Practical Issues with TLS Client Certificate Authentication

Arnis Parsovs
Software Technology and Applications Competence Center, Estonia
University of Tartu, Estonia
arnis@ut.ee

***Abstract*—The most widely used secure Internet communication standard TLS (Transport Layer Security) has an optional client certificate authentication feature that in theory has significant security advantages over HTML form-based password authentication. In this paper we discuss practical security and usability issues related to TLS client certificate authentication stemming from the server side and browser implementations. In particular we analyze Apache `mod_ssl` implementation on server side and the most popular browsers – Mozilla Firefox, Google Chrome and Microsoft Internet Explorer on client side. We complement our paper with a case study performed in Estonia where TLS client certificate authentication is widely used. We present our recommendations for TLS implementations on the client and server side to improve the security and usability of TLS client certificate authentication.**

I. Introduction

Use of TLS protocol is a standard way to secure Internet connection between a client's browser and HTTP web servers. Most commonly TLS is used to authenticate the server, in order to assure the client that the communication is being performed with a legitimate party and that the secrecy and the integrity of all data exchanged between the client and the server is ensured. To authenticate oneself to the server, the client usually has to submit some identification data (such as a username) and a secret that is shared between the client and the server (such as a password) over an already established server-authenticated secure channel. There are various ways how this process can fail ending up with the client disclosing his shared secret to an attacker rather than to a legitimate server. As a consequence, the attacker can access the victim's account on a legitimate server, obtain the victim's private information stored on the server and perform actions in the service with the privileges of the victim. In practice, the attacker can obtain a shared secret, for example: by phishing attacks; by compromising another web service where the victim might have reused the same secret; by tricking the victim into sending the secret over a channel that is not protected by TLS; by compromising a trusted certificate authority (CA) or by in any other way fraudulently obtaining the certificate of a legitimate server signed by a trusted CA and then using the certificate in a man-in-the-middle (MITM) attack.

Active security research is being done to improve password security, educate users to resist phishing attacks, and to fix CA trust issues [1], [2]. However, the attacks mentioned above can be prevented or their impact can be greatly reduced by using TLS client certificate authentication (CCA), since TLS CCA on TLS protocol level protects the client's account on a legitimate server from a MITM attacker even in the case of a very powerful attacker who has obtained a valid certificate signed by a trusted CA and thus is able to impersonate the legitimate server. We believe that TLS CCA has great potential in improving Internet security and therefore in this paper we discuss current issues with TLS CCA and provide solutions that will improve the security of TLS CCA and enable its usage on a larger scale.

With this paper, we make the following contributions:

- We provide first systematic analysis of issues that are related to deploying TLS CCA in practice.
- We present a detailed report of a case study in which the TLS CCA deployments of 87 Estonian service providers are analyzed.
- We give a list of recommendations on how to solve the problems identified to improve and facilitate the use of TLS CCA.

The rest of the paper is organized as follows. Section II provides a description of the TLS protocol in a server authenticated and client authenticated setting and gives a brief security analysis of the protocol. Section III discusses TLS CCA related issues. Section IV analyzes the case study of Estonian service providers who provide the TLS CCA option. Our recommendations for both the client and server side are provided in Section V. The related work is discussed in Section VI. Section VII concludes the paper.

II. TLS Overview

TLS protocol and its predecessor SSL (Secure Sockets Layer) has several versions. In this paper we are referring to TLS protocol assuming the most widely supported TLS version 1.0 [3]. The description given here applies also to the latest protocol versions 1.1 and 1.2 since they do not introduce any changes related to TLS CCA.

A. Server Authenticated TLS Handshake

Fig. 1 shows protocol messages exchanged between a client and server during TLS protocol negotiation. Before encrypted and integrity protected application data can be exchanged, the TLS handshake has to be completed to negotiate commonly supported cipher suites and other protocol parameters.

TLS connection is being initiated by the client's `ClientHello` message which contains a list of cipher suites that the client supports and the client's randomness which is unpredictable for every TLS connection. The server replies with the `ServerHello` message, which contains the server's randomness and one cipher suite selected from the client's `ClientHello` message. This cipher suite will be used for the TLS session key exchange, encryption and integrity protection. The server then in the `Certificate` message provides one or more X.509 certificates that should be used by the client

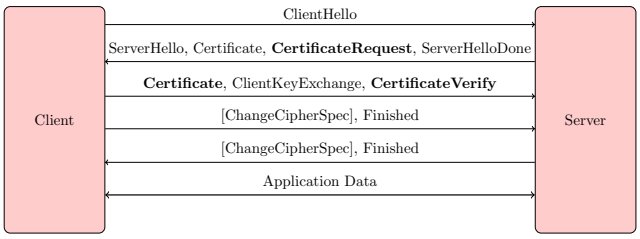


Fig. 1: TLS handshake involving CCA and using RSA as key exchange mechanism. The messages related to optional client certificate authentication are emboldened.

to build the certificate chain. The first certificate given in the server's `Certificate` message is assumed to be the server's certificate with a corresponding private key being in the server's possession. The server finishes his round by sending a `ServerHelloDone` message. In turn, the client generates a random value – the pre-master secret which should be used by both parties to derive symmetric keys used for the chosen cipher suite and encrypts it with the server's public key obtained from the server's certificate. The encrypted pre-master secret is sent to the server in the `ClientKeyExchange` message. Then the client sends a `ChangeCipherSpec` message which signals to the other party that all further messages coming from the client will be protected by the negotiated cipher suite and symmetric keys. The last handshake message `Finished` is sent already encrypted and contains a hash of all previous messages exchanged between the client and the server. The server compares the decrypted hash with the hash of all previous handshake messages. The client does a similar verification with the server's `Finished` message and the exchange of encrypted application data can now be performed.

Since the client encrypts the pre-master secret with the public key of the server, only the server which is in possession of the corresponding private key can decrypt the content of the `ClientKeyExchange` message and thus obtain symmetric keys which are used to protect further communication. In general, TLS is secure against passive and active network attacks, as long as the client uses a public key that really belongs to the server he is willing to communicate with. In practice, public-key infrastructure (PKI) is used to verify authenticity of the public key provided in the server's `Certificate` message.

B. Client Authenticated TLS Handshake

In the client authenticated TLS handshake the server additionally requests the client's certificate by sending a `CertificateRequest` message. This message contains a list of distinguished names (DNs) of CAs that the server trusts.

This list may be used by the client to choose an appropriate certificate that should be sent to the server in the client's `Certificate` message. The client's `Certificate` message similarly to the server's `Certificate` message may contain several certificates that may be used by the server to build the certificate chain up to the CA that the server trusts. If a client does not have a certificate or does not want to perform CCA, the client can send an empty `Certificate` message and then the server can decide whether to complete the handshake without the client's certificate.

The proof that the client has access to the private key that corresponds to the public key in the client's certificate, is given by calculating the hash of all the previous handshake messages exchanged between the client and the server so far and signing it with the client's private key. This signature is sent in the client's `CertificateVerify` message. The `CertificateVerify` message is omitted if the client has sent an empty `Certificate` message.

The main TLS CCA advantage compared to authentication methods in which a shared secret is disclosed to the server, is that in the TLS CCA process a client proves to the server that he has access to the private key, but the private key itself is never disclosed to the server.

As can be seen, the signature given using the client's private key is bound to the client's and the server's randomness, the server's certificate and the encrypted pre-master secret sent by the client. This means that even the attacker that has obtained the signature in a MITM attack, cannot reuse the signature in any other TLS handshake either to the legitimate server or to any other server. As a result, in cases where TLS CCA is used, the attacker cannot impersonate the victim to the legitimate server even if the attacker is able to impersonate the legitimate server. Server impersonation attack is still a problem if the attacker's goal is to obtain sensitive data from the user and not from the server, for example, in a scenario where a user submits sensitive data to the server right away, without being able to detect the impersonation attack in another way (e.g., by visually noticing that the authenticated environment provided by the attacker does not fully replicate the personalized environment of the legitimate server).

Therefore, we see that at least in theory TLS CCA is secure even against a very powerful attacker that is able to successfully impersonate the legitimate server.

C. Renegotiation

The TLS server at any time can send to the client a `HelloRequest` message thereby requesting the client to initiate a new TLS handshake. In the renegotiation process the messages of the new handshake are cryptographically protected by the cipher suite negotiated in the previous handshake. The practical use of renegotiation could be, for example, to negotiate a stronger cipher suite or to perform CCA if the client on the application level wants to access server resources that must be protected by such security measures according to the server's local security policy.

