Parsing X.509 Certificates with OpenSSL and C

Zakir Durumeric (/) | October 13, 2013

While OpenSSL (https://www.openssl.org/) has become one of the defacto libraries for performing SSL and TLS operations, the library is surprisingly opaque and its documentation is, at times, abysmal. As part of our recent research, we have been performing Internet-wide scans of HTTPS hosts in order to better understand the HTTPS ecosystem (Analysis of the HTTPS Certificate Ecosystem (https://jhalderm.com/pub/papers/https-imc13.pdf), ZMap: Fast Internet-Wide Scanning and its Security Applications (https://zmap.io)). We use OpenSSL for many of these operations including parsing X.509 certificates. However, in order to parse and validate certificates, our team had to dig through parts of the OpenSSL code base and multiple sources of documention to find the correct functions to parse each piece of data. This post is intended to document many of these operations in a single location in order to hopefully alleviate this painful process for others.

If you have found other pieces of code particularly helpful, please don't hesitate to send them along (mailto:zakir@umich.edu) and we'll update the post. I want to note that if you're starting to develop against OpenSSL, O'Reilly's *Network Security with OpenSSL* (http://www.amazon.com /Network-Security-OpenSSL-John-Viega/dp/059600270X) is an incredibly helpful resource; the book contains many snippets and pieces of documentation that I was not able to find anywhere online. I also want to thank James Kasten (https://jdkasten.com/) who helped find and document several of these solutions.

Creating an OpenSSL X509 Object

All of the operations we discuss start with either a single X.509 certificate or a "stack" of certificates. OpenSSL represents a single certificate with an `X509` struct and a list of certificates, such as the certificate chain presented during a TLS handshake as a `STACK_OF(X509)`. Given that the parsing and validation stems from here, it only seems reasonable to start with how to create or access an X509 object. A few common scenarios are:

1. You have initiated an SSL or TLS connection using OpenSSL.

In this case, you have access to an OpenSSL `SSL` struct from which you can extract the presented certificate as well as the entire certificate chain that the server presented to the client. In our specific case, we use libevent to perform TLS connections and can access the SSL struct from the libevent bufferevent: `SSL *ssl = bufferevent_openssl_get_ssl(bev)`. This will clearly be different depending on how you complete your connection. However, once you have your SSL context, the server certificate and presented chain can be extracted as follows:

```
#include <openssl/x509.h>
#include <openssl/x509v3.h>

X509 *cert = SSL_get_peer_certificate(ssl);
STACK_OF(X509) *sk = SSL_get_peer_cert_chain(ssl);
```

We have found that at times, OpenSSL will produce an empty certificate chain (`SSL_get_peer_cert_chain` will come back `NULL`) even though a client certificate has been presented (the server certificate is generally presented as the first certificate in the stack along with the remaining chain). It's unclear to us why this happens, but it's not a deal breaker, as it's easy to create a new stack of certificates:

```
X509 *cert = SSL_get_peer_certificate(ssl);
STACK_OF(X509) *sk = sk_X509_new_null();
sk_X509_push(sk, cert);
```

2. You have stored a certificate on disk as a PEM file.

For reference, a PEM file is the Base64-encoded version of an X.509 certificate, which should look similar to the following:

```
-----BEGIN CERTIFICATE-----
MIIHIDCCBgigAwIBAgIIMrM8cLO76sYwDQYJKoZIhvcNAQEFBQAwSTELMAkGA1UE
BhMCVVMxEzARBgNVBAoTCkdvb2dsZSBJbmMxJTAjBgNVBAMTHEdvb2dsZSBJbnRl
iftrJvzAOMAPY5b/klZvqH6Ddubg/hUVPkiv4mr5MfWfglCQdFF1EBGNoZSFAU7y
ZkGENAvDmv+5xVCZELeiWA2PoNV4m/SW6NHrF7gz4MwQssqP9dGMbKPOF/D2nxic
TnD5WkGMCWpLgqDWWRoOrt6xf0BPWukQBDMHULlZgXzNtoGlEnwztLlnf0I/WWIS
eBSyDTeFJfopvoqXuws23X486fdKcCAV1n/Nl6y2z+uVvcyTRxY2/jegmV0n0kHf
gfcKzw==
-----END CERTIFICATE-----
```

