Analysis of the HIP Base Exchange Protocol

Tuomas Aura[1], Aarthi Nagarajan[2], and Andrei Gurtov[3]

[1] Microsoft Research, Cambridge, United Kingdom, tuomaura@microsoft.com
[2] Technische Universität Hamburg-Harburg, Germany, aarthi.nagarajan@tu-harburg.de
[3] Helsinki Institute for Information Technology, Finland, gurtov@cs.helsinki.fi

Abstract The Host Identity Protocol (HIP) is an Internet security and multi-addressing mechanism specified by the IETF. HIP introduces a new layer between the transport and network layers of the TCP/IP stack that maps host identifiers to network locations, thus separating the two conflicting roles that IP addresses have in the current Internet. This paper analyzes the security and functionality of the HIP base exchange, which is a classic key exchange protocol with some novel features for authentication and DoS protection. The base exchange is the most stable part of the HIP specification with multiple existing implementations. We point out several security issues in the current protocol and propose changes that are compatible with the goals of HIP.

1 Introduction

The Host Identity Protocol (HIP) is a multi-addressing and mobility solution for the IPv4 and IPv6 Internet. HIP is also a security protocol that defines host identifiers for naming the endpoints and performs authentication and creation of IPsec security associations between them. A new protocol layer is added into the TCP/IP stack between the network and transport layers. The new layer maps the host identifiers to network addresses and vice versa. This achieves the main architectural goal of HIP: the separation of identifiers from locations. In the traditional TCP/IP architecture, IP addresses serve both roles, which creates problems for mobility and multi-homing.

The *host identity* (HI) in HIP is a public key. This kind of identifier is self-certifying in the sense that it can be used to verify signatures without access to certificates or a public-key infrastructure. The host identity is usually represented by the *host identity tag* (HIT), which is a 128-bit hash of the HI. IPv4 and IPv6 addresses in HIP are purely locations. The protocol is composed of three major parts. The endpoints first establish session keys with the HIP *base exchange* [10], after which all packets are protected using IPsec ESP [9]. Finally, there is a readdressing mechanism to support IP address changes with mobility and multi-homing.

In this paper, focus on the HIP base exchange as a cryptographic key-exchange protocol. We analyze its security with emphasis on denial-of-service (DoS) issues. Several protocol details were found to be vulnerable to DoS attacks or accidental deadlocking. Additionally, we point out a minor issue with key freshness. To fix these problems, we propose feasible solutions that are in line with the goals of the HIP protocol. (Note that this paper does not fully explain the thinking behind the HIP protocol [11] and we do not try argue in favor of or against adopting HIP as a part of the Internet protocol stack.) We start by introducing the HIP base exchange in Section 2 and discuss the identified problems and solutions in the following sections.

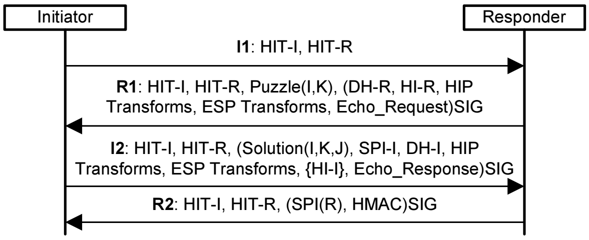
Initiator
Responder
I1: HIT-I, HIT-R
R1: HIT-I, HIT-R, Puzzle(I,K), (DH-R, HI-R, HIP Transforms, ESP Transforms, Echo_Request)SIG
I2: HIT-I, HIT-R, (Solution(I,K,J), SPI-I, DH-I, HIP Transforms, ESP Transforms, {HI-I}, Echo_Response)SIG
R2: HIT-I, HIT-R, (SPI(R), HMAC)SIG

Figure 1 HIP base exchange messages

2 HIP Base Exchange

The main building block of the HIP protocols is the HIP base exchange [10]. It is used to establish a pair of IPsec security associations (SA) between two hosts. The base exchange is built around a classic authenticated Diffie-Hellman key exchange but there are some unusual features related to DoS-protection. No certificates are required for the authentication because the HITs are self-certifying. The protocol can be compared with key exchange protocols like IKE [7] and IKEv2 [4] and evaluated against most of the same security requirements [13].

