

# 5

---

## Key Exchange

---

### 5.1 Introduction

The major problem of secret-key encryption is how to pass the key between Bob and Alice, without Eve listening (Figure 5.1). The two main methods for this is to either use a key exchange protocol (such as Diffie-Hellman) or to encrypt the key with a public key and pass it to the other side, and use the private key to decrypt it. With Diffie-Hellman we use the difficulty of solving discrete logarithms, and where we have to solve for  $x$  (which is the discrete logarithm of  $y$  with respect to a base  $g$  modulo  $p$ ):

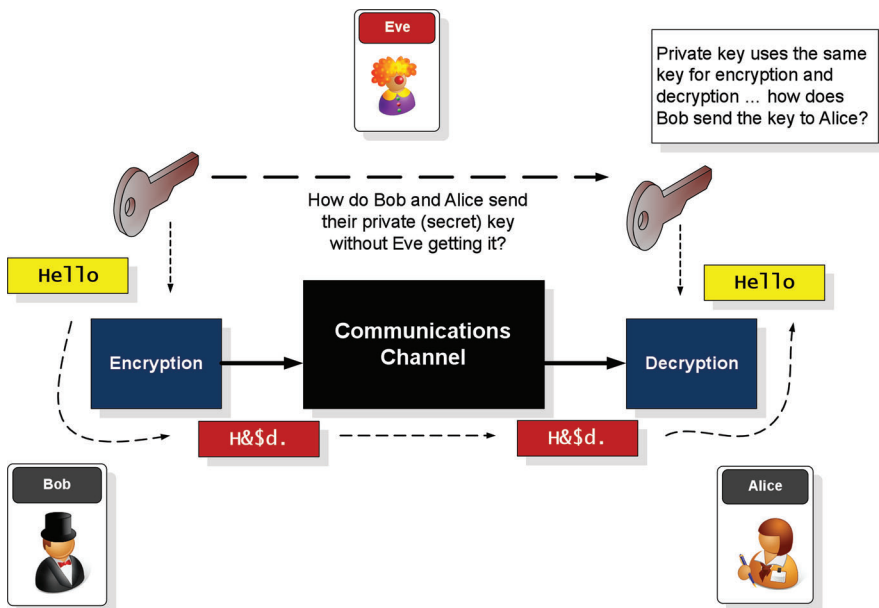
$$y = g^x \mod p$$

With public key methods, we use the difficulty of the factorising a value into its prime number factors, or use elliptic curve methods. Overall for Bob and Alice to generate a symmetric key (a secret key), they assume that Eve is listening to their communications, and talk openly, and where at the end of the key negotiation they will have the same secret key, but Eve, even though she has been listening, will not.

Bob could thus communicate openly Alice, and end up with an agreed secret key, but how does Bob actually know that he is communicating with Alice, and that the key they negotiate is the same? As we will see in a later chapter, we often have to use key pairs as part of the identification process, as we need to check the identity of one or more of the entities involved in the key exchange process. These key pairs (a public and a private key) can either static, and which are created from a trusted source, such as from a trusted digital certificate, or which can be generated for each connection. From a trust point-of-view the ones created from the trusted source are more likely to be trusted, but as they are static, a leakage of the private key could compromise any of the key exchanges created with the key pair. Another method is to create the key pairs when a secret key is required, and where a new key is created for each key exchange. This is typically defined as a session key.

In some situations we need to make sure that we are connecting to a trusted end source, as Eve could break the communication process, and become a man-in-the-middle. We thus often validate at least one of the entities involved in the negotiation. This is normally achieved through one side passing its public key, and to sign some data with their private key, and where the other side verifies that it has signed the data with the required private key. Normally it is the server which proves its identity to the client.

An important concept within key exchange is the usage of **forward secrecy** (FS), which means that a compromise of the long-term keys will not compromise any previous session keys. For example if we send the public key of the server to the client, and then the client sends back a session key for the connection which is encrypted with the public key of the server, then the server will then decrypt this and determine the session key. A leakage of the public key of the server would then cause all the sessions which used this specific public key to be compromised. FS thus aims to overcome this by making sure that all the sessions keys could not be compromised, even though the long-term key was compromised.



**Figure 5.1** Passing the secret key.

Another major concept is where the key is **ephemeral**. With some key exchange methods the same key will be generated if the same parameters are used on either side. This can cause problems as an intruder could guess the key, or even where the key was static and never changed. With ephemeral methods, a different key is used for each connection, and, again, the leakage of any long-term key would not cause all the associated session keys to be breached.

