|  |  |
| --- | --- |
| **6.4. Diffie-Hellman**  The Diffie-Hellman public key cryptosystem predates RSA and is in fact the oldest public key system still in use. It is less general than RSA (it does neither encryption nor signatures), but it offers better performance for what it does. If Diffie-Hellman neither encrypts nor signs, how can it be called a cryptosystem? What does it do?  Diffie-Hellman allows two individuals to agree on a shared key, even though they can only exchange messages in public. In other words, assume our famous two people, Alice and Bob, want to have a secret number that they share so that they can start encrypting messages to each other. But the only way they can talk is by some means of communication that lots of people can overhear. For instance, they might be sending messages across a network, or on a telephone that might be tapped, or they might be shouting at each other across a crowded room, or they might for some reason have to resort to communication by placing ads in the personals section of the local newspaper.   | Dear    Bob,   the     magic      number     is 1890289304789234789279189028902. Wish you were here. Love, Alice. | | --- | |

In its original conception, the Diffie-Hellman algorithm has limited functionality, since the only thing it really accomplishes is having a secret number that both Alice and Bob know, and nobody else can figure out based on the messages they overhear between Alice and Bob. Neither Alice nor Bob start out with any secrets, yet after the exchange of two messages that the world can overhear, Alice and Bob will know a secret number. Once they know a secret number, they can use conventional cryptography (secret key cryptography like DES, for instance) for encryption. Diffie-Hellman is actually used for key establishment (getting two things to agree on a common secret key) in some applications, for instance data link encryption on a LAN.

A weakness of Diffie-Hellman is that although two individuals can agree on a shared secret key, there is no authentication, which means that Alice might be establishing a secret key with a bad guy. We will talk about this more after we force you to read through how Diffie-Hellman works.

To start out, there are numbers p and g, where p is a large prime and g is a number less than p with some restrictions that aren't too important for a basic understanding of the algorithm. p and g are known beforehand and can be publicly known. For instance, Alice could choose a p and g and send them (publicly) to Bob, for instance by publishing them in The New York Times.

| Dear Bob, I'd like our prime |
| --- |
| to be 128903289023 and our |
| g to be 23489. Love, Alice. |

Once Alice and Bob agree on a p and g, each chooses a 512-bit number at random and keeps it secret. Let's call Alice's secret number SA and Bob's secret number SB. Each raises g to their secret number, mod p. The result is that Alice computes some number TA and Bob computes some number TB. They exchange their Ts. Finally, each raises the received T to their secret number.

| Alice picks SA at random. Bob picks SB at random. |
| --- |
| Alice computes TA = gSA mod p. Bob computes TB = gSB mod p. |
| They exchange Ts (in either order or simultaneously): TA TB |
| Now each raises the number they receive to their private secret number: |
| Alice computes TBSA mod p. Bob computes TAS B mod p. |
| They will both come up with the same number. That is because |
| TBSA = (gSB)SA = gSBSA = gSASB = (gSA)SB = TASB mod p. |

Nobody else can calculate gSASB in a reasonable amount of time even though they know gSA and gSB. If they could compute [discrete logarithms](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/app02.html#gloss01_073), i.e. figure out SA based on seeing gSA, then they could figure out the Alice/Bob shared key. But we assume they can't compute discrete logarithms, because of the Fundamental Tenet of Cryptography (mathematicians haven't figured out how to do that easily in spite of considerable effort, or at least they haven't told us they have).

This property is magical:

TA=23 mod 5

TB=25 mod 5

Alice: 23 mod 5

Bob: 35 mod 5

**6.4.1. The Bucket Brigade/Man-in-the-Middle Attack**

If Alice receives TB indirectly, there is no way for her to know for sure whether the number came from Bob. She will establish a secret key with whoever transmitted TB, but it certainly might not be Bob. Let's assume Alice is talking to X, who may or may not be Bob. Once Alice and X establish a secret key, they can encrypt all their messages so that only Alice and X can read them. Let's say the first thing Alice and X exchange in their encrypted communication is a password that Alice and Bob have previously agreed upon, one password that Bob is to say to Alice, perhaps The fish are green, and one that Alice is to say to Bob, for instance The moon sets at midnight. If Alice receives the expected password from X, can she assume she is talking to Bob? (Think about this a [bit](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/app02.html#gloss01_021) before reading the next paragraphÂ—it's fun. Hint: Obviously, there must be some subtle attack or we wouldn't claim it was an interesting question.)

