the legitimate server. If the attacker attempts to intercept such a connection, she will be detected by the browser, i.e., no server impersonation should be possible.

In contrast, server invariance, embraces the fact that the attacker can successfully impersonate the server. As such, we distinguish two scenarios concerning the browser's *first connection* to a particular server, during a browsing session: *(i)* The first connection is *not intercepted* by the attacker. Then, server invariance implies that the attacker is allowed to *intercept none* of the subsequent connections to that server. *(ii)* The first connection is *intercepted* by the attacker. Then, server invariance implies that the attacker has to *intercept all* subsequent connections to that server. In either scenario, if the attacker violates server invariance, she will be detected.

We consider server invariance as a *transient* property whose scope is one browsing session. Server invariance is *reset* whenever the browser restarts, i.e., the attacker is allowed again to choose whether to intercept or not the first connection to the server.

3.2.4 Towards Implementing Server Invariance

In order to implement server invariance, it is important to understand the implications of the fact that the attacker is allowed to impersonate the server. Namely, the attacker can intercept the first connection and influence the entire HTTP response, which clearly cannot be blindly trusted. Therefore, techniques that assume the attacker is able to influence only a part of the HTTP response, such as *Content Security Policy* (CSP) [50] for mitigating *Cross-Site-Scripting* (XSS) [44], as well as techniques that assume the first connection is trusted (i.e., not intercepted by the attacker), such as pinning, cannot be applied directly for implementing server invariance.

Instead, a server invariance protocol should consist of *two phases*, namely *invariance initialization* and *invariance verification* – initialization and verification for brevity. In the initialization phase, which is executed in the first connection to the server during a browsing session, the browser establishes a *point of reference*. Then, in subsequent connections to the same server, the verification phase is executed, where the browser verifies that the point of reference remains unchanged.

Server Public Keys. Assuming that we only consider MITM+certificate attackers, we could leverage the public key of the server as the point of reference. Even if the attacker intercepts the first connection, she will not be able to let any subsequent connections reach the legitimate server, because the server's public key will be different from the attacker's. Nevertheless, servers of the same domain may use different public keys and also, cross-origin interacting domains will have different keys. To solve this issue, we have to "tie" all the involved public keys together, to reflect the fact that they belong to the same entity and thus server invariance holds across all these domains. We sketch a couple of techniques, either of which could be relatively easily integrated with CSP in order to implement server invariance.

As one option, during initialization (first connection), the server sends a list of all the involved domains and all their public keys to the browser, and the latter uses the witnessed key as well as the list as the point of reference. Then, in subsequent connections, the browser verifies *(i)* that the public key which the server presents is contained in the list which was received during initialization, and *(ii)* that the server agrees on the authenticity of the public key that was originally witnessed by the browser during initialization. Notice how this differs from pinning, which would blindly trust the initial connection, and thus would not seek to verify the authenticity of the initial connection, and consequently of the received list, upon subsequent connections.

Another option is to apply a solution similar to the one described in [32, §4.5], where each public key has to be signed with every other private key. Then, during initialization, the server presents a list of all public keys and signatures to the browser, so that the latter can verify server invariance in the subsequent connections, based on the presented server key of each connection.

The techniques described above are useful when only considering MITM+certificate attacks. However, we also seek to mitigate MITM+key attacks, which implies that we cannot rely on server public keys. Moreover, the above techniques may increase the required maintenance effort as the number of involved public keys increases. Instead, we aim for an "install-and-forget" approach.

Our High-Level Approach. In SISCA we choose to implement server invariance as a simple challenge/response protocol. In the initialization phase (first connection) the browser sets up a fresh challenge/response pair (which acts as the point of reference) with the server. Then, in the verification phase (subsequent connections) the browser challenges the server to verify server invariance, i.e. that it is the same entity with which the browser executed the initialization.

SISCA has to be executed before any HTTP traffic influenced by the attacker is processed by the browser or the server. We choose to implement the protocol at the application layer, over established TLS sessions via an HTTP header, named `X-Server-Inv`, and transmitted together with the *first* HTTP request/response pair over a particular TLS connection. 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]).

