ARCHITECTURE OF AN IKEV2 PROTOCOL IMPLEMENTATION

Stjepan Groš and Vlado Glavinić
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3
10000 Zagreb, Croatia
email: {stjepan.gros, vlado.glavinic}@fer.hr

ABSTRACT
In this paper we describe the architecture of an IKEv2 protocol implementation. The architecture has been designed with the aim to make the implementation meet a number of good characteristics: to be as fast as possible, scalable, easy to understand and enhance, as well as portable across different operating systems and processor architectures. As a byproduct we endeavored to specify an architectural pattern that might be used to build other, similar network protocols. In order to make this pattern even more attractive, parts of the architecture are separated as a generic software framework, thus allowing reuse of the code base. The implementation itself is written in the C programming language but borrows some of the concepts from object oriented programming. More specifically, it both enforces the access to private data structures of the different modules and uses messages, i.e. objects, to communicate different events among subsystems.

KEY WORDS
architectural patterns, software frameworks, IKEv2 protocol, protocol implementation

1 Introduction

The implementation of any real-world network protocol is quite a demanding chore, especially if done from scratch with none or minimal previous experience with implementation tasks, since it requires the operationalization of a number of more advanced functionality elements like timers, communication with the kernel network subsystem, inter-entity message handling, and complex state machine implementation.

The Internet Key Exchange protocol, presently in its version 2 - IKEv2 [1], is a good example of a complex network protocol. It is a key exchange protocol used within the IPsec architecture [2] that also serves for authentication and authorization purposes. IKEv2 is meant to be used on clients, as well as on VPN gateways. Thus, it might work in quite a demanding environment supporting a large number of concurrent clients. There are different methods to assure that the required clients be able to "connect" without experiencing significant delays, what includes the use of cryptographic hardware accelerators or increasing the number of available VPN gateways and/or using multiprocessor gateways. Still, all this techniques might fall short of the given task if the implementation itself is of a poor quality.

In this paper we describe the architecture used to implement the IKEv2 protocol, as well as relevant implementation details. Our goal was to achieve a fast implementation capable of handling a number of clients without significant delays, and, in addition to this, to scale well to faster machines. Basing on the experiences achieved, we developed an architectural pattern that could be used to build similar high performance applications. In order to make this pattern more attractive, the implementation itself is subdivided into a reusable part, and the IKEv2 protocol specific part, the former representing a software framework to be used by other implementers.

Apart from our IKEv2 [3] implementation there are also several other related open source ones. The first of them is the OpenIKEv2 [4] implementation, written in the C++ programming language, and also providing a reusable framework, however for IKEv2 specific implementation only. Conversely, the racoon2 [5] is a more ambitious implementation, as it also implements both IKEv1 [6] and KINK [7] protocols, but hasn't got any reusable code. Finally, there is the OpenSwan [8], primarily being an IKEv1 protocol implementation, the IKEv2 protocol being a recent addition to this implementation. Both racoon2 and OpenSwan are written in the C++ programming language. On the other hand, ACE [9] is a software framework for building network applications, which in this respect has a similar purpose as our IKEv2 implementation; in comparison to our implementation it is more general, quite complex and written in the C++ programming language.

As of this writing, we are in the final stage of the IKEv2 implementation, with measurements of the implementation performance following immediately afterwards. In order to validate the reusability potential of both the architectural pattern and the framework, we reuse them in some other protocol implementations presently in their early development stage [10].

The remaining of the paper is structured as follows. In Section 2 we first provide a short overview of the IKEv2 protocol. Then, in Section 3 we describe the different aspects of the architecture, emphasizing the software framework and the subsystems specific to IKEv2. The very de-

tails of the implementation are described in Section 4. Finally, the conclusion and some hints for future work are given in Section 5.