Client can also initiate renegotiation by sending `ClientHello` message at any time, but there is no practical reason for client to do that, therefore client initiated renegotiations are usually disabled by the servers.

D. Session Resumption

The TLS session resumption feature allows to perform an abbreviated TLS handshake by skipping the key agreement phase. The client can resume a TLS session by specifying a session identifier in the `ClientHello` message from the TLS session negotiated before. If the TLS server is willing to resume the session, the server will reply with the same session identifier in the `ServerHello` message, otherwise a new session identifier will be included and a full handshake will be performed. The abbreviated handshake reduces the cost by one round-trip time across the network, plus one private key operation on the server side (and one on the client side if CCA has to be performed).

Session resumption is especially important when the private key used for CCA is stored in a cryptographic token such as

a smart card. If session resumption is not used, the smart card would have to perform a signing operation on every HTTP request that initiates a new TLS connection. That would add significant delay in the TLS handshake process and would rapidly reach the private key use limit that is often enforced by smart cards.

Note that the effectiveness of the TLS session resumption depends also on a particular browser and browsing characteristics. For example, if the user navigates to another page while a page has not been fully loaded, the current TLS session might be aborted and therefore will become non-resumable.

III. Practical Issues

In this section we will summarize both already known problems and describe some we have discovered.

To enable TLS CCA on the server side, most of the service providers use an Apache HTTP server with the module `mod_ssl` [4] that relies on the OpenSSL library to implement TLS functionality. In this section we will analyze server side issues considering only Apache `mod_ssl` and in particular version 2.2, which currently is the version shipped within most operating systems. All CCA related aspects described here also apply to the latest stable version 2.4 (with the exception of client certificate revocation checking).

On the client side we will analyze TLS CCA support as implemented in three most popular browsers – Mozilla Firefox version 19.0 on Linux and Windows 7, Google Chrome version 25.0 on Linux and Windows 7, and Microsoft Internet Explorer (IE) version 9.0 on Windows 7.

A. Apache `mod_ssl` Configuration

There are several `mod_ssl` configuration directives that help to configure CCA on a server side. However, since the official `mod_ssl` documentation might give a wrong understanding of the behavior they introduce, we will give a brief but clear description here.

`SSLVerifyClient` is the main configuration directive regulating TLS CCA. This directive can be specified in a server wide context or per-directory context. When specified in a server wide context, the setting applies to the initial TLS handshake. If specified in a directory context, the server will request TLS renegotiation after receiving an HTTP request for resource in that directory. If the renegotiation is successful, the request will be processed and the response will be sent back over the renegotiated session.

The default value `none` of `SSLVerifyClient` does not require CCA, therefore the server will not include a `CertificateRequest` message in the TLS handshake. The value `require` will require CCA thereby the `CertificateRequest` message will be included in the handshake. If the client will not provide any certificate in the client's `Certificate` message or `mod_ssl` will fail to verify the certificate provided, the TLS handshake will be aborted and a fatal TLS alert message will be sent to the client. The value `optional` is the same as `require`, but an empty client's `Certificate` message will be tolerated. The last possible value `optional_no_ca` is the same as `optional`, but in addition it allows to submit a client's certificate, that does not chain up to the CA trusted by the server (because of the bug in OpenSSL [5] not yet valid or expired non-self-signed client certificates will also be accepted).

The `SSLCACertificateFile` directive specifies a location where self-signed root CA certificates trusted by the server and their intermediate CA certificates are located.

The `SSLVerifyDepth` directive sets a limit for the length that certificate chain built in verification process may have.

The `SSLCADNRequestFile` directive specifies a location of certificates which will be used to build a list of DNs, that will be sent in the server's `CertificateRequest` handshake message. If this directive is not set, the DNs of certificates in `SSLCACertificateFile` will be used for this purpose.

After a successful TLS handshake, the `mod_ssl` will set environment variables that can be used by the server-side application to perform further authorization based on the data from CCA. For example, variable `SSL_CLIENT_CERT` will contain the PEM-encoded client certificate and variable `SSL_CLIENT_VERIFY` will contain the value `NONE`, `SUCCESS`, `GENEROUS` or `FAILED:`*`reason`* depending on whether the client certificate was provided and whether the client certificate chained up to the CA trusted by the server (in case `SSLVerifyClient` is set to `optional_no_ca`). Because of the bug in `mod_ssl` [6] the variable `SSL_CLIENT_VERIFY` after session resumption may be falsely set to `SUCCESS` even if the client certificate does not chain up to the CA trusted by the server and should have been set to `GENEROUS` instead. By analyzing the `mod_ssl` source code we also found that variable `SSL_CLIENT_VERIFY` will never be set to value `FAILED:`*`reason`* contrary to the official documentation.

B. Verification

Certificate verification is performed by OpenSSL using the verification algorithm described in [7]. Certificates specified in `SSLCACertificateFile` are loaded into the OpenSSL trust store. It is important to emphasize here that for a successful verification the client certificate chain must build up not only to a certificate found in the trust store, but this certificate has to be self-signed. In addition, note that OpenSSL will use extra certificates provided in the client's `Certificate` message in building the certificate chain. As a result, the configuration options provided by `mod_ssl` are not sufficient to implement CCA securely in an advanced public-key infrastructure setting, without performing additional verification on application level.

For instance, the Estonian CA "AS Sertifitseerimiskeskus" (SK) has a root certificate and several intermediate CA certificates, but only one particular intermediate CA issues authentication certificates for natural persons that are loaded into an Estonian ID card (a smart card storing corresponding RSA keys). A straightforward approach as instructed by SK [8] to put into `SSLCACertificateFile` the SK root certificate and the certificate of an intermediate CA issuing ID card authentication certificates, will actually not guarantee that in case of successful authentication the accepted client certificate is signed by that particular intermediate CA. The verification process will be successful also in the case where a client certificate is issued by SK root CA directly or by any other intermediate CA issued by the SK root CA.

This can be exploited in practice by, for example, compromising some other intermediate CA that is used for other purposes (e.g., testing) and therefore has weaker protection, or by fraudulently obtaining a certificate from another interme-

diate CA that is meant to be used in a different context and thereby has a lower assurance level and identity verification requirements in the registration process, and finally, it might hold that intermediate CAs are operated by mutually hostile parties.

It is not trivial to perform certificate chain post-verification at application level. The `mod_ssl` exports the client certificate in `SSL_CLIENT_CERT` environment variable and extra certificates provided in the client's `Certificate` message in the `SSL_CLIENT_CERT_CHAIN_`*n* variables, but the chain used in the verification process is not exported.

Most of the TLS CCA deployments in practice use `require` or `optional` value for `SSLVerifyClient` configuration directive and as we have found from private discussion with service providers, the only verification step performed at application level is to check whether the `commonName` attribute of the subject DN in the client certificate contains the name and personal identity code of the natural person in the expected form (`mod_ssl` provides subject DN in environment variable `SSL_CLIENT_S_DN` or it can be extracted from the client certificate using, for example, function `openssl_x509_parse()` available in PHP). If this check succeeds, the person is assumed to be authenticated and the server-side application continues with the authorization using the extracted personal identity code.

C. Reconnaissance

All `mod_ssl` and OpenSSL client certificate verification checks are done before the client has sent his `CertificateVerify` message proving that he is in possession of a corresponding private key. This allows the enumeration of the server's trust settings by using certificates which may be easily obtained by the attacker. For example, an attacker could send an arbitrary certificate in the client's `Certificate` message and observe whether the server aborts the handshake by sending an alert message before receiving the client's `CertificateVerify` message. If a server does not abort the connection, the attacker can conclude that the certificate in question is indeed located in the server's `SSLCACertificateFile` trust store (unless `SSLVerifyClient` is set to `optional_no_ca`). Note that the server's trust settings in the form of DNs of the CAs located in the server's trust store are also disclosed in the `CertificateRequest` message, if not explicitly overriden by the `SSLCADNRequestFile` directive.

The value of `SSLVerifyClient` can be similarly determined, for instance, by sending an arbitrary self-signed client certificate in a TLS CCA hanshake. If the handshake succeeds, the `SSLVerifyClient` is set to `optional_no_ca`. Otherwise, an empty `Certificate` message can be sent to distinguish between `optional` and `require`.

