Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS

Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute

***Abstract*—TLS was designed as a transparent channel abstraction to allow developers with no cryptographic expertise to protect their application against attackers that may control some clients, some servers, and may have the capability to tamper with network connections. However, the security guarantees of TLS fall short of those of a secure channel, leading to a variety of attacks.**

We show how some widespread false beliefs about these guarantees can be exploited to attack popular applications and defeat several standard authentication methods that rely too naively on TLS. We present new client impersonation attacks against TLS renegotiations, wireless networks, challenge-response protocols, and channel-bound cookies. Our attacks exploit combinations of RSA and Diffie-Hellman key exchange, session resumption, and renegotiation to bypass many recent countermeasures. We also demonstrate new ways to exploit known weaknesses of HTTP over TLS. We investigate the root causes for these attacks and propose new countermeasures. At the protocol level, we design and implement two new TLS extensions that strengthen the authentication guarantees of the handshake. At the application level, we develop an exemplary HTTPS client library that implements several mitigations, on top of a previously verified TLS implementation, and verify that their composition provides strong, simple application security.

I. Transparent Transport Layer Security?

TLS is the main Internet Standard for secure communications and still, after 20 years of practice, the security it provides to applications remains problematic.

I-A Applications vs Protocols.

By design, TLS intends to provide a drop-in replacement of the basic networking functions, such as `connect`, `accept`, `read` and `write`, that can effortlessly protect any application against a network attacker without the need to understand the protocol or its underlying cryptography. Pragmatically, TLS offers much flexibility, so the security properties provided by the protocol [43, 35, 32, 29] and its implementations [20, 14, 15] depend on how TLS is used. For instance, if the application enables an unsuitable ciphersuite [4], uses compression [25], or ignores state changes [45], it opens itself to attacks. Furthermore, applications-level security mechanisms increasingly seek to benefit from the underlying TLS connection by reusing its authenticated peer identities, key materials [48], and unique identifiers [6].

As a consequence, TLS libraries provide low-level APIs that expose many details of the cryptographic mechanisms and certificates negotiated during successive handshakes. Some application-level libraries, such as CURL, seek to recover the simplicity of a secure channel by implementing an abstraction layer that smooths over the details of TLS by managing sessions, validating certificates, etc. Meanwhile, TLS applications continue to rely on URLs, passwords, and cookies; they mix secure and insecure transports; and they often ignore lower-level signals such as handshake completion, session resumption, and truncated connections.

Many persistent problems can be blamed on a mismatch between the authentication guarantees expected by the application and those actually provided by TLS. To illustrate our point, we list below a few myths about those guarantees, which we debunk in this paper. Once a connection is established:

1) the principal at the other end cannot change;
2) the master secret is shared only between the two peers, so it can be used to derive fresh application-level keys;
3) the `tls-unique` channel binding [6] uniquely identifies the connection;
4) the connection authenticates the whole data stream, so it is safe to start processing application data as it arrives.

The first is widely believed to be ensured by the TLS renegotiation extension [49]. The second and third are used for man-in-the-middle protections in tunneled protocols like PEAP and some authentication modes in SASL and GSS-API. The fourth forms the basis of HTTPS sessions on the web.

These assumptions are false, and this enables various attacks, even against applications using the latest, fully-patched TLS 1.2 implementations. Whether these attacks should be blamed on the protocol or its usage, we argue that the transport and application protocols must be analyzed together to achieve reliable, meaningful, application-level security.

On the other hand, our paper does not challenge the cryptographic security of the core constructions of TLS—most of our attacks apply even under the (theoretical) assumption that clients and servers only use cryptographically strong ciphersuites, as formalized, for example, in [15, 35, 29, 16].

I-B New Attacks over TLS.

We report new practical attacks against applications that rely on TLS for their security. The first family of attacks uses a combination of successive TLS handshakes over multiple connections to disrupt client authentication. The second family of attacks targets HTTPS message integrity but may apply to other application protocols.

Triple Handshakes Considered Harmful (§V, §VI) We first point out *unknown key-share* [17] vulnerabilities in RSA, DHE, and abbreviated handshakes, and we compose them to implement a malicious TLS proxy that can synchronize the keys on separate connections with honest peers. These vulnerabilities do not in themselves constitute attacks on the integrity and confidentiality guarantees of TLS. However, we show that they enable new man-in-the-middle attacks that

break a variety of authentication mechanisms built over TLS, including (a) client-authenticated TLS renegotiation—for example if a client presents her certificate to two TLS servers, one can impersonate the client at the other; (b) compound authentication in tunneled protocols; (c) channel bindings for application-level challenge-response protocols; and (d) channel bindings for bearer tokens. We report concrete attacks against published specifications and popular applications in all these categories, including mainstream browsers and HTTP client libraries, VPN applications, wireless applications, and mail and chat servers.

Truncating Headers & Forcing Cookies (§III) Independently, we show that web browsers and servers often ignore TLS disconnections and tolerate ill-formed messages, thereby enabling *message truncations*. Although this vulnerability is generally known [13, 52], we show how to apply truncation to HTTP headers and HTML forms, opening new exploits. In particular, our attacks completely defeat cookie-based authentication. We also show new exploits based on known attack vectors like cookie-forcing and its use for login CSRF [12, 18]. In particular, we show that building new application-level protocols such as single sign-on and synchronization protocols using cookies is foolhardy; they amplify login CSRF attacks and enable network attackers to steal users' private files.

I-C Towards Verified Application Security. In light of the two families of attacks outlined above, how to ensure that a TLS application properly handles its interactions with the TLS API? How to reliably lift TLS security to application security? Broadly, we can either build countermeasures into TLS; or carefully implement and verify simpler security APIs over TLS; or, less robustly, promote prudent practices for writing secure applications over TLS.

Proposed TLS Extensions (§VII) One approach is to strengthen the protocol to provide more robust security. To this end, we propose two new TLS extensions that prevent the attacks of §VI without the need to change applications. These extensions have a negligible impact on performance and code complexity, and can be deployed while preserving backward compatibility. They apply to all protocol versions from SSL3 to TLS 1.2, as well as DTLS. To validate them experimentally, we implemented and tested patches for two existing TLS implementations: OpenSSL and miTLS. As future work, we plan to formally model their security benefits by extending the verified cryptographic model of miTLS [15, 16].

Simple Verified HTTPS over TLS (§VIII) In principle, carefully-written applications can defend against these attacks, without the need to change TLS. To validate our main recommendations, and show that "transparent" application security can indeed be achieved over TLS, we program miHTTPS: a simple HTTPS library in F#, similar to CURL, on top of miTLS. We specify its intended security properties and we verify them using F7, a type-based verification tool. Thus, we formally relate the precise, low-level TLS API provided by miTLS to a simpler, more abstract HTTPS API. In combination, we obtain the first cryptographically-verified implementation for HTTPS. In its current state, miHTTPS is a proof-of-concept: it does not nearly provide the flexibility required by modern browsers and web services. However, it automatically handles all the details of the underlying TLS connections, including multiple handshakes, resumption and negotiation, and truncations.

I-D Main Contributions. We describe a new class of man-in-the-middle attacks against authentication over TLS, targeting the resumption and renegotiation features of the handshake. We also present new exploits on HTTPS sessions based on cookie-forcing and truncation. We apply these attacks to break the expected authentication guarantees of several state-of-the-art protocols, libraries, applications, and web services. We have contacted many vendors, proposing immediate mitigations and countermeasures, as well as more long-term fixes to the corresponding protocol specifications. Our TLS-level proposals are consolidated in patches for OpenSSL and miTLS. We have also built and verified a basic high-level HTTPS API on top of miTLS, to validate our main application-level recommendations in a simplified setting.

Contents §II reviews the dangers of application security over TLS. §III illustrates these dangers by presenting new attacks caused by truncating HTTPS traffic and forcing cookies. §IV recalls the relevant protocol aspects of TLS. §V describes a malicious TLS proxy that synchronizes connections between TLS clients and servers. §VI presents new proxy-based attacks on applications that use client authentication. §VII discusses TLS countermeasures, implemented in OpenSSL and miTLS. §VIII illustrates application-level countermeasures, demonstrating a simple, provably secure HTTPS API on top of miTLS. §IX discusses impact, limitations and responsible disclosure of the attacks presented in this paper.

Online Materials An extended version of this paper, the two patches implementing our proposed countermeasures for OpenSSL and for miTLS, our verified implementation of miHTTPS and further experimental data are available online at https://secure-resumption.com.

II. TLS Interfaces and their Safe Usage

Modern clients and servers interact with TLS in ways far beyond the original intended interface. We discuss typical usages of the protocol, relevant to the attacks of §III and §VI.

II-A Session and Certificate Management. HTTP is by far the most widely used application protocol over TLS. Even the most basic HTTP operation, getting a file from a given URL, may require multiple connections to different servers due to redirections, authentication requests, temporary errors, and many other factors. Thus, any HTTPS client must manage and isolate multiple TLS sessions with different principals: if a client ever uses a cached session with the wrong server, the security guarantees of TLS collapse.

Similarly, any TLS application must implement a server certificate validation procedure, which can combine subject name and certificate purpose validation, pinning of certification authorities, trust on first use (TOFU), among others [22, 28]. Once again, any error in this process may completely void the security guarantees of TLS.

While session and certificate management are critical to the security of the protocol, they are implemented at the application level in contradiction to the network abstraction of TLS. Even when TLS libraries provide default functionality for these operations, they are not necessarily secure; for instance, OpenSSL shares the client-side session cache between all connections, even if they are to different hosts, unless it is explicitly partitioned by the application.

II-B EXPOSURE TO TLS EVENTS. Another recurrent problem with TLS APIs is the way they should expose transport-level events to the application. In this paper, we focus on two events that can lead to attacks if ignored by the application: renegotiation, and TCP connection closure.

Once a TLS connection is established, most applications typically only use `read`, `write` and `close`. How can a TLS library notify the application when renegotiation occurs? What if the cipher or the peer certificate changes? At best, the `read` primitive can return a non-fatal error code (like in GnuTLS) which the application can either ignore or use to enforce further checks on the new parameters. At worst, the change is only visible if the application keeps polling specific session parameters. To protect applications that ignore such events from man-in-the-middle attacks [45], most TLS libraries implement a protocol extension [49]. §VI-A shows how these applications can still be attacked despite this countermeasure.

