Everything You've Always Wanted to Know About Certificate Validation With OpenSSL (but Were Afraid to Ask)

Alban Diquet — alban[at]isecpartners[dot]com

iSEC Partners, Inc
123 Mission Street, Suite 1020
San Francisco, CA 94105
https://www.isecpartners.com

October 29, 2012

Abstract

Applications that need to securely communicate with a remote server commonly rely on the TLS[1]protocol. Various libraries are available for developers who want to leverage this protocol — one of the most popular being OpenSSL. Securely connecting to a server is not merely a matter of simply "enabling" TLS, but requires a great deal of care when implementing the client application in order to actually get the security benefits of using TLS. Most critically, TLS clients must validate the server's identity by verifying its X.509 certificate upon connection. Certificate validation is an extremely complex task, even when relying on a library such as OpenSSL. This paper provides application developers with clear and simple guidelines on how to perform certificate validation within a TLS client using OpenSSL.

1 Introduction

Although TLS is widely known for encrypting communication between two parties, verifying the identify of the server by validating its X.509 certificate is critical. Without proper certificate validation, the TLS client cannot ensure that it is talking to the real server instead of an attacker masquerading as the server. Failing to perform this step renders meaningless the confidentiality and integrity guarantees of TLS[2].

Implementing certificate validation within a TLS client application is critical, but it's also a complex task. At a high-level, this validation is a two-step process:

1. The server's certificate chain must be validated.
2. The server's hostname must be verified.

[1]Note that although OpenSSL supports both SSL and TLS, we refer almost exclusively to TLS in this document, as "SSL" is no longer current and should not be used.

[2]It is sometimes assumed that "something is better than nothing" with SSL/TLS, because it obscures traffic from a passive network monitor; however, given the negligible technical and cost differences between implementing passive versus active network interception, any attacker capable of one is almost certainly capable of both. All network attackers should be assumed to be "active".

Numerous applications[3] relying on OpenSSL failed to properly implement one or both of these steps, resulting in the application's TLS connections being vulnerable to man-in-the-middle attacks, thereby defeating the purpose of using TLS.

Application developers are not the only ones to blame. OpenSSL's APIs are sometimes poorly documented or confusing, and there is very little information available that describes how to properly do the validation. Additionally, as opposed to other TLS libraries, OpenSSL does not provide any utility function to perform the second step, validating the server's hostname[4]. Requiring application developers to implement hostname validation themselves has led to numerous critical security vulnerabilities.

Ideally, developers should instead rely on a well-tested, higher level library such as libCURL[5] that already performs certificate validation. Otherwise, if OpenSSL must be used directly, **section 2** and **section 3** of this paper provide detailed guidelines and sample code on how to perform this validation.

Rather than handling all the possible scenarios or corner cases that could arise when validating X.509 certificates, the goal is to provide simple and secure guidelines that should work for most applications. Known limitations are described in **section 4**.

The code snippets provided in this paper implement a simple HTTPS client that securely connects to `https://www.google.com`. The full working client as well as the code snippets provided are all available on iSEC Partners' GitHub at `https://github.com/iSECPartners/ssl-conservatory/`.

2 Validating the certificate chain

When connecting to a server, a TLS client first must validate the certificate chain returned by the server in order to ensure that the server's certificate is trusted.

2.1 Use OpenSSL's default validator

Validating the server's certificate chain with OpenSSL is relatively straight-forward. OpenSSL provides a default built-in validator that, given a list of trusted CA certificates, verifies the validity of a certificate chain by performing numerous checks including ensuring that:

- The server certificate chains up to a trusted CA certificate.
- Certificates within the chain have valid signatures.
- Certificates within the chain are not expired.

It is possible to override this default validation using the `SSL_CTX_set_cert_verify_callback()`[6] function. However, this should **never** be attempted, because validating a certificate chain is an extremely difficult task; OpenSSL's built-in validator should always be used.

The rest of this section will focus on how to properly configure and use the OpenSSL default validator.

[3] One notable example is Python: `http://bugs.python.org/issue1589`

[4] Helper functions to perform hostname validation were added to OpenSSL 1.1.0 but this version has not been released yet.