In this case, you can access the certificate as follows:

```
#include <stdio.h>
#include <openssl/x509.h>
#include <openssl/x509v3.h>

FILE *fp = fopen(path, "r");
if (!fp) {
        fprintf(stderr, "unable to open: %s\n", path);
        return EXIT_FAILURE;
}

X509 *cert = PEM_read_X509(fp, NULL, NULL, NULL);
if (!cert) {
        fprintf(stderr, "unable to parse certificate in: %s\n", path);
        fclose(fp);
        return EXIT_FAILURE;
}

// any additional processing would go here..

X509_free(cert);
fclose(fp);
```

3. You have access to the raw certificate in memory.

In the case that you have access to the raw encoding of the certificate in memory, you can parse it as follows. This is useful if you have stored raw certificates in a database or similar data store.

```
#include <openssl/x509.h>
#include <openssl/x509v3.h>
#include <openssl/bio.h>

const unsigned char *data = ... ;
size_t len = ... ;

X509 *cert = d2i_X509(NULL, &data, len);
if (!cert) {
        fprintf(stderr, "unable to parse certificate in memory\n");
        return EXIT_FAILURE;
}

// any additional processing would go here..

X509_free(cert);
```

4. You have access to the Base64 encoded PEM in memory.

```
char* pemCertString = ..... (includes "-----BEGIN/END CERTIFICATE-----")
size_t certLen = strlen(pemCertString);

BIO* certBio = BIO_new(BIO_s_mem());
BIO_write(certBio, pemCertString, certLen);
X509* certX509 = PEM_read_bio_X509(certBio, NULL, NULL, NULL);
if (!certX509) {
    fprintf(stderr, "unable to parse certificate in memory\n");
    return EXIT_FAILURE;
}

// do stuff

BIO_free(certBio);
X509_free(certX509);
```

Parsing Certificates

Now that we have access to a certificate in OpenSSL, we'll focus on how to extract useful data from the certificate. We don't include the `#include` s in every statement, but use the following headers throughout our codebase:

```
#include <openssl/x509v3.h>
#include <openssl/bn.h>
#include <openssl/asn1.h>
#include <openssl/x509.h>
#include <openssl/x509_vfy.h>
#include <openssl/pem.h>
#include <openssl/bio.h>

OpenSSL_add_all_algorithms();
```

You will also need the development versions of the OpenSSL libraries and to compile with `-lssl` .

Subject and Issuer

The certificate subject and issuer can be easily extracted and represented as a single string as follows:

```
char *subj = X509_NAME_oneline(X509_get_subject_name(cert), NULL, 0);
char *issuer = X509_NAME_oneline(X509_get_issuer_name(cert), NULL, 0);
```

These can be freed by calling `OPENSSL_free`.

By default, the subject and issuer are returned in the following form:

```
/C=US/ST=California/L=Mountain View/O=Google Inc/CN=*.google.com
```

If you want to convert these into a more traditional looking DN, such as:

```
C=US, ST=Texas, L=Austin, O=Polycom Inc., OU=Video Division, CN=a.digitalnetbr.net
```

they can be converted with the following code:

```
int i, curr_spot = 0;
char *s = tmpBuf + 1; /* skip the first slash */
char *c = s;
while (1) {
        if (((*s == '/') && ((s[1] >= 'A') && (s[1] <= 'Z') &&
                        ((s[2] == '=') || ((s[2] >= 'A') && (s[2] <= 'Z')
                        && (s[3] == '='))))) || (*s == '\0')) {
                i = s - c;
                strncpy(destination + curr_spot, c, i);
                curr_spot += i;
                assert(curr_spot < size);
                c = s + 1; /* skip following slash */
                if (*s != '\0') {
                        strncpy(destination + curr_spot, ", ", 2);
                        curr_spot += 2;
                }
        }
        if (*s == '\0')
                break;
        ++s;
}
```