Assume p and g are publicly known (if not, Alice can put them into her message). Alice places the ad Dear Bob. I'd like to talk to you. 8389. Love, Alice. Suppose there's a bad guy, Mr. X, who works for the newspaper. He makes one copy of the newspaper with Alice's ad printed as Alice wished, and bribes the newspaper deliverer to give that copy to Alice. Meanwhile, Mr. X picks his own SX and computes gSX mod p. He edits the ad slightly by substituting this number instead of 8389, and has that version printed in the rest of the newspapers. Later, Bob replies, by ordering the ad So pleased to talk to you. My magic number is 9267. Love, Bob. Mr. X makes one copy of the newspaper with Bob's ad printed as Bob wished, and arranges for Bob to receive that copy. Mr. X edits Bob's ad slightly to substitute his own number for Bob's, and arranges for the newspaper with that version of the ad to get to Alice. Mr. X computes KAX = 8389SX and uses that for talking to Alice, and computes KBX = 9267SX and uses that for talking to Bob.

**Figure 6-4. Bucket Brigade/Man-in-the-Middle Attack**

Now suppose Alice sends an encrypted message (to Mr. X) which includes the password The fish are green. Mr. X can decrypt the message because it is encrypted using the key he shares with Alice. Mr. X reencrypts and transmits Alice's password to Bob, which reassures Bob that he is indeed talking to Alice, and Bob then transmits (encrypted, to Mr. X) The moon sets at midnight. Mr. X decrypts Bob's message, extracts the password, and reencrypts and transmits the password to Alice. Now both Alice and Bob think they are talking to each other.

To guard against this threat, perhaps Alice and Bob should transmit the actual secret number they think they are using, over the encrypted channel they have established? That won't work either. Alice sends the message I think we are using KAX; however, Mr. X decrypts the message and edits it to be I think we are using KBX before encrypting and forwarding it on to Bob.

Suppose instead Alice and Bob attempt to reassure themselves that they are indeed talking to each other by asking each other personal questions. If Bob asks What movie did we see the first night we met in Paris?, Mr. X merely decrypts the message with KBX, encrypts it with KAX, and forwards the message on to Alice. When Alice replies, Mr. X merely forwards the message on to Bob. (We're assuming Alices's memory of past evenings makes it worthwhile for Mr. X to actually wait for her answer rather than making one up himself.)

The name [**bucket brigade attack**](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/app02.html#gloss01_023) comes from the way firefighters of old formed a line of people between a water source and a fire and passed full buckets toward the fire and empty buckets back. But the less politically correct term [**man-in-the-middle**](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/app02.html#gloss01_125) has become more common. After establishing the shared keys, Mr. X passes messages back and forth and can examine them and/or modify them as they go. Using Diffie-Hellman alone, there's really nothing Alice and Bob can do to detect the intruder through whom they are communicating. As a result, this form of Diffie-Hellman is only secure against passive attack where the intruder just watches the messages.

**6.4.2. Defenses Against Man-in-the-Middle Attack**

**6.4.2.1. Published Diffie-Hellman Numbers**

One technique by which Diffie-Hellman can be secure against active attacks is for each person to have a somewhat permanent public and secret number instead of inventing one for each exchange. For this to work, everyone (in the communicating set) has to agree on a common p and g. The public numbers are then all published by some means that is assumed reliable (for instance, through a PKI, see [Chapter 15](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/ch15.html#ch15) PKI (Public Key Infrastructure)).

To the extent an intruder can't get in and modify the published public numbers, this makes Diffie-Hellman immune to active attacks. It has the additional advantage of eliminating the first two messages of the protocol. Knowing my own secret and looking up the public number of the person with whom I want to communicate, I can compute a key that the two of us will share for all messages we send to one another.