## 5.2 Diffie-Hellman Key Exchange

The problem of creating a shared symmetric key over a public network was solved by Whitfield Diffie in 1975, who created the Diffie-Hellman method. With this method, **Bob** and **Alice** generate two random values, and perform some calculations (Figure 5.2 and Figure 5.3), and pass the result of the calculations to each other. We first use two shared values ( $g$  – a generator, and  $p$  – a prime number). Bob then generates a random number ( $x$ ) and Alice generates a random number ( $y$ ). Next:

Bob computes  $A = g^x \bmod p$

Alice computes  $B = g^y \bmod p$

Bob sends  $A$  to Alice, and Alice sends  $B$  to Bob. The result becomes:

Bob computes  $\text{Key} = B^x \bmod p$

Alice computes  $\text{Key} = A^y \bmod p$

This will give be the same shared key (which is  $g^{xy} \bmod p$ ).

The basics of the operation is that we agree on the generator ( $g$ ) and a prime number ( $p$ ), which are agreed by both Bob and Alice (Figure 5.4). Alice and Bob generate their values ( $a$  and  $b$ ), and where Alice passes  $g^a \bmod p$  and Bob passes  $g^b \bmod p$ . Once they raise the received values to their random value that they have created, they will end up with the same shared key ( $g^{ab} \bmod p$ ).



Once these values have been received at either end, Bob and Alice will have the same secret key, which Eve cannot compute (without extensive computation). Diffie-Hellman is used in many applications, such as in VPNs (Virtual Private Networks), SSH, and secure FTP. The following shows a trace of a connection to a secure FTP site:

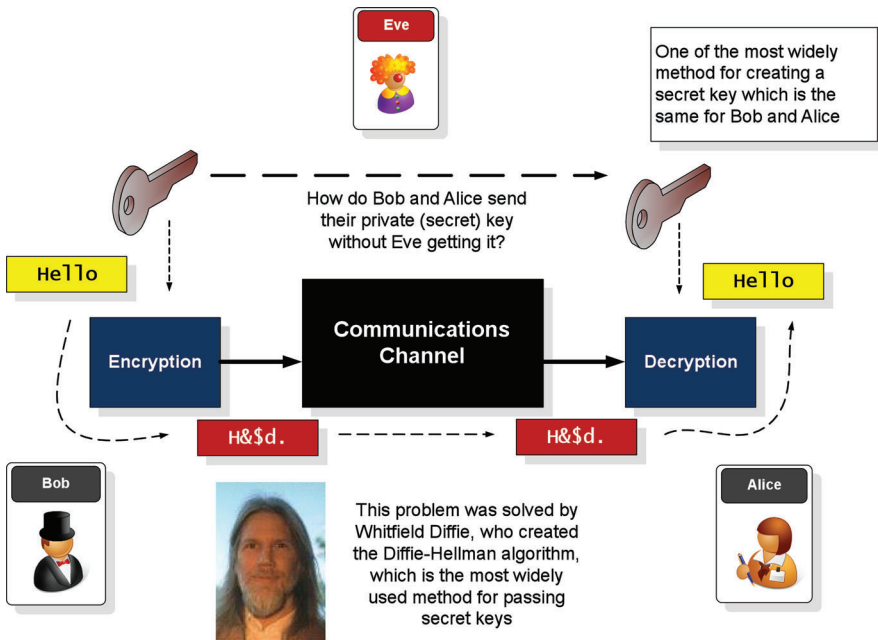
```

STATUS:> Initializing SFTP21 module...
STATUS:> Resolving host name mysite.com...
STATUS:> Host name mysite.com resolved: ip = 1.2.3.4.
STATUS:> Connecting to SFTP server ftp1.napier.ac.uk:22 (ip = 1.2.3.4)
        Key Method: Diffie-Hellman-group1-SHA1
        Host Key Algorithm: SSH-RSA
        Session Cipher: 192 bit TripleDES-cbc
        Session MAC: HMAC-MD5
        Session Compressor/Decompressor: ZLIB
STATUS:> Getting working directory...
STATUS:> Home directory: /home/test

```

Where it can be seen that this is a secure FTP transaction, the **encryption** being used is **3DES** (TripleDES), the message **authentication** method is **HMAC-MD5** and the **key exchange** is Diffie-Hellman. Overall Diffie-Hellman has three groups: Group 1, Group 3 or Group 5, which vary in the size of the prime number used.