It is also possible to extract particular elements from the subject. For example, the following code will iterate over all the values in the subject:

```
X509_NAME *subj = X509_get_subject_name(cert);

for (int i = 0; i < X509_NAME_entry_count(subj); i++) {
        X509_NAME_ENTRY *e = X509_NAME_get_entry(subj, i);
        ASN1_STRING *d = X509_NAME_ENTRY_get_data(e);
        char *str = ASN1_STRING_data(d);
}
```

or

```
for (;;) {
    int lastpos = X509_NAME_get_index_by_NID(subj, NID_commonName, lastpos);
    if (lastpos == -1)
        break;
    X509_NAME_ENTRY *e = X509_NAME_get_entry(subj, lastpos);
    /* Do something with e */
}
```

Cryptographic (e.g. SHA-1) Fingerprint

We can calculate the SHA-1 fingerprint (or any other fingerprint) with the following code:

```
#define SHA1LEN 20
char buf[SHA1LEN];

const EVP_MD *digest = EVP_sha1();
unsigned len;

int rc = X509_digest(cert, digest, (unsigned char*) buf, &len);
if (rc == 0 || len != SHA1LEN) {
        return EXIT_FAILURE;
}
return EXIT_SUCCESS;
```

This will produce the raw fingerprint. This can be converted to the human readable hex version as follows:

```
void hex_encode(unsigned char* readbuf, void *writebuf, size_t len)
{
        for(size_t i=0; i < len; i++) {
                char *l = (char*) (2*i + ((intptr_t) writebuf));
                sprintf(l, "%02x", readbuf[i]);
        }
}

char strbuf[2*SHA1LEN+1];
hex_encode(buf, strbuf, SHA1LEN);
```

Version

Parsing the certificate version is straight-foward; the only oddity is that it is zero-indexed:

```
int version = ((int) X509_get_version(cert)) + 1;
```

Serial Number

Serial numbers can be arbitrarily large as well as positive or negative. As such, we handle it as a string instead of a typical integer in our processing.

```
#define SERIAL_NUM_LEN 1000;
char serial_number[SERIAL_NUM_LEN+1];

ASN1_INTEGER *serial = X509_get_serialNumber(cert);

BIGNUM *bn = ASN1_INTEGER_to_BN(serial, NULL);
if (!bn) {
        fprintf(stderr, "unable to convert ASN1INTEGER to BN\n");
        return EXIT_FAILURE;
}

char *tmp = BN_bn2dec(bn);
if (!tmp) {
        fprintf(stderr, "unable to convert BN to decimal string.\n");
        BN_free(bn);
        return EXIT_FAILURE;
}

if (strlen(tmp) >= len) {
        fprintf(stderr, "buffer length shorter than serial number\n");
        BN_free(bn);
        OPENSSL_free(tmp);
        return EXIT_FAILURE;
}

strncpy(buf, tmp, len);
BN_free(bn);
OPENSSL_free(tmp);
```

Signature Algorithm

The signature algorithm on a certificate is stored as an OpenSSSL NID:

```
int pkey_nid = OBJ_obj2nid(cert->cert_info->key->algor->algorithm);

if (pkey_nid == NID_undef) {
        fprintf(stderr, "unable to find specified signature algorithm name.\n");
        return EXIT_FAILURE;
}
```

This can be translated into a string representation (either short name or long description):

```
char sigalgo_name[SIG_ALGO_LEN+1];
const char* sslbuf = OBJ_nid2ln(pkey_nid);

if (strlen(sslbuf) > PUBKEY_ALGO_LEN) {
        fprintf(stderr, "public key algorithm name longer than allocated buffer.\n");
        return EXIT_FAILURE;
}

strncpy(buf, sslbuf, PUBKEY_ALGO_LEN);
```

This will result in a string such as `sha1WithRSAEncryption` or `md5WithRSAEncryption`.

