IPSECKEY based Authentication for strongSwan using DNSSEC

Reto Guadagnini

August 29, 2012
IPSECKEY based Authentication for strongSwan using DNSSEC

Student
Reto Guadagnini
Via Dim Lej 5
CH-7500 St. Moritz

Advisor
Prof. Dr. Andreas Steffen
Hochschule für Technik Rapperswil
Oberseestrasse 10
CH-8640 Rapperswil
Abstract

strongSwan is a free implementation of the IKE protocol for Linux which allows the creation of IPsec based VPNs. strongSwan supports all authentication methods that are part of the IKEv2 protocol standard. In a previous project we proposed a new authentication method that we called IPSECKEY-AUTH. This new method stores the public keys of the VPN gateways as IPSECKEY resource records in the domain name system (DNS) and uses them for the authentication of the VPN endpoints.

This document refines the definition of the IPSECKEY-AUTH method, describes the architecture that we developed for its implementation in strongSwan and shows the final integration into strongSwan. Tests that we conducted on virtual machines show that our implementation efforts were successful and that the IPSECKEY-AUTH method implementation for strongSwan is ready to be tested in the field.
Contents

1 Project description ........................................................................................................... 8
   1.1 Goals .................................................................................................................. 8
   1.2 Expected results ............................................................................................... 8

2 Overview ......................................................................................................................... 9

3 Basic concept ............................................................................................................... 10
   3.1 Pubkey authentication ................................................................................... 10
   3.2 IPSECKEY-AUTH ......................................................................................... 12
      3.2.1 IPSECKEY Resource Record .......................................................... 13
      3.2.2 Lifetime of an IPSECKEY ................................................................. 14
   3.3 What do we have to implement? .................................................................... 15

4 Analysis of the strongSwan architecture ................................................................. 16
   4.1 Programming language .............................................................................. 16
   4.2 Static view .................................................................................................... 19
   4.3 Runtime View ............................................................................................... 21
      4.3.1 Authentification of VPN gateways and their IKE messages ............ 22

5 Architecture and implementation ................................................................................. 23
   5.1 DNS resolver ................................................................................................. 24
      5.1.1 Resolver interface ................................................................................. 25
      5.1.2 Unbound resolver plug-in ................................................................. 26
   5.2 IPSECKEY credential set ........................................................................... 33
      5.2.1 IPSECKEY credential set implementation .................................. 33
      5.2.2 Configuration of the IPSECKEY plug-in ....................................... 34

6 Testing .......................................................................................................................... 35
   6.1 Test environment ............................................................................................. 35
      6.1.1 Setup of the test environment .......................................................... 36
      6.1.2 Start of the test environment ............................................................. 43
   6.2 Tests ................................................................................................................ 44
      6.2.1 Successful authentication with IPSECKEYs ..................................... 44
      6.2.2 Retrieve the local public key from the DNS .................................... 45
      6.2.3 IPSECKEY RRSIG signature expired ........................................... 47
      6.2.4 DNS server not available ................................................................. 48

R. Guadagnini
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Open questions/points</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Sources</td>
<td>52</td>
</tr>
<tr>
<td>10</td>
<td>List of Acronyms</td>
<td>54</td>
</tr>
<tr>
<td>11</td>
<td>Appendix</td>
<td>55</td>
</tr>
<tr>
<td>11.1</td>
<td>Project plan</td>
<td>55</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Differences to the initial project plan</td>
<td>56</td>
</tr>
<tr>
<td>11.2</td>
<td>Listings</td>
<td>56</td>
</tr>
</tbody>
</table>
Figures

Figure 1: Pubkey based authentication ................................................................. 10
Figure 2: IPSECKEY-AUTH authentication method .............................................. 12
Figure 3: IPSECKEY RR format ........................................................................... 13
Figure 4: Size of the “Algorithm” field of an IPSECKEY RR .............................. 14
Figure 5: Corrected version of the IPSECKEY RRs format .................................. 14
Figure 6: Important components of strongSwan ..................................................... 19
Figure 7: Interface and plug-in based architecture of the libstrongswan [16] ......... 20
Figure 8: Architecture of Charon [18] .................................................................. 20
Figure 9: Runtime architecture of strongSwan (Charon) ........................................ 21
Figure 10: Parts of strongSwan which are responsible for the authentication [19] .... 22
Figure 11: Architecture of the IPSECKEY-AUTH method implementation .......... 23
Figure 12: Architecture of the DNS resolver facility for strongSwan (adapted from [2]) .................. 24
Figure 13: Resolver interface .............................................................................. 25
Figure 14: DNS traffic created by the program from Listing 26 ............................ 27
Figure 15: DNS traffic created by the program from Listing 3 ............................. 30
Figure 16: Implementation of the unbound resolver plug-in ............................... 31
Figure 17: IPSECKEY credential set for strongSwan (adapted from [19]) .......... 33
Figure 18: Implementation of the IPSECKEY plug-in ......................................... 33
Figure 19: DUMM test environment .................................................................. 35
Listings

Listing 1: Public interface of the class foo (content of the file “foo.h”) ........................................ 17
Listing 2: Private part of the class foo (content of the file “foo.c”) ................................................... 17
Listing 3: Trust chain validation with unbound and ldns ................................................................. 29
Listing 4: Configuration of the unbound resolver plug-in ............................................................... 32
Listing 5: Configuration of the IPSECKEY plug-in............................................................................ 34
Listing 6: Ruby script ”swancreate.rb” which we used to create the test environment .......... 37
Listing 7: Content of ”resolv.conf”..................................................................................................... 37
Listing 8: Content of ”/etc/trust.anchors” ......................................................................................... 37
Listing 9: Content of the ”named.conf.local” BIND configuration file ........................................... 38
Listing 10: Content of the ”named.conf.options” BIND configuration file ..................................... 38
Listing 11: Content of the file ”zones.strongSwan” ............................................................................ 38
Listing 12: Zonefile ”strongSwanZones/moon.local/db.moon.local” ............................................. 40
Listing 13: Zonefile ”strongSwanZones/sun.local/db.sun.local” ..................................................... 41
Listing 14: Zonefile ”strongSwanZones/local/db.local” ................................................................. 39
Listing 15: Contents of ”strongswan.conf” ....................................................................................... 41
Listing 16: Contents of ”ipsec.conf” .................................................................................................. 41
Listing 17: Contents of ”ipsec.secrets” ............................................................................................... 42
Listing 18: Contents of ”strongswan.conf” ........................................................................................ 42
Listing 19: Contents of ”ipsec.conf” .................................................................................................. 43
Listing 20: Contents of ”ipsec.secrets” ............................................................................................... 43
Listing 21: Ruby script ”swanstart.rb” .............................................................................................. 44
Listing 22: Log of a successfully established VPN with the IPSECKEY-AUTH method ............ 45
Listing 23: Log which shows how the local public key was retrieved from the DNS .............. 46
Listing 24: Log which results if the used IPSECKEYs are expired .................................................. 48
Listing 25: Log file in the case if no DNS server is present .............................................................. 49
Listing 26: Trust chain validation with ldns ....................................................................................... 58
1 Project description

strongSwan is a free implementation of the IKEv1 and IKEv2 protocols. It allows the creation of IPsec based VPNs under Linux [1]. For IKEv2 [12] strongSwan currently supports three different methods for the authentication of VPN endpoints:

- Pre-Shared Keys (PSK)
- Extensible Authentication Protocol (EAP)
- X.509 and PGP certificates

In a previous project [2] we investigated a possible fourth authentication method:

- Authentication with IPSECKEY DNS Resource Records and DNSSEC

This fourth authentication method (which we call IPSECKEY-AUTH for simplicity) uses special Resource Records to store the public keys of VPN endpoints in the DNS. The type of these Resource Records is called IPSECKEY and is defined in RFC 4025 [3].

With the DNS Security Extensions (DNSSEC) it is possible to authenticate Resource Records that are stored in the DNS. Therefore DNSSEC can be used to authenticate IPSECKEY Resource Records and the public keys of the VPN endpoints that are stored in them. This means that the public key of a VPN endpoint can be retrieved by querying the DNS for the corresponding IPSECKEY Resource Record. DNSSEC guarantees that the queried public key really is the public key of the VPN endpoint. Thus this public key can be used to authenticate the VPN endpoint.

In [2] we showed that this authentication method should work in theory and that the technologies for its implementation are available. The objective of this project is the implementation of this new authentication method for the strongSwan open source VPN software.

1.1 Goals

The main goal of this project is the implementation of the proposed IPSECKEY-AUTH authentication method for strongSwan.

The implementation must be able to authenticate a VPN endpoint using an IPSECKEY RR from the DNS. This IPSECKEY RR is to be retrieved by a forward DNS-lookup and must be validated via the DNSSEC trust chain.

1.2 Expected results

- Concept and software architecture for an IPSECKEY-AUTH strongSwan plug-in which allows the authentication of VPN endpoints using IPSECKEY DNS Resource Records and DNSSEC.
- Implementation, testing and documentation of the IPSECKEY-AUTH strongSwan plug-in.
2 Overview

The present project report documents our procedure for the implementation of the IPSECKEY-AUTH method for strongSwan and its results. We have already introduced the basic ideas of the IPSECKEY-AUTH authentication method in [2], but we will summarize and refine them here, so that the reader gets a complete and actual picture of our solution. After this introduction in the IPSECKEY-AUTH method we will describe the results of our analysis of the strongSwan architecture, which we conducted to develop an understanding of the system that we had to extend. Then we will describe the architecture that we developed for the implementation of the IPSECKEY-AUTH method and its implementation. We will also show how we have tested the implementation and the results of these tests. Finally we will mention open questions in conjunction with the IPSECKEY-AUTH method and its implementation in strongSwan.
3 Basic concept

In the present chapter we will repeat the most important insights which we gained in our previous project [2] about the IPSECKEY-AUTH method, so that the reader knows the basic concepts which are necessary to understand the rest of this report.

strongSwan supports the so called “Pubkey” authentication method. The proposed IPSECKEY-AUTH authentication method bases on this “Pubkey” method, so we start with a short description of the “Pubkey” method. Then we will show how this method can be extended to the IPSECKEY-AUTH method. Finally we will close this introductory chapter with some thoughts on the implementation of the IPSECKEY-AUTH method in strongSwan.

3.1 Pubkey authentication

For the description of the “Pubkey” authentication method we use the following scenario:

![Figure 1: Pubkey based authentication](image)

In our scenario we have two strongSwan VPN gateways:

- "moon.foo.net"
- "sun.bar.net"

Both of them are connected to the internet and their respective LAN. The VPN gateways are configured to build up an IPSec VPN tunnel between them over the internet, which connects their respective LANs to a single “Virtual Private Network”. They use the IKEv2 protocol to establish this tunnel. IKEv2 uses four messages (green and blue arrows in Figure 1) to establish the VPN tunnel [12]. Now we have a look at these four messages (which form the internet key exchange) to understand the IKEv2 protocol and how the “Pubkey” authentication method is integrated in it:

1.) The VPN gateway “moon.foo.net” initiates (thus it is called the “initiator”) the internet key exchange by sending an IKE_SA_INIT message to the other VPN gateway “sun.bar.net” (IKEv2 is a so called peer to peer protocol. This means that both VPN gateways could initiate an internet key exchange. In our example we chose the gateway “moon.foo.net” as initiator). The IKE_SA_INIT message contains information for a Diffie-Hellman key exchange and proposes security parameters for the IKE_SA (the IKE_SA is used the secure the IKE messages and is built up between the two VPN gateways with the first two IKE_SA_INIT messages).
2.) The VPN gateway “sun.bar.net” receives the IKE_SA_INIT message and answers (which makes it the “responder”) with an IKE_SA_INIT message. This second IKE_SA_INIT message completes the Diffie-Hellman key exchange between the VPN gateways and the establishment of the IKE_SA (the responder chooses from the proposed security parameters the parameters that he will use for the IKE_SA and tells his decision the “initiator” through the IKE_SA_INIT message).