Since SSL3, the closure of a connection must be notified to the other party with an authenticated protocol alert called `close_notify`. Without this graceful closure, a man-in-the-middle may have closed the TCP connection in the middle of a TLS connection. To make this distinction, TLS libraries should return a special error code when truncation is detected, signaling to the application not to process any partial data that may be buffered. However, in several implementations, the `read` primitive returns the number of bytes read, while error checking requires manual verification of a different parameter. Many applications do not distinguish between normal and unexpected closure, sometimes deliberately for compatibility.

Another class of problems appears when TLS is an optional feature of the application protocol, or if state is shared between encrypted and plaintext connections. §III illustrates how to exploit these issues against HTTP.

II-C CLIENT AUTHENTICATION. Applications can use various mechanisms for client authentication: client certificates (e.g. in browsers, for virtual private networks, and for wireless access points), bearer tokens (e.g. HTTP sessions cookies and OAuth access tokens), or challenge-responses protocols (e.g. HTTP digest authentication, and several SASL mechanisms used by mail and chat servers).

TLS client authentication is generally considered the safest, but is seldom used. Weaker mechanisms that rely on bearer tokens are more common, but they allow complete long-term impersonation of the user when a token is compromised. Challenge-response authentication within TLS tunnels offers better protection, but is still vulnerable to man-in-the-middle attacks [8, 41]: if the user is willing to authenticate on a server controlled by the attacker, the attacker can forward a challenge from a different server to impersonate the user at that server.

To address the shortcomings of authentication at the application level, new solutions have been recently proposed to expose values taken from the TLS handshake to applications in order to *bind* their bearer tokens and challenge-response protocols to the underlying TLS channel. Hence, tunneled wireless protocols like PEAP [42] use compound authentication schemes [44] to protect against rogue access points. SASL mechanisms like SCRAM [39] use TLS channel bindings [6], in particular the `tls-unique` binding, to prevent man-in-the-middle attacks even on anonymous TLS connections. Channel ID [10], a follow up to Origin-Bound Certificates [24], proposes that the client generate a long-lived pair of keys associated with each top-level domain it connects to. The public key is treated as a client identifier and, by binding bearer tokens such as cookies to this public key, the server can ensure they can only be used by the client they have been issued for, thus mitigating token compromise. §VI studies the assumptions such mechanisms make about TLS and presents attacks on a number of them.

III. TRANSPORT-LAYER ATTACKS ON HTTPS

As a case study of the API problems of §II, we consider the use of HTTP over TLS [47]. In HTTP, messages consist of two parts: the headers and an optional body, separated by an empty line. Headers consist of colon-separated name-value pairs, each terminated by a line break. The first header line is special: in requests, it contains the method (either `GET` or `POST`), path, and protocol version; in responses, it contains the protocol version, status code, and status message. The HTTP body is formatted according to the headers: by default, its length is specified in the `Content-Length` header; if the `Content-Transfer-Encoding` header is set to `chunked`, the body is a sequence of fragments, each prefixed by the fragment length, terminated by an empty fragment.

Due to the variety of (not necessarily correct) HTTP implementations, most clients are very permissive when parsing HTTP. For instance, they often accept message bodies whose length does not match the one indicated in the headers, or missing the last empty fragment in the `chunked` encoding.

For authentication, almost all websites rely on cookies, which are name-value pairs set by servers in the `Set-Cookie` header and sent back by clients in the `Cookie` header of subsequent requests. The cookie store is shared between HTTP and HTTPS connections, opening up a variety of attacks.

III-A COOKIE INTEGRITY. Modern web security policies are expressed in terms of *origin*, i.e., the combination of protocol, domain and port. Hence, HTTP requests and JavaScript interactions are unrestricted within the same origin, and strictly regulated across different origins [57]. In contrast, cookie policies rely on domain and path; furthermore, cookies may be set for any domain suffix and path prefix of the current page, e.g. `http://y.x.com/a` can set cookies with domain `x.com` and path `/`. This discrepancy causes major problems:

- *Protocol*: since there is no separation between HTTP and HTTPS, by default, cookies set on encrypted connections are also attached to plaintext requests, in plain sight of the attacker. To prevent this, the `secure` flag can be sent when setting the cookie to indicate to the browser

never to send this cookie unencrypted. This protects the confidentiality of cookies, but not their integrity, as it still possible to overwrite `secure` cookies over HTTP.
- *Domain*: domains prefixed with a dot will match any subdomain. Thus, a request to `a.x.com` attaches cookies set for `.x.com`, but not those set for `b.x.com`. A page may set cookies on any of its own domain suffix that is not a public (such as "com" or "co.uk"), leading to related-domain attacks.
- *Port*: since the port number is ignored, and even if a website is only served over TLS, an attacker can still use some unencrypted port to tamper with its cookies.

Cookies with the same name but different domain or path are stored separately; all matching cookies are sent back in the `Cookie` header in an unspecified order. Finally, there is a limit on the number of cookies that can be stored for each top-level domain name (e.g. `x.co.uk`). Beyond this limit, typically around 1000, older cookies are automatically deleted. Thus, an attacker can reliably delete legitimately set cookies.

Cookie forcing, *cookie fixation*, and *cookie tossing* all refer to tampering with cookies, either from the network or from a related subdomain. These issues have been well known for years, and many proposals address them [12, 18, 24], but there is still no way to defend against cookie forcing by a network attacker that works on all current browsers. Experimentally, we were able to force sessions on the top 10 Alexa websites in the US, despite the mitigations deployed on some of them.

Worse, the impact of such forcing attacks has increased considerably recently. For instance, many websites rely on single sign-on services for authentication. If the session on the identity provider (such as Facebook, Twitter or Google) is replaced with the attacker's, the victim may unwittingly associate his accounts on many websites with the attacker's identity, even after leaving the attacker's network. Furthermore, in modern websites, many operations are performed asynchronously. Thus, if a session is forced onto the browser before such an action, it may be associated with the attacker's account without any feedback to the user. Finally, some browsers rely on web login forms to provide features such as synchronization of tabs, bookmarks and stored passwords. We found that login CSRF attacks could trigger such features; even though a user confirmation dialog is shown with the account name of the forced session, it provides a tempting phishing target.

III-B THE COOKIE CUTTER ATTACK. As discussed in §II, most HTTP software does not enforce proper TLS termination, letting the attacker truncate a message at any TLS-fragment boundary by closing the underlying TCP connection. If the attacker controls the length of some of the contents of the message, he may chose a specific truncation point. Although this pattern has been exploited before to delete entire HTTP requests or to truncate message bodies [13, 52], we demonstrate new truncation attacks *within headers* of HTTP messages.

A network attacker can trigger a request with any path and parameters (in fact, any website can trigger such requests to any other website) and inject data into its `Cookie` header using forcing techniques, thus controlling the TLS fragmentation of the request. In response headers, when a redirection occurs, for instance after a successful login, the new URL given in the `Location` header typically includes parameters taken from the request (e.g., the page the user was trying to access before logging in). Such parameters are often under attacker control, and allow targeted truncation in response headers as well.

TABLE I. TLS TRUNCATION IN BROWSERS

	In-Header truncation	Content-Length ignored	Missing last chunked fragment ignored
Android 4.2.2 Browser	✓	✓	✓
Android Chrome 27	✓	✓	✓
Android Chrome 28	✗	✗	✓
Android Firefox 24	✗	✓	✓
Safari Mobile 7.0.2	✓	✓	✓
Opera Classic 12.1	✓	✓	✓
Internet Explorer 10	✗	✓	✓

Truncating Responses Recall that browsers do not attach cookies set with the `secure` flag to HTTP requests. In the `Set-Cookie` header, however, the flag occurs *after* the cookie, so the attacker can selectively truncate it and redirect the user to an unencrypted URL to recover the cookie value. Concretely, consider a login form at `https://x.com/login?go=P` that sets a session cookie and redirects the user to `https://x.com/P`. The headers of the response are as follows:

```
HTTP/1.1 302 Redirect
Location: https://x.com/P
Set-Cookie: SID=[AuthenticationToken]; secure
Content-Length: 0
```

The attacker can chose `P` such that the first TLS fragment ends just before '`;`' and close the connection before the second fragment is sent, allowing the cookie to be stored without the secure flag (and thus, visible to the attacker over HTTP). We successfully mounted this attack against Google Accounts.

The attack is possible because some browsers, including Chrome, Opera, and Safari, accepted incomplete HTTP responses (missing an empty line at the end of headers). We reported the vulnerability to each vendor; their responses are given in §IX-A. Table I summarizes the possible truncations in current browsers; we focus on mobile versions because they are more likely to connect to untrusted networks. While header-truncation attacks have mostly been fixed, chunked-body-truncation attacks remain possible on HTML and JavaScript.

Truncating Requests While most servers do not accept truncated headers, some do accept a truncated body. In the case of `POST` requests, typically used when submitting a form, the parameters are sent in the body of the request. This is most notably the case of requests sent through Apache SAPI modules, such as PHP. The main difficulty when truncating a `POST` request is to guess the length of the body parameters, which may be difficult since they often contain user input.

Consider a scenario where the victim invites one of her friend `bob@domain.com` on a social network where the attacker wants to access her profile. The attacker registers the domain `domain.co` and monitors the victim as she accesses the invitation page (for instance, by inspecting the length of the returned page). The query to truncate is of the form:

```
POST /invite.php HTTP/1.1
Host: socialnetwork.com
Content-Type: application/x-www-form-urlencoded
Cookie: SID=X; ForcedByAttacker=Z
Content-Length: 64

csrf_token=Y&invite=bob@domain.com
```

When the query is sent, the attacker truncates it such that the invitation will be sent to `bob@domain.co`. The victim gets a blank page due to the truncation, and may try the request again. Meanwhile, the attacker receives credentials to access the victim's profile. We were able to mount this attack on a popular social network that uses Apache and PHP.

III-C TLS CONNECTION INTEGRITY. Because most users connect to websites using plain HTTP, even if a website redirects all unencrypted connections to HTTPS, it is easy for a man in the middle to forward HTTPS contents over HTTP to the user, rewriting all links and pointers to encrypted pages. This attack, called SSL stripping [37], is very popular thanks to simple tools to mount it on public wireless networks.

To protect against SSL stripping, several browsers support HTTP Strict Transport Security [30] (HSTS), which introduces a `Strict-Transport-Security` header for websites to indicate that the browser should always connect to its domain over TLS, regardless of the port. The header includes a `max-age` value, specifying how long this indication should be enforced, and an optional `includeSubDomains` flag, indicating that the policy also applies to all subdomains.

HSTS has several known weaknesses. The first problem is bootstrapping: the user may use HTTP the first time it connects to the website, before receiving the HSTS header in the response. This bootstrapping problem is typically mitigated by browsers that use a pre-registered HSTS domain list for sensitive websites that wish to opt-in to this feature.

