CHAPTER

11

Diffie-Hellman

For our discussion of public-key cryptography, we're going to follow the historical path. Public-key cryptography was really started by Whitfield Diffie and Martin Hellman when they published their article ''New Directions in Cryptography'' in 1976 [33].

So far in this book, we've only talked about encryption and authentication with shared secret keys. But where do we get those shared secret keys from? If you have 10 friends you want to communicate with, you can meet them all and exchange a secret key with each of these friends for future use. But like all keys, these keys should be refreshed regularly, so at some point you will have to meet and exchange keys all over again. A total of 45 keys are needed for a group of 10 friends. But as the group gets larger, the number of keys grows quadratically. For 100 people all communicating with each other, you need 4950 keys. Specifically, in a group of N people, we would need $N(N-1)/2$ keys. This quickly becomes unmanageable.

Diffie and Hellman posed the question of whether it would be possible to do this more efficiently. Suppose you have an encryption algorithm where the encryption and decryption keys are different. You could publish your encryption key and keep your decryption key secret. Anyone could then send you an encrypted message, and only you could decrypt it. This would solve the problem of having to distribute so many different keys.

Diffie and Hellman posed the question, but they could only provide a partial answer. Their partial solution is today known as the Diffie-Hellman key exchange protocol, often shortened to DH protocol [33].

The DH protocol is a really nifty idea. It turns out that two people communicating over an insecure line can agree on a secret key in such a way that both of them end up with the same key, without divulging it to someone who is listening in on their conversation.

11.1 Groups

If you've read the last chapter, it won't surprise you that primes are involved. For the rest of this chapter, p is a large prime. Think of p as being 2000 to 4000 bits long. Most of our computations in this chapter will be modulo p—in many places we will not specify this again explicitly. The DH protocol uses $\mathbb{Z}_p^*$, the multiplicative group modulo p that we discussed in Section 10.3.3.

Choose any g in the group and consider the numbers $1, g, g^2, g^3, \ldots,$ all modulo p, of course. This is an infinite sequence of numbers, but there is only a finite set of numbers in $\mathbb{Z}_p^*$. (Remember, $\mathbb{Z}_p^*$ is the numbers $1, \ldots, p-1$ together with the operation of multiplication modulo p.) At some point, the numbers must start to repeat. Let us assume this happens at $g^i = g^j$ with $i < j$. As we can do division modulo p, we can divide each side by g^i and get $1 = g^{j-i}$. In other words, there is a number $q := j - i$ such that $g^q = 1 \pmod{p}$. We call the smallest positive value q for which $g^q = 1 \pmod{p}$ the *order* of g. (Unfortunately, there is quite a bit of terminology associated with this stuff. We feel it is better to use the standard terminology than invent our own words; this will avoid confusion when reading other books.)

If we keep on multiplying gs, we can reach the numbers $1, g, g^2, \ldots, g^{q-1}$. After that, the sequence repeats as $g^q = 1$. We say that g is a generator and that it generates the set $1, g, g^2, \ldots, g^{q-1}$. The number of elements that can be written as a power of g is exactly q, the order of g.

One property of multiplication modulo p is that there is at least one g that generates the entire group. That is, there is at least one g value for which $q = p - 1$. So instead of thinking of $\mathbb{Z}_p^*$ as the numbers $1, \ldots, p-1$, we can also think of them as $1, g, g^2, \ldots, g^{p-2}$. A g that generates the entire group is called a *primitive element* of the group.

Other values of g can generate smaller sets. Observe that if we multiply two numbers from the set generated by g, we get another power of g, and therefore another element from the set. If you go through all the math, it turns out that the set generated by g is another group. That is, you can multiply and divide in this group just as you can in the large group modulo p. These smaller groups are called subgroups (see Section 10.3.3). They will be important in various attacks.

There is one last thing to explain. For any element g, the order of g is a divisor of $p - 1$. This isn't too hard to see. Choose g to be a primitive element. Let h be any other element. As g generates the whole group, there is an x such

that $h = g^x$. Now consider the elements generated by h. These are $1, h, h^2, h^3, \ldots$ which are equal to $1, g^x, g^{2x}, g^{3x}, \ldots$. (All our computations are still modulo p, of course.) The order of h is the smallest q at which $h^q = 1$, which is the same as saying it is the smallest q such that $g^{xq} = 1$. For any t, $g^t = 1$ is the same as saying $t = 0 \pmod{p-1}$. So q is the smallest q such that $xq = 0 \pmod{p-1}$. This happens when $q = (p-1)/\gcd(x, p-1)$. So q is obviously a factor of $p-1$.