[5] `https://curl.haxx.se/libcurl/`

[6] `https://www.openssl.org/docs/ssl/SSL_CTX_set_cert_verify_callback.html`

2.2 Configuring OpenSSL's default certificate chain validator

2.2.1 Enabling certificate chain validation

First, peer certificate validation has to be enabled within the SSL_CTX[7] object:

```
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode, int (*verify_callback)(int, X509_STORE_CTX *));
```

- The `mode` should be set to `SSL_VERIFY_PEER` to enable certificate validation.
- An optional `verify_callback` can be provided, which will be called after OpenSSL's default validator has checked a certificate within the certificate chain. It can be used to get more information about a certificate error in case validation failed, or to do additional checks on the certificates. It should be noted that this is a different callback than the one described in section 2.1 on the preceding page.

2.2.2 Configuring the trust store to be used

A file or a folder containing a list of PEM-encoded trusted CA certificates must be provided to the SSL_CTX[8]:

```
int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile, const char *CApath);
```

The appropriate trust store to use is dependent upon the type of application:

- Most applications only have to connect to one or a few application servers. Therefore, the trust store should only contain the CA certificates needed to connect to those servers. Restricting the list of trusted CA certificate in such way is a security practice called certificate pinning.[9]
- Applications that need to be able to connect to any server on the Internet (such as browsers) could instead rely on Mozilla's list of root certificates used in Firefox. The list is not directly available in PEM format, but Adam Langley has written a tool[10] to extract and encode the certificates so that they can directly be used with OpenSSL. Using such a large trust store in an application is far from ideal, because it then requires the application developers to keep the trust store up to date. For example, in the event of a certificate authority being compromised,[11] application users will be exposed until the application's trust store is patched.

2.2.3 Sample Code

The following sample code enables certificate validation within our test client. The trust store we're going to use is a file called *VerisignClass3PublicPrimaryCertificationAuthority.pem* which contains the Verisign CA certificate used to sign www.google.com's server certificate.

This PEM file was obtained by browsing to `https://www.google.com` using Firefox, inspecting the certificate chain, and exporting the CA certificate.

[7] https://www.openssl.org/docs/ssl/SSL_CTX_set_verify.html

[8] https://www.openssl.org/docs/ssl/SSL_CTX_load_verify_locations.html

[9] http://blog.thoughtcrime.org/authenticity-is-broken-in-ssl-but-your-app-ha

[10] https://github.com/agl/extract-nss-root-certs

[11] https://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html

```
#define TRUSTED_CA_PATHNAME "VerisignClass3PublicPrimaryCertificationAuthority.pem"

ssl_ctx = SSL_CTX_new(TLSv1_client_method());

// Enable certificate validation
SSL_CTX_set_verify(ssl_ctx, SSL_VERIFY_PEER, NULL);

// Configure the trust store to be used
if (SSL_CTX_load_verify_locations(ssl_ctx, TRUSTED_CA_PATHNAME, NULL) != 1){
  fprintf(stderr, "Couldn't load certificate trust store.\n");
  return -1;
}
```

2.3 Checking the result of the certificate chain validation

2.3.1 Obtaining the result of the validation

After enabling certificate chain validation and providing a trust store within the SSL_CTX object, subsequent SSL objects generated using this SSL_CTX will automatically perform certificate chain validation. More specifically, the SSL handshake will fail if the certificate chain sent back by the server does not pass the validation.

If the handshake fails, the following function[12] can be used to recover the result of the certificate chain validation:

```
long SSL_get_verify_result(const SSL *ssl);
```

There are numerous possible return values all described in the OpenSSL documentation[13]. Notable values include:

- `X509_V_OK`: The verification succeeded or no peer certificate was presented. If the handshake failed, receiving this return value means that the handshake did not fail because of a certificate error. Other reasons include the server rejecting the handshake, network congestion, etc...
- `X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT`: Verification failed because the server's certificate is self-signed and isn't part of the trust store.
- `X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN`: Verification failed because the CA certificate up the certificate chain is not part of the trust store.

