SSL Server Rating Guide for TLS Client Certificate Authentication

Seminar Report for Research Seminar in Cryptography

Kristen Gilden (A93721)
Author

Arnis Paršovs
Supervisor

***Abstract*—This paper presents a list of tests that can be automatically run to verify the correct server configuration of TLS Client Certificate Authentication. A possible design for a testing engine with a web front-end is proposed to run these tests by a web browser without the need of browser extensions. Finally, a rating guide is proposed to summarize test results.**

1. Introduction

The Transport Layer Security (TLS) protocol is designed to allow client/server applications communicate in a way to prevent eavesdropping, tampering, or message forgery [1]. It has gained mainstream adoption over the years with 43% of top 100 websites using modern TLS setup to encrypt their traffic [2].

The mainstream setup is to have only the server provide a certificate of proof claiming it really is the expected service and not a malicious third-party. By means of X.509 Public Key Infrastructure clients check whether the given certificate is the cryptographically signed descendant of any of its trusted certificates. On the other hand, clients typically don't present any certificate — instead a combination of username and password is used. However, TLS also specifies the use of client certificates. Clients may present their own certificate which the servers can use to authenticate the client.

Use of TLS Client Certificate Authentication (hereinafter CCA) is rather uncommon. The exception here is Estonia, which has deployed a national Public Key Infrastructure (PKI) — each citizen is by law required to obtain a government-issued smart-card, which contains personal certificates. Local service providers have taken advantage of this and today numerous web services in Estonia support client certificate authentication.

Configuring a web server to securely implement TLS is non-trivial. Most older versions of TLS (including SSL) contain vulnerabilities. Certain configurations leave the server exposed to exploits, effectively negating the use of TLS. Choosing the list of supported ciphers is a balancing act between security and client compatibility. Fortunately, tooling against these problems is rather advanced. However, due to the limited use of TLS CCA, there is little material on secure configuration.

The purpose of this paper is to propose a design for building an automatic service, which would be able to verify TLS CCA configuration. Arnis Paršovs has extensively studied problems arising with TLS CCA [3]. This paper shall build on his work to demonstrate how probing for these problems could be solved in an automated fashion. It shall also propose a rating scheme to objectively compare configurations of different services. Finally, the paper shall describe one possible design of such a service.

1.1. Scope

The paper shall focus on passive tests only. None of the tests should require access to a user's private key. These tests can listen data exchanged during TLS session negotiation or establish a TLS session with either the target server or client browser.

Furthermore, only Apache's TLS implementation is considered. The Apache web server implements TLS by a separate module, `mod_ssl` [4]. Thus some of the tests specifically exploit the quirks of this setup. For example, in section "Verification Depth" the described method only works, if the target is using `mod_ssl`. Tests for other TLS server implementations will be left for future versions of the planned system.

No restrictions are imposed as to which concrete Certificate Authority is being trusted by the server to have issued client certificates. The tests described should work for the majority of authorities and their certificates. However, since the reference paper [3] is targeting Estonia's national X.509 PKI, then the automatic tests are mostly targeting problems found there.

2. Automated Checks

In this section the tests described can be run without requiring any human input. Some of the tests can require the user to use their private key upon initiating a new TLS session, yet none require the user to ever reveal their private key. This is because some of the tests need access to the user's valid client certificate.

2.1. Client Certificate Privacy

If a server has enabled TLS CCA on a per-server context, the client certificates will be sent to the server in plaintext

during TLS connection establishment [4]. This can be a significant violation of the client's privacy, because the certificate might contain personally identifiable information. Upon failing the test, the user is advised to move the `SSLVerifyClient` mod_ssl directive to a per-directory context.

The automatic check must distinguish between 2 cases in order to determine whether the server leaks the client certificate. A server passes the test, if the TLS handshake follows the procedure outlined in Fig. 1. Conversely, it fails the test, if the server behaves like described in Fig. 2. In any other case (e.g. the `certificate_request(13)` is never sent, server not supporting TLS) the case is considered not applicable.