Second, HSTS preserves cookie integrity only when enabled on the top level domain with the `includeSubDomains` flag, and if the user visits this domain first [18]. This is an expensive requirement for large websites, as it forces all contents for the entire domain to be served over HTTPS. We found that not a single website from the top 10,000 Alexa list is using the `includeSubDomains` option on their top-level domain, even though some are indeed using HSTS. Thus, in practice, HSTS is not used to prevent cookie forcing attacks.

We found a new attack to bypass HSTS on some clients. A network attacker can truncate the `Strict-Transport-Security` header after the first digit of the `max-age` parameter. If the client accepts and processes this header, the HSTS entry for that website will expire after at most ten seconds, after which HTTP connections to the domain will be allowed again, *even if* the domain has pre-registered to the HSTS domain list on the browser.

Concretely, to attack `x.com`, the man-in-the-middle takes any HTTP request for any server and redirects it to a page on `x.com` that returns a parameter-dependent `Location` header followed by the `Strict-Transport-Security` header. We successfully tested the attack on Chrome, Opera, and Safari. We further note that by using this attack first, a network attacker can re-enable SSL stripping, cookie forcing, and the cookie `secure` flag truncation attack above even on websites that enable HSTS, defeating the purpose of this standard.

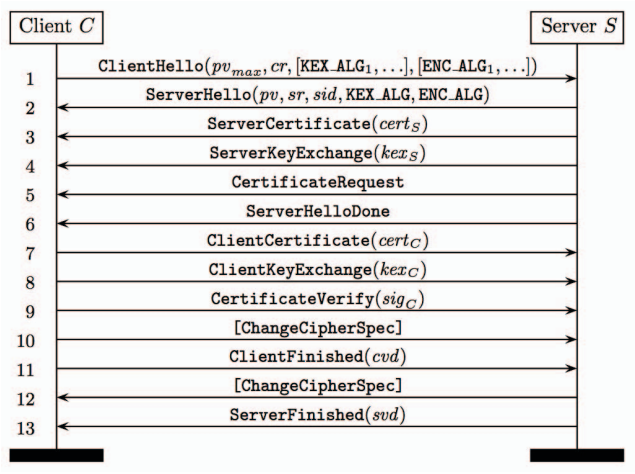

Fig. 1. The TLS Handshake

For websites that do not deploy HSTS, browser extensions have been developed to force the use of HTTPS on a given list of websites. However, it is worth noting that such ad hoc mechanisms have their own flaws. For example, HTTPS Everywhere [2] allows HTTP connections when the server port is non-standard. Cookie policies ignore the port number, so various attacks like cookie forcing remain possible.

IV. TLS PROTOCOL: CONNECTIONS, SESSIONS, EPOCHS

The TLS protocol is commonly used over TCP connections to provide confidentiality and integrity for the bytestreams exchanged between a client (C) and a server (S). Next, we recall the main subprotocols of TLS and the attacks directly relevant to this paper. (The online version discusses other prior attacks on handshake integrity.) We assume some familiarity with TLS; we refer to the standard [23] for the details and to other papers for a discussion of previous proofs [35, 43] and attacks [40, 22].

IV-A FULL HANDSHAKE. Once a TCP connection has been established between a client and a server, the TLS handshake protocol begins. The goals of the handshake are to authenticate the server and (optionally) the client; to negotiate protocol versions, ciphersuites, and extensions; to derive authenticated encryption keys for the connection; and to ensure agreement on all negotiated parameters. (A ciphersuite selects a key exchange mechanism `KEX_ALG` for the handshake and an authenticated encryption mechanism `ENC_ALG` for the record protocol.)

Figure 1 shows the full handshake with mutual authentication. First, the client sends a client hello message with a maximum protocol version pv_{max}, a random nonce cr, and a set of proposed ciphersuites and extensions. The server chooses a version pv, a ciphersuite, and a subset of these extensions, and responds with its own nonce sr and a session

identifier sid. The server then sends its X.509 certificate chain $cert_S$ and public key pk_S. Depending on KEX_ALG, it may send additional key materials in a key exchange message kex_S. It may also send a certificate request message if it requires client authentication.

The client responds with its own certificate chain $cert_C$ and public key pk_C (if required), followed by its own key exchange message kex_C. If the client sends its certificate, it also sends a signed hash sig_C of the current log (log_{1-8}, obtained by concatenating messages 1–8) in a certificate verify message.

At this point in the protocol, both the client and the server can compute a shared pre-master secret pms from kex_C and kex_S, then use pms along with the nonces to derive a master secret ms, and use ms to derive keys for the connection and to verify the handshake integrity; these computations are detailed below. To complete the handshake, the client signals a change of keys with a change cipher spec (CCS) message followed by a finished message that contains the client verify data cvd obtained by MACing the current handshake log (log_{1-9}) with key ms. Similarly, the server sends its own CCS and a finished message that contains the server verify data svd, obtained by MACing the whole handshake $log_{1-9,11}$. (The CCS messages are not included in the logs.)

When the client is not authenticated, messages 5, 7, 9 are omitted. When the server does not contribute to the key exchange, e.g. with RSA, message 4 is omitted.

RSA Handshake If the key exchange in the negotiated ciphersuite is RSA, the calculations go as follows, where log_{1-8} is the log before message 9, log_{1-9} is the log before message 11, and $log_{1-9,11}$ is the log before message 13. (The server key exchange value kex_S is not used.)

$$
\begin{aligned}
pms &= [pv_{max}]\|[\text{46 bytes randomly generated by } C] \\
sig_C &= \mathsf{signed}(sk_C, log_{1-8}) \\
kex_C &= \mathsf{rsa}(pk_S, pms) \\
ms &= \mathsf{prf}(pms, \text{“}\texttt{master secret}\text{”}, cr|sr) \\
keys &= \mathsf{prf}(ms, \text{“}\texttt{key expansion}\text{”}, sr|cr) \\
cvd &= \mathsf{prf}(ms, \text{“}\texttt{client finished}\text{”}, \mathsf{hash}(log_{1-9})) \\
svd &= \mathsf{prf}(ms, \text{“}\texttt{server finished}\text{”}, \mathsf{hash}(log_{1-9,11}))
\end{aligned}
$$

DHE Handshake If the negotiated key exchange is ephemeral Diffie-Hellman (DHE), then S chooses group parameters (p, g) and a fresh key pair (K_S, g^{K_S}); it sends (p, g, g^{K_S}) in kex_S, signed along with cr and sr with its private key sk_S. The client generates its own key pair (K_C, g^{K_C}) and responds with $kex_C = g^{K_C}$. Both parties compute $pms = g^{K_C * K_S}$. The rest of the computations are the same.

$$
\begin{aligned}
kex_S &= \mathsf{signed}(sk_S, cr|sr|p|g|g^{K_S} \bmod p) \\
kex_C &= g^{K_C} \bmod p \\
pms &= g^{K_C * K_S} \bmod p \text{ (with leading 0s stripped)}
\end{aligned}
$$

Other variations Besides RSA and DHE, mainstream TLS implementations support variations of the Diffie-Hellman key exchange implemented using elliptic curves. The handshake for these is similar to DHE, but with some notable differences. For example, most ECDHE implementations only accept named curves within a fixed set, whereas DHE allows the server to choose arbitrary DH group parameters.

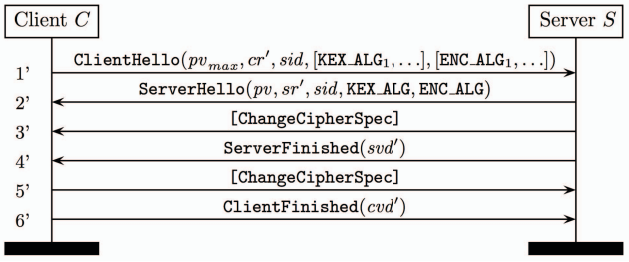

Fig. 2. Abbreviated TLS Handshake

Other key exchanges are less common on the web but useful in other applications. In TLS-PSK, the client and server authenticate one another using a pre-shared key instead of certificates. In TLS-SRP, the client uses a low-entropy password instead of a certificate. In DH_anon, both client and server remain anonymous, so the connection is protected from passive eavesdroppers but not from man-in-the-middle attackers.

IV-B THE RECORD PROTOCOL. Once established, a TLS connection provides two independent channels, one in each direction; the record protocol protects data on these two channels, using the authenticated-encryption scheme and keys provided by the handshake. Application data is split into a stream of fragments that are delivered in-order. There is no correlation (at the TLS level) between the two directions. When the client or server wishes to terminate the connection, it sends a `close_notify` alert to signal the end of its writing stream, and it may wait for the peer's `close_notify` before closing the connection. If both peers perform this graceful closure, they can both be sure that they received all data. However, this is seldom the case in practice.

There are several attacks on the confidentiality of the record protocol [e.g. 5]; attacks on integrity are less common [e.g. 15].

IV-C SESSION RESUMPTION. Full handshakes involve multiple round-trips, public key operations, and (possibly) certificate-revocation checks, increasing latency and server load [53]. In addition, abbreviated handshakes enable clients and servers that have already established a session to quickly set up new connections. Instead of establishing a new master secret, both parties reuse the master secret from that recent session (cached on both ends), as shown in Figure 2.

The format of the cached session data depends on the TLS implementation, but [50] recommends that it contains at least the master secret, protocol version, ciphersuite, and compression method, along with any certificate used.

The client sends a client hello, requesting the server to resume the session sid, with a new client nonce cr'. If the server has cached this session, it may then respond with a server hello with a new server nonce sr' and the same sid and algorithms as in the initial handshake. The server then immediately sends its CCS and finished message, computed as a MAC for the abbreviated handshake log. The client responds with its own CCS and finished message, computed as a MAC of the whole resumption log. The computation of

keys and verify data are as follows, where log'_{1-2} consists of the messages 1' and 2', while $log'_{1-2,4}$ includes 1', 2' and 4':

$$\begin{aligned} ms &= [\text{cached for } (S, sid)] \\ keys &= \mathsf{prf}(ms, \text{“}\texttt{key expansion}\text{”}, sr'|cr') \\ svd &= \mathsf{prf}(ms, \text{“}\texttt{server finished}\text{”}, \mathsf{hash}(log'_{1-2})) \\ cvd &= \mathsf{prf}(ms, \text{“}\texttt{client finished}\text{”}, \mathsf{hash}(log'_{1-2,4})) \end{aligned}$$

The completion of an abbreviated handshake implicitly confirms to each participant that they share the same session master secret. Hence, if both peers are honest, they must have matching session parameters—those negotiated in the initial handshake. Because of its efficiency, resumption is aggressively used on TLS connections. It is supported by default in all major web browsers and web servers. A recent TLS extension enables servers to store their cached sessions at the client within encrypted tickets [50]; this mechanism makes it possible for clients to maintain long-lived sessions with stateless server farms, at little cost to the servers.

