

Elevating Website Security: Generating and Deploying a Self-Signed SSL Certificate with HashiCorp's Vault and NGINX

Welcome to the comprehensive guide on using HashiCorp's Vault and NGINX to host a website on your local host with a self-signed certificate. This tutorial will walk you through the entire process of securely managing secrets with HashiCorp's Vault while setting up NGINX as a web server to host your website. By utilizing a self-signed certificate, we will establish a secure HTTPS connection for your local website, ensuring encrypted communication between the client and the server.

Throughout this guide, we will cover all the necessary steps, including generating a self-signed certificate for your local host, configuring NGINX to utilize HTTPS, importing the certificate into HashiCorp's Vault for secure storage, retrieving the certificate from Vault, and finally, hosting the website using NGINX with the self-signed certificate. By following these instructions, you will gain valuable insights into securely managing secrets, setting up a web server, and securing your website with a self-signed certificate.

Whether you are a developer testing a website locally, setting up a development environment, or simply exploring the process of managing secrets and deploying secure websites, this guide will provide you with the knowledge and practical steps to accomplish your goal. So, let's get started and dive into the exciting world of using HashiCorp's Vault and NGINX to host a website on your local host with a self-signed certificate!

Section 1: Pre-requisite knowledge

1.1 Data handling across the internet and SSL Certificates

What are asymmetric keys and how are they used to handle data across the internet?

Asymmetric keys, also known as public-key cryptography, are cryptographic keys that come in pairs: a public key and a private key. This form of encryption is widely used across the internet to secure data transmission, ensure confidentiality, integrity, and authenticity of communication. Here's how asymmetric keys are used to handle data across the internet:

- **Key Generation:** The process begins by generating a pair of asymmetric keys. The private key is kept secret by the owner, while the corresponding public key is shared with others.
- **Encryption:** To send encrypted data, the sender uses the recipient's public key to encrypt the message. This ensures that only the intended recipient, who possesses the corresponding private key, can decrypt and access the original message.

- **Decryption:** Upon receiving the encrypted message, the recipient uses their private key to decrypt it and retrieve the original content. Since the private key is kept secret, only the recipient can perform this decryption operation.
- **Digital Signatures:** Asymmetric keys are also used for digital signatures, which provide data integrity and authentication. To sign a message, the sender uses their private key to create a unique digital signature, which is then appended to the message. The recipient can verify the authenticity of the message by using the sender's public key to validate the digital signature.
- **Authentication and Key Exchange:** Asymmetric keys are used in protocols like Transport Layer Security (TLS) to establish secure connections between clients and servers. During the TLS handshake, the server presents its public key to the client, which is then used for encrypting the session key. The client encrypts the session key with the server's public key, ensuring only the server can decrypt it with the corresponding private key. This shared session key is then used for symmetric encryption, which is more efficient for bulk data transmission. Asymmetric key encryption is computationally intensive compared to symmetric key encryption, so it is often used in combination with symmetric encryption. The asymmetric keys are primarily used for securely exchanging symmetric keys, while the symmetric keys handle the actual encryption and decryption of the data.

What is an SSL Certificate?

An SSL certificate, also known as a Secure Sockets Layer certificate, is a digital certificate that provides authentication and encryption for secure communication over the internet. SSL certificates are used to establish secure connections between web servers and clients, ensuring that data transmitted between them remains confidential and protected from unauthorized access.

Here are some key aspects of SSL certificates:

- **Encryption:** SSL certificates facilitate encryption of data during transmission. When a website has an SSL certificate installed, the communication between the web server and the client's browser is encrypted, making it difficult for unauthorized individuals to intercept and understand the transmitted data. This encryption helps protect sensitive information, such as login credentials, credit card details, and personal data.
- **Authentication:** SSL certificates provide authentication, assuring visitors that they are communicating with the intended website and not an imposter or a malicious entity. The certificate verifies the identity of the website and associates it with a cryptographic key pair. This authentication is achieved through the trusted Certificate Authorities (CAs) that issue SSL certificates after verifying the ownership and legitimacy of the website.
- **Trust Indicators:** Websites with SSL certificates display trust indicators to visitors, such as a padlock icon in the browser's address bar, the "https://" protocol prefix, or a green address bar in some cases. These indicators signal that the website has implemented SSL encryption and that the connection is secure. They instill confidence in visitors, reassuring them that their data is protected during their interaction with the website.
- **Types of SSL Certificates:** SSL certificates come in different types, such as domain-validated (DV), organization-validated (OV), and extended validation (EV) certificates. The level of validation and information displayed in the certificate varies among these types. EV certificates, for example, involve a rigorous validation process and display the organization's name prominently in the browser's address bar, providing an enhanced level of trust to users.

- **Certificate Authorities (CAs):** SSL certificates are issued by trusted Certificate Authorities, which are organizations that have been authorized to validate and issue certificates. Web browsers and operating systems maintain a list of trusted CAs, and if a website's SSL certificate is signed by one of these trusted CAs, it is considered valid and trusted by the browser.

SSL certificates play a vital role in securing online transactions, protecting sensitive data, and establishing trust between websites and visitors. They are an essential component of modern web security and are widely used to ensure secure communication on websites, e-commerce platforms, email servers, and other online services.

When it comes to identity, what issue does asymmetric encryption hold and how does the use of certificates handle this problem?

One problem that arises is the issue of trust and verifying the authenticity of an entity's identity in the digital realm. The use of certificates helps address this problem by providing a trusted means of verifying the identity of individuals, organizations, or devices. The main problem with identity in the digital world is that anyone can claim to be someone or something they are not. This creates a challenge when trying to establish trust and ensure that you are communicating with the intended party. For example, when accessing a website, you want to be certain that you are connecting to the legitimate website and not an imposter trying to steal your information. This is where certificates come into play. Certificates are digital documents issued by trusted third-party entities known as Certificate Authorities (CAs). These certificates bind an entity's identity (such as a website or an individual) to a cryptographic key pair (consisting of a public key and a private key).

Certificates work in the following way to handle the problem of identity:

- **Certificate Issuance:** The entity seeking a certificate generates a key pair and creates a Certificate Signing Request (CSR) containing their public key and identity information. The CSR is then sent to a trusted CA.
- **Identity Verification:** The CA verifies the identity of the entity through various validation processes. This can involve checking domain ownership, verifying legal documents, or conducting background checks for individuals.
- **Certificate Signing:** If the CA is satisfied with the identity verification, they digitally sign the CSR using their private key. This creates a digital certificate that binds the entity's identity to their public key.
- **Certificate Distribution:** The CA provides the signed certificate back to the entity. The certificate contains information about the entity's identity, the public key, and the CA's digital signature.
- **Certificate Validation:** When the entity presents its certificate to other parties (such as website visitors), the certificate is checked for validity. This involves verifying the CA's digital signature using the CA's public key, ensuring the certificate has not expired, and checking for any revocation status.

By using certificates, the problem of identity is addressed in the following ways:

1. **Trust:** Certificates are issued by trusted CAs, which are organizations that have established their credibility and integrity. By relying on these trusted authorities, users can have confidence in the authenticity of the identities presented in certificates.