However, we have discovered that an attacker can also determine the `SSLVerifyDepth` value if the attacker has any root certificate that is located in the server's `SSLCACertificateFile` trust store. In order to do that an attacker has to send a fake certificate chain in the client's `Certificate` message, that chains up to any root certificate in the server's `SSLCACertificateFile` trust store. Certificates in the fake chain have to contain the correct issuer description so that the certificate chain building algorithm in OpenSSL succeeds, however, signatures on the certificates must be invalid. If the `SSLVerifyDepth` has been exceeded, the server will abort the handshake by sending a `certificate_unknown` alert message. However, if the `SSLVerifyDepth` was not exceeded, the OpenSSL will continue with signature verification on the certificate chain which will fail and the TLS handshake will be aborted with a `decryption_failed` alert message. If the `SSLVerifyClient` option `optional_no_ca` is used, the attacker can send a fake chain that chains up to any attacker's chosen self-signed root CA certificate that is included in the attacker's `Certificate` message. In this case if the `SSLVerifyDepth` has been exceeded the handshake will fail with an `unknown_ca` alert message.

D. Privacy of Client's Certificate

As can be seen in Fig. 1, the client's `Certificate` message is sent before encryption has been applied, therefore, the content of a client's certificate is available to passive network attackers. Certificates issued by CAs to natural persons usually contain at least the name and personal identity code of a person. Even if a certificate does not include any personal data, the public key in the certificate can be used by a passive network attacker to track the client. A TLS extension [9] has been proposed for standardization to the IETF Network Working Group that allows a client certificate to be encrypted in the initial TLS handshake by moving the client's `Certificate` message after the client's `ChangeCipherSpec` message. However, this extension has not yet been standardized and even if it had, it would take years before it is widely deployed and supported.

A practical workaround that can be used to protect the client certificate is for a server to request the client certificate on renegotiation. This way the client's `Certificate` message will be encrypted by the cipher suite negotiated by the previous TLS negotiation. As stated before, this can be configured by specifying `mod_ssl` configuration directive `SSLVerifyClient` in the Apache directory context. However, the renegotiation has a negative impact on performance, since an additional handshake is performed when renegotiating. The performance can be improved if the TLS session resumption is enabled by both a client and a server. Unfortunately, recent versions of `mod_ssl` do not support resumption of TLS sessions that have been established on renegotiation [10].

We have observed that none of the browsers analyzed warn the user about the privacy leak if the TLS server requests CCA on initial TLS negotiation.

E. Denial-of-service Attack Vectors

The TLS CCA enabled `mod_ssl` faces additional denial-of-service (DoS) attack vectors. On the server side the OpenSSL has to perform ASN.1 parsing on arbitrary certificates that are provided in the client's `Certificate` message. The signatures on a chain built has to be verified and the client's signature of handshake messages provided in the client's `CertificateVerify` message must also be verified. Parsing of ASN.1 structures is not a trivial task and OpenSSL has had security vulnerabilities that can be exploited if ASN.1 parsing is performed on untrusted data [11]. Recently a DoS attack tool[1] has been released which exploits computational asymmetry in a server authenticated TLS handshake.

[1] http://www.thc.org/thc-ssl-dos/

But what impact does the TLS CCA on a server might have? First, there are several limitations enforced by OpenSSL:

- the handshake will fail if the client's `Certificate` message is larger than 100 kilobytes;
- the handshake will fail if the `CertificateVerify` message is larger than 512 bytes, which means that the maximum length of the RSA modulus that the client's public key can have is 4096 bits.
- chain building will stop when its depth exceeds 100;
- the signatures of the certificate chain will not be verified if the chain is larger than the depth configured by the `mod_ssl SSLVerifyDepth` directive;

In addition to that, signature verification will not be performed and will fail if the modulus of the RSA public key is larger than 16384 bits or if the public exponent for the RSA public key whose modulus is larger than 3072 bits is larger than 64 bits. An experiment shows that with an "Intel Core 2 Duo U7700@1.33GHz" processor 26 RSA verifications per second can be performed with a public key having 16384-bit modulus and maximum allowed 64-bit public exponent, and only 20 RSA verifications per second with a public key having 3072-bit modulus and 3072-bit public exponent. However, in order to exploit that, an attacker has to have a certificate whose public key has these characteristics and is signed by CA that chains up to a root CA certificate located in server's `SSLCACertificateFile` trust store.

Unfortunately, if the `SSLClientVerify` is set to `optional_no_ca`, the attacker can provide an arbitrary certificate chain in the `Certificate` message and signatures will be verified by OpenSSL as long as the chain does not exceed the depth limit configured by `SSLVerifyDepth`. The default configuration value for `SSLVerifyDepth` is 1, which would result in OpenSSL performing two signature verification operations per handshake – one to verify signature on the certificate chain and another to verify the signature given in the `CertificateVerify` message. The worst case scenario, where `SSLClientVerify` is set to `optional_no_ca` option and `SSLVerifyDepth` is set to 100, would allow an attacker to send a certificate chain in a `Certificate` message containing 75 certificates (whose public keys have 3072-bit modulus and 3072-bit public exponent) reaching near 100KB of the maximum allowed for `Certificate` message size and would force a server that has the processor described above to spend 4 seconds of wall time to process the handshake.

Another DoS attack vector exists if the server requests TLS CCA on renegotiation after receiving HTTP request. The problem in this scenario is that the request body of HTTP request has to be buffered by the server before renegotiation is performed and the request body can be processed or discarded by the target resource. To prevent possible memory exhaustion DoS attack, `mod_ssl` provides configuration directive `SSLRenegBufferSize`, which limits request body size that may be buffered by the server (the default value is 256KB). If the client sends larger request body the server will return error `413 Request Entity Too Large`. This creates a problem if the server must support large POST requests. The workaround that can be used by the client is to include header field `Expect: 100-continue` in the request header sent to the server, and sending the request body only after the TLS CCA has been performed and the server has responded with status code `100 Continue`. Unfortunately, this mechanism is not used by the browsers and therefore servers that require TLS CCA on renegotiation and must support large POST requests will have to set `SSLRenegBufferSize` to a large value, thereby providing DoS attack vector.

F. Freshness of the Client's Proof

In a CCA process the client proves to the server that he has access to the private key that corresponds to the public key in the client's certificate. This is done by signing a message digest of previously exchanged TLS handshake messages and providing a signature in the client's `CertificateVerify` message. For a server side application it is crucial to determine the freshness of this proof. For example, the server's security policy might require that before some authenticated action can be performed, the client must prove that he has had recent access to his smart card. We have found that there is no environment variable provided by `mod_ssl` that could be used to reliably determine freshness of the proof given in the TLS CCA process. From the protocol description we see that the client cannot compute a signature before the server has sent its `ServerHello` message which contains the server's randomness and therefore cannot be predicted by the client, while the contents of all further messages up to the client's `CertificateVerify` message can be predicted by the client. Therefore the freshness of a client's possession of the private key could be determined by the timestamp of the moment when the server has sent its `ServerHello` message.

We found that even if the TLS session resumption is not enabled, the time of the HTTP request cannot be reliably used by the server-side application to calculate the freshness of the client's proof of possession. The assumption that the TLS connection will timeout if the `CertificateVerify` message is not provided soon enough after the `ServerHello` message, does not hold. OpenSSL itself does not enforce a timeout, but Apache has a default timeout set to 300 seconds configurable by the `TimeOut` directive. However, this timeout counter will be reset whenever new data is added to the read or write buffer. By using TLS fragmentation and sending empty TLS message fragments the TLS connection can be kept open for an unlimited amount of time before sending a `CertificateVerify` message.

After DoS attack tool Slowloris[2] was released (the tool tries to open many connections to the target server and keep them open as long as possible), the Apache starting from version 2.2.15 ships module `mod_reqtimeout` [12] that imposes restrictions on the minimum transfer speed and the time limit a client is allowed to stay connected. Debian GNU/Linux and most software distributions based on Debian ship Apache with `mod_reqtimeout` enabled by default. The default configuration will force the connection to be terminated if the client's HTTP request header is not received within 40 seconds. However, the total time for receiving an HTTP request body is not enforced as long as the first byte of the request body is received in the first 10 seconds and then the minimum data transfer rate of 500 bytes per second is maintained. Since the TLS renegotiation process is part of the request body receiving process, the renegotiation handshake can be kept open for an unlimited amount of time as long as the data rate of 500 bytes per second is maintained. In practice, one TLS record containing an AES encrypted empty handshake message that

[2]http://ha.ckers.org/slowloris/

is padded to the maximum allowed length will have a size of 277 bytes, therefore two such TLS records must be sent in a second to maintain the required data rate of 500 bytes per second.

