A PKI case study: Implementing the Server-based Certificate Validation Protocol

MARIUS MARIAN
University of Craiova
Department of Automation
ROMANIA
marius.marian@cs.ucv.ro

EUGEN SENDROIU
University of Birmingham
Department of Computer Science
UNITED KINGDOM
eugen_sen@yahoo.com

Abstract: Validation of a public-key certificate is a complex and resource-consuming task. If public-key technology is to be widely deployed in a variety of applications and environments, the amount of processing an application needs to perform before it can accept a digital certificate needs to be reduced. There is a large number of applications that can make use of public-key certificates, but to be trustworthy these applications must handle the overhead of constructing and validating the certification paths. The Server-based Certificate Validation Protocol (SCVP) is an Internet protocol for determining the path between a digital certificate - X.509 or Attribute Certificate - and a trusted root (Delegated Path Discovery) and the validation of that path (Delegated Path Validation) according to a particular validation policy. This paper presents an implementation of the SCVP protocol for the OpenSSL cryptographic library.

Key–Words: PKI, certification path discovery and validation, SCVP

1 Introduction

Public-key cryptography, also known as asymmetric cryptography, is a form of cryptography in which the key used to encrypt a message differs from the key used to decrypt it. In public-key cryptography, a user has a pair of cryptographic keys: a public key and a private key. The keys are related mathematically, but the private key cannot be practically derived from the public key. The problem of securely exchanging cryptographic keys between entities that want to protect their communication is thus eliminated.

A central problem for public-key cryptography is proving that a public key is authentic, and has not been tampered with or replaced by a malicious third party. The usual approach to this problem is to use a public-key infrastructure (PKI) [1], in which one or more third parties, known as certification authorities (CA), certify ownership of key pairs.

A PKI is an arrangement that binds public keys with respective user identities by means of a CA. The binding is established through the registration and issuance process, which, depending on the level of assurance the binding has, may be carried out by software at a CA, or under human supervision. The PKI role that assures this binding is called the registration authority (RA). For each user, the user identity, the public key, their binding, validity conditions and other attributes are made unforgeable in public key certificates issued by the CA. A digital certificate or public key certificate is an electronic document which incorporates a digital signature to bind together a public key with an identity information such as the name of a person or an organization, their address, and so forth. The certificate can be used to verify that a public key belongs to an individual. In a typical PKI scheme, the signature will belong to a CA.

SCVP [2] allows a client to delegate certification path construction and certification path validation to a server, allows simplification of client implementations and use of a set of predefined validation policies which are useful in the context of privately-owned public-key infrastructures for management purposes.

SCVP reduces this overhead for two classes of certificate-using applications. The first class of applications wants just two things: confirmation that the public key belongs to the identity named in the certificate and confirmation that the public key can be used for the intended purpose. Such clients can completely delegate certification path construction and validation to the SCVP server. This is often referred to as delegated path validation (DPV) [3]. The second class of applications can perform certification path validation, but they lack a reliable or efficient method of constructing a valid certification path. Such clients delegate certification path construction to the SCVP server, but not validation of the returned certification path. This is often referred to as delegated path discovery (DPD) [3].
1.1 Server-based Certificate Validation Protocol

Suppose Alice and Bob need to share a message, but Alice may not be familiar with Bob’s certificate authority. This scenario is common when Alice and Bob have different employers and their certificates were issued by their employer’s CA. In this case, Bob’s certificate may also include his CA’s public key signed by a higher-level CA2, which might be recognized by Alice. This process leads to a hierarchy of certificates, and to even more complex trust relationships. PKI mostly refers to the software that manages certificates in a large-scale setting. In X.509 PKI systems, the hierarchy of certificates is always a top-down tree, with a root certificate at the top, representing a CA that is so central to the scheme that it does not need to be authenticated by some trusted third party.

The primary goal of SCVP is to make it easier to deploy PKI-enabled applications by delegating path discovery and/or validation processing to a server, and to allow central administration of validation policies within an organization. SCVP can be used by clients that do much of the certificate processing themselves but simply want an untrusted server to collect information for them. However, when the client has complete trust in the SCVP server, SCVP can be used to delegate the work of certification path construction and validation, and SCVP can be used to ensure that policies are consistently enforced throughout an organization.