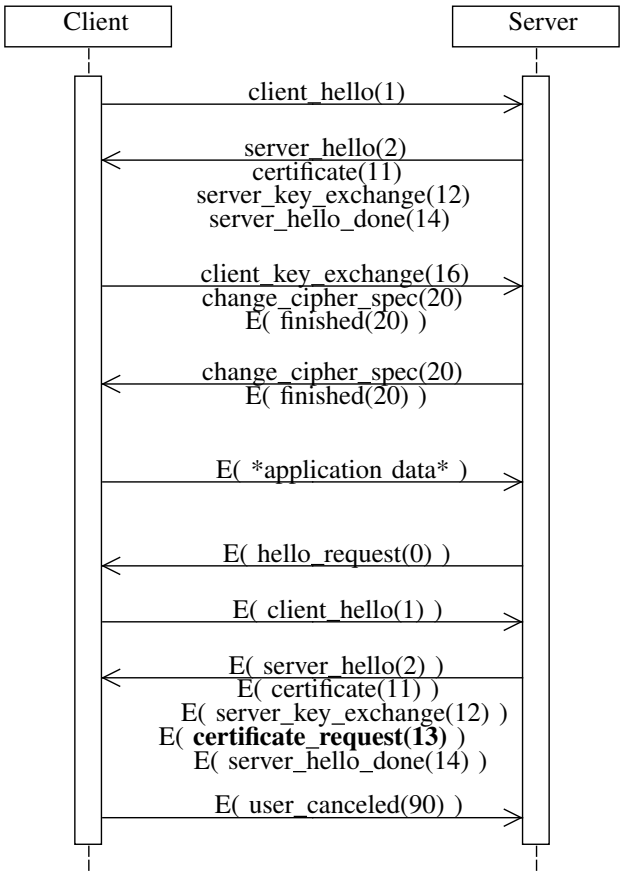

Figure 1. Preserving privacy in TLS CCA.

2.2. Missing CAs

In order for a server to verify client certificate's validity, it has to have access to all certificates chaining down to a client's certificate from a server's trusted certificate by means of signatures. If a server has not been configured to include these intermediate certificates within the TLS handshake (i.e. the Distinguished Names of these certificates are missing in the `certificate_request(13)` TLS handshake message) and the client is also missing them,

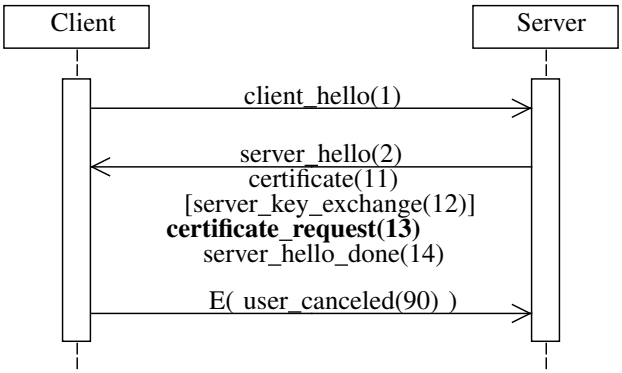

Figure 2. TLS CCA leaking Client Certificate.

then TLS CCA will fail [3]. This is not a security issue per se, but may prevent legitimate users from accessing the service.

In this test we can initiate the TLS handshake and wait for `certificate_request(13)`. The issuer Distinguished Name (DN – used to uniquely identify a subject within the X.509 certificate system) of the client's certificate is already known prior to starting the test. The test will pass, if the issuer DN is found within the list of given DN-s. Otherwise, we can conclude that either the server expects its clients to also present intermediate certificate chain or the given certificate is not trusted by the server. Either way the test fails.

2.3. Spare CAs

Detecting whether a server is overly trusting and accepts clients of too many CA-s would require domain knowledge. However, a commonly occurring bad practice is that server's certificate chain is declared under `mod_ssl` directive `SSLCACertificateFile` instead of `SSLCertificateChainFile`. This in turn results in the server also trusting any certificate signed by the server's certificate CA [3]. This is something which can be detected automatically.

To make sure the server has not been mistakenly configured to trust all of its server certificate CA's signed certificates, we can run the following test. First, the DN of the client certificate is noted. Then a TLS handshake is initiated with the server. The server presents its chain of certificates in the `certificate(11)` message. Next we obtain the list of supported DN-s from the server's `certificate_request(13)` message. The test passes, if these two sets of DN-s do not intersect. In the event of a failure the system will instruct the user to move its server certificates under the `mod_ssl` directive `SSLCertificateChainFile`.