As a result, a large part of servers requiring TLS CCA might not put a time constraint on an attacker for obtaining a signature once the data to be signed is known. This can be exploited, for example, in the Estonian ID card case if the victim provides a padding oracle to the attacker. Since forging a signature using a padding oracle attack requires thousands of oracle calls to the ID card, the attacker can obtain forged signature only several hours after the data to be signed is known [13]. We see that this limitation might not be a problem and therefore a padding oracle attack can be practically exploited to perform CCA on behalf of the victim if the attacker has access to such oracle.

Even if a server-side application could obtain the time when the client performed CCA, we face another problem. Server-side scripting engines do not provide function calls that could be used to communicate to the underlying `mod_ssl` that a particular TLS session should be cleared in order to request client certificate re-authentication, and as it will be discussed further, browsers also do not provide API that could be used to force a TLS session deletion from a client side.

G. Certificate Revocation Checking

At `mod_ssl` level client certificate revocation checking can be performed using a certificate revocation list (CRL). The `SSLCARevocationFile` directive specifies a location where CRLs used for certificate chain revocation checking are located. Only direct CRLs are supported and only these certificates will be checked for which CRLs are provided. When CRL files are updated, the Apache process must be gracefully restarted (using signal `USR1`) in order to reload CRLs. The TLS CCA handshake will fail if an outdated CRL is specified i.e., if the `nextUpdate` is in the past. However, it is not checked whether `thisUpdate` is not in the future and therefore outdated CRLs used by servers with a backward system time will not be detected.

In `mod_ssl` version 2.4, the revocation checks can be performed also using Online Certificate Status Protocol (OCSP).

H. CCA Audit Trail

HTML form-based password authentication provides a flexible way to record authentication success and failures at application level. However, in the case of `mod_ssl`, the CCA failure resulting from any reason will end with a TLS handshake failure without reaching application level. Therefore CCA auditing must be done on `mod_ssl` level.

Unfortunately, the logging provided by `mod_ssl` is not sufficient. When using `LogLevel` value `error`, only general CCA success and failure messages are logged. When `LogLevel` is set to value `debug`, which provides maximum verbosity level, the subject DN of the certificate and the packets received by OpenSSL are also dumped into the log file, however, if the CCA is performed on renegotiation, the packets of the handshake dumped are in an encrypted form.

In order to fully benefit from the auditing process, the full TLS CCA handshake should be logged in a decrypted form. This would allow to investigate precise reasons for authentication failure, and in a case of successful CCA, the handshake messages could be used by a server as proof that the server had interaction with the client who possesses the private key corresponding to the client's certificate. The proof would even give an indication about the time period, since the randomness in the `ClientHello` and `ServerHello` messages contains the timestamp of the client and server and the server's `Certificate` message contains the validity period of the certificate.

I. CCA Support in Browsers

When a server requests the CCA, the browser has to ask the user whether to perform CCA and which certificate should be used for that. The client certificate selection windows as implemented by browsers can be seen in Fig. 2. The Chrome browser has a different selection window on Linux and Windows, since on Linux it uses NSS library to implement TLS while on Windows it uses SChannel and Windows certificate management facilities.

As can be seen, the client certificate selection windows provide different levels of detail. Chrome uses the address from the address bar to indicate CCA requesting party, while Firefox shows information from the server's certificate. Unfortunately, the IE in its client certificate selection window does not even try to inform which party is requesting CCA.

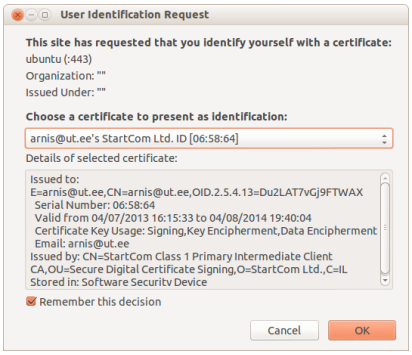
The list of certificates in the selection window are filtered according to the list of CA DNs provided in the server's `CertificateRequest` message. If a server sends an empty list of DNs, the browser offers to select any client certificate that the user has. The certificates whose purpose specified in the certificate extensions is not consistent with CCA are not listed in the certificate selection window. In addition, IE and Chrome will not list client certificates that are not yet valid or have expired. If the user has no suitable client certificates for authentication the certificate selection window is not shown and an empty client `Certificate` handshake message is sent.

Firefox certificate selection window has an additional option "Remember this decision" which is enabled by default. In practice, browsing with this option disabled would significantly degrade the user experience, since the user would be asked to select the certificate again whenever a new TLS session is established with the server. IE and Chrome do not even support disabling this option and automatically cache certificate choice for a particular server until the browser has been restarted or the user has manually cleared the TLS cache (in Firefox: "Tools" – "Clear Recent History" – "Active Logins", in IE: "Tools" – "Internet Options" – "Content" – "Clear SSL state", in Chrome not implemented [14]).

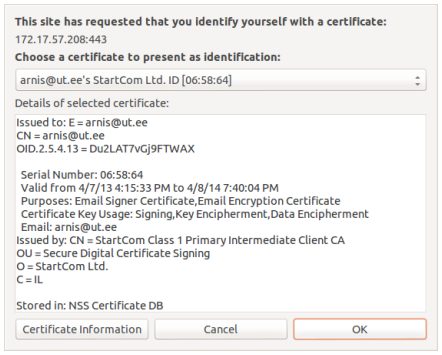
However, the caching of client certificate choice creates another usability and security issue. If the user has selected the wrong certificate and wants to authenticate with a different one or wants to prevent the browser from performing further CCA with the server (e.g., after logout), the user has to restart the browser or manually clear the TLS cache disturbing all connections in the current browsing session.

Firefox exposes a non-standard JavaScript function `window.crypto.logout()` [15] which clears the TLS session cache and certificate choice for the server in whose browser security domain the function is called. This function can be used by server side applications to provide true logout functionality and the possibility for the user to re-authenticate with a different client certificate.

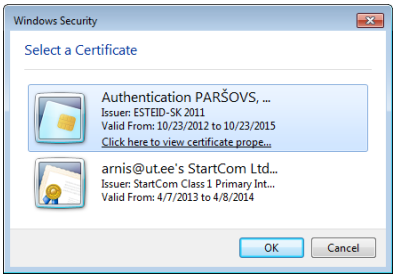
Unfortunately, similar functionality is not provided by Chrome [16]. However, Chrome clears the client certificate



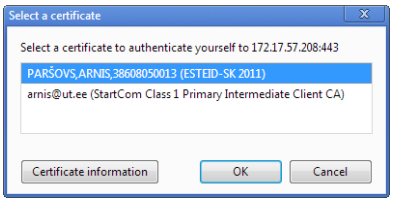
(a) Firefox under Linux



(b) Chrome under Linux



(c) IE under Windows 7



(d) Chrome under Windows 7

Fig. 2: Client certificate selection windows

choice if the TLS CCA handshake is not successful, thereby preventing deadlock in case the user has chosen the wrong client certificate which is being refused by the server at TLS handshake level (Firefox and IE do not prevent deadlock in this situation). To implement CCA logout functionality in Chrome, the behavior described above can be used as a workaround. For example, by configuring on server side and then requesting in browser specific server resource that will fail TLS CCA handshake if requested:

```
<Directory /var/www/handshake_fail/>
SSLVerifyClient require
SSLVerifyDepth 0
</Directory>
```

In the case of IE, the IE specific JavaScript function `document.execCommand()` with the parameter `"ClearAuthenticationCache"` can be used. However, it should not be used by responsible web sites, since when called not only the TLS session and client certificate choice will be deleted, but also all session information including HTTP cookies and authentication, and not only for the site who called it but all sites in the current browser session [17]. In our opinion the behavior provided by this function creates a security issue itself.

The lack of standard JavaScript API that could be called by web sites to clear client certificate choice and TLS session cache, prevents web sites using TLS CCA to exercise login and logout functionality as currently available when HTML form-based password authentication is used.

An alternative solution is to implement the logout functionality in a browser user interface (UI). The suggested UI would show in the browser address bar the client certificate that has been used for TLS CCA to the server and would allow to change client certificate choice (see Fig. 3) [18].