Public Key

Parsing the public key on a certificate is type-specific. Here, we provide information on how to extract which type of key is included and to parse RSA and DSA keys:

```
char pubkey_algoname[PUBKEY_ALGO_LEN];

int pubkey_algonid = OBJ_obj2nid(cert->cert_info->key->algor->algorithm);

if (pubkey_algonid == NID_undef) {
        fprintf(stderr, "unable to find specified public key algorithm name.\n");
        return EXIT_FAILURE;
}

const char* sslbuf = OBJ_nid2ln(pubkey_algonid);
assert(strlen(sslbuf) < PUBKEY_ALGO_LEN);
strncpy(buf, sslbuf, PUBKEY_ALGO_LEN);

if (pubkey_algonid == NID_rsaEncryption || pubkey_algonid == NID_dsa) {

        EVP_PKEY *pkey = X509_get_pubkey(cert);
        IFNULL_FAIL(pkey, "unable to extract public key from certificate");

        RSA *rsa_key;
        DSA *dsa_key;
        char *rsa_e_dec, *rsa_n_hex, *dsa_p_hex, *dsa_p_hex,
                 *dsa_q_hex, *dsa_g_hex, *dsa_y_hex;

        switch(pubkey_algonid) {

                case NID_rsaEncryption:

                        rsa_key = pkey->pkey.rsa;
                        IFNULL_FAIL(rsa_key, "unable to extract RSA public key");

                        rsa_e_dec = BN_bn2dec(rsa_key->e);
                        IFNULL_FAIL(rsa_e_dec,  "unable to extract rsa exponent");

                        rsa_n_hex = BN_bn2hex(rsa_key->n);
                        IFNULL_FAIL(rsa_n_hex,  "unable to extract rsa modulus");

                        break;

                case NID_dsa:

                        dsa_key = pkey->pkey.dsa;
                        IFNULL_FAIL(dsa_key, "unable to extract DSA pkey");

                        dsa_p_hex = BN_bn2hex(dsa_key->p);
                        IFNULL_FAIL(dsa_p_hex, "unable to extract DSA p");

                        dsa_q_hex = BN_bn2hex(dsa_key->q);
                        IFNULL_FAIL(dsa_q_hex, "unable to extract DSA q");

                        dsa_g_hex = BN_bn2hex(dsa_key->g);
                        IFNULL_FAIL(dsa_g_hex, "unable to extract DSA g");

                        dsa_y_hex = BN_bn2hex(dsa_key->pub_key);
                        IFNULL_FAIL(dsa_y_hex, "unable to extract DSA y");

                        break;

                default:
                        break;
        }

        EVP_PKEY_free(pkey);
}
```

Validity Period

OpenSSL represents the not-valid-after (expiration) and not-valid-before as `ASN1_TIME` objects, which can be extracted as follows:

```
ASN1_TIME *not_before = X509_get_notBefore(cert);
ASN1_TIME *not_after = X509_get_notAfter(cert);
```

These can be converted into ISO-8601 timestamps using the following code:

```
#define DATE_LEN 128

int convert_ASN1TIME(ASN1_TIME *t, char* buf, size_t len)
{
        int rc;
        BIO *b = BIO_new(BIO_s_mem());
        rc = ASN1_TIME_print(b, t);
        if (rc <= 0) {
                log_error("fetchdaemon", "ASN1_TIME_print failed or wrote no data.\n");
                BIO_free(b);
                return EXIT_FAILURE;
        }
        rc = BIO_gets(b, buf, len);
        if (rc <= 0) {
                log_error("fetchdaemon", "BIO_gets call failed to transfer contents to bu
f");
                BIO_free(b);
                return EXIT_FAILURE;
        }
        BIO_free(b);
        return EXIT_SUCCESS;
}

char not_after_str[DATE_LEN];
convert_ASN1TIME(not_after, not_after_str, DATE_LEN);

char not_before_str[DATE_LEN];
convert_ASN1TIME(not_before, not_before_str, DATE_LEN);
```

CA Status

Checking whether a certificate is a valid CA certificate is not a boolean operation as you might expect. There are several avenues through which a certificate can be interpreted as CA certificate. As such, instead of directly checking various X.509 extensions, it is more reliable to use `X509_check_ca`. Any value >= 1 is considered a CA certificate whereas 0 is not a CA certificate.

```
int raw = X509_check_ca(cert);
```