Untrusted SCVP servers can provide clients the certification paths. They can also provide clients the revocation information, such as Certificate Revocation Lists (CRLs) and Online Certificate Status Protocol (OCSP) responses that the clients need to validate the certification paths constructed by the SCVP server. These services can be valuable to clients that do not implement the protocols needed to find and download intermediate certificates, CRLs, and OCSP responses.

Trusted SCVP servers can perform certification path construction and validation for the client. For a client using these services, the client inherently trusts the SCVP server as much as it would its own certification path validation software (if it contained such software). There are two main reasons that a client may want to trust such an SCVP server:

1. The client does not want to incur the overhead of including certification path validation software and running it for each certificate it receives
2. The client is in an organization or community that wants to centralize management of validation policies. These policies might dictate that particular trust anchors are to be used and the types of policy checking that are to be performed during certification path validation

1.2 The OpenSSL cryptographic library

The OpenSSL Project [4] is a collaborative effort to develop a robust, commercial-grade, full-featured, and open-source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols [5] as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation. The cryptography library provides the most popular algorithms for symmetric key and public-key cryptography algorithms, and message digests [6]. It also provides a pseudorandom number generator, and support for manipulating common certificate formats and managing key material. There are also general-purpose helper libraries for buffer manipulation and manipulation of arbitrary precision numbers. Additionally, OpenSSL supports most common cryptographic acceleration hardware.

The paper presents the work performed to extend the OpenSSL library in order to support SCVP.

2 Project Description and Implementation

The project is clearly divided into two parts: the OpenSSL library extension supporting SCVP and a simple implementation for demonstration of the usefulness of the added feature. Because the library addition follows the architecture of the whole OpenSSL library [4, 6], this section will focus on the application.

2.1 The client-server paradigm

The high level design is that of a typical distributed system, having two main components: the client and the server that communicate over a computer network. The client software initiates the communication session, while the server waits for requests from clients and serves the requests by communicating back responses with useful information. This model (see figure 1) was chosen because the main idea of the implementation was to strip the client from the processing overhead. The client is a simple console application which is started from the command line with some mandatory parameters. It then takes the parameters and stores them in a SCVP request structures. After the SCVP request structure was built, it will be encoded and sent over the network to the specified
server. After the request is sent, the client waits for the server to respond. When a response is received, the client makes all the necessary checking to ensure that the response is valid and that it is coming from the desired server. If all goes well, the client will output the information that was requested from the server. As mentioned earlier, this client is for illustration purposes. A typical client would have obtained the certificate information from a party with which it wants to communicate, and upon receiving a response it will decide if the party is trustworthy or not.

The server is a partial implementation of a real server, with only a subset of the features specified by SCVP [2]. Note that although the server is not complete it contains all the obligatory features specified by the document, and some optional features as well. The server is split in to distinguished parts. First there is the main thread, which initializes the server with everything it needs to properly operate. It then remains in a continuous loop until it is stopped by the user and waits for client connections. Upon receiving a connection it will create a worker thread, which will handle all other operations verify the request, perform the actions specified by the request and then send back a response. A point is worth attention here. For sending back responses, the initial design consisted of a separate thread, which would pop up requests from a queue and then send does requests back to the server. While this separation would have made the code clearer and more flexible, it would have introduced unpredictable long response times. This drawback is simply unacceptable, since one of the uses of a SCVP server is to reduce the time needed to validate a certification path. It is true that even with the chosen architecture, the server still has the unpredictability of the thread schedule, but this could be wiped out by supporting this software on a real time operating system.

2.2 ASN.1 Structures

The internal structures for storing the data follow closely the ASN.1 [7] modules specification of the SCVP protocol. Someone might argue that it might be easier to create more abstract structures, but this design was chosen for two reasons. First, and most important, this is the structure that all protocols implementation follow inside the OpenSSL library. Second, because this is an implementation of a communication protocol, much of the code handles encoding and decoding messages as well as sending and receiving them over a network. Hence, is more convenient to keep the structure of the data closer to the protocol description. The specification of the protocol is huge [2], and the declaration of the structures is even more substantial. We will present here one of the important structures, the validation request. For convenience, the specification of this object in ASN.1 is introduced. This also demonstrates how closely the specification and the implementation are.

