

Cryptography

mlowman

November 2007

1 Intro

Cryptography is what is used to hide information. You store info in some way that it doesn't make sense, and then you convert it back the other way. The simplest kind of cipher is a Caesar cipher: simple letter-for-letter substitution. But here we deal with crypto information much more advanced. Understand that a cryptographic algorithm has only one requirement: that it be reversible, or that some meaning at least can be gleaned from ciphertext. Also, all cryptographic techniques are breakable save one: the one-time pad. A one-time pad relies upon completely and totally random sheets of information, and each can only be used once, which makes it hard to use. Also, there's communication of the pad... Anyways. Back to business.

2 Types

2.1 Hashes

A hash is an algorithm that takes some binary data as input, does something to it, and returns something completely different. Ideally, a hash function will be highly chaotic: A change of a single bit of input will result in an utterly changed output. Other very important qualities are that the function be one-way: once hashed, the original data cannot be reconstructed. And lastly, collision-finding should be impossible. Yes, there must be more than one input data set that will produce a given hash, but you shouldn't be able to find out what it is. This is a strengthening of that last quality.

No function is ideal. However, a well-constructed function with no flaws

should require a great deal of time with today's computing power: some algorithms have claimed that at the time, every computer trying at the same time wouldn't be able to crack the hash before the estimated end of the universe.

2.1.1 Uses

One good example: passwords. If the original password were stored in plain-text on a box, a hacker gaining access would learn every user's password. In addition, the superuser would by definition know everyone's password. Obviously unacceptable. Instead, the computer stores a hash of the password. If the hash of the user input matches the stored hash, you have a match! Problems: a weak hash can be cracked. A "brute-force" attack, where an attacker attempts to discover a password for a hash she found by just goes down the dictionary, can crack a weak password.

2.2 Strong Encryption Algorithm

A strong key (password-type-thing, but lots of bits long) is used with a function to encrypt text. The same key brings the text back. This can be pretty secure. but let's look at a different problem. That key has to be known by both parties. The entire point is to enable encryption across insecure channels, right? (Right.) But let's say the key was compromised. You must immediately change your key, but you can't securely send one! Problem. Or perhaps the first establishment of a channel. You need personal contact, probably. What to do... Use the next kind of encryption, which is incidentally also the most awesome.

2.3 Public-Private Keys

The idea here is that there are two keys. They're mathematically related (usually security nowadays comes from really huge numbers that are a few incredible large primes multiplied. To solve the algorithm, you must factor that mind-boggling huge number. This is one of the most difficult problems possible in computation today). One's called the public key, and the other the private. Anything encrypted by the public can only be decrypted by the private, and vice-versa.

See the awesomeness? The public key can be transmitted by yelling it, for all

you care. In fact, it is most useful if the maximum number of people know it. What can you do with this tool? A few things.

2.3.1 Encrpyt

Say Alice wants to send a message to Bob. Everyone knows Bob's public key, including Alice. So Alice takes her message and encrpyts it with Bob's public key. Alice sends it over email. Mallory intercepts all of Alice's communications, looking for this email, but he can't decrpyt it. Bob, the only person in the world with the private key to match his public one, uses his private key to decrypt the message. He keeps his key in encrypted form on him at all times: there is only one copy in his pocket and one copy in a Swiss bank vault. The NSA also intercepts the email (at considerably less effort, since they can intercept everyone's email at not extra effort). They use their huge cluster, constructed at the hardware level with the sole purpose of cracking this kind of key, to generate the private key. Alice gets pwned, but there isn't much she could have done anyways against the awesome power of the NSA. So she invents a giant space laser...

2.3.2 Sign

Sometimes it doesn't matter who knows some one said something; you just want to be very sure that a *certain* someone said something. So you sign your message. This uses the information as the input and the private key as the key; then everyone can decrpyt the contents with your public key and see that they match the email. Change the email (say, *Give Bob to money* to *Give Mallory the money*) and everyone know that the message doesn't match the signature. Something's wrong...

3 Trust

So we decided that public-private keys are pretty wicked. We will use them for all our information. But what about the problem of the public keys. To send someone a message, you need their public key. But let's say that Mallory intercepts all traffic between Bob and Alice. Alice asks for Bob's public key so that she can send him secure data, and Bob responds. But Mallory blocks that response, saves the public key, and replaces the public key with his own before allowing the message to continue on its merry way.

Ouch. What's a way to check that the key Alice got is right? She could call Bob and ask him to read his own key's checksum, ensuring that they're the same. But if Alice didn't know Bob, maybe Mallory could pretend to be Bob... There are two ways to handle this situation.

3.1 Web of Trust

This is where if you meet Bob, and you're sure that Bob Bob and that the key he holds is in fact his public key, then you sign it. It's the same as signing a piece of mail. By signing, you say "Yes, this is Bob's key." And if enough people sign other people's keys, eventually you end up with a set of relationships that leads you to Bob. Maybe two different people that you trust (and know the keys of) sign Bob's key. There are lots of little additional things, too: people have different weights and the most thorough ones at checking identity are assigned more weight, but that's the basic idea.

