Table of Contents

Abstract

We have built a package of applications that provide security over a wide variety of use cases. We've implemented a root Certificate Authority (CA) and used it to sign our TLS certificate in order to browse our own websites securely. We have a VPN to securely link two computers over a private network, in addition to using IPsec to link all communications at the network level. Finally, we've also setup our own OpenSSH server to securely manage remote machines while preventing potential potential backdoors.

Introduction

To implement this package, we needed to work with many codependent, moving parts. To create our own certificate authority we generated a new (RSA) root key using OpenSSL. We then used this key to generate a valid TLS certificate, which we served using a Go HTTPS server.

For our VPN configuration, we setup a server to use Diffie-Hellman key exchange and RSA keys with a length of 2048 bits. We also generated a new certificate and key to validate the server.

When a client connects, the server uses an implementation of the Diffie-Hellman key exchange and RSA encryption to create a secure tunnel between the two nodes.

In our OpenSSH implementation we used PuTTY key generation to create public and private keys for each user. Each user maintains their own private key and username, which means multiple authorized users can take advantage of the VPN. To create a connection, the users send their username to the server. The server responds with a unique message, encrypted with the user's public key. The message is then decrypted by the user and a checksum for the message is returned to the server. Once the server validates the checksum, a secure connection is established.

To use IPsec, we generated a 30 byte random key using OpenSSL's secure random function. This key is used as our private shared key (PSK). We then distribute the shared key to all interested clients. Once authenticated, users have a secure connection to the server that operates at the internet layer (as opposed to OpenVPN, which operates at the application layer).

Configuration

SSH

We used VirtualBox to run Ubuntu 14.04, which we set up with OpenSSH - a secure and very popular SSH server.

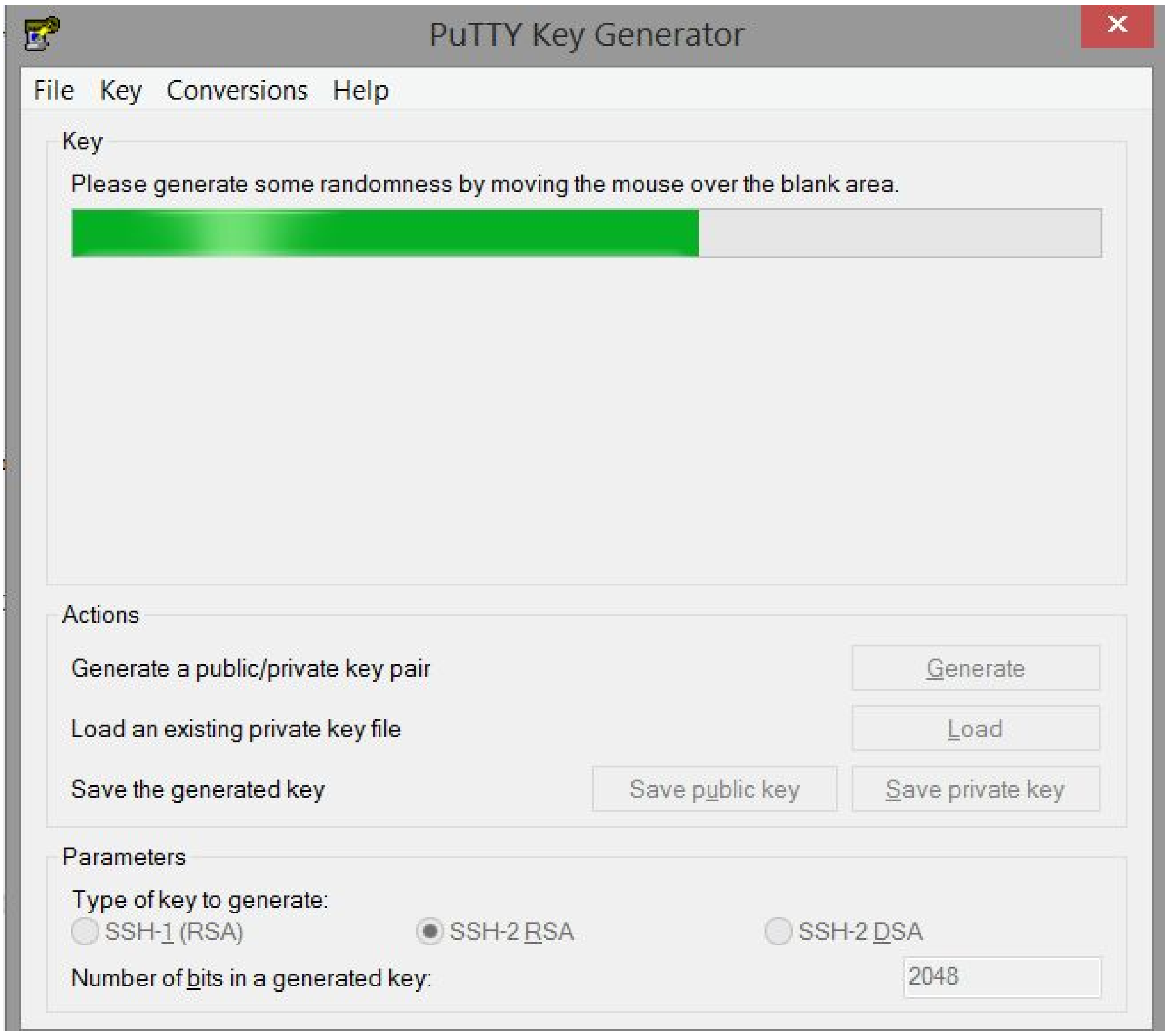
```
sudo apt-get install openssh-server
```

Then we adjusted our SSH config file: /etc/ssh/sshd_config to disable root login and password authentication, using the following settings:

```
PermitRootLogin no

PasswordAuthentication no
```