The fields of the Validation Request ASN.1 structure is given below:

CVRequest ::= SEQUENCE {
    cvRequestVersion INTEGER DEFAULT 1,
    query Query,
    requestorRef [0] GeneralNames OPTIONAL,
    requestNonce [1] OCTET STRING OPTIONAL,
    requestorName [2] GeneralName OPTIONAL,
    responderName [3] GeneralName OPTIONAL,
    requestExtensions[4] Extensions OPTIONAL,
    signatureAlg [5] AlgorithmIdentifier OPTIONAL,
    hashAlg[6] OBJECT IDENTIFIER OPTIONAL,
    requestorText [7] UTF8String (SIZE (1..256)) OPTIONAL
}

The actual C implementation follows:

typedef struct scvp_request_st
{
    ASN1_INTEGER *cvRequestVersion;
    SCVP_QUERY *query;
    GENERAL_NAME *requestorRef;
    SCVP_QUERY *requestorName;
    SCVP_QUERY *responderName;
    X509_EXTENSION *requestExtensions;
    AlgorithmIdentifier *signatureAlg;
    OBJECT IDENTIFIER *hashAlg;
    UTF8String *requestorText;
} SCVP_REQ;

The cvRequestVersion item defines the version of the SCVP CVRequest used in a request. The query item specifies one or more certificates that are the subject of the request; the certificates can be either public-key certificates [1] or attribute certificates [8].
The optional requestorRef item contains a list of names identifying SCVP servers, and it is intended for use in environments where SCVP relay is employed. Although requestorRef is encoded as a SEQUENCE, no order is implied. The requestorRef item is used to detect looping in some configurations. The optional requestNonce item contains a request identifier generated by the SCVP client. If the client includes a requestNonce value in the request, it is expressing a preference that the SCVP server should return a non-cached response. The signatureAlg item contains an algorithm identifier specifying which algorithm the server should use to sign the response message. The hashAlg item contains an object identifier indicating which hash algorithm the server should use to compute the hash value for the requestHash item in the response.

2.3 System Architecture

The client software can be understood by following 3 basic layers: input, network, output. The input layer parses the command line arguments and verifies that it has enough information to proceed. The necessary information is:

- The certificate(s) subject to validation.
- The checks that it wants the server to perform.
- Indication of the information that the server should return.

If the client does not receive these minimum information it will output the usage syntax of the program. Note that there are over a dozen of parameters and, in most of the environments many of them should be required as input such as trust anchor, x509 v3 extensions and others. The validation policy reference can specify a validation policy which the server should use when is creating and validation the certification path. For the purpose of this implementation we specify only the default validation policy, but this can be extended to support other validation policies as well. The validation algorithm parameter specifies the algorithm that the server should use when constructing and validating a path. This parameter has been provided for future releases of the client software, but for the time being this parameter is not used since the only algorithm supported is the Basic Algorithm defined in Section 6 of [1]. A userPolicySet containing the anyPolicy OID indicates a user-initial-policy-set of any-policy. The policyFlags value specifies controls the way the policy should be handled. The inhibitPolicyMapping bit specifies an input to the certification path validation algorithm, and it controls whether policy mapping is allowed during certification path validation (see [1], Section 6.1.1). If the client wants the server to inhibit policy mapping, inhibitPolicyMapping is set to TRUE in the request. The requireExplicitPolicy bit specifies an input to the certification path validation algorithm, and it controls whether there must be at least one valid policy in the certificate policies extension (see [1], Section 6.1.1). The inhibitAnyPolicy bit specifies an input to the certification path validation algorithm, and it controls whether the anyPolicy OID is processed or ignored when evaluating certificate policy. If the client wants the server to ignore the anyPolicy OID, inhibitAnyPolicy MUST be set to TRUE in the request. The trustAnchors item specifies the trust anchors at which the certification path must terminate if the path is to be considered valid by the SCVP server for the request. If this item is specified, the server will try to build a certification path to one of these anchors and not to any other. After the information from the command line has been parsed a request is created using this information and send to the network layer for further processing. The network layer takes the request and encodes it into ASN1 DER format. It then creates a HTTP POST request using 1.1 version of HTTP protocol, sets the mime type Content-type header field to application/scvp-cv-request and puts the size of the DER encoded request in to the Content-Length header. It then appends the SCVP request and sends the buffer on the network, along with the size of the entire request. After the request has been send, the layer waits for a response from the server. Upon receiving a response it first checks the HTTP headers and extracts the encoded response. It decodes the response and check the response for validity. The response is further encoded with the servers private key, so it takes the servers certificate which, for the purpose of this project is supposed to be valid and uses the public key information to decode the response. Finally it sends the request to the output layer.