 **Web link (Diffie-Hellman):** <http://asecuritysite.com/encryption/diffie>  
 **Web link (Diffie-Hellman real):** <http://asecuritysite.com/encryption/diffie2>



**Figure 5.2** Diffie-Helman method.

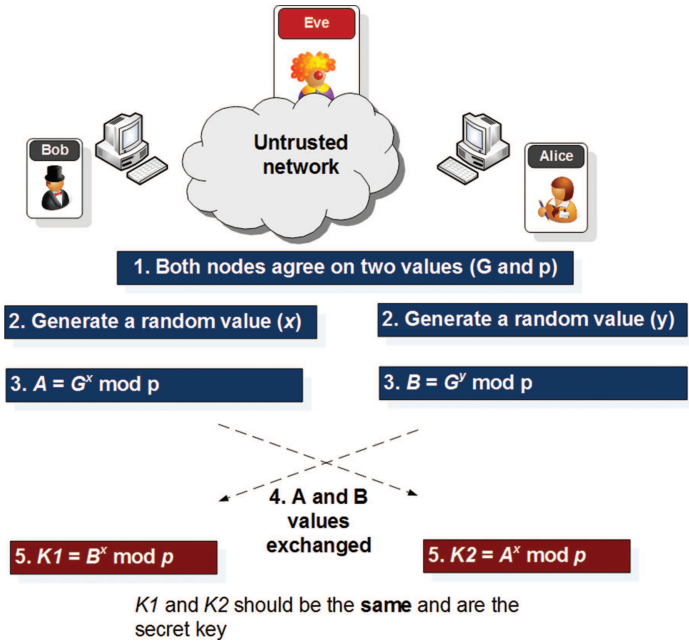


Figure 5.3 Diffie-Hellman process.

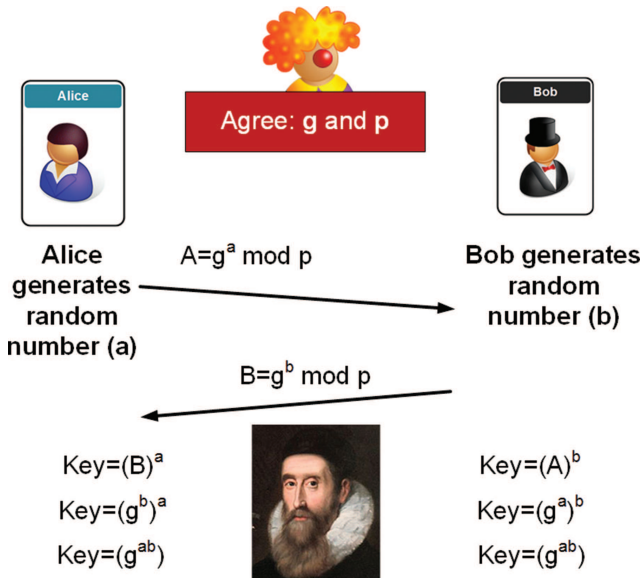


Figure 5.4 Outline of Diffie-Hellman process.

## 172 *Key Exchange*

A simple Python program to calculate small values of  $G$  and  $n$  is:

```
import random
import base64
import hashlib

g=11
p=1001

x=random.randint(5, 10)
y=random.randint(10,20)

A=(g**x) % p
B=(g**y) % p

print 'g: ',g,' (a shared value), n: ',p,' (a prime number)'

print '\nAlice calculates:'
print 'a (Alice random): ',x
print 'Alice value (A): ',A,' (g^a) mod p'

print '\nBob calculates:'
print 'b (Bob random): ',y
print 'Bob value (B): ',B,' (g^b) mod p'

print '\nAlice calculates:'
keyA=(B**x) % p
print 'Key: ',keyA,' (B^a) mod p'
print 'Key: ',hashlib.sha256(str(keyA)).hexdigest()

print '\nBob calculates:'
keyB=(A**y) % p
print 'Key: ',keyB,' (A^b) mod p'
print 'Key: ',hashlib.sha256(str(keyB)).hexdigest()
```

which gives a sample run of:

```
g:  11  (a shared value), n:  1001  (a prime number)

Alice calculates:
a (Alice random):  7
Alice value (A):  704  (g^a) mod p

Bob calculates:
b (Bob random):  16
Bob value (B):  627  (g^b) mod p

Alice calculates:
Key:  627  (B^a) mod p
```

```

Key: 9a35532c7499c19daeacafc961657409c7280ce59d7ae1a3606dd638
    ac3d99ec

Bob calculates:
Key: 627 (A^b) mod p
Key: 9a35532c7499c19daeacafc961657409c7280ce59d7ae1a3606dd638
    ac3d99ec