3.) The initiator (“moon.foo.net”) sends an IKE_AUTH message to the responder (“sun.bar.net”). This message is protected by the IKE_SA that was established through the IKE_SA_INIT messages between the VPN gateways. The IKE_AUTH message contains in the “Auth” field a signature over parts of the first IKE_SA_INIT message that was calculated by the initiator with his private key. Besides the signature, the IKE_AUTH message contains also information about the identity of the initiator and the CHILD_SA he would like to establish with the responder (the CHILD_SA defines an IPSec VPN tunnel).

The responder receives the IKE_AUTH message. Now the “Pubkey” authentication method comes in action:

The responder loads the public key which is bound to the identity (“moon.foo.net”) of the initiator from his local file system or retrieves it as X.509 certificate directly from the initiator (the certificate is transmitted through the IKEv2 protocol and is validated with the help of a PKI) and uses it to verify the signature of the first IKE_SA_INIT message and thus to verify the identity of the initiator (authentication of the initiator).

4.) If the responder was able to authenticate the initiator successfully with the respective public key, he generates an IKE_AUTH message with the chosen parameters for the CHILD_SA and stores a signature over the second IKE_SA_INIT message in the “Auth” field of this message. He then sends this IKE_AUTH message to the initiator. The initiator receives the IKE_AUTH message from the responder. Now the “Pubkey” authentication method comes again in action:

The initiator loads the public key which is bound to the identity (“sun.bar.net”) of the responder from his local file system or retrieves it as X.509 certificate directly from the initiator (the certificate is transmitted through the IKEv2 protocol and is validated with the help of a PKI) and uses it to verify the signature of the second IKE_SA_INIT message and thus to verify the identity of the responder (authentication of the responder).

If the initiator was able to successfully validate the identity of the responder, the CHILD_SA between the initiator and responder is established and the VPN tunnel which connects the two VPN gateways and their LANs becomes active.
### 3.2 IPSECKEY-AUTH

To demonstrate the IPSECKEY-AUTH authentication method we extend the scenario, which we used to describe the “Pubkey” authentication method, as follows:

![Diagram of IPSECKEY-AUTH authentication method](image)

**Figure 2: IPSECKEY-AUTH authentication method**

The only difference to the “Pubkey” scenario is that we added the “Domain Name System” (DNS), which is required by the IPSECKEY-AUTH method. For the “Pubkey” authentication method the public key of the VPN gateway “moon.foo.net” has to be stored in the file system of the VPN gateway “sun.bar.net” and vice versa. This means, that the public keys have to be exchanged over an authenticated channel between the VPN gateways before the IKEv2 protocol can be used to create a VPN tunnel between them (the system operator could store the public keys for example on an USB stick and install them manually on the VPN gateways).

The IPSECKEY-AUTH authentication method uses the DNS to store the public keys of the VPN gateways. The public key of a VPN gateway is stored in a special resource record called IPSECKEY (specified in [3]) in the DNS. Now we explain how the IPSECKEY-AUTH method is integrated in the IKEv2 protocol. We therefore have a look at the IKEv2 messages which are exchanged between the VPN gateways in our extended scenario (Figure 2) and compare them to the messages of the “Pubkey” authentication scenario (Figure 1):

1.) Remains unchanged.

2.) Remains unchanged.

3.) The way in which the responder retrieves the public key of the initiator ("moon.foo.net") is changed. The responder queries the DNS for the IPSECKEY RR of the initiator "moon.foo.net" to retrieve the public key of the initiator (4.) a) in Figure 2). He validates the retrieved IPSECKEY RR using DNSSEC (DNSSEC guarantees the authenticity of the retrieved IPSECKEY and the public key which is contained in it). Then he uses the public key, which is contained in the IPSECKEY RR, to verify the signature, which is contained in the first IKE_AUTH message, and thus to verify the identity of the initiator (→ authentication of the initiator). The other operations which are part of step 3.) remain unchanged.
4.) The way in which the initiator retrieves the public key of the responder (“sun.bar.net”) is also changed. The initiator queries the DNS for the IPSECKEY RR of the responder “sun.bar.net” to retrieve the public key of the responder (4.) b) in Figure 2). He validates the retrieved IPSECKEY RR using DNSSEC. Then he uses the public key, which is contained in the IPSECKEY RR, to verify the signature, which is contained in the second IKE_AUTH message, and thus to verify the identity of the responder (→ authentication of the responder). The other operations which are part of step 4.) remain unchanged.

As shown the IPSECKEY-AUTH authentication method changes just the way in which the public keys of the VPN gateways are retrieved. The other operations remain the same as with the “Pubkey” authentication method. Especially the IPSECKEY-AUTH method does not change the IKEv2 protocol at all, because the IKEv2 protocol specification does not say how the public keys have to be retrieved [12].

3.2.1 IPSECKEY Resource Record

The format of an IPSECKEY resource record is defined as follows [3]:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precedence</td>
<td>gateway type</td>
<td>algorithm</td>
<td>gateway</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gateway</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>public key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*Figure 3: IPSECKEY RR format*

For a description of the various fields we refer to the report of our previous project [2] or the original RFC 4025 [3].

We noticed that the algorithm field is just 7 bit long, what is an error in the ASCII graphics of the RFC 4025 [3] (red numbers in Figure 3).

We therefore checked how the DNS server BIND 9 handles IPSECKEY RR and noticed that he uses 8 bits for the algorithm field:
After a discussion with Prof. A. Steffen we decided, to use an 8 bit long "Algorithm" field in our implementation. This led us to the following format for IPSECKEY RRs:

![Corrected version of the IPSECKEY RRs format](image)

**Figure 5: Corrected version of the IPSECKEY RRs format**

### 3.2.2 Lifetime of an IPSECKEY

The IPSECKEY-AUTH method limits the lifetime of an IPSECKEY. It uses the “Signature Inception” and “Signature Expiration” values of the RRSIGs of the IPSECKEY to determine the validity period of the IPSECKEY.
3.3 What do we have to implement?

We would like to implement the IPSECKEY-AUTH method in strongSwan. We already investigated in [2] which components we have to implement for the IPSECKEY-AUTH method. We have to implement:

- A DNS Resolver plug-in
- A plug-in which fetches the IPSECKEY RRs through the DNS Resolver plug-in from the DNS and uses them for the authentication of the VPN gateways

In [2] we evaluated various DNS Resolver libraries and decided that the ldns library [4] would be the most suitable library for the implementation of the DNS Resolver plug-in.
4 Analysis of the strongSwan architecture

To design an implementation of the IPSECKEY-AUTH method for strongSwan, some knowledge of the strongSwan architecture is necessary. So our initial step was the investigation of the strongSwan 4.6.2 / 4.6.3 architecture. To gain an insight in the architecture of strongSwan we read parts of the strongSwan source code with its comments, used the information from the strongSwan Wiki (available via [1]) and talked with Prof. Andreas Steffen and Tobias Brunner from the strongSwan development team. We also consulted project reports from earlier strongSwan related projects, which were performed at the Hochschule für Technik Rapperswil (HSR).

Because we had to extend the authentication process of strongSwan to implement the IPSECKEY-AUTH method we led our focus during our investigations on the authentication process. Therefore the result of our investigation is an overview over the strongSwan architecture with focus on the authentication process.

4.1 Programming language

Nearly 100% of the strongSwan source code is written in C. C is a procedural, structured and static typed language [13]. strongSwan is implemented in a handcrafted object oriented style of C [14].

We use the example class “foo” to show how classes are implemented in this handcrafted object oriented C style. The class “foo” consists of a private field “val” of type “int” and two public methods “get_val()” and “destroy()”. The method “get_val()” returns the value of the field “val” and the method “destroy()” destroys an instance of the class. The class foo is implemented with the help of two structs “foo_t” and “private_foo_t” in two separate files “foo.h” and “foo.c”. The struct “foo_t” represents the public interface of the class and is contained in the file “foo.h”:

```c
/**
 * @defgroup foo foo
 * @{
 * @ingroup foo
 */
#endif
#define FOO_H_
#define FOO_H_

typedef struct foo_t foo_t;
#include <library.h>
/**
 * Class foo.
 * The public interface of the class foo is formed by this struct.
 */
struct foo_t {
  /**<
   * The class foo has a public getter method "get_val" which returns
   * the value of the private field "val".
   * @return val
   */
  int (*get_val)(foo_t *this);
  /**<
   * Destroy the foo instance.
   */
  void (*destroy) (foo_t *this);
};
```
Listing 1: Public interface of the class foo (content of the file "foo.h")

The file “foo.h” (Listing 1) contains beside the “foo_t” struct also the declaration of the constructor function “foo_create” of the foo class. The “foo_t” struct contains two fields “get_val” and “destroy” which store pointers to the functions (methods) that are applicable to objects of the foo class. A client uses an instance of the foo class through a pointer of the type “foo_t *”. He is therefore only able to use the public functions (methods) of the class. The second struct “private_foo_t” is defined together with the functions (methods) and the constructor of the class in the file “foo.c”:

Listing 2: Private part of the class foo (content of the file "foo.c")
The second struct “private_foo_t” (Listing 2) represents the private internals of the foo class. Its first field has to be called “public” and has to be of the type “foo_t” to ensure that the class implements its public interface that is defined by the struct “foo_t”. The following fields of the “private_foo_t” struct can be freely chosen and represent the private fields of the foo class. Beneath the struct “private_foo_t” the functions/methods of the foo class are defined with the help of the “METHOD” macro. These functions/methods take a pointer (normally called “this”) of type “private_foo_t *” as parameter which points to the foo object (represented by the “private_foo_t” struct) on which they have to operate. At the end of the file “foo.c” the constructor function “foo_create” of the foo class is defined. The constructor function creates an instance of the “private_foo_t” struct which represents an object of the foo class, initializes all the fields of this struct and returns a pointer of type “foo_t *” to the field “public” of this struct through which a client can access the public part of the created object.

The demonstrated style of object oriented programming in C supports dynamic binding and polymorphism. For simplicity inheritance is only used through interfaces (an interface is just another struct. A field of the type of this struct has to be included as the first field in the public interface struct of a class, if the class should implement the interface). As shown in Listing 1 and Listing 2 strongSwan contains various macros which facilitate the development in this handcrafted object oriented C style (the macros mainly prevent cryptic expressions which result from the object oriented approach).

To automate the strongSwan building process the strongSwan project uses the GNU Build System (also known as Autotools).
4.2 Static view

strongSwan mainly consists of two Unix/Linux daemons “Charon” and “Pluto” which implement the IKE protocol. Charon is responsible for the IKEv2 protocol, while Pluto implements the first version of the IKE protocol. Because our goal is the implementation of the IPSECKEY-AUTH method for Charon we only cover its architecture and we leave Pluto aside.

Note: strongSwan 5.0, released at the end of June 2012, does not contain Pluto anymore. Instead it contains a monolithic version of Charon which supports both IKEv1 and IKEv2 [22]. Our analysis, design and implementation bases on the version 4.6.x of strongSwan, which was the actual version during the most time of our project.