2 IKEv2 Short Overview

Within the IPsec architecture, data traffic is protected before leaving host or gateway, and is validated and decrypted on reception. The protection is done by using *security associations* (SAs), also called Child SA. Each SA defines both a cryptographic algorithm and the keys to use for data flow protection in a single direction. Thus, to protect a bidirectional traffic, two SAs are necessary. Though SAs can be managed manually this is an error prone process, which misses some features like rekeying and authorization, therefore the IKEv2 protocol has been developed with the specific purpose of establishing and managing SAs. The IKEv2 protocol is a request/response protocol working on top of the UDP protocol. Each IKEv2 protocol entity sends and receives request and response messages, which are composed of a header and one or more payloads, depending on the particular message in question. A request message and its response is denoted as an *exchange*. When an SA has to be established, a IKEv2 daemon (termed Initiator), sends the request to the other IKEv2 daemon (termed Responder). The first exchange has the purpose of establishing an IKE SA that protects all further communication between those two IKEv2 protocol implementations. This first exchange is the only one plaintext, while all other messages, apart from the header, are encrypted. In the second exchange, the two IKEv2 daemons authenticate each other and establish two SAs. From that point on, these two IKEv2 daemons can establish additional Child SAs, rekey and delete old ones, etc. To finish any further communication IKEv2 daemons delete the IKE SA, which also deletes any associated Child SAs.

The way the IKEv2 protocol implementation works is depicted in Figure 1. A client application executes on the left host, trying to communicate with a server application on the right one. The applied security policy specifies that this communication has to be encrypted and integrity protected (the exact way of specifying this policy being, in this simple use case, outside of the scope of the IKEv2 protocol implementation). It is assumed however that the policy is specified by the system administrator or the user, and stored in the kernel's *Security Policy Database* (SPD).

When the client application sends the first packet to the server application (step 1), the kernel checks the SPD and concludes that this traffic should be protected. SPD instructs the kernel *what* to protect, while *how* to protect is specified in a separate database named *Security Association Database* (SAD), again within the kernel. Because there was no previous communication, hence SAD is empty, the kernel contacts the IKEv2 protocol implementation (step 2) – or a *daemon* in Unix parlance. The IKEv2 daemon (*Initiator*) contacts its counterpart on the destination host (*Responder*); these daemons exchange messages (step 3) in

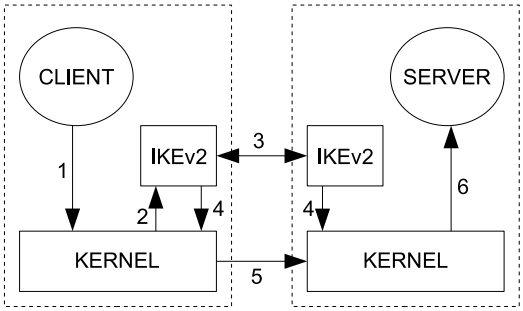

Figure 1. Basic use case of IKEv2 protocol implementation

order to authenticate and authorize each other, and to negotiate the cryptographic algorithms and keys, the negotiated parameters being stored into SAD (step 4). Now, the cryptographically protected traffic starts to flow from the left to the right host (step 5), and is eventually delivered as plaintext to the server application. The traffic in the other direction is also protected, but does not involve IKEv2 daemons any more. Note that step 3, though shown to flow directly between IKEv2 daemons, is in reality flowing through the kernel; it is however not protected by the kernel, but by the two IKEv2 daemons themselves.

3 The Architecture

The core of the IKEv2 protocol implementation is based on state machines. While realizing their behavior, they are nevertheless isolated from the specifics of the "wire format" of different IKEv2 messages as well as from the interface to in-kernel IPsec related databases.

Figure 2 illustrates the *static* view of the architecture of the IKEv2 protocol implementation. Shaded boxes belong to the framework while the unshaded ones are specific to IKEv2, hence they are treated separately. We denote each element in the figure a *subsystem*, thus we have a networking subsystem, a message subsystem, etc. The asynchronous communication among different subsystems is implemented via asynchronous queues, and in some cases, via a callback mechanism. In case queues are used for communication, the subsystem that waits for messages has its own thread.

As it can be seen from the figure, there are multiple inputs into the IKEv2 application. All the messages received from the network are captured by the network subsystem, which subsequently encapsulates the received data into the `network_msg` structure and after some interlayer processing sends them to the appropriate upper layer subsystem. In other words, if the received message is (i) an IKEv2 message, it is forwarded to the message subsystem, if it is (ii) a RADIUS message, it is forwarded to the respective RADIUS subsystem, while if it is (iii) a DHCP related message, it is forwarded to the CFG one. Upon receiving the message from the networking subsystem, each of

