Experimenting with early opportunistic key agreement

Catharina Candolin[1] & Janne Lundberg[1] & Pekka Nikander[2]

1: Laboratory for Theoretical Computer Science,
Helsinki University of Technology,
P.B. 5400, FIN-02015 HUT, Finland
catharina.candolin,janne.lundberg@hut.fi
2: Ericsson Research NomadicLab,
LMF/TR, FIN-02420 Jorvas, Finland
pekka.nikander@nomadiclab.com

Abstract

IPv6 is used for a variety of tasks, such as autoconfiguration, neighbor detection, and router discovery. The often recommended or required approach to secure these functions is to use IPSec. However, in order to use IPSec, a protocol for establishing Security Associations is needed. The currently prominent method for dynamically creating the Security Associations is to use the Internet Key Exchange (IKE) protocol. However, IKE requires that the underlying IP stack is functional to the extent that UDP can be used. Thus, a chicken-and-egg problem arises from the fact that IPSec is needed to configure IP, and IP is needed to configure IPSec. To overcome this problem, a weak and cheap key agreement protocol can be utilized to create temporary, initial Security Associations that are used until a proper key agreement protocol can be run. In this paper, we present a protocol that can be used prior to e.g. IKE, and describe our implementation as a proof of concept. The protocol is designed to provide protection against resource exhausting denial-of-service attacks as well as reachability information with as few protocol rounds as possible. Thus, the protocol is especially suitable for wireless networks, for example, ad hoc networks.

1. Introduction

IPv6 [4] is used for a number of functions, including address configuration, neighbor detection, and router discovery. To protect these functions, it is either suggested or recommended to use IPSec [8]. The currently prominent method for creating IPSec Security Associations, the Internet Key Exchange (IKE) [6] protocol, requires that the underlying IP stack is fully functional, at least to the point that UDP may be used. Another problem with the IKE protocol is that it is relatively heavy. These two problems open up a potential source for various denial-of-service attacks. Furthermore, a chicken-and-egg problem arises from the fact that IPSec is needed to configure IP, and fully configured IP is needed to establish the IPSec Security Associations.

To solve the problems presented above, a weak early key agreement protocol can be used, as suggested in [10]. The purpose of the protocol is to establish initial Security Associations that can be used, for example, for autoconfiguration purposes. In this paper, we introduce a DoS-resistant anonymous key agreement protocol that establishes such initial Security Associations before it is possible to run a full blown key agreement protocol, for example IKE. We also briefly describe our implementation, which was made as an experiment to show that the protocol can be realized fairly easily.

The rest of this paper is organized as follows: in Section 2, the previously mentioned chicken-and-egg problem is further analysed and previous work done for an authentication protocol is presented. In Section 3, our early key agreement protocol is described and analyzed with respect to the requirements specified. Section 4 contains a description of our implementation, and Section 5 concludes the paper.

2. Background

The stateless autoconfiguration specification [12] defines a method of providing initial boot time configuration for IP hosts. If autoconfiguration is to be used over wireless networks, such as ad hoc networks, a wide variety of security attacks become possible due to the lack of physical protection of the network medium. To protect wireless hosts from becoming victims to masquerading or denial-of-service attacks, security can be provided on the link layer, or the autoconfiguration mechanism can be improved to deal with potential threats. The protocol described in this paper adopts the latter approach.

In the stateless autoconfiguration mechanism, a host sends a series of Neighbor Solicitations [9] to the local link. These messages contain a tentative IP address that the host would like to use as its link local address. If the address is already in use by another host, that host will send a Neighbor Advertisement as a reply, and the booting host must choose a new address. If no replies are received, the host may start using the selected address. Once the host has acquired its address, it sends a series of Route Solicitation messages to find the local routers. The local routers reply with Route Advertisement messages. The Router Advertisement typically contains a set of routing prefixes that the host can use for creating a globally routable address.

To provide security, IPSec AH [7] can be used to protect the Neighbor Discovery messages. AH can also be used to verify that the Neighbor and Router Advertisements came from valid and authorized hosts. However, this requires manually configured Security Associations, since the only way of automatically creating Security Associations is by using IKE, but IKE in turn requires an IP address in order to be functional.