The main components of strongSwan are:

Figure 6: Important components of strongSwan

The Charon component has to start and initialize the Charon daemon. The most parts of the Charon daemon are implemented in the libcharon and libstrongswan. strongSwan uses for the authentication of VPN endpoints classes from the libcharon. The libcharon uses classes of the libhydra and the libstrongswan. The libhydra contains daemon specific code which is used by both daemons Charon and Pluto [15]. The libstrongswan offers classes which provide basic functionalities for strongSwan like cryptographic algorithms and the other strongSwan components relay on the libstrongswan. The strongSwan components highly use interfaces and implement a plug-in mechanism. These interfaces and the plug-in mechanism make strongSwan highly configurable and extendable.

The following figure illustrates some of the interfaces of the libstrongswan and the corresponding plug-ins which implement one of these interfaces:
As we can see from Figure 7, the libstrongswan defines an interface called “crypto” which allows us to access cryptographic facilities. The “aes” plug-in provides specific cryptographic facilities which conform to the “crypto” interface and can therefore be accessed through the interface if the “aes” plug-in was loaded. This is a typical example for the use of interfaces and plug-ins in strongSwan.

We conclude our static view on strongSwan with the architecture of the IKEv2 daemon Charon:

Charon’s design is multithreaded. Charon uses a pool of threads (called “Processor” in Figure 8) to perform its tasks. The daemon stores incoming IKE messages in the “receiver” queue and its threads process (and generate answers) the incoming IKE messages according to the
IKEv2 protocol and the SA to which they belong. Over its credential component Charon retrieves the credentials which it uses for the authentication of the VPN end points. The backend allows Charon to read his configuration from various sources (for example an SQL database could be used as backend). Over the kernel interface Charon installs the IPSec SAs in the operating system kernel. The kernel is responsible to apply the ESP/AH protocols as described in the IPSec SA, Charon’s task is limited to the IKEv2 protocol.

4.3 Runtime View

We found in [17] a diagram, which gives a good overview over the multithreaded architecture of strongSwan. We recreated and adapted this diagram to describe strongSwan from a runtime perspective:

![Figure 9: Runtime architecture of strongSwan (Charon)](image)

Charon’s Processor contains a pool of threads and a Jobqueue. The processor picks a job from the Jobqueue and assigns the job to a thread. The thread then performs the job (it executes the function which is contained in the job) and notifies the Processor, when it has finished the job. After the thread has finished his job it waits until the Processor assigns a new job. To do the work which is described in a job, the thread uses other components of strongSwan like the ike_sa_manager.

The Jobqueue of the Processor is fed by the Receiver and the Scheduler. The Receiver listens on the Socket for incoming IKE messages. If he receives an IKE message over the Socket he creates a “process_message_job” for the message and adds this job to the Jobqueue of the Processor. The IKE message is then processed through the Processor and its threads. The Receiver has no separate thread. There is just a “receiver” job which is executed regularly by the Processor.

Through the Scheduler it is possible to schedule jobs for an execution in the future. The Scheduler has internally a heap in which he stores the scheduled jobs ordered by the time
when they have to be executed. Like the Receiver, the Scheduler has a “scheduler” job instead of a separate thread. This “scheduler” job is regularly executed by the Processor and adds the scheduled jobs, which should be actually executed, to the Jobqueue of the Processor.

Jobs can generate new IKE messages while they are processed. To send such a message the job calls the “send” function of the Sender with the message (packet) as parameter. The Sender has a Sendqueue in which he buffers the outgoing messages until he was able to send them over the Socket. Instead of an own thread, the Sender also uses a special “sender” job, to perform his work.

A job can consist of various tasks. All tasks of a job are handled by the thread to which the job was assigned.

### 4.3.1 Authentication of VPN gateways and their IKE messages

The authenticator which is part of libcharon is used by strongSwan (Charon) to authenticate the VPN gateways and their IKE messages. The private/public keys which the authenticator needs to sign and authenticate IKE messages (and therefore to authenticate the VPN gateways) are stored in so called “credential-sets”. The credential-sets are maintained by the credential-manager:

![Figure 10: Parts of strongSwan which are responsible for the authentication [19]](image)

Based on the ID of a VPN gateway and with the help of the credential-manager the authenticator searches in the different credential-sets for the keys, which he needs to authenticate the VPN gateways and their IKE messages.

There are different classes for the different authentication methods (EAP, PSK, Pubkey) which implement the authenticator interface and act as authenticator. The “ike_auth” task chooses the appropriate authenticator and performs with his help the authentication.
5 Architecture and implementation

As shown in chapter 3, the IPSECKEY-AUTH method that we intend to implement in strongSwan corresponds to the “Pubkey” authentication method. The only difference between the IPSECKEY-AUTH method and the classical “Pubkey” authentication method is the way in which the public keys of the VPN gateways are retrieved. The easiest way to implement the IPSECKEY-AUTH method in strongSwan is therefore to extend the “Pubkey” method, so that it can also retrieve the required public keys by querying the DNS for IPSECKEY RRs.

We decided to implement the IPSECKEY-AUTH method by extending the “Pubkey” method. The authenticator of the “Pubkey” method retrieves the keys, which he needs for the authentication, through the credential manager as shown in chapter 4.3.1. The credential manager retrieves the desired keys through the various credential sets. Therefore we decided to implement a new type of credential set “ipseckey credential set” through which the “Pubkey” authenticator can retrieve the keys by querying the DNS for IPSECKEY RRs. The behavior of the “Pubkey” method with such an “ipseckey credential set” corresponds to the IPSECKEY-AUTH method; therefore we get what we want through this extension: An implementation of the IPSECKEY-AUTH authentication method for strongSwan. The “ipseckey credential set” needs a DNS resolver to retrieve the IPSECKEYs from the DNS. This DNS resolver has to guarantee the authenticity of the retrieved IPSECKEY RRs via DNSSEC. We will define an interface for DNS resolvers in the libstrongswan and create a plug-in which implements this interface to implement the required DNS resolver in strongSwan. These thoughts led us to the following architecture for the implementation of IPSECKEY-AUTH authentication method:

![Architecture of the IPSECKEY-AUTH method implementation](image-url)

Figure 11: Architecture of the IPSECKEY-AUTH method implementation
As we can see from Figure 11, we have to implement an “ipseckey credential set”, a resolver interface and a resolver plug-in to add the IPSECKEY-AUTH method to strongSwan. We have to register our new “ipseckey credential set” at the credential manager. The “Pubkey” authenticator can then use IPSECKEYs to authenticate VPN gateways. He retrieves the IPSECKEYs through our “ipseckey credential set” which in turn retrieves them with the help of the DNS resolver plug-in (red arrows in Figure 11).

In the following parts of this chapter we will describe in detail how we have realized the components of the architecture that we have divided here (Figure 11) to implement the IPSECKEY-AUTH method in strongSwan. The implementation will base on version 4.6.3 of strongSwan.

5.1 DNS resolver

This chapter describes the design and implementation of the DNS resolver facility for strongSwan.

We decided to extend the libstrongswan with an interface for DNS resolvers and to implement a concrete resolver as a libstrongswan plug-in (“unbound”) which implements the resolver interface:

![DNS resolver interface](image)

*Figure 12: Architecture of the DNS resolver facility for strongSwan (adapted from [2])*

This modular architecture enables us to change the concrete resolver implementation any time if necessary.

In the following sections we describe the resolver interface, the implementation of the corresponding resolver plug-in and the design decisions we made. Finally we describe the test case that we are using to test our implementation.
5.1.1 Resolver interface

Our resolver interface was inspired by the ldns [4] and libunbound [5] DNS resolver libraries and consists of a package in the libstrongswan with the following interfaces and classes:

**Figure 13: Resolver interface**

5.1.1.1 resolver_t

A DNS resolver has to implement the interface "resolver_t" which provides the query method. The query method encapsulates the main functionality of a DNS resolver: Querying the DNS for the desired RRsets. When the query method of the resolver is called the resolver queries the DNS for the desired RRset and returns as a result an object of type "resolver_response_t" which contains the results of the query (desired RRset, security status, etc.).

5.1.1.2 resolver_manager_t

In the libstrongswan an object of class "resolver_manager_t" is used to manage the implementations of the resolver interface (resolver plug-ins). Resolver plug-ins can register the constructor of their resolver implementation through a call to the method "add_res" of the resolver_manager.

If another component of strongSwan needs a DNS resolver it will call the create function of the resolver_manager, which delivers an instance of the registered resolver to it.

5.1.1.3 resolver_response_t

As a result of a query a resolver returns an object which conforms to the interface "resolver_response_t", thus it returns a resolver_response. A resolver_response contains information about the results of a query (was the query successful, DNSSEC security state of the results, etc.) and the actual results of the query (the queried RRset if the query was successful). The results of a query and the information about them can be accessed through the methods of the resolver_response.
5.1.1.4 dnssec_status_t
A resolver_response has a security state of type “dnssec_status_t” which indicates how trustworthy the data in response is in terms of the security mechanisms provided by DNSSEC. According to RFC 4033 [8] there are four DNSSEC security states “secure”, “insecure”, “bogus” and “indeterminate” for which we defined the enumeration type “dnssec_status_t”.
A DNS resolver implementation (plug-in) which supports the validation of DNS RRsets through DNSSEC will set the security state of the response properly. DNS resolver implementations with no DNSSEC support will have to set the security state of a response always to “indeterminate”.

5.1.1.5 rr_set_t
An object of the class “rr_set_t” represents an RRset. An RRset consists of a set of RRs with the same label, type and class but different data as defined in RFC 2181 [9]. Our class “rr_set_t” uses a list of RRs to store all RRs which form the RRset. These RRs can be accessed through an enumerator. Besides the RRs of the RRset an object of class “rr_set_t” also contains the DNSSEC signature RRs (RRSIGs) which sign the RRs of the RRset in a separate list. The RRSIG RRs can also be accessed through an enumerator.

5.1.1.6 rr_t
Objects of class “rr_t” represent RRs as defined in RFC 1035 [10].

5.1.1.7 rr_type_t
According to RFC 1035 [10] each RR has a type. The possible types are assigned by the IANA (for an actual list look at [11]) and we represent them through the enumeration type “rr_type_t”.

5.1.1.8 rr_class_t
Besides a type a RR has also a class [10]. The IANA publishes a list with the classes of RRs which are actually defined [11]. We use the enumeration type “rr_class_t” to represent the different RR classes.

5.1.2 Unbound resolver plug-in
In this chapter we will describe how we used the libunbound [5] to implement the “unbound” DNS resolver plug-in. But first we will show how we came up to use the libunbound (in conjunction with the ldns library [4]) instead of the various other DNS resolver libraries that exist.

5.1.2.1 Why libunbound?
We decided in [2] that the ldns resolver library would be most suitable for the implementation of a DNS resolver in the libstrongswan. Before we have started with the implementation of an ldns based resolver plug-in for the libstrongswan we wrote a little program (source code in Listing 26 chapter 11.2) which does just that what a resolver plug-in has to do for our implementation of the IPSECKEY-AUTH method with the help of the ldns library:
1. Load a trust anchor for DNSSEC.
2. Query the DNS for a RR.
3. Validate the retrieved RR with DNSSEC using the trust anchor.

Our little example program uses the DNSKEY RRs of a domain (given by an URL) as trust anchor. Thus it queries in step one this “trust anchor domain” for its DNSKEY RRs and uses them as trust anchor (this method to retrieve the trust anchor is not secure at all, but it is sufficient for demonstration purposes). Step three of the program is implemented by a call to the ldns function “ldns_verify_trusted()”. In step three the whole trust chain validation is performed, for this reason step three is security critical. Thus we investigated how the function “ldns_verify_trusted()” implements this trust chain validation.