Unfortunately this test will not be valid, if the server certificate CA is also used to sign client certificates. This would cause a false positive. This should be a fairly uncommon practice — as of present date no such configuration has been found. For example, AS Sertifitseerimiskeskus,

an Estonian CA, which issues personal certificates as well as server certificates, uses different certificates for signing identities of citizens and websites.

2.4. Verification Depth

A server should have as short certificate chain as necessary to trust a valid client certificate, which is controlled by the `mod_ssl` directive `SSLVerifyDepth`. [3] states that only the root certificate is required to determine the verification depth.

For this test the system has to start building incrementally longer certificate chains, where the root certificate would match the server's expected client root certificate. For `mod_ssl` the signatures do not need to be valid: the server will respond with a `certificate_unknown(46)` alert message once the chain length exceeds that of the `SSLVerifyDepth`. If the length is greater than what is necessary to verify the client's certificate, the test will fail.

If the server is missing some of its intermediate certificates and the user is also not presenting these, the test cannot be run. This is because for an arbitrary certificate chain there exists no standardized service to fetch these certificates. However, the testing system could leverage past test runs and try to use certificates from previous runs.

The test implies knowing the root certificate of the client certificate signature chain. While the root certificate is not usually presented by the server during the TLS handshake, it can be obtained pre-emptively. One such way would be to load the system with all trusted root certificates of all the major web browsers. Another way would be to let the users upload their own root certificates, but for the average user this would be too complicated.

2.5. CCA Request

Depending on the `mod_ssl SSLVerifyClient` directive the server either accepts any valid signed certificate (`optional_no_ca`), requires certificates of only trusted CAs (`require`) or allows client certificate to be absent (`optional`).

The test could first send a self-signed certificate. If the handshake succeeds, the directive must be set to `optional_no_ca` and the test is considered as failed. Otherwise the test could attempt a handshake without sending any certificate. If the handshake fails, we can conclude that the directive is set to `required` – a passing test; a succeeding handshake implies the `required` setting [3] and the test would be considered as "partially passed".

Note that the `optional` directive has a legitimate use case. Most applications tend to use this to present a user-friendly error message to the client. However, a buggy implementation causing the check to be by-passed would be a possibility, albeit unlikely.

2.6. CCA Handshake Timeout Enforcement

Testing for CCA handshake timeout enforcement as a third-party is complex. Only the simplest case can be tested. If the target server does not enforce CCA via TLS renegotiation and Apache's `mod_reqtimeout` [5] module is not enabled, the system can start sending empty TLS handshake messages once the server has sent its `server_hello(2)` message. If the connection can be kept up for at least 5 minutes, the test could already note this down as a failure.

When CCA is done via TLS renegotiation and `mod_reqtimeout` is enabled, the system would have to resort to a more elaborate trick. The system first establishes a regular TLS connection without CCA. Then, after the server has sent its second `server_hello(2)` message, the system can start sending at least 2 empty TLS records per second as described in [3]. Once again, if the connection can be kept up for at least 5 minutes, the test can be noted as a failure.

3. Architecture

All of the tests described in the previous section can be implemented without a need to be in possession of the client's private key. As such, the system could theoretically be a regular web application. There are several constraints and limitations inherent to the nature of the mutually authenticated connection. The following is a set of principles by which the system should be designed.

1) Scanning can be done by a standard web browser without a need for extra plugins. This severely limits the kinds of tests and usability of the system. However, this makes the application at least somewhat more usable by laypeople.
2) The client need not to upload their private key to the system. Requiring this would undermine the very purpose of this system — to improve security of TLS CCA applications on the web. Furthermore, some clients have their private key stored in a smart card, making this non-viable.
3) Historical reports must be accessible in the future. However, due to client certificates containing personally identifiable information, the choice should be left to the user.

3.1. System design

Figure 3 demonstrates the planned design of the checker. Arrows point away from the initiator. The central web application (Web App) is responsible for communicating with the client and providing an user interface, as well as showing the progress of the current scan. The application will be built on top of a web server, such as Apache or Nginx. Persistent data – scan status, previous reports – is stored in the database (DB).