Alternatively, we could implement the server invariance protocol in SISCA as a TLS extension, i.e., at the transport layer. However, we argue that the application layer is more appropriate, since server invariance includes notions such as web origin and cross-origin communication that naturally belong to the application layer.

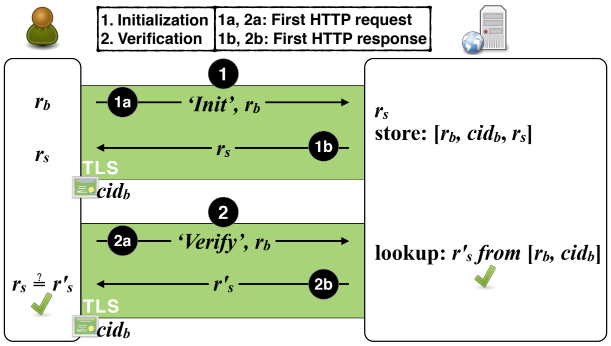


Figure 7: An example challenge/response-based server invariance protocol requiring per-client server state.

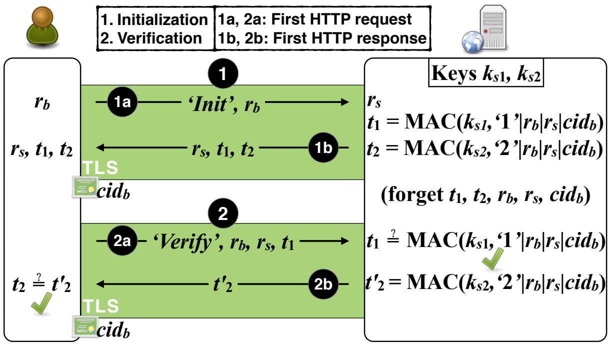


Figure 8: Basic SISCA protocol.

Figure 7 depicts a simple example of how a protocol based on our aproach can look like. In this example protocol, during the initialization phase, the browser and server generate random numbers r_b and r_s, which they both store (the server also stores the browser's Channel ID cid_b). The browser subsequently uses r_b as a challenge during the verification phase, expecting the response r_s by the server. The latter looks up r_s by using r_b and cid_b. For the shake of brevity, we do not analyze this example, but we make the following important remarks.

First, this example requires the server to store per-client state. This may be undesirable and it also makes it harder for multiple servers belonging to the same entity to share the common state which is needed in order to be able to correctly execute the protocol. For this reason, SISCA uses symmetric cryptography (MAC), in order to securely offload the state to the clients.

Second, during the verification phase, the server should process the incoming HTTP request, only if the lookup succeeds. If it fails, it means that the attacker intercepted the first connection (initialization phase) and that the incoming request may be malicious. We explain this concept further in the analysis of the SISCA protocol. We note that due to this fact, SISCA uses a second MAC tag in order to enable the server perform this check.

3.2.5 Basic Protocol

We now describe the server invariance protocol of SISCA in detail. 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.

Figure 8 illustrates the protocol, assuming no attack. Prior to the protocol execution, the server, `www.example.com`, generates two keys k_{s1} and k_{s2}, called *SISCA keys*. The same SISCA keys are used for all protocol executions (i.e., not for a specific client) and are never disclosed to other parties. Moreover, recall that the server and client deploy Channel ID-based authentication. Each TLS connection will therefore have a Channel ID cid_b, that is created by the user's browser. As already mentioned, the protocol consists of two phases.