```

## 5.3 Creating the Generator

The value of  $g$  must be selected so that every value of  $x$  gives a unique value ( $Y$ ) for a given prime number ( $p$ ):

$$Y = g^x \mod p$$

What we want is that each value of  $x$  that we choose should give us a unique value of  $Y$  (obviously between 0 and  $p-1$ ), so if we have  $g = 3$  and  $p = 5$ , then we can compute the  $Y$  values of:

$$\begin{aligned}
 3^1 \mod 5 &\rightarrow 3 \\
 3^2 \mod 5 &\rightarrow 4 \\
 3^3 \mod 5 &\rightarrow 2 \\
 3^4 \mod 5 &\rightarrow 1
 \end{aligned}$$

and which is known as a cyclic group ( $Z_p$ ) where each value is unique as an output (up to a value at  $p-1$ ). We can create these values with the following Python code (for possible  $g$  values up to  $p-1$ ):

```

import sys
import random
p=11
for x in range (1,p):
    rand = x
    exp=1
    next = rand % p

    while (next <> 1 ):
        next = (next*rand) % p
        exp = exp+1

    if (exp==p-1):
        print rand

```

## 174 Key Exchange

For  $g$  values with  $p = 11$ , we get possible generators of 2, 6, 7 and 8. We can get this by also calculating the values for  $g^x \bmod p$  for different generator values and  $x$  values, but using  $p = 11$ . In this case we see that  $g = 3$ ,  $g = 4$ ,  $g = 5$  and  $g = 9$  have repeated values as an output, so they cannot be used:

p	11							
Generator	2	3	4	5	6	7	8	9
x	$g^x \bmod p$	$g^x \bmod p$	$g^x \bmod p$	$g^x \bmod p$	$g^x \bmod p$	$g^x \bmod p$	$g^x \bmod p$	$g^x \bmod p$
2	4	9	5	3	3	5	9	4
3	8	5	9	4	7	2	6	3
4	5	4	3	9	9	3	4	5
5	10	1	1	1	10	10	10	1
6	9	3	4	5	5	4	3	9
7	7	9	5	3	8	6	2	4
8	3	5	9	4	4	9	5	3
9	6	4	3	9	2	8	7	5
10	1	1	1	1	1	1	1	1

The strength of the Diffie-Hellman method normally relates to the size of the prime number bases which are used in the key exchange, where Group 5 uses a 1,536-bit prime number, Group 2 uses a 1,024-bit prime, and Group 1 uses a 768-bit prime number. In the following we use Openssl to create a 768-bit key:

```
C:\> openssl dhparam -out dhparams.pem 768 -text

C:\> type dhparams.pem
Diffie-Hellman-Parameters: (768 bit)
  prime:
    00:d0:37:c2:95:64:02:ea:12:2b:51:50:a2:84:6c:
    71:6a:3e:2c:a9:80:e2:65:b2:a5:ee:77:26:22:31:
    66:9e:fc:c8:09:94:e8:9d:f4:cd:bf:d2:37:b2:fb:
    b8:38:2c:87:28:38:dc:95:24:73:06:d3:d9:1f:af:
    78:01:10:6a:7e:56:4e:7b:ee:b4:8d:6b:4d:b5:9b:
    93:c6:f1:74:60:01:0d:96:7e:85:ca:b8:1f:f7:bc:
    43:b7:40:4d:4e:87:e3
  generator: 2 (0x2)
-----BEGIN DH PARAMETERS-----
MGYCYQDQN8KVZALqEitRUKKEbHFqPiypgOJlsqXudyYiMWae/MgJlOid9M2/0jey
+7g4LIcoNyVJHMG09kfr3gBEGp+Vk577rSNa021m5PG8XRgAQ2WfoXKuB/3vEO3
QE1Oh+MCAQI=
-----END DH PARAMETERS-----
```