To do so we reused the DNS zonefiles from [2] to build a DNS test scenario. We then used this scenario to test our program as follows:

We executed it with the domain “local.” as trust anchor and the A RR of the domain “ns1.moon.local” as RR to query. During its execution we captured with Wireshark all DNS packets that have been exchanged with the DNS:

![Figure 14: DNS traffic created by the program from Listing 26](image)

Figure 14 shows the result of the capture. As we can see from the capture our program first queries the DNS for the DNSKEY RRs of the domain “local.” to get the trust anchor (Step 1; packet number 4-5 in Figure 14). Then it queries the DNS for the desired A RR of the domain “ns1.moon.local.” (Step 2; packet number 6-7 in Figure 14). Finally it queries the DNS for all DNSKEY/DS RRs which form the trust chain from the queried A RR up to the trust anchor (Step 3; packet number 8-13 in Figure 14). This means, that the function “ldns_verify_trusted()”, which we call in our example program to implement step three, builds up the whole trust chain on its own and uses this trust chain to validate the queried A RR from step 2.

→ The validation of a RR is performed by the function “ldns_verify_trusted()” as expected. Unfortunately we noticed during our experiments with our little example program, that the ldns library does not do any caching. Each time we use the “ldns_verify_trusted()” function to validate a RR the whole trust chain is built up again by querying the DNS for the necessary
DNSKEY/DS Resource Records.
The absence of caching mechanisms in the ldns library is not ideal for the performance of our implementation of the IPSECKEY-AUTH authentication method in strongSwan. So we talked with Prof. Andreas Steffen about this performance issue. He said that performance is critical for the proposed IPSECKEY-AUTH authentication mechanism and therefore there must be some caching functionality in the DNS resolver. During further investigations (reading the source code of the ldns library) we were not able to find any caching mechanisms in the ldns library. Thus we posted a question to the user mailing list of the ldns library asking if there really are any caching mechanisms in the library and received the following answer:

"Proper caching is not an ambition of ldns; At NLnet Labs we also distribute a resolver library that does just that: libunbound. I suggest you to have a look at it. ..." [6]

Now we had four possibilities:

1. Use the ldns library and implement an own caching mechanism.
2. Use the libunbound (at which we already had a look in [2]).
3. Search for a suitable DNS resolver library.
4. Build the whole DNS resolver plug-in from scratch.

Because we already had evaluated a number of DNS resolver libraries in [2] and the libunbound did not looked bad during our evaluation (the libunbound is easy to use and thread safe [5]) we tried to use it to implement our little example program again. We were able to reimplement our little example program using the libunbound and noticed, that the libunbound really does caching. But we also noticed, that the libunbound does not offer the parsing functionality for DNS packets that we need to access the RRs and RRSIGs which are contained in them. So we successfully tried to use the libunbound in conjunction with the ldns library. The following program demonstrates the results of our efforts:

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <arpa/inet.h>
#include <unbound.h>
#include <ldns/ldns.h>

int main(void)
{
    struct ub_ctx* ctx;
    struct ub_result* result;
    int retval;
    char *resolv_conf = "/etc/resolv.conf";
    char *trusted_key_file = "root_keys";
    /* query the A RR of the domain */
    char *url_to_query = "www.switch.ch";

    /* create context */
    ctx = ub_ctx_create();
    if (!ctx) {__asm__ "movl %eax, %edx" ;
        printf("error: could not create unbound context\n"); return 1; }
    /* read resolv.conf for DNS proxy settings (from DHCP) */
    if (ub_ctx_resolvconf(ctx, resolv_conf) != 0) {
        printf("error reading resolv.conf: %s. errno says: %s\n", ub_strerror(retval), strerror(errno)); return 1; }
    /* read /etc/hosts for locally supplied host addresses */
```
if((retval=ub_ctx_hosts(ctx, "/etc/hosts")) != 0) {
    printf("error reading hosts: %s. errno says: %s\n", 
           ub_strerror(retval), strerror(errno));
    return 1;
}

/* read public keys for DNSSEC verification from file */
if((retval=ub_ctx_add_ta_file(ctx, trusted_key_file)) != 0) {
    printf("error adding keys: %s\n", ub_strerror(retval));
    return 1;
}

/* perform three queries and trust chain validations for the A RR */
for (i=0; i<3; i++) {
    printf("Query number %i: \n", i);
    printf("===============\n");
    retval = ub_resolve(ctx, url_to_query, 
        /* TYPE A (IPv4 address) */ , 
        /* CLASS IN (internet) */ , &result);
    if(retval != 0) {
        printf("resolve error: %s\n", ub_strerror(retval));
        return 1;
    }
    /* use the ldns library to parse the received DNS packet*/
    ldns_status status;
    ldns_pkt *packet;
    ldns_rr_list *rr_set;
    if (result->havedata) {
        status = ldns_wire2pkt(&packet, (uint8_t*)result->answer_packet, 
            (size_t)result->answer_len);
        if (status != LDNS_STATUS_OK) { 
            printf("Error while parsing the DNS packet.\n");
        }
        /* get all A RRs which are contained in the answer section 
         of the DNS packet */
        rr_set = ldns_pkt_rr_list_by_type(packet, 
            LDNS_RR_TYPE_A, 
            LDNS_SECTION_ANSWER);
        /* print all of these RRs */
        printf("A RRs contained in the answer section of the DNS packet:\n");
        ldns_rr_list_sort(rr_set);
        ldns_rr_list_print(stdout, rr_set);
    } else {
        printf("Result has no data.\n");
    }
    /* show security status */
    if(result->secure) 
        printf("Result is secure\n");
    else if(result->bogus)
        printf("Result is bogus: %s\n", result->why_bogus);
    else
        printf("Result is insecure\n");
    /* clean up */
    ldns_rr_list_deep_free(rr_set);
    ldns_pkt_free(packet);
    ub_resolve_free(result);
}
/* destroy the unbound context */
ub_ctx_delete(ctx);
return 0;
}

Listing 3: Trust chain validation with unbound and ldns

The program in Listing 3 creates a context (instance) of the libunbound resolver and reads
the trust anchor from a file. Then it queries a domain three times for its A RRs and performs
each time a validation of the retrieved resource records. It prints the A RRs, which are
retrieved in each query, to the standard output.
For the DNS query operations the functions of the libunbound are used. The printing of the received RRs is realized using the functions of the ldns library.

We tested this program as follows:

The program queries the DNS for the A RRs of the domain www.switch.ch. We decided to use the DNSKEYs of the root domain “.” as trust anchor. We retrieved and stored them as follows in a text file (this method to retrieve the trust anchor is not secure at all, but it is sufficient for demonstration purposes):

```
dig . DNSKEY > root_keys
```

Then we started Wireshark to capture the DNS packets, that our program exchanges with the DNS Server and executed our program. The following picture shows the resulting Wireshark capture:

![Wireshark Capture]

**Figure 15:** DNS traffic created by the program from Listing 3

As we can see from Figure 15 our program first queries the DNS for the A RRs of www.switch.ch (packet number 30-33 in Figure 15). Then it builds up the trust chain from the root domain “.” (which is used as a trust anchor) down to the domain “switch.ch” by querying the corresponding DNSKEY/DS Resource Records (packet number 34-43 in Figure 15). This trust chain is then used by our program to validate the queried A RRs of the domain www.switch.ch. Our program queries the A RRs of the domain www.switch.ch for three times and performs a validation of these Resource Records each time when they were queried.

Figure 15 shows that the A RRs of the domain www.switch.ch were queried only once and that the trust chain for their validation was also only built up once. So what happened? The libunbound, which our program used to perform the DNS queries, cached the results of DNS queries which had already been performed. This means that our second and third query was directly answered by libunbound, which is exactly the behavior that we need for our DNS resolver implementation.

This led us to the following decision:
We use libunbound in combination with the ldns library to implement the resolver plug-in. So we get the best of both libraries: The caching functionality from the libunbound and the parsing functions of the ldns library.

5.1.2.2 Unbound resolver plug-in implementation
The unbound resolver plug-in that we implemented consists of the following classes:

Figure 16: Implementation of the unbound resolver plug-in

Now we will describe these classes in more detail.

5.1.2.2.1 unbound_plugin_t
The unbound_plugin_t class implements the plugin_t interface of the libstrongswan. It registers the unbound plug-in by the resolver_manager of the libstrongswan. Through the resolver_manager of the libstrongswan clients are then able to use the unbound resolver plug-in as DNS resolver.

5.1.2.2.2 unbound_resolver_t
The unbound_resolver_t class implements the resolver_t interface. It implements a security aware DNS resolver (a resolver which performs trust chain validation through DNSSEC; see RFC 4033 [8]) using the libunbound DNS resolver library.

5.1.2.2.3 unbound_response_t
This class implements the resolver_response_t interface. It therefore represents a resolver response to a DNS query. The unbound Resolver creates an object of this class through the constructor function "unbound_response_create_frm_libub_response()" out of the response that he gets from libunbound. The constructor function uses the ldns library to parse libunbound response.
5.1.2.4 unbound_rr_t
Instances of the unbound_rr_t class represent DNS RRs and conform to the interface rr_t. The constructor function of the unbound_rr_t class uses the lDNS library to parse the resource record which he gets as pointer to an ldns_rr struct.

5.1.2.3 Configuration of the unbound resolver plug-in
The unbound resolver plug-in can be configured through the following entries in the file “/etc/strongswan.conf”:

```plaintext
libstrongswan {
    plugins {
        resolver {
            resolv_conf = PATH_TO_RESOLV_CONF
            trust_anchor = PATH_TO_TA_FILE
        }
    }
}
```

Listing 4: Configuration of the unbound resolver plug-in

The configuration option “resolv_conf” tells the unbound plug-in were it can find the “resolv.conf” file, which is normally located under “/etc/resolv.conf”. The resolv.conf file tells the unbound plug-in which DNS servers it should use to perform the DNS queries. If the “resolv_conf” option is not set “/etc/resolv.conf” is used as default value.

The second option “trust_anchor” indicates the path to a file, where the unbound plug-in can find the trust anchors (DS/DNSKEY records) for the DNSSEC validation. The unbound plug-in will use these trust anchors to validate the retrieved RRs via DNSSEC (we have already covered the DNSSEC validation process in [2]). The file has to contain the trust anchors in the zone-file format: [domainname] [type] [rdata contents]. “/etc/trust.anchors” is used as default for the “trust_anchor” option if it is not set.

5.1.2.4 Test of the unbound resolver plug-in
To test our implementation of the unbound resolver plug-in we wrote an application called “dnssec” which uses the unbound plug-in to query DNS RRs from the BIND DNS server that we installed locally on our development machine.

To successfully run the “dnssec” program, the zonefiles that we already used in [2] have to be used by the locally installed BIND DNS server. The program can be run through the following command if strongSwan was compiled and installed:

```plaintext
ipsec dnssec
```

The tests with the “dnssec” application were successful, so we were convinced that our implementation of the resolver plug-in worked and we moved on to implement the other components for the IPSECKEY-AUTH method.

Note: We left the “dnssec” application in the strongSwan branch which we used to implement the IPSECKEY-AUTH method for documentation purposes. It should be removed when our branch is merged with the official strongSwan branch.
5.2 IPSECKEY credential set

As discussed at the beginning of chapter 5, beside the DNS resolver the IPSECKEY credential set is the other component that we have to implement in order to support the IPSECKEY-AUTH method in strongSwan:

![Diagram of IPSECKEY credential set](image)

**Figure 17: IPSECKEY credential set for strongSwan (adapted from [19])**

The ipseckey credential set has to conform to the credential_set_t interface and must be registered by the credential-manager. It will deliver the desired public keys, which are stored as IPSECKEY RRs in the DNS, to the authenticator. To retrieve the IPSECKEY RRs it will use the DNS resolver plug-in.

5.2.1 IPSECKEY credential set implementation

The IPSECKEY credential set is implemented as a plug-in (which is called “ipseckey”) for libcharon:

![Diagram of ipseckey plug-in](image)

**Figure 18: Implementation of the IPSECKEY plug-in**
5.2.1.1 ipseckey_plugin_t
The ipseckey_plugin creates an ipseckey_cred set and through the resolver-manager of the
libstrongswan a DNS resolver instance that is needed by this ipseckey_cred set to retrieve
the IPSECKEYs from the DNS. It registers the created ipseckey_cred set by the credential-
manager so that it can be used by the pubkey_authenticator to authenticate the VPN
gateways and their messages.

5.2.1.2 ipseckey_t
The class ipseckey_t represents IPSECKEY RRs as defined in RFC 4025 [3] but with one
difference: As described in chapter 3.2.1 we use 8 bits for the algorithm field of an IPSECKEY
RR instead of the 7 bits which were specified in RFC 4025.

5.2.1.3 ipseckey_cred_t
The ipseckey_cred_t class represents an IPSECKEY credential set. It is the central
component for the implementation of the IPSECKEY-AUTH method. Its method
“create_cert_enumerator()” returns an enumerator over the IPSECKEYs that belong to a
specific ID (FQDN). To create the enumerator, the “create_cert_enumerator()” method
queries the DNS with the help of the DNS resolver, which we have implemented in the
libstrongswan, for the required IPSECKEY RRs. The method checks if the DNSSEC
validation of these IPSECKEYs was successful, wraps them into objects of the class
pubkey_cert_t (more exactly in CERT_TRUSTED_PUBKEY objects) and returns an
enumerator over these “Pubkey” certificates. These “Pubkey” certificates are then used by the
pubkey_authenticator as they were ordinary “Pubkey” certificates.
Prof. A. Steffen extended the pubkey_cert_t class in version 4.6.3 of strongSwan with the
fields “notBefore” and “notAfter”, so that we are able to wrap the IPSECKEYs into objects of
the pubkey_cert_t class. These fields are necessary to store the validity period of the
IPSECKEYs. The “Signature Inception” value of the IPSECKEY’s RRSIG is used for the field
“notBefore” and the “Signature Expiration” value of the IPSECKEY’s RRSIG for the field
“notAfter” (we have defined the lifetime of an IPSECKEY in chapter 3.2.2).

5.2.2 Configuration of the IPSECKEY plug-in
If the IPSECKEY plug-in is loaded it can be enabled/disabled globally through the following
option in the “strongswan.conf” file:

```
charon {
  plugins {
    ipseckey {
      enable = TRUE
    }
  }
}
```

Listing 5: Configuration of the IPSECKEY plug-in
The default value is “FALSE”, thus the IPSECKEY plug-in is disabled by default.
6 Testing

To test our implementation of the IPSECKEY-AUTH method we used the “DUMM” (Dynamic UML Mesh Modeler) framework which is part of strongSwan [20]. DUMM can be used to set up virtual networks with User-Mode Linux (UML) virtual machines as clients. We created a virtual test environment with the help of DUMM and ran various test scenarios in it. The following subchapters document our test environment, the test scenarios that we have used and results of our tests. They demonstrate therefore also how the IPSECKEY-AUTH method can be configured in strongSwan and give ideas for possible deployment scenarios.

6.1 Test environment

We have created with the help of DUMM the following virtual test environment:

![DUMM test environment](image)

**Figure 19: DUMM test environment**

The test environment consists of five virtual linux machines. We will establish during our tests between the machines “gateway.moon.local” and “gateway.sun.local” a VPN with strongSwan, which connects the two private subnets 10.1.0.0/24 and 10.2.0.0/24. The subnet 192.168.0.0/24 is considered as a public network (although 192.168.0.0/24 clearly is a private address space). There is a BIND 9 DNS server running on the machine “Winnetou” which is part of the public subnet in the test environment. This DNS server contains the IPSECKEY RRs of the VPN gateways in its zonefiles. Its zones are secured through DNSSEC. If we use the IPSECKEY-AUTH method to authenticate the VPN gateways, they will query the DNS server for the relevant IPSECKEY RRs.
6.1.1 Setup of the test environment

We followed the How-to [20] to install the DUMM framework under "/home/reto/umldir" on our machine. Then we installed the extra software packages that we need for our tests as follows:

```bash
cd /home/reto/umldir
sudo -s
chroot master /bin/sh
apt-get install libunbound2
apt-get install bind9
apt-get install dnsutils
apt-get install ldnsutils
apt-get install libgmp-dev
apt-get install libcurl4-ssl-dev
```

We configured and installed strongSwan as follows in the DUMM test environment:

```bash
sudo -s
./configure --disable-pluto --enable-unbound --enable-ipseckey --prefix=/usr --sysconfigdir=/etc
DESTDIR=/home/reto/umldir/master make install
```

The configure parameters "--enable-unbound" and "--enable-ipseckey" enable our plug-ins which implement the IPSECKEY-AUTH method.

Now we were able to create the test environment through the following Ruby script:

```ruby
# Create Clients
alice = Guest.new("alice", "/home/reto/umldir/linux-dumm/linux", "/home/reto/umldir/master", "mem=64M conl=xterm")
bob = Guest.new("bob", "/home/reto/umldir/linux-dumm/linux", "/home/reto/umldir/master", "mem=64M conl=xterm")

# Create Gateways
moon = Guest.new("moon", "/home/reto/umldir/linux-dumm/linux", "/home/reto/umldir/master", "mem=96M conl=xterm")
sun = Guest.new("sun", "/home/reto/umldir/linux-dumm/linux", "/home/reto/umldir/master", "mem=96M conl=xterm")

# Create the DNS Server
winnetou = Guest.new("winnetou", "/home/reto/umldir/linux-dumm/linux", "/home/reto/umldir/master", "mem=96M conl=xterm")

# Create Bridges
br0 = Bridge.new("br0")
br1 = Bridge.new("br1")
br2 = Bridge.new("br2")

# Start VMs
alice.start
bob.start
moon.start
sun.start
winnetou.start

# Set hostname
alice.exec("echo alice > /etc/hostname")
bob.exec("echo bob > /etc/hostname")
moon.exec("echo moon > /etc/hostname")
sun.exec("echo sun > /etc/hostname")
winnetou.exec("echo winnetou > /etc/hostname")

# Attach the network interfaces
alice.add("eth0").connect(br1).add("10.1.0.10")
bob.add("eth0").connect(br2).add("10.2.0.10")
moon.add("eth1").connect(br1).add("10.1.0.1")
moon.add("eth0").connect(br0).add("192.168.0.1")
```
By executing this script, with the following commands, we created the test environment depicted in Figure 19:

```bash
sudo -s
ipsec irdumm swancreate.rb
```

We defined “winnetou” as DNS server for all our virtual machines through the entry:

```bash
nameserver 192.168.0.150
```

Finally we created the file “/etc/trust.anchors” with the trust anchors which strongSwan should use to validate the retrieved IPSECKEY RRs via DNSSEC:

```
; This is a zone-signing key, keyid 28338, for local.
; Created: 20111024112118 (Mon Oct 24 13:21:18 2011)
; Publish: 20111024112118 (Mon Oct 24 13:21:18 2011)
; Activate: 20111024112118 (Mon Oct 24 13:21:18 2011)
local. IN DNSKEY 256 3 5 AwEAAcuTF68pvPAO/W9P31p5hkrwWF8pv9gb66TBYJjc+6+oZ/iOami71 SLCTmLKN4y9Zi7muAYjQP8qFrkUOFYzKW6S7rDAVFmbzUDejG7pgdF yjtT6B7z0E+ybLPTUGKHaXyAwGAMG521GT+bSPqAPwGdgsuKOM4NWr3eB md/+derj ; This is a key-signing key, keyid 34354, for local.
; Created: 20111024112328 (Mon Oct 24 13:23:28 2011)
local. IN DNSKEY 257 3 5 AwEAAcH3CtFrKzr569N+37t1rXzCJ5QFMW2+x+jHdeFUj/cgv2i8gMH xwlwsR2VcWhCCTLGxw4a4Dejndzhi1Bj+TkEBRST8roaxDJ0vdiWhJYu/x J2rWvU8gWv9U0Ld+yv8CCKX0zdsn0WmYzfrD31KpLx+TKMC4HoGyKrZ4F gKCNZwXjOLF5wsQWUolerUt6e5a3N3eJz4EuqEkWf8aKBojcc7F75+7X2j H6r7p9r4sa7e3PhnoOAmwdZHkdQaK58fVnC6pXC6eTYVsJZGnlfi+h IQw3uTN7RGA1gYP8IeMa7m6VetmCXCsHfgoP4xtanDOzOLf1EX8fwmr Xk4jPiQ7zck=
```
We use the DNSKEY RRs of the domain "local." as trust anchors (a description of the DNS domains and zonefiles that we have used follows in chapter 6.1.1.1).

Now we were finished with the global configuration and started with the configuration of the individual virtual machines. We will describe the configuration of these machines in the following subchapters.

6.1.1.1 Configuration of winnetou

On the machine "winnetou", we had to configure the BIND DNS server. We created the following zonefiles / configuration files with the IPSECKEY RRs in the directory "/home/reto/umldir/guests/winnetou/diff/etc/bind", which represents the bind configuration directory of the machine "winnetou", to configure BIND (the zonefiles are slightly adapted versions of the zonefiles from [2]):

Note: We truncated some of the keys and default comments to improve the readability of the zonefiles.

```
include "/etc/bind/zones.strongSwan";
```

Listing 9: Content of the "named.conf.local" BIND configuration file

```
options {
    directory "/var/cache/bind";
    auth-nxdomain no;    # conform to RFC1035
    listen-on-v6 { any; };

    // Turn DNSSEC on
    dnssec-enable yes;
    dnssec-validation yes;
};
```

Listing 10: Content of the "named.conf.options" BIND configuration file