We use the term session resumption when the same TLS session is used on multiple connections, but the abbreviated handshake may also be used on an existing TLS connection to refresh keys and reset sequence numbers. At the end of each handshake, we say that the connection enters a new *epoch*.

IV-D RENEGOTIATION: CHANGING EPOCHS. A client or a server may request a new handshake on an established TLS connection, e.g. to renegotiate the session parameters. The handshake proceeds as described above, except that its messages are exchanged on the encrypted TLS connection. When the handshake completes, both parties share a new session, and their connection enters a new epoch, switching to the keys derived from the new session.

There are many reasons why an application may want to renegotiate a TLS session when it already has a working TLS connection. The first is client authentication. On some servers, client authentication is required only when accessing protected resources. For instance, Apache triggers renegotiation and requires a client certificate on first access to a protected directory. This design improves user experience and helps protect privacy by requesting authentication only when needed, and prevents the client certificate being sent in the clear during the initial handshake. Other reasons may be to upgrade the ciphersuite or replace an expiring certificate [49, §5]. Even in this case, the server may need to provide a new certificate that supports, say, ECDSA signing instead of RSA. Consequently, in many renegotiations, the client and server certificates and identities after renegotiation may differ from those of the previous handshake. Without additional protections, such identity changes can lead to impersonation attacks.

Renegotiation Attack Protecting the renegotiation under the keys of the previous handshake is not enough to prevent man-in-the-middle attacks. An active network attacker can intercept an initial handshake from a client to a server and forward it as a renegotiation within an existing TLS connection between the attacker and the server. As a result, any data that the attacker sent before the renegotiation gets attributed to the client, leading to a powerful impersonation attack [45].

In response to this attack, a new 'mandatory' TLS extension has been proposed and deployed for all versions of TLS [49]. This extension includes the verify data of the previous handshake within the client and server hello messages of the renegotiation handshake, thereby cryptographically binding the two handshakes (and, recursively, any preceding handshake on the same connection). As a result, as each handshake completes, both peers can be confident that they agree on all epochs on their connection. Informally, the principals at each endpoint must remain the same, even if the certificates change.

As shown in §V, this countermeasure still does not suffice to eliminate renegotiation attacks across *several* connections.

IV-E IMPLEMENTATIONS AND APIS. There are several popular implementations of TLS, including OpenSSL, GnuTLS, NSS, JSSE, and SChannel. Here, we briefly discuss the miTLS verified reference implementation [15], whose API is distinctive in the detailed connection information that it offers to its applications. As such, miTLS is an ideal experimental tool on which to evaluate attacks and implement countermeasures.

The miTLS API consists of functions to initiate and accept connections, send and receive data, and instigate session resumption, re-keying, and renegotiation. Each of these functions returns a connection handle and a *ConnectionInfo* structure, which details the current epoch in each direction (they can differ). For each epoch, it includes the nonces and verify data and points to a *SessionInfo* structure with the epoch's session parameters (including ciphersuites and peer identities). It also points to the previous epochs on the connection (if any).

The API encodes the security assumptions and guarantees of TLS as pre- and post-conditions on the connection state. The application cannot send or receive data unless the connection is in the *Open* state, which means that a handshake has successfully completed with an authorized peer. When a handshake completes at an endpoint, the API guarantees that, if all the principals mentioned in the *ConnectionInfo* are honest, then there is exactly one other endpoint that has a matching *ConnectionInfo* and keys. Every application data fragment sent or received is indexed by the epoch it was sent on, which means that miTLS will never confuse or concatenate two data fragments that were received on different epochs; it is left to the application to decide whether to combine them. If the connection uses the renegotiation indication extension, the application gets an additional guarantee that the new epoch is linked to the old epoch. If at any point in a connection, miTLS receives a fatal alert or raises an error, the connection is no longer usable for reading or writing data. If the connection is gracefully closed, miTLS guarantees that each endpoint has received the entire data stream sent by its peer. Otherwise, it only guarantees that a prefix of the stream has been received.

V. A MAN-IN-THE-MIDDLE TLS PROXY SERVER

We consider the following scenario. Suppose an honest TLS client C connects to a TLS server A that is controlled by the attacker. A then connects to an honest TLS server S, and acts as a man-in-the-middle proxy between C and S, ferrying data between C and S across the two independent connections. Of course, A can still read and tamper with selected fragments.

Now, suppose that A establishes the *same keys* on both TLS connections. We will show in this section how A can achieve this. Then A does not have to decrypt and reencrypt traffic between the two connections and may instead step out of the way, allowing C and S to talk directly to one another, making A's intervention difficult to detect even with sophisticated timing measurements [9].

On its own, the scenario above does not constitute a serious attack on either connection, since both C and S are aware that they are connected to A. However, the ability of A to synchronize keys across two connections can be a stepping stone towards more dangerous attacks, as we will show in §VI.

In the cryptographic key-exchange literature, this kind of key synchronization is called an unknown key-share attack [17, 34], whereby two honest parties share a key but one of them does not realize with whom it shares its key; their mutual belief in the shared secret is violated [54]. In Abadi's terminology [3], these attacks do not disrupt any access control goals based on *responsibility*, but they enable an attacker to take *credit* for an honest principal's message. So, if the application that uses the protocol does not reliably confirm both peers' identities, impersonation attacks may appear [36].

In the rest of this section, we show how a malicious server A can synchronize TLS keys with C and S. To build this malicious server, we exploit three independent weaknesses in the RSA handshake, the DHE handshake, and the abbreviated handshake. We do not make any assumption about application behavior, and use only standard mechanisms implemented by mainstream TLS libraries.

V-A SYNCHRONIZING RSA. Suppose C sends a client hello to A offering an RSA ciphersuite. A then forwards the client hello to S. When S responds with the server hello, A forwards it to C. Hence, the client and server nonces cr , sr and the session identifier sid are the same for both connections.

Next, when S sends its certificate $cert_S$ to A, A instead sends its own certificate $cert_A$ to C. Now, C generates a pre-master secret pms, encrypts it under pk_A, and sends it to A. A decrypts pms, re-encrypts it under pk_S, and sends it to S. Hence, both connections have the same pms and (since the nonces are equal) the same master secret and connection keys, all of which are now shared between C, S, and A. Finally, A completes the handshake on both connections, using ms to compute correct verify data. The messages tampered by A are illustrated in Figure 3 (Connection 1).

At this point, C and S cache the same session that they both associate with A (as represented by $cert_A$ on C, and optionally, A's client certificate on S). The new epochs on the two connections are distinguishable only by the client and server verify data, which differ on the two connections. However, messages from one connection can be freely forwarded to the other, since the keys match. Consequently, if A stepped out of the way, C and S can continue exchanging messages without realizing that the principal on the other end has changed.

Variants and Mitigations The above trace is robust to variations in the key exchange. If S demands a client certificate, A can provide its own certificate, and this does not affect the synchronization of the master secret or connection keys. If both

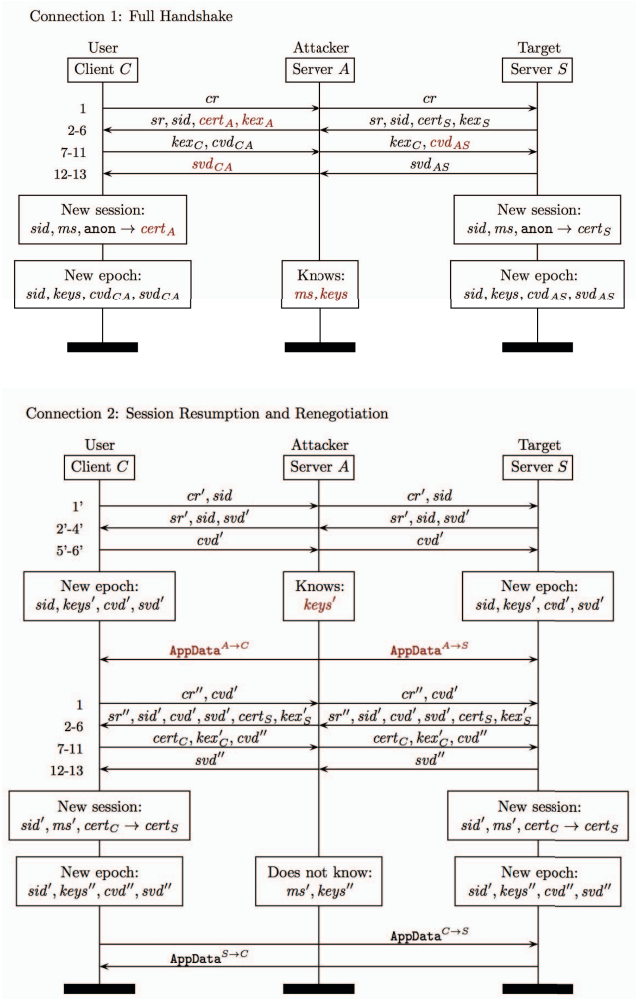

Fig. 3. Triple handshake attack by a malicious server on client-authenticated TLS renegotiation: (1) RSA/DHE full handshake, (2) abbreviated handshake for session resumption, (3) secure (RFC 5746 [49]) renegotiation handshake

C and S support RSA but prefer a different key exchange, say ECDHE, A can still force them both to use RSA by offering only RSA in its client and server hellos.

The RSA key exchange does not ensure different keys on different connections, and there is no standard mitigations that implementations can employ to prevent it. This behavior would not surprise a cryptographer or protocol expert, since only C contributes to the key exchange. However, it is only occasionally mentioned in protocol specifications [48, §5] and continues to surprise protocol designers. As shown in §VI, such connection synchronizations can defeat the man-in-the-middle protection used in tunneled protocols like PEAP.

V-B SYNCHRONIZING DHE. Suppose that C (or S) refuses RSA ciphersuites, but accepts some DHE ciphersuite. We show that A can still synchronize the two connections, because the DHE key exchange allows the server to pick and sign arbitrary Diffie-Hellman group parameters, and any client that

accepts the server certificate and signature implicitly trusts those parameters.

In this scenario, A substitutes its own certificate for S's (as with RSA), then changes the Diffie-Hellman group parameters in the server key exchange message, and finally changes the client's public key in the client key exchange message.