Here's a simple example. Let's choose $p = 7$. If we choose $g = 3$ then g is a primitive element because $1, g, g^2, \ldots, g^5 = 1, 3, 2, 6, 4, 5$. (Again, all computations are modulo p.) The element $h = 2$ generates the subgroup $1, h, h^2 = 1, 2, 4$ because $h^3 = 2^3 \bmod 7 = 1$. The element $h = 6$ generates the subgroup $1, 6$. These subgroups have sizes 3 and 2 respectively, which are both divisors of $p-1$.

This also explains parts of the Fermat test we talked about in Section 10.4.1. Fermat's test is based on the fact that for any a we have $a^{p-1} = 1$. This is easy to check. Let g be a generator of $\mathbb{Z}_p^*$, and let x be such that $g^x = a$. As g is a generator of the entire group, there is always such an x. But now $a^{p-1} = g^{x(p-1)} = (g^{p-1})^x = 1^x = 1$.

11.2 Basic DH

For the original DH protocol, we first choose a large prime p, and a primitive element g that generates the whole group $\mathbb{Z}_p^*$. Both p and g are public constants in this protocol, and we assume that all parties, including the attackers, know them. The protocol is shown in Figure 11.1. This is one of the usual ways in which we write cryptographic protocols. There are two parties involved: Alice and Bob. Time progresses from the top to the bottom. First Alice chooses a random x in $\mathbb{Z}_p^*$, which is the same as choosing a random number in $1, \ldots, p-1$. She computes $g^x \bmod p$ and sends the result to Bob. Bob in turn chooses a random y in $\mathbb{Z}_p^*$. He computes $g^y \bmod p$ and sends the result to Alice. The final result k is defined as g^{xy}. Alice can compute this by raising the g^y she got from Bob to the power x that she knows. (High-school math: $(g^y)^x = g^{xy}$.) Similarly, Bob can compute k as $(g^x)^y$. They both end up with the same value k, which they can use as a secret key.

But what about an attacker? The attacker gets to see g^x and g^y, but not x or y. The problem of computing g^{xy} given g^x and g^y is known as the Diffie-Hellman problem, or DH problem for short. As long as p and g are chosen correctly, there is no known efficient algorithm to compute this. The best method known is to first compute x from g^x, after which the attacker can compute k as $(g^y)^x$ just like Alice did. In the real numbers, computing x from g^x is called the logarithm function, which you find on any scientific calculator. In the finite field $\mathbb{Z}_p^*$, it is called a *discrete logarithm*, and in general the problem of computing x from g^x in a finite group is known as the discrete logarithm problem, or DL problem.

Alice		Bob
$x \in_{\mathcal{R}} \mathbb{Z}_p^*$		
	$\xrightarrow{\quad g^x \quad}$	
		$y \in_{\mathcal{R}} \mathbb{Z}_p^*$
	$\xleftarrow{\quad g^y \quad}$	
$k \leftarrow (g^y)^x$		$k \leftarrow (g^x)^y$

Figure 11.1: The original Diffie-Hellman protocol.

The original DH protocol can be used in many ways. We've written it as an exchange of messages between two parties. Another way of using it is to let everybody choose a random x, and publish $g^x \pmod p$ in the digital equivalent of a phone book. If Alice wants to communicate with Bob securely, she gets g^y from the phone book, and using her x, computes g^{xy}. Bob can similarly compute g^{xy} without any interaction with Alice. This makes the system usable in settings such as e-mail where there is no direct interaction.

11.3 Man in the Middle

The one thing that DH does not protect against is the so-called man-in-the-middle attack.[1] Look back at the protocol. Alice knows she is communicating with somebody, but she does not know with whom she is communicating. Eve can sit in the middle of the protocol and pretend to be Bob when speaking to Alice, and pretend to be Alice when speaking to Bob. This is shown in Figure 11.2. To Alice, this protocol looks just like the original DH protocol. There is no way in which Alice can detect that she is talking to Eve, not Bob. The same holds for Bob. Eve can keep up these pretenses for as long as she likes. Suppose Alice and Bob start to communicate using the secret key they think they have set up. All Eve needs to do is forward all the communications between Alice and Bob. Of course, Eve has to decrypt all the data she gets from Alice that was encrypted with key k, and then encrypt it again with key k' to send to Bob. She has to do the same with the traffic in the other direction as well, but that is not a lot of work.