2. **Authentication:** Certificates provide a way to authenticate the identity of an entity. The CA's digital signature on the certificate assures the recipient that the public key presented by the entity indeed belongs to the claimed identity.
3. **Non-Repudiation:** Certificates support non-repudiation, which means that the entity cannot deny their association with the certificate. The digital signature from the CA provides a proof of the entity's identity, preventing them from disowning their actions or communications.

Certificates play a crucial role in addressing the problem of identity in the digital world by providing trusted means of verifying identities. They establish trust, authenticate entities, and enable non-repudiation, enhancing security and enabling secure communication across various online platforms.

Who are these issuing authorities and why will I trust them?

The issuing authorities of SSL certificates, also known as Certificate Authorities (CAs), are trusted organizations responsible for verifying the identity and authenticity of entities requesting SSL certificates. These CAs play a crucial role in establishing trust in the digital certificate ecosystem.

Here's how trust in Certificate Authorities is established:

- **Trusted Root CAs:** Web browsers and operating systems maintain a list of pre-installed trusted root CAs. These root CAs are considered inherently trusted, as they have undergone rigorous vetting and auditing processes to ensure their reliability and security. Examples of well-known root CAs include Digicert, Symantec, GlobalSign, Let's Encrypt, and Comodo.
- **Certification Authorities/Browser Forum:** The Certification Authorities/Browser (CA/B) Forum is an industry consortium that sets standards and guidelines for CAs and browser vendors. The forum defines policies and practices that CAs must follow to maintain trust. These include requirements for certificate issuance, validation procedures, and security measures.
- **Certificate Issuance Process:** CAs follow specific processes to issue SSL certificates. They verify the ownership and control of the domain or organization requesting the certificate. This process typically involves verifying domain registration records, conducting email or phone verification, or validating legal documentation for organizations. The CAs must adhere to the validation guidelines established by the CA/B Forum to ensure consistent and reliable certificate issuance.
- **Auditing and Compliance:** CAs undergo regular audits and assessments to validate their compliance with industry standards and best practices. These audits verify that CAs are following the prescribed procedures, maintaining security controls, and protecting their private key infrastructure. Audit reports, such as WebTrust or ETSI standards, provide evidence of the CA's compliance.
- **Certificate Revocation:** In cases where a CA has issued a compromised or fraudulent certificate, or if a certificate needs to be revoked for any reason, CAs maintain Certificate Revocation Lists (CRLs) or use Online Certificate Status Protocol (OCSP) to indicate the revoked status of certificates. Web browsers and other relying parties regularly check the revocation status of certificates to ensure their validity.
- **Browser and Operating System Trust:** Web browsers and operating systems trust the root CAs by including their root certificates in their default trust stores. This trust is based on the rigorous vetting and auditing processes that the root CAs have undergone. Browsers and operating systems periodically update their trust stores to add or remove CAs based on their adherence to standards and practices.

It's important to note that while CAs play a critical role in establishing trust, they are not infallible. There have been instances where CAs have issued certificates erroneously or under fraudulent circumstances. However, the industry has mechanisms in place to detect and address such incidents, such as browser blacklisting of compromised certificates and public scrutiny.

Trust in Certificate Authorities is established through the inclusion of their root certificates in trusted browser and operating system stores, adherence to industry standards and audits, validation processes, and the ability to revoke compromised certificates. These measures collectively help ensure the integrity and reliability of SSL certificates and maintain trust in the digital certificate ecosystem.

Can I be my own Certifying Authority?

Yes, it is possible to become your own Certifying Authority (CA) and issue your own digital certificates. This is known as setting up a private CA. However, it's important to note that the certificates issued by a private CA will not be automatically trusted by web browsers and operating systems, as they are preconfigured to trust specific root CAs.

Here are the basic steps involved in setting up your own private CA:

- **Generate a Root Certificate:** The first step is to generate a self-signed root certificate. This certificate will act as the root of trust for your private CA. You can use tools like OpenSSL to generate the root certificate.
- **Establish Certificate Policies and Procedures:** Define the policies and procedures for your private CA, including the validation processes, certificate issuance guidelines, and security measures. This helps ensure consistency and security in the certificate issuance process.
- **Set up Certificate Authority Infrastructure:** Create the infrastructure for your private CA, which typically includes a secure server or hardware module to store and protect the private key associated with the root certificate. This infrastructure is critical for maintaining the security and integrity of your private CA.
- **Issue and Manage Certificates:** With your private CA infrastructure in place, you can start issuing digital certificates. These certificates can be used for various purposes such as securing web servers, email communication, or internal systems. You will need to follow your defined policies and procedures for verifying identities and issuing certificates.
- **Distribute and Install Trust:** Since your private CA is not pre-trusted by default, you would need to manually distribute and install the root certificate of your private CA to the clients or systems that need to trust your certificates. This ensures that they recognize and trust your private CA as a valid authority.

It's important to understand that while you can set up your own private CA, the certificates issued by it will only be trusted within the specific systems or environments where you have distributed and installed the root certificate. They will not be automatically recognized as trusted by web browsers or other systems unless you go through a process of getting your root certificate included in their trust stores. Setting up and managing a private CA requires careful consideration of security practices, infrastructure, and trust establishment. It is typically more suitable for internal use within organizations or closed systems rather than for publicly trusted certificates used on the internet.

What are the benefits in me being my own CA?

Being your own Certifying Authority (CA) offers several benefits:

- **Flexibility and Control:** As your own CA, you have complete control over the issuance and management of digital certificates. You can tailor the certificate issuance process to meet your specific needs and define your own policies and procedures. This level of flexibility allows you to adapt the CA operations to the unique requirements of your organization or environment.
- **Cost Savings:** By operating your own CA, you eliminate the need to rely on third-party commercial CAs for obtaining certificates. This can result in significant cost savings, especially if you require a large number of certificates or have long-term certificate needs.
- **Rapid Certificate Issuance:** With your own CA, you can issue certificates quickly without having to wait for approval or validation from external CAs. This is particularly beneficial in fast-paced environments where certificates need to be provisioned rapidly to support new services, systems, or development workflows.
- **Enhanced Security:** Being your own CA allows you to enforce strong security measures throughout the certificate lifecycle. You have control over the security of the private keys associated with the CA, ensuring they are stored and managed in a highly secure manner. This reduces the risk of private key compromise and unauthorized certificate issuance.
- **Custom Trust:** By distributing and installing your CA's root certificate within your organization or specific systems, you establish a custom trust model. This means that the certificates issued by your CA will be automatically trusted within your defined trust boundaries. It provides a higher level of assurance and control over the trustworthiness of the certificates used within your environment.
- **Internal PKI:** Running your own CA enables the establishment of an internal Public Key Infrastructure (PKI) within your organization. This allows you to issue and manage certificates for internal services, systems, and users. It facilitates secure communication, authentication, and data protection within your organization's infrastructure.
- **Isolation and Privacy:** By having your own CA, you can ensure that your organization's certificates and cryptographic operations remain isolated from external entities. This helps maintain the privacy and confidentiality of your internal communications and prevents the reliance on external CAs that may have access to your certificate issuance activities.