In this section, we outline the base exchange messages I1, R1, I2 and R2 shown in Figure 1. The Initiator first sends an empty message I1 to the Responder. It triggers the next message R1 from the Responder. All HIP packets contain the initiator and responder identity tags (HIT-I and HIT-R) in the header.

Even before the Responder receives the I1 message, it precomputes a partial R1 message. The precomputed R1 includes the HIT-R, the Responder's Diffie-Hellman key, the Responder host identity HI-R (i.e., a public key), the proposed cryptographic algorithms for the rest of the base exchange (HIP transforms), the proposed IPsec algorithms (ESP transforms), and an Echo_Request field. The Echo_Request contains data that the Initiator returns unmodified in the following message I2. It is important that the responder sends R1 without creating any protocol state. The Echo_Request can be used to store some data in a stateless way. The responder signs the message. The HIT-I and the Puzzle field are left empty at this point. These two fields are populated after receiving an I1 and they are not protected by the signature.

The Puzzle parameter in R1 contains a cryptographic puzzle [3,4], which the Initiator is required to solve before sending the following packet I2. The idea is that the Initiator is forced to perform a moderately expensive brute-force computation before the Responder commits its computational resources to the protocol or creates a protocol state. The puzzle has three components: the puzzle nonce I, the difficulty level K, and the solution J. It is easiest to explain how a puzzle solution is verified: First, concatenate I, the host identity tags HIT-I and HIT-R, and the solution J. Then, compute the SHA-1 hash of the concatenation. Finally, check that the K low-order bits of the hash are all zeros.

$$\text{Ltrunc}(\ \text{SHA-1}(\ I \mid \text{HIT-I} \mid \text{HIT-R} \mid J\ ),\ K\ ) == 0$$

The Initiator must do a brute-force search for the value of J, which takes $O(2^K)$ trials. The Responder, on the other hand, can verify the solution by computing a single hash.

On receiving R1, the initiator checks that it has sent a corresponding I1 and verifies the signature using the public key HI-R. If the signature is ok, it solves the puzzle and creates the message I2. I2 includes the puzzle and its solution, the Initiator's Diffie-Hellman key, the HIP and ESP transforms proposed by the Initiator, a security parameter index (SPI) for the Responder-to-Initiator IPsec SA, the Initiator public key (HI-I) encrypted using the new session key, and the Echo_Response. A signature covers the entire message. Key material for the session keys is computed as a SHA-1 hash of the Diffie-Hellman shared secret Kij:

KEYMAT_k = SHA-1(Kij, | sort(HIT-I | HIT-R) | k) for k = 1, 2,...

On receiving I2, the Responder verifies the puzzle solution. If it is correct, the Responder computes the session keys, decrypts HI-I, and verifies the signature on I2. The Responder then sends R2, which contains the SPI for the Initiator-to-Responder IPsec SA, an HMAC computed using the session key, and a signature.

For the Initiator, the exchange is concluded by the receipt of R2 and the verification of the HMAC and the signature. The HMAC confirms the establishment of the session key. For the Responder, the key confirmation is provided by the first inbound IPsec packet that is protected with the new security association.

3 Replays of R1

In this section, we consider replays of the R1 message. As explained earlier, R1 is partially signed. There is, however, nothing in R1 to prove its freshness.

Before explaining why we think the freshness of R1 should be checked, we'll consider arguments against such protection. First, it is infeasible to include a nonce in the signed part of R1 because that would prevent the Responder precomputing the signature. Thus, nonce-based replay protection appears not to work. Second, timestamps have well-known problems with clock synchronization. Third, there are features in the protocol that mitigate the consequences of R1 replays. The signature on the last message, R2, covers the session key (indirectly by covering the HMAC). Thus, the Initiator detects the replay of an old Diffie-Hellman key in R1 when it receives R2. For these reasons, the freshness of R1 many not appear very important.