With a digital phone book, this attack is harder. As long as the publisher of the book verifies the identity of everybody when they send in their g^x, Alice knows she is using Bob's g^x. We'll discuss other solutions when we talk about digital signatures and PKIs later on in this book.

[1]The terminology may look similar, but a man-in-the-middle attack is different than a meet-in-the-middle attack from Section 2.7.2.

Alice		Eve		Bob
$x \in_{\mathcal{R}} \mathbb{Z}_p^*$				
	$\xrightarrow{g^x}$			
		$v \in_{\mathcal{R}} \mathbb{Z}_p^*$		
			$\xrightarrow{g^v}$	
				$y \in_{\mathcal{R}} \mathbb{Z}_p^*$
			$\xleftarrow{g^y}$	
		$w \in_{\mathcal{R}} \mathbb{Z}_p^*$		
	$\xleftarrow{g^w}$			
$k \leftarrow (g^w)^x$		$k \leftarrow (g^x)^w$		
		$k' \leftarrow (g^y)^v$		$k' \leftarrow (g^v)^y$

Figure 11.2: Diffie-Hellman protocol with Eve in the middle.

There is at least one setting where the man-in-the-middle attack can be addressed without further infrastructure. If the key k is used to encrypt a phone conversation (or a video link), Alice can talk to Bob and recognize him by his voice. Let h be a hash function of some sort. If Bob reads the first few digits of $h(k)$ to Alice, then Alice can verify that Bob is using the same key she is. Alice can read the next few digits of $h(k)$ to Bob, to allow Bob to do the same verification. This works, but only in situations where you can tie knowledge of the key k to the actual person on the other side. In most computer communications, this solution is not possible. And if Eve ever succeeds in building a speech synthesizer that can emulate Bob, it all falls apart. Finally, the biggest problem with this solution is that it requires discipline from the users, which is risky since users regularly ignore security procedures. In general, it is much better to have technical mechanisms for thwarting man-in-the-middle attacks.

11.4 Pitfalls

Implementing the DH protocol can be a bit tricky. For example, if Eve intercepts the communications and replaces both g^x and g^y with the number 1, then both Alice and Bob will end up with $k = 1$. The result is a key negotiation protocol that looks as if it completed successfully, except that Eve knows the resulting key. That is bad, and we will have to prevent this attack in some way.

A second problem is if the generator g is not a primitive element of $\mathbb{Z}_p^*$ but rather generates only a small subgroup. Maybe g has an order of one million.

In that case, the set $\{1, g, g^2, \ldots, g^{q-1}\}$ only contains a million elements. As k is in this set, Eve can easily search for the correct key. Obviously, one of the requirements is that g must have a high order. But who chooses p and g? All users are using the same values, so most of them get these values from someone else. To be safe, they have to verify that p and g are chosen properly. Alice and Bob should each check that p is prime, and that g is a primitive element modulo p.

The subgroups modulo p form a separate problem. Eve's attack of replacing g^x with the number 1 is easy to counter by having Bob check for this. But Eve could also replace g^x with the number h, where h has a small order. The key that Bob derives now comes from the small set generated by h, and Eve can try all possible values to find k. (Of course, Eve can play the same attack against Alice.) What both Alice and Bob have to do is verify that the numbers they receive do not generate small subgroups.

Let's have a look at the subgroups. Working modulo a prime, all (multiplicative) subgroups can be generated from a single element. The entire group $\mathbb{Z}_p^*$ consists of the elements $1, \ldots, p-1$ for a total of $p-1$ elements. Each subgroup is of the form $1, h, h^2, h^3, \ldots, h^{q-1}$ for some h and where q is the order of h. As we discussed earlier, it turns out that q must be a divisor of $p-1$. In other words: the size of any subgroup is a divisor of $p-1$. The converse also holds: for any divisor d of $p-1$ there is a single subgroup of size d. If we don't want any small subgroups, then we must avoid small divisors of $p-1$.