You're also required to note that operating your own CA also comes with responsibilities, such as implementing strong security practices, ensuring proper certificate management, and maintaining the integrity of the CA infrastructure. However, the benefits of being your own CA can outweigh these considerations, particularly for organizations that require greater control, customization, and cost efficiency in managing their digital certificates.

1.2 Structure and working of HashiCorp's Vault

What is HashiCorp's Vault?

HashiCorp's Vault is an open-source software tool designed for securely managing and storing sensitive data, such as passwords, API keys, cryptographic keys, and other secrets. It provides a centralized solution for managing secrets across different applications and infrastructure.

Key features and concepts of HashiCorp Vault include:

- **Secret Management:** Vault allows you to securely store and manage secrets in a central repository. Secrets can be stored, accessed, and revoked programmatically or via a user-friendly interface. Vault supports various secret types, including key-value pairs, certificates, database credentials, and dynamic secrets.
- **Encryption and Access Control:** Vault encrypts and protects secrets at rest using encryption algorithms and employs access control mechanisms to enforce fine-grained access policies. Access to secrets is controlled through authentication methods, which can include username/password, tokens, certificates, or integration with external identity providers.
- **Dynamic Secrets:** Vault can generate dynamic secrets on-demand for various systems and services. These dynamic secrets have short-lived lifetimes, reducing the risk of compromised credentials. Examples include generating dynamic database credentials, cloud service tokens, or SSH keys.
- **Secret Rotation:** Vault provides mechanisms for securely rotating secrets, such as automatically generating new credentials and securely distributing them to applications. This helps mitigate the risks associated with long-lived or static credentials.
- **Auditing and Monitoring:** Vault logs and audits all access and operations, providing visibility into who accessed which secrets and when. This audit trail helps meet compliance requirements and facilitates troubleshooting and forensic analysis.
- **Integration and Extensibility:** Vault integrates with various authentication providers, including Active Directory, LDAP, AWS IAM, and GitHub. It can also integrate with external systems for key management and secrets retrieval. Additionally, Vault offers an API and a plugin architecture that allows extending its capabilities and integrating with other tools and workflows.
- **High Availability and Scalability:** Vault supports high availability deployments to ensure the availability of secrets even in the event of failures. It can be deployed in a clustered configuration for scalability and fault tolerance, enabling the handling of large-scale deployments and demanding workloads.

Vault is widely used in cloud-native and DevOps environments, where securing and managing secrets is crucial. It helps organizations improve security, simplify secrets management, and adhere to security best practices by providing a secure and scalable solution for secret storage and access control.

The Structure and Workign of HashiCorp's Vault

HashiCorp's Vault is a highly flexible and secure secrets management tool that helps organizations store, manage, and distribute sensitive data, such as passwords, API keys, encryption keys, and certificates. Vault operates based on a client-server architecture, where the Vault server manages the secure storage and retrieval of secrets, while clients interact with the server to access and manage those secrets.

Here's an overview of the structure and working of HashiCorp's Vault:

- **Vault Server:** The Vault server is the core component responsible for storing and managing secrets. It securely stores secrets in an encrypted format and provides access control mechanisms to ensure

that only authorized users or systems can access the stored secrets. The server can be deployed in a high-availability configuration to ensure reliability and fault tolerance.

- **Authentication and Authorization:** Vault supports various authentication methods to verify the identity of clients. Users or systems must authenticate themselves before accessing the stored secrets. Vault integrates with external authentication providers like LDAP, Active Directory, and cloud identity services. After authentication, Vault enforces access control policies to determine the level of authorization for each client.
- **Secrets Engine:** Vault incorporates different secrets engines to handle various types of secrets. Secrets engines are modules responsible for generating, storing, and managing secrets. Examples of secrets engines include the Key/Value secrets engine for storing key-value pairs, Database secrets engine for dynamically generating database credentials, and PKI secrets engine for managing certificates and private keys.
- **Encryption and Data Protection:** Vault employs encryption to protect secrets at rest and in transit. The secrets are encrypted using strong cryptographic algorithms, and Vault manages the encryption keys securely. This ensures that even if the underlying storage or network is compromised, the secrets remain protected.
- **Dynamic Secrets:** Vault introduces the concept of dynamic secrets, which are secrets generated on-demand for specific systems or applications. Dynamic secrets have a limited lifespan, reducing the risk of compromised credentials. For example, Vault can generate dynamic database credentials that expire after a specified period, enhancing security.
- **Secret Lifecycle Management:** Vault provides mechanisms for securely rotating and revoking secrets. It supports secret rotation, where new secrets are generated and distributed while the old ones are revoked. This helps mitigate the risks associated with long-lived or compromised secrets. Vault also tracks secret metadata and enables auditing of secret access and usage.
- **Auditing and Logging:** Vault logs and audits all access and operations performed by clients. It maintains an audit trail of actions, including who accessed which secrets and when. This audit trail is crucial for compliance purposes, as well as for monitoring and investigating security incidents.
- **Integration and Extensibility:** Vault offers an extensive API and a plugin architecture that enables integration with other tools, systems, and workflows. It supports integration with external key management systems, cloud providers, identity providers, and orchestration platforms. This extensibility allows Vault to be seamlessly integrated into existing environments and workflows.

HashiCorp's Vault provides a secure and scalable solution for managing secrets, ensuring that sensitive data remains protected throughout its lifecycle. Its flexible architecture, support for multiple authentication methods, secrets engines, and auditing capabilities make it a valuable tool for organizations seeking robust secrets management and data protection.

How does HashiCorp's Vault help me in becoming my own Certifying Authority?

Becoming your own Certifying Authority (CA) involves setting up your own infrastructure to issue and manage digital certificates. While HashiCorp's Vault is not specifically designed to function as a CA, it can play a role in securely managing the private keys and secrets associated with operating a CA.

While Vault is not specifically designed to function as a CA, it can assist in securely managing the private keys and secrets associated with operating a CA. Here's how Vault can contribute to the CA process:

1. **Secret Management:** Vault can securely store and manage the private keys associated with your CA, ensuring they are protected and accessible only to authorized users or systems.
2. **Key Protection:** Vault provides cryptographic key management features, including secure key storage, access controls, and key rotation mechanisms. These features help protect the private keys used for signing certificates and ensure their confidentiality and integrity.
3. **Secrets Encryption:** Vault can encrypt and store sensitive data, such as CA root certificates, intermediate certificates, and other secrets associated with the CA infrastructure. This ensures that the sensitive information remains secure and confidential.
4. **Access Control:** Vault offers robust access control mechanisms to manage who can access and manage the CA's secrets. It supports various authentication methods and fine-grained access policies, enabling you to control and restrict access to the CA's secrets based on user roles and permissions.
5. **Auditability and Compliance:** Vault provides auditing and logging capabilities, allowing you to track and monitor access to the CA secrets. This audit trail helps meet compliance requirements and facilitates security monitoring and forensic analysis.

While Vault does not provide the core functionality of a CA, it can serve as a secure and centralized secrets management platform to safeguard the critical components of a CA infrastructure, such as private keys, certificates, and other sensitive data.