There is, however, another type of attack based on replaying R1: the attacker spoofs R1 and tricks the Initiator into solving the wrong puzzle. The attacker can send a trickle of replayed R1 messages to the Initiator with random I values. If the frequency of the spoofed R1 messages is higher than the roundtrip between the Initiator and Responder, the first R1 to arrive at the Initiator after it has sent I1 is always a replay. This prevents the Initiator from ever solving an authentic puzzle.

Problem 1: Attacker can replay the signed parts of R1 and trick the Initiator into solving the wrong puzzle. This results in denial-of-service for the Initiator because the solution is rejected by the Responder.

The seriousness of the above DoS attack is increased by the ease of obtaining the replay material. No sniffing is necessary; the attacker can obtain the partially signed R1 by sending an I1 message to the same Responder.

There is a simple mechanism for preventing the attack: include a nonce in I1 and in the unsigned part of R1. The nonce prevents the attacker from replaying R1 unless it can sniff the corresponding I1. The cost of adding the nonce is low and it reduces significantly the threat of R1 replays.

Proposed solution 1: Add a nonce of the Initiator to I1 and to the unsigned part of R1 to prevent replays of R1.

The lack of this kind of "cookies" in the current HIP specification may be the influence of [3] where the first protocol message was potentially a broadcast message and, thus, could not contain a per-responder nonce. In HIP, the first message I1 is always unicast and therefore a nonce can be added.

4 Reuse of Diffie-Hellman Keys

Diffie-Hellman public keys (g^x and g^y) are often reused in order to amortize the cost of public-key generation over multiple key exchanges. The trade-off is the loss of forward secrecy: old session keys can be recovered as long as the key owner stores the private exponent. Another consequence of the key reuse is that the Diffie-Hellman shared secret (Kij = g^{xy}) is not guaranteed to be fresh. The usual solution is to compute the session key as a hash of Kij and nonces from both participants.

In the HIP base exchange, Diffie-Hellman keys are sometimes reused. First, the Responder uses the same key in all R1 messages over a time period (Delta). Second, the same host acting simultaneously as the Initiator and as the Responder uses the same key in both roles. (While the reuse is not mandated by the specification, it is probably necessary in practice to avoid further complicating the protocol state machine. See Section 7.) Nevertheless, the HIP base exchange does not include nonces in the session-key computation.

The lack of nonces may, in fact, lead to a vulnerability. If the same two hosts perform the base exchange twice within the time Delta (i.e., the time during which Diffie-Hellman keys are reused), they end up with the same session keys. In practice, this depends on the timing in the implementations at each end-point. Such dependence on the implementation detail is, of course, not acceptable in a security protocol.

Problem 2: Reuse of Diffie-Hellman keys may result in reuse of session keys.

The puzzle I and the solution J are, in effect, nonces of the Responder and Initiator. Thus, they can be used for freshness in the session-key generation:

$$\text{KEYMAT\_k} = \text{SHA-1}(\text{Kij} \mid \text{sort}(\text{HIT-I} \mid \text{HIT-R}) \mid \text{I} \mid \text{J} \mid \text{k})$$

Proposed solution 2: The nonces I and J should be hashed into the key material.

5 Puzzle Implementation

The client puzzles force the Initiator to perform a moderately expensive computation before the Responder commits its computational resources or creates a protocol state. The Initiator, in effect, pays for the resources of the Responder by solving the puzzle, which is why this kind of mechanism is sometimes called hash cash. During DoS attacks or otherwise heavy load, the Responder increases the price. The HIP puzzle mechanism originates from [3] but some changes have been made in order to address specific security concerns. In this section, we analyze the HIP puzzles and suggest changes that are compatible with the current implementations.

The following requirements for the puzzles can be derived from [10] or [3]:

1. The Responder must not verify the signature on I2 or do other expensive computation before it has verified the puzzle solution. It must verify at most one signature for each puzzle solved by the Initiators (or attackers).
2. The Responder must not create a per-Initiator or per-session state before verifying the puzzle solution. It must only store a small amount of information for each puzzle solved by the Initiators (or attackers).
3. The cost of creating and verifying a puzzle must be small, preferably requiring only one computation of a one-way hash function by the Responder.
4. The attacker must not be able to pre-compute solutions for a burst attack. That is, the solutions must remain valid only for a short time period.
5. The Responder must not reject correct solutions sent by an honest Initiator because the attacker has previously solved the same puzzle.
6. In order to verify the Initiator IP address, the Responder must recognize I in I2 as the same nonce that it previously sent to the source address of I2.
7. The Responder must accept at most one correct puzzle solution for each I1/R1 exchange that takes place. (We will later argue that Requirement 7 is unnecessary even though it is implied in [10].)