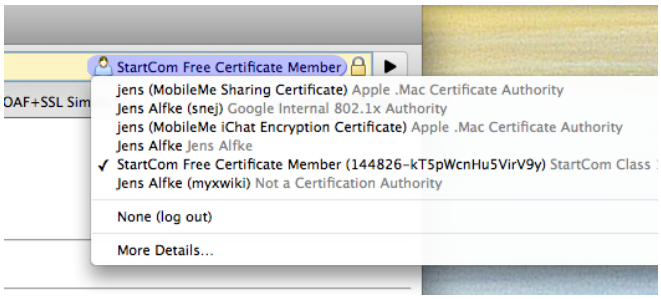


Fig. 3: Chrome TLS CCA UI improvements proposed [18]

Unfortunately, we find little value in these UI improvements, since this functionality will not be usable in cases where TLS CCA is used only in the authentication phase or when TLS CCA is performed by a separate server.

Another significant usability issue in multiple client certificate use scenario is that client certificate selection windows of the browsers provide client certificates sorted by some fixed rules and not by their use frequency.

J. *Client Certificate Enrollment*

In Estonia client certificates are distributed to residents by including them in their ID card, which is a mandatory identity document in the form of a smart card. The ID card contains a public-key certificate signed by the state supported CA and the corresponding RSA key pair that can be used for TLS CCA.

However, it is a wrong assumption that TLS CCA requires the use of PKI and CAs. A client certificate that is issued by a trusted CA is required only in cases where the client has no account on a server and where the server would be required to know the user's government issued identity for opening such an account. Most service providers care only about a user's identity established after the registration process. Therefore, any service provider in the process of account registration or after that can register a client's certificate that can be used to further authenticate the user. The server can register the client certificate by performing TLS CCA with a certificate chosen by the user or if the user does not have or does not want to use an existing certificate, the server can initiate the user's browser to perform key pair generation and certificate enrollment using HTML5 `<keygen>` element supported by both Chrome and Firefox. If the user has a cryptographic token (e.g., smart card) and proper PKCS#11 module loaded into the browser, the RSA key pair can be generated and certificate loaded right into the token. In IE similar functionality exists using CertEnroll ActiveX control [19]. Using a client certificate just as a transport for a public key that is bound to the user's account does not benefit from centralized certificate revocation and replacement, but password-based authentication does not have these benefits either.

Compared to password authentication, CCA has a client certificate portability problem. Client certificates loaded into cryptographic tokens are portable, however, modern mobile devices that are increasingly used for browsing do not have interfaces for cryptographic tokens or if they have, the use of external hardware with mobile devices is rather inconvenient. Certificates and their corresponding private keys which are stored in a browser can be manually exported and imported in other devices. Unfortunately, Firefox Sync[3] – a secure browser synchronization feature in Firefox, does not provide synchronization of client certificates. As an alternative, password authentication could be used to authenticate to a service provider in order to obtain a new device specific client certificate.

K. Security Against Server Impersonation

Recent incidents involving compromise of trusted CAs DigiNotar [20] and Comodo [21] and reported misbehavior of CAs Trustwave [22] and TURKTRUST [23] show that MITM attacks using fraudulent certificates issued by trusted CAs are very realistic. Even if an attacker is not powerful enough to obtain a certificate issued by a trusted CA, he can perform a MITM attack using a self-generated certificate and if the victim clicks through browser security warnings, achieve successful impersonation of the legitimate server.

As it was stated before, if the TLS CCA is used, even a very powerful attacker that can successfully impersonate a legitimate server, cannot reuse the proof given in a `CertificateVerify` message to impersonate a victim to the legitimate server. Unfortunately, such an attacker can still compromise the client's account on the legitimate server by exploiting design features of modern web browsers. In particular, same-origin policy [24] which will not isolate client-side scripts that are served over impersonated TLS connection to the attacker and client authenticated TLS connection to the legitimate server. As a result, such an attacker will be able to execute a cross-site scripting attack in the context of the legitimate server which is enough to retrieve information available to the victim's browser and to execute actions chosen by the attacker [25].

While this attack brings additional complexity to the MITM attack, an attack like this cannot be excluded in highly targeted attacks.

Note that this complicated cross-site scripting attack is required only in case the legitimate server requires TLS CCA also after the authentication phase. If the legitimate server binds the authenticated session only to HTTP cookie, the attacker can impersonate the client by obtaining the HTTP cookie that will be disclosed by victim's browser's same-origin policy sending it over the impersonated TLS connection.

In order to prevent these attacks, a browser's same-origin policy should be changed to isolate content that is served over connections that are authenticated with different server certificates. We encourage browsers to implement support for strong locked same-origin-policy as described in [25] and enable web sites to opt-in similarly as it is possible to opt-in for HTTP Strict Transport Security policy [26]. The availability of strong locked same-origin-policy would also bring a practical security value to Extended Validation (EV) server certificates, since an attacker who has fraudulently obtained a low-assurance Domain Validation (DV) certificate would not be able to hijack an EV authenticated connection to the legitimate server [27].

Note that an opt-in for a strong locked same-origin-policy will not help against attackers who are capable of impersonating TLS servers, if the web site is importing and exporting content to other origins [28].

IV. Estonian Case Study

A. Background

Every Estonian citizen who has reached 15 years of age must obtain a state issued ID card in the form of a smart card that contains two public-key certificates and corresponding RSA key pairs. One can be used for TLS CCA and another for qualified digital signatures. This has led to the high popularity of TLS CCA in Estonia where every major service provider has implemented a TLS CCA option for authentication to their e-services. We have analyzed 87 public Estonian web sites (all that we were aware of) which provide a TLS CCA option.

The goal of the case study was to find out how TLS CCA is deployed in practice and how the issues described in this paper are handled in real world TLS CCA deployments. All service providers were tested using a black-box method, therefore, the results we got might have other explanations than the ones provided here.

The tests were performed in the time period between March 5, 2013 and April 19, 2013. The service providers and testing results related to `mod_ssl` deployments are provided in Table I. While this paper's focus is on Apache `mod_ssl` TLS CCA implementation, we have also performed common tests on other CCA implementations provided in Table II. Our guesses of the server software used are provided in column "Server".

B. Evaluation and Results

The testing methodology and results are discussed below.

1) Spare CAs: The certificates that are included in an Estonian ID card are issued by SK intermediate CA ESTEID-SK 2007. Since the SK root CA certificate will expire soon, the certificates issued after July 2011 are issued by intermediate

[3]https://www.mozilla.org/mobile/sync/

CA ESTEID-SK 2011 that itself is issued by the new root CA of SK. Therefore, the server side trust store should have 4 certificates configured – two SK root CA certificates and two corresponding intermediate CA certificates.

We have analyzed a list of CA DNs provided in the server's `CertificateRequest` message to detect spare (possibly superfluous) and missing certificates in the server's trust store. The list of spare certificates have been pruned by removing certificates that have been issued by these two SK root CAs, since the presence of intermediate certificates in the trust store do not affect trust settings as described in Section III-B.

The positive number in the column "Spare" most likely indicates a server misconfiguration and may allow an attacker to impersonate a user if the attacker is able to obtain the client certificate that contains the user's personal data and is signed by any of the spare CAs or their subordinate CAs.

Some of these misconfigurations can be explained by a bad undocumented `mod_ssl` practice, since certificates in `SSLCACertificateFile` are used also to build the server's certificate chain provided in the server's `Certificate` message. We have found several `mod_ssl` tutorials on the Internet in which readers are instructed to place CA certificates of the server's certificate into the `SSLCACertificateFile` rather than `SSLCertificateChainFile`. Our observations confirm this narrative, since most of the spare CAs are CAs that have also issued a server's certificate.

2) Missing CAs: The absence of SK root CA DNs or an empty list of DNs in a server's `CertificateRequest` message will not create a problem. However, the presence of root CA DNs without intermediate DNs will make TLS CCA fail if the certificate store in the user's browser does not have these intermediate certificates installed. The service providers `e-ope.ee`, `justask.ee`, `ox.ee` and all service providers using Microsoft Internet Information Services (IIS) were found to be missing one or both intermediate CA DNs in their `CertificateRequest` message.

The service providers `g4s.ee`, `justask.ee`, `ox.ee` and `ut.ee` were found to be missing the new root CA certificate of SK in their trust store, thereby denying TLS CCA with client certificates issued after July 2011.