In this case we use the value of 2 for the generator (which is often the default value for the generator), but can also use a value of 5 (using the “-5” option). For a  $g$  value of 2, and a prime number ( $p$ ) of 11, we get (safe  $g$  values are 2, 6, 7 and 8):



$$\begin{array}{ll}
2^1 \bmod 11 = 2 & 2^2 \bmod 11 = 4 \\
2^3 \bmod 11 = 8 & 2^4 \bmod 11 = 5 \\
2^5 \bmod 11 = 10 & 2^6 \bmod 11 = 9 \\
2^7 \bmod 11 = 7 & 2^8 \bmod 11 = 3 \\
2^9 \bmod 11 = 6 & 2^{10} \bmod 11 = 1
\end{array}$$

If we use a generator ( $g$ ) of 5, and generate a 768-bit prime number, we get:

```

Diffie-Hellman-Parameters: (768 bit)
  prime:
    00:8b:48:7b:80:7c:fe:69:6a:a6:30:29:08:3b:e7:
    2b:c6:90:8b:68:63:6b:ff:ba:29:5a:52:9e:98:a7:
    d8:4a:1b:2f:fe:e6:35:e8:af:de:51:6b:5f:e8:2f:
    79:aa:6a:65:ed:85:64:99:ce:84:e3:b3:0c:37:77:
    47:78:d3:33:45:da:4e:0b:49:82:83:c1:7b:2a:c7:
    8d:11:8e:e2:7b:93:2c:85:46:62:6c:93:a5:25:88:
    3a:83:fd:fd:10:e5:f7
  generator: 5 (0x5)
-----BEGIN DH PARAMETERS-----
MGYCYQCLSHuAfP5paqYwKQg75yvGkItoY2v/uilaUp6Yp9hKGy/+5jXor95Ra1/o
L3mqamXthWSZzoTjsw3d0d40zNF2k4LSYKDwXsqx40RjuJ7kyyFRmJsk6UliDqD
/f0Q5fcCAQU=
-----END DH PARAMETERS-----

```

If we take a simple example of a generator of 5 we cannot use a prime number of 11, or 13, so let's use 17 (where safe generate values are 3, 5, 6, 7, 10, 11, 12 and 14):

$$\begin{array}{ll}
5^1 \bmod 17 = 5 & 5^2 \bmod 17 = 8 \\
5^3 \bmod 17 = 6 & 5^4 \bmod 17 = 13 \\
5^5 \bmod 17 = 14 & 5^6 \bmod 17 = 2 \\
5^7 \bmod 17 = 10 & 5^8 \bmod 17 = 16 \\
5^9 \bmod 17 = 12 & 5^{10} \bmod 17 = 9 \\
5^{11} \bmod 17 = 11 & 5^{12} \bmod 17 = 4 \\
5^{13} \bmod 17 = 3 & 5^{14} \bmod 17 = 15 \\
5^{15} \bmod 17 = 7 & 5^{16} \bmod 17 = 1
\end{array}$$

## 5.4 Diffie-Hellman Examples

Let's select a prime number of:

$$p = 11$$

## 176 *Key Exchange*

Safe values for  $g$  are 2, 6, 7 and 8, so let's select  $g = 7$ . Bob and Alice generate random numbers ( $x$  and  $y$ ):

$$x = 3 \quad y = 4$$

Bob calculates  $A$ :

$$A = g^x \pmod{p} = 7^3 \pmod{11} = 343 \pmod{11} = 2$$

Alice calculates  $B$ :

$$B = g^y \pmod{p} = 7^4 \pmod{11} = 2401 \pmod{11} = 3$$

They swap values and they generate the key:

$$\begin{aligned} \text{Key (Bob)} &= B^x \pmod{p} = 3^3 \pmod{11} = 27 \pmod{11} = 5 \\ \text{Key (Alice)} &= A^y \pmod{p} = 2^4 \pmod{11} = 16 \pmod{11} = 5 \end{aligned}$$

This is their shared key. As another example, let's select:

$$p = 3049$$

A safe value of  $p$  for 3,049 is  $g = 282$ . Bob and Alice generate random numbers ( $x$  and  $y$ ):

$$x = 21 \quad y = 6$$

Bob calculates  $A$ :

$$A = 282^{21} \pmod{3,049} = 438$$

Alice calculates  $B$ :

$$B = 282^6 \pmod{3,049} = 1,924$$

They swap values and they generate the shared key:

$$\begin{aligned} \text{Key (Bob)} &= 1924^{21} \pmod{3,049} = 2,736 \\ \text{Key (Alice)} &= 438^6 \pmod{3,049} = 2,736 \end{aligned}$$

## 5.5 Ephemeral Diffie-Hellman with RSA (DHE-RSA)

The problem with DH is that if Bob and Alice generate the same values each time, they will always end up with the same secret key. With Ephemeral Diffie-Hellman (DHE) a different key is used for each connection, and a leakage of the public key would still mean that all of the communications were secure. Within DHE-RSA, the server signs the Diffie-Hellman parameter (using a private key from an RSA key pair) to create a pre-master secret, and where a master key is created which is then used to generate a shared symmetric encryption key.