A naive puzzle implementation will send a random number I in every R1 and store the random number until it either receives a solution or a time Delta has passed. The naive puzzles fail Requirement 2 because a small state is created for each received I1. An attacker can exploit this by flooding the Responder with I1 messages.[1]

The puzzle implementation proposed in [10] (Appendix D) tries to address all of the above requirements. The basic idea is that the Responder has a fixed-size table of pre-generated random I values (I_k for k=0...n-1), called *cookies*, and it selects one of them for each R1 message by computing the index to the cookie table as a function of HIT-I and HIT-R and the Initiator and Responder IP addresses (IP-I and IP-R). The function is not a strong cryptographic hash but an inexpensive combination of XOR operations. Upon receiving a solution in I2, the Responder recomputes the index k to the cookie table and checks that the I in I2 matches the value in the table. The Responder then verifies the puzzle solution by computing a SHA-1 hash. If the solution is correct, it marks the nonce I_k as used. Only the first solution to each puzzle is accepted. A background process replaces the used nonces I_k with new ones within the time period Delta.

Unfortunately, the pseudo-random function described in the specification is linear and it is easy for the attacker to create a collision with the honest Initiator. The function is simply an XOR of the Responder's secret 1-byte key r and the bytes of the Initiator and Responder HITs and IP addresses:

index = XOR of r and all bytes of HIT-I, HIT-R, IP-I and IP-R

[1] The naive algorithm is presented in Section 4.1.1 of [10]. The rest of the specification talks about the cookie-table mechanisms.

The attacker can cause an index collision with an honest Initiator by selecting the Initiator HIT in the spoofed I2 message as follows:

HIT-A[m] = HIT-I[m] for m=0,...,14
HIT-A[15] = XOR of HIT-I[15] and all bytes of IP-A and IP-I

The attacker obtains an R1 by sending an I1 from its own IP address (IP-A), solves the puzzle for HIT-A and IP-A, and then sends an I2 from IP-A using HIT-A as the initiator HIT. The Responder accepts this solution and rejects the one sent later by the honest initiator because the puzzle has already been used. The signature on the attacker's I2 is invalid because HIT-A is not a hash of any public key, but the Responder must mark the puzzle used even when the signature is invalid.

Problem 3: The pseudo-random function for selecting the puzzle is linear. Thus, the attacker can cause collisions and consume puzzles of honest initiators by solving them.

The obvious solution is to compute the index with a second-preimage-resistant hash function. The trade-off is that this adds two hash computations to the total cost of creating and verifying a puzzle. We suggest using a standard function like SHA-1 because it would be non-trivial to design a lower-cost non-linear function that nevertheless has sufficient strength to match puzzle difficulties up to $O(2^{64})$.

Proposed solution 3: The pseudo-random function used to select the value of I should be a strong hash function, such as SHA-1.

There is another problem with the cookie table. Even if we use SHA-1 to index the table, the attacker can still solve a lot of puzzles so that a significant portion of the puzzles in the table remain used at any point of time. The attacker can do this from its own IP address by picking random HITs and by solving the puzzles for them. For example, in order to consume n/2 nonces, the attacker has to solve approximately 0.69*n puzzles. The suggested table size is n=256, which means that the attacker needs to solve about 177 puzzles during the time Delta to cause the exchange to fail for 50% of honest Initiators. Even with a larger table, the birthday paradox ensures that the attacker can block out a small number of legitimate connection attempts.

Problem 4: With any realistic cookie-table size, the attacker can cause some index collisions and, thus, authentication failures.

The purpose of the cookie table is to implement Requirements 6 and 7 above, i.e., to bind the Initiator IP address to the puzzles and to prevent the reuse of puzzles. If an attacker wants to flood the Responder with correct puzzle solutions, it has to repeat the I1/R1 exchange and it must use its own IP address.