Initialization. The initilization phase occurs once the browser establishes a TLS connection to `www.example.com`, for the *first time* in a browsing session (upper connection in Figure 8). The browser picks a random number r_b. It then sends $\langle$‘*Init*’$,r_b\rangle$ to the server (‘*Init*’ is a string constant), within the *first* HTTP request[3] over that connection. Upon receiving this message, the server chooses a random number r_s and computes the following message authentication tags:

$$t_1 = \mathrm{MAC}(k_{s1}, \text{‘1’}|r_b|r_s|cid_b) \quad (1)$$

$$t_2 = \mathrm{MAC}(k_{s2}, \text{‘2’}|r_b|r_s|cid_b) \quad (2)$$

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

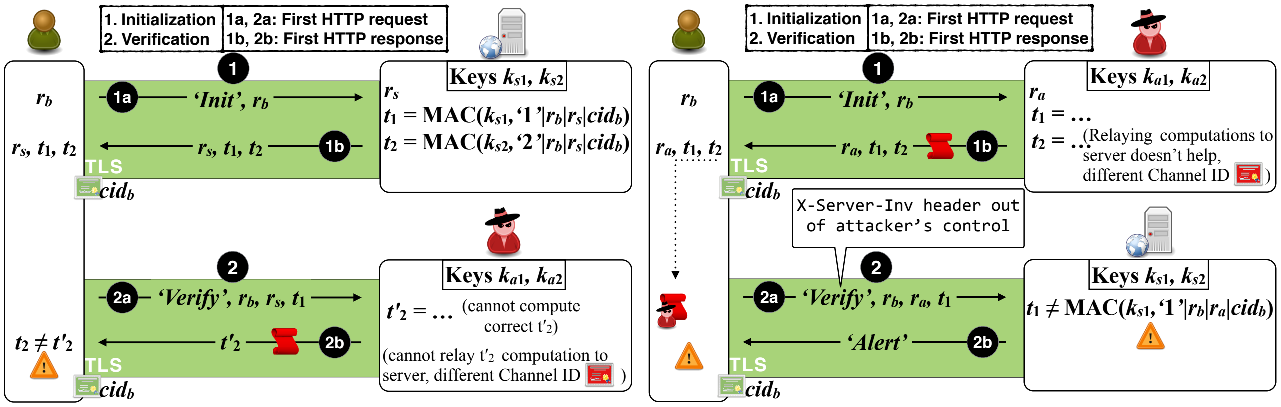
Finally, the server sends $\langle r_s, t_1, t_2\rangle$ to the browser within its first HTTP response. The browser stores $\langle r_b, r_s, t_1, t_2\rangle$, while the server does not store any client-specific information. At this point, the initialization phase is complete. Subsequent HTTP requests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

Verification. The verification phase takes place upon every subsequent TLS connection to `www.example.com`, which occurs within the same browsing session (lower connection in Figure 8). Like in the first phase, the protocol messages are exchanged within the *first* HTTP request/response pair. The browser sends $\langle$‘*Verify*’$,r_b,r_s,t_1\rangle$ to the server, as part of the first request. After receiving the request, and *before* processing it, the server first checks if

$$t_1 \overset{?}{=} \mathrm{MAC}(k_{s1}, \text{‘1’}|r_b|r_s|cid_b). \quad (3)$$

---

[3] 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 the verification phase.

(b) Attacker intercepts the initialization phase.

Figure 9: Resilience of SISCA to MITM-SITB (conventional MITM is prevented by Channel-ID based authentication).

Here, cid_b corresponds to the Channel ID of the TLS session within which the protocol is currently 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

$$t'_2 = \text{MAC}(k_{s2}, \text{‘2’}|r_b|r_s|cid_b), \tag{4}$$

processes the received request, and passes $\langle t'_2 \rangle$ within the HTTP response to the browser. Finally, the browser checks if $t'_2 \overset{?}{=} t_2$ and if it succeeds, the server and the browser conclude that they are not under a MITM attack.

Analysis When Under Attack. Figure 9 illustrates how the protocol detects and prevents MITM attacks. Recall that, due to the usage of Channel ID-based authentication, the attacker cannot perform the conventional attack (Figures 1, 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 a MITM-SITB attack.