Normally when we create a shared key we create a tunnelled connection between a client and a server. This is normally defined through SSL (Secure Socket Layer) or TLS (Transport Layer Security), and where a client initially connects to a server. We then define the tunnel type (such as TLS or SSL), the key exchange method (such as DHE-RSA), a symmetric key method to be used for the encryption process (such as 256-bit AES with CBC) and a hashing method (such as SHA). This can be defined as a string as:


TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA

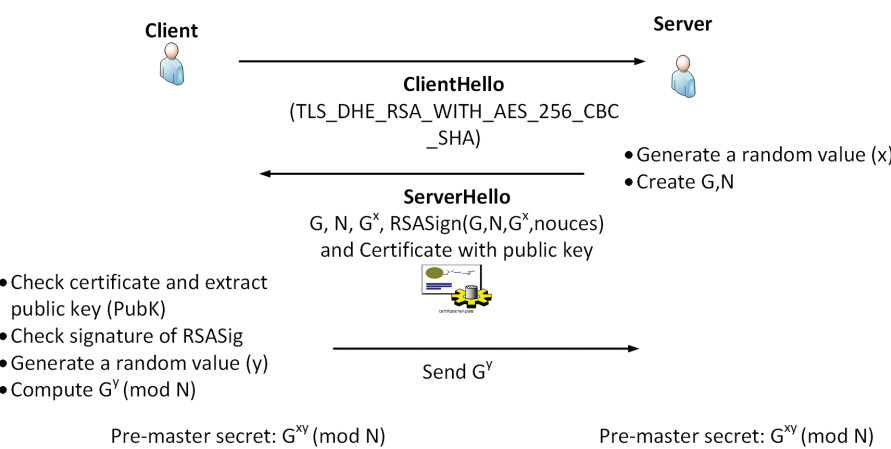
and is contained in a ClientHello message that goes from the client to the server. A ServerHello is then returned with the digital certificate of the server and which contains the public key of the server. A simplified handshake is defined in Figure 5.5 where the client sends the definition for a TLS cipher suite. In this case we are using a handshaking method of DHE-RSA, a 256-bit AES-CBC shared key, and with a SHA hash signature. The server will then generate a random value ( $x$ ) and create a value for  $g$  (the generator) and  $p$  (the prime number). Along with this the server will then generate  $g^x$  and take the  $g$ ,  $p$  and  $g^x$  parameters and encrypt them with the private key of the server. This creates a signature for the server.

Next the server will create a message with  $g$ ,  $p$ ,  $g^x$ , and the signature of nonces (random values) and the Diffie-Hellman parameters ( $g$ ,  $p$ , and  $g^x$ ). The server then sends this message with a digital certificate containing its public key. When the client receives it, it will check the certificate for its validity, and then extract the public key. The client then checks the signature by decrypting the signed value and checks it against the parameters already contained in the message ( $g$ ,  $p$ , and  $g^x$ ). If the values are the same, the server has been validated. Now, as with DH, the client will create a random value of  $y$ , and sends the value of  $g^y$  back to the server. The  $g^{xy}$  value will then be

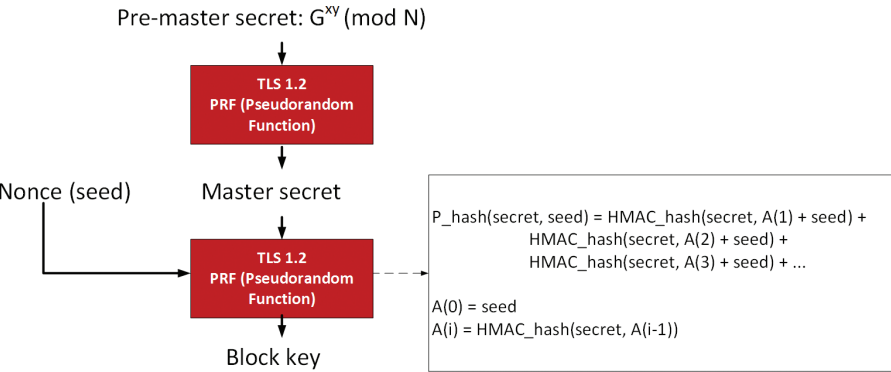
the pre-master secret. In this way the client knows that it has received valid values of the DH parameters, and can trust that the connection does not have a man-in-the-middle.