Suppose S offers a prime p, generator g, and public key $P_S = g^{K_S} \bmod p$. A replaces p with the non-prime value $p' = P_S(P_S - 1)$ and signs the parameters with its own private key. When C sends its own key exchange message with public key $P_C = g^{K_C} \bmod p'$, the attacker replaces it with the public key g and sends it to S. Our choice of p' ensures that P_S has order 1 in the group $Z^*_{p'}$, or equivalently $\forall x > 0, P_S^x = P_S \bmod p'$. Other values of the form $p' = q(P_S - 1)$ also lead to P_S having a low order in $Z^*_{p'}$. Upon receiving this message, C computes

$$\begin{aligned} pms &= P_S^{K_C} \bmod P_S(P_S - 1) \\ &= P_S \bmod P_S(P_S - 1) \\ &= P_S \text{ (with leading 0s stripped)} \end{aligned}$$

while S computes $pms = g^{K_S} \bmod p = P_S$. Finally, both connections share the same pms, ms, and derived keys.

Variants and Mitigations The authenticated Diffie-Hellman key exchange is not intrinsically vulnerable to a man-in-the-middle, as long as both parties use the same, well chosen group. The key to this attack is that the attacker is able to make C accept a group with a non-prime order. In fact, p' above is always even (and may cause errors with implementations that rely on Montgomery reduction for modular exponentiation) but it is easy to find odd non-primes that work just as well.

The attack fails if C checks that p' is prime. Yet, none of the mainstream TLS implementations perform a full primality check because it is deemed too expensive. A probabilistic primality check could help, but may not guarantee that the attacker cannot find a p' that defeats it. An alternative mitigation would be to standardize a few known good Diffie-Hellman groups for use in TLS. Indeed, this is the approach taken in protocols like IKEv2 and in TLS variants like SRP.

Even when clients and servers use known groups, care must be taken to validate the public key received from the peer. Otherwise, they may become vulnerable to *small subgroup* attacks [see e.g. 7, 46] which have been exploited in previous TLS attacks [55, 38]. Barker et al. [11] define a procedure for checking public keys, but we found that many TLS implementations do not implement it. We analyzed TLS clients and servers to check whether they accept degenerate public keys (with small orders) like 0, 1, and -1; these keys always lead to $pms \in \{0, 1, -1\}$. While 0 and 1 are rejected by most implementation (to mitigate [38]), we found that NSS, SChannel, and JSSE do accept -1. On the web, we found that all web browsers and about 12% of DHE-enabled servers of the top 10,000 Alexa list also accept -1. Such clients and servers are vulnerable to our key synchronization attack, since the pms can be forced to be the same on both connections (with high probability), even if these clients and servers only accept known primes and correctly sample their keys.

The elliptic curve version of DHE (ECDHE) allows servers to offer arbitrary curves, and so theoretically suffers from the same attack, but all the TLS implementations we tested only support well-known named curves standardized by NIST.

V-C Synchronizing Abbreviated Handshakes. Suppose C, A, and S have synchronized sessions and connections, as described above. If C attempts to resume the session with A over a new connection, A can then synchronize this new connection with a new connection to S. In fact, abbreviated handshakes are easier to synchronize than full handshakes.

When C sends its client hello requesting session resumption on a new connection, A simply forwards the request to S, and forwards S's response to C unchanged. C and S complete the handshake through A, re-using the master secret known to C, S, and A, as shown in the top half of Connection 2 in Figure 3. The resulting epochs on the two connections have the same keys, also shared with A. The new epochs are, in fact, more synchronized than the epochs on the original connection: the client and server verify data on these epochs are also the same. Hence, after resumption, the only noticeable difference between the two connections is that the C-A connection has a session with server identity $cert_A$ while the A-S connection has a session with server identity $cert_S$. All other differences have been erased. This is important for the attacks in §VI.

The ease with which resumed sessions can be synchronized exposes the weak authentication guarantees of the abbreviated handshake. It only ensures that the client and server share the same master secret, whereas applications may (and do) assume that they share the same session, which we show is not the case. To obtain stronger guarantees from this handshake, in §VII we propose a TLS extension, similar to [49], that links the resumption handshake to the original session.

VI. Attacks on Client Authentication over TLS

TLS is most commonly used in the anonymous-client mode, where only the server is authenticated. Consequently, applications often deploy their own mechanisms and protocols to authenticate users after the TLS handshake has finished.

Previous work shows that layering a client authentication protocol within a server-authenticated secure channel is vulnerable to generic man-in-the-middle attacks [8, 41]; Ray's renegotiation attack [45] is also an instance of this pattern. If an attacker A can see application-level protocol messages between C and S, it can tunnel these messages through its own connection with S, thereby impersonating C at S.

This attack is possible in three scenarios. First, if the client C uses the same application-level credentials on encrypted and unencrypted channels. Second, if C uses the same credentials on different servers, one of which could be malicious. Third, if C fails to correctly validate the server identity and confuses a malicious server A with an honest server S. In all these cases, the application-level protocol should guarantee that the credentials released by C to A cannot be used by A at S.

A common pattern to enforce this guarantee is to cryptographically bind the (inner) application authentication to the (outer) underlying TLS channel [8, 6, 49]. This binding helps only inasmuch as the inner protocol employs strong keys (public or secret) or a passphrase-based challenge-response scheme

resistant to dictionary attacks. Conversely, bearer tokens cannot be protected. In this section, we discuss four such binding mechanisms, and show how to break their guarantees using the synchronizing TLS proxy of §V.

VI-A THE TRIPLE HANDSHAKE ATTACK. Suppose the attacker A has an anonymous-client TLS connection to server S. When A tries to access a user-protected resource, S triggers a renegotiation to require A to authenticate as a valid user, with a client certificate or some other credential (PSK, SRP, etc.). This pattern is enabled, for example, on the Apache web server, when a client tries to access a protected directory.

A wants to authenticate to S as C (without C's credentials). More generally, even if A has previously authenticated to S, it wants to change its authenticated identity to C.

Before explaining our attack, it is useful to recall the 2009 renegotiation attack [45] and countermeasure [49], which cryptographically binds each handshake on a connection to the preceding one, by passing the verify data of the previous handshake (if there was one) in the client and server hellos of the new handshake. Therefore, if A initiates a full handshake with S, but later tries to forward C's handshake to S as a renegotiation, the verify data in C's hello would not match A's handshake, prompting the server to reject the renegotiation.

What if a session is resumed on a new connection? The first handshake now is an abbreviated handshake; it only authenticates the session master secret, not the whole session. Thus, the renegotiation countermeasure does nothing to bind the new connection to the old session. This re-enables the man-in-the-middle impersonation attack it was meant to fix.

Assume the adversary A has set up synchronized sessions and connections with C and S. If C resumes the session on a new connection, A can resume the same session on a new connection to S. As discussed in §V-C, at the end of the abbreviated handshake, the verify data on both connections is the same. Now, if C or S initiates a client-authenticated TLS renegotiation, A can simply forward all messages from C to S and back, making no changes. The client and server hellos will refer to the verify data from the abbreviated handshake and thus be accepted by both parties. This triple handshake across two connections is depicted in Figure 3.

At the end of the renegotiation, from TLS's viewpoint, C and S share a new mutually-authenticated session. A does not have the keys to this new session, but it may have injected data in both directions before the renegotiation, and this data may now be mistakenly attributed by C to S, and vice versa. In other words, the TLS peer on the connection has changed, and the application may not realize it, defeating the purpose of the secure renegotiation extension.

Preconditions and Variations The attack above works regardless of whether the renegotiation uses client certificates, PSK, or SRP to authenticate the client, and even if the initial handshake also used client authentication.

The main precondition is that the client be willing to use the same authentication credentials on A and S. This is reasonable for public-key certificates, which are often used as universal identity assertions when issued by trusted CAs. For SRP or PSK credentials, this may not seem as likely, but these key exchanges are typically used to provide both server and client authentication, and hence, they both offer several ciphersuites that do not use server certificates at all.

The second precondition is that the client and server should be willing to accept new mutual identities during renegotiation. Accepting a change of client identity (or client authentication on an anonymous session) is one of the purposes of renegotiation, but accepting a change of server may seem unusual. We experimentally tested a wide variety of TLS client applications, including mainstream browsers, popular HTTPS libraries such as CURL, serf, and neon, version control systems, VPN clients, mail clients, etc. We found that a vast majority of them *silently* accept a change of server identity during renegotiation, and thus are vulnerable to our impersonation attack.

Why does this not contradict proofs of the TLS handshake? Most proofs [e.g. 35, 32] ignore renegotiation and resumption; [14] supports resumption but not renegotiation; [29] considers renegotiation but not resumption; [15] supports both but relies on the application to correctly handle epoch changes.

Web Exploit and Mitigation As a concrete example, we implemented the above attack as a web server acting as a synchronizing proxy between a browser C and an honest website S. After proxying the initial handshake and session resumption, A can tamper with the connection in many ways, before instigating renegotiation:

- A can send a POST message to S which will get subsequently attributed to C after renegotiation.
- A can send a page with JavaScript to C, so that the script gets executed later, in the client-authenticated session.
- A can source a client-authenticated page from S in one frame at C while reading its contents from another frame sourced at A, bypassing the same origin policy (XSS).

All of these attacks can be used to subvert both user authentication on the server and same-origin protections on the browser. Protections like CSRF tokens and Content Security Policy do not help since the page's origin is no longer reliable.

We have disclosed this vulnerability to a number of browser vendors. The easiest mitigation is for web browsers to refuse a change of server identity during renegotiation (since their UI can hardly convey a HTTPS mashup of several origins); some of them have already made this change in response to our report. For web servers and other HTTPS applications, we believe that restricting peer certificate changes would be a good default as well, with a careful review of the UI and API design in the cases when the identity is expected to change.

VI-B BREAKING COMPOUND AUTHENTICATION IN TUNNELED PROTOCOLS. Wireless authentication protocols such as EAP-TLS [51], PEAP [42] and EAP-TTLS [27] are particularly susceptible to man-in-the-middle attacks even over TLS [8] because of the ease with which other wireless devices and rogue access points can fool naive clients into connecting to them [19]. To protect against such attacks, some of these protocols adopted new compound authentication mechanisms [44] that cryptographically bind the inner EAP authentication protocol with the outer TLS tunnel.