```
// Zones for the strongSwan UML tests
zone "local." {
    type master;
    file "/etc/bind/strongSwanZones/local/db.local.signed";
};

zone "moon.local." {
    type master;
    file "/etc/bind/strongSwanZones/moon.local/db.moon.local.signed";
};

zone "sun.local." {
    type master;
    file "/etc/bind/strongSwanZones/sun.local/db.sun.local.signed";
};
```

Listing 11: Content of the file "zones.strongSwan"
IPSECKEY based Authentication for strongSwan using DNSSEC

Listing 12: Zonefile "strongswanzone/local/db.local"

The following zone contains the IPSECKEY RR with the IPSECKEY of the VPN gateway "gateway.moon.local". Because strongSwan currently does not contain a tool to generate keys in the Base64-encoded RFC 3110 DNSKEY format (which is one of the key formats which is supported by IPSECKEY RRs [3]), we used the already generated keys from [21] as IPSECKEYs:

; Zonefile for the local zone
;
$TTL 604800
@  IN  SOA  ns1.local. root.local. ( 
  1 ;Serial
  604800 ;Refresh
  86400 ;Retry
  2419200 ;Expire
  604800 ) ;Negative Cache TTL
;
; Name servers of this zone
@  IN  NS  ns1.local.
ns1  IN  A  192.168.0.150
;
; Delegation of subdomain "moon"
moon  IN  NS  ns1.moon.local.
nsl.moon  IN  A  192.168.0.150
;
; Delegation Signer RR for the subdomain "moon"
moon.local.  IN  DS  17476 5 1 F56C3074E8BC17CC74454F215A825A745B812D38
moon.local.  IN  DS  17476 5 2 336F3A564A75F7663616F5DC3ECA7D4406F73C899CAF3265032FE1F6 7F3225A9
;
; Delegation of subdomain "sun"
sun  IN  NS  ns1.sun.local.
ns1.sun  IN  A  192.168.0.150
;
; Delegation Signer RR for the subdomain "sun"
sun.local.  IN  DS  15466 5 1 B8E6BF6FF99B3D5AA9226A5F7AE793F613491618
sun.local.  IN  DS  15466 5 2 D3075F94EBB5490580658F6F0F4CDB40B4DA1D1B1540CFA3CDE661 D6E6573C
;
; This is a zone-signing key, keyid 28338, for local.
local.  IN  DNSKEY  256 3 5 AwEAAcuTF68pvPAO/W9P31P5hkrvWSPV9sgb6TBYYj+6+oZ/oami71
; This is a key-signing key, keyid 34354, for local.
local.  IN  DNSKEY  257 3 5 AwEAAcH3CttFrKw5r569N+37R1rXzCj5QMri2+x+jhdeFuj/cgv218gMH

The following zone contains the IPSECKEY RR with the IPSECKEY of the VPN gateway "gateway.moon.local". Because strongSwan currently does not contain a tool to generate keys in the Base64-encoded RFC 3110 DNSKEY format (which is one of the key formats which is supported by IPSECKEY RRs [3]), we used the already generated keys from [21] as IPSECKEYs:
The following zone contains the IPSECKEY RR with the IPSECKEY of the VPN gateway "gateway.sun.local". Because strongSwan currently does not contain a tool to generate keys in the Base64-encoded RFC 3110 DNSKEY format (which is one of the key formats which is supported by IPSECKEY RRs [3]), we used the already generated keys from [21] as IPSECKEYs:

```
Listing 13: Zonefile "strongSwanZones/moon.local/db.moon.local"
```

```plaintext
@ IN NS ns1.moon.local.
nsl IN A 192.168.0.150 ;
; Entries for the VPN Gateway "gateway.moon.local"
gateway IN A 192.168.0.1 192.168.0.1 IN PTR gateway.moon.local.
gateway IN IPSECKEY ( 10 1 2 192.168.0.1
AQN+mkeECF5Bm7XnDkkfmgny/TZndTkN1XzFZWV7nJroM3cTk3zMtdSPX8hY9GQxVGWSsmUBq7mGA5Qx39JpRNpzyxW7wRcMbwqDquG1FRfb1LzV1ixdXQGSLUNAxOnqDI/h5fckqTuZlE4q3F4PmQAwzWVWaTZQ1gXXqUgKI1N6218Hm2vbvNRE/CBHUFmaCzl1jckvavPCqBLZsRTx9b/Mi+qD6xt7K9RpYHmtaGCJ95edIbY65ZkspgHwU88/3N66xZdD0kJOA3oFbw1kHkFyaGWFB2+fc7L6fYq0wr/d84tQdOXEn3BwLTrVKo7+6AxDrMI0I+b1D2nd9cxj
)
; This is a key-signing key, keyid 17476, for moon.local.
moon.local. IN DNSKEY 257 3 5 AwEAAcJCn1hpu3RcBUOC2hUv0TPqPeauOuTF9WlWicamM42U8TGJNvs
; This is a zone-signing key, keyid 58377, for moon.local.
moon.local. IN DNSKEY 256 3 5 AwEAAc2PmR7pj99X1P02cIgZgaHFTvqsuDI1JVjR7QBp5prUmaeRmPX
```

```plaintext
The following zone contains the IPSECKEY RR with the IPSECKEY of the VPN gateway "gateway.sun.local". Because strongSwan currently does not contain a tool to generate keys in the Base64-encoded RFC 3110 DNSKEY format (which is one of the key formats which is supported by IPSECKEY RRs [3]), we used the already generated keys from [21] as IPSECKEYs:

```
Listing 13: Zonefile "strongSwanZones/moon.local/db.moon.local"
```

```plaintext
@ IN NS ns1.moon.local.
nsl IN A 192.168.0.150 ;
; Entries for the VPN Gateway "gateway.moon.local"
gateway IN A 192.168.0.1 192.168.0.1 IN PTR gateway.moon.local.
gateway IN IPSECKEY ( 10 1 2 192.168.0.1
AQN+mkeECF5Bm7XnDkkfmgny/TZndTkN1XzFZWV7nJroM3cTk3zMtdSPX8hY9GQxVGWSsmUBq7mGA5Qx39JpRNpzyxW7wRcMbwqDquG1FRfb1LzV1ixdXQGSLUNAxOnqDI/h5fckqTuZlE4q3F4PmQAwzWVWaTZQ1gXXqUgKI1N6218Hm2vbvNRE/CBHUFmaCzl1jckvavPCqBLZsRTx9b/Mi+qD6xt7K9RpYHmtaGCJ95edIbY65ZkspgHwU88/3N66xZdD0kJOA3oFbw1kHkFyaGWFB2+fc7L6fYq0wr/d84tQdOXEn3BwLTrVKo7+6AxDrMI0I+b1D2nd9cxj
)
; This is a key-signing key, keyid 17476, for moon.local.
moon.local. IN DNSKEY 257 3 5 AwEAAcJCn1hpu3RcBUOC2hUv0TPqPeauOuTF9WlWicamM42U8TGJNvs
; This is a zone-signing key, keyid 58377, for moon.local.
moon.local. IN DNSKEY 256 3 5 AwEAAc2PmR7pj99X1P02cIgZgaHFTvqsuDI1JVjR7QBp5prUmaeRmPX
```
IPSECKEY based Authentication for strongSwan using DNSSEC

Listing 14: Zonefile "strongSwanZones/sun.local/db.sun.local"

Before the zonefiles can be used by BIND, they have to be signed. We therefore signed the zonefiles with the command “dnssec-signzone” (for more details on the signing of a zonefile we refer to [2]).

6.1.1.2 Configuration of “gateway.moon.local”

On the machine “gateway.moon.local” we configured strongSwan as follows:

We have loaded the pubkey, unbound and ipseckey plug-in and enabled the IPSECKEY plug-in in the file “strongswan.conf”:

Listing 15: Contents of "strongswan.conf"

We created a connection called “net-net” which represents the VPN of our test environment:

Listing 16: Contents of "ipsec.conf"

By using the value “@gateway.sun.local” as “rightid” we told strongSwan that it should use the IPSECKEY-AUTH method to authenticate the other VPN gateway and that the public key of the other VPN gateway is available as IPSECKEY under the name “gateway.sun.local” in
the DNS. The “leftauth=pubkey” and “leftid=@gateway.moon.local” entries allow the other
gateway to authenticate the “gateway.moon.local” gateway through the IPSECKEY-AUTH
method. With the option “leftrsasigkey=moonPub.der” we defined that strongSwan can find
the local public key (the public key of “gateway.moon.local”) in the file “moonPub.der”.

In the file “ipsec.secrets” we specified were strongSwan can find the local private key:

```bash
# /etc/ipsec.secrets - strongSwan IPsec secrets file
: RSA moonKey.der
```

**Listing 17: Contents of "ipsec.secrets"**

We stored the private key in the file “/etc/ipsec.d/private/moonKey.der” and the corresponding
public key in “/etc/ipsec.d/public/certs/moonPub.der”.

### 6.1.1.3 Configuration of “gateway.sun.local”
On the machine “gateway.moon.local” we configured strongSwan as follows:

We have loaded the pubkey, unbound and ipseckey plug-in and enabled the IPSECKEY plug-
in in the file “strongswan.conf”:

```bash
# /etc/strongswan.conf - strongSwan configuration file
charon {
  load = sha1 sha2 md5 aes des hmac gmp dnskey pem pkcs1 pubkey
  random curl kernel-netlink socket-default stroke updown unbound
  ipseckey
    plugins {
      ipseckey {
        enable = yes
      }
    }
}
```

**Listing 18: Contents of "strongswan.conf"**

We created a connection called “net-net” which represents the VPN of our test environment:

```bash
# /etc/ipsec.conf - strongSwan IPsec configuration file
config setup
  plutostart=no
conn %default
  ike lifetime=60m
  keylife=20m
  rekey margin=3m
  keying tries=1
  key exchange=ikev2
conn net-net
  left=192.168.0.2
  left subnet=10.2.0.0/16
  left id=@gateway.sun.local
  leftrsasigkey=sunPub.der
  left firewall=yes
  right=192.168.0.1
  right_subnet=10.1.0.0/16
  right id=@gateway.moon.local
```

R. Guadagnini
auto=add

Listing 19: Contents of "ipsec.conf"

By using the value "@gateway.moon.local" as "rightid" we told strongSwan that it should use the IPSECKEY-AUTH method to authenticate the other VPN gateway and that the public key of the other VPN gateway is available as IPSECKEY under the name "gateway.moon.local" in the DNS. The "leftauth=pubkey" and "leftid=@gateway.sun.local" entries allow the other gateway to authenticate the "gateway.sun.local" gateway through the IPSECKEY-AUTH method. With the option "leftrsasigkey=sunPub.der" we defined that strongSwan can find the local public key (the public key of "gateway.sun.local") in the file "sunPub.der".

In the file "ipsec.secrets" we specified were strongSwan can find the local private key:

```
# /etc/ipsec.secrets - strongSwan IPsec secrets file
: RSA sunKey.der
```

Listing 20: Contents of "ipsec.secrets"

We stored the private key in the file “/etc/ipsec.d/private/sunKey.der” and the corresponding public key in “/etc/ipsec.d/public/certs/sunPub.der”.

6.1.1.4 Configuration of “Alice” and “Bob”

Alice and Bob were already configured for our needs through the Ruby script from Listing 6.

6.1.2 Start of the test environment

To start our DUMM test environment and its virtual machines we wrote another Ruby script:

```
# Create Bridges
br0 = Bridge.new("br0")
br1 = Bridge.new("br1")
br2 = Bridge.new("br2")

# Get the existing VMs
alice = Guest["alice"]
bob = Guest["bob"]
moon = Guest["moon"]
sun = Guest["sun"]
winnetou = Guest["winnetou"]

# Start VMs
alice.start
bob.start
moon.start
sun.start
winnetou.start

# Set hostname
alice.exec("echo alice > /etc/hostname")
bob.exec("echo bob > /etc/hostname")
moon.exec("echo moon > /etc/hostname")
sun.exec("echo sun > /etc/hostname")
winnetou.exec("echo winnetou > /etc/hostname")

# Attach the network interfaces
alice.add("eth0").connect(br1).add("10.1.0.10")
```
bob.add("eth0").connect(br2).add("10.2.0.10")
moon.add("eth1").connect(br1).add("10.1.0.1")
moon.add("eth0").connect(br0).add("192.168.0.1")
sun.add("eth0").connect(br0).add("192.168.0.2")
sun.add("eth1").connect(br2).add("10.2.0.1")
winnetou.add("eth0").connect(br0).add("192.168.0.150")

Listing 21: Ruby script ”swanstart.rb”

“swanstart.rb” starts the virtual machines of our test scenario, creates virtual bridges which
connect the virtual machines and configures the network interfaces of the virtual machines.
The test environment is started with the following commands:

```
audo -s
ipsec irdumm swanstart.rb
```

6.2 Tests

Based on our test scenario we executed four tests to test our implementation of the
IPSECKEY-AUTH method. We will describe these tests and their results in this chapter.