We now have a pre-master secret, as illustrated in Figure 5.6, which is shared by the client and server, and which can then be used to create a master key by using a PRF (Pseudorandom Function). In TLS 1.2 this is created using an HMCA-SHA256 hashed value (and which will generate a 256-bit key). To create the actual key used we feed the master key and the nonce into the PRF and generate the shared key for the session.

 **Web link (DHE-RSA):** <http://asecuritysite.com/dhe>



**Figure 5.5** Example DHE-RSA process.



**Figure 5.6** Key generation.

## 5.6 (Ephemeral) Elliptic Curve Diffie-Hellman (ECDHE)

ECDHE can be used to create a shared key between Bob and Alice. Initially, as with the Elliptic Curve method, we define some domain parameters, such as  $p$ ,  $a$ ,  $b$ , and  $G$  (see the previous chapter). From this Bob and Alice each generate an elliptic curve key pair, either with a static version where the keys have been generated by a trusted source, or with a dynamic method. We then have a private key ( $d$ ) which is a random number and where the public key ( $P$ ) is equal to  $dG$  (where  $G$  is a point on a defined elliptic curve). This gives us a public key for Alice ( $P_A$ ), and one for Bob ( $P_B$ ), along with a private key for Alice ( $d_A$ ) and Bob ( $d_B$ ).

Bob computes  $(x_k, y_k) = d_B P_A$   
 Alice computes  $(x_k, y_k) = d_A P_B$

The value of  $x_k$  then becomes the shared secret. The reason we can assume this is that:

$$d_A P_B = d_A d_B G = d_B d_A G = d_B P_A$$

The keys can either come from a digital certificate (which uses static keys) or can be ephemeral, and where the public and private keys are generated for each connection. Normally, too, the secret values ( $x_k$ ) are hashed in order to remove weaknesses within the Diffie-Hellman key exchange process.

## 5.7 Diffie-Hellman Weaknesses

Netscape first defined SSL (Secure Socket Layer) Version 1.0 in 1993, and eventually, in 1996, released a standard which is still widely used: SSL 3.0. While many in the industry used it, it did not become an RFC standard until 2011 (which was assigned RFC 6101). SSL has now been exposed by many problems including FREAK (“Factoring RSA Export Keys”) and was introduced to comply with US Cryptography Export Regulations, where the keys used for exportable software were limited to 512-bits or less (and were defined as RSA EXPORT keys – DHE\_EXPORT). The RFC states:

The server key exchange message is sent by the server if it has no certificate, has a certificate only used for signing (e.g., DSS [DSS] certificates, signing-only RSA [RSA] certificates), or FORTEZZA KEA key exchange is used. This message is not used if the server certificate contains Diffie-Hellman [DH1] parameters.

Note: According to current US export law, RSA moduli larger than 512 bits may not be used for key exchange in software exported from the US. With this message, larger RSA keys may be used as signature-only certificates to sign temporary shorter RSA keys for key exchange.

In 2015, a paper entitled *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice* – showed that it was fairly easy to precompute on values for two popular Diffie-Hellman parameters (and which use the DHE\_EXPORT cipher set). The research team found that one of them was used as a default in the around 7% of the Top 1 million web sites and was hard coded into the Apache httpd service. Overall, at the time, it was found that over 3% of Web sites were still using the default. The parameters were (where we see a generator value of 2):

```
Diffie-Hellman-Parameters: (512 bit)
prime:
  00:9f:db:8b:8a:00:45:44:f0:04:5f:17:37:d0:ba:
  2e:0b:27:4c:df:1a:9f:58:82:18:fb:43:53:16:a1:
  6e:37:41:71:fd:19:d8:d8:f3:7c:39:bf:86:3f:d6:
  0e:3e:30:06:80:a3:03:0c:6e:4c:37:57:d0:8f:70:
  e6:aa:87:10:33
generator: 2 (0x2)
```