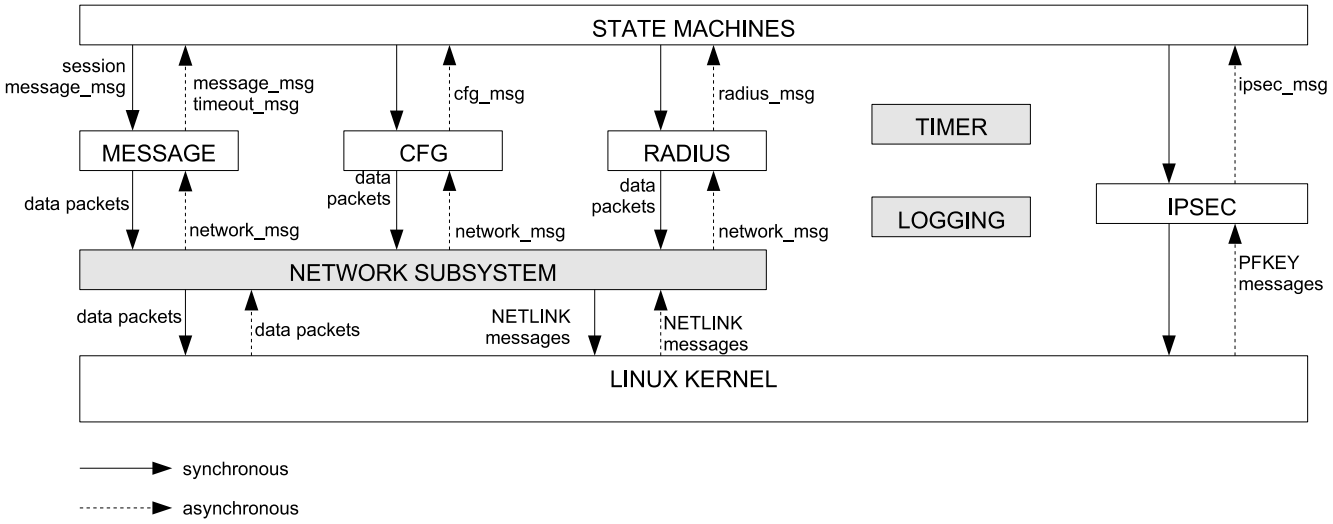

Figure 2. Static architecture view of IKEv2 protocol implementation

the aforementioned subsystems does the necessary decoding and creates its own message format that is forwarded to the respective state machines.

The other input to the IKEv2 protocol implementation comes from the IPsec databases in the kernel (SPD and SAD). The IPsec subsystem of the IKEv2 daemon communicates with the kernel through the PF_KEY interface described in [11]. This interface is underspecified hence Linux (as well as other operating systems) introduced a number of extensions, here including SPD management and NAT traversal. Additionally, as NETLINK is Linux's native interface to the in-kernel network subsystem, the IKEv2 daemon has been planned to use either PF_KEY or NETLINK's NETLINK_XFRM, whichever be available and/or most convenient.

Not all the components that make IKEv2 implementation are shown in the static architecture view, the reason being that many of them are only passive in nature. However, the more important ones will be also mentioned later in this section.

The dynamic view of the IKEv2 architecture is shown in Figure 3. By a "dynamic" architecture we understand both threads (represented by circles in Figure 3) and communication channels between them. There exist two thread types: those constantly running throughout IKEv2's lifetime (and denoted with a `thread` suffix), and those invoked by another thread (denoted with a `_cb` suffix). Communication queues are represented by squares in Figure 3. The association of threads (and coupled queues if there are any) to some specific subsystem is denoted by suitable prefixes (e.g. `message_`, `sm_`).

From the dynamic architecture view it can be seen that the input network traffic is handled by the `network_cb` thread, which, after processing the received data and creating the `network_msg` structure, forwards messages into the appropriate receiving subsystem queue.