There is another reason for wanting large subgroups. It turns out that if the prime factorization of $p-1$ is known, then computing the discrete log of g^x can be broken down into a set of discrete log computations over subgroups.

This is a problem. If p is a large prime, then $p-1$ is always even, and therefore divisible by 2. Thus there is a subgroup with two elements; it consists of the elements 1 and $p-1$. But apart from this subgroup that is always present, we can avoid other small subgroups by insisting that $p-1$ have no other small factors.

11.5 Safe Primes

One solution is to use a *safe prime* for p. A safe prime is a (large enough) prime p of the form $2q+1$ where q is also prime. The multiplicative group $\mathbb{Z}_p^*$ now has the following subgroups:

- The trivial subgroup consisting only of the number 1.
- The subgroup of size 2, consisting of 1 and $p-1$.
- The subgroup of size q.
- The full group of size $2q$.

The first two are trivial to avoid. The third is the group we want to use. The full group has one remaining problem. Consider the set of all numbers modulo p that can be written as a square of some other number (modulo p, of course). It turns out that exactly half the numbers in $1, \ldots, p-1$ are squares, and the other half are non-squares. Any generator of the entire group is a non-square. (If it were a square, then raising it to some power could never generate a non-square, so it does not generate the whole group.)

There is a mathematical function called the Legendre symbol that determines whether a number modulo p is a square or not, without ever needing to find the root. There are efficient algorithms for computing the Legendre symbol. So if g is a non-square and you send out g^x, then any observer, such as Eve, can immediately determine whether x is even or odd. If x is even, then g^x is a square. If x is odd, then g^x is a non-square. As Eve can determine the squareness of a number using the Legendre symbol function, she can determine whether x is odd or even; Eve cannot learn the value x, except for the least significant bit. The solution for avoiding this problem is to use only squares modulo p. This is exactly the subgroup of order q. Another nice property is that q is prime, so there are no further subgroups we have to worry about.

Here is how to use a safe prime. Choose (p, q) such that $p = 2q + 1$ and both p and q are prime. (You can use the isPrime function to do this on a trial-and-error basis.) Choose a random number α in the range $2, \ldots, p-2$ and set $g = \alpha^2 \pmod{p}$. Check that $g \neq 1$ and $g \neq p-1$. (If g is one of these forbidden values, choose another α and try again.) The resulting parameter set (p, q, g) is suitable for use in the Diffie-Hellman protocol.

Every time Alice (or Bob) receives a value that is supposed to be a power of g, she (or he) must check that the value received is indeed in the subgroup generated by g. When you use a safe prime as described above, you can use the Legendre symbol function to check for proper subgroup membership. There is also a simpler but slower method. A number r is a square if and only if $r^q = 1 \pmod{p}$. You also want to forbid the value 1, as its use always leads to problems. So the full test is: $r \neq 1 \wedge r^q \bmod p = 1$.

11.6 Using a Smaller Subgroup

The disadvantage of using the safe prime approach is that it is inefficient. If the prime p is n bits long, then q is $n-1$ bits long and so all exponents are $n-1$ bits long. The average exponentiation will take about $3n/2$ multiplications of numbers modulo p. For large primes p, this is quite a lot of work.

The standard solution is to use a smaller subgroup. Here is how that is done. We start by choosing q as a 256-bit prime. (In other words: $2^{255} < q < 2^{256}$). Next we find a (much) larger prime p such that $p = Nq + 1$ for some arbitrary

value N. To do this, we choose N randomly in the suitable range, compute p as $Nq + 1$, and check whether p is prime. As p must be odd, it is easy to see that N must be even. The prime p will be thousands of bits long.

Next, we have to find an element of order q. We do that in a similar fashion to the safe prime case. Choose a random α in $\mathbb{Z}_p^*$ and set $g := \alpha^N$. Now verify that $g \neq 1$ and $g^q = 1$. (The case $g = p - 1$ is covered by the second test, as q is odd.) If g is not satisfactory, choose a different α and try again. The resulting parameter set (p, q, g) is suitable for use in the Diffie-Hellman protocol.