3) Verification Depth: The verification depth value set by `mod_ssl SSLVerifyDepth` directive (column "Depth" in Table I) was enumerated using the method described in Section III-C. As it was explained in Section III-B, the `mod_ssl` does not allow to limit intermediate CA certificates that the valid client certificate chain may have. However, at least the length of the chain can be limited using the `SSLVerifyDepth` configuration directive. The correct value for the Estonian ID card certificate chain is 2, since in addition to the end entity certificate the chain contains intermediate and root CA certificates. However, as can be seen in Table I, almost half of the service providers have extended their verification depth constraint. While it does not create an immediate threat, the depth constraint of 2 can prevent attacks in certain cases, for example, if the CA mistakenly enables CA trust bits in the end entity certificate and does not have path length constraints in CA certificates as it was in TURKTRUST incident [23].

4) CCA Bound Sessions: The column "Bind" tells whether the service provider requires the presence of a CCA TLS connection only at the login phase or also after when the user performs actions in the authenticated environment.

TABLE I: TLS CCA as deployed by the service providers running Apache `mod_ssl`.

Service provider	Spare	Request	Depth	Timeout	Privacy	Resume	Bind	Validity
Banking:								
citadele.ee	0	optional	3	–	–	+	+	OCSP
krediidipank.ee	5	optional	2	–	–	+	+	OCSP
seb.ee	2	optional	2	–	–	+	+	OCSP
tbb.ee	4	require	3	–	+	–	–*	OCSP
unicreditbank.ee	0	require	5	–	–	+	?	?
versobank.com	0	optional	3	40 sec	–	+	?	?
Education:								
e-ope.ee	4	optional	3	–	–	+	+	–
eek.ee	0	require	2	–	+	–	?	?
ekool.eu	0	optional	2	–*	+	–	–	–
emu.ee	0	require	2	–	+	–	–	OCSP 2007
tlu.ee	0	require	2	9 hour	–	+	–	?
Government:								
ariregister.rik.ee	0	optional	6	–	+	–	–	CRL
digidoc.sk.ee	4	require	2	–	+	–	–	OCSP
e-register.ee	0	require	3	–	+	–	–	OCSP
e-toimik.ee	0	optional	2	–	+	–	–	OCSP
eesti.ee	5	require	2	–*	+	–	–	OCSP
emta.ee	0	optional	3	–	–	+	–	OCSP
epa.ee	0	require	2	–*	+	–	–	–
ettevotjaportaal.rik.ee	105	optional	6	–	+	–	–	CRL
kala.envir.ee	0	require	10	–*	+	–	?	?
kassa.fin.ee	31	require	3	–	+	–	?	?
kinnistuportaal.rik.ee	2	optional	2	–	+	–	–	CRL
kinnistusraamat.rik.ee	0	optional	2	–	+	–	–	CRL
paberivaba.ark.ee	0	optional	3	–*	+	–	–	–
pensionikeskus.ee	2	require	2	–	+	–	–	OCSP 2007
politsei.ee	1	require	2	–	+	–	–	OCSP
pria.ee	0	require	2	–*	+	–	–	–
riigihanked.riik.ee	31	require	2	–	+	–	–	CRL
sk.ee	7	require	2	–	+	–	–	OCSP
stat.ee	73	require	2	–	+	+	–	CRL
tootukassa.ee	9	optional	3	–	+	–	–	OCSP
Health:								
arst.ee	0	require	3	–	+	–	–	CRL
medicum.ee	0	require	2	–	+	–	–	–
Insurance:								
ergo.ee	0	require	2	≈30 min	+	–	–	CRL
iizi.ee	2	require	3	≈15 min	+	–	?	CRL 2007
mandatumlife.ee	0	require	3	–	+	–	?	?
Utility:								
arhivaar.ee	0	optional	2	≈15 min	+	–	–	CRL
arvekeskus.ee	2	require	3	–	+	–	–	OCSP
e-seif.ee	0	require	9	–	+	–	–	–
eestiloto.ee	0	require	3	–	–	+	+	OCSP
elisa.ee	1	optional	2	–	+	–	–	OCSP
energia.ee	2	require	2	9 hour	–	+	+	CRL
eparkimine.ee	3	require	3	–*	+	–	–	–
g4s.ee	0	require	2	≈30 min	–	–	?	CRL 2007
gaas.ee	0	require	2	–*	+	–	?	OCSP
korteriyhistu.net	0	optional	2	–*	+	–	–	OCSP
parkimine.ee	2	require	3	–	+	–	–	–
pilet.ee	0	require	2	–	+	–	–	–
stv.ee	1	require	3	–*	+	–	–	CRL
tallinn.ee	0	require	2	–	+	–	–	–
tele2.ee	9	optional	3	–	+	–	–	OCSP
zone.ee	1	optional	3	–	–	+	–	CRL
Other:								
apollo.ee	2	require	3	9 hour	+	–	–	CRL
auto24.ee	0	optional	2	–	–	+	–	CRL
credit24.ee	0	require	2	–	–	+	–	OCSP
era.ee	0	require	2	20 sec	–	+	+	–
hinnavaatlus.ee	0	require	3	–	–	+	–	CRL
justask.ee	0	optional	3	–	+	–	?	?
laen.ee	0	require	3	9 hour	+	–	–	CRL
openid.ee	0	require	3	40 sec	–	+	–	OCSP
osta.ee	0	optional	2	9 hour	–	+	–	CRL
ox.ee	0	optional	3	–	+	–	–	–
partnerkaart.ee	2	require	2	–	+	–	?	?
rahvakogu.ee	0	require	2	–*	+	–	–	–
skaart.ee	0	require	2	–	+	–	–	–

TABLE II: TLS CCA as deployed by the service providers running other TLS server software.

Service provider	Server	Spare	Request	Timeout	Privacy	Resume	Bind	Validity
Banking:								
danskebank.ee	BigIP	0	optional_any	1 min	–	+	+	OCSP
lhv.ee	IIS/6.0	24	optional_any	6 min	+	+	+	OCSP
nordea.ee	?	6	require	1 min	–	+	–*	OCSP
swedbank.ee	BigIP	0	optional_any*	1 hour	–	+	+	OCSP
Education:								
ehis.ee	Oracle-AS	10	optional	5 min	–	+	–	?
etis.ee	IIS/7.5	6	optional_any*	2 min	+	+	–	CRL
ttu.ee	Oracle-AS	14	optional	5 min	–	+	–	?
ut.ee	Oracle-AS	0	optional	5 min	–	+	–	CRL 2007
Government:								
eas.ee	IIS/7.5	13	optional_any*	2 min	+	+	–	OCSP
Health:								
digilugu.ee	BigIP	0	optional_any	10 sec	–	–	–	OCSP
digiregistratuur.ee	IIS/6.0	27	optional_any	5 min	+	+	–	OCSP
Insurance:								
compensalife.eu	IIS/6.0	21	optional_any*	5 min	+	+	?	?
kindlustus.ee	IIS/7.0	5	optional_any*	2 min	+	–	–	CRL
Utility:								
arved.ee	?	1	optional	–	+	+	–	OCSP
dormitorium.ee	IIS/7.x	9	optional_any*	2 min	+	–	–	CRL
elektrum.ee	Nginx	0	optional_any	1 min	–	–	–	OCSP
elering.ee	Jetty?	1	require	–	–	+	+	OCSP
elion.ee	Tomcat?	0	optional_any	1 min	–	+	–	–
emt.ee	BigIP	0	optional_any	10 sec	–	–	–	OCSP
imatraelekter.ee	IIS/6.0	25	optional_any*	6 min	+	+	–	CRL
kyla.ee	IIS/6.0	20	optional_any	6 min	+	+	–	CRL
tallinnavesi.ee	IIS/7.5	7	optional_any*	2 min	+	–	–	CRL

We see that only a few service providers (mostly banks) do require a CCA connection also after the authenticated HTTP session has been established. The lack of CCA bound sessions can be explained by the additional complexity that is required if the support for password authentication must also be provided. Instead of setting up two URL locations for CCA and non-CCA authenticated users, the service providers set a CCA URL where the user is redirected only at the login phase, but after an authenticated HTTP session is established (by use of HTTP cookie) the browser is redirected back to the common server path where CCA is not required. This type of federated authentication approach has more security risks, since the authenticated session is protected only by an HTTP cookie which might be stolen by an attacker capable of impersonating the server or leak in another way (e.g., via cross-site scripting attack). In contrast, TLS key material from a client certificate authenticated TLS session may be obtained by the attacker only if the attacker is able to compromise one of the endpoints.

It is interesting to note that while the service providers `tbb.ee` and `nordea.ee` required a TLS CCA connection also after the login phase, the application did not verify whether the client certificate presented is of the same person that was presented at the login phase.

In some service provider web sites we did not have an account, therefore, we could not test if the authenticated session is bound to the CCA or not. For these service providers the rows are marked with a question mark.