Another group was found within the OpenSSL library (dh512.pem), and defined with:

```
Diffie-Hellman-Parameters: (512 bit)
prime:
  00:da:58:3c:16:d9:85:22:89:d0:e4:af:75:6f:4c:
  ca:92:dd:4b:e5:33:b8:04:fb:0f:ed:94:ef:9c:8a:
  44:03:ed:57:46:50:d3:69:99:db:29:d7:76:27:6b:
  a2:d3:d4:12:e2:18:f4:dd:1e:08:4c:f6:d8:00:3e:
  7c:47:74:e8:33
generator: 2 (0x2)
```

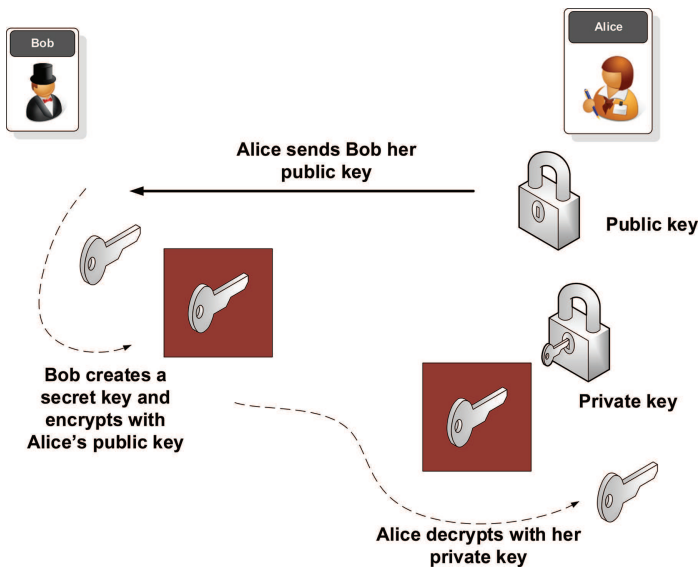
The DHE\_EXPORT Downgrade attack then involves forcing the key negotiation process to default to 512-bit prime numbers. For this the client only offers DHE\_EXPORT for the key negotiation, and the server, if it is setup for this, will accept it. The precomputation of 512-bit keys with  $g$  values of 2 and 5 (which are common) are within a reasonable time limits. The ways to overcome the problems are to:

- **Disable Export Cipher Suites.** With this we disable any negotiation of the key using export grade ciphers. This will have no effect on connections, as no existing version of Web browsers actually depend on export level ciphers.
- **Use (Ephemeral) Elliptic-Curve Diffie-Hellman (ECDHE).** This method uses a key exchange method based on an Elliptic-Curve Diffie-Hellman (ECDH) key exchange.
- **Uses a strong group.** With this we make sure that we are using the strong generation of a prime number which cannot be precomputing. Currently 2,048-bit prime numbers are recommended. A strong prime number is generated with “openssl dhparam -out dhparams.pem 2048”. Normally this will take a few minutes to compute, as it involves a random process, so can only be used to statically assign the parameters.

## 5.8 Using the Public Key to Pass a Secret Key

Diffie-Hellman methods have been used extensively to create a shared secret key, but suffers from a man-in-the-middle attack, where Eve sits in-between and passes the values back and forward, and negotiates two keys: one between Bob and Eve, and the other between Alice and Eve. An improved method is to use public key encryption, where Alice passes her public key to Bob, and then Bob creates an encryption key and encrypts this with Alice's public key. Alice then receives this and decrypts the key her the private key, to reveal the shared key. As Alice is the only one to have the private key to match the public key, so the method is secure (Figure 5.7). The major problem with this method is that a breach of Alice's private key would compromise all the previous key exchanges. Eve may also trick Bob with a fake public key for Alice.

With key exchange we typically have a time-out for the key to be used, after which time the key is renegotiated. This allows a smaller time window for Eve to determine the key. The FREAK (Factoring RSA Export Keys) vulnerability caused many problems as the negotiation used a 512-bit public key, where the 512-bit private key can be determined using graphic processors running in the Cloud. If Eve determines the private key associated with the public key, she can read all of the communications sent using the secret key. The key pair can thus be static (such as from a digital certificate which has been created from a trusted provider), or can be generated for each connection.



**Figure 5.7** Sharing a key using public-key encryption.

## 5.9 Lab/Tutorial

The lab and tutorial related to this chapter is available on-line at:

<http://asecuritysite.com/crypto05>