In PEAP, when the inner protocol is MSChapv2 [1] for example, the inner protocol generates a session key (`ISK`)

that is combined with a tunnel key (TK) generated from the outer TLS connection's master secret (and client and server randoms) to derive a compound authentication key (CMK) and encryption key (CSK) for subsequent use between the wireless device and access point. The idea is that these keys will only be known to devices that participated both in the outer TLS handshake and the inner EAP authentication.

$$\texttt{TK} = \mathsf{prf}(ms, \text{“}\texttt{client EAP encryption}\text{”}, cr|sr)$$
$$\texttt{CMK}|\texttt{CSK} = \mathsf{prf}'(\texttt{TK}, \texttt{ISK})$$

PEAP also features *fast reconnect*, an API for TLS session resumption: as it moves from one wireless access point to another and needs to reconnect, the client simply resumes its TLS session and skips the inner authentication protocol. In this case, ISK is set to 0s so the compound authentication and encryption keys depend only on TK. This mechanism presumes that the tunnel key is unique on every connection; our synchronizing TLS proxy breaks this assumption and leads to a new attack.

As usual, A sets up synchronized connections with C and S and forwards the untampered MSChapv2 exchange to let C authenticate to S, negotiate ISK, combine it with TK, and derive CMK and CSK. Since A only knows TK, he cannot read or tamper with any messages after the authentication. Nonetheless, if A uses fast reconnect to resume the TLS session with S, the inner EAP authentication is skipped, and the new compound keys are only derived from TK. Yet, S still associates the connection with C, resulting in a complete impersonation by A, without any involvement from C.

Preconditions and Mitigations To make the attack work, the malicious access point must convince the user to trust its certificate, which can be achieved in a number of cases [19].

The mitigation for tunneled protocols is not straightforward. At the TLS level, a more general mitigation would be to change the master secret computation, as we discuss in §VII. In PEAP, one possibility is to change the tunnel key computation to include the server's identity, represented by the server's certificate or its hash:

$$\texttt{TK} = \mathsf{prf}(ms, \text{“}\texttt{client EAP encryption}\text{”}, cr|sr|cert_S)$$

VI-C Breaking TLS Channel Bindings. Channel bindings [56] are a generic protocol composition mechanism, whereby a transport-level cryptographic protocol such as IPsec, SSH, or TLS can expose specific session and connection parameters to applications, most notably to bind authentication mechanisms to the underlying secure channel. Their stated goal is to establish that "no man-in-the-middle exists between two end-points that have been authenticated at one network layer but are using a secure channel at a lower network layer". TLS implementations expose three channel bindings to applications [6]; we consider one of them here and another (tls-server-end-point) in the online material. The 'tls-unique' channel binding for a given TLS connection is defined as the first finished message in the most recent handshake on the connection. If the most recent handshake is a full handshake, this value is the client verify data cvd; if it is an abbreviated handshake, it is the server verify data svd. The intent is that tls-unique be a unique representative of the current epoch, shared only between the two peers who established the epoch. Our synchronized session resumption breaks it by establishing different connections with honest peers that have the same tls-unique value.

To see how this can be concretely exploited, consider the SCRAM-SHA-1-PLUS protocol [39] used in the SASL and GSS-API families of authentication mechanisms in a variety of applications like messaging (XMPP), mail (SMTP, IMAP), and directory services (LDAP). SCRAM is a challenge-response protocol where the client and server store different keys ($\texttt{CK}_p, \texttt{SK}_p$) derived from a user's password (p), and use them to authenticate one another. When used over TLS, the first two messages contain client and server nonces and the tls-unique value for the underlying TLS connection. The last two messages contain MACs over these values, for authentication and channel binding:

1. $C \to S:$ $u, cn, \texttt{tls-unique}$
2. $S \to C:$ cn, sn, s, i
3. $C \to S:$ $cn, sn, \texttt{ClientProof}(\texttt{CK}_p, log_{1,2,3})$
4. $C \to S:$ $cn, sn, \texttt{ServerSignature}(\texttt{SK}_p, log_{1,2,3})$

In our attack, C establishes, then resumes a session with A, who synchronizes a connection with S to have the same tls-unique value. A then forwards the SCRAM messages between C and S. Since the server identity is not part of the exchange and the tls-unique values match, the SCRAM authentication succeeds, enabling A to impersonate C at S.

A precondition for the attack is that C be willing to accept A's certificate, and this is already considered a security risk for SCRAM-like protocols, since they then become vulnerable to dictionary attacks. However, the tls-unique protection is meant to protect users from impersonation even if the TLS protocol uses an anonymous key exchange [39, §9]. Our attack shows that this is not the case.

To prevent this attack without impacting TLS, we recommend significant changes to the specification of tls-unique in §VII. With such modifications, tls-unique may possibly become truly unique across connections.

VI-D Breaking Channel-Bound Tokens on the Web. Channel ID is a TLS extension [10], implemented by Chrome and all Google servers, that aims to bind web authentication tokens such as cookies to a cryptographic *channel* between a client and a server, without the need for client certificates. A channel can be long-lived (at least as long as cookies) and consists of many TLS sessions and connections. Channel ID is a follow-up to the previously published origin-bound certificates proposal of Dietz et al. [24], which was considered impractical to implement and deploy.

A TLS client that supports Channel ID generates and stores a public-private elliptic curve key pair ($pk_{cid,S}, sk_{cid,S}$) associated to each domain name S that it connects to. The TLS handshake is modified so that, instead of a client certificate and certificate verify message, the client sends a Channel ID authentication message that contains the public key (a point on the P-256 elliptic curve) and an ECDSA signature of the handshake log using the private key. To protect the privacy

of the client's public key from passive eavesdroppers, the authentication message is sent encrypted after the client's CCS message, but this does not affect its authentication properties.

The main protocol goal is that, unlike bearer tokens, the client's Channel ID cannot be used by a malicious server A to impersonate the client on a different server S, even if C accidentally connects to A using its Channel ID for S. In fact, this should be impossible even if A obtains the private key of a certificate valid for S, provided Channel ID is only enabled with forward-secret ciphersuites such as DHE [10, §6]. Consequently, an application that binds its tokens to the Channel ID make them unusable on a different TLS client without the associated private key. A typical example is for S to create a cookie by signing the session identifier with the Channel ID public key:

$$c = \mathsf{signed}(sk_S, [sid, pk_{cid}])$$

S would then only accept this cookie over a TLS connection authenticated by sk_{cid}, so stealing the cookie is of no use.

Attack and Mitigation The security of Channel ID relies on the uniqueness of the handshake log (log_c). If the attacker A can create a session to S with the same log, it can reuse C's Channel ID signature to impersonate C at S. Our synchronizing proxy achieves exactly this feat after resumption.

Suppose C establishes, then resumes a TLS session with A. A can synchronize a connection to S such that the log in the resumption handshake is identical between C-A and A-S. Hence, the Channel ID signature on the resumption handshake can be replayed to S, allowing A to successfully impersonate C. Henceforth, A can obtain S's channel-bound cookies meant for C and freely use them on this connection. This attack is well within the threat model of Channel ID. The Channel ID authors promptly responded to our report and in response, the protocol specification is being revised to include the hash of the original handshake in the Channel ID signature of abbreviated handshakes.

VII. Countermeasures

We propose several countermeasures at the TLS level that prevent our man-in-the-middle attacks at their source with few or no changes required to application-level mechanisms. The ideas behind these proposals emerged from discussions with various implementors and protocol experts and we are cautiously optimistic about their adoption. Since new protocol extensions and features can take a long time to propagate, we also discuss short-term mitigations for various applications.

VII-A A new channel binding.

In §V-C and §VI-C, we found that neither the session id, nor the master secret, nor the `tls-unique` channel binding served as unique representatives for a TLS session. Hence, we propose a new TLS channel binding, called `tls-session-hash`, that captures all the negotiated parameters for a session.

We define `tls-session-hash` for a given TLS session as the hash of the handshake messages up to and including the client key exchange message in the original handshake that created the session. The hash function used depends on the protocol version. For TLS 1.2, this is the hash function in the ciphersuite. For SSL3 and earlier versions of TLS, this is the concatenation of MD5 and SHA1 hashes. We require that TLS implementations compute and store `tls-session-hash` within its session structure and expose it to implementations.

Why this definition? We only hash messages up to the client key exchange, because at this point the negotiation is complete and all the inputs to the master secret are available, so most TLS implementations will create (but not cache) the session structure. Notably, the hashed log includes the nonces, the ciphersuite, key exchange messages, client and server certificates, and any identities passed in protocol extensions.

Our definition of the hash functions matches those used for the finished messages in SSL3 and TLS 1.0–1.2; hence, implementations already keep a running hash of the log and we just re-use its value. Implementing this channel binding increases the cached session size by a single hash, and has no performance impact.

We define a new hash value instead of reusing the client or server verify data for three reasons. (1) It is compatible with stateless servers [50], which must send the session ticket *before* the server finished message, so the server verify data is not available yet. (2) Being longer than the verify data, the session hash offers stronger collision resistance. While collisions may be less problematic for (the usually few) renegotiations on a single connection, a session can be long-lived and frequently resumed. (3) We could have reused the input to the client verify data, but it would not offer any clear advantages, and our current definition is more suitable for our proposed extensions.

Recommended Usage We recommend that protocols such as SCRAM use `tls-session-hash` rather than `tls-unique` for channel binding. To fix Channel ID, we recommend that the signature on abbreviated handshakes include the `tls-session-hash` of the resumed session. To derive application keys from the master secret, like in PEAP, we recommend adding `tls-session-hash` to the PRF.

VII-B Context Binding for Master Secrets.

We propose a new extension for all versions of TLS and DTLS that causes negotiated session parameters to be included in the master secret computation, following the principle of *context binding* [21], whereby computed keys should be usage-specific.

As usual, the extension is signaled in the client and server hello messages; if both peers support it, the handshake proceeds as usual, except that the master secret is computed as:

$$ms = \mathsf{prf}(pms, \text{“}\texttt{extended master secret}\text{”}, \texttt{tls-session-hash})$$

The inclusion of `tls-session-hash`, instead of just the pair of nonces, ensures that the resulting master secret depends on all the negotiated session parameters. The master secret implicitly authenticates these parameters, and different sessions will have different master secrets, foiling our attacks.

We find this solution elegant since it protects all TLS handshake modes (including RSA and DHE) and protocol versions, and it allows application-level protocols like PEAP to safely use the TLS master-secret without any changes. The idea of including additional materials in the master secret

computation is not new [21, 31, 3] but our proposal merits more detailed analysis, which we leave for future work.

VII-C SECURE RESUMPTION INDICATION. We propose a mandatory extension for all versions of TLS and DTLS that complements the renegotiation indication extension [49] by also protecting session resumption across multiple connections.