**6.4.2.2. Authenticated Diffie-Hellman**

If Alice and Bob know some sort of secret with which they can authenticate each other, either a shared secret key or knowledge of each other's public keys (and their own private keys), then they can use this secret to prove that it was they. who generated their Diffie-Hellman values. We call such an exchange an authenticated Diffie-Hellman exchange. The proof can be done simultaneously with sending the Diffie-Hellman value, or after the Diffie-Hellman exchange. Examples are:

* Encrypt the Diffie-Hellman exchange with the pre-shared secret.
* Encrypt the Diffie-Hellman value with the other side's public key (see [Homework Problem 2](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/ch06lev1sec9.html#ch06qa1q2)).
* [Sign](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/app02.html#gloss01_204) the Diffie-Hellman value with your private key.
* Following the Diffie-Hellman exchange, transmit a hash of the agreed-upon shared Diffie-Hellman value, your name, and the pre-shared secret.
* Following the Diffie-Hellman exchange, transmit a hash of the pre-shared secret and the Diffie-Hellman value you transmitted.

**6.4.3. Encryption with Diffie-Hellman**

We've described the lack of authentication with Diffie-Hellman. There's another disadvantage with classic Diffie-Hellman: in order for two individuals to communicate, they have to first have an active exchange. This is easy to remedy. Suppose Alice is working late and wants to send an encrypted message to Bob that Bob will be able to read when he shows up at work the next morning at 7 AM (and Alice has no intention of being around at 7 AM).

First everyone computes a public key, which consists of the three numbers <p, g, T>, where T = gS mod p, for the private key S. These public keys are displayed in a reliable and public place (like being published in The New York Times).

So Bob has published a <pB, gB, TB>.

If Alice wants to send Bob an encrypted message, she picks a random number SA, computes gBSA mod pB and computes KAB = TBSA mod pB, and uses that as the encryption key to share with Bob. She uses KAB to encrypt the message according to any secret key cryptographic technique, and sends the encrypted message, along with gBSA mod pB to Bob. Bob raises gBSA mod pB to his own secret SB, and thereby calculates KAB which enables him to decrypt the message.

**6.4.4. ElGamal Signatures**

Using the same sort of keys as Diffie-Hellman, ElGamal came up with a signature scheme [[ELGA85](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/biblio01.html#biblio01_055)]. It is much harder to understand than signing with RSA. While it's important to know that it's possible and to understand the political and performance implications, the mathematics is tedious and unintuitive. We reluctantly recommend that all but true die-hards skip this section.

ElGamal signatures require each individual to have a long-term public/private key pair (the public key being <g, p, T> and the secret key being S, where gS mod p = T, as described for Diffie-Hellman), and (surprisingly) require an individual to generate a new and different public/private key pair for each item that needs to be signed. Luckily the per-message public/private key pair is easy to compute. For a particular message m, choose a random number Sm and (using the same g and p as in the long-term key) compute gSm mod p = Tm. To use ElGamal, there has to be a message digest function that is well-known. Given a message m, to compute a signature, you first compute the message digest of m|Tm. Call that message digest dm. Then you calculate Sm + dmS mod (p-1), which is the signature. Let's call that X (since we've already used S for the Secret number, and some people sign documents with an X).

m is transmitted, along with X and Tm. To verify this signature, you compute dm and check that gX = TmTdm mod p. This will be true, assuming the signature is valid, because

| gX = gSm+dmS = gSmgdmS = TmTdm mod p. |
| --- |

This is not terrifically intuitive. The important things to try to convince oneself of are:

* If the signature is done correctly, the verification will succeed.
* If the message is modified after being signed, the inputs to the signature function will have changed and with overwhelming probability the signature will not match the modified message.
* Knowledge of the signature will not help divulge the signer's private key, S.
* Someone without knowledge of S will not be able to produce a valid signature.

**6.4.5. Diffie-Hellman DetailsÂ—Safe Primes**

While Diffie-Hellman works with any prime p and any number g, it is less secure if p and g don't have additional mathematical properties.

It turns out that for obscure mathematical reasons it is particularly nice if (p-1)/2 is also prime. A prime p that satisfies this additional constraint is called a [**safe prime**](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/app02.html#gloss01_194), or a Sophie Germain prime. It is also particularly nice if gx 1 mod p unless x = 0 mod p-1. If p is a safe prime, this is satisfied by any g -1 mod p for which g(p-1)/2 = -1 mod p, which is true for almost half of all mod p numbers.