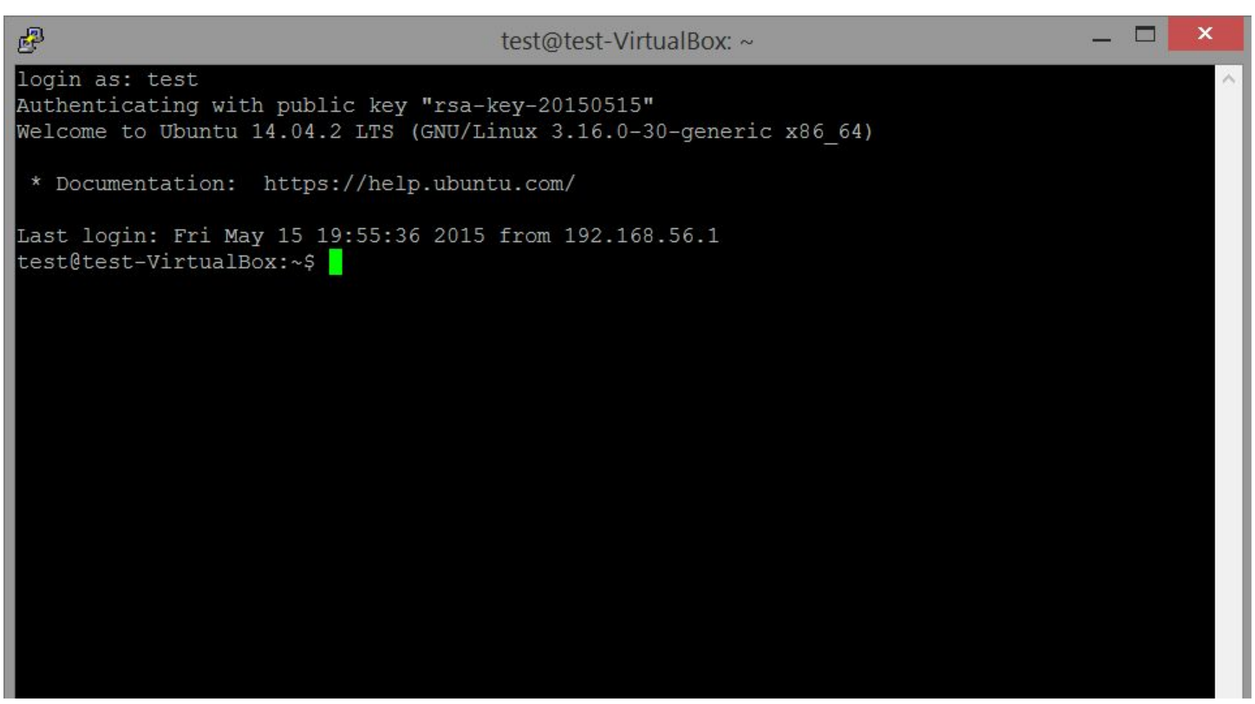
Finally, we generated new SSH keys for each client that wanted to connect. The Windows users among us used PuTTY's key generation, while the *nix users among us used ssh-keygen. The defaults worked great for us, as it generates an RSA key compatible with SSH version 2 (current).



Once this was setup, we could connect to our new server. We retrieved the IP by using ip addr from the VM's shell. Afterward we copied the host machine's public key into the virtual machine's SSH key folder (getting them on the VM initially can be done with either shared folders or dragging and dropping).

```
cp id_rsa* $HOME/.ssh/
```

After that's complete logging in will be as simple as providing the correct username, as the SSH client will automatically use the keys stored in that folder.



OpenVPN

To setup OpenVPN, we first created a new server via Digital Ocean's web admin panel. Referencing a Digital Ocean tutorial, we configured the server as follows:

1. Install the OpenVPN package and the easy-rsa tools

```
apt-get install openvpn easy-rsa
```

2. Copy the sample configuration file as a base

```
gunzip -c /usr/share/doc/openvpn/examples/sample-config-files/server.conf.gz \
    > /etc/openvpn/server.conf
```

3. Adjust the configuration to be more secure.

```
$EDITOR /etc/openvpn/server.conf
```

First, change the key length to 2048 bits to further secure them. Look for the line

```
dh dh1024.pem
```

And change it to:

```
dh2048.pem
```

Uncomment this part so that the VPN server passes the client traffic to the destination:

```
push "redirect-gateway def1 bypass-dhcp"
```

Uncomment this to prevent DNS requests leaking outside the VPN connection (these are the OpenDNS servers):

```
push "dhcp-option DNS 208.67.222.222"

push "dhcp-option DNS 208.67.220.220"
```

Lastly, uncomment this to restrict the user access to the bare minimum (helps to combat privilege escalation attacks):

```
user nobody

group nogroup
```

4. Enable packet forwarding

In order for OpenVPN to work, packet forwarding needs to be enabled on the IPv4 interface:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

We also need to adjust the sysctl.conf file:

```
echo "net.ipv4.ip_forward=1" >> /etc/sysctl.conf
```

5. Now we're going to setup the Uncomplicated FireWall (UFW)

OpenVPN uses UDP on port 1194, so we need to allow traffic via that port:

```
ufw allow ssh

ufw allow 1194/udp
```

We also need to change the DEFAULT_FORWARD_POLICY for ufw:

```
$EDITOR /etc/default/ufw

DEFAULT_FORWARD_POLICY="ACCEPT"
```

6. Add some "before" rules to UFW's config

```
vim /etc/ufw/before.rules
```

Added this to the beginning of the file:

```
# START OPENVPN RULES

# NAT table rules

*nat

:POSTROUTING ACCEPT [0:0]

# Allow traffic from OpenVPN client to eth0

-A POSTROUTING -s 10.8.0.0/8 -o eth0 -j MASQUERADE

COMMIT

# END OPENVPN RULES
```

Next, we need to enable ufw:

```
ufw enable
```

Now we need to create a new certificate authority (or you can use the one we generated before):

```
cp -r /usr/share/easy-rsa/ /etc/openvpn

mkdir -p /etc/openvpn/easy-rsa/keys

$EDITOR /etc/openvpn/easy-rsa/vars
```

We'll use "server" as our key name.

```
export KEY_NAME="server"
```

Now we need to generate a Diffe-Hellman key:

```
openssl dhparam -out /etc/openvpn/dh2048.pem 2048
```

Change into the easy-rsa directory:

```
cd /etc/openvpn/easy-rsa
```

Initialize the CA:

```
source ./vars

./clean-all

./build-ca
```

The last one executes an interactive OpenSSL command that will guide you through the process of creating your certificate.

Finally, you can generate the key and certificate for your OpenVPN server:

```
./build-key-server "$KEY_NAME"
```

Following the interactive prompts, adjust the preferences to your liking.

