



## **Interoperability Problems Still Common in IKE**

*Short Diffie-Hellman Secrets Can Break IKE Interoperability*



QA Cafe © 2005  
<http://www.qacafe.com>

# Interoperability Problems Still Common in IKE

## *Short Diffie-Hellman Secrets Can Break IKE Interoperability*

### **Summary**

---

Although IKE has been out for a long time, and considered by the Internet community to be a mature protocol, there are still low-level problems affecting interoperability. Specifically at issue here is how an implementation represents internal keying material that is shorter than expected. If implementations do not handle the keying material in the same fashion, interoperability problems will occur.

While some vendors have successfully worked around these issues, problems still linger in even the most current revisions of products. This paper takes an in-depth look at the cause of the problem, and offers ways to detect it in existing implementations using the CDRouter-IKE test suite from QA Cafe.

### **The Problem**

---

IKE employs the Diffie-Hellman key agreement protocol to choose a shared key used in protecting Phase 1 (and Phase 2 when using PFS) exchanges. For mathematical reasons we will not get into, the shared key is expected to have the same number of bits as the Diffie-Hellman group used to generate it. This is not always the case.

There is a small chance that in some instances, the shared key can be several bits shorter than expected. Since it is used inside a hashing function to generate keying material, it is very important that interoperable implementations handle these short keys the same way.

From RFC 2409 - section 5:

```
The Diffie-Hellman public value passed in a KE payload, in either a phase 1 or phase 2 exchange, MUST be the length of the negotiated Diffie-Hellman group enforced, if necessary, by pre-pending the value with zeros.
```

RFC 2409 makes no mention of the shared secret or what to do in the case where it is shorter than the Diffie-Hellman group. Should it be padded up to the expected length like the public value is?

## Some Discussion

---

A thread posted to the IPsec mailing list ([ipsec@lists.tislabs.com](mailto:ipsec@lists.tislabs.com)) in April of 1999 brought up the topic of how to handle a short shared secret. Archived at [www.sandelman.ottawa.on.ca/ipsec/1999/04/threads.html](http://www.sandelman.ottawa.on.ca/ipsec/1999/04/threads.html) in the discussion, titled: “representation of IKE DH shared secret” IPsec mailing list participant Tero Kivinen suggested the following rewrite of the specification:

```
gxy is the Diffie-Hellman shared secret. When this value is included in the SKEYID calculation as a input for PRF it MUST be prepended with zero bits up to 8-bit boundary, so that it has same length in octets [sic] than group prime number p. When this value is used in the hash calculation it MUST be in network byte order.
```

From: [www.sandelman.ottawa.on.ca/ipsec/1999/04/msg00092.html](http://www.sandelman.ottawa.on.ca/ipsec/1999/04/msg00092.html)

Dan Harkins, co-author of the IKE RFC 2409, agreed with Kivinen in a reply to the mailing list. Saying,

```
Sounds good. I will incorporate this and similar verbage on EC into the document.
```

From: [www.sandelman.ottawa.on.ca/ipsec/1999/04/msg00097.html](http://www.sandelman.ottawa.on.ca/ipsec/1999/04/msg00097.html)

The IKE RFC was last updated in November 1998 and this discussion didn't occur until April of 1999. That is most likely the reason the updated wording never made it into the RFC.

Further support for this position can be found in RFC 2631 (Diffie-Hellman Key Agreement Method) under section 2.1.2 where it discusses the shared key ZZ:

```
Leading zeros MUST be preserved, so that ZZ occupies as many octets as p. [RFC2613]
```

At the time, early IKE Bakeoffs were discovering that all the interoperable implementations pre-pended zeros to short shared keys prior to using them in the hash function. They made sure that the shared key followed the same length requirements as KE did in the RFC.

## Some Math

---

What follows is a discussion of the underlying math that the Diffie-Hellman protocol is based on. If math is not your thing, you may choose to skip this section and continue straight to the next section.

At its core, the Diffie-Hellman key agreement protocol is nothing more than a bit of multiplication. To quickly demonstrate the underlying principles, consider the following situation between Alice and Bob.

Alice and Bob each choose their own separate private values. Alice chooses  $a$  and Bob chooses  $b$ . They have already agreed on values for parameters  $g$  and  $p$ . Both parties compute their public keys ( $Y_a$  and  $Y_b$  respectively) as follows:

$$\begin{array}{ll} \text{alice public key:} & Y_a = g^a \text{ mod } p \\ \text{bob public key:} & Y_b = g^b \text{ mod } p \end{array}$$

They may safely exchange the values of  $Y_a$  and  $Y_b$  in the clear, without any concern about compromising the secrecy of  $K$ , the shared secret key. Once they have exchanged their public keys, Alice computes:

$$K = Y_b^a \text{ mod } p$$

And Bob:

$$K = Y_a^b \text{ mod } p$$

Since  $g^{ab} = g^{ba}$ , Alice and Bob are each able to compute the same value for  $K$ .

In order for the math to protect our shared key  $K$ , the choices for  $a$ ,  $b$ , and  $p$  are very important. They need to be Big Numbers. Bigger than the majority of programming languages can support by themselves. Thankfully, we have some help in that area.

## OpenSSL to the Rescue

---

The folks working on the OpenSSL project have given us not only a library that helps compute these Big Numbers, but an entire API that takes care of the Diffie-Hellman key agreement protocol.

First, we look at the DH group of functions provided by OpenSSL. These functions implement the Diffie-Hellman key agreement protocol, and handle all related exponentiation and factoring.