In Figure 9 we illustrate two possible attack scenarios (based on the discussion of Section 3.2.3) and we show why the attacker fails in both. In Figure 9a 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 (Eq. (4)). As a result, the user's browser rejects the attacker's response and terminates the session, notifying the user (no user decision is required). Even if the attacker pushes a malicious script in her response, it will not get a chance of being executed.

In the second scenario, depicted in Figure 9b, the attacker intercepts the first TLS connection to `www.example.com`. She thus 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 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 (Eq. (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 $\langle$‘*Alert*’$\rangle$ 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 even a single malicious request from being accepted and processed by the server.

We conclude our analysis, with a few remarks that are 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 from eavesdropping on already established TLS connections, it is essential that these connections have TLS forward secrecy enabled.

Third, when considering MITM+key attacks it is reasonable to assume that the attacker can also extract the SISCA keys, similar to the private key of the server. As stated in the assumptions (Section 3.2.2) and explained in Section 3.2.8, SISCA keys, unlike the private key, can be frequently rotated. SISCA can thus resist MITM+key attacks, assuming no persistent server compromise.

Finally, the attacker can choose not to reply at all, when executing SISCA 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 independently, i.e., different *protocol instances*, when loading web pages and documents that belong to different origins. For example, assume that the browser 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 and its initialization phase will be executed on the first TLS connection. If the browser further navigates to pages belonging 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 verification phase of the previously created SISCA instance corresponding to `www.example.com` (same origin). When the browser navigates to another website (different origin), say `www.another.com`, then a 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). Also any HTTP redirections during navigation that lead to different origins will cause the corresponding SISCA instances for those origins to be created and used.

3.2.6 Cross-Origin Communication

Until now we assumed that accessing the web pages of `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 involved domains belong to, and are administered by the *same entity*, such that the required SISCA keys, k_{s1} and k_{s2}, can be shared across all relevant servers.

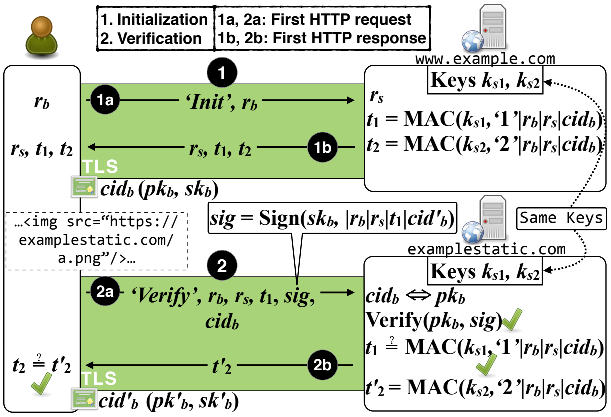


Figure 10: SISCA adapted for cross-origin communication (the origins share the same SISCA keys), when the browser uses a different Channel ID for each origin. Here, `www.example.com` performs a cross-origin request to `examplestatic.com`.

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`. The browser will create a TLS connection to `static.example.com` and will execute the verification phase of the SISCA instance that corresponds to `www.example.com`. Any potential HTTP redirections will also use the SISCA instance of the initiating origin, `www.example.com`.

Different Channel IDs. The basic protocol we described in Section 3.2.5 also works in the cross-origin communication scenario, provided that the Channel ID used by the browser is the *same*. The Channel ID specification draft already recommends using the same Channel ID for servers within the same top-level domain [4] (to account for top-level domain channel-bound cookies). For example, 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 cases, SISCA has to account for using different Channel IDs across domains, when cross-origin communication takes place.

Figure 10 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). At some later point in time, 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 verification phase will have to be executed over a TLS connection with Channel ID cid'_b.

As Figure 10 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. To achieve this, the browser endorses 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. 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 cid_b for the subsequent steps of the verification phase, which remain unchanged.

Overlapping Cross-Origin Access. Browsers typically send multiple HTTP requests over the same network connection (persistent connections [23]). 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.