1.3 NGINX and its services

What is NGINX?

NGINX (pronounced "engine X") is a popular open-source web server and reverse proxy server known for its high performance, scalability, and robustness. Originally created to solve the C10k problem (handling a large number of concurrent connections), NGINX has gained significant popularity and is widely used as a web server, load balancer, reverse proxy, and caching server.

Here are some key features and functionalities of NGINX:

- **Web Server:** NGINX can serve static and dynamic web content, handling HTTP and HTTPS requests. It efficiently processes incoming requests and serves responses to clients, making it suitable for hosting websites and web applications.
- **Reverse Proxy:** NGINX acts as an intermediary between clients and backend servers, forwarding client requests to the appropriate servers and relaying responses back to the clients. It helps distribute incoming traffic across multiple backend servers, improving performance, scalability, and availability.
- **Load Balancer:** NGINX can balance the load across multiple backend servers by intelligently distributing incoming requests based on various algorithms such as round-robin, least connections, IP hash, and more. This helps distribute the workload evenly and improves the overall performance and resilience of the system.
- **Caching:** NGINX can cache static and dynamic content, reducing the load on backend servers and improving response times for subsequent requests. It can cache responses based on configurable

rules, such as URL patterns or HTTP headers, effectively serving cached content to clients, eliminating the need for repeated processing.

- **SSL/TLS Termination:** NGINX can handle SSL/TLS encryption and decryption, acting as a termination point for secure connections. It offloads the computational overhead of SSL/TLS encryption from backend servers, improving their performance and simplifying the management of SSL/TLS certificates.
- **Security and Access Control:** NGINX provides various security features and options to protect web applications, including access control based on IP addresses, rate limiting to mitigate DoS attacks, and request filtering to block malicious requests. It offers granular control over request handling and can be integrated with additional security tools and modules.
- **High Performance and Scalability:** NGINX is known for its high-performance architecture, optimized for handling a large number of concurrent connections and delivering content quickly. It efficiently utilizes system resources, making it suitable for high-traffic websites and applications.
- **Flexible Configuration:** NGINX uses a declarative configuration syntax that is highly flexible and allows customization of server behavior. It supports advanced configuration options, including rewrite rules, header manipulation, URL redirection, and more, providing fine-grained control over how requests are handled.

NGINX is widely adopted by organizations of all sizes, from small businesses to large enterprises, as well as by popular websites and web applications. Its combination of performance, scalability, flexibility, and extensive feature set makes it a powerful tool for serving web content, load balancing, and enhancing the overall performance and security of web applications.

How will NGINX be of use to us?

We will utilize NGINX as the web server to host our website. NGINX is a high-performance web server known for its efficiency, scalability, and robustness. By leveraging NGINX, we can ensure that our website is delivered to users with optimal speed and reliability. With NGINX as our web server, we can handle a large number of concurrent connections and efficiently serve static and dynamic content. This means that our website will be able to handle high volumes of traffic without compromising performance.

NGINX supports SSL/TLS encryption, allowing us to secure our website with HTTPS. It can handle the SSL/TLS termination process, relieving our backend server from the computational overhead. This ensures that the communication between our website and users is encrypted and secure.

NGINX's flexibility and extensive configuration options enable us to set up URL rewriting and redirection, improving the user experience and search engine optimization of our website. We can easily define rules to redirect or rewrite URLs, ensuring that users can access our content through clean and user-friendly URLs.

Section 2: Utilising HashiCorp's Vault services to become your own Certifying Authority

2.1 Installing and Initializing HashiCorp's Vault

For our purpose, we will be using a Linux based Operating System; Ubuntu in our case. Kindly follow the below steps to install and initialize your Vault.

Step 1: Installing HashiCorp's Vault

Open a new terminal and execute the following command to install Vault.

```
~$ sudo apt-get install vault
```

This command will install Vault in your Ubuntu system.

Upon successful installation, you will be able to view Vault's 'help' menu by executing the following command.

```
~$ vault
```

Step 2: Initializing the Vault

We'll start by initializing the server. Run the below command to start the server in 'dev' mode. The 'dev' mode simply a pre-unsealed vault that has already been configure to meet the basic requirement.

```
~$ vault server -dev
```

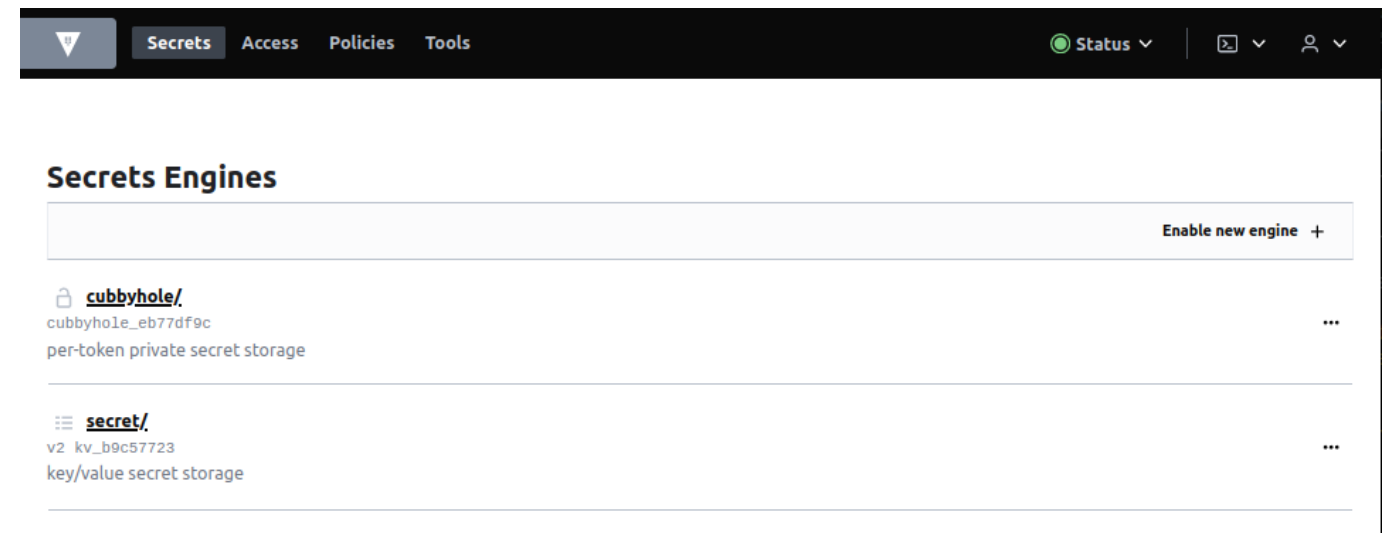
Kindly note that you must never start the Vault in 'dev' mode during production. This mode must only be used for demonstration and testing purposes.

Now that we've initialized the Vault, we'll be presented with the localhost address the server has been hosted in and the token required to access the Vault server.

Keep in mind that this token must not be shared with others. As this wil act as a 'password' to access our Vault, anyone with this token will be able to access your Vault, which may void the purpose of a vault.