Other X.509 Extensions

Certificates can contain any other arbitrary extensions. The following code will loop through all of the extensions on a certificate and print them out:

```
STACK_OF(X509_EXTENSION) *exts = cert->cert_info->extensions;

int num_of_exts;
if (exts) {
        num_of_exts = sk_X509_EXTENSION_num(exts);
} else {
        num_of_exts = 0
}

IFNEG_FAIL(num_of_exts, "error parsing number of X509v3 extensions.");

for (int i=0; i < num_of_exts; i++) {

        X509_EXTENSION *ex = sk_X509_EXTENSION_value(exts, i);
        IFNULL_FAIL(ex, "unable to extract extension from stack");
        ASN1_OBJECT *obj = X509_EXTENSION_get_object(ex);
        IFNULL_FAIL(obj, "unable to extract ASN1 object from extension");

        BIO *ext_bio = BIO_new(BIO_s_mem());
        IFNULL_FAIL(ext_bio, "unable to allocate memory for extension value BIO");
        if (!X509V3_EXT_print(ext_bio, ex, 0, 0)) {
                M_ASN1_OCTET_STRING_print(ext_bio, ex->value);
        }

        BUF_MEM *bptr;
        BIO_get_mem_ptr(ext_bio, &bptr);
        BIO_set_close(ext_bio, BIO_NOCLOSE);

        // remove newlines
        int lastchar = bptr->length;
        if (lastchar > 1 && (bptr->data[lastchar-1] == '\n' || bptr->data[lastchar-1] ==
'\r')) {
                bptr->data[lastchar-1] = (char) 0;
        }
        if (lastchar > 0 && (bptr->data[lastchar] == '\n' || bptr->data[lastchar] == '\r
')) {
                bptr->data[lastchar] = (char) 0;
        }

        BIO_free(ext_bio);

        unsigned nid = OBJ_obj2nid(obj);
        if (nid == NID_undef) {
                // no lookup found for the provided OID so nid came back as undefined.
                char extname[EXTNAME_LEN];
                OBJ_obj2txt(extname, EXTNAME_LEN, (const ASN1_OBJECT *) obj, 1);
                printf("extension name is %s\n", extname);
        } else {
                // the OID translated to a NID which implies that the OID has a known sn/
ln
                const char *c_ext_name = OBJ_nid2ln(nid);
                IFNULL_FAIL(c_ext_name, "invalid X509v3 extension name");
                printf("extension name is %s\n", c_ext_name);
        }

        printf("extension length is %u\n", bptr->length)
        printf("extension value is %s\n", bptr->data)
}
```

Misordered Certificate Chains

At times, we'll receive misordered certificate chains. The following code will attempt to reorder certificates to construct a rational certificate chain based on each certificate's subject and issuer string. The algorithm is O(n^2), but we generally only receive two or three certificates and in the majority-case, they will already be in the correct order.

```
        STACK_OF(X509) *r_sk = sk_X509_new_null();
        sk_X509_push(r_sk, sk_X509_value(st, 0));

        for (int i=1; i < sk_X509_num(st); i++) {
                X509 *prev = sk_X509_value(r_sk, i-1);
                X509 *next = NULL;
                for (int j=1; j < sk_X509_num(st); j++) {
                        X509 *cand = sk_X509_value(st, j);
                        if (!X509_NAME_cmp(cand->cert_info->subject, prev->cert_info->iss
uer)
                                        || j == sk_X509_num(st) - 1) {
                                next = cand;
                                break;
                        }
                }
                if (next) {
                        sk_X509_push(r_sk, next);
                } else {
                        // we're unable to figure out the correct stack so just use the o
riginal one provided.
                        sk_X509_free(r_sk);
                        r_sk = sk_X509_dup(st);
                        break;
                }
        }
```

Validating Certificates

In our scans, we oftentimes use multiple CA stores in order to emulate different browsers. Here, we describe how we create specialized stores and validate against them.