The output layer will handle the information in the response. The first step is to check that the response is valid, meaning that it has all the required fields. If these fields are present, it proceeds with checking the response status and validation time, information which is provided to the user. If the response is SUCCESS it will handle the other fields, otherwise the application will exit. At this step, if ev-
erything is successful, it will print out all the other information specified in the request. The modules of the

![Diagram of SCVP responder](image.png)

**Figure 2: The SCVP responder**

SCVP server implementation are depicted in figure 2. The input module allows the user to configure several parameters of the server. The user can input the path to the store with certificates from where the server should lookup certificates. A point is worth making here. Usually, when a program is offering services as SCVP does, it also specifies which trust anchors does it supports. For example, OCSP lists the trust anchors which it supports (such as VeriSign, ValiCert, Equinox, etc.). The OCSP has access to the online revocation repositories that it claims to serve. A real SCVP server would have similar behavior, serving some certificate authorities, and having access to their certificate and revocation repositories, but because the URLs of the repositories are not public and are protected with passwords, this server supposes that the certificates are already available. This could be a real challenge though, because, for performance consideration, a real SCVP server should, when its idle, gather this information from repository and store it locally. Other parameters that can be configured are the database file and the trust anchor file. This was implemented for portability since, currently there is no platform-independent connectivity system.

The processing core, as its name implies, is the main engine of the application. It defines the flow and the logic of the program. It controls every other module, passing it information and gathering useful data from it. This module is started by the internetworking module, when the latter one receives a request. For each request a new processing core is created which handles it. After it has received the request the core will ask the ASN.1 module to verify the syntax of the request, and this is correct extract the request, decode it at return it back to the core. If the ASN.1 module will return an error, the error code is extract, sent to the responder module and the core will finish its work. If the request was valid, the core will then interact with the database module. It will send it the information for each certificate (issuer name and serial number), and the module will return a stack of certificates. Next, the core will pass the entire information to the validation module, which will return back an error code, and if the error code is 0, it will also return the required information (certification path, public key information etc.) Finally, the server will create the response and send it to the responder module, which will encode it, sign it using the servers private key, encapsulate it in an HTTP response and send it back to the client.

This module is the first one that gets control after the bootstrap code has finish its execution. It will first initialize its parameters for internetworking operation the TCP/IP stack, the listening socket, the port and IP address. After that it sits on a loop until the user stops it from the command line. It waits for requests and, for each request received, it will construct a worker thread. The worker thread will than start the processing core with the input parameters.

This module will be started by the processing core and it will receive as input a request buffer. It will first check the HTTP syntax for correctness and if it passes this test it will attempt to decode the SCVP request from the content of the HTTP request. The resulting structure will be the one returned back to the processing core. The module will perform additional checks to ensure that the request has all the required information, that the request version is supported by the server and that the checks and want backs required by the client can be satisfied by this server. There are also some custom checks which the module performs but which are not constrains of the RFC specification. It will check for a compatibility between the checks and want backs required by the client. For instance, if a request contains only path validation check, and it requires back information about revocation, the server will return an error response, since the revocation is not specify as a check in the request. After the module has returned the information to the core, it will stop its execution, but will be employed several times later. Most notably, this module will be necessary when the want backs are populated. This is because a want back must be send as an encoded ASN1 octet string. The ASN.1 module has to detect which type of structure it needs to encode and then apply the corresponding method for DER encoding of the structure.

The database module is entitled with the job of gathering PKI information from the database. This information includes certificates, revocation lists, OCSP responders locator information, CRL repository infor-
mation and cached responses which were previously generated. Because this application was supposed to be cross platform no single database connectivity driver could be used. To accommodate this inconvenience separate modules would have been necessary for different platforms. However, for this demonstration purpose we choose to represent the database as a collection of files and a directory. The directory contains all the certificate files, stored in BASE64 PEM encoding, the CRL files stored similarly and the cached responses files which are stored as clear text. In addition to these files there are some other global files which replace the database functionality. A file is needed to identify each certificate in the directory. Each entry in the file consists of the certificates issuer name, its associated serial number and the location of the file on disk. This information is needed to be able to retrieve a certificate when one is indicated by references and not by its contents. The second file contains information about the locations of trust anchor certificates. This is an optional file and the server runs successfully without it, but this will require all clients to specify at least one trust anchor in the response. As already said, the database module is employed whenever a certificate is question is referenced by some information and is not present as an entity.