We suggest a way of generating puzzles that does not require the Responder to store a table of nonces: compute the puzzle I as a hash of the Initiator HIT and IP address, and a periodically changing secret key K_{Res} known only to the Responder.

$$I = \text{SHA-1}(\text{IP-I} \mid \text{IP-R} \mid \text{HIT-I} \mid \text{HIT-R} \mid K_{Res})$$

The Responder does not need to store any information after sending this puzzle in R1. When it receives I2, it can recompute I from the information in that message.

Proposed solution 4: In order to bind the puzzle to the Initiator IP address, compute the puzzle I as a SHA-1 hash of the address. This gives the same level of security as the cookie-table with the same computational cost and less memory.

It should be noted that the cookie-table mechanism implements Requirement 7 but our alternative solution does not. That is, we do not prevent the Initiator from solving the same puzzle multiple times during the time Delta. It is not clear what would be achieved by forcing the Initiator to perform the I1/R1 exchange more frequently.

Next, we turn our attention to another feature in the puzzle mechanism that does not achieve its intended purpose. It is suggested in [10] (Section 4.1.1 and in Appendix D) that if the Responder receives multiple false solutions from the same IP address and HIT, it should block further I2 messages from this source for a period of time. The problem is that this contradicts Requirement 2, i.e., not creating any per-Initiator state at the Responder until a correct puzzle solution is verified. The attacker can exhaust the blocking mechanism by flooding the Responder with false puzzle solutions from spoofed IP addresses.

Problem 5: If the Responder blocks I2 packets from HITs or IP addresses after receiving false cookie solutions, the blocking mechanism is vulnerable to a flooding attack.

It is not easy to define any robust criteria for filtering incoming puzzle solutions without verifying them. Any such filtering mechanism can probably be circumvented or, worse, exploited in DoS attacks. It is, therefore, better to design the system without the blocking.

Proposed solution 5: The responder should not create any state after receiving false puzzle solutions.

Finally, we suggest a complete puzzle mechanism that solves the above problems while maintaining compatibility with the current HIP specification.

- The Responder has a secret key K_{Res} that it generates periodically, once in every Delta. The Responder remembers the two last values of K_{Res}.
- The Responder uses one pre-signed message R1. The signature is recomputed periodically when the Diffie-Hellman key is replaced, which happens slightly less often than the generation of a new K_{Res}.
- The Responder computes the puzzle nonce I as the SHA-1 hash of the newest K_{Res} and the Initiator and Responder HITs and IP addresses. It computes the value of I on the fly for each R1 and forgets the value after sending R1.
- A 1-bit key counter is incremented every time a new key K_{Res} is generated. The value of this counter is sent in R1 and I2 with the puzzle.[2]

[2] The counter bit can be sent in the Opaque field of the puzzle data structure or in an Echo_Request field, both of which are returned unmodified in I2.

- On receiving I2, the Responder recomputes I from K_{Res} and the HITs and IP addresses, which it takes from the I2 message. The correct K_{Res} is identified by the 1-bit counter.
- The Responder then compares the computed value of I with the one in I2. If the values match, it verifies the puzzle solution J by computing the SHA-1 hash.
- If the puzzle solution is correct, the Responder stores the puzzle I and the correct solution J. (Alternatively, the Responder can store the Initiator HIT and IP address and the correct solution J.) Separate storage is maintained for the latest and second latest value of K_{Res}. When the older K_{Res} is deleted, the corresponding storage is purged as well.

6 Encryption of Initiator HI in I2

The Initiator host identity HI-I in the I2 message is encrypted with the new session key. In this section, we argue that the encryption is unnecessary and bad for security.