We can create a store based on a particular file with the following:

```
X509_STORE *s = X509_STORE_new();
if (s == NULL) {
        fprintf(stderr, "unable to create new X509 store.\n");
        return NULL;
}
int rc = X509_STORE_load_locations(s, store_path, NULL);
if (rc != 1) {
        fprintf(stderr, "unable to load certificates at %s to store\n", store_path);
        X509_STORE_free(s);
        return NULL;
}
return s;
```

And then validate certificates against the store with the following:

```
X509_STORE_CTX *ctx = X509_STORE_CTX_new();
if (!ctx) {
        fprintf(stderr, "unable to create STORE CTX\n");
        return -1;
}
if (X509_STORE_CTX_init(ctx, store, cert, st) != 1) {
        fprintf(stderr, "unable to initialize STORE CTX.\n");
        X509_STORE_CTX_free(ctx);
        return -1;
}
int rc = X509_verify_cert(ctx);
X509_STORE_CTX_free(ctx);
return rc;
```

It's worth noting that self-signed certificates will always fail OpenSSL's validation. While this might make sense in most client applications, we are oftentimes interested in other errors that might be present. We validate self-signed certificates by adding them into a temporary store and then validating against it. It's a bick hackish, but is much easier than re-implementing OpenSSL's validation techniques.

```
X509_STORE *s = X509_STORE_new();
int num = sk_X509_num(sk);
X509 *top = sk_X509_value(st, num-1);
X509_STORE_add_cert(s, top);
X509_STORE_CTX *ctx = X509_STORE_CTX_new();
X509_STORE_CTX_init(ctx, s, cert, st);
int rc = X509_verify_cert(ctx);
if (rc == 1) {
        // validated OK. either trusted or self signed.
} else {
        // validation failed
        int err = X509_STORE_CTX_get_error(ctx);
}

// any additional processing..

X509_STORE_CTX_free(ctx);
X509_STORE_free(s);
```

Sometimes you will also find that you just need to check whether a certificate has been issued by a trusted source instead of just considering whether it is currently valid, which can be done using `X509_check_issued`. For example, if you wanted to check whether a certificate was self-signed:

```
if (X509_check_issued(cert, cert) == X509_V_OK) {
        is_self_signed = 1;
} else {
        is_self_signed = 0;
}
```

Helper Functions

There are several other functions that were used in troubleshooting and might be of help while you're developing code against OpenSSL.

Print out the basic information about a certificate:

```
#define MAX_LENGTH 1024

void print_certificate(X509* cert) {
        char subj[MAX_LENGTH+1];
        char issuer[MAX_LENGTH+1];
        X509_NAME_oneline(X509_get_subject_name(cert), subj, MAX_LENGTH);
        X509_NAME_oneline(X509_get_issuer_name(cert), issuer, MAX_LENGTH);
        printf("certificate: %s\n", subj);
        printf("\tissuer: %s\n\n", issuer);
}
```

Print out each certificate in a given stack:

```
void print_stack(STACK_OF(X509)* sk)
{
        unsigned len = sk_X509_num(sk);
        printf("Begin Certificate Stack:\n");
        for(unsigned i=0; i<len; i++) {
                X509 *cert = sk_X509_value(sk, i);
                print_certificate(cert);
        }
        printf("End Certificate Stack\n");
}
```

Check whether two certificate stacks are identical:

```
int certparse_sk_X509_cmp(STACK_OF(X509) *a, STACK_OF(X509) *b)
{
        int a_len = sk_X509_num(a);
        int b_len = sk_X509_num(b);
        if (a_len != b_len) {
                return 1;
        }
        for (int i=0; i < a_len; i++) {
                if (X509_cmp(sk_X509_value(a, i), sk_X509_value(b, i))) {
                        return 1;
                }
        }
        return 0;
}
```

Check whether the subject and issuer string on a certificate are identical:

```
int certparse_subjeqissuer(X509 *cert)
{
        char *s = X509_NAME_oneline(X509_get_subject_name(cert), NULL, 0);
        char *i = X509_NAME_oneline(X509_get_issuer_name(cert), NULL, 0);
        int rc = strcmp(s, i);
        OPENSSL_free(s);
        OPENSSL_free(i);
        return (!rc);
}
```