In the above scenario, the browser executes the 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 the same connection to send a cross-origin request from `shop.example.com` to `static.example.com`, it once again executes the 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` [40]. 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[4]. 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 attack the user account at `shop.example.com`, by intercepting the first connection to `www.example.com` (or any other `example.com` subdomain), or vice versa.

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 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 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 and Downgrade Attacks. SISCA must be incrementally deployable, which means that it must maintain compatibility with legacy web servers, without compromising the security of the SISCA-enabled servers. Moreover, websites must be able to opt for partial support. As an example, a domain implements SISCA but still needs to perform cross-origin requests to a another 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).

The above 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 simply ignore the `X-Server-Inv` header, sent 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 providing false exception claims or exception lists in her response.

To prevent downgrade attacks, the browser should verify any exception that was received during the initializa-

---

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

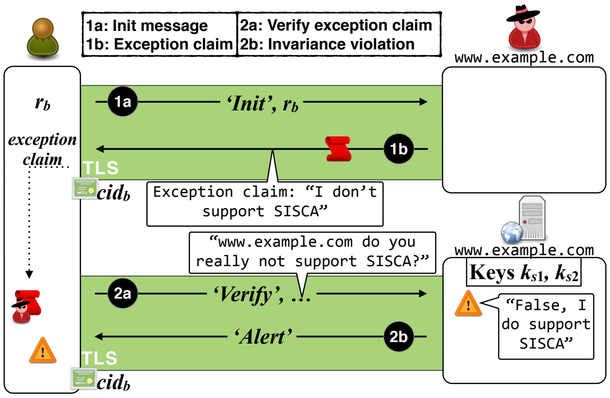


Figure 11: Preventing downgrade attacks (same-origin case).

tion phase, upon every subsequent connection. If the attacker intercepted the initialization phase and replied with fake exception claims, then if any of the subsequent connections reaches the legitimate server, the browser, with the help of the legitimate server, would detect the attack. This scenario is illustrated in Figure 11.

Regarding cross-origin communication, in order to help SISCA-enabled legitimate servers detect fake exception lists previously received by the browser, SISCA protocol messages should include (in the `X-Server-Inv` header) the origin associated with the SISCA instance. Suppose for example, that the browser executes the initialization phase with `www.example.com` which supports SISCA (executes the protocol normally), but also includes an exception list stating that it performs cross-origin requests to `shop.example.com` which does not support SISCA. Whenever the browser connects to `shop.example.com` to perform a cross-origin request from `www.example.com`, the browser includes the origin of the SISCA instance (`www.example.com`) and asks `shop.example.com` whether it indeed does not support SISCA with respect to that origin. Assuming that the connection was not intercepted, `shop.example.com` can leverage the supplied origin information to decide whether the exception reported by the browser is valid. If not, then it should abort processing the request and notify the browser of the detected attack. Note that the above 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 can be indeed the case if `www.example.com` includes active content [39] (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 [41]. Furthermore, depending on the use case, it may be possible to use iframes to isolate 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).

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 preventing user impersonation, such embeddings are safe and do not undermine the security offered by SISCA.

3.2.7 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 [23, 25] can give a MITM attacker the opportunity to subvert SISCA.

In brief, during one browsing session, the attacker intercepts all TLS connections and ensures that a legitimate, yet maliciously modified 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) [23], 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 that the local version matches the version of the server (using the `If-None-Match` header, as currently done).

3.2.8 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 the 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. At 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 can enable 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 [34]. According to Twitter's official blog [30], Twitter engineers have implemented such a key distribution mechanism.

3.2.9 SISCA Benefits and Drawbacks

SISCA offers the following advantages regarding MITM+certificate attack prevention. Compared to multipath probing solutions, SISCA does not rely on any third party infrastructure, trusted or not. 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 pinning solutions (except preloaded pins). Moreover, in SISCA no user decision is necessary whenever server invariance violation is detected. This can occur either due to an attack or due to an internal server fault, thus the browser can abort (possibly after retrying) the session. SISCA is scalable since it can be deployed incrementally by web providers (assuming browser support). Finally, SISCA resists MITM+key attacks, assuming that the attacker does not persistently compromise the server.