For each scan the web application will create a detached process — Runner. It is responsible for coordinating the tests, collecting the results and persisting it in the database. Before running any of the tests, it retrieves the client and server certificates from the Sniffer. It then forwards all of this data to each of the tests (#1, #2, etc.), which can be

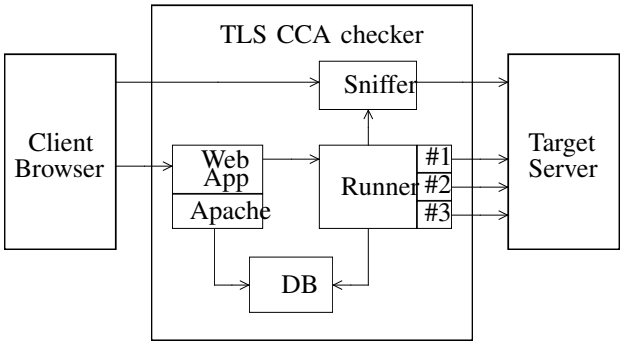

Figure 3. System Design of TLS CCA checker

run in parallel. Should the tests require access to the target server, they are free to do so.

The most interesting component of the entire system is the sniffer. Once the test is initiated, it will connect to the target server and record its TLS setup. It will then configure a per-scan virtual host and present a very similar `certificate_request(13)` message to the original client. The virtual host can have its server certificate either be as part of a wildcard certificate or a new certificate from Let's Encrypt [7] can be created on the fly. The host will then be injected to the scan page, forcing the client to load it and provide its client certificate. This way we have obtained certificates from both of the parties.

3.2. Rating

Ideally the rating should be a single value to convey the level of configuration correctness. Also, for regular TLS with just server certificates there already exists countless automatic rating systems. The predominant is the scanner by Qualys SSL Labs [8]. Similarly to Qualys, the CCA configuration checker shall have a final score in the range of $0 - 100$ and assigns a letter grade as follows.

1) A — score ≥ 85
2) B — score ≥ 65
3) C — score ≥ 50
4) D — score ≥ 35
5) E — score ≥ 20
6) F — score < 20

Each above mentioned test shall provide its own score in the range $0 - 100$. The result will be the average of all of the grades. If for some reason a test cannot be run, it will result in a score of 0. If a test is not applicable to the setup of the given server and the security of the service is not dependant on it (i.e. a test supports only Apache, but a Nginx server is being tested), then the final score will be computed as if this test does not exist.

4. Conclusion

Automatically checking TLS CCA configuration is a complex software engineering problem. This paper has demonstrated some of the tests, which can be run in an automated manner. Further work should be done to verify the findings by implementing these tests.

While this paper focused mostly on specific problems regarding Apache's `mod_ssl` plugin, further studies are needed to see which tests can be software agnostic and which extra tests would be necessary to test other platforms as well. Another unexplored direction would be designing a tool to check the configuration in a white-box manner — by running the tool on the server itself.

References

[1] T. Dierks, "The transport layer security (TLS) protocol version 1.2," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246. Accessed: Nov. 25, 2016.

[2] "Transparency report Google,". [Online]. Available: https://www.google.com/transparencyreport/https/grid/?hl=en. Accessed: Nov. 25, 2016.

[3] A. Parsovs, "Practical Issues with TLS Client Certificate Authentication," in NDSS, 2014.

[4] "Mod_ssl - Apache HTTP server version 2.4,". [Online]. Available: https://httpd.apache.org/docs/current/mod/mod_ssl.html. Accessed: Nov. 25, 2016.

[5] "Mod_reqtimeout - Apache HTTP server version 2.4,". [Online]. Available: http://httpd.apache.org/docs/2.4/mod/mod_reqtimeout.html. Accessed: Nov. 25, 2016.

[6] "SSL Server Rating Guide,". [Online]. Available: https://www.ssllabs.com/downloads/SSL_Server_Rating_Guide.pdf. Accessed: Nov. 25, 2016.

[7] "Lets Encrypt - free SSL/TLS certificates," 2016. [Online]. Available: https://letsencrypt.org/. Accessed: Nov. 25, 2016.

[8] "SSL server test," 2009. [Online]. Available: https://www.ssllabs.com/ssltest/index.html. Accessed: Nov. 25, 2016.