5) Privacy of Client's Certificate: The column "Privacy" shows whether the service provider protects the privacy of a client certificate by requesting CCA on renegotiation or the CCA is performed on initial TLS negotiation and the client certificate is sent to the server over an unencrypted channel.

We see that one-third of all service providers do not protect the privacy of client certificates. This can be explained by the `mod_ssl` documentation that does not warn about the consequences if the `SSLVerifyClient` directive is configured in a server wide context.

Another explanation could be that renegotiation has been disabled on the server side as a temporary fix when TLS renegotiation vulnerability (CVE-2009-3555) was discovered, and has been left disabled also after the TLS renegotiation indication extension [29] that fixes the vulnerability was standardized and implemented in browsers.

6) CCA Session Resumption: The column "Resume" indicates whether the server supports TLS session resumption of client certificate authenticated sessions. We see that in `mod_ssl` deployments the CCA sessions that have been established on initial negotiation do support resumption, while all deployments (except `stat.ee`) that require CCA on renegotiation do not support resumption for these sessions. This can be the result of the `mod_ssl` bug described in Section III-D.

While most of the service providers fail to resume CCA sessions, the support for CCA session resumption is only important for these service providers who serve the content over TLS CCA connection also after the login phase.

7) CCA request: For `mod_ssl` deployments the column "Request" shows the value configured by the `SSLVerifyClient` directive. As described in Section III-A the value `require` will result in a failed TLS handshake if for whatever reason the verification of the client certificate fails. Part of the service providers use the value `optional` that will allow the service provider to provide possibly personalized HTTP response in case the client has no client certificate. In all other cases the handshake will fail with an error message provided by the browser.

With non-`mod_ssl` deployments (Table II) the situation is different. Most of them at the TLS handshake level will tolerate the absence of a client certificate and presence of any client certificate as long as the client can prove possession of the corresponding private key (marked as `optional_any`; `optional` and `require` mimics `mod_ssl` behavior). If the client provides an invalid certificate, the HTTP error message is returned by the web server or by the application.

We have observed that the service providers `compensalife.eu`, `dormitorium.ee`, `eas.ee`, `etis.ee`, `imatraelekter.ee`, `kindlustus.ee` and `tallinnavesi.ee` (all using IIS) allow successful authentication with a qualified digital signature certificate whose key usage is inconsistent with TLS CCA (certificate key usage extension has only `nonRepudiation` bit asserted). Apparently this certificate verification step is missed by the IIS server and therefore must be performed by the application. We have informed Microsoft about the flaw.

The service provider `swedbank.ee` was found to miss not only certificate key usage verification, but also signature verification on a client certificate received. The flaw allowed to successfully login into any user account if the user ID of the person was known. The service provider has been informed about this flaw and the flaw has already been fixed.

8) CCA Handshake Timeout Enforcement: The column "Timeout" shows for how long the handshake can be kept open after the server has sent the `ServerHello` message, but before the client has sent the `CertificateVerify` message. In the case of service providers marked "–", we were able to keep the connection open for more than 12 hours. The service providers marked with "*" had `mod_reqtimeout` enabled, but since the CCA was requested on renegotiation, we could still keep the connection open using the method described in Section III-F.

To measure timeout for non-`mod_ssl` deployments which did not support empty TLS record fragments (Oracle-AS, `danskebank.ee`, `nordea.ee`) and TLS record fragmentation (IIS 6.0), we appended the client's `Certificate` message with dummy certificates to enlarge the handshake and sent TLS records split to 1 byte TCP packets, one per second.

9) Client Certificate Revocation Checking: The validity of authentication certificates issued by SK can be checked using CRLs, which are issued every 12 hours or by using SK paid validity service provided using OCSP.

To test whether the service provider performs revocation checks for client certificates issued under both SK intermediate CAs, we used two client certificates issued to one of the authors of this study – ID card certificate issued by SK intermediate CA ESTEID 2007 and Digi-ID[4] certificate issued by SK intermediate CA ESTEID 2011.

If the access was denied right after the certificate was revoked, we concluded that OCSP is being used. If the access was denied only after the serial number of the revoked

[4]Estonian residents may also obtain a Digi-ID smart card. The only difference from an ID card is that it cannot be used for physical identification and the `organizationName` attribute of subject DN in Digi-ID authentication certificate contains "ESTEID (Digi-ID)" instead of "ESTEID".

certificate appeared on CRL, we concluded that CRL is used. If we could successfully authenticate even after `nextUpdate` specified in the previous CRL was reached, we concluded that no revocation checks are performed (marked as "–").

In some service provider web sites we did not have an account, therefore, we could not test whether the revocation checks are performed on the application level after login. In these cases the rows are marked with a question mark.

We observed a strange behavior in the case of service providers `pensionikeskus.ee` and `emu.ee`. When our ID card certificate was revoked, the access using Digi-ID certificate was also denied by the service provider. We suspect that these service providers use LDAP instead of OCSP for certificate revocation checking. According to the ESTEID Certification Policy [30], the SK LDAP directory contains only valid unexpired certificates, therefore LDAP can be used to check certificate status. However, these service providers apparently use fixed `organizationName` field in the LDAP search query to check the existence of any ID card authentication certificate in LDAP issued to the person, without comparing the certificate in LDAP with the client certificate received. In fact, it is likely that more service providers marked "OCSP" use LDAP for revocation checking.

The use of LDAP for revocation checking in addition to CRL and OCSP has a security advantage, since not only the serial number of the certificate can be verified, but the whole certificate. This will allow to detect fraudulent certificates that have been issued with the serial numbers of legitimate ones as seen in DigiNotar compromise [20]. However, since the LDAP traffic is not cryptographically protected, it cannot be trusted in case of the attacker who is able to perform a MITM attack between the service provider and SK LDAP.

When testing handshake timeout enforcement, we observed that in the case of service providers `apollo.ee`, `energia.ee`, `laen.ee`, `osta.ee` and `tlu.ee` the TCP connection is closed a short before the time specified in `nextUpdate` of CRLs is reached. Apparently these service providers use a non-graceful Apache restart after updating CRLs.

10) Other Observations: We did not observe any service provider using `window.crypto.logout()` to implement CCA logout functionality.

The service provider `seb.ee` was the only one that, after logging out and trying to login, was able to detect that the TLS CCA session established on previous login is used. However, the service provider was still not able to clear the TLS session on the server side and asked the user to restart his browser in order to re-authenticate.

V. Our Recommendations

A. For Service Providers

The service providers deploying TLS CCA with the current options available, are suggested to make sure that:

- There are no spare CA certificates in the `SSLCACertificateFile`.
- The CA certificates that are direct issuers of client certificates are specified in the `SSLCADNRequestFile`.
- The additional checks are performed at the application level to verify that the client certificate is issued by the intermediate CA intended (at least the issuer name of client certificate is verified).
- The `SSLVerifyDepth` constrain is configured to the minimal value required.
- The `SSLVerifyClient` directive is specified in directory context in order to perform TLS CCA on renegotiation thereby preserving the privacy of the client certificate.
- An Apache module such as `mod_reqtimeout` is correctly used to enforce CCA handshake timeout.
- Client certificate revocation checks are performed and if LDAP is used, the client certificate is binary compared to the one found in the LDAP directory.
- To implement the CCA logout functionality, the Firefox specific `window.crypto.logout()` function and Chrome workaround described in Section III-I is used.
- The cipher suites providing perfect forward secrecy are disabled (unless forward secrecy is more important than audit functionality) and packet captures of TLS handshake messages are saved.
- In case of highly sensitive services the authenticated session is bound to TLS CCA (although it has a negative performance impact from the `mod_ssl` bug described in Section III-D).

Meanwhile, we encourage service providers all over the world to implement a PKI-less TLS CCA option as described in Section III-J.

B. For `mod_ssl` Developers

As a relatively simple fix, the `mod_ssl` bugs [10] and [6] should be fixed, and other undocumented issues pointed out through this paper should be reflected in the official `mod_ssl` documentation [4].

Next, we suggest a `mod_ssl` redesign that would provide a flexible yet simple and secure configuration for most TLS CCA use cases. We suggest three configuration values for `SSLVerifyClient` configuration directive:

- `none` - CCA is not required (current `none` behavior)
- `require_success` - require successful CCA or TLS handshake fails (current `require` behavior)
- `require_any` - optionally request any client certificate.