In some key exchange protocols, such as IKE [4], the endpoint identifiers, or the certificates containing the identifiers, are encrypted to enhance user privacy. The host identifiers in HIP could be similarly encrypted to prevent an eavesdropper from identifying the hosts. There are, however, several reasons why the encryption does not make sense. First, the privacy issue is mitigated by the fact that the host identifiers are public keys and not user or machine names. Second, the HITs that appear in every message header are hashes of the host identifier and, thus, uniquely identify the hosts. Encrypting the HI achieves little as long as the HIT is sent unprotected. Third, the Responder identity in R1 is sent in plaintext. An attacker that impersonates the Responder can easily discover the Initiator's identity by reversing the roles and sending an I1 message to the Initiator. It will receive the peer's HI in R1. (The HIP specification suggests that a privacy-conscious HIP host may refuse to act as the Responder but that will lead to communications failure if both endpoints follow the same policy.) For these reasons, the encryption is ineffective as a privacy mechanism.

There could be other reasons for the encryption. First, it could be a freshness check: the encryption is a function of the session key, which is a function of the Responder's Diffie-Hellman key, which changes for every time Delta. Thus, valid encryption links message I2 to a fresh value generated by the Responder. But this freshness check is superfluous because the puzzle nonce I already provides freshness for the I2 message. Second, the encryption could serve as key confirmation: by encrypting with the session key, the Initiator proves to the Responder that it knows the session key. But this is clearly not the intention because there is a separate mechanism for key confirmation. (The protocol state machine requires the Responder to wait in the R2-SENT state for a valid ESP packet from the Initiator before moving to the ESTABLISHED state.) Hence, neither freshness nor key confirmation is a valid motivation for encrypting the HI.

In addition to being unnecessary, the encryption of the Responder HI prevents NAT and firewall support [1,5,6] for HIP. The catch is that when the HI is encrypted, middle boxes in the network cannot verify the signature on I2 and, thus, cannot safely create a state for the HIP association. On the other hand, if the HI is not encrypted, a stateful middle box like a NAT can process I2 and create a protocol state for the

Initiator. A firewall can also verify the puzzle and signature on I2, thus making it possible to push the I1/R1 exchange into the firewall and to filter false puzzle solutions at the firewall. The encryption of HI-I prevents such middle-box implementations. (See [11,14,15] for details.)

Problem 6: The encryption of the Initiator HI in the I2 message does not provide any privacy protection and prevents HIP support in firewalls and NATs.

The solution is obvious:

Proposed solution 6: Do not encrypt the Initiator HI in I2.

7 State Machine Issues

The base-exchange protocol state machine is shown in Figure 2. The Initiator moves from the UNASSOCIATED state via I1-SENT, on receiving R1, to I2-SENT and, finally, on receiving R2, to ESTABLISHED. The Responder remains in UNASSOCIATED until it receives and verifies I2, after which it moves to R2-SENT and, finally, on receiving a valid ESP packet, to ESTABLISHED. These basic state transitions are sensible and have been thoroughly tested. On the other hand, the recovery from packet loss and handshake collisions, where the hosts act simultaneously in the Initiator and Responder roles, is ad-hoc and not fully specified. In this section, we discuss problems with these less frequent state transitions.

The first issue is that there is a timeout transition for the Responder from the R2-SENT to ESTABLISHED. The reason for waiting for the ESP packet is key confirmation: only after receiving the ESP packet the Responder knows that the Initiator received a fresh Diffie-Hellman key (rather than a replay) in R1 and has computed the valid session key. The timeout transition breaks this mechanism.

Problem 7a: The timeout transition to the ESTABLISHED state breaks key confirmation.

An alternative would be a timeout transition to UNASSOCIATED instead of the ESTABLISHED state. This could, however, cause an infinite loop (livelock) in which the hosts never reach the ESTABLISHED state, even if no messages are lost or spoofed. Consider the following sequence of events: both hosts initiate the protocol by sending I1 and move to the I1-SENT state; both hosts receive I1 and respond with R1; both hosts receive R1, respond with I2 and move to I2-SENT; both hosts receive I2, send R2 and move to the R2-SENT state; both hosts receive R2 and drop it; after a timeout, both hosts move back to UNASSOCIATED and start the same process form the beginning.

Problem 7b: If the timeout transition is changed to lead to UNASSOCIATED instead of ESTABLISHED, a livelock may occur where neither party ever reaches ESTABLISHED.