As in [49], the extension is signaled in the client and server hello messages (see §IV-C), but only when proposing and accepting resumption, respectively. It contains the `tls-session-hash` value of the session being resumed. Peers supporting the extension must check that this value matches the one recorded in their locally stored session before proceeding with the abbreviated handshake. The exchanged session hashes are authenticated by the master secret in the finished messages of the resumption, cryptographically binding the new connection to the resumed session. If one of the peers does not support the extension, the other should refuse session resumption and may instead offer a full handshake.

VII-D SUMMARY OF MITIGATIONS. We implemented the session hash channel binding and our two extensions as patches to OpenSSL and miTLS, and we tested their interoperability for all versions of TLS and DTLS. Our patches fit well into the code structure and have no visible effect on performance.

Independently, applications that rely on existing TLS APIs can mitigate the attacks of this paper by following some conservative design principles, at some cost to their functionality.

1) Do not allow the peer to renegotiate its certificate.
2) Do not use `tls-unique` after session resumption .
3) To derive application keys from the TLS master secret, hash the session's certificates into the derivation.
4) Buffer application data until its semantics is unambiguous; discard it if the TLS connection is torn down.
5) Do not share secret cookies between HTTP and HTTPS connections, or between different origins.

VIII. VERIFIED APPLICATION SECURITY OVER TLS

VIII-A MIHTTPS: A BASIC HTTPS CLIENT. To validate our application-level recommendations and show that one can indeed achieve transparent application-level security over TLS, we build and verify an exemplary HTTPS library, at the same level of abstraction as the CURL library, for example, but with fewer features. Its client command-line interface is as follows:

```
$ mihttps --help
Usage: mihttps [options] REQUEST
  --host=NAME    https server host name
  --channel=ID   channel identifier
  --client=NAME  authenticated client name
```

Our goal is to provide (1) a basic API with strong implicit security; and (2) a flexible implementation that supports typical mechanisms available in HTTP (cookies) and TLS (multiple connections, renegotiation, resumption, late client authentication). miHTTPS consists of 600 lines of F# coded on top of the miTLS verified reference implementation [15]. In particular, our client automatically processes HTTP 1.0 headers, cookies, etc, and interoperates with existing, unmodified web servers. We tested e.g. authenticated webmail access to Roundcube.

(We refer to the online materials for a more detailed description of miHTTPS, its code, and its verification.)

Secure Channels Our main communication abstraction is a long-term, stateful channel between a client and a host. Each client may create any number of channels and use them to request documents from URLs at different hosts; each channel supports parallel requests, as required e.g. when loading a web page that includes numerous resources. Each request may asynchronously return a document (in any order).

Such channels are *not* reliable: requests and responses may get lost or delayed, and their sender have no explicit acknowledgment of peer reception. Instead, responses confirm requests, and cookies attached to requests confirm prior responses.

In the command line, the `host=NAME` option indicates that a new channel should be created and its ID returned, whereas `channel=ID` indicates the local identifier of an existing channel to reuse. These application-level channels are not primitive in HTTPS or TLS; they intuitively account for a series of related requests issued by a client. For example, a user may have long-lived authenticated channels to every host she trusts, plus shorter-lived anonymous channels. The server is always authenticated. The user may use the `client=NAME` option, where `NAME` refers to a valid client certificate she owns to be used to authenticate her requests on the channel.

Simplifications We associate a unique host name to each channel, treating each host as a separate principal: thus, we do not deal with related sub-domains, redirects, or wildcards in certificate names. We also do not support mixtures of HTTP and HTTPS. Thus, we avoid many complications with cookies discussed in §II and §III. (Applications may still multiplex between hosts and protocols on top of our interface—what matters is that we do not share private state between channels.)

Client and Server Credentials We rely on the public-key infrastructure for X.509 certificates, and require that client and host names exactly match their certificates' common names. Our threat model does not cover certificates mis-issued to the adversary, or issued for different purposes with a common name that matches an honest principal.

Credentials are associated with the whole channel, once and for all. The host name cannot be changed, preventing the renegotiation attack of §VI-A. The client can decide to authenticate later on an anonymous channel, and from the server's viewpoint, this suffices to attribute all requests on the channel to that client. From the client's viewpoint, binding her name to the channel before a particular request guarantees that the server will only process it after client authentication.

Local State and Cookies Our channels maintain local, private state, including e.g. open connections, live sessions, cookies, and the names associated with the channel. Our channels also buffer request and response fragments, in order to deliver only whole HTTPS messages to the application—this simply foils truncation attacks, including those of §III-B.

At the server, we partition incoming requests into separate channels and track requests received from each client by attaching a (locally stored) fresh random cookie to each response. The set of responses actually received can then be

inferred from the cookies attached to latter requests. (Assuming sufficient cookie storage space and entropy to prevent collisions, this pattern provides accurate tracking information.)

VIII-B SECURITY GOALS (INFORMAL). We primarily focus on application-level channel integrity—see the online version for privacy. We follow the cryptographic model of [15] and configure honest clients and servers to only negotiate strong ciphersuites and algorithms [as defined by 15]. We show that, with overwhelming probability, the following properties hold:

1) Request Integrity: when an honest server accepts a request and attributes it to a channel bound to honest server and client names, the client has indeed sent the request on that channel, with matching principal names.
2) Response Integrity: when an honest client accepts a document in reply to a request to an honest server, that server has indeed sent the document in response to this request. (This property is sometimes called correlation.)
3) Tracking: when an honest server accepts a request echoing the cookie of a response on a channel with an honest client, the client indeed received this response.

Property 1 excludes any mis-attribution of a request to a client. Properties 1 and 2 apply to whole messages, thereby excluding truncations. This is achieved by parsing and buffering message fragments until the whole message has been received, decrypted, and authenticated.

VIII-C miHTTPS: SECURE TYPED INTERFACE. We follow the modular type-based cryptographic verification method [26] that was used to obtain the main security theorem for the miTLS API [15]. They specify computational security for various constructions and protocols using precise typed interfaces (instead of code-based games or ideal functionalities). They employ an expressive refinement-based type system for F#, write detailed typed annotations (4,000 lines for miTLS), and verify their code against them automatically using F7, an extended typechecker, coupled with Z3, an SMT solver.

The verification effort for miHTTPS consists of specifying its typed API and letting F7 typecheck its 600 lines of code, using the lower-level, verified, precisely-typed API of miTLS. In the rest of the section, we outline the types we use to capture the security goals of §VIII-B.

Figure 4 shows fragments of our typed specification for miHTTPS, focusing on the main functions for the client. It defines a type for names—plain strings used as common names in certificates—and for channels: **type** (;host:*name*)*chan*. This type is *indexed* by a value, host, itself of type *name*, recording in the type that the channel should be used only for communications with servers with a valid certificate for host. This type is also *abstract*, hiding its representation, so that only our miHTTPS implementation can access it; applications can just pass channels as arguments to the API, but they cannot access their internal states (and so cannot accidentally leak keys) or modify the host index (and so cannot get confused between channels to different hosts).

Our API has 3 main modules, and is parameterized by an application module, *Data*, provided by the application, that defines types for requests (URLs) and responses (documents). These types are both abstract and indexed. Their indexes

```
1  type name = string (* common names for both clients & hosts *)
2  type (;host:name) chan
3  predicate Honest of name (* no compromised certificate *)
4  predicate Client of name * host:name * (;host)chan
5
6  module Data (* defined by the application *)
7    type (;host,chan)request
8    type (;host,chan,request)document
9  module Certificate (...)
10 module Server (...)
11 module Client
12   val create: h:name → (;h) chan
13   val request: h:name → c:((;h)chan) →
14       (a:name{Client(c,a)})option → r:(;h,c)request → unit
15   val poll: h:name → c:((;h)chan) →
16       (r:(;h,c)request * (;h,c,r)document) option
```

Fig. 4. miHTTPS interface (excerpt)

specify the host, the channel, and the request (for responses), so only the application above miHTTPS can create and access values at those types. They yield strong, information-theoretic security: provided that the channel is between honest client and server, type safety ensures that our protocol stack, including HTTPS, TLS, TCP, and any network adversary, cannot read their content (except for their size after encryption), tamper with their content, or move contents from one channel to another. Essentially, the protocol can only pass requests unchanged from clients to servers, and similarly for responses.

The *Certificate* module manages certificates. Reflecting our threat model, it has functions for generating certificates for *Honest* names and endorsing keys for dishonest names.

The *Server* module defines the API for miHTTPS servers.

The *Client* module is the actual API used by client applications, such as our command-line client. It has functions for creating a new channel towards a fixed host h, for sending requests (with optional client authentication), and for polling responses to prior requests. These functions have precise value-dependent types specifying their pre- and post-conditions. For instance, *request* takes 4 parameters: the target host h; an existing channel c for that host; an optional client name a authorized by the user for that channel (as indicated by the predicate *Client*(c,a)); and a request for that host and channel.

IX. IMPACT AND LIMITATIONS

We have presented a series of attacks on authentication mechanisms built within and over TLS. Table II summarizes these new attacks and compares them to previous attacks, in terms of their impact and limitations. The table lists preconditions for each attack: what the attacker must be capable of; how the application (mis-)uses TLS; and whether previous mitigations block the attack (✗) or not (✓).