At the heart of OpenSSL's Diffie-Hellman support is a struct that has the following members: (There is more to it than this, but these are the ones important to this discussion)

```
struct DH {
    BIGNUM *p;
    BIGNUM *g;
    BIGNUM *pub_key;
    BIGNUM *priv_key;
}
```

Look familiar? The DH struct holds all of the necessary information to compute and share Diffie-Hellman keys. We install values for p and g, and OpenSSL takes care of the rest.

```
DH *dh;
dh = DH_new();
dh->p = p;
dh->g = g;
DH_generate_key(dh);
```

DH\_generate\_key produces values for the private-key which we keep to ourselves, and the public-key which we share with the other peer.

Once the IKE peer responds with their own public-key value (pub), we can use it to compute the shared key. This time we put our private-key (priv) from above into the struct as well.

```
int i = 0;
dh->p = p;
dh->g = g;
dh->priv_key = priv;

i = DH_compute_key(shared_key, pub, dh);
```

The variable shared\_key should be an unsigned char\* that will hold the secret shared-key. The length of shared\_key in bytes is the return value of DH\_compute\_key, stored in the variable i.

As was mentioned earlier, this shared key should be the same length as the Diffie-Hellman group that was used to compute it. In our example, that group means the Prime Number p. OpenSSL provides us with another function useful here: DH\_size(dh) which returns the *expected* length in bytes of the shared key.

What is very important to note, is that DH\_size() doesn't know what the shared key will actually be, just how long it is expected to be. When we use DH\_compute\_key() to actually handle the math, its return value (i in our example) will be the *actual* length of the shared key.

## **What's The Significance?**

---

When OpenSSL computes a shared key, it only counts the significant bits in that number. Consider using the 768-bit prime number defined in Oakley group 1 in the Diffie-Hellman protocol. The resulting shared key should also have 768 bits. However, generating a key with some specific number of bits doesn't always set the highest bits in that key. There is a 1/256 chance that the result can have no significant bits set in the first byte of the bit stream. Statically, 1 in 256 keys will be short. Because OpenSSL may not include bytes with leading zeros, the shared key may not be the expected length.

## **What's an Implementation To Do?**

---

At this point we have the secret key stored in the variable `shared_key`, and from `DH_compute_key()` we have the length of that key stored in the variable `i`. As we now know, the actual length of the shared key is not always the same as the expected length.

It is important for the implementation to realize that the key may be shorter than expected, and compensate by pre-pending zeros to the key before applying a hash function to it.

Since the public transmitted in KE is also required to have zeros pre-pended to it, it is suggested that the same method used there is applied to the shared key to enforce its length.

## **A Way Around**

---

Because this problem has a fairly good probability of occurring, it is not uncommon. Several vendors have managed to avoid the leading-zero problem altogether. In the middle of the DH-Key negotiation process if a private key that is shorter than the Diffie-Hellman group is calculated, the current Phase 1 or Phase 2 exchange is abandoned and the device will reinitiate a new one. The chances of the next exchange also having a troublesome key are much lower, so this second attempt should succeed, and does not negatively affect the overall establishment of the IKE phase.

## **Is My Device Affected?**

---

This problem can be difficult to diagnose in existing implementations. Many vendors test their devices back-to-back with the same identical device. In that scenario, both

devices will use the same short key to generate the same keying material, and the connection will always succeed.

If the device is tested against another implementation that correctly adds leading zeros to a short shared key, then the keying material they use for encryption of future IKE payloads will not match. Upon receipt by the other side, the payload will not be able to be decrypted. This can appear in logs and test results as anything from a PAYLOAD-MALFORMED to an INVALID-PAYLOAD-TYPE error. Errors such as those can easily be attributed to a mangled or fragmented packet or other common network interference.

Typically in that situation, the packet is dropped, and the Phase 1 or 2 negotiation is restarted eventually. Usually sooner than later, and the VPN connection is established. Since the problem is actually in the internal representation of the shared key, it is difficult to track down. Combined with the infrequency of short keys, this error could easily slip through interoperability testing.

Testing the device against a tool that is able to highlight specific decryption errors and the related keying material will help greatly in tracking down this problem. It is also helpful if the device has a log that displays internal variables for debugging purposes.

## **The Solution**

---

QA Cafe's CDRouter-IKE test suite provides a straight forward testing environment for detecting the DH-key problem. Prior to the release of QA Cafe's CDRouter-IKE test suite, automatic testing for DH-key issues was not available. CDRouter-IKE now offers a quick and deterministic way to verify if an implementation is at risk for DH-key issues.

CDRouter-IKE offers over 50 unique test cases that range from basic IKE behavior to specific interoperability problems such as the DH-key issue. During IKE Phase 1 and 2 exchanges, once CDRouter-IKE learns the peer's public key, it searches for a corresponding key pair that will produce a zero-leading private key. This allows CDRouter-IKE to force this condition to occur and quickly verify if an IKE implementation is at risk.

## More Information / References

---

CDRouter-IKE <http://www.qacafe.com/show/ike>

*Network Security Essentials 2nd Edition*. William Stallings c.2003  
Pearson Education Inc. p.75

RSA Security - <http://www.rsasecurity.com/rsalabs/node.asp?id=2248>

Modular Exponentiation on Wikipedia -  
[http://en.wikipedia.org/wiki/Modular\\_exponentiation](http://en.wikipedia.org/wiki/Modular_exponentiation)

[RFC2631] <http://www.faqs.org/rfcs/rfc2631.html>

## NOTICE

CDRouter™ is a registered trademark of QA Cafe. With out the prior written consent of QA Cafe, no part of the document may be reproduced by any means.

## Additional Information

For additional information, contact:

QA Cafe  
155 Fleet Street  
Portsmouth, NH 03801  
Tel: (877) 332 0784  
Email: [info@qacafe.com](mailto:info@qacafe.com)