It is computationally expensive to choose p and g. Theoretically, one only needs to do it once. You could keep using the same p and g. It could even be a standard, and everyone could use the same p and g. However, that is not advisable. It turns out to be possible, though incredibly space-and computation-intensive, to calculate a large table based on a single p, which would allow you to compute discrete logarithms for that p. A similar scheme would allow you to break RSA, in the sense of being able to calculate someone's private key based on their public key, but there is not that much incentive to do so because that would only allow you to break one person's key. If the p for Diffie-Hellman were broken, then every key exchange based on Diffie-Hellman could be broken. Simply changing p occasionally will eliminate this threat.

**6.3. RSA**

RSA is named after its inventors, Rivest, Shamir, and Adleman. It is a public key cryptographic algorithm that does encryption as well as decryption. The key length is variable. Anyone using RSA can choose a long key for enhanced security, or a short key for efficiency. The most commonly used key length for RSA is 512 bits.

The block size in RSA (the chunk of data to be encrypted) is also variable. The plaintext block must be smaller than the key length. The ciphertext block will be the length of the key. RSA is much slower to compute than popular secret key algorithms like DES and IDEA. As a result, RSA does not tend to get used for encrypting long messages. Mostly it is used to encrypt a secret key, and then secret key cryptography is used to actually encrypt the message.

**6.3.1. RSA Algorithm**

First, you need to generate a public key and a corresponding private key. Choose two large primes p and q (probably around 256 bits each). Multiply them together, and call the result n. The factors p and q will remain secret. (You won't tell anybody, and it's practically impossible to factor numbers that large.)

To generate your public key, choose a number e that is relatively prime to φ(n). Since you know p and q, you know φ(n)Âit's (p-1)(q-1). Your public key is <e,n>.

To generate your private key, find the number d that is the multiplicative inverse of e mod φ(n). <d,n> is your private key.

To encrypt a message m (< n), someone using your public key should compute ciphertext c = me mod n. Only you will be able to decrypt c, using your private key to compute m = cd mod n. Also, only you can sign a message m (< n) with signature s = md mod n based on your private key. Anyone can verify your signature by checking that m = se mod n.

That's all there is to RSA. Now there are some questions we should ask.

* Why does it work? (E.g., will decrypting an encrypted message get the original message back?)
* Why is it secure? (E.g., given e and n, why can't someone easily compute d?)
* Are the operations encryption, decryption, signing, and verifying signatures all sufficiently efficient to be practical?
* How do we find big primes?

**6.3.2. Why Does RSA Work?**

RSA does arithmetic mod n, where n = pq. We know that φ(n) = (p-1)(q-1). We've chosen d and e such that de = 1 mod φ(n). Therefore, for any x, xde = x mod n. An RSA encryption consists of taking x and raising it to e. If we take the result and raise it to the d (i.e., perform RSA decryption), we'll get (xe)d, which equals xed, which is the same as x. So we see that decryption reverses encryption.

In the case of signature generation, x is first raised to the d power to get the signature and then the signature is raised to the e power for verification; the result, xde, will equal x.

**6.3.3. Why Is RSA Secure?**

We don't know for sure that RSA is secure. We can only depend on the Fundamental Tenet of CryptographyÂ—lots of smart people have been trying to figure out how to break RSA, and they haven't come up with anything yet.

The real premise behind RSA's security is the assumption that factoring a big number is hard. The best known factoring methods are really slow. To factor a 512-bit number with the best known techniques would take about thirty thousand MIPS-years [[ROBS95](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/biblio01.html#biblio01_144)]. We suspect that a better technique is to wait a few years and then use the best known technique.

If you can factor quickly, you can break RSA. Suppose you are given Alice's public key <e,n>. If you could find e's exponentiative inverse mod n, then you'd have figured out Alice's private key <d,n>. How can you find e's exponentiative inverse? Alice did it by knowing the factors of n, allowing her to compute φ(n). She found the number that was e's multiplicative inverse mod φ(n). She didn't have to factor nÂshe started with primes p and q and multiplied them together to get n. You can do what Alice did if you can factor n to get p and q.

We do not know that factoring n is the only way of breaking RSA. We know that breaking RSA (for example, having an efficient means of finding d, given e and n) is no more difficult than factoring [[CORM91](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/biblio01.html#biblio01_030)], but there might be some other means of breaking RSA.