This module performs operation on three distinct entities: certificate, CRL, cached response. For each of these entities the module can perform retrieval or store operations. For certificate operations, the module expects as input either a stack of references for retrieval purpose, or a stack of certificates for store purpose. When it receives references the module will access the certificates file and parse it line by line. It first compares the retrieved issuer name with the issuer name if each reference. Upon a successful match it will compare the serial number from the one in the references. If this matches too, it will read the certificate from the location specified in the file and it will create a certificate structure from the contents of the certificate file. However the module doesn’t assume yet that the certificate was found. For this certificate to be considered valid it needs to pass the hash test. The reference has a hash field which specifies the algorithm used for calculating the hash value and an octet string representing the actual value. The database module performs the hash operation on the retrieved certificate and compares it with the hash in the reference. Is this matches the certificate was found and it is placed in a stack which will be the output of the modules computation. If the module receives as parameter a stack of certificates his job is to store these certificates on disk and add additional entries to the certificates file. In order to do this, the module performs the following steps for each certificate in the stack:

- Compute the hash value of the Subject name of the certificate. This is needed because the file will be named with this value.
- Create a file with the computed name and write the certificate content to it. The file will be created with base64 encoding and it will be of PEM type.
- Add an entry to the certificates file comprised of the certificates issuer name, the serial number and the path to the created file.

The validation module is the most important module of the server. It performs the functionality that this server is supposed to provide. The validation module receives as input the stack of certificates that are subject to the request, a sequence of checks that it should perform, the output which it must generate this are the want backs that the client has specified through the request and a stack of certificates representing trust anchors. Note that the validation module is based heavily on the existing library code to perform its actions. The functionality to validate a single certificate is already implemented and its a major part of the library. This module only handles the validation of the chain without being worried about actions performed on a single certificate, such as period validation, signature validation and policy constrains validation. After being started with the stack of certificates, the validation module starts to pop certificates one at a time from the stack. For each certificate in the stack the following actions are performed:

- If the key usages field from the request is present, check the certificates key usage against the key usages field. If the latter set is a subset of the former one then proceed otherwise stop and restart the process with a new certificate from the stack. This certificate will be returned with the appropriate error message invalid key usage.
- Start the certificate chain creation sub-module with this certificate and the trust anchors as parameters. Note that, although we say that the trust anchors are parameters to many modules, in the actual implementation they are global variables and can be accessed by all the modules. This sub-module will be detailed below.
- If the sub-module does not return a path, get the generated error message and keep it for later use when the response will be created and restart the process with another certificate, if one exists.
• If the chain creation resulted in a chain, store this chain for latter use and check the actions that this module should do. If the desired action is to build a path then return the path to the validation module. Otherwise proceed to the next step.

• For each certificate in the created path, initialize a validation context this is an internal OpenSSL structure used to validated a certificate. The startup data depends on the type of actions the server should perform and on extensions fields presented in the request. If the actions should validate the certification path then just add the extensions fields to the context variable. On the other hand, if revocation checking should also be performed, then also add CRL and OCSP verification to the context. This is done by specifying any CRL repositories (files, x500 directories etc) and any OCSP responders specified.

The chain creation sub-module takes a certificate as input and stack of trust anchors. This module creates a chain with the certificate in question as the first one. It then searches the certificates locations for the certificates of the issuer of the current certificate. When a new certificate is found, it is added to the chain, the current certificate becomes the one just retrieved and the process starts again. This process stops until one of the trust anchors is added to the chain.

The responder module is the simplest of them all. The only complicated part of it is to generate the want backs from the results of the computations and to interact with the ASN.1 module in order to create an encoded response. It is also in charge of signing the response and populate the response fields according to the error messages received from the core. After all these tasks, it communicate the response back to client.

3 Conclusion

Although much work has been done on implementing the certificate validation protocol, little has been done on implementing the validation policy protocol. This is very important for the future use of this protocol, since this protocol is more likely to be used inside enterprise boundaries than on the public internet. In this context, the validation policies are crucial components of the system, since a company has different levels of security at different levels inside their networks, and granting access to the entertainment server is not that same thing as granting access to the research or the finance information. The latter two would require much more security and would have more strict policies.

Another component which was not implemented at all is the validation of Attribute Certificates (AC). This is because OpenSSL doesn’t currently support this type of certificates. Adding support for them was not the scope of this project, mainly because this is a considerable larger task than adding support for SCVP. The work for adding AC support is however in progress.

References:


