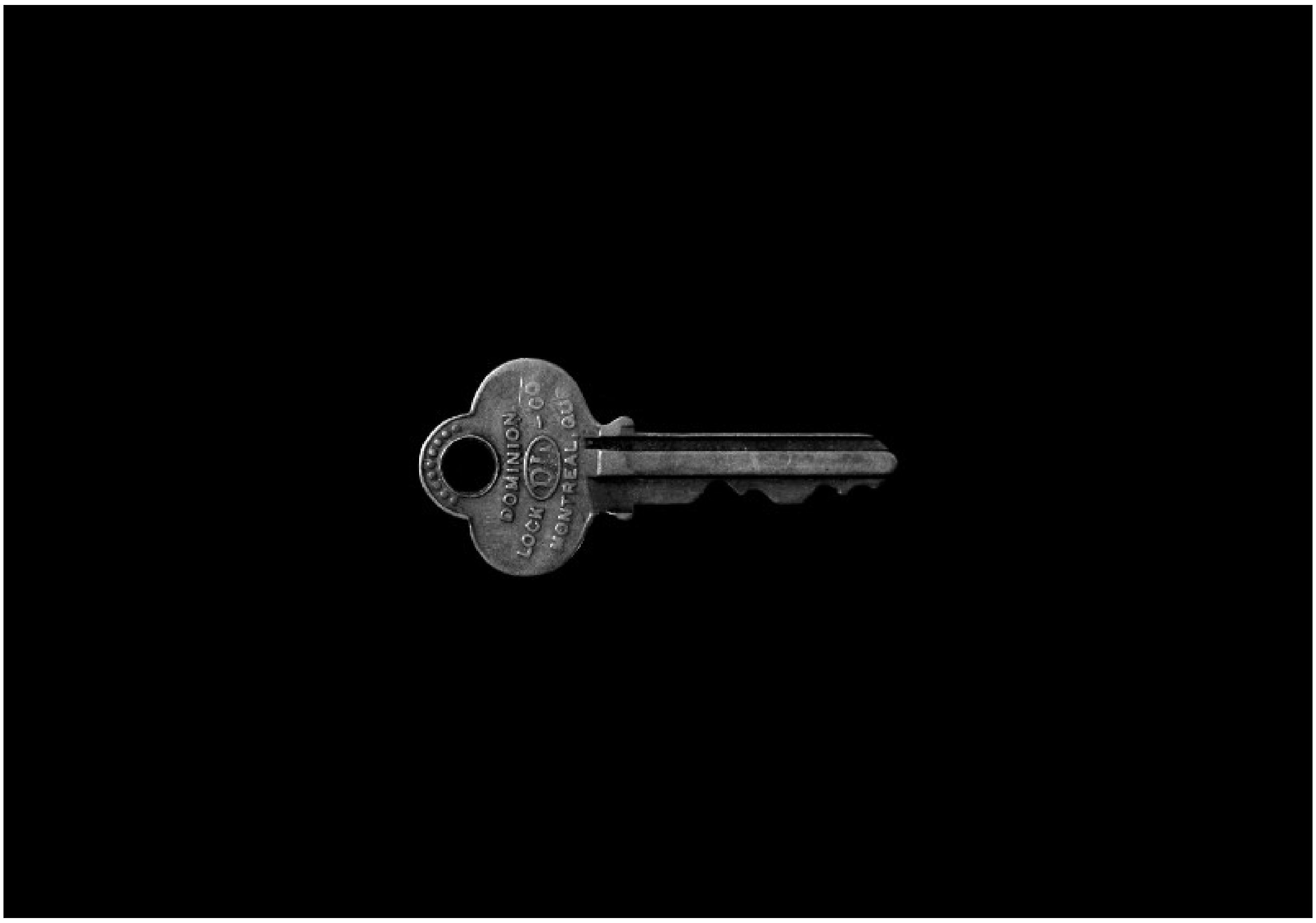
6 MARCH 2019 / #SECURITY

A top-down introduction to SSH and how it enables secure data-sharing



by Sam Ollason

This article will take a high-level and top-down approach to explain how SSH works and how it is used for securely

We will look at how an SSH session is actually 'secure' and how computers establish and set-up an SSH session in the first place. We will also look at the benefits of using SSH.

Note: This is intended as future notes to myself, but I hope you learn something from it too!

What is SSH?

SSH is short for 'secure shell'. It is a protocol for sharing data between two computers over the internet.

A protocol is essentially a set of rules that define the language that computers can use to communicate.

Typically, the two computers involved are your computer (the 'client') and a remote server (the 'host').

Why do we care?

Secure communications between computers

Whenever two computers communicate over the internet we want to be sure that our messages can't be intercepted and understood by anyone listening to the messages.

Imagine sending your bank details over the internet to buy something online. If your messages weren't encrypted, then any computer that was listening or any computer that received the messages to pass onwards may be able to see your account number and your password. That isn't good!

I believe this is an important concept to understand for anyone who

Donate

Secure access to remote computers

Using SSH to check authentication is a more secure way of authentication than using a password. We will explore how this works below.

How is SSH secure?

SSH is a secure way of sending communications between two computers.

By 'secure', I mean a way of encoding the messages on a client computer such that the only other computer that can decode the messages and understand them is the host. This encoding/decoding is called **encryption,** so what we really mean here is SSH is secure because it uses an **encrypted communication channel.**

How is a SSH session established?

There are several processes that need to happen between two computers in order for an SSH session to begin.

1. First we need a way of setting up a secure method of exchanging messages between the computers. We need to set up an **encrypted channel.**
2. We need a way of checking that the data received by the host hasn't been tampered with. This called **verification** and here we are verifying the integrity of the data that is sent by the client.
3. Verification (again). We need a way of checking that the computer we are communicating with isn't an imposter. This is another form of verification but here we are verifying the identity of the computer.

After these three steps, we can now communicate securely with a remote computer.

After these steps, we can share ‘secret’ data securely and we can also check if a client has permission to access a host in a more secure way than using a password. This process is called **authentication using asymmetric encryption.**

Each of these sections below will go into more detail on these steps.

Setting up an encrypted channel

A core part of the SSH protocol is that is it secure (it is in even in the name!), meaning all information that is sent using SSH is encrypted.

How does this information get encrypted?

Encrypting essentially just means ‘jumbling up the letters’ using some clever maths. Both computers need to have a way of encrypting the information so that only the other computer can decrypt the information and understand it.

How does this work?

Both computers have an identical version of a **symmetric key.** The symmetric key is just a string of letters stored somewhere on the computers. The computers can use the symmetric keys to encrypt and also decrypt messages sent to them.

Using this symmetric key approach is called **symmetric encryption.** The ‘symmetric’ part comes from the fact the symmetric key on each computer is identical. This approach works really well … but it symmetric key.

A problem

How do both computers know what the symmetric key is?

One computer could create it and send it in a message over the internet. But the messages wouldn't be encrypted yet, so anyone intercepting the messages would instantly have the symmetric key … and can decrypt all future communications. That's bad!

This is sometimes called the 'key-exchange' problem. It is clear that we need to add another step in the process before we can use symmetric keys.

A solution

A solution to the 'key-exchange' problem above is that both computers share some public information with each other (it is 'public' meaning they don't mind if anyone intercepts it) and combine this with some information on their own computer to **independently** create **identical** symmetric keys.

These symmetric keys can then be used in symmetric encryption in the way outlined above.

How this works

Both computers each have their own private key and public key. Together they form a **key-pair**. The computers **share their public keys** with each other over the internet. So, at this point in the process each computer knows

- its own private key,
- its own public key,

Donate

Generating Symmetric Keys

Both computers then use these 3 pieces of information to independently generate an **identical** symmetric key.

Each computer uses a mathematical algorithm which uses the 3 inputs mentioned above. This algorithm is part of the Diffie-Hellman key exchange algorithm. The algorithm that will be executed on each computer is something like this:

```
Hostpub_2 = other computer's public keypub_1 = my public keypri_
```

```
f(pub_2, pub_1, pri_1) = abcdefg // Symmetric Key
```

```
Client:f(pub_1, pub_2, pri_2) = abcdefg // Symmetric Key
```

The important thing to take away here is that computers have **shared only public information** over the internet **but have still been able to create symmetric keys!**

The approach of using key-pairs and sharing public information to generate identical symmetric keys is called **asymmetric encryption**. It is called 'asymmetric' because both computers start off with their own, different, key pairs.

So far: we have seen how to use asymmetric encryption to computers *in a secure way* (solving the key-exchange problem) and then securely exchange information between computers using symmetric keys for encryption and decryption.

Verification

So we can communicate securely. But the next part of the process of establishing an SSH session is to verify that the data hasn't been tampered with as it has been transmitted **and** that the other computer is actually who it is says it is.

Why do we need this?

Another computer could impersonate one of the computers and initiate the key exchange above. So how do we **securely** figure out that the message is actually from the other computer and not from an imposter?

Hashing

We have to use a **hash** function. This is just a mathematical function that takes inputs and produces a string of a fixed size.

The important feature of this function is that it is virtually impossible to work out what the inputs were just using the outputs.

After a client and a host have generated their symmetric keys, the client will use a hashing function to generate a HMAC. This just stands for "hash-based message authentication code". This is just another string of characters/numbers. The client will send this HMAC to the server for verification.

The ingredients to the hashing function are

- The symmetric key on the client

Donate

contained in a ‘package’ of information)

- The (encrypted!!!) message contents

An example with fake data:

```
symm_key       = abcdefgpkge_no        = 13encr_message   = encr
```

```
Hash(symm_key, pkge_no, encr_message) = *HMAC* // Hashed value
```

How does the host use this information?

When the host receives the HMAC, it can use **the same** hash function with these three ingredients:

- its own copy of the (identical!) symmetric key,
- the package sequence number,
- and the encrypted message.

If the hashed value it computes is the same as the HMAC it received from the client, then we have verified that the connecting computer is the same as the computer who has the symmetric key.

Remember that only the host and client know what the symmetric key is and no other computers do!

So here it doesn’t matter that the host doesn’t know the decoded contents of the encrypted message —the host has still verified the

Donate

The beauty of this approach is that we have not just verified the identity of the client and made sure that the data hasn’t been tampered, but we have done so securely (without **without sharing any private information)**.

Summary: we used a hash function on the client and then on the host to verify data integrity and verify the identity of the client.

Authentication

The final part of the securely communicating with remote computers is:

even if we have generated symmetric keys with the connecting computer and

even if we are using the symmetric keys to communicate securely and

even if the connecting computer is genuinely the client we expect and not an imposter,

then we have set up an SSH session … but does the connecting computer have **permission** to access the contents of the host?

This is called 'authentication': the act of checking permissions and access rights.

There are two ways of checking authentication:

1—Using a password

The client can send the host an (encrypted) message containing a password. The host can decrypt the message and check the password in a database to check if the client has permission to access the specified 'user' (area of the computer). Job done.

2 — Using key-pairs and asymmetric encryption

Earlier, we saw how asymmetric encryption can use two key-pairs to securely generate identical symmetric keys on both the client and

Donate

password.
This is a very high-level approach to the how the process works:

Setting up:

On the client, head to the terminal and use a command to generate a public key and a private key (under the surface it uses 'RSA', a mathematical algorithm) on the client. Copy the **public** key (NOT the private key!) to the clipboard.

I repeat: Copy the **PUBLIC** key (**NOT THE PRIVATE** KEY!) to the clipboard.

Then, in the terminal on the client, use a password to remotely log in to the host. Paste the public key of the client into the appropriate folder on the host alongside any other public keys.

Now, the host has

- It's own public/private key-pair
- The public key of the client

Looking at the section above on the key-exchange algorithm, you can see how the host has all the ingredients it needs to generate a symmetric key!

Challenging:

When the client wants to connect, the host can use issue a 'challenge' by sending a message that has been encrypted (with the host's symmetric key) and say: *'I will only authorise you access if you can decrypt this message!'.*

The client then has

Donate

- its own public and private key
- the public key of the host
- the encrypted message

So now the client has everything needed to generate an (identical) symmetric key … and decrypt the message! It can decrypt the message and send confirmation that is has ‘succeeded’ in the challenge back to the host.

The host is satisfied that the connecting client is authorised and grants permission for access.

Why bother using the second approach?

This is seen as more secure than simply using a password because a bot can use a ‘brute force’ approach to keep using lots of combinations to guess your password, but they will not have they right key-pairs for the second approach to work.

Further reading:

SSH Tutorial for Beginners - How Does SSH Work
*SSH, or Secure Shell, is a remote administration protocol that allows users to control and modify their remote servers…*www.hostinger.com

https://www.udemy.com/the-complete-junior-to-senior-web-developer-roadmap/

Conclusion

SSH is an important tool used to remotely control other computers.

SSH is secure because both computers can encrypt and decrypt

Donate

encryption').

The main steps to initiate an SSH session are:

1. **Setting up an encrypted channel.** Using asymmetric encryption to solve the key-exchange problem which independently generates identical symmetric keys on both computers without sharing any private information.
2. **Verification:** Using hashing on both computers to verify the identity of the connecting computer
3. Verification (again). Using hashing on both computers to verify data integrity hasn't been compromised in transmission.