When we use this smaller subgroup, the values that Alice and Bob will exchange are all in the subgroup generated by g. But Eve could interfere and substitute a completely different value. Therefore, every time Alice or Bob receives a value that is supposed to be in the subgroup generated by g, he or she should check that it actually is. This check is the same as in the safe prime case. A number r is in the proper subgroup if $r \neq 1 \wedge r^q \bmod p = 1$. Of course, they should also check that r is not outside the set of modulo-p numbers, so the full check becomes $1 < r < p \wedge r^q = 1$.

For all numbers r in the subgroup generated by g we have that $r^q = 1$. So if you ever need to raise number r to a power e, you only have to compute $r^{e \bmod q}$, which can be considerably less work than computing r^e directly if e is much larger than q.

How much more efficient is the subgroup case? The large prime p is at least 2000 bits long. In the safe-prime situation, computing a general g^x takes about 3000 multiplications. In our subgroup case, g^x takes about 384 multiplies because x can be reduced modulo q and is therefore only 256 bits long. This is a savings of a factor of nearly eight. When p grows larger, the savings increase further. This is the reason that subgroups are widely used.

11.7 The Size of *p*

Choosing the right sizes for the parameters of a DH system is difficult. Up to now, we have been using the requirement that an attacker must spend 2^{128} steps to attack the system. That was an easy target for all the symmetric key primitives. Public-key operations like the DH system are far more expensive to start with, and the computational cost grows much more quickly with the desired security level.

If we keep to our requirement of forcing the attacker to use 2^{128} steps to attack the system, the prime p should be about 6800 bits long. In practical systems today, that would be a real problem from a performance point of view.

There is a big difference between key sizes for symmetric primitives and key sizes for public-key primitives like DH. Never, ever fall into the trap of comparing a symmetric key size (such as 128 or 256 bits) to the size of a public

key that can be thousands of bits. Public-key sizes are always much larger than symmetric-key sizes.[2]

Public-key operations are far slower than the encryption and authentication functions we presented earlier. In most systems, the symmetric-key operations are insignificant, whereas the public-key operations can have a real effect on performance. We must therefore look much more closely at the performance aspects of public-key operations.

Symmetric-key sizes are typically fixed in a system. Once you design your system to use a particular block cipher and hash function, you also fix the key size. That means that the symmetric key size is fixed for the life of the system. Public-key sizes, on the other hand, are almost always variable. This makes it much easier to change the key size. Our intent in this book is to design a system that will be used for 30 years, and the data must be kept secure for 20 years after it has first been processed. The symmetric key size must be chosen large enough to protect the data up to 50 years from now. But the variable-sized public keys only have to protect the data for the next 20 years. After all, all keys have a limited lifetime. A public key might be valid for one year, and should protect data for 20 more years. This means that the public key only needs to protect data 21 years, rather than the 50 years needed for symmetric keys. Each year, you generate a new public key, and you can choose larger public keys as computing technology progresses.

The best estimates of how large your prime p needs to be can be found in [85]. A prime of 2048 bits can be expected to secure data until around 2022; 3072 bits is secure until 2038; and 4096 bits until 2050. The 6800 bits we mentioned above is derived from the formulas in [85]. That is the size of p if you want to force the attacker to perform 2^{128} steps in an attack.

Be very careful with these types of predictions. There is some reasonable basis for these numbers, but predicting the future is always dangerous. We might be able to make some sensible predictions about key sizes for the next 10 years, but making predictions about what things will be like 50 years from now is really rather silly. Just compare the current state of the art in computers and cryptography with the situation 50 years ago. The predictions in [85] are by far the best estimates we have; nevertheless, treat them with caution.

So what are we to do? As cryptographic designers, we have to choose a key size that will be secure for at least the next 20 years. Obviously 2048 bits is a lower bound. Larger is better, but larger keys have a significant extra cost. In the face of so much uncertainty, we would like to be conservative. So here is our advice: as of today, use 2048 bits as an absolute minimum. (And don't forget that as time passes this minimum will grow.) If at all possible from a performance point of view, use 4096 bits, or as close to 4096 bits as you can

[2]This holds for the public-key schemes we discuss in this book. Other public-key schemes, such as those based on elliptic curves, can have completely different key size parameters.

afford. Furthermore, make absolutely sure that your system can handle sizes up to 8192 bits. This will save the day if there are unexpected developments in attacking public-key systems. Improvements in cryptanalysis will most likely lead to attacks on smaller key sizes. Switching to a very much larger key size can be done while the system is in the field. It will cost some performance, but the basic operation of the system will be preserved. This is far better than losing all security and having to reengineer the system, which is what you would have to do if the system could not use larger keys.