Move the server certificate and key, and the CA certificate and key into /etc/openvpn:

```
cp /etc/openvpn/easy-rsa/keys/{server.crt,server.key,ca.crt} /etc/openvpn
```

Enable and start the service:

```
service openvpn start

service openvpn enable
```

Now we need to generate a new key for the client:

```
cd etc/openvpn/easy-rsa
```

Next, copy the sample configuration into place:

```
cp /usr/share/doc/openvpn/examples/sample-config-files/client.conf
/etc/openvpn/easy-rsa/keys/client.ovpn
```

Finally, download the following files and setup your host OS's VPN client (Windows uses OpenVPN, Mac users can use Tunnel, etc.):

- `YOUR_CLIENT_NAME.crt`
- `YOUR_CLIENT_NAME.key`
- `client.ovpn`
- `ca.crt`

IPsec

When implementing IPsec, we created a new Ubuntu 14.04 server on DigitalOcean. Using an IPsec guide (https://raymii.org/s/tutorials/IPSEC_L2TP_vpn_with_Ubuntu_14.04.html) as a reference, we configured the server as follows:

We installed OpenSwan and a couple of related packages:

```
apt-get install -y openswan xl2tpd ppp lsof
```

Then we setup a simple firewall using iptables:

```
iptables -t nat -A POSTROUTING -j SNAT --to-source "$(ip addr)" -o eth+
```

Then we enabled kernel-level IP packet forwarding and disabled ICP redirects:

```
echo "net.ipv4.ip_forward = 1" >> /etc/sysctl.conf

echo "net.ipv4.conf.all.accept_redirects = 0" >> /etc/sysctl.conf

echo "net.ipv4.conf.all.send_redirects = 0" >> /etc/sysctl.conf

echo "net.ipv4.conf.default.rp_filter = 0" >> /etc/sysctl.conf

echo "net.ipv4.conf.default.accept_source_route = 0" >> /etc/sysctl.conf

echo "net.ipv4.conf.default.send_redirects = 0" >> /etc/sysctl.conf

echo "net.ipv4.icmp_ignore_bogus_error_responses = 1" >> /etc/sysctl.conf
```

This also required some additions to the ipv4 conf files:

```
for vpn in /proc/sys/net/ipv4/conf/*;

do

    echo 0 > $vpn/accept_redirects;

    echo 0 > $vpn/send_redirects;

done
```

Apply them via `sysctl -p`.

Now that we've setup our network interfaces, we need to adjust our IPsec configuration file (note: change OUR_SERVER_IP_ADDRESS to the IP address of your server):

```
tee /etc/ipsec.conf << EOF

config setup
```

```
dumpdir=/var/run/pluto/

#in what directory should things started by setup (notably the Pluto daemon) be allowed to dump core?

nat_traversal=yes

#whether to accept/offer to support NAT (NAPT, also known as "IP Masquerade") workaround for IPsec

virtual_private=%v4:10.0.0.0/8,%v4:192.168.0.0/16,%v4:172.16.0.0/12,%v6:fd00::/8,%v6:fe80::/10

#contains the networks that are allowed as subnet= for the remote client. In other words, the address ranges that may live behind a NAT router through which a client connects.


protostack=netkey

#decide which protocol stack is going to be used.

force_keepalive=yes

keep_alive=60

# Send a keep-alive packet every 60 seconds.


conn L2TP-PSK-noNAT

authby=secret

#shared secret. Use rsasig for certificates.

pfs=no

#Disable pfs

auto=add

#the IPsec tunnel should be started and routes created when the ipsec daemon itself starts.

keyingtries=3

#Only negotiate a conn. 3 times.


ikelifetime=8h

keylife=1h


ike=aes256-sha1,aes128-sha1,3des-sha1

phase2alg=aes256-sha1,aes128-sha1,3des-sha1

# https://lists.openswan.org/pipermail/users/2014-April/022947.html

# specifies the phase 1 encryption scheme, the hashing algorithm, and the diffie-hellman group. The modp1024 is for Diffie-Hellman
```

```
type=transport

#because we use l2tp as tunnel protocol


left= OUR_SERVER_IP_ADDRESS

#fill in server IP above


leftprotoport=17/1701

right=%any

rightprotoport=17/%any


dpddelay=10

# Dead Peer Dectection (RFC 3706) keepalives delay

dpdtimeout=20

#  length of time (in seconds) we will idle without hearing either an R_U_THERE poll from our peer, or an R_U_THERE_ACK reply.

dpdaction=clear

# When a DPD enabled peer is declared dead, what action should be taken. clear means the eroute and SA with both be cleared.

EOF
```

Next, we need to add a valid secret:

```
echo "$(ip addr) %any: PSK \"$(openssl rand -hex 30)\"" >> /etc/ipsec.secrets
```

This will create a line in our secrets file that looks like this:

```
45.55.208.96 %any: PSK "1066318118bffb80f529294f9021fc9d4c466dc375b05ba38f5cb2c587c8"
```

Let's make sure our IPsec configuration is valid by running `ipsec verify`.

Next, we need to setup xl2tpd

```
tee /etc/xl2tpd/xl2tpd.conf << EOF

[global]

ipsec saref = yes

saref refinfo = 30


;debug avp = yes

;debug network = yes

;debug state = yes

;debug tunnel = yes
```

```
[lns default]

; the range of IPs to give to incoming clients

ip range = 172.16.1.30-172.16.1.100

; the local IP for the VPN server

local ip = 172.16.1.1

; disallow PAP authentication

refuse pap = yes

require authentication = yes

;make sure this is turned off in production

;ppp debug = yes

pppoptfile = /etc/ppp/options.xl2tpd

length bit = yes

EOF
```

Next, we need to change some more PPP options (specifically the xl2tpd ones):

```
tee /etc/ppp/options.xl2tpd << EOF

require-mschap-v2

ms-dns 8.8.8.8

ms-dns 8.8.4.4

auth

mtu 1200

mru 1000

crtscts

hide-password

modem

name l2tpd

proxyarp

lcp-echo-interval 30

lcp-echo-failure 4

EOF
```

In the above example, ms-dns is the DNS address to give to the client. We're using Google's public DNS for now. Proxyarp is a setting that controls IPsec's integration with the system's address resolution protocol (ARP) table. When enabled, it means IPsec peers will appear to other systems to be on the local ethernet. The name l2tpd clause is a reference to the contents of the PPP authentication file.