In out case, our vault server has been hosted in the address <http://127.0.0.1:8200>. Surfing your browser with address will lead you to the landing page of your Vault, we're you'll be queried with your token.

With this, you will have successfully initiated your Vault and logged in and the homepage you're presented with would look something like this.



2.2 Setting up the root Certifying Authority

The root Certifying Authority (CA), also known as the root certificate authority, is the highest level of authority in a public key infrastructure (PKI) system. It is the entity that issues and signs the root certificate, which is the foundation of trust for all other certificates within the PKI.

The root CA is responsible for establishing trust by digitally signing and issuing certificates for intermediate CAs and end-entity certificates. It acts as the ultimate source of trust in the certificate chain. Root certificates are typically self-signed, meaning they are signed with their own private key, and their public key is distributed and trusted by operating systems, web browsers, and other certificate validation mechanisms. The inclusion of a root certificate in a trusted store is what allows the certificates issued by that root CA to be trusted. Root certificates are pre-installed in operating systems and web browsers and are used to verify the authenticity and integrity of certificates presented by websites, services, or entities in the PKI ecosystem.

It's important to note that the private key of a root CA should be securely stored and protected since compromising the root CA's private key would undermine the entire PKI system's security and trust.

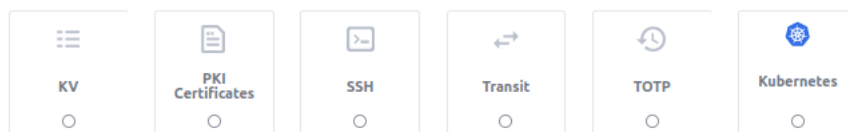
Root CAs are typically operated by trusted organizations, such as commercial CAs, government entities, or large enterprises. The trustworthiness of a root CA is established through a thorough validation process, adherence to industry standards and best practices, and regular audits or assessments to ensure the integrity of their operations.

Step 1: Enabling PKI Certificates in a directory

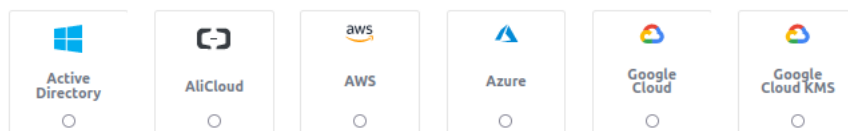
1. Under the *Secrets* tab, start by navigating into the *Enable new engine* option. You will find yourself in this menu:

Enable a Secrets Engine

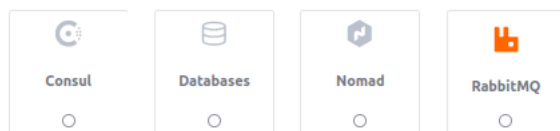
Generic



Cloud



Infra



Next

In this menu, click on 'Next' after selecting **PKI Certificates** under the 'Generic' section.

2. Now you'll have to select a path in which you'd like to access your root Certificate Authority from. By default it is set to 'pki'. However, we may change it according to our convenience. In our case, we'll be going with 'Root' as our path.

Enable a Secrets Engine

Path

pki

Method Options

Enable Engine

Back

Under *Method Options* enable **Max Lease TTL**. Now you may set the maximum duration your root may issue a certificate with. In our case, we'll go with **87600 hours** (10 years).

☐ Seal wrap ⓘ

☐ Default Lease TTL

Vault will use the default lease duration.

☒ Max Lease TTL

Lease will expire after

87600

hours

Ater choosing your maximum TTL, click on **Enable Engine** to enable your PKI engine.

Step 2: Configuring your root CA

1. Within this directory you just created, under the *Configuration* tab, click on **Configure**. Then, proceed by clicking on **Configure CA**. Here, make sure the 'CA Type' is set as *root*. Now, enter a common name. This will be your domain. For our purpose, we'll use **domain.com** as our domain. Upon deployment, it is extremely recommended to enter your address as well. Upon following all the instruction, your page should look somewhat like this.

Configure PKI

View backend >

CA certificate URLs CRL Tidy

Configure CA Certificate

CA Type

root

☐ Upload PEM bundle

Type

internal

Common name

domain.com

Options

Address Options

Save Cancel

Exit by clicking on **Save**.

2. What has been generated is your certificate. You may save this certificate in **.crt** or **.pem** format. We'll be saving it as **rootCA.pem**. Copy the certificate and save it with a name of your choicewith one of the given extensions.
3. After saving it, switch over to the *URLs* tab. Paste the following under the respective fields.

- Issuing certificates:

```
<IP_ADDRESS_VAULT_IS_HOSTED_IN>:
<PORT_NUMBER>/v1/<DIRECTORY_OF_ROOT_CA>/ca
```

In our case, it'll be **http://127.0.0.1:8200/v1/Root/ca**

- CRL Distribution Points:

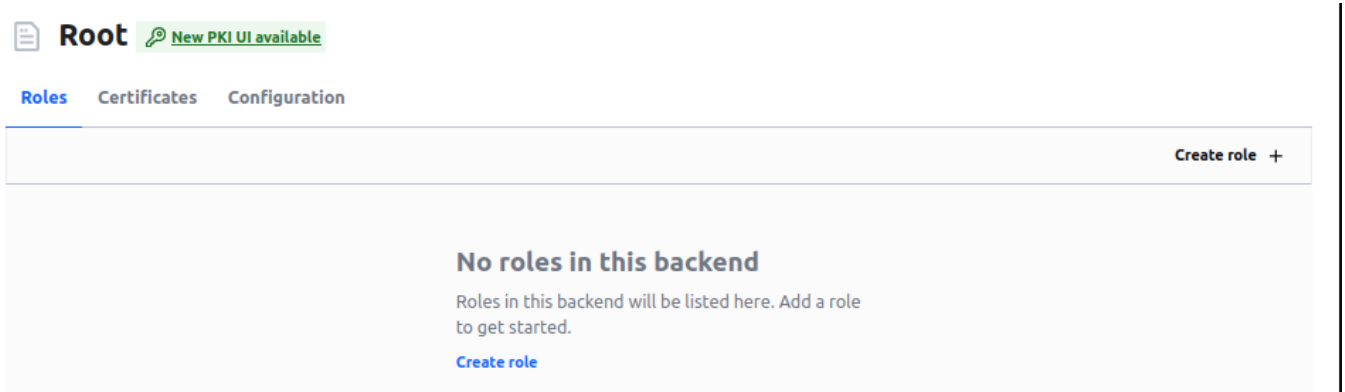
```
<IP_ADDRESS_VAULT_IS_HOSTED_IN>:
<PORT_NUMBER>/v1/<DIRECTORY_OF_ROOT_CA>/crl
```

In our case, it'll be `http://127.0.0.1:8200/v1/Root/crl`

Save it before proceeding to the next step.

2.3 Issuing a certificate for a sub-domain directly from the root CA (Not recommended)

From the *Secrets* tab, go to the directory you have your root CA configured in.



Under the *Roles* tab, click on **Create role** to create a new role.