Due to this chicken-and-egg problem, IKE cannot be used in the initial phase to provide security for IP configuration. A simple key agreement protocol is thus needed prior to running IKE.

2.1. Previous work

In [1][2], a DoS-resistant authentication protocol using puzzles is introduced. The main idea is to allow the server to generate puzzles that the client must solve before the server creates any protocol state or performs heavy computations. Creating a puzzle and verifying the solution must be inexpensive for the server, and the cost of solving the puzzle should be adjustable by the server. The client should not be able to precompute solutions to the puzzle, and the server must be able to give the same puzzle to several clients. However, if one client knows the solution to the puzzle, it should not be able to help another client to solve the puzzle.

The puzzle used is a brute force reversal of a hash function, such as MD5 [11] or SHA1 [5]. The main criteria of the hash function is that the cost of reversing it should be adjustable by the server. Both MD5 and SHA1 meet this requirement. To create new puzzles, the server periodically generates a nonce N_S, which it sends to the clients. The server also decides upon the difficulty level k of the puzzle. The cost of solving the puzzle is exponentially dependent on k. If $k = 0$, no work is required, and if $k = 128$ (in the case of MD5), the client would have to reverse the whole hash function. The latter is assumed to be impossible, and suitable values for k are thus 0–64 bits. The nonce N_S together with k form the puzzle that the client must solve.

Upon reception of the puzzle, the client generates a nonce N_C. If the client reuses N_S, the client creates a new puzzle by creating new values of N_C. Furthermore, including the client nonce prevents an attacker from consuming a specific client's puzzles by computing solutions and submitting them to the server prior to the given client. The client proceeds by repeating the following operation until it finds the correct solution:

result = hash(client_identity, N_S, N_C, solution)

In order to prevent clients from impersonating other clients, the identity is included as a parameter in the puzzle. Solving the puzzle for one client does not help in finding solutions for other clients. Thus, impersonation attacks become infeasible, as the solution must be recomputed for each client separately. Furthermore, since the server refreshes the value of N_S periodically, it is impossible to precompute the solutions. However, a client may reuse the N_S by solving the puzzle with a new N_C.

The client sends the solution of the puzzle back to the server. The server verifies that the solution is correct and checks that it came within a valid time frame. For each value of N_S, the server keeps track of correctly solved instances of the puzzle to prevent solutions from being reused.

The puzzle presented in [2] is assumed to satisfy the criteria for good puzzles. Due to the nonce created by the server combined with the time limit, the solution to the puzzle cannot be precomputed. The protocol includes the identity of the client, so that the solution computed by one client does not help another.

3. A weak key agreement protocol

To solve the chicken-and-egg problem presented in Section 2, a key agreement protocol is needed to establish initial Security Associations. To provide denial-of-service protection, we adopt the puzzle based approach described in Section 2.1.

The purpose of the protocol is to create key material that is to be used only for a short period of time. Thus, the protocol does not need to withstand active attacks, but should rather be simple and efficient to implement.

The goals of the protocol are the following:

1. To establish an unauthenticated (or partially authenticated) secret shared key for the client and server. The key should have the following properties:

    - **key freshness**: the key should not have been used in previous runs of the protocol.
    - **key control**: both parties should participate in creating the key.

    Properties such as key authentication and forward secrecy are not needed for the initial security associations and can therefore be neglected.

2. To provide the server with some level of protection against resource consuming denial-of-service attacks launched by the client. The client should commit to the communications before the server will create any state or perform heavy computations.

3. To ensure to both the client and the server that the other party is reachable at the claimed location.

3.1. Protocol description

To create the keying material between the client and the server, the protocol relies on Diffie-Hellman based key agreement. Both parties possess a public/private Diffie-Hellman key pair. The server also holds two other cryptographic keys: the *state key* and the *signature key*. The former is used to calculate a MAC over the current state of the server and the latter is used for creating signatures.