Adding users

Every user should be defined in the /etc/ppp/chap-secrets file. Below is an example with a username of "test" and a password of "12345Ll." that's allowed access from any IP address.

```
# Secrets for authentication using CHAP

# client    server   secret                 IP addresses

test        l2tpd    12345Ll.                *
```

To make sure everything has the newest config files restart openswan and xl2tpd:

```
service ipsec restart

service xl2tpd restart
```

Certificate authority

We used the following command to create a root CA certificate:

```
openssl genrsa -out rootCA.key 2048
```

We then signed our new certificate (newcrt.crt), using the following command:

```
openssl req -x509 -new -nodes -key rootCA.key -days 365 -out newcrt.crt
```

Next, we need to create a certificate for the site in question:

```
# generate a key for the website

openssl genrsa -out host.key 2048

# generate a CSR from the key

openssl req -new -key host.key -out host.csr

# generate a signed cert from the CSR

openssl x509 -req -in host.csr -CA rootCA.crt -CAkey rootCA.key -
CAcreateserial -out host.crt -days 365
```

This way we can configure any normal web server to use the certificate in HTTPS communication. We setup a simple HTTPS web server in Go to verify this. Here's the source:

```go
package main

import (
	"flag"
	"fmt"
	"log"
	"net/http"
	"os"
)

// Fire takes a set of CLI arguments and returns an exit code (and maybe an
// error).
func fire(args []string) (int, error) {
	fset := flag.NewFlagSet("", flag.ExitOnError)

	// setup some command line flags
	var address, certfile, keyfile string

	fset.StringVar(&address, "address", ":3003", "the address to listen on")
	fset.StringVar(&keyfile, "keyfile", "host.key", "the keyfile to use")
	fset.StringVar(&certfile, "certfile", "host.crt", "the certfile to use")
```

```go
	// attempt to parse the flags
	if err := fset.Parse(args); err != nil {
		return 1, err
	}

	fmt.Println("Listening to HTTPS on address: " + address)

	// this blocks unless the listen fails (in which case we return a failing
	// exit code and the resulting error).
	return 1, http.ListenAndServeTLS(address, certfile, keyfile,
		// create an anonymous http handler
		http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
			rw.Header().Set("Content-Type", "text/plain")
			if _, err := rw.Write([]byte("Hello from TLS!")); err != nil {
				log.Printf("couldn't write to response: %v", err)
			}
		}))
}

// run the actual program, handling errors as appropriate.
func main() {
	code, err := fire(os.Args)
	if err != nil {
		fmt.Errorf("Fatal error encountered: %s", err.Error())
	}

	os.Exit(code)
}
```

Security Protocols

SSH

SSH is organized as three protocols that run on top of TCP. These three protocols include the User Authentication Protocol, Connection Protocol and the Transport Layer Protocol. The User Authentication Protocol provides the means for the user to be authenticated to the server requiring a username, service name and method name. The Connection Protocol assumes that that authentication is in use and runs on top of the SSH Transport Protocol. This protocol multiplexes multiple logical channels over a single SSH connection. The Transport Layer Protocol provides several services including data integrity, data confidentiality, server authentication and compression.

VPN

A VPN is used to transmit data securely through an encrypted tunnel between a network and a remote user. The information transmitted through the tunnel cannot be read by anyone else because system secures both the outside network and the private network.

Usually, the first step to security is using a firewall to authenticate a connection between the client and the host. Secondly the next step is to encrypt the data so only the receiver can decrypt it. A bridged VPN allows the clients to appear as though they're connected through a local area network (LAN). VPNs can generally run on both layer 2 (Data Link) or layer 3 (Network).

In layer 2 VPN, VPNs virtualize the datalink layer. L2 frames are transported between locations. It's similar to a cable connecting two switches. The VPN has to handle all the basic properties of an Ethernet network such as learning MAC addresses and replicating broadcast and multicast frames. This is done through tunneling frames over the VPN. Conceptually L2 VPNs are simpler than L3 VPNs. On the downside however, it suffers from all the security issues and instabilities of L2.

In layer 3 VPN, VPNs virtualize the network layer so that it's possible to route your "internal" networks over a public infrastructure. Each side of the connection is on a different subnet, and IP packets are routed through the VPN. The design is more scalable than a L2 VPN and offers more security. It however offers less transparency.

IPsec

Internet Protocol Security secures internet protocol communication by authenticating and encrypting each IP packet during host-to-host, network-to-host, or network-to-network traffic using either its transport or tunneling mode. IPSec supports network-level peer authentication, data origin authentication, data integrity, confidentiality, and replay protection. IPSec is an open standard that uses Authentication Headers, Encapsulating Security Payloads, and Security Associations protocols to perform various functions.

Authentication Header protects the IP payload and all header fields (IPv4) of an IP datagram except for the mutable fields. It operates directly on top of IP using IP protocol number 51, and guarantees connectionless integrity, data origin authentication of IP packets, and replay protection using the sliding window technique and discarding old packets.

Encapsulating Security Payload provides origin authenticity, integrity, and confidentiality of packets. It supports encryption only and authentication only configurations, but doesn't provide integrity and authentication for the entire IP packet in transport mode like Authentication Header. Since tunnel mode encapsulates the entire IP packet with a new packet header, protection is provided for all encapsulated data. SEP operates using IP protocol number 50.

To decide what protection to provide for an outgoing packet, IPSec uses the Security parameter index and the destination address in the packet header to identify the security association of the packet. Similarly, incoming packets are checked for the necessary decryption and verification keys from the Security Association Database. All security associations are established using the Internet Security Association and Key Management Protocol (ISKAMP) which is implemented by manual configuration using shared secrets, Internet Key Exchange (IKE and IKE2), Kerberized Internet Negotiation of Keys (KINK) and IPSECKEY DNS records.

The cryptographic algorithms defined for use with IPSec include HMAC_SHA1/SHA2 (integrity and authenticity), 3DES-CBC (confidentiality), AES-CBC (confidentiality), and AES-GCM (confidentiality and authentication).

SSL Certificate

SSL Certificates are small data files that digitally link a cryptographic key to an organization's details. The web server activates the https protocol and allows a secure connection from the server to the browser when the certificate is present. In order to implement this security, the web server creates two keys, a public and private key. The public key is contained in a Certificate Signing Request (CSR). The CSR is validated by the Certificate Authority and issues a SSL Certificate. This certificate is matched by the web server to a private key. Once the match is validated a secure connection can be established.

TLS

Transport layer Security provides three essential services, encryption, authentication and integrity. To establish a secure connection TLC, which runs over TCP, requires a three-way handshake. A public/private key exchange, Diffie-Hellman or RSA, are used to share a symmetric private key. Server and Client verify Mac addresses after which a tunnel is established where application data may be passed.

Testing and Field Evaluation

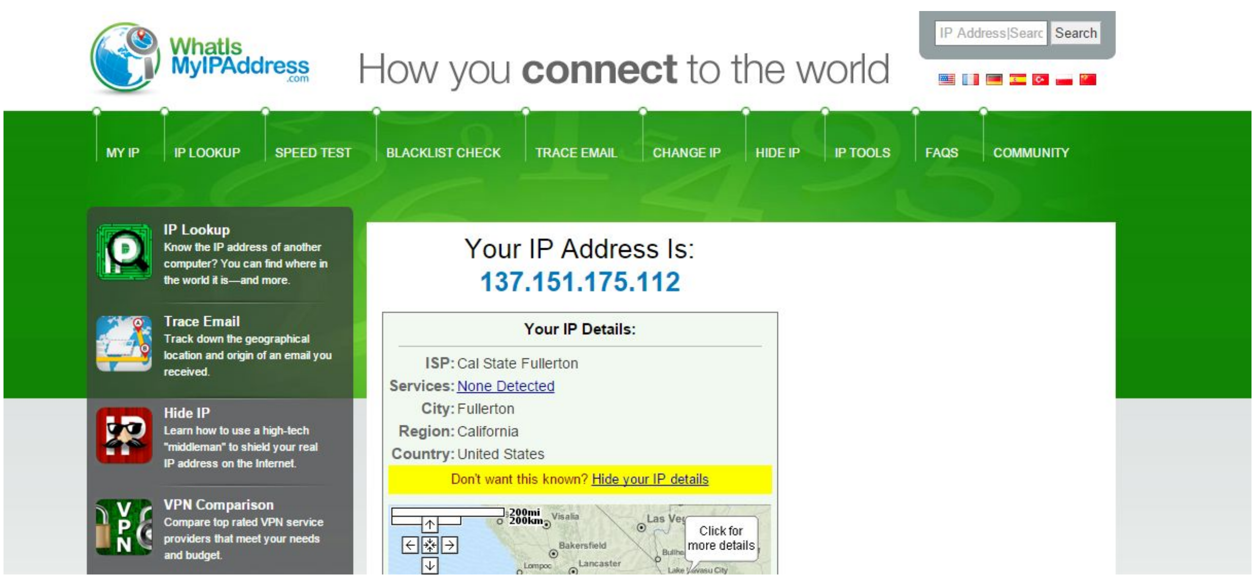
SSH

In order to test SSH, we attempt to login in with a username that is not registered with the server. The server refused our key and denies login to unknown users.

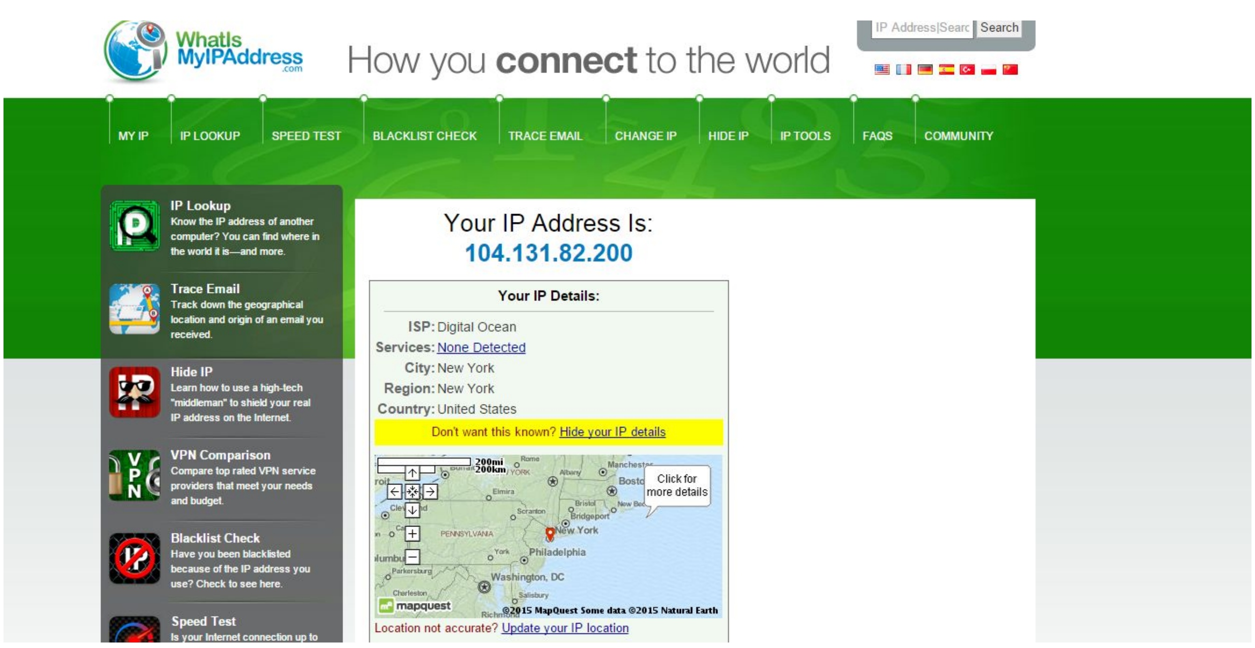
OpenVPN

First we test to see the initial IP address. The easiest way to demonstrate this is by visiting https://whatismyipaddress.com. Then, once we've connected the OpenVPN link we should see a different IP address (the server's) appear.

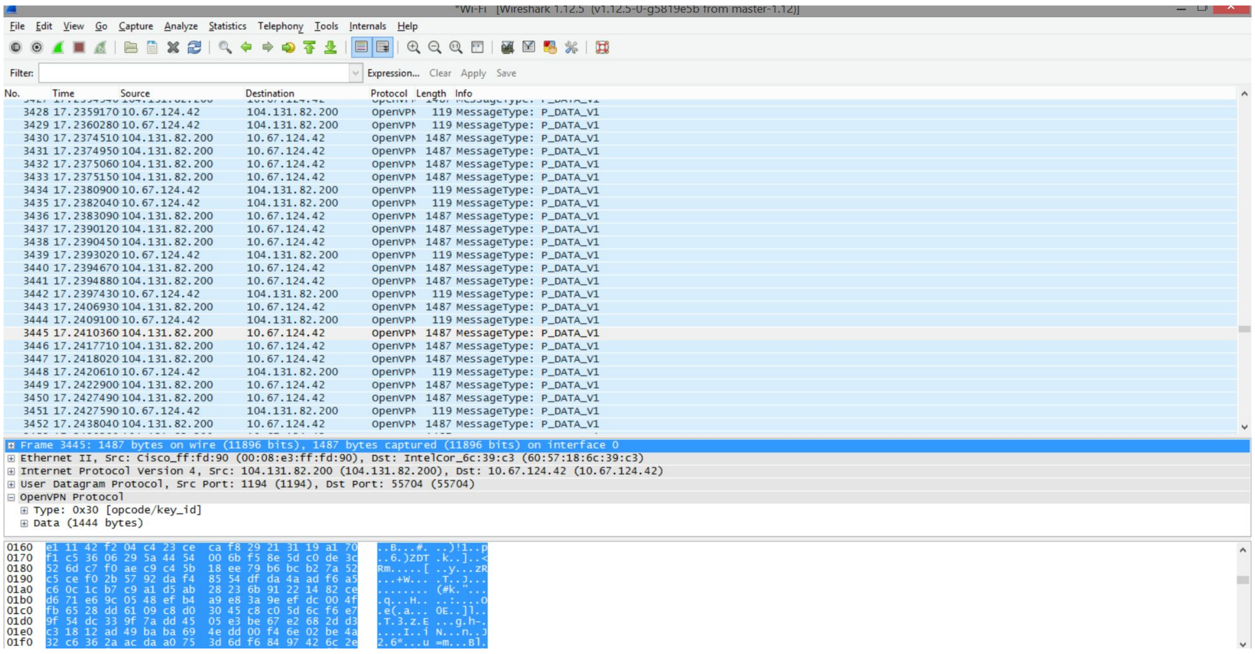
Before connecting to OpenVPN:



After connecting to OpenVPN:



Next, we used Wireshark to confirm our observation:

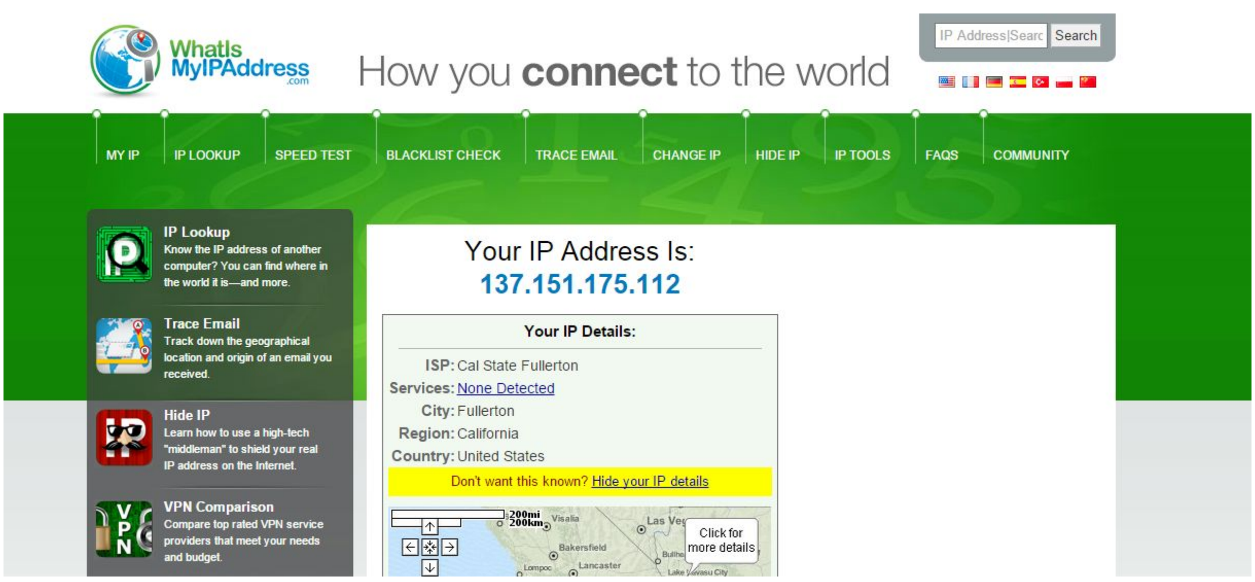


As you can see, our network traffic is being handled by OpenVPN.

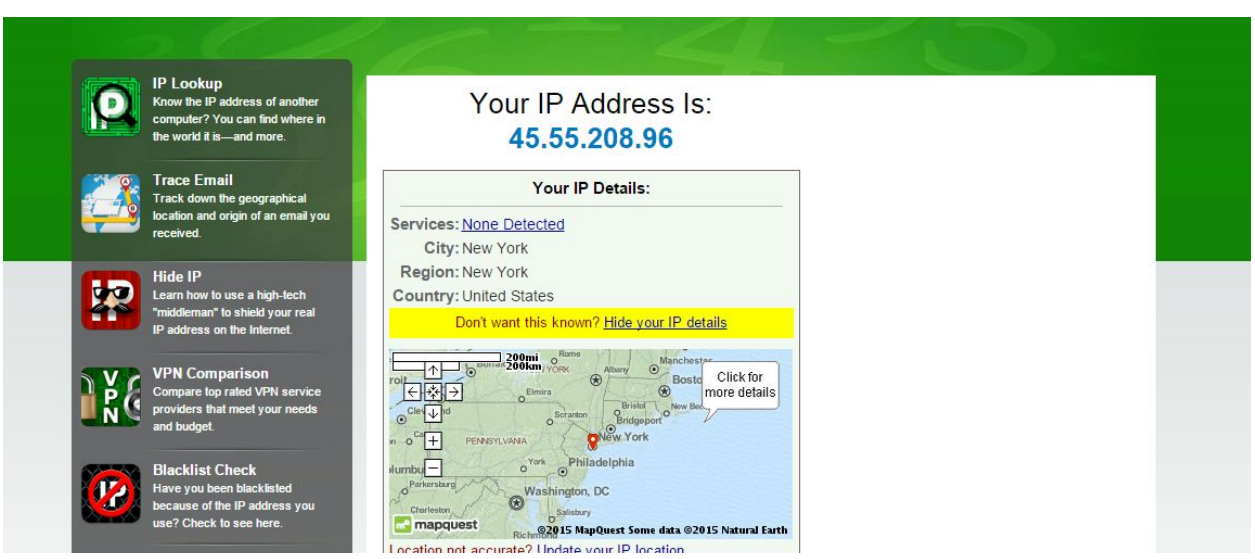
IPsec

First we test to see the initial IP address. The easiest way to demonstrate this is by visiting https://whatismyipaddress.com. Then, once we've connected the IPsec link we should see a different IP address (the server's) appear.