6.2.1 Successful authentication with IPSECKEYs

To test if the IPSECKEY-AUTH implementation works when all components are running and
configured properly we have started the VPN connection, which we have configured in our
test environment, as follows:

We started first the test environment with its virtual machines:

```
audo -s
ipsec irdumm swanstart.rb
```

Next we logged in on the virtual machines “gateway.moon.local” and “gateway.sun.local” and
started strongSwan on both of them by executing the following command on both machines:

```
ipsec start
```
Finally we executed the command

\texttt{ipsec up net-net}

on the VPN gateway “gateway.moon.local” to set up the VPN.

The resulting log indicates that the VPN was successfully established between the VPN gateways by using the IPSECKEY RRs from the DNS for the authentication of the VPN gateways:

\begin{verbatim}
initiating IKE_SA net-net[1] to 192.168.0.2
generating IKE_SA_INIT request 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) ]
sending packet: from 192.168.0.1[500] to 192.168.0.2[500]
received packet: from 192.168.0.2[500] to 192.168.0.1[500]
parsed IKE_SA_INIT response 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) N(MULT_AUTH) ]
authentication of 'gateway.moon.local' (myself) with RSA signature successful
establishing CHILD_SA net-net

generating IKE_AUTH request 1 [ IDi N(INIT_CONTACT) IDr AUTH SA TSi TSr N(MOBIKE_SUP) N(ADD 4_ADDR) N(MULT_AUTH) N(SAF ONLY) ]
sending packet: from 192.168.0.1[4500] to 192.168.0.2[4500]
received packet: from 192.168.0.2[4500] to 192.168.0.1[4500]
parsed IKE_AUTH response 1 [ IDr AUTH SA TSi TSr N(AUTH_LFT) N(MOBIKE_SUP) N(ADD_4_ADDR) ]
ipseckey_cred: Performing a DNS query for the IPSECKEY RRs of the domain gateway.sun.local
ipseckey_cred: Enumerating over IPSECKEY certificates using trusted certificate "gateway.sun.local"
authentication of 'gateway.sun.local' with RSA signature successful
IKE_SA net-net[1] established between 192.168.0.1[gateway.moon.local][192.168.0.2[gateway.sun.local]
scheduling reauthentication in 3273s
maximum IKE_SA lifetime 3453s
\end{verbatim}

**Listing 22: Log of a successfully established VPN with the IPSECKEY-AUTH method**

The log entries in red show that the IPSECKEY plug-in queried the DNS with the help of the DNS resolver plug-in to retrieve the IPSECKEY RRs of the other VPN gateway. These IPSECKEY RRs are then used to authenticate the other VPN gateway. The authentication process is completed successfully and the VPN is established.

To test if the VPN connection really works we pinged from the client “Alice” the Client “Bob” by executing the ping command on Alice:

\texttt{ping 10.2.0.10}

The ping was successful, which means that the VPN was successfully established with the help of the IPSECKEY-AUTH method. While we were executing the ping, we used Wireshark to capture traffic on the bridge “br0”. The capture showed that the ping packets were transported through the VPN tunnel, which also shows that the VPN was really established.

\rightarrow \, The IPSECKEY-AUTH method works as expected.

### 6.2.2 Retrieve the local public key from the DNS

If public key based authentication is used, the strongSwan VPN gateways have to know their own public key. In our test scenario we stored the public keys on the VPN gateways in the directory “/etc/ipsec.d/public/certs”. With our implementation of the IPSECKEY-AUTH method the VPN gateways are able to retrieve their own public key also as IPSECKEY RR from the
DNS.

To achieve that strongSwan tries to retrieve the own public key from the DNS, we have to remove the “leftrrsasigkey” entry from his config file “ipsec.conf”. We therefore removed the entry

```
leftrrsasigkey=moonPub.der
```

from the “ipsec.conf” file of the machine “gateway.moon.local” and the entry

```
leftrrsasigkey=sunPub.der
```

from the “ipsec.conf” file of the machine “gateway.sun.local”.

To test the behavior of our implementation of the IPSECKEY-AUTH method in this case we then started our virtual test environment:

```
sudo -s
ipsec irdumm swanstart.rb
```

Then we logged in on the virtual machines “gateway.moon.local” and “gateway.sun.local” and started strongSwan on both of them by executing the following command on both machines:

```
ipsec start
```

Finally we executed the command

```
ipsec up net-net
```

on the VPN gateway “gateway.moon.local” to set up the VPN.

The resulting log indicates that the VPN was successfully established between the VPN gateways:

```
initiating IKE_SA net-net[1] to 192.168.0.2
  generating  IKE_SA_INIT request 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) ]
  sending packet: from 192.168.0.1[500] to 192.168.0.2[500]
  parsed  IKE_SA_INIT response 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) N(MULT_AUTH) ]
  ipseckey_cred: Performing a DNS query for the IPSECKEY RRs of the domain gateway.moon.local
  ipseckey_cred: Enumerating over IPSECKEY certificates
  authentication of 'gateway.moon.local' (myself) with RSA signature successful
  establishing CHILD SA net-net
  generating IKE AUTH request 1 [ IDi N(INIT_CONTACT) IDr AUTH SA TSi TSr N(MOBIKE_SUP) N(ADD 4 ADDR) N(MULT AUTH) N(EAP_ONLY) ]
  sending packet: from 192.168.0.1[4500] to 192.168.0.2[4500]
  received packet: from 192.168.0.2[4500] to 192.168.0.1[4500]
  parsed  IKE_AUTH response 1 [ IDr AUTH SA TSi TSr N(AUTH_LFT) N(MOBIKE_SUP) N(ADD 4 ADDR) ]
  ipseckey_cred: Performing a DNS query for the IPSECKEY RRs of the domain gateway.sun.local
  ipseckey_cred: Enumerating over IPSECKEY certificates
  using trusted certificate "gateway.sun.local"
  authentication of 'gateway.sun.local' with RSA signature successful
  IKE_SA net-net[1] established between 192.168.0.1[gateway.moon.local]...192.168.0.2[gateway.sun.local]
  scheduling reauthentication in 3241s
  maximum IKE_SA lifetime 3421s
```

Listing 23: Log which shows how the local public key was retrieved from the DNS
The log shows, that the VPN was successfully established. We can also see from the log, that both the local public key (green) and the public key of the other VPN gateway (red) were retrieved as IPSECKEYs from the DNS.

To test if the VPN connection really works we pinged from the client “Alice” the Client “Bob” by executing the ping command on Alice:

```
ping 10.2.0.10
```

The ping was successful, which means that the VPN was successfully established.

➔ With our implementation of the IPSECKEY-AUTH method strongSwan VPN gateways are able to retrieve their own public key as IPSECKEY from the DNS.

**Note:** For performance reasons we do not recommend the configuration used in this test. With this configuration each VPN gateway has to query the DNS for two sets of IPSECKEYs, its own (containing its own public key) and the set of the other VPN gateway. Querying the DNS is an expensive operation; we therefore recommend to use the configuration from chapter 6.2.1 which causes that each VPN gateway reads its own public key from its local file system. Reading from the local file system is much cheaper than performing a DNS query.

### 6.2.3 IPSECKEY RRSIG signature expired

An IPSECKEY has a limited lifetime (see chapter 3.2.2 for a definition of its lifetime). With this test we have tested how our implementation of the IPSECKEY-AUTH method behaves in the case of an expired IPSECKEY. We therefore re-signed the zonefiles of our test scenario with a short signature lifetime and waited until the signatures had expired.

We then started our test environment through:

```
sudo -s
ipsec irdumm swanstart.rb
```

Then we logged in on the virtual machines “gateway.moon.local” and “gateway.sun.local” and started strongSwan on both of them by executing the following command on both machines:

```
ipsec start
```

Finally we executed the command

```
ipsec up net-net
```

on the VPN gateway “gateway.moon.local” to set up the VPN.
The resulting log looks as follows:

initiating IKE_SA net-net[2] to 192.168.0.1
generating IKE_SA_INIT request 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) ]
sending packet: from 192.168.0.2[500] to 192.168.0.1[500]
received packet: from 192.168.0.1[500] to 192.168.0.2[500]
parsed IKE_SA_INIT response 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) N(MULT_AUTH) ]
ipseckey_cred: Performing a DNS query for the IPSECKEY RRs of the domain gateway.sun.local
ipseckey_cred: DNSSEC security state of the IPSECKEY RRs of the domain gateway.sun.local is not SECURE as required
ipseckey_cred: Performing a DNS query for the IPSECKEY RRs of the domain gateway.sun.local
ipseckey_cred: DNSSEC security state of the IPSECKEY RRs of the domain gateway.sun.local is not SECURE as required
no private key found for 'gateway.sun.local'

Listing 24: Log which results if the used IPSECKEYs are expired

As we can see from the log, strongSwan does not establish the VPN because it does not regard the retrieved IPSECKEYs as secure (red entries in Listing 24).

 ➔ Our implementation of the IPSECKEY-AUTH method detects expired IPSECKEYs and declines to use them for authentication. Thus it behaves as expected.

### 6.2.4 DNS server not available

With this test case we have tested the behavior of our implementation of the IPSECKEY-AUTH method in the case in which the DNS server with the IPSECKEYs is not available.

First we have started our test environment through:

```
sudo -s
ipse -r dummy swanstart.rb
```

Then we have logged in on the machine “Winnetou” and stopped the DNS server by executing the command:

```
service bind9 stop
```

We started strongSwan on the virtual machines “gateway.moon.local” and “gateway.sun.local” by executing the following command on both of them:

```
ipsec start
```

Finally we executed the command

```
ipsec up net-net
```
on the VPN gateway “gateway.moon.local” to set up the VPN.
The resulting log file looks as follows:

```plaintext
initiating IKE_SA net-net[1] to 192.168.0.2
generating IKE_SA_INIT request 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) ]
sending packet: from 192.168.0.1[500] to 192.168.0.2[500]
received packet: from 192.168.0.2[500] to 192.168.0.1[500]
parsed IKE_SA_INIT response 0 [ SA KE No N(NATD_S_IP) N(NATD_D_IP) N(MULT_AUT) ]
authentication of 'gateway.moon.local' (myself) with RSA signature successful
establishing CHILD SA net-net
generating IKE_AUTH request 1 [ IDi N(INIT_CONTACT) IDr AUTH SA TSi TSr N(MOBIKE_SUP) N(ADD_4_ADDR) N(MULT_AUTH) N(EAP_ONLY) ]
sending packet: from 192.168.0.1[4500] to 192.168.0.2[4500]
retransmit 1 of request with message ID 1
sending packet: from 192.168.0.1[4500] to 192.168.0.2[4500]
retransmit 2 of request with message ID 1
sending packet: from 192.168.0.1[4500] to 192.168.0.2[4500]
received packet: from 192.168.0.2[4500] to 192.168.0.1[4500]
parsed IKE_AUTH response 1 [ N(AUTH_FAILED) ]
received AUTHENTICATION_FAILED notify error
```

Listing 25: Log file in the case if no DNS server is present

The log file shows, that strongSwan was not able to establish the VPN because the authentication process failed. We expected this behavior, because strongSwan cannot retrieve the public keys (IPSECKEYs) for the authentication if no DNS server is present.

- As expected strongSwan is not able to build up the VPN with the help of the IPSECKEY-AUTH method if no DNS server is present. Because of the DNS timeouts and the retransmits it takes quite a long time until strongSwan recognizes that it is not possible to establish the VPN.
7 Open questions/points

Because of time and resource constraints we were not able to cover all questions during this project. We therefore list the open questions/points here, so that they can be addressed in the future:

- Currently strongSwan does not contain a tool to create public keys in the Base64-encoded RFC 3110 DNSKEY format, which is required to store the RSA public keys as IPSECKEYs in the DNS. Such a tool has to be implemented in the future.