The main disadvantage of SISCA is that it only protects against MITM attackers whose goal is to impersonate the user to the server. This is arguably the most common and impactful attacker goal. SISCA does not protect against attackers whose objective is to provide fake content to the user. In such cases the attacker can simply intercept *all* connections and interact with the user by serving her own, fake content. In contrast, the techniques that focus on ensuring the correctness of server authentication (Section 3.1) can protect against such attacks (for MITM+certificate attackers). As a result, a recommended strategy would be to use SISCA in conjunction with any of these techniques. Finally, SISCA requires coordination between an entity's different domains, in the sense they must have access to the same SISCA keys. This is needed for securing cross-origin communication and, depending on the scale of the entity, can be challenging from an engineering perspective.

3.2.10 Interaction With Other Web Technologies

SPDY. SPDY [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 are pushed to the SPDY connection.

Furthermore, SPDY IP Pooling 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 Channel IDs (recall that different Channel IDs may need to be used for different origins, but now there is only one TLS connection). SISCA is compatible with IP Pooling, as long as the browser manages the multiplexed HTTP sessions independently, with respect to the execution of the SISCA protocol.

WebSocket. SISCA is compatible with the WebSocket protocol [21], 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 [27] is an HTML5 feature that allows a web application to store data locally in the browser. SISCA can protect `code.sessionStorage` (temporary storage), but does not prevent a MITM attacker from accessing information stored in `window.localStorage` (permanent storage), so no sensitive information should be stored there.

Offline Web Applications. HTML5 offers Offline Web Applications [26] which allow a website to create an offline version, stored locally in the browser. As with regular file caching (see Section 3.2.7), 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 Channel IDs and SISCA are not integrated with these technologies[5], the attacker will not be able to impersonate the user and compromise his account 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, provided that 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, plus the origin of the SISCA instance that has to be sent as well. The server's lengthiest message is 132 bytes long.

We finally verified that our implementation successfully blocks our proof of concept MITM-SITB attack.

Performance Evaluation. To assess the performance overhead imposed by SISCA (the server invariance part, not the overhead due to Channel IDs), we measured the latency of HTTP request/response roundtrips, with SISCA enabled and disabled. For the measurements we used a 4KB HTML page, as well as an 84KB jQuery compressed file, retrieved over a domain that we set up as being "cookieless". Chromium ran on a Macbook Pro laptop (2.3GHz CPU, 8GB RAM) and Apache ran on a typical server machine (six core Intel Xeon 2.53GHz, 12GB RAM), connected through the campus network.

We found that the overhead of the basic SISCA protocol is negligible, as no increase in latency was measured (averaged over 300 repetitions). Moreover, the HTTP request to the cookieless domain was able to fit in a single outgoing packet (a typically desired objective).

Regarding cross-origin communication over different Channel IDs (see Section 3.2.6), approximately 180 bytes are further added to the request (one ECDSA public key and signature in Base64 encoding), which can still fit in a single packet (for cookieless domain requests). Furthermore, the server has to perform one ECDSA signature verification. This overhead could be minimized, if the browser used the same Channel ID, not only for subdomains of the same top-level domain, but also for domains belonging 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.

Finally, recall that a SISCA instance is executed only once per TLS connection and not on every HTTP request/response.

4 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 [10], which, among others, surveys existing primitives that try to enhance the CA trust model in order to more effectively address MITM 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 [32]. In this attack, called *dynamic pharming*, the attacker exploits DNS rebinding vulnerabilities in browsers, by dynamically manipulating DNS records for the target server, in order to force the user's browser to connect either to the attacker (to inject her script) or to the legitimate server.

MITM-SITB is therefore very similar to dynamic pharming in that it leverages server impersonation to serve the script to the victim's browser. Dynamic pharming focuses on the attacker's ability to control the client's network traffic via DNS attacks, while in this paper we do not make such assumptions. Instead, MITM-SITB can leverage any form of MITM where the attacker controls the communication to the client (e.g., an attacker sitting on a backbone) and relies 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. Dynamic pharming can equally be used to successfully attack Channel ID-based solutions. Recently, the act of leveraging script injection via server impersonation against TLS client authentication was also discussed in [47].

We note that MITM-SITB (as well as dynamic pharming) differs from *Man-In-the-Browser* (MITB) [45]. 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, MITM-SITB is different from XSS [44]. In XSS the attacker is able

---

[5] 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.

to influence only parts of the served document (by exploiting a script injection vulnerability), while in MITM-SITB she is able to impersonate the server and thus influence the entire HTTP response sent to the browser. SISCA does not prevent MITB or XSS and addressing these attacks is orthogonal to our work.

To prevent dynamic pharming, the locked same-origin policy (SOP) was proposed [32]. Weak locked SOP considers attackers with invalid certificates, while strong locked SOP also defends against attackers with valid, mis-issued certificates. Strong locked SOP refines the concept of origin by including the public key of the server and can also accommodate for multiple server keys. Strong locked SOP isolates web objects coming from connections with not endorsed server public keys in a separate security context (i.e., different origin). Strong locked SOP per se does not prevent a MITM attacker from mounting a conventional MITM attack in order to impersonate the user. A strong client authentication solution should be used in conjunction, as with SISCA.

Locked SOP does not resist MITM+key attacks, as SISCA does. Moreover, locked SOP is not able to secure cross-origin active content inclusion. 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 [31]. SISCA can secure cross-origin inclusions as long as the involved domains belong to the same entity and thus share the same SISCA keys.

The current Channel ID specification [4] was recently found to be vulnerable to *triple handshake attacks* [7], which affect TLS client authentication in general. The mitigation proposed in [7] has already been implemented in the version of Chromium that we used in this work. SISCA assumes that Channel IDs work as expected, so eliminating triple handshake attacks is essential for its security. However, we note that addressing triple handshake attacks does not prevent MITM-SITB attacks.

Recent work has proposed leveraging Channel ID-based authentication to strengthen federated login [16] and Cloud authorization credentials [8], against MITM attacks and credential theft in general. However, such proposals fail to address MITM attacks, as they are susceptible to MITM-SITB, unless augmented with server invariance, as we propose in this paper with SISCA.

Server invariance is based on *sender invariance* which was formally defined in [17]. SISCA is inspired by this notion, assuming that the server's authenticity cannot be established via server certificate verification and instead trying to enforce the weaker property of invariance.

5 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 gain access to the user's account and data. Striving to defeat this type of attack is essential, especially given the recent revelations that government agencies (e.g., the NSA) mount such attacks in order to perform mass surveillance against users of major internet services [18, 48].

We showed that strong client authentication alone, such as the recently proposed Channel ID-based authentication, cannot prevent such attacks. Instead, strong client authentication needs to be complemented with the concept of server invariance, which is a weaker, yet easier to enforce requirement than server authentication. Our solution, SISCA, shows that server invariance can be implemented with minimal additional cost on top of the proposed Channel ID-based approaches, and can be deployed incrementally, thus making it a scalable solution. Given its security benefits, we believe that SISCA can act as an additional, strong protection layer in conjunction with existing proposals that focus on amending today's server authentication issues, towards the effective prevention of TLS MITM attacks.

Acknowledgements

We thank Kari Kostiainen, Arnis Parsovs, Hubert Ritzdorf, Mario Strasser, Dan Wallach, Der-Yeuan Yu and the anonymous reviewers for their helpful feedback and suggestions. Some of the icons used in this work were adapted from `opensecurityarchitecture.org`.

References

[1] The Heartbleed Bug. `http://heartbleed.com/`.

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

[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/`.

[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`, 2013.

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

[6] Belshe, M., and Peon, R. SPDY protocol (IETF Internet-Draf). `http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00`, 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. In *IEEE SP (Oakland), 2014*.