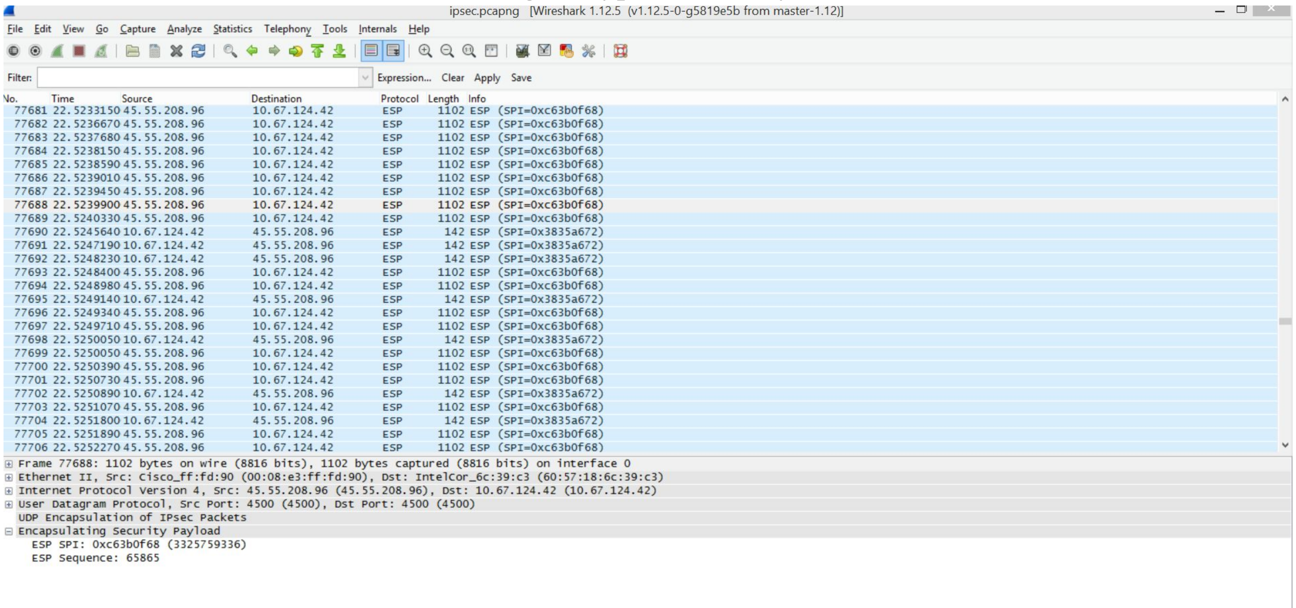
Before connection:



After connection:



Next, let's confirm that the traffic is being encrypted correctly:



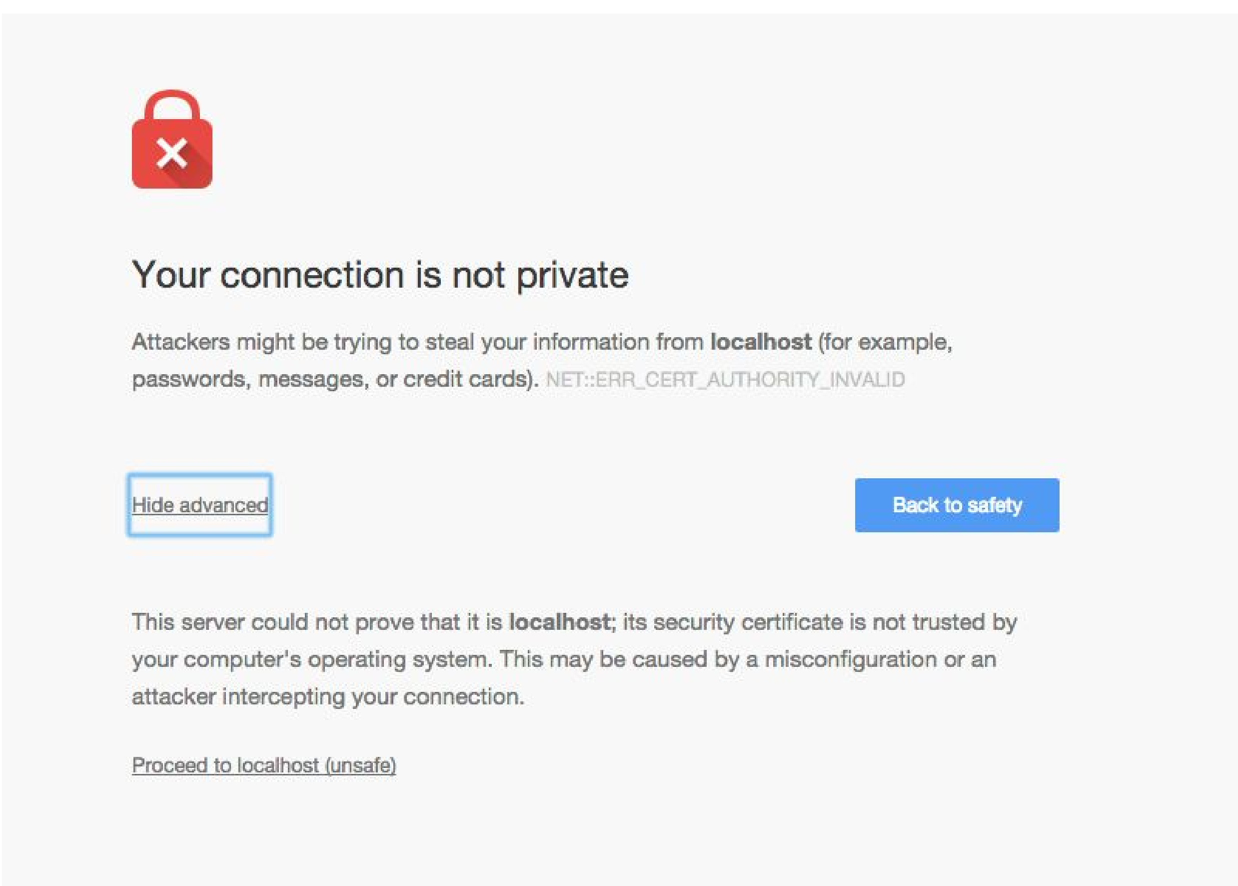
The Wireshark results confirm our previous test: our traffic is being encrypted in ESP mode.