To provide protection against denial-of-service attacks, a puzzle based on a cryptographic hash function (MD5 or SHA1) is used. The server generates a nonce and applies the hash function to it. It then zeroes the k first bits of the result and sends the resulting string together with the value of n to the client. This puzzle is recomputed within a certain time interval to avoid replay or masquerading attacks. The client solves the puzzle by computing the n bits by brute-force. Once it succeeds, it sends the result to the server, which verifies that the solution is correct. Only then will the server commit itself to the communication.

The protocol is depicted in Figure 1. To ensure both parties about the reachability of the other party, a three round protocol is needed. When the client wishes to engage in a communication with the server, it sends a `Trigger` message, which initializes the connection. The message does not cause the server to store any state.

Upon reception of the `Trigger`, the server sends the puzzle to the client by generating a `Request` packet containing the time information, the MAC, its public key, the puzzle, and the signature. The state information is thus passed to the client instead of being stored at the server. Also, the format of the time information does not need to be determined, as it is only used by the server.

CLIENT

<addr_client>
dhc := new Diffie−Hellman private key
DHC := new Diffie−Hellman public key

SERVER

<addr_server>
<state_key_server>
<sign_key_server>

dhs := new Diffie−Hellman private key
DHS := new Diffie−Hellman public key

Trigger

UTC_TIME := current time
PUZZLE := create_new
STATE := <addr_server, addr_client, sport, lport,
UTC_TIME, PUZZLE>

MAC := HMAC(state_key_server, STATE)

SIGS := SIGN(sign_key_server, <UTC_TIME,
MAC, DHS, PUZZLE>)

REQUEST = <UTC_TIME, MAC, DHS, PUZZLE, SIGS>

optional: verify SIGS
SOLUTION := solve(PUZZLE, addr_client)
KEYMAT := Diffie−Hellman(dhc, DHS)
SIGC := SIGN(KEYMAT,
<PUZZLE, SOLUTION, MAC, SIGS>)

RESPONSE = <SOLUTION, MAC, DHC, SIGC>

verify SOLUTION(PUZZLE, addr_client)
STATE := <addr_client, sport, lport, SIGS>
verify MAC
KEYMAT := Diffie−Hellman(dhs, DHC)
verify SIGC

Figure 1: The protocol messages

When the client receives the `Request`, it may verify the signature to check that it was created by the server. This feature is optional. It then solves the puzzle, generates the shared secret key (according to Diffie-Hellman key generation), and creates a signature with the shared key of the puzzle, the corresponding solution, the MAC, and the state information sent by the server. This is added to the `Response` packet, which is sent back to the server.

The server now knows that the client was reachable at the given address. It then proceeds to verify the solution and generates the shared key. After that, the signature is checked. If all checks succeed, then the protocol terminates successfully.

3.2. Analysis

It can be stated that the protocol meets its goals if the following assumptions hold:

1. The Diffie-Hellman key exchange establishes a shared secret key that is known to both the client and the server, but cannot be computed by a passive adversary. This property has been previously proved in the literature, and the proof is thus omitted from this paper.
2. Solving the puzzle in fact is hard. This can be proved by showing that the chosen hash function is indeed a one-way function, and thus the only way to solve the puzzle is by brute force, which has complexity O(2^{n-1}), where n is the number of bits to retrieve.

The reachability of the client and server is verified by the three round nature of the protocol. Once the client receives the `Request` message, it is assured of the fact that the server is residing at the claimed location. Furthermore, upon receiving the `Response` message, the server can verify that the client in turn is reachable.

Client port		Server port	
Server SPI			
MAC type	Group type	Sig req type	Sig resp type
Time			
Puzzle type	K	Zero	
Puzzle−R			
Puzzle−bits			
Server public DH key			
SigS			
MAC			

Figure 2: The request message

Protection against denial-of-service attacks is provided, since the server does not store any state before it has verified the solution to the puzzle, and the puzzle in turn is assumed to be hard to solve. Furthermore, the time limit deployed ensures that previous solutions cannot be reused, and the added client identity ensures that a solution cannot be redistributed to other clients.