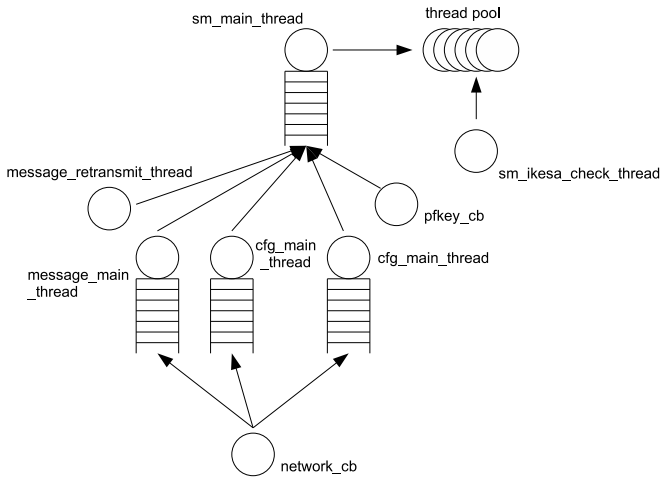

Figure 3. Dynamic architecture view of IKEv2 protocol implementation

There, the structure is taken over, processed, and a new message sent to the state machine queue to undergo a short preprocessing step (searching for the appropriate data structure describing the peer to whom the message belongs), and is eventually pushed into the thread pool. The thread pool does the main processing of requests, generates the appropriate responses, and processes those received. Since the load on the IKEv2 daemon (as is anyway the case with every network application) depends on the environment where it is used, the number of working threads in the thread pool should be configurable.

The description of IKEv2 architecture components is detailed in the remainder of this section.

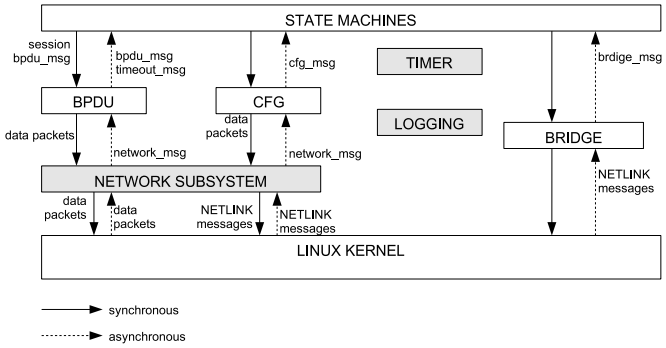

Figure 4. Static architecture view of RSTP protocol implementation

3.1 The Framework

The framework is the reusable IKEv2 architecture part, which consists of networking, timer and logging subsystems. Additionally, it also includes netlib, config, and crypto modules, not shown in Figure 2.

The purpose of the netlib module is to hide differences between different socket addresses, but also to allow manipulation of network addresses. The main structure for that purpose is `netaddr`, as this functionality is hard to achieve with the standard socket structures. One of the salient features of `netaddr` is reference counting in order to save memory and increase efficiency when handling duplicate addresses. The config module is used to read the configuration data necessary to adjust the behavior of the network application (i.e. IKEv2). At the current stage of development, it is intended to be further enhanced by introducing a higher degree of modularization in order to be easily extensible without need to substantially rewrite the parser each time it is reused. Finally, the crypto module contains frequently used cryptographic (e.g. hashing, encryption) functions, and also supports certificate functions. As of this writing, certificates are specific to IKEv2 requirements thus they are not yet treated as a part of the framework.

The networking subsystem offers the registration function, which opens a socket (either anonymous, serving for sending data only, or bound to a specific port/address). An important registration function parameter is the queue to which the received data is sent after being appropriately preprocessed. Preprocessing includes obtaining network level data, like source and destination IP addresses. Registered sockets can be closed at any time during the networking subsystem lifetime. Users of the networking subsystem are also offered an appropriate function to send data, respective parameters encompassing destination address embedded in the `netaddr` structure. The send function is synchronous, as it is not expected to block. In order to send data it is not necessary to previously register a socket because one will be created dynamically. The networking subsystem currently supports UDP and link layer sockets with planned extensions to both SCTP and TCP protocols.

Apart from sending and receiving data, the networking subsystem also monitors both network interfaces and IP addresses. This is necessary so that the application can have better control over data it sends or receives, but also because some applications require knowledge of available IP addresses and interfaces. Monitoring will be further expanded in order for network subsystem to be able to asynchronously notify upper subsystems of possible dynamic changes.

Similar to the networking subsystem, the timer one offers functions to register and cancel timeouts. Timeouts can be one-shot, or recurring. During timeout registration two ways to notify timeout event are offered: (i) via queues, and (ii) callback functions which are allocated a separate thread. It should be noted that in case of callback functions, to avoid congestion, processing should be non-blocking and fast.

The generic architecture and the framework can be applied to other network protocols implementations. E.g., one that is in the early development stage is the RSTP protocol [12]. Its architecture, which is based on the aforementioned framework and uses similar concepts as in the IKEv2 implementation, is shown in Figure 4. Naturally, as this is a completely different protocol, showing a different behavior, it possesses different state machines.