Convert an OpenSSL error constant into a human readable string:

```
const char* get_validation_errstr(long e) {
        switch ((int) e) {
                case X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT:
                        return "ERR_UNABLE_TO_GET_ISSUER_CERT";
                case X509_V_ERR_UNABLE_TO_GET_CRL:
                        return "ERR_UNABLE_TO_GET_CRL";
                case X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE:
                        return "ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE";
                case X509_V_ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE:
                        return "ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE";
                case X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY:
                        return "ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY";
                case X509_V_ERR_CERT_SIGNATURE_FAILURE:
                        return "ERR_CERT_SIGNATURE_FAILURE";
                case X509_V_ERR_CRL_SIGNATURE_FAILURE:
                        return "ERR_CRL_SIGNATURE_FAILURE";
                case X509_V_ERR_CERT_NOT_YET_VALID:
                        return "ERR_CERT_NOT_YET_VALID";
                case X509_V_ERR_CERT_HAS_EXPIRED:
                        return "ERR_CERT_HAS_EXPIRED";
                case X509_V_ERR_CRL_NOT_YET_VALID:
                        return "ERR_CRL_NOT_YET_VALID";
                case X509_V_ERR_CRL_HAS_EXPIRED:
                        return "ERR_CRL_HAS_EXPIRED";
                case X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD:
                        return "ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD";
                case X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD:
                        return "ERR_ERROR_IN_CERT_NOT_AFTER_FIELD";
                case X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD:
                        return "ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD";
                case X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD:
                        return "ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD";
                case X509_V_ERR_OUT_OF_MEM:
                        return "ERR_OUT_OF_MEM";
                case X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT:
                        return "ERR_DEPTH_ZERO_SELF_SIGNED_CERT";
                case X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN:
                        return "ERR_SELF_SIGNED_CERT_IN_CHAIN";
                case X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY:
                        return "ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY";
                case X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE:
                        return "ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE";
                case X509_V_ERR_CERT_CHAIN_TOO_LONG:
                        return "ERR_CERT_CHAIN_TOO_LONG";
                case X509_V_ERR_CERT_REVOKED:
                        return "ERR_CERT_REVOKED";
                case X509_V_ERR_INVALID_CA:
                        return "ERR_INVALID_CA";
                case X509_V_ERR_PATH_LENGTH_EXCEEDED:
                        return "ERR_PATH_LENGTH_EXCEEDED";
                case X509_V_ERR_INVALID_PURPOSE:
                        return "ERR_INVALID_PURPOSE";
                case X509_V_ERR_CERT_UNTRUSTED:
                        return "ERR_CERT_UNTRUSTED";
                case X509_V_ERR_CERT_REJECTED:
                        return "ERR_CERT_REJECTED";
                case X509_V_ERR_SUBJECT_ISSUER_MISMATCH:
                        return "ERR_SUBJECT_ISSUER_MISMATCH";
                case X509_V_ERR_AKID_SKID_MISMATCH:
                        return "ERR_AKID_SKID_MISMATCH";
                case X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH:
                        return "ERR_AKID_ISSUER_SERIAL_MISMATCH";
                case X509_V_ERR_KEYUSAGE_NO_CERTSIGN:
                        return "ERR_KEYUSAGE_NO_CERTSIGN";
                case X509_V_ERR_INVALID_EXTENSION:
                        return "ERR_INVALID_EXTENSION";
                case X509_V_ERR_INVALID_POLICY_EXTENSION:
                        return "ERR_INVALID_POLICY_EXTENSION";
                case X509_V_ERR_NO_EXPLICIT_POLICY:
                        return "ERR_NO_EXPLICIT_POLICY";
                case X509_V_ERR_APPLICATION_VERIFICATION:
                        return "ERR_APPLICATION_VERIFICATION";
                default:
                        return "ERR_UNKNOWN";
        }
}
```

I hope this helps. As I stated earlier, if you find other pieces of information useful, let me know and we'll get things updated. Similarly, if you find that any of the examples don't work, let me know.

Thanks to Jordan Whitehead for various corrections.