3.2 Single Authority

One person has the responsibility for the whole company, or something like that. They sign your key, and that makes it valid. If they don't sign, it's not valid. As simple as that. Verisign will sign your key if you pay them money: they're called a Certificate Authority, or CA. Note that by design, there is only one CA for you.

4 SSL

Properly called TLS, this is a method for certifying/encrypting a web connection. It runs on port 443 instead of 80 (good to know when asking about firewalls and such). Every browser has a built-in list of accredited root CAs. As a site admin, you create a random public-private key pair. Then you generate a CSR (Certificate Signing Request) that you send along with your public key to your chosen CA. The CA signs with their private key (known only to the CA), and sends you back a certificate. You "install" this certificate, and your webserver (assuming proper configuration) uses it to encrypt communication.

4.1 Other Services

OpenVPN can use a few methods of encryption, but the easiest and most secure is certificate-based. Also, one WPA auth mechanism involves certs. Very secure.

5 SSL Setup

This is how to use SSL in your own site. First, install apache (or any other server you like). Ensure you have SSL support available/enabled. With apache, this means enabling the module.

5.1 Site Setup

By the way, this explanation will assume debian. Other distros will be different, but you can figure it out. The easiest thing to do is to copy the site file *default* from sites-available to, let's say *secure* in the same dir. Edit the file to add a `:443`, like so: `<VirtualHost *:443>`. Of course, it won't be an asterix with multiple vhosts. Also edit the other site file to add a `80`, since it should no longer glom on to all connections. Make a symlink from sites-available to the file in sites-enabled: from the sites-enabled dir, `ln -s ../sites-available/default 050-secure`.

5.2 Certificate Generation

For this example, we will not pay money; we'll be our own CA. Every single thing to do with this will use openssl, so check that we've got this installed. Now, since we'll be the CA, let's set all that up. I'll put mine in the dir */etc/apache2/cafiles*.

5.2.1 Private Key

To create our CA's private key, we use the command `openssl genrsa -out ca_private.key -des3 1024`. This means that we use the RSA algorithm to make the key, which is 1024 bits long (pretty secure, but 2048 would be even more secure). We output the key to the specified file, and encrypted it by using the DES encryption algorithm 3 times in a row. It will ask for a

passphrase, which you'll need to decrypt the key. Leave out the `-des3` option to be insecure, but not have a passphrase.

5.2.2 Certificate

From our private key, now we will create a certificate. Now, certificates generally extend down a line: someone higher up signs this to make it legal, and it goes down the chain. This will be a root CA, or the top of a chain. It's really quite simple: `openssl req -x509 -new -key ca_private.key -out ca.crt -days 365`. The `req` means we generate a request (although we also sign it in the same step, since it's self-signed). The `x509` says we're creating a certificate, not a normal key (X.509 is the cert standard def). The `new` creates a new cert, instead of waiting for a csr on stdin. And this will be valid for 365 days from today. Now fill out all your personal information, and POOF. You have a cert. (P.S.: did you change the perms to 600 on the private key? Don't want people to see that...)

5.2.3 Web Server Keys

Now that we've got our CA, we need something for the CA to authorize. Make a new directory for your server keys, and remember that normally the CA would be somewhere else. Secure communications, remember? OK. The generation of the private server key's actually exactly the same; this time I'll call mine `serv_private.key`. Command: `openssl genrsa -out serv_private.key -des3 1024`. Again, remember that this extremely important key should be set to 600. Now generate a signing request: `openssl req -new -key serv_private.key -out serv_request.csr`. This contains the public key and the info for the cert that is to be made. It asks the CA to make a cert, signed with their secure private key, saying that the private key stored on our soon-to-be SSL server goes with the data we gave them.

5.2.4 Server Certificate

Now we send our csr (Certificate Signing Request) across the net to the CA...which just happens to be a dir over. As the CA, be sure to take precautions to verify the accuracy of the info in the request: run `openssl req -in serv_request.csr -text -verify -noout`, which checks the request, prints it in plain-text form, and doesn't print the encoded version. You see the data in the Subject field? Make sure that's you. Now that you're satisfied, sign the

cert with this command: `openssl x509 -req -days 365 -in serv_request.csr -out ssl_server.crt -CA ca.crt -CAkey ca_private.key -CAcreateserial`. Now move the newly created and verified certificate back to the SSL server.

5.2.5 Notes

Let's make sure we understand this: only the private keys were vitally important. Were anything else to be intercepted, it's fine! And the private keys are never sent...well, anywhere. But the SSL server cannot send encrypted data without it. And the CA cannot verify new certificates without it.

5.3 Finishing Setup

Now the server has a private key (`serv_private.key`) to encrypt data. And it also has a certificate (`ssl_server.crt`) that contains the public key for our server, information identifying our server, and a signature by our trusted authority. Setting up Apache to use these is simple:

SSLEngine on

SSLCertificateFile /path/to/server/cert

SSLCertificateKeyFile /path/to/server/key

And lastly, since you encrypted your private key, you'll need to specify it every time you start apache.