The environment variable `SSL_CLIENT_VERIFY_RESULT` should be set based on the verification result – `NONE`, `SUCCESS` or `FAILED:`*`reason`* (the verification must fail on first error to reduce DoS attack vectors).

In addition, the certificate chain building process should succeed when any certificate loaded in the trust store is reached. This will allow to enforce verification using specific intermediate CA certificate and authorize specific client certificates by storing them directly in the trust store.

The `SSLVerifyClient` value `require_success` is useful for closed deployments where no HTTP requests should be processed for unauthenticated clients. On the other hand, the value `require_any` gives opportunity for application to provide a personalized error messages in the case of CCA failure, and at the same time allows to use client certificates just as a transport for the public key whose authenticity is established by an out-of-band mechanism (PKI-less CCA).

In order to enable TLS CCA re-authentication, we recommend that special response header (e.g., `X-TLS-Reset`) is introduced, that if set by server-side application, would force `mod_ssl` to delete associated TLS session, after returning the response.

Finally, the application level should have access to the timestamp that specifies the freshness of the proof given in the TLS CCA process and the certificate chain used in the verification process. The logging of decrypted TLS handshake messages as described in Section III-H should also be implemented.

C. For Browser Vendors

Browser vendors analyzed in this study can relatively easily improve security and usability of TLS CCA by:

- Showing a warning message that informs about the privacy leak if the client certificate is requested on initial negotiation.
- Clearing the cached client certificate choice if the TLS CCA handshake fails (applies to Firefox and IE).
- Using or falling back to (if request fails with `417` response code) `Expect: 100-continue` mechanism when sending large HTTP POST requests over TLS.
- Making the client certificate used for the CCA to the server last time as the default certificate choice in the client certificate selection window.

A bigger challenge, however, is to standardize JavaScript API that could be used separately to delete the TLS session cache from a client side (thereby providing support for client certificate re-authentication in the background) and the cached client certificate choice (thereby providing TLS CCA logout functionality).

In addition, we encourage to extend the HTTP Strict Transport Security policy [26] mechanism with opt-in for strong locked same-origin-policy.

VI. Related work

Ristic in his SSL Labs research effort [31] has created TLS deployment best practices, TLS server rating guide and performed an Internet TLS survey that analyzes TLS server deployments on the Internet. However, his work does not cover aspects of TLS CCA.

Hess et al. in [32] describe the TLS protocol related limitations of CCA. While the issues listed there can be a limitation in a specific use scenarios, they are not a stumbling-block for everyday TLS CCA use case. Some limitations enumerated there, such as client certificate disclosure, can be easily solved by renegotiation as described in this paper.

Dietz et al. in their work [33] give a list of reasons why TLS CCA does not work in today's web. In our opinion, all the issues listed there (with the exception of the portability issue) can be solved if the recommendations provided in this paper are implemented.

VII. Conclusion

In this paper we have described several issues related to TLS CCA use in practice and have provided a list of solutions that can be implemented at an application level without requiring changes to the TLS protocol. While this study has shown that there is a room for improving TLS CCA implementations, the Estonian example shows that the problem of secure user identity on the Internet has a solution and more importantly the solution works in practice. We hope that the suggestions for improvements provided in this paper find support in the respective communities thereby proliferating TLS CCA use on much larger scale.

Acknowledgement

We would like to thank Dominique Unruh, Peter Gutmann, Ivan Ristic, Tiit Pikma and Martin Paljak (CERT-EE) for their feedback on this paper.

This research was supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, Estonian National Electoral Committee and Estonian Doctoral School in Information and Communication Technology, IKTDK.

References

[1] D. Wendlandt, D. G. Andersen, and A. Perrig, "Perspectives: improving SSH-style host authentication with multi-path probing," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 321–334. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404041

[2] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC draft, Internet Engineering Task Force, Apr. 2013, http://tools.ietf.org/html/draft-laurie-pki-sunlight-12.

[3] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246 (Proposed Standard), Internet Engineering Task Force, Jan. 1999, http://www.ietf.org/rfc/rfc2246.txt.

[4] "Apache mod_ssl module," http://httpd.apache.org/docs/2.2/mod/mod_ssl.html.

[5] "Bug 2768: internal_verify() hides errors from callbacks," http://rt.openssl.org/Ticket/Display.html?id=2768.

[6] "Bug 53193: SSLVerifyClient optional_no_ca wrong SSL_CLIENT_VERIFY," https://issues.apache.org/bugzilla/show_bug.cgi?id=53193.

[7] "OpenSSL Verify Operation," http://www.openssl.org/docs/apps/verify.html.

[8] AS Sertifitseerimiskeskus, "Configuring Apache web server to support ID-card certificates. v1.03," http://www.id.ee/public/Configuring_Apache_web_server_to_support_ID.pdf.

[9] A. Langley, "Transport Layer Security (TLS) Encrypted Client Certificates," Internet Draft, Oct. 2011, http://tools.ietf.org/html/draft-agl-tls-encryptedclientcerts-00.

[10] "Bug 47055: SSLVerifyClient + Directory doesn't use cache sessions," https://issues.apache.org/bugzilla/show_bug.cgi?id=47055.

[11] "ASN.1 Denial of Service Attacks (CVE-2006-2937, CVE-2006-2940)," http://www.openssl.org/news/secadv_20060928.txt.

[12] "Apache mod_reqtimeout module," http://httpd.apache.org/docs/2.2/mod/mod_reqtimeout.html.

[13] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient Padding Oracle Attacks on Cryptographic Hardware," INRIA, Rapport de recherche RR-7944, Apr. 2012. [Online]. Available: http://hal.inria.fr/hal-00691958

[14] "Issue 90454: Feature: Add the ability to purge the SSL session cache for a browsing session," http://code.google.com/p/chromium/issues/detail?id=90454.

[15] Mozilla, "JavaScript crypto object," Jan. 2012, https://developer.mozilla.org/en/docs/JavaScript_crypto.

[16] "Issue 90676: window.crypto.logout() and login() don't work," http://code.google.com/p/chromium/issues/detail?id=90676.

[17] E. Lawrence, "Understanding Session Lifetime," http://blogs.msdn.com/b/ieinternals/archive/2010/04/05/understanding-browser-session-lifetime.aspx.

[18] "Issue 29784: User Interface Improvement for Client Certificate Usage," http://code.google.com/p/chromium/issues/detail?id=29784.

[19] A. C. Magencio, "How to create a certificate request with CertEnroll," Jan. 2009, http://blogs.msdn.com/b/alejacma/archive/2009/01/28/how-to-create-a-certificate-request-with-certenroll-javascript.aspx.

[20] Fox-IT BV, "Report of the investigation into the DigiNotar Certificate Authority breach," Aug. 2012, http://www.rijksoverheid.nl/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update.html.

[21] Comodo, "Report of Incident," Mar. 2011, https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.

[22] Trustwave, "Clarifying The Trustwave CA Policy Update," Feb. 2012, http://blog.spiderlabs.com/2012/02/clarifying-the-trustwave-ca-policy-update.html.

[23] TURKTRUST, "Public announcements concerning the security advisory," Feb. 2013, http://turktrust.com.tr/en/kamuoyu-aciklamasi-en.html.

[24] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*, 1st ed. San Francisco, CA, USA: No Starch Press, 2011.

[25] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 58–71. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315254

[26] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," RFC 6797 (Proposed Standard), Internet Engineering Task Force, Nov. 2012, http://www.ietf.org/rfc/rfc6797.txt.

[27] A. Satirov and M. Zusman, "Breaking the security myths of extended validation SSL certificates," 2009, http://www.blackhat.com/presentations/bh-usa-09/ZUSMAN/BHUSA09-Zusman-AttackExtSSL-SLIDES.pdf.

[28] C. Jackson and A. Barth, "Beware of Finer-Grained Origins," in *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008. [Online]. Available: http://seclab.stanford.edu/websec/origins/fgo.pdf

[29] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension," RFC 5746 (Proposed Standard), Internet Engineering Task Force, Feb. 2010, http://tools.ietf.org/html/rfc5746.

[30] AS Sertifitseerimiskeskus, "ESTEID Card Certification Policy, Version 3.3," https://sk.ee/upload/files/SK-CP-ESTEID-20120901v3_3_en.pdf.

[31] I. Ristic, "SSL Labs Projects," https://www.ssllabs.com/projects/index.html.

[32] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith, "Advanced Client/Server Authentication in TLS," 2002.

[33] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach, "Origin-bound certificates: a fresh approach to strong client authentication for the web," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362809