Some applications require data to be kept secret for much longer than 20 years. In these cases, you need to use the larger keys now.

11.8 Practical Rules

Here are our practical rules for setting up a subgroup you can use for the DH protocol.

Choose q as a 256-bit prime. (There are collision-style attacks on the exponent in DH, so all our exponents should be 256 bits long to force the attacker to use at least 2^{128} operations.) Choose p as a large prime of the form $Nq + 1$ for some integer N. (See Section 11.7 for a discussion of how large p should be. Computing the corresponding range for N is trivial.) Choose a random g such that $g \neq 1$ and $g^q = 1$. (The easy way to do this is to choose a random α, set $g = \alpha^N$, and check g for suitability. Try another α if g fails the criteria.)

Any party receiving the subgroup description (p, q, g) should verify that:

- Both p and q are prime, q is 256 bits long, and p is sufficiently large. (Don't trust keys that are too small.)
- q is a divisor of $(p - 1)$.
- $g \neq 1$ and $g^q = 1$.

This should be done even if the description is provided by a trusted source. You would be amazed at how often systems fail in some interesting way, especially when they are under attack. Checking a set (p, q, g) takes a little time, but in most systems the same subgroup is used for a long time, so these checks need only be performed once.

Any time a party receives a number r that is supposed to be in the subgroup, it should be verified that $1 < r < p$ and $r^q = 1$. Note that $r = 1$ is *not* allowed.

Using these rules, we get the version of the Diffie-Hellman protocol shown in Figure 11.3. Both parties start by checking the group parameters. Each of them only has to do this once at start-up, not every time they run a DH protocol. (They should do it after every reboot or reinitialization, however, because the parameters could have changed.)

Alice		**Bob**
known: (p, q, g)		known: (p, q, g)
check (p, q, g) parameters		check (p, q, g) parameters
$x \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{X := g^x}$	
		$1 \overset{?}{<} X \overset{?}{<} p,\ X^q \overset{?}{=} 1$
		$y \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$\xleftarrow{Y := g^y}$	
$1 \overset{?}{<} Y \overset{?}{<} p,\ Y^q \overset{?}{=} 1$		
$k \leftarrow (Y)^x$		$k \leftarrow (X)^y$

Figure 11.3: Diffie-Hellman in a subgroup.

The rest of the protocol is very much the same as the original DH protocol in Figure 11.1. Alice and Bob now use the subgroup, so the two exponents x and y are in the range $1, \ldots, q-1$. Both Alice and Bob check that the number they receive is in the proper subgroup to avoid any small-subgroup attacks by Eve.

The notation we use for the checks is a relational operator (such as $=$ or $<$) with a question mark above it. This means that Alice (or Bob) should check that the relation holds. If it does, then everything is all right. If the relation is not correct, then Alice has to assume that she is under attack. The standard behavior is to stop the execution of the protocol, not send any other messages, and destroy all protocol-specific data. For example, in this protocol Alice should destroy x and Y if the last set of checks fails. See Section 13.5.5 for a detailed discussion of how to handle these failures.

This protocol describes a secure variant of DH, but it should not be used in exactly this form. The result k has to be hashed before it is used by the rest of the system. See Section 14.6 for a more detailed discussion.

11.9 What Can Go Wrong?

Very few books or articles talk about the importance of checking that the numbers you receive are in the correct subgroup. Niels first found this problem in the Internet Key Exchange (IKE) protocol of IPsec [60]. Some of the IKE protocols include a DH exchange. As IKE has to operate in the real world, it has to deal with lost messages. So IKE specifies that if Bob receives no answer, he should resend his last message. IKE does not specify how Alice should process the message that Bob sent again. And it is easy for Alice to make a serious mistake.

For simplicity, let us suppose Alice and Bob use the DH protocol in the subgroup illustrated in Figure 11.3 without checking that X and Y are proper values. Furthermore, after this exchange Alice starts using the new key k to send an encrypted and authenticated message to Bob that contains some further protocol data. (This is a very common situation, and similar situations can occur in IKE.)