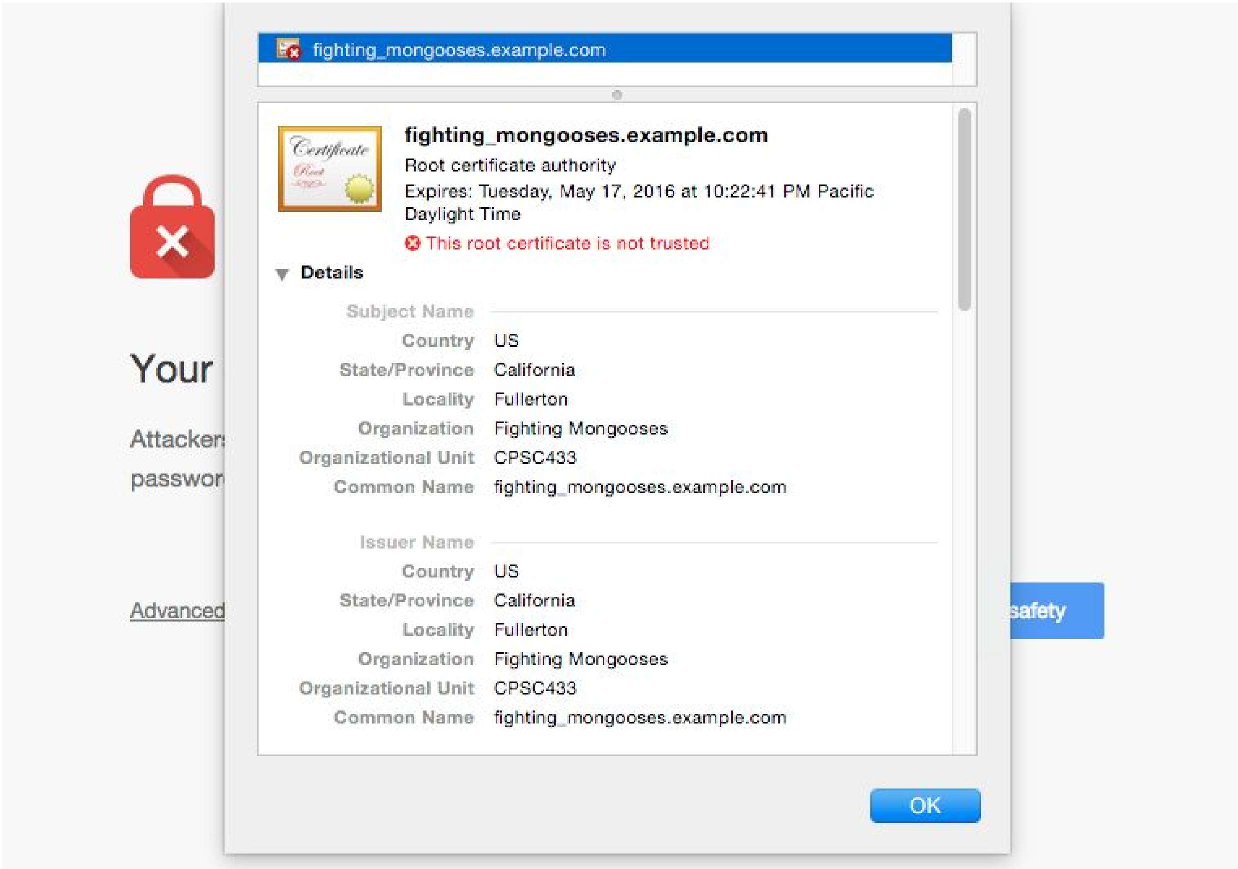
Certificate authority

The first step to testing the certificate authority is to run the HTTPS server from the same directory the CA files were generated:

```
go build -o server main.go && ./server
```

Once that's running, you can visit https://localhost:3003 and verify that the connection is secure through the browser's certificate details panel:





As you can see, a new (untrusted) root CA has been created. Manually adding the certificate to the operating system's whitelist is beyond the scope of this guide.

Conclusion

We have used a number of different protocols to implement security within a system. We have used both symmetric and asymmetric encryption in order to create secure tunnels to send and receive data confidentially. The Certificate Authority confirms the authenticity of the server allowing client and server to maintain a trusted and secure connection using https. The Virtual Private Network allows a user to remotely access private networks. This allows the user to access remote resources as if they were on the local system, while securely encrypting all packets that are sent and received. SSH encrypts the data transferred between two systems even if the systems are on the same network, unlike VPN. IPSec provides a secure tunnel to transport packets securely between two systems protecting packets at the IP packet layer. All of these tools allow us to share and extract data in a number of different situations.

References

- https://www.ubuntu.com/server
- https://raymii.org/s/tutorials/IPSEC_L2TP_vpn_with_Ubuntu_14.04.html
- https://www.digitalocean.com/community/tutorials/how-to-configure-ssh-key-based-authentication-on-a-linux-server
- https://www.digitalocean.com/community/tutorials/how-to-set-up-multiple-ssl-certificates-on-one-ip-with-apache-on-ubuntu-12-04
- https://www.digitalocean.com/community/tutorials/how-to-set-up-apache-with-a-free-signed-ssl-certificate-on-a-vps
- https://www.digitalocean.com/community/tutorials/how-to-create-an-ssl-certificate-on-nginx-for-ubuntu-14-04
- https://www.digitalocean.com/community/tutorials/how-to-create-an-ssh-ca-to-validate-hosts-and-clients-with-ubuntu
- https://www.digitalocean.com/community/tutorials/how-to-set-up-an-openvpn-server-on-ubuntu-14-04
- https://github.com/ciarand/cryptography-practical-skills
- https://help.ubuntu.com/community/SSH/OpenSSH/Configuring
- https://thepcspy.com/read/making-ssh-secure/
- http://security.stackexchange.com/questions/20803/how-does-ssl-tls-work
- https://security.stackexchange.com/questions/23227/how-does-ssh-public-key-authentication-
- work
- https://www.digitalocean.com/community/tutorials/how-to-set-up-an-openvpn-server-on-ubuntu-14-04