4. Implementation

As a proof of concept, we implemented the key agreement protocol as a user space program in Linux. The implementation takes advantage of ICMPv6 [3] messages to simplify communication with the IPv6 layer. The `Trigger` is implemented as an Echo Request message.

The server uses a time interval of one minute to generate a puzzle and a time stamp, which it signs using its private *signature key*. The signature covers the packet depicted in Figure 2, starting from the `MAC-type` field until and including the `Server public DH key` field. The IPv6 address of the server is appended to the data. Upon reception of a `Trigger`, the server sends the same puzzle as a response to all clients until the puzzle is changed.

The format of the `Request` message is depicted in Figure 2. The `Request` message is carried in an ICMP packet, which is omitted from the picture for simplicity.

The puzzle contains three fields: the number of bits to solve, K, a random 64-bit value `Puzzle-R`, and the target bits, `Puzzle-bits`. The client must find such a 32-bit number, `Puzzle-x`, that the K least significant bits of `result = hash(Puzzle-R | Client address | Puzzle-x)` matches the value of `Puzzle-bits`. The only efficient way of solving the puzzle is by brute force, and the solution cannot be shared by several clients as it is dependent on the IP address of the client.

The `MAC`-field is an HMAC-value which is calculated over the whole message in Figure 2, using the private *state key* of the server. Upon reception of a `Trigger`, the server signs its state and adds it to the `Request` message. However, the server does not store this state at this point of the protocol run. The field is later used by the server to authenticate the state when it is sent back by the client in the `Response` message.

The client may verify the signature that it receives in the `Request` packet, but this check is optional. It then solves the puzzle and computes a shared secret key using its own private key and the public Diffie-Hellman key of the server. The entire `Request` message is inserted into the `Server state` field of the `Response` message depicted in Figure 3. Finally, the `Response` message is signed with the shared secret key that was

Server state
Client SPI
Puzzle–x
Client public DH key
Signature

Figure 3: The response message

created as a result of the Diffie-Hellman exchange, and the message is sent to the server.

When the server receives a `Response` message, it first checks that a solution for the current puzzle has not been received from the client before, and that the state information that was sent in the `Request` message has not been modified. Next, the time stamp and the puzzle solution are verified. If the time stamp has not expired and the puzzle solution is correct, the server computes the shared secret key using the public Diffie-Hellman key of the client and its own private Diffie-Hellman key. Finally, the server checks the signature in the response message.

The Security Association is created once the steps described above have been successfully completed.

5. Conclusion

A major problem with IPv6 autoconfiguration, especially when performed over wireless networks, is the chicken-and-egg problem related to security. In order to securely provide an IP address, the IPSec IKE protocol is needed, but in order to run IKE, an IP address is needed. To overcome this problem, a simple protocol for establishing initial security associations is needed. The protocol does not need to withstand active attacks, but it must provide protection from resource consuming denial-of-service attacks launched by a malicious initiator. In this paper, we have described a Diffie-Hellman based protocol, which relies on three round messaging to prove reachability and client puzzles to provide protection against denial-of-service.

References

[1] T. Aura. *Authorization and availability—Aspects of Open Network Security*. PhD thesis, Helsinki University of Technology, 2000.

[2] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant Authentication with Client Puzzles. In *Cambridge Security Protocols Workshop 2000*, 2000.

[3] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. Request for Comments 2463, 1998.

[4] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Request for Comments 2460, December 1998.

[5] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). Request for Comments 3174, 2001.

[6] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). Request for Comments 2409, 1998.

[7] S. Kent and R. Atkinson. IP Authentication Header. Request for Comments 2402, 1998.

[8] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Request for Comments 2401, 1998.

[9] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). Request for Comments 2461, 1998.

[10] P. Nikander. Denial-of-Service, Address Ownership, and Early Authentication in the IPv6 World. In *Cambridge Security Protocols Workshop 2001*, 2001.

[11] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, IETF, April 1992.

[12] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. Request for Comments 2462, 1998.