[8] Birgisson, A., Politz, J. G., Erlingsson, U., Taly, A., Vrable, M., and Lentczner, M. Macaroons: Cookies with contextual caveats for decentralized authorization in the Cloud. In *NDSS, 2014*.

[9] Bright, P. Independent Iranian Hacker Claims Responsibility for Comodo Hack. `http://www.wired.com/2011/03/comodo_hack/`.

[10] 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*.

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

[12] Czeskis, A., and Balfanz, D. Protected login. In *USEC, 2012*.

[13] Czeskis, A., Dietz, M., Kohno, T., Wallach, D., and Balfanz, D. Strengthening user authentication through opportunistic cryptographic identity assertions. In *CCS, 2012*.

[14] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) protocol, version 1.2 (RFC 5246). `http://tools.ietf.org/html/rfc5246`, 2008.

[15] 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*.

[16] Dietz, M., and Wallach, D. S. Hardening Persona – Improving federated web login. In *NDSS, 2014*.

[17] Drielsma, P. H., Mödersheim, S., Viganò, L., and Basin, D. Formalizing and analyzing sender invariance. In *FAST, 2006*.

[18] Eckersley, P. A Syrian MITM attack against Facebook. `https://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook`.

[19] Eckersley, P. The Sovereign Keys project. `https://www.eff.org/sovereign-keys`.

[20] 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`, 2013.

[21] Fette, I., and Melnikov, A. The WebSocket protocol (RFC 6455). `http://tools.ietf.org/html/rfc6455`, 2011.

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

[23] 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`, 1999.

[24] Google Developers. Minimize request overhead. `https://developers.google.com/speed/docs/best-practices/request`.

[25] Google Developers. Optimize caching. `https://developers.google.com/speed/docs/best-practices/caching`.

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

[27] Hickson, I. Web storage (W3C Recommendation). `http://www.w3.org/TR/webstorage/`.

[28] 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`.

[29] 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`, 2012.

[30] Hoffman-Andrews, J. Forward secrecy at Twitter. `https://blog.twitter.com/2013/forward-secrecy-at-twitter-0`.

[31] Jackson, C., and Barth, A. Beware of finer-grained origins. In *Web 2.0 Security and Privacy, 2008*.

[32] Karlof, C., Shankar, U., Tygar, J. D., and Wagner, D. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS, 2007*.

[33] Kim, T. H.-J., Huang, L.-S., Perrig, A., Jackson, C., and Gligor, V. Accountable Key Infrastructure: A proposal for a public-key validation infrastructure. In *WWW, 2013*.

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

[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`.

[36] Laurie, B., Langley, A., and Kasper, E. Certificate transparency (RFC 6992). `http://tools.ietf.org/html/rfc6962`, 2013.

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

[38] Marlinspike, M., and Perrin, T. Trust Assertions for Certificate Keys (TACK) (IETF Internet-Draft). `http://tack.io/draft.html`, 2013.

[39] Mozilla Developer Network. Mixed content. `https://developer.mozilla.org/en-US/docs/Security/MixedContent`.

[40] Mozilla Developer Network. Same-origin policy. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript`.

[41] 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*.

[42] 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 (2006), 2238–2246.

[43] Oppliger, R., Hauser, R., and Basin, D. SSL/TLS session-aware user authentication revisited. *Computers & Security 27*, 3-4 (2008), 64–70.

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

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

[46] Paola, S. D., and Fedon, G. Subverting Ajax. 23rd Chaos Communication Congress, 2006.

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

[48] 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`.

[49] Soghoian, C., and Stamm, S. Certified lies: Detecting and defeating government interception attacks against SSL. In *FC, 2011*.

[50] Sterne, B., and Barth, A. Content Security Policy 1.0 (W3C Candidate Recommendation). `http://www.w3.org/TR/CSP/`.

[51] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., and Cranor, L. F. Crying wolf: An empirical study of SSL warning effectiveness. In *USENIX Security, 2009*.

[52] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX ATC, 2008*.