1. Now, assign a name to it. We'll go with `rootRole`.
2. Under the **Hide Domain Handling** drop down, enable **Allow subdomains** and add your domain (`domain.com` in our case) in the text box provided under the title *Allowed domains*.
3. Now at the top write of the webpage, there will be an option to open a console. Click on this option and execute the command below.

```
read -field=default <DIRECTORY_OF_ROOT>/config/issuers
```

In our case, it'll be `read -field=default Root/config/issuers`. Copy the ID you receive and paste it under the *Issuer ref* field.

4. Under the *Show options*, you'll find an option to set the Max TTL. This is nothing but the maximum duration of certificate the role can issue. Note that this will take input in seconds; which means you'll have to convert your duration to seconds before setting up the value. Leave it blank will allow it to take the default duration as the Max TTL, which is the Max TTL you set up for the corresponding authority the role is created in.
5. Proceed by clicking on *Create role*.

From the *Roles* tab, click on the role you just created. In this menu, you'll be able to issue certificates for your subdomains.

1. Enter the subdomain you would like to issue the certificate for. We'll be going with `root.domain.com`.
2. Select `pem_bundle` as the format.
3. Under *Options* enter the duration of the certificate i.e. the lease period. When the period you entered exceeds the max duration that can be allotted by the role, you'll be issued with that maximum duration as the lease period of your certificate. We'll set our period as `720 hours` (30 days).

Here's what your panel would look like.

Issue Certificate

The screenshot shows a web form titled "Issue Certificate". It contains three main sections: "Common name" with a text input field containing "root.domain.com"; "Format" with a dropdown menu currently showing "pem"; and "Options" which is expanded to show a text input field for duration, containing "720 hours". At the bottom of the form are two buttons: "Generate" (in blue) and "Cancel" (in grey).

Proceed by generating your certificate.

What we see here are the details of the certificate you just issued for your subdomain. Click on `Copy credentials` and save it in any readable format. You'll be copying all this details in the form of a `.json` file so whichever format you choose, as long as you can read it, you may use it. We'll not be using this anywhere as an entire file. It only holds the credentials that you may need later.

Now what interests us are the following fields:

- **Certificate:** This is your public key. You'll be needing this later on in this guide. Copy this key and save it as a `.crt` file. We'll be saving it as `rootPublic.crt`.

While saving this file, if you find yourself having more than 1 section, the second section is what the public key is. You may remove the rest.

- **Private key:** This is the private key you'll be using later on as a pair with your public key. Copy this key and save it as a `.key` file. We'll be saving it as `rootPrivate.key`.
- **CA chain:** You will need this for uploading it in your browser later on. This is the certificate your browser will use to ensure that your domain is secure. Save it as a `.pem` file. We'll be saving it as `rootAuthority.pem`.

Make sure to copy everything including the `'-----CERTIFICATE-----'` part.

2.4 Issuing a certificate for a sub-domain through an intermediate CA

An intermediate Certifying Authority (CA) is a subordinate entity within a public key infrastructure (PKI) system. It sits between the root CA and end-entity CAs, forming a hierarchical chain of trust. The purpose of an intermediate CA is to enhance the scalability and security of the PKI by delegating the responsibility of certificate issuance and management from the root CA to intermediate CAs. The root CA issues and signs the intermediate CA's certificate, establishing trust in the intermediate CA.

Intermediate CAs operate under the authority and policies set by the root CA. They can issue their own certificates, sign certificates for end entities (such as websites or individuals), and manage the lifecycle of these certificates. Intermediate CAs have their own private key and use it to sign the certificates they issue, while their own certificate is signed by the root CA.

The use of intermediate CAs provides several advantages:

- **Scalability:** By delegating certificate issuance to intermediate CAs, the root CA's workload is reduced, allowing for more efficient management of the PKI system. This scalability is especially important in large-scale PKI deployments with numerous certificates being issued.
- **Flexibility:** Intermediate CAs allow for more granular control and customization of certificate policies and management. Different intermediate CAs can be created to serve specific purposes or organizational units within an enterprise, each with its own set of policies and controls.
- **Compartmentalization of Risk:** With intermediate CAs, the impact of a compromise or security breach is contained within the specific domain of that intermediate CA. If an intermediate CA's private key is compromised, it does not directly affect the trustworthiness of the root CA or other intermediate CAs in the hierarchy.
- **Certificate Hierarchy:** The use of intermediate CAs creates a chain of trust that extends from the root CA to end-entity certificates. This hierarchy provides a clear and verifiable path of trust, enabling validation and verification of certificates through the certification path.







When validating a certificate, the trustworthiness of an intermediate CA is established by verifying its certificate against the root CA's certificate. The trustworthiness of the root CA is assumed, as its certificate is typically pre-installed and trusted in the client's trust store.

Step 1: Enabling PKI Certificates in a directory







1. Under the *Secrets* tab, start by navigating into the *Enable new engine* option. You will find yourself in this menu:

Enable a Secrets Engine





Generic

 KV <input type="radio"/>	 PKI Certificates <input type="radio"/>	 SSH <input type="radio"/>	 Transit <input type="radio"/>	 TOTP <input type="radio"/>	 Kubernetes <input type="radio"/>
--	--	---	---	--	---

Cloud

 Active Directory <input type="radio"/>	 AliCloud <input type="radio"/>	 AWS <input type="radio"/>	 Azure <input type="radio"/>	 Google Cloud <input type="radio"/>	 Google Cloud KMS <input type="radio"/>
--	--	---	---	--	---

Infra

 Consul <input type="radio"/>	 Databases <input type="radio"/>	 Nomad <input type="radio"/>	 RabbitMQ <input type="radio"/>
--	---	---	--

Next

In this menu, click on 'Next' after selecting **PKI Certificates** under the 'Generic' section.

- Now you'll have to select a path in which you'd like to access your intermediate Certificate Authority from. By default it is set to 'pki'. However, we may change it according to our convenience. In our case, we'll be going with 'Intermediate' as our path.

Enable a Secrets Engine

Path

Intermediate

Method Options

Enable Engine

Back

Under *Method Options* enable **Max Lease TTL**. Now you may set the maximum duration your root may issue a certificate with. In our case, we'll go with **48300 hours** (5 years).

☐ Seal wrap ⓘ

☐ Default Lease TTL

Vault will use the default lease duration.

☒ Max Lease TTL

Lease will expire after

48300

hours



After choosing your maximum TTL, click on **Enable Engine** to enable your PKI engine.

Step 2: Configuring your intermediate CA

1. Within this directory you just created, under the *Configuration* tab, click on **Configure**. Then, proceed by clicking on **Configure CA**. Here, make sure the 'CA Type' is set as *intermediate*. Now, enter a common name. This will be your domain. For our purpose, we'll use **domain.com Intermediate Certificate Authority** as our domain. Upon deployment, it is extremely recommended to enter your address as well. Upon following all the instruction, your page should look somewhat like this.

Configure PKI

View backend >

CA certificate URLs CRL Tidy

Configure CA Certificate

CA Type

intermediate

☐ Upload PEM bundle

Type

internal

Common name

domain.com Intermediate Certificate Authority

Options

Address Options

Save Cancel

Exit by clicking on **Save**.