We can resolve this problem by creating an asymmetry between the participants. A convenient place to do that is when both hosts are in the I2-SENT state and both receive an I2 packet. One of the hosts should continue in the Initiator role and the other should assume the Responder role. (In the current state-machine specification,

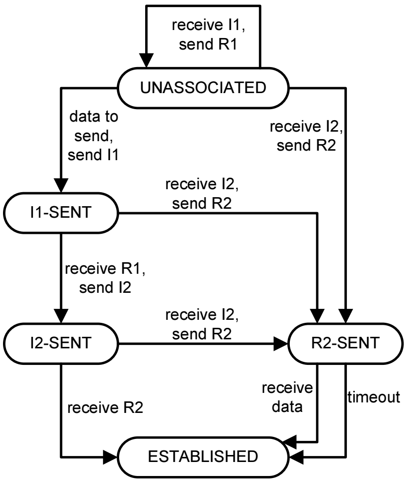

Figure 2 Partial protocol state machine.

both move to R2-SENT, i.e., select the Responder role.) One way to assign the roles to the hosts is to compare their HITs as if they were integers. We have (arbitrarily) decided to make the host with lower HIT the Initiator.

Proposed solution 7:
(a) The timeout transition from the R2-SENT state should lead to the UNASSOCIATED state.
(b) In I1-SENT, if a host receives I1 and the local HIT is lower than the peer HIT, the host should drop the received I1 and remain in I1-SENT. Otherwise, the host should process the received I1, send R1, and remain in I1-SENT.
(c) In I2-SENT, if a host receives I2 and the local HIT is lower than the peer HIT, the host should drop the received I2 and remain in I2-SENT. Otherwise, the host should process the received I2, send R2, and go to R2-SENT.

This way, both hosts cannot end up in the R2-SENT state at the same time and liveness of the protocol is guaranteed unless there is repeated message loss. The difference between solutions 7(b) and 7(c) is that 7(b) saves two messages but 7(c) is more robust in the presence of middle boxes. We suggest implementing both.

Above, we have only considered the protocol states as listed in the HIP specification. In reality, the state also comprises the SPI values, HIP and ESP transforms, and cryptographic keys. In the following, we will consider how these values are treated in the state transitions, in particular, when two handshakes collide. The Diffie-Hellman keys and the HIP and ESP transforms are selected together and can be considered parts of the same data structure. We will only discuss the keys but the same considerations apply to the transforms as well. If the nonces I and J are used in the session-key generation (see Section 3), we also need to consider the nonces a part of the protocol state.

It is not specified when the Diffie-Hellman keys should be replaced. This is particularly a problem if the handshakes collide and new Diffie-Hellman keys are generated during the exchange. In that case, the hosts may end up with inconsistent views of the keys. Specifically, if a host receives I2 in the I2-SENT state, which *peer* Diffie-Hellman key should it use for computing KEYMAT, the one from the just-arrived I2 or the one received earlier in R1? And which *local* Diffie-Hellman key should it use, the one it sent in I2 or the one it sent earlier in R1? The consistency problem is even more acute for the nonces I and J because they are always fresh. It turns out that if the hosts behave symmetrically, as a straight-forward implementation would do, any choice of keys would be wrong.

Problem 8: It is not specified which Diffie-Hellman keys (and nonces) a host should use to compute the session key if it receives I2 in the I2-SENT state. If the hosts behave symmetrically, they may end up with different session keys.

We assume that the proposed solutions 7(a) and (c) are implemented by all hosts but 7(b) only by some. The following rules can then be used to select consistent keys and nonces.

Proposed solutions 8: In the I2-SENT state, if a host receives I2 and the local HIT is lower than the peer HIT, the host should use the *peer* Diffie-Hellman key and nonce I from the R1 packet it received earlier. It should also use the *local* Diffie-Hellman key and nonce J from the I2 packet it sent earlier. Otherwise, it should use the *peer* Diffie-Hellman key and nonce J from the just-received I2, and the *local* Diffie-Hellman key and nonce I from the R1 packet it sent earlier.

A similar confusion occurs with the SPI values in the unmodified protocol. It is not specified which SPI values should be used to create the IPsec SAs if handshakes collide. It is possible that the hosts end up with different pairs of SPIs. The ambiguity is resolved by either one of the protocol changes 7(b) and 7(c).