- We only tested our implementation of the IPSECKEY-AUTH method in a virtual test scenario. Further tests with “real” DNS servers, VPN gateways and clients have to be conducted.

- Because of our limited number of tests and the virtual test scenario that we have used to conduct them, we are not able to answer questions about the performance of the IPSECKEY-AUTH method. Further tests in this direction should be performed in the future.

- We have not performed any field tests with the IPSECKEY-AUTH method yet. We therefore recommend to mark the “IPSECKEY” plug-in, which implements the IPSECKEY-AUTH method, as experimental until these tests have been successfully conducted. The “unbound” DNS resolver plug-in which we have also developed during this project can be used without constraints.

- We implemented the IPSECKEY-AUTH method in a separate branch of strongSwan which bases on the strongSwan version 4.6.3. This branch has to be merged with the official strongSwan branch.

- The directory “strongswan/src/dnssec” contains an application that we have used to test the implementation of the DNS resolver plug-in. We left this application there for documentation purposes, because it is part of the work that we have done during this project. The application should be removed before our branch of strongSwan is merged with the official branch.
8 Conclusion

During the project we were able to implement the proposed IPSECKEY-AUTH method for strongSwan. The first tests with the IPSECKEY-AUTH method that we have performed in our virtual test environment look promisingly so we are optimistic that the method should be applicable in the field. One of the biggest problems that might appear in the field is a lack of performance because of the long time that it takes to query the DNS for the IPSECKEYs and to validate them through DNSSEC. With the implementation of a caching resolver plug-in we tried to address this problem. Future field tests have to show if this measure is sufficient or whether further measures are necessary.

The modular architecture of strongSwan and the use of C in an object oriented style as programming language for strongSwan proved very useful during our project. It allowed us to integrate the IPSECKEY-AUTH method into strongSwan with only minimal changes to the rest of the system. The use of C in an object oriented style has from our point of view one drawback: There is no special support for this programming style in the common integrated development environments. So we were not able to use such nice features as automatic code completion, which we are used to from other programming languages and their development environments. strongSwans source code is well documented through the strongSwan wiki and the source code comments, but we missed a picture which shows the multi threaded architecture of strongSwan at one glimpse. We therefore suggest adding Figure 9 from this report with a short description to the strongSwan wiki. We think that this figure gives in conjunction with Figure 8 a good overview over the strongSwan architecture, which is very useful for someone who is new to the strongSwan project.
9 Sources


[16] Documentation of the libstrongswan:


[18] Documentation of the Charon daemon:

[19] Documentation strongSwan’s credential manager class:

[20] Documentation of DUMM:

[21] strongSwan test scenario which uses the “Pubkey” authentication method:

[22] Changelog of strongSwan 5.0:
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>DNSSEC</td>
<td>DNS Security Extensions</td>
</tr>
<tr>
<td>DS</td>
<td>Delegation Signer</td>
</tr>
<tr>
<td>DUMM</td>
<td>Dynamic UML Mesh Modeler</td>
</tr>
<tr>
<td>EAP</td>
<td>Extensible Authentication Protocol</td>
</tr>
<tr>
<td>FQDN</td>
<td>Fully Qualified Domain Name</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU's Not Unix</td>
</tr>
<tr>
<td>IANA</td>
<td>Internet Assigned Numbers Authority</td>
</tr>
<tr>
<td>ID</td>
<td>Identity</td>
</tr>
<tr>
<td>IKE</td>
<td>Internet Key Exchange</td>
</tr>
<tr>
<td>PGP</td>
<td>Pretty Good Privacy</td>
</tr>
<tr>
<td>PKI</td>
<td>Public Key Infrastructure</td>
</tr>
<tr>
<td>PSK</td>
<td>Pre-shared key</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comments</td>
</tr>
<tr>
<td>RR</td>
<td>Resource Record</td>
</tr>
<tr>
<td>RRSIG</td>
<td>Resource Record Signature</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest Shamir Adleman</td>
</tr>
<tr>
<td>SA</td>
<td>Security Association</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>User-Mode Linux</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
</tbody>
</table>
# 11.1 Projektplan

## Projektplan: MSE Projektarbeit 2

<table>
<thead>
<tr>
<th>Paket Nr.</th>
<th>Arbeitspaketname</th>
<th>Kalenderwoche</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Projektmanagement</td>
<td>8, 9, 11, 12, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29, 30</td>
</tr>
<tr>
<td>1</td>
<td>Bericht</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Analyse strongSwan</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DNS Resolver</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IPSECKEY plug-in</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Systemtest/Bugfixing</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Meilensteine

<table>
<thead>
<tr>
<th>Meilenstein</th>
<th>M1: Analyse der Architektur von strongSwan abgeschlossen</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M2: Implementierung des DNS Resolvers für strongSwan abgeschlossen und getestet</td>
</tr>
<tr>
<td></td>
<td>M3: Systemtest erfolgreich abgeschlossen</td>
</tr>
<tr>
<td></td>
<td>M4: Abgabe</td>
</tr>
</tbody>
</table>
11.1.1 Differences to the initial project plan

The main difference between the final project plan and our initial planning is the duration of the "DNS Resolver" task. Because we noticed during the implementation of the DNS resolver for strongSwan, that the ldns DNS resolver library, which we had chosen for strongSwan in [2], does not support caching we had to go a step back and look again for a suitable DNS resolver library (see also chapter 5.1.2.1). This and our lack of experience with the object oriented C style in which strongSwan is implemented caused that we had much longer to complete the "DNS Resolver" task than initially planned.

11.2 Listings

```c
#include <stdint.h>
#include <stdlib.h>
#include <ldns/ldns.h>

/* Determines the DNSKEY RRs for the domain "dname". */
ldns_rr_list *get_dnskey_rr_of_domain(ldns_resolver *resolver, char *dname) {
    ldns_rdf *domain = NULL;
    ldns_pkt *paket = NULL;
    ldns_rr_list *dnskey_rr = NULL;
    printf("Try to get the DNSKEY RRs for the domain: %s\n", dname);
    domain = ldns_dname_new_frm_str(dname);
    paket = ldns_resolver_query(resolver, domain, LDNS_RR_TYPE_DNSKEY,
                      LDNS_RR_CLASS_IN, LDNS_RD);
    if (!paket) {
        printf("No answer to the query.\n");
        ldns_rdf_deep_free(domain);
        return NULL;
    }
    dnskey_rr = ldns_pkt_rr_list_by_type(paket, LDNS_RR_TYPE_DNSKEY,
                          LDNS_SECTION_ANSWER);
    if (!dnskey_rr) {
        printf("Could'nt retrieve the DNSKEY RRs for the domain: %s\n", dname);
        ldns_rdf_deep_free(domain);
        ldns_pkt_free(paket);
        return NULL;
    }
    printf("Successfully retrieved the DNSKEY RRs for %s\n", dname);
    ldns_rdf deep_free(domain);
    ldns_pkt free(paket);
    return dnskey_rr;
}

/* Use the DNSKEYs of the domain/zone "ta name" as trust anchor for * the resolver. */
bool set_trust_anchor(ldns_resolver *resolver, char *ta_dname) {
    ldns_rr_list *trust_anchor_keys = NULL;
    ldns status status;
    const char *error_str;
    printf("Try to get the keys which form the trust anchor:\n");
    trust_anchor_keys = get_dnskey_rr_of_domain(resolver, ta_dname);
    if (!trust_anchor_keys) {
        printf("Could'nt retrieve the keys which form the trusted anchor:\n");
        return 0;
    }
    printf("Trust anchor keys retrieved.\n");
}
```
printf("Use one of them as trust anchor for the resolver.\n");
status = ldns resolver push dnssec anchor(resolver, 
ldns_rr_list_pop_rr(trust_anchor_keys));
if (status != LDNS_STATUS_OK) {
    error str = ldns get errorstr by id(status);
    printf("ldns error string: %s\n", error str);
    return 0;
}
return 1;
}
int main(int argc, char *argv[])
{
	/* URL to query */
	char *dname = "ns1.moon.local";
	ldns_rdf *domain = ldns_dname_new_frm_str(dname);
	/* URL of the trust anchor */
	char *ta_dname = "local";
	ldns_resolver *resolver = NULL;
	ldns_pkt *paket = NULL;
	ldns_rr *rr_set = NULL;
	ldns_rr_list *rrsig = NULL;
	ldns_status status;
	const char *error str;

	/* 1.) create a new resolver which queries the name servers 
	mentioned in /home/reto/ldns_test/resolv.conf */
	printf("1.) Creating a ldns resolver instance...\n");
	status = ldns_resolver_new_frm_file(&resolver, 
"/home/reto/ldns-test/resolv.conf");

	/* check if the resolver was created successfully */
	if (status != LDNS_STATUS_OK) {
	    printf("Could not create resolver\n");
	    ldns_rdf_deep_free(domain);
	    ldns_resolver_deep_free(resolver);
	    exit;
	}
	/* activate DNSSEC support */
	dns resolver set dnssec(resolver, 1);
	dns resolver set edns udp size(resolver, 4096);
	printf("ldns resolver created.\n");
	dname) {
	    printf("Failed to set the trust anchor of the resolver.\n");
	    ldns_rdf_deep_free(domain);
	    ldns_resolver_deep_free(resolver);
	    exit(EXIT FAILURE);
	}
	printf("Trust anchor set.\n");

	/* 3.) Perform a query and validate the resulting RR(set) */
	paket = ldns resolver query(resolver, 
domain, 
LDNS_RR_TYPE_A, 
LDNS_RR_CLASS IN, 
LDNS RD);

	/* handle the results of the query */
	if (!paket)
	    printf("Query failed: Did not receive answer paket.\n");
	    ldns_rdf_deep_free(domain);
	    ldns_resolver_deep_free(resolver);
	    exit(EXIT FAILURE);
	else {
	    /* read the A records from the answer section of the answer packet */
	rr_set = ldns_pkt_rr_list_by_type(paket, 
LDNS_RR_TYPE_A, 
LDNS_SECTION_ANSWER);
IPSECKEY based Authentication for strongSwan using DNSSEC

Listing 26: Trust chain validation with ldns

```c
rrsig = ldns pkt rr list by type(paket,
        LDNS_RR_TYPE_RRSIG,
        LDNS_SECTION_ANSWER);

/* check if the query was successful */
if (!rr_set || !rrsig) {
    printf("invalid answer to the query!\n");
    ldns rdf deep free(domain);
    ldns resolver deep free(resolver);
    ldns_pkt free(paket);
    ldns_rr_list_deep_free(rr_set);
    ldns rr_list deep free(rrsig);
    exit(EXIT_FAILURE);
}

ldns_rr_list_sort(rr_set);
ldns rr_list print(stdout, rr set);
printf("Trying to validate the RRset:\n");
status = ldns_verify_trusted(resolver, rr_set, rrsig, NULL);

if (status != LDNS STATUS_OK) {
    error_str = ldns_get_errorstr_by_id(status);
    printf("RRset is not valid!\n");
    printf("ldns error string: %s\n", error str);
    ldns rdf deep free(domain);
    ldns resolver deep free(resolver);
    ldns_pkt free(paket);
    ldns rr_list deep free(rr_set);
    ldns rr list deep free(rrsig);
    exit(EXIT_FAILURE);
}
printf("RRset is valid!\n");

/* clean up */
ldns rdf deep free(domain);
ldns resolver deep free(resolver);
ldns_pkt free(paket);
ldns_rr_list_deep_free(rr_set);
ldns rr_list deep free(rrsig);
exit(EXIT_SUCCESS);
```

R. Guadagnini
58