2.3.2 Recovering the server's certificate

If the handshake succeeded, the application must retrieve[14] the server certificate:

```
X509 *SSL_get_peer_certificate(const SSL *ssl);
```

This ensures that a certificate was actually sent back by the server. If the handshake succeeded but no certificate was received, it means that the SSL connection is using an insecure anonymous cipher suite for transport security. Such cipher suites are vulnerable to man-in-the-middle attacks and should never be used, as it provides no server authentication.

[12]http://www.openssl.org/docs/ssl/SSL_get_verify_result.html

[13]https://www.openssl.org/docs/apps/verify.html

[14]http://www.openssl.org/docs/ssl/SSL_get_peer_certificate.html

2.3.3 Sample Code

The following sample code performs the TLS handshake. If the handshake fails, it then checks whether the failure was due to a certificate error, and prints the error. If the handshake succeeds, it ensures that the server sent back a certificate.

```
// Do the SSL handshake
if(SSL_do_handshake(ssl) <= 0) {
  // SSL Handshake failed

  long verify_err = SSL_get_verify_result(ssl);
  if (verify_err != X509_V_OK) {
    // It failed because the certificate chain validation failed
    fprintf(stderr, "Certificate chain validation failed: %s\n", X509_verify_cert_error_string(verify_err));
    return -1;
  }

// Recover the server's certificate
server_cert =  SSL_get_peer_certificate(ssl);
if (server_cert == NULL) {
  // The handshake was successful although the server did not provide a certificate
  // Most likely using an insecure anonymous cipher suite... get out!
  return -1;
}
```

If everything passes, the client knows a valid certificate has been received. But who is the client talking to ?

3 Validating the server hostname

Once the certificate chain has been validated, the TLS client has to verify the identity of the server by ensuring that the server certificate was signed for the hostname to which it is trying to connect to.

The identity of the server can be specified in the *Subject Alternative Name* extension or in the *Common Name* within the *Subject* field of the certificate. According to RFC 6125,[15] if a certificate contains the *Subject Alternative Name* extension, the *Common Name* field should be ignored when validating the server's identity.

3.1 OpenSSL shortcomings

The biggest challenge with hostname validation is that prior to version 1.1.0[16], OpenSSL does not provide any function to perform this critical task. Therefore, it is up to application developers to properly extract relevant fields from the server's certificate and compare them with the hostname the TLS client wants to connect to, in order to validate the server's identity.

The remainder of this paper focuses on correctly implementing hostname validation.

[15] https://tools.ietf.org/html/rfc6125

[16] This version has not been released yet.

3.2 The *Subject Alternative Name* extension

3.2.1 Extracting hostnames from the *Subject Alternative Name* extension

If present in the certificate, the *Subject Alternative Name* extension can contain various fields including DNS names, IP addresses, email addresses or X.400 addresses. In the context of validating a TLS server's identity, DNS names must be extracted from the extension and compared with the server's expected hostname.

3.2.2 Sample Code

X.509 extensions are complex, and the OpenSSL functions to manipulate them are poorly documented. As there are numerous ways to extract names from the *Subject Alternative Name* extension, the following sample code tries to make it as concise and simple as possible. It extracts DNS names from the extension, and compares each of them with the expected hostname until either a match is found or all the names within the extension have been tested.

```
typedef enum {
  MatchFound,
  MatchNotFound,
  NoSANPresent,
  MalformedCertificate,
  Error
} HostnameValidationResult;

/**
* Tries to find a match for hostname in the certificate's Subject Alternative Name extension.
*
* Returns MatchFound if a match was found.
* Returns MatchNotFound if no matches were found.
* Returns MalformedCertificate if any of the hostnames had a NUL character embedded in it.
* Returns NoSANPresent if the SAN extension was not present in the certificate.
*/
static HostnameValidationResult matches_subject_alternative_name(const char *hostname, const X509 *server_cert) {
  HostnameValidationResult result = MatchNotFound;
  int i;
  int san_names_nb = -1;
  STACK_OF(GENERAL_NAME) *san_names = NULL;

  // Try to extract the names within the SAN extension from the certificate
  san_names = X509_get_ext_d2i((X509 *) server_cert, NID_subject_alt_name, NULL, NULL);
  if (san_names == NULL) {
    return NoSANPresent;
  }
  san_names_nb = sk_GENERAL_NAME_num(san_names);

  // Check each name within the extension
  for (i=0; i<san_names_nb; i++) {
    const GENERAL_NAME *current_name = sk_GENERAL_NAME_value(san_names, i);

    if (current_name->type == GEN_DNS) {
      // Current name is a DNS name, let's check it
      char *dns_name = (char *) ASN1_STRING_data(current_name->d.dNSName);

      // Make sure there isn't an embedded NUL character in the DNS name
      if (ASN1_STRING_length(current_name->d.dNSName) != strlen(dns_name)) {
        result = MalformedCertificate;
        break;
      }
      else { // Compare expected hostname with the DNS name
        if (strcasecmp(hostname, dns_name) == 0) {
          result = MatchFound;
          break;
        }
      }
    }
  }
  sk_GENERAL_NAME_pop_free(san_names, GENERAL_NAME_free);

  return result;
}
```

3.3 The *Common Name* field

3.3.1 Extracting the *Common Name* field

If the *Subject Alternative Name* extension is not part of the certificate, the *Common Name* within the *Subject* field must be checked against the expected hostname.

3.3.2 Sample code

The following sample code extracts the *Common Name* from the server's certificate and compares it with the expected hostname:

```
/**
* Tries to find a match for hostname in the certificate's Common Name field.
*
* Returns MatchFound if a match was found.
* Returns MatchNotFound if no matches were found.
* Returns MalformedCertificate if the Common Name had a NUL character embedded in it.
* Returns Error if the Common Name could not be extracted.
*/
static HostnameValidationResult matches_common_name(const char *hostname, const X509 *server_cert) {
  int common_name_loc = -1;
  X509_NAME_ENTRY *common_name_entry = NULL;
  ASN1_STRING *common_name_asn1 = NULL;
  char *common_name_str = NULL;

  // Find the position of the CN field in the Subject field of the certificate
  common_name_loc = X509_NAME_get_index_by_NID(X509_get_subject_name((X509 *) server_cert), NID_commonName, -1);
  if (common_name_loc < 0) {
    return Error;
  }

  // Extract the CN field
  common_name_entry = X509_NAME_get_entry(X509_get_subject_name((X509 *) server_cert), common_name_loc);
  if (common_name_entry == NULL) {
    return Error;
  }

  // Convert the CN field to a C string
  common_name_asn1 = X509_NAME_ENTRY_get_data(common_name_entry);
  if (common_name_asn1 == NULL) {
    return Error;
  }
  common_name_str = (char *) ASN1_STRING_data(common_name_asn1);

  // Make sure there isn't an embedded NUL character in the CN
  if (ASN1_STRING_length(common_name_asn1) != strlen(common_name_str)) {
    return MalformedCertificate;
  }
  // Compare expected hostname with the CN
  if (strcasecmp(hostname, common_name_str) == 0) {
    return MatchFound;
  }
  else {
    return MatchNotFound;
  }
}
```

3.4 Wrapping up hostname validation

The following sample code makes use of the previously defined functions to perform hostname validation, given an expected hostname and the server's certificate. If the validation is successful, the TLS client has authenticated the server and the TLS connection is secure; the client can start sending application data.

```
/**
* Validates the server's identity by looking for the expected hostname in the
* server's certificate. As described in RFC 6125, it first tries to find a match
* in the Subject Alternative Name extension. If the extension is not present in
* the certificate, it checks the Common Name instead.
*
* Returns MatchFound if a match was found.
* Returns MatchNotFound if no matches were found.
* Returns MalformedCertificate if any of the hostnames had a NUL character embedded in it.
* Returns Error if there was an error.
*/
HostnameValidationResult validate_hostname(const char *hostname, const X509 *server_cert) {
  HostnameValidationResult result;

  if((hostname == NULL) || (server_cert == NULL))
    return Error;

  // First try the Subject Alternative Names extension
  result = matches_subject_alternative_name(hostname, server_cert);
  if (result == NoSANPresent) {
    // Extension was not found: try the Common Name
    result = matches_common_name(hostname, server_cert);
  }

  return result;
}
```

4 Known limitations

The certificate validation guidelines described above are intended to work for most applications. However, they have some limitations, described in the following subsections.

4.1 Revocation

If a certificate is compromised, the CA that issued the certificate can invalidate it before it expires using two mechanisms: Certificate Revocation lists (CRL) and the Online Certificate Status Protocol (OCSP). During the TLS handshake, clients can query the CA using those mechanisms to ensure that the server certificate has not been revoked. The certificate validation guidelines described in this paper do not check for revocation.

Current revocation mechanisms suffer from various problems[17] that are outside the scope of this paper. In the name of usability, those issues have led browsers to do soft-fail checking, where the browser checks for revocation but does nothing if a problem occurs (for example if the OCSP responder cannot be reached).

Still, not performing revocation checking is far from ideal and definitely weakens the security of the TLS channel. This situation is also evolving, and one possible solution to fix revocation is a mechanism called "OCSP Stapling"[18], which was added to OpenSSL 0.9.8g. Guidelines on how to implement revocation checking with OpenSSL using OCSP stapling may be described in a future version of this paper.

4.2 Wildcard certificates

The hostname validation functions described in this paper only perform a string comparison between the expected hostname and names extracted from the certificate. RFC 2818[19] specifies that "names may contain the wildcard character * which is considered to match any single domain name component or component fragment".

Supporting wildcard certificates requires manually parsing the name to find the wildcard character, ensuring that it is in a valid location within the domain, and then trying to match the pattern with the server's expected hostname.

Performing wildcard certificate validation is not trivial, and it is possible to get an idea on how it should be done by looking at NSS'[20] or libCURL[21]'s implementations.

It should be noted that a domain administrator can sometimes avoid deploying a wildcard certificate by putting all the possible names in the *Subject Alternative Name* extension, or using multiple certificates along with *Server Name Indication*[22]

4.3 Other various limitations

Other scenarios where the validation code will not work include:

- Certificates issued for IP addresses.
- More complex certificates, for example:
  - Certificates that have multiple *Common Names*.[23]
  - Certificates with a "Name Constraints" extension[24].
- Servers performing client authentication.
- Exotic cipher suites such as the SRP or PSK cipher suites.

Readers are encouraged to contact the author if any major functionality is missing from this paper.

[17] http://www.imperialviolet.org/2011/03/18/revocation.html

[18] https://en.wikipedia.org/wiki/OCSP_stapling

[19] https://www.ietf.org/rfc/rfc2818.txt

[20] http://mxr.mozilla.org/mozilla/source/security/nss/lib/certdb/certdb.c

[21] https://github.com/bagder/curl/blob/master/lib/ssluse.c

[22] https://en.wikipedia.org/wiki/Server_Name_Indication

[23] As described in RFC 6125: https://tools.ietf.org/html/rfc6125

[24] The "Name Constraints" extension is not properly supported by OpenSSL prior to version 1.0.0.

5 Conclusion

Following the guidelines described in this paper will ensure that your TLS client is securely connecting to its intended server, thereby preventing attackers on the network from performing man-in-the-middle attacks.

On the server side, Ivan Ristic's "SSL/TLS Deployment Best Practices Guide"[25] provides excellent information on how to deploy and configure a TLS server.

The full code for our test client including utility functions to validate the server's hostname is available at: `https://github.com/iSECPartners/ssl-conservatory/`.

6 Acknowledgments

Thanks to David Thiel, Alex Garbutt, Chris Palmer, Paul Youn, Andy Grant, Gabe Pike and Chris Palmer for reviewing various versions of this paper. Their valuable feedback is very much appreciated. Thanks to B.J Orvis for providing an amazing SSL/TLS test suite with relevant test certificates.

[25] `https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices_1.0.pdf`