2. Copy what has been generated, and save it temporarily in a location of your choice in any readable format. This is the signing request that has been generated by your intermediate CA. You'll be getting this signed from the root CA you created earlier.

Configure PKI

View backend >

CA certificate URLs CRL Tidy

Configure CA Certificate

CSR

Common name

domain.com Intermediate Certificate Authority

Copy CSR Back

Now, to get it signed, follow these steps:

- Head back to the root directory and navigate to the **Configure** panel.
- Click on **Sign intermediate**.
- Now where it queries for the CSR, paste what you just copied.
- Under **Common name** type your domain name (**domain.com** in our case).

Sign intermediate

Certificate Signing Request (CSR)

```
SVTND8hcCLFDENBCIL07JYBFDIILXmCslh+f3aFFNPXTakwE90xTBq6vfkU1J/RS
Unl79LWb9vL0wJw+wjU53oKALT8bTXZ4DOI5AbhddK49MTEG+J4icYUhgXxZylV
ZKwM5jG/lw5GFAhrY7Py3ruhDCMNYOEuZZAIREdDmSqk9b/f95iiqyZit5x+VJHN
c80uafeaWw+7vHN8ghY6T3c+BX+22s0MZmlEr/V9gBmYSakiz0s3h6Ck/WwJNPUw
0vu3BZsuCK3S4vMwVZLGBm4=
-----END CERTIFICATE REQUEST-----
```

Common name

domain.com

Format

pem

☐ Use CSR values

▼ Options

▼ Address Options

Save

Cancel

Proceed by saving it.

What has been generated is the permission to be an intermediate certificate. Copy this as you'll be requiring it in the next step.

Now head back to the **Configure** panel in the *Intermediate* directory and click on **Set signed intermediate**. Here, you'll be pasting the permission you just copied. After this, you'll be presented with the Certificate Authority chain to act as an intermediate authority. Copy this certificate and save it with a **.pem** extension. We'll be saving it as **intermediateAuthority.pem**. We'll require this certificate later as we will be uploading it in our browser.

3. After saving it, switch over to the **URLs** tab. Paste the following under the respective fields.

- Issuing certificates:

<IP_ADDRESS_VAULT_IS_HOSTED_IN>:

<PORT_NUMBER>/v1/<DIRECTORY_OF_INTERMEDIATE_CA>/ca

In our case, it'll be **http://127.0.0.1:8200/v1/Intermediate/ca**

- CRL Distribution Points:

<IP_ADDRESS_VAULT_IS_HOSTED_IN>:

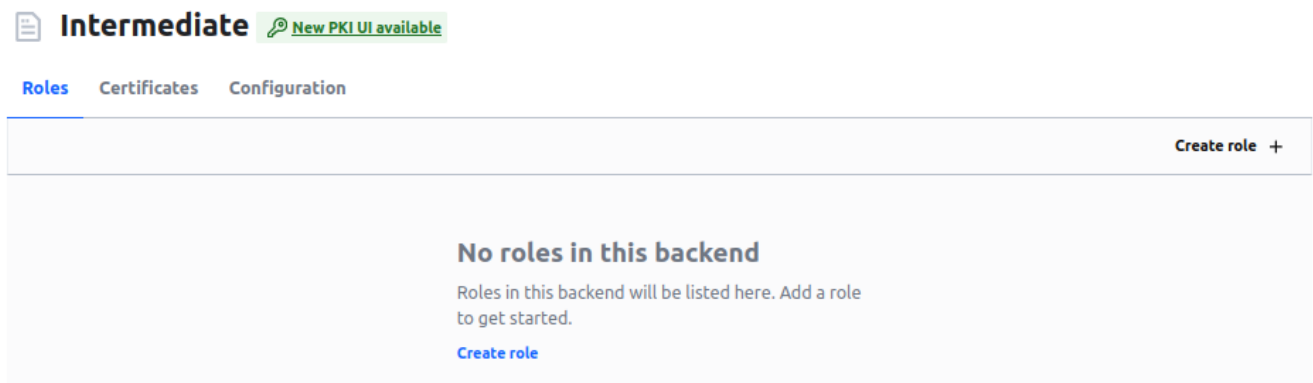
<PORT_NUMBER>/v1/<DIRECTORY_OF_INTERMEDIATE_CA>/crl

In our case, it'll be **http://127.0.0.1:8200/v1/Intermediate/crl**

Save it before proceeding to the next step.

Step 3: Creating a role that is capable of issuing your sub-domain with a valid certificate and issuing a certificate for your sub-domain

From the *Secrets* tab, go to the directory you have your intermediate CA configured in.



Under the *Roles* tab, click on **Create role** to create a new role.

1. Now, assign a name to it. We'll go with `intermediateRole`.
2. Under the **Hide Domain Handling** drop down, enable **Allow subdomains** and add your domain (`domain.com` in our case) in the text box provided under the title *Allowed domains*.
3. Now at the top write of the webpage, there will be an option to open a console. Click on this option and execute the command below.

```
read -field=default <DIRECTORY_OF_INTERMEDIATE>/config/issuers
```

In our case, it'll be `read -field=default Intermediate/config/issuers`. Copy the ID you receive and paste it under the *Issuer ref* field.

4. Under the *Show options*, you'll find an option to set the Max TTL. This is nothing but the maximum duration of certificate the role can issue. Note that this will take input in seconds; which means you'll have to convert your duration to seconds before setting up the value. Leave it blank will allow it to take the default duration as the Max TTL, which is the Max TTL you set up for the corresponding authority the role is created in.
5. Proceed by clicking on *Create role*.

From the *Roles* tab, click on the role you just created. In this menu, you'll be able to issue certificates for your subdomains.

1. Enter the subdomain you would like to issue the certificate for. We'll be going with `intermediate.domain.com`.
2. Select `pem` as the format.

- Under *Options* enter the duration of the certificate i.e. the lease period. When the period you entered exceed the max duration that can be allotted by the role, you'll be issued with that maximum duration as the lease period of your certificate. We'll set our period as **720 hours** (30 days).

Here's what your panel would look like.

Issue Certificate

Common name

Format

Options

Generate

Cancel

Proceed by generating your certificate.

What we see here are the details of the certificate you just issued for your subdomain. Click on **Copy credentials** and save it in any readable format. You'll be copying all this details in the form of a **.json** file so whichever format you choose, as long as you can read it, you may use it. We'll not be using this anywhere as an entire file. It only holds the credentials that you may need later.

Now what interests us are the following fields:

- Certificate:** This is your public key. You'll be needing this later on in this guide. Copy this key and save it as a **.crt** file. We'll be saving it as **intermediatePublic.crt**.

While saving this file, if you find yourself having more than 1 section, the second section is what the public key is. You may remove the rest.

- Private key:** This is the private key you'll be using later on as a pair with your public key. Copy this key and save it as a **.key** file. We'll be saving it as **intermediatePrivate.key**.
- CA chain:** You will need this for uploading it in your browser later on. This is the certificate your browser will use to ensure that your domain is secure. Save it as a **.pem** file. We'll be saving it as **intermediateAuthority.pem**.