We formalized the protocol state machine, including the Diffie-Hellman keys and SPIs, and used the Zing model checker to verify deadlock-freeness and consistency of the keys, nonces and SPIs after our proposed changes to the protocol. Further work is required to verify the absence of livelocks.

8 Conclusion

In this paper, we analyzed the security of the HIP base exchange protocol. The base exchange is a fairly basic authenticated Diffie-Hellman key exchange that is further simplified by the fact that the host identities in HIP are self-certifying. We did not find major attacks against the authentication and key-exchange apart from a small issue with the freshness of the Diffie-Hellman keys.

The more interesting security properties in the HIP base exchange relate to denial-of-service: the protocol uses client puzzles with several novel details to protect against resource-exhaustion DoS attacks. We showed that these details require modification to provide the intended protection. In particular, an attacker was able trick the honest Initiator into solving the wrong puzzles, and it was able to consume the puzzles of the honest Initiator by solving them first. Moreover, the protocol state machine requires changes to prevent deadlocks and livelocks.

Our analysis of the abstract protocol complements in an important way the detailed protocol design, specification, implementation, and testing that happens during the IETF standardization process. It should be remembered that HIP is as much a multi-addressing and mobility protocol as a security protocol. We suggested solutions to all the discovered problems and believe that the proposed protocol changes fit well into the HIP framework without compromising its original goals.

References

1. Bernard Aboba and William Dixon. IPsec-network address translation (NAT) compatibility requirements. RFC 3715, IETF, March 2004.
2. Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In Proc. CONCUR 2004, volume 3170 of LNCS. Springer, August 2004.
3. Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In Proc. Security Protocols Workshop 2000, volume 2133 of LNCS, pages 170-181, Cambridge, UK, April 2000. Springer.
4. Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Advances in Cryptology - Proc. CRYPTO '92, volume 740 of LNCS, pages 139-147, Santa Barbara, CA USA, August 1992. Springer.
5. Kjeld Borch Egevang and Paul Francis. The IP network address translator (NAT). RFC 1631, IETF, May 1994.
6. Ned Freed. Behavior of and requirements for Internet firewalls. RFC 2979, IETF, October 2000.
7. Dan Harkins and Dave Carrel. The Internet key exchange (IKE). RFC 2409, IETF Network Working Group, November 1998.
8. Charlie Kaufman (Ed.). Internet key exchange (IKEv2) protocol. Internet-Draft draft-ietf-ipsec-ikev2-17, IETF IPsec WG, September 2004. Work in progress.
9. Stephen Kent and Randall Atkinson. IP encapsulating security payload (ESP). RFC 2406, IETF, November 1998.
10. Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas R. Henderson. Host identity protocol. Internet Draft draft-ietf-hip-base-01, IETF HIP WG, October 2004. (http://www.watersprings.org/pub/id/draft-ietf-hip-base-01.txt)
11. Aarthi Nagarajan. Security issues of locator-identifier split and middlebox traversal for future Internet architectures. Master's thesis, Technische Universität Hamburg-Harburg, Germany, November 2004.
12. Pekka Nikander, Jukka Ylitalo, and Jorma Wall. Integrating security, mobility, and multi-homing in a HIP way. In Proc. NDSS '03, pages 87-99, San Diego, CA USA, February 2003.
13. Radia Perlman and Charlie Kaufman. Key exchange in IPSec: Analysis of IKE. IEEE Internet Computing, 4(6):50-56, November/December 2000.
14. Hannes Tschofenig, Aarthi Nagarajan, Vesa Torvinen, Jukka Ylitalo, and Murugaraj Shanmugam. NAT and firewall traversal for HIP. Internet-Draft draft-tschofenig-hiprg-hip-natfw-traversal-00, October 2004. Work in progress.
15. Hannes Tschofenig, Aarthi Nagarajan, Murugaraj Shanmugam, Jukka Ylitalo and Andrei Gurtov. Traversing Middle Boxes with Host Identity Protocol. Proc. ACISP '05, July 2005, Brisbane, Australia.