The BPDU subsystem takes care of creating (coding) and interpreting (decoding) RSTP specific messages. Since RSTP manipulates interfaces, and not IPsec databases, this part of the architecture is also changed. The CFG subsystem offers the possibility for an RSTPd application to be controlled via same external means, e.g. a command line interface that connects via the network. This is actually quite common for different network applications, hence this module might be added in the future as a part of the framework. The dynamic architectures of the both implementations differ in only a small details.

3.2 IKEv2 specific subsystems

Specific subsystems of the IKEv2 protocol implementation encompass the following ones: the state machines subsystem, the message subsystem with the payload module, the

CFG module with different providers, the RADIUS subsystem, the supplicant module, as well as the IPsec subsystem. During source code configuration process, some modules can be excluded from the compilation phase thus making the IKEv2 daemon smaller and more efficient.

IKEv2 messages are composed of one or more payloads. Thus, the functions for creating and parsing payloads are placed in a separate module. When an IKEv2 message has to be sent, the state machines subsystem invokes the appropriate function from the message subsystem. This function calls in turn the appropriate functions for creating payloads. Eventually, the message subsystem invokes the network subsystem function for data to be actually sent. When an IKEv2 message is received from the network, the networking subsystem queues it to the message subsystem, which subsequently verifies and decodes it by calling appropriate functions from the payload module hence generating the appropriate `message_msg` structure. Finally, this structure is queued to the state machines subsystem for processing.

Generating messages is a very complex task because of many possibilities payload combinations. Because of that, generating some of the more complex messages is an iterative two-step process. The first step is represented by the main payload generating function that loops until the complete IKEv2 message is created. This function calls another - helper function, which determines the next payload to be included in the message, based on the current payload and data in appropriate structures.

The CFG module allows the IKEv2 daemon to obtain the IP configuration parameters from external sources. Each potential source is represented by a single provider: a DHCPv4 provider, a DHCPv6 or a private provider, this latter being different by actually using a private pool of configuration data instead of services being provided by a network server. The DHCPv4 provider is specific as it is neither a DHCPv4 relay nor a DHCPv4 client. On the contrary, it behaves as a combination of those two roles.

The RADIUS subsystem is used during EAP authentication simply relaying received EAP messages from the initiator to the RADIUS server, and vice versa. The initiator includes a supplicant module, but doesn't use any thread of its own. This stems from the fact that the calling supplicant functions are not expected to introduce any delays, therefore they can be implemented synchronously, what is actually the case. Behind the supplicant module there is an essentially customized version of the wpa_supplicant [13].

4 Implementation

IKEv2 is implemented in the C programming language, mostly because of the following two reasons: (i) C is one of the most efficient programming languages currently in use, and (ii) is the programming language of choice for the majority of system programs. In order to assure portability across different platforms we chose the Glib2 [14], which is a C library distributed with almost all existing Linux distributions and heavily used in the GNOME desktop environment. This library is also ported to the MS Windows family of operating systems. GLib2 provides the implementation of different basic algorithms and data structures, and also gives access to threads and thread pools in an operating system independent way. GLib2's scheduling of threads and events further simplifies the task of developing any application using it. It is very intensively used throughout the IKEv2 protocol implementation and we believe that it made IKEv2 even more efficient and smaller.

All the modules and subsystems are placed in separate files, as well as data structures exchanged between different modules. For each structure there are constructor and destructor functions, as well as getter and setter methods along with functions providing some additional functionality. This loosely mimics classes and methods from OO programming languages. All the functions belonging to a subsystem or module have a specific prefix in order to be easily recognizable and to avoid name clashes.

Currently, all the source code is placed inside a single directory (`src/`) but at some later time the framework will be moved to a separate directory in order to be more easily includable into other network applications built upon the same architecture. The whole IKEv2 implementation consists of over 46,000 lines of code. The most complex part is certainly the state machine module of about 6,000 lines, implementing three major state machines: (i) the initiator (seven states), (ii) the responder (seven states), and (iii) a separate state machine handling the behavior of IKEv2 for possible later events, e.g. creating additional Child SAs (eight states). In this latter case, the initiator and responder state machines have additional five shared states. The message module is larger in terms of number of lines, but is less complex in terms of functionality. Still, because it is IKEv2 deamon's entry point for network traffic, thus representing a possible point of attack, the message module bears some additional complexity that makes it not so easy to program.