Make sure to copy everything including the '-----CERTIFICATE-----' part.

Section 3: Setting up a server to host your website using NGINX and getting it certified

3.1 Installing and Initializing NGINX

Step 1: Installing NGINX

Open a new terminal and execute the following command to install Vault.

```
~$ sudo apt-get install nginx
```

This command will install NGINX in your Ubuntu system.

Upon successful installation, you will be able to view NGINX's 'welcome' menu by executing the following command.

```
~$ nginx
```

Step 2: Initializing NGINX

Follow these steps to get your webpage up and running using NGINX. Make sure to have prior knowledge in handling files and navigation within your OS. As we are working on a Linux based OS, The steps or commands may vary if you're using an OS other than Ubuntu. Let's now get started with the steps:

- Open a new terminal and change your directory to the root by simply executing this command in your terminal.

```
~$ cd
```

- Now, you'll require root access as you'll be working with files within the hidden directory and requesting for the `sudo` access in each step would be troublesome. Now, to get the root access once an for all so that you don't have to request for it in every necessary command, simply execute this command and enter your password.

```
~$ sudo su
```

NOTE: If you've decided to skip this step and were to encounter a permission denial problem further in the guide, re-execute the command by using `sudo` as a suffix for the command.

- Navigate to the directory where we will be storing our `.html` file. You may achiev this by using the following command.

```
~$ cd /var/www
```

- Here, we'll create a new directory. In this guide, we'll call this directory by the name `Domain`. You can create a directory using the below command.

```
~$ mkdir Domain
```

Now that you've created your directory, navigate into it using the `cd` command.

```
~$ cd Domain
```

Notethat this step is completely optional and only serves the purpose of making the directory look a bit clean. you mayfeel free to skip this step completely.

- Here, we'll create our `.html` file. Let's call our's `domain.html`. Create the file using the the below command.

```
~$ vi domain.html
```

This is the Vim environment and is notorious for its inconvenient CLI. You you can relate to this, try using any other editor such as `nano`. In this guide, due to the reasons stated above, we'll be using `nano` from here.

- Now we'll be navigating into one of NGINX's directory where the sites that are available to be used are stored. For this, we mayse this command.

```
~$ cd /etc/nginx/sites-available
```

If you were to list the contents of this directory, you'll find that there is already a file named `default`. For our own website, we'll be requiring the same configuration and will be copying this file into its current direct with a different name, and making our changes within this new file. Let's call our file `Domain`. Execute the below command.

```
~$ cp default /etc/nginx/sites-available/Domain
```

- Now, edit the contents of the file and make the following changes.
 - Under the first server block, change the port number from `80` to `443` or any other of you choice. Also, make sure to remove `default_server` rght next to those port numbers. Make sure that you change the port number for both the lines.
 - Futher down the file, you'll ind a statement that mentions the root of the file. By default it is set to `/var/www/html`. But since we've store all our `.html` files in a directory named `Domain`, we'll be changing this location to that. So now, the new location teh root will hold is `/var/www/Domain`.
 - Listed in the next line are the names of the `.html` file. Replace `index.html` with your file's name. `domain.html` in our case. Do the same for other files as well.
 - Exit by saving the file.
- We'll now navigate to a different directory. Use this command to go to the directory where NGINX's enabled sites are stored.

```
~$ cd /etc/nginx/sites-enabled
```

- Here, we'll be generating a link between the `Domain` file we just configured and teh `Domain` file we will eventually be storing in this directory as well. This way, any changes you make in any one of the files will be reflected in the other as well. Use the folowing command for this purpose.

```
~$ ln -s /etc/nginx/sites-available/Domain /etc/nginx/sites-enabled/Domain
```

- We'll now have to map the port number to the domain name. For this, we'll be switching into the directory where all the hosts are included, and add our domain into it. `~$ nano /etc/hosts`

As mentioned earlier, we'll be using `nano` instead of `Vim`.

- Add the local host, which is `127.0.0.1:443` and your domain name next to it. In our case, it is `domain.com`.
- You may confirm if NGINX has been configured properly by checking if it shows a *successful* upon executing this command.

```
~$ nginx -t
```

If it is not successful, recheck your configuration files and try again.

- Now you may start the NGINX service.

```
~$ service nginx start
```

You may check its status as well.

```
~$ service nginx status
```

- Open your browser and visit the port number in which you created your website in. In this guide, we'll be searching for `http://127.0.0.1:443`.

You should be able to view your website!

And since we mapped your domain name to the localhost, you'll be able to visit your website using your domain name as well.

NOTE: You haven't added the certificate for your domain yet. So make sure you search using `http://` instead of `https://`.

3.2 Securing your website using the Self-Signed Certificates

You've generated 2 different certificates. One that was issued directly by the root Authority for the sub-domain `root.domain.com` and the other was issued by the intermediate Authority with the sub-domain `intermediate.domain.com`. In this guide, for the sake of simplicity, we'll just be using the domain we generated using our root Authority.

Step 1: Importing the certificate into your browser

When recalled, while creating the certificate, we saved a certificate with the name `rootAuthority.pem`. We're required to import this certificate into our browser. In other words, it's the CA chain for the corresponding certificate we saved as a `.pem` file. Our browser will use this to check if the domain is trusted or not.

1. Open your browser settings and look for **Privacy** settings.
2. Look for an option that lets you view the certificates.
3. In here, you will come across a tab titled **Authority** to which you must switch to.
4. Upload the certificate (**rootAuthority.pem** in our case).
5. Select the option that gives permission for all the websites with this domain to be trusted and save it.

Step 2: Setting up the private and public certificate in NGINX

- Switch back to your terminal and use the below command to edit your configuration file.

```
~$ nano /etc/nginx/sites-available/Domain
```

It won't matter which directory (i.e. **sites-available** or **sites-enabled**) you make your changes in as the changes made in one will reflect in the other.

- In this file, next to your port number in the server block, type **ssl** after separating it with a space. Type it for both of them. This will let NGINX know that SSL has been enabled.
- Move a bit further, and in the line where it mentions **server_name**, replace the **_** with the sub-domain you would like to trust (the one you have issued the certificate for). In our case, it is **root.domain.com**.
- Make sure that the directory you have your private and public keys in doesn't have any spaces in them. If they do, move them to a more convenient place without spaces, or include *single quotes* while mentioning their directory in the step that follows.
- In the file you've been editing till now, at the end of the **server** block, add the following lines.

```
ssl_certificate <THE_DIRECTORY_OF_PUBLIC_KEY>/rootPublic.crt;  
ssl_certificate_key <THE_DIRECTORY_OF_PRIVATE_KEY>/rootPrivate.key;
```

Save the file and exit the editor.

- You will now have to add your new domain in your **hosts** file. You can edit this file by using this command.

```
~$ nano /etc/hosts
```

- Finally, restart NGINX with this command.

```
~$ service nginx restart
```

Restart your browser, and visit your sub-domain. You will notice that your browser now trusts your sub-domain. You can ensure this by viewing the **Secure** icon towards the left of your address bar. Here's what the address bar along with the *Secure* icon should look like.