Here is the dangerous behavior by Alice: when she receives a resend of the second message containing Y, she simply recomputes the key k and sends the appropriate reply to Bob. Sounds entirely harmless, right? But the attacker Eve can now start to play games. Let d be a small divisor of $(p-1)$. Eve can replace Y by an element of order d. Alice's key k is now limited to d possible values, and is completely determined by Y and $(x \bmod d)$. Eve tries all possible values for $(x \bmod d)$, computes the key k that Alice would have gotten, and tries to decrypt the next message that Alice sends. If Eve guesses $(x \bmod d)$ correctly, this message will decrypt properly, and Eve has learned $(x \bmod d)$.

But what if $p-1$ contains a number of small factors $(d_1, d_2, \ldots, d_k)$? Then Eve can run this attack repeatedly for each of these factors and learn $(x \bmod d_1), \ldots, (x \bmod d_k)$. Using the general form of the Chinese Remainder Theorem (see Section 12.2) she can combine this knowledge to obtain $(x \bmod d_1 d_2 d_3 \cdots d_k)$. So if the product of all small divisors of $p-1$ is large, Eve can get a significant amount of information about x. As x is supposed to be secret, this is always a bad development. In this particular case, Eve can finish by forwarding the original Y to Alice and letting Alice and Bob complete the protocol. But Eve has collected enough information about x that she can now find the key k that Alice and Bob use.

To be quite clear: this is not an attack on IKE. It is an attack on an implementation of IKE that is allowed by the standard [60]. Still, in our opinion the protocol should include enough information for a competent programmer to create a secure implementation. Leaving this type of information out is dangerous, as somebody somewhere will implement it the wrong way. (We have not verified whether this attack applies to newer versions of IKE.)

For this attack to work, Eve has to be lucky enough to have a $p-1$ with sufficient small divisors. We are designing against an adversary that can perform 2^{128} steps of computing. This allows Eve to take advantage of all divisors of $p-1$ up to 2^{128} or so. We've never seen a good analysis of the probabilities of how much information Eve could get, but a quick estimate indicates that on average Eve will be able to get approximately 128 bits of information about x from the factors smaller than 2^{128}. She can then attack the unknown part of x using a collision-style attack, and as x is only 256 bits long, this leads to a real attack. At least, it would if we didn't check that X and Y were in the proper subgroup.

The attack becomes even easier if Eve was the person selecting the subgroup (p, q, g). She may have put the small divisors into $p - 1$ herself when she selected p in the first place. Or maybe she sat on the committee that recommended certain parameters for a standard. This isn't as crazy as it seems. The U.S. government, in the form of NIST, helpfully provides primes that can be used with DSA, a signature scheme that uses subgroups like this. Other parts of that same U.S. government (e.g., NSA, CIA, FBI) have a vested interest in being able to break into private communications. We certainly don't want to imply that these primes are bad, but it is something that you would want to check before you use them. This is easy to do; in fact, NIST published an algorithm for choosing parameters that does not insert additional small factors, and you can check whether the algorithm was indeed followed. But few people ever do.

In the end, the simplest solution is to check that every value you receive is in the proper subgroup. All other ways of stopping small subgroup attacks are much more complicated. You could try to detect the small factors of $p - 1$ directly, but that is way too complicated. You could require the person who generated the parameter set to provide the factorization of $p - 1$, but that adds a great deal of complexity to the whole system. Verifying that the received values are in the right subgroup is a bit of work, but it is by far the simplest and most robust solution.

11.10 Exercises

Exercise 11.1 Assume 200 people wish to communicate securely using symmetric keys, one symmetric key for each pair of people. How many symmetric keys would this system use in total?

Exercise 11.2 What are the subgroups generated by 3, 7, and 10 in the multiplicative group of integers modulo $p = 11$?

Exercise 11.3 Why is a number r a square modulo p, $p = 2q + 1$ and p and q both prime, if and only if $r^q = 1 \pmod{p}$.

Exercise 11.4 What problems, if any, could arise if Alice uses the same x and g^x for all her communications with Bob, and Bob uses the same y and g^y for all his communications with Alice?

Exercise 11.5 Alice and Bob wish to agree on a 256-bit AES key. They are trying to decide between using 256-bit, 512-bit, or some other length DH public keys g^x and g^y. What would be your recommendation to them?