For example, the second row indicates that the cookie cutter attack of §III-B requires a network attacker *and* a client application that processes truncated HTTP headers over TLS *and* a server application that allows chosen plaintexts before the `Set-Cookie` header. Its advantage over previous TLS truncation attacks is a higher impact: it enables full HTTPS session hijacking (by stealing session cookies) between mainstream web browsers and popular websites such as Google and Facebook. Conversely, our variant of network-based session

TABLE II. SUMMARY OF ATTACKS: NOVELTY, IMPACT AND PRECONDITIONS

		Attacker Abilities				API Assumptions				Mitigations			
Attack	Broken Mechanism	1	2	3	4	5	6	7	8	9	10	11	Refs
TLS Truncation	HTTPS Session (Tampered)		✓			✓							[13, 52]
*Cookie Cutter	HTTPS Session (Hijacked)		✓			✓	✓						§III-B
Session Forcing (Server)	HTTPS Session (Login CSRF)	✓			✓					✓			[12, 18]
Session Forcing (Net)			✓							✗			
*Truncation+Session Forcing	HTTPS Session (Login CSRF)		✓			✓	✓			✓			§III-C
TLS Renegotiation (Ray)	TLS Client Auth (Certificate)		✓								✗		[49, 45]
TLS Renegotiation (Rex)		✓		✓					✓		✗		
*Triple Handshake (RSA)	TLS Client Auth (Certificate)	✓		✓					✓		✓		§VI-A
*Triple Handshake (DHE)		✓		✓				✓	✓		✓		§V-B
MITM Tunnel Auth (Net)	EAP (Certificate, Password)		✓							✗		✗	[8]
MITM Tunnel Auth (Server)	EAP (Certificate)	✓		✓						✓		✗	
*MITM Compound Auth	EAP (Certificate)	✓		✓						✓		✓	§VI-B
*MITM Channel Bindings	SASL (SCRAM-Password)	✓		✓						✓		✓	§VI-C
*MITM Channel ID	Channel ID (Public-Key)	✓			✓					✓		✓	§VI-D

1. Client connects to untrusted server
2. Active network attacker
3. Client authenticates on untrusted server
4. Attacker controls one subdomain on trusted server
5. Application accepts truncated TLS streams
6. Application sends attacker-chosen plaintext in channel
7. Client accepts unknown DH groups/degenerate public keys
8. Client accepts server certificate change during renegotitation
9. HSTS: Require TLS for all actions on trusted server
10. Require renegotiation indication extension
11. Bind authentication protocol to TLS channel

forcing (fifth row, §III-C) has the same impact as previous attacks; its novelty is that it bypasses their HSTS mitigation.

Our new attacks on TLS renegotiation, PEAP, SASL, and Channel ID are server-based man-in-the-middle attacks. They require that a client be willing to connect and authenticate with some credential (e.g. an X.509 certificate) at an untrusted server. The resulting attack is that the untrusted server can impersonate the client at any trusted server that accepts the same credential. The precondition that the client be willing to use its credential at an untrusted server is restrictive: it is more reasonable for public-key certificates than for server-specific tokens such as passwords. Still, such man-in-the-middle attacks by malicious servers were meant to be prevented by various channel-binding mechanisms built into these protocols, and our attacks show that these mitigations are insufficient.

Our triple handshake attack on TLS renegotiation (§VI-A) bypasses the renegotiation indication countermeasure, but it applies only to servers that authenticate clients with certificates during renegotiation. Such server configurations are not widespread, but can still be found in banks, certificate authorities, and VPN services. Furthermore, our impersonation attacks apply only to clients that are willing to accept a change of server certificates during renegotiation. Our experiments show that these and other preconditions in the table are frequently met by popular web browsers and TLS and HTTPS libraries.

IX-A RESPONSIBLE DISCLOSURE. We reported the attacks to several software vendors and suggested short-term fixes that invalidate the preconditions of these attacks. We summarize their responses below. In light of our findings, we advocate that all applications that rely on TLS carefully review their use of TLS libraries and implement similar fixes if necessary.

- Chromium (used by Chrome, Android, Opera): Header truncation attacks prevented in CVE-2013-2853. Server certificate change during renegotiation prevented in CVE-2013-6628.
- SChannel (used by Internet Explorer): Degenerate Diffie-Hellman public keys and server certificate change during renegotiation both prevented by a security update.
- NSS (used by Firefox): Degenerate Diffie-Hellman public keys prevented in CVE-2014-1491.
- Channel ID (implemented in Chrome): Impersonation attack prevented by using only ECDHE ciphersuites; specification revised to use session hashes (§VII-A).
- Safari: Notified of header truncation attack on June 13, 2013. Notified of an incorrect renegotiation behavior on January 10, 2014, which was fixed in a later update.
- Apache: Notified of POST message truncation in `mod_php` on April 29, 2013. Acknowledged, not fixed.

These short-term fixes, however, do not address our attacks on channel bindings in SASL and compound authentication in PEAP. More generally, our findings falsify the assumptions made by the authors and users of various protocol specifications [23, 49, 48, 51, 27, 42, 1, 6, 39, 33, 10]. A more systematic fix would be to strengthen the TLS protocol itself to provide these stronger expected authentication properties.

We contacted various members of the TLS working group, including authors of the renegotiation extension [49]. They acknowledged the attack and we are collaborating on two internet drafts that describe the mechanisms proposed in §VII. We informed authors of TLS channel bindings [6] of our attacks and they acknowledged that `tls-unique` in its current form should not be used after resumption. Discussions on revising the channel binding specification are ongoing.

The security of our proposed extensions remains to be formally evaluated. We plan to extend the cryptographic proofs of miTLS to precisely model these extensions and verify that they provide stronger authentication guarantees for TLS.

ACKNOWLEDGEMENTS

We thank Martín Abadi, Bruno Blanchet, Catalin Hritcu, Markulf Kohlweiss, Adam Langley, Marsh Ray, Martin Rex, Matthew Smith, Santiago Zanella-Beguelin and the anonymous referees for their comments on this work.

REFERENCES

[1] [MS-PEAP]: Protected Extensible Authentication Protocol (PEAP). http://msdn.microsoft.com/en-us/library/cc238354.aspx, 2013.

[2] HTTPS Everywhere. https://www.eff.org/https-everywhere, 2014.

[3] M. Abadi. Security protocols and their properties. In *Foundations of Secure Computation*, 2000.

[4] N. AlFardan, D. Bernstein, K. Paterson, B. Poettering, and J. Schuldt. On the Security of RC4 in TLS. In *USENIX Security*, 2013.

[5] N. J. AlFardan and K. G. Paterson. Lucky thirteen: breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[6] J. Altman, N. Williams, and L. Zhu. Channel Bindings for TLS. IETF RFC 5929, 2010.

[7] R. Anderson and S. Vaudenay. Minding your p's and q's. In *ASIACRYPT*, 1996.

[8] N. Asokan, V. Niemi, and K. Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *Security Protocols*. 2005.

[9] B. Aziz and G. Hamilton. Detecting man-in-the-middle attacks by precise timing. In *SECUREWARE*, 2009.

[10] D. Balfanz and R. Hamilton. Transport Layer Security (TLS) Channel IDs. IETF Internet Draft v01, 2013.

[11] E. Barker, D. Johnson, and M. Smid. *NIST Special Publication 800-56A Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*, 2007.

[12] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *ACM CCS*, 2008.

[13] D. Berbecaru and A. Lioy. On the Robustness of Applications Based on the SSL and TLS Security Protocols. In *PKI*. 2007.

[14] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified Cryptographic Implementations for TLS. *ACM TISSEC*, 15(1):1–32, 2012.

[15] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE S&P*, 2013.

[16] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Zanella-Beguelin. Proving the TLS handshake (as it is). 2013. Unpublished Draft.

[17] S. Blake-Wilson and A. Menezes. Unknown key-share attacks on the station-to-station (STS) protocol. In *PKC*, 1999.

[18] A. Bortz, A. Barth, and A. Czeskis. Origin cookies: Session integrity for Web applications. In *W2SP*, 2011.

[19] A. Cassola, W. Robertson, E. Kirda, and G. Noubir. A practical, targeted, and stealthy attack against WPA enterprise authentication. In *NDSS*, 2013.

[20] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE CSF*, 2009.

[21] L. Chen. *NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions*, 2009.

[22] J. Clark and P. van Oorschot. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *IEEE S&P*, 2013.

[23] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[24] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-bound certificates: a fresh approach to strong client authentication for the web. In *USENIX Security*, 2012.

[25] T. Duong and J. Rizzo. The CRIME attack. In *Ekoparty*, 2012.

[26] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *ACM CCS*, 2011.

[27] P. Funk and S. Blake-Wilson. Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0. IETF RFC 5281, 2008.

[28] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM CCS*, 2012.

[29] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. In *ACM CCS*, 2013.

[30] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). IETF RFC 6797, 2012.

[31] P. Hoffman. Additional Master Secret Inputs for TLS. IETF RFC 6358, 2012.

[32] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, 2012.

[33] S. Josefsson and N. Williams. Using GSS-API Mechanisms in SASL: The GS2 Mechanism Family. IETF RFC 5801, 2010.

[34] B. S. Kaliski Jr. An unknown key-share attack on the MQV key agreement protocol. *ACM TISSEC*, 4(3):275–288, 2001.

[35] H. Krawczyk, K. G. Paterson, and H. Wee. On the Security of the TLS Protocol: A Systematic Analysis. In *CRYPTO*, 2013.

[36] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[37] M. Marlinspike. More Tricks For Defeating SSL In Practice. *Black Hat USA*, 2009.

[38] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[39] A. Menon-Sen, N. Williams, A. Melnikov, and C. Newman. Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms. IETF RFC 5802, 2010.

[40] C. Meyer and J. Schwenk. Lessons learned from previous SSL/TLS attacks – A brief chronology of attacks and weaknesses. In *IACR Cryptology ePrint Archive*, 2013.

[41] R. Oppliger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication – Or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2238–2246, 2006.

[42] A. Palekar, D. Simon, J. Salowey, H. Zhou, G. Zorn, and S. Josefsson. Protected EAP protocol (PEAP) version 2. IETF Internet Draft v10, 2004.

[43] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, 2011.

[44] J. Puthenkulam, V. Lortz, A. Palekar, D. Simon, and B. Aboba. The compound authentication binding problem. IETF Internet Draft v04, 2003.

[45] M. Ray and S. Dispensa. Renegotiating TLS, 2009.

[46] J.-F. Raymond and A. Stiglic. Security issues in the Diffie-Hellman key agreement protocol. *IEEE Transactions on Information Theory*, 22:1–17, 2000.

[47] E. Rescorla. HTTP over TLS. IETF RFC 2818, 2000.

[48] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). IETF RFC 5705, 2010.

[49] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. TLS renegotiation indication extension. IETF RFC 5746, 2010.

[50] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. TLS session resumption without server-side state. IETF RFC 5077, 2008.

[51] D. Simon, B. Aboba, and R. Hurst. The EAP-TLS Authentication Protocol. IETF RFC 5216, 2008.

[52] B. Smyth and A. Pironti. Truncating TLS Connections to Violate Beliefs in Web Applications. In *USENIX WOOT*, 2013.

[53] E. Stark, L.-S. Huang, D. Israni, C. Jackson, and D. Boneh. The case for prefetching and prevalidating TLS server certificates. In *NDSS*, 2012.

[54] P. van Oorschot. Extending cryptographic logics of belief to key agreement protocols. In *ACM CCS*, 1993.

[55] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Electronic Commerce*, 1996.

[56] N. Williams. On the use of channel bindings to secure channels. IETF RFC 5056, 2007.

[57] M. Zalewski. Browser Security Handbook. http://code.google.com/p/browsersec/.