We can then use SSH to securely send data between the computers. One important use case of this is for **authentication.** Although you can use a password, using asymmetric encryption to check the connecting 'client' has permission to access the 'host' is is seen as more secure.

If you are interested in leveling up your SSH, I seriously recommend this course. I found it really useful to sharpen up some of my skills! (*disclaimer:* I have no links or ties to the author or the platform. I took the course a while ago and found it really good!)

Thanks for reading!

If this article was helpful, tweet it.

Donate

Countinue reading about

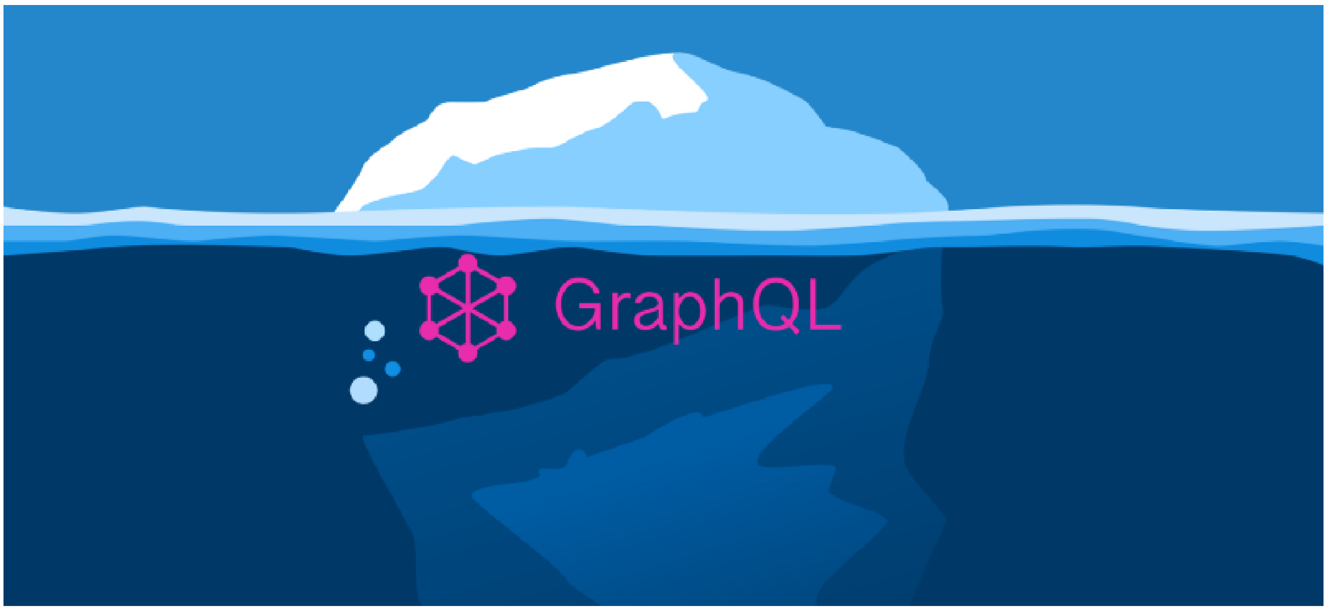
Security

What is TLS? Transport Layer Security Encryption Explained in Plain English

How to Use HTML to Open a Link in a New Tab

Everything You Need to Know About AWS S3

See all 140 posts →



#GRAPHQL

Why GraphQL is the key to staying out of technical debt

BURKE HOLLAND 2 YEARS AGO

Donate

#JAVASCRIPT

How to title case a sentence in JavaScript

DYLAN ATTAL 2 YEARS AGO

Trending Guides

JavaScript Closure	JavaScript Promise
CSS Box Shadow	What is GitHub?
Python List Append	Python Sort List
JavaScript Array Sort	Comments in JSON
Symlink in Linux	What is Kanban?

Donate

What is DNS?

CSS Media Queries

Primary Key SQL
SQL Update Statement

HTML Entities
Excel VBA

Screenshot on PC

LOOKUP in Excel

What is a Proxy Server?

Arrow Function JavaScript

Cat Command in Linux

Remove Duplicates in Excel

CSS Background Image

dllhost.exe COM Surrogate

HTML Background Color

Boolean Algebra Truth Table

CSS Comment Example

Video Chat for Android

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty

Code of Conduct Privacy Policy Terms of Service Copyright Policy