Note that it's possible to misuse RSA. For instance, let's say I'm going to send Alice a message divulging the name of the Cabinet member who allegedly once hired a kid to mow his/her lawn, and didn't fill out all the proper IRS forms. Bob knows that's what I'm going to transmit. I'll encrypt the text string which is the guilty person's name using Alice's public key. Bob can't possibly decrypt it, because we believe RSA is secure. So what can Bob learn from eavesdropping on the encrypted data?

Well, Bob can't decrypt, but he can encrypt. He knows I'm sending one of fourteen possible messages. He takes each Cabinet member's name and encrypts it with Alice's public key. One of them will match my messageÂ—unless I use RSA properly. In §[6.3.6](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/ch06lev1sec3.html#ch06lev2sec9) Public-Key Cryptography Standard (PKCS) we'll discuss how to use RSA properly. For now, a simple thing I can do to prevent Bob from guessing my message, encrypting with Alice's public key, and checking the result, is to concatenate the name with a large random number, say 64 bits long. Then instead of fourteen possible messages for Bob to check, there are 14x264, and checking that many messages is computationally infeasible.

### 9.6. Eavesdropping and Server Database Reading

Public key technology makes it easy to do authentication in a way that is both secure from eavesdropping and from an intruder reading the server database. Alice knows her own private key. Bob stores Alice's public key. An intruder who reads Bob's database (and therefore obtains Alice's public key) will not be able to use the information to impersonate Alice. Authentication is done by Alice using her private key to perform a cryptographic operation on a value Bob supplies, and then transmitting the result to Bob. Bob checks the result using Alice's public key.

Without public key cryptography, it's difficult to protect against both eavesdropping and server database reading with a single protocol, although it is easy to do one or the other. For instance, let's try a protocol such as is used when a person logs into a system such as UNIX. Bob (the computer authenticating the user Alice) stores a hash of Alice's password. Let's say Alice's password is fiddlesticks. In this protocol someone, say Trudy, who accessed Bob's database would not be able to impersonate Alice. She could do an off-line password-guessing attack if she were to obtain Alice's hashed password from Bob's database, but that would not help her if Alice had chosen a good password. But if Trudy were to eavesdrop when Alice was proving her identity to Bob, Trudy would obtain Alice's password. So this protocol is secure against server database disclosure but not against eavesdropping.

Now consider another protocol (see below). Let's assume that Bob stores Alice's actual secret. In this case Trudy, if she were to eavesdrop, would not be able to obtain information to allow her to impersonate Alice (except by an off-line password-guessing attack). But if she were to read Bob's database, she'd obtain Alice's secret. We'd almost claim it was impossible to simultaneously protect against eavesdropping and server database disclosure without using public key cryptography. However, an elegant protocol invented by Leslie Lamport can claim to accomplish this feat. Lamport's scheme does have a serious drawback, though. A user can only be authenticated a small finite number of times (such as 1000) before it is necessary to reinstall the user's information at the server; that is, someone with privileges at the server and knowledge of the user's password must reconfigure the server with the user's security information. Phil Karn has implemented Lamport's scheme in his S/Key software, and it is being used in the Internet. Lamport's scheme is described in §[12.2](mk:@MSITStore:C:\Users\XuYan\Desktop\Security%20HW%205\0130460192.chm::/ch12lev1sec2.html#ch12lev1sec2) Lamport's Hash.(要求定期更换密码！)

## Password Hash Quirk

There was an operating system (which we'll allow to remain nameless) that stored hashes of passwords, and the password hash used by that operating system had an interesting property. We are purposely not giving the exact details, but the general idea is that there was a magic character sequence X such that the hash of any string S was the same as the hash of X concatenated with S. This is not exactly a security flaw, but instead is a wonderfully user-friendly feature. How can this odd feature be useful?

Suppose the system administrator made the common policy decision that user passwords had to be some minimum length. This particular operating system enforced the policy only when you set your password (as opposed to when you logged in). Now suppose you wanted a password you could type quickly, like FOO. The operating system would not let you set your password to FOO because it's not long enough. But it would let you set it to X concatenated with FOO. Let's say the magic string was %#v27dR678riwueyru3ir3. You'd set your password to %#v27dR678riwueyru3ir3FOO. That would be the last time you'd have to type %#v27dR678riwueyru3ir3FOO. From then on, FOO would work just fine, since the hash of FOO is the same as the hash of %#v27dR678riwueyru3ir3FOO.