The build infrastructure is based on the classical GNU Tools, i.e. automake and autoconf for preparing the part of the build infrastructure that performs code configuration. After code configuration, the make utility invokes the gcc compiler to compile the code and build the binary code. The tools for preparing the build infrastructure allow the code to be more generic so as the target platform features are dynamically discovered prior to compilation phase, and applied to the compilation process. The build system permits the user to influence what functionality will be included in the compiled application binary code.

5 Conclusion and Future Work

In this paper we present the architecture of an IKEv2 protocol implementation along with the most interesting implementation details. As the IKEv2 protocol is a good example of complex protocols, we found it useful to develop the implementation by taking special care of reuse issues. Because of that we identified the reusable protocol imple-

mentation parts as a framework, providing the means for future developers to produce some more efficient and less error prone protocol implementations in a shorter time. In comparison to the referent ACE framework, our one is less complex, thus also easier to understand and deploy.

As of this writing, our IKEv2 protocol implementation is operational, and is undergoing a series of tests for evaluating its performance, with special emphasis being given to processing and memory requirements put forth by the framework, what will provide us with a feedback to be used to improve it.

We are currently studying the possiblity to add mobile extensions to the IKEv2 protocol, and more specifically, the MOBIKE ones [15]. In our opinion this is quite a demanding task since the Linux kernel doesn't implement all the necessary add-ons like the address changes for the entries in the SAD, so as to force us to implement them along with the necessary changes to IKEv2.

At a later stage of our development endeavor, we plan to implement some features described in recently published RFCs, like the multiple authentication exchanges [16]. Long term goals are certainly related to the results of the "Better than nothing security" working group [17] and solutions to some PF_KEY problems we experienced during our IKEv2 development.

Acknowledgment
This paper describes the results of research being carried out within projects 036-0361994-1995 *Universal Middleware Platform for e-Learning Systems* funded by the Ministry of Science, Education and Sports of the Republic of Croatia and of IKEv2 Step2 funded by Siemens Networks d. d. Zagreb.

References

[1] C. Kaufman, Ed., *Internet Key Exchange (IKEv2) Protocol*, RFC4306, December 2005.

[2] S. Kent, K. Seo, *Security Architecture for the Internet Protocol*, RFC4301, December 2005.

[3] *IKEv2 Implementation*, http://ikev2.sourceforge.net/, May 2007.

[4] *OpenIKEv2 Implementation*, http://openikev2.sourceforge.net/, May 2007.

[5] *TheRacoon2Project*, http://www.racoon2.wide.ad.jp/, May 2007.

[6] D. Harkins, D. Carrel, *The Internet Key Exchange (IKE)*, RFC2409, November 1998.

[7] S. Sakane, K. Kamada, M. Thomas, J. Vilhuber, *Kerberized Internet Negotiation of Keys (KINK)*, RFC4430, March 2006.

[8] *Openswan*, http://www.openswan.org/, May 2007.

[9] D. C. Schmidt, *The ADAPTIVE Communication Environment (ACE)*, http://www.cs.wustl.edu/ schmidt/ACE.html, May 2007.

[10] IEEE Computer Society, *802.1D, IEEE Standard for Localand metropolitan area networks, Media Access Control (MAC) Bridges*, IEEE, June 2004.

[11] D. McDonald, C. Metz, B. Phan, *PF_KEY Key Management API, Version 2*, RFC2367, July 1998.

[12] *RSTPd Implementation*, http://rstpd.sourceforge.net/, May 2007.

[13] J. Vučak, L. Jelenković, M. Golub, Implementation of EAP authentication into IKEv2 protocol, *Proceedings of Information Systems Security*, MIPRO Internation Convention, pp. 173-176. Opatija, Croatia, May 2007.

[14] *GLib2 API documentation*, http://developer.gnome.org/doc/API/2.0/glib/index.html, May 2007.

[15] P. Eronen, Ed. *IKEv2 Mobility and Multihoming Protocol (MOBIKE)*, RFC4555, June 2006.

[16] P. Eronen, J. Korhonen, *Multiple Authentication Exchanges in the Internet Key Exchange (IKEv2) Protocol*, RFC4739, November 2006.

[17] *Better-Than-Nothing Security (btns) Working Group*, http://www.ietf.org/html.charters/btns-charter.html, May 2007.