The technology partner for entrepeneurs
www.talpor.com

Jul 19

3 Comments

SSL/TLS certificates beginner's tutorial

This is a beginner's tutorial on SSL certificates (which by now should be called TLS certificates, but old habits die hard). I'll cover both how they function, and how to create a SSl/TLS certificate using OpenSSL, either self-signed or signed by a CA.

As a disclaimer, getting security wrong is very easy, and I'm not an expert. If your systems handle anything vital such as credit card information, payments, personal information, weapons systems, etc, then hire an expert, use already available services and just use the code provided to you by people that knows what they are doing. This is not gonna make you an expert on SSL/TLS, not even a novice, it will just make you able to undertand to a basic level what you're talking/hearing about. Things that are essential to TLS secure setup and that are ***NOT*** addressed here include:

- How to secure your keys
- How to provision them
- How to secure your servers
- What protocols/algorithms/ciphers/modes of operation should your server allow

- A whole lot more

Seriously, be mindful about how much the information/services that you administer are worth, and what could happen if they were compromised. Then consider carefully if you need to hire extra help or educate yourself more in the subject, security breaches can seriusly f@#* you up, your close ones, and people you dont even know.

That said, this tutorial is divided in 7 parts:

Introduction to SSL/TLS

SSL (Secure Socket Layer) is an old protocol deprecated in favor of **TLS (Transport Layer Security)**.

TLS is a protocol for secure transmission of data heavily based on SSLv3. It offers **confidentiality, integrity and authentication**. In layman’s terms that means that it:

- Confidentiality: hides the content of the messages
- Integrity : detects when they’ve been tampered with

- Authentication: ensures that whoever is sending them is who he says he is.

Additionally it detects missing and duplicated messages.

TLS is the primary way to secure web traffic, and is mostly used for that purpose. A whole lot of pages trust that TLS is secure (from the smallest online shop to Facebook), that is why things like POODLE and Heartbleed receive so much press.

At this point I want to make something abundantly clear, SSL IS BROKEN, ALL THREE VERSIONS. The latest one, version 3, had its confidentiality severely compromised by POODLE. DO NOT USE SSL.

Confidentiality and integrity in TLS

First I'll briefly explain how TLS offers confidentiality and integrity. We will leave authentication at the end as that will be the bulk of the tutorial.

I will be using a lot of crypto jargon here so do not fret if any of this seems too alien to you, generally you won't be fiddling with this directly as TLS negotiates much of this for you.

However I do recommend getting familiar with the basics of public key encryption. Not the inner workings mind you, just how they are used in practice to encrypt/authenticate.

If you are really interested on these subjects I'd recommend researching:

- AES
- Block ciphers modes of operation
- Cryptographic hashes
- SHA-2
- HMAC

- Key exchange
- Diffie-Hellman
- Public key cryptography

The Stanford course in Coursera about Cryptography is an awesome starting point.

For confidentiality TLS uses either Diffie-Hellman (elliptic curve mode supported) or RSA for the key exchange. With the whole Heartbleed debacle, it is now recommended using a key exchange mechanism with forward secrecy, which implies Diffie-Hellman in this case. The actual encryption is normally done using AES with several modes available (CBC, CCM, GCM).

For integrity normally HMAC is used, today is pretty much mandatory to use SHA256.

Basics of authentication in TLS

Authentication is the part that you will most likely have to fiddle with the most and the one that actually costs you money, it is also essential for the security of your communications: You. Cannot. Have. Secure. Communications. Without. Authentication.

How is that you might ask? Well the thing about confidentiality and integrity is that they are worthless without authentication. If there is no way to ensure that the guy that says "I am gmail.com I swear" is in fact gmail.com and not evil8yoldhacker.com, then you could potentially encrypt and validate a spurious connection.

So without authentication you would be open to, for example, MitM (Man in the Middle) attacks. But how can you authenticate a server if you have never seen it before, and let alone exchanged any credentials with it?

The answer is **TLS certificates**. Certificates are just a public key with a bunch of information attached to it, such as the FQDN being authenticated, a contact email, the “issued on” and “expires on" dates, among other things. The server stores and keeps secret the corresponding private key. TLS uses these keys to authenticate the server to the client.

Recall that in public key cryptography, messages encrypted with the public key can only be decrypted using the private key, but messages encrypted with the private key can be decrypted with either. The owner of the keys keeps the private key secret, and distributes the public key freely.

Now, the usual way to authenticating someone using public key cryptography is the following (Assume that Bob wants to authenticate Alice):

1. Bob sends Alice a random message encrypted with Alice's public key
2. Alice decrypts the message using her private key and sends it back to Bob
3. Bob compares the message from Alice against the one that it send. If they match Alice is who she say she is, because only her could have decrypted the message, because only she has her private key

TLS uses a variation of this technique, but is essentially the same.

Now, even with this trick validation of a certificate is not straightforward, so for didactic purposes we'll begin with an oversimplified, naive and flawed solution.

Solution **#1**:

1. A hash of the certificate is made, encrypted with the private key, and then appended to the certificate to create a new certificate
2. The server sends this new certificate to the clients that connect to it
3. To verify the certificate the client decrypts the hash using the public key of the certificate, then calculates its own hash and compares them, if they are equal the certificate is valid
4. It then sends a random message to the server encrypted with the provided public key, if the server sends the original unencrypted message back, then is considered authenticated

This process ensures that:

- The provided public key correspond to the private key used to encrypt the hash
- The server has access to the private key

The encrypted hash created appended to the certificate is called a **digital signature**. In this example, the server has digitally signed its own certificate, this is called a **self-signed certificate**.

Now, this scheme does not authenticates at all if you think about it. If an attacker manages to intercept the communications or divert the traffic, it can replace the public key on the certificate with his own, redo the digital signature with his own private key, and feed that to the client.

The problem lies in the fact that all the information necessary for verification is provided by the server, so the only thing you can be sure of is that the party that you’re talking to has the private key corresponding to the public key that it itself provided.

This is why when you connect to an entity with a self signed certificate browsers will give a warning, they cannot ensure that whoever you are communicating with is who they say they are.

However, these kind of certificates are really useful in some scenarios. They can be created free of charge, quickly, and with little hassle. Thus they are good for some internal communications and prototyping.

Solution **#2**:

So that did not work out. How can we solve it? Well, since the problem is that the server provides all the information for authenticating the certificate, why don’t we just move some of that information to the client?

Let us drive over to the client after lunch with a copy of the certificate in an USB drive, and store it directly on the client. Later when the server sends it’s certificate:

1. The client generates its own hash of the certificate and decrypts the provided hash with the public key of the *copy of the certificate that was provided earlier in the USB drive*, this

way it ensures that: 1. the certificate has not changed, 2. whoever signed it has the correct private key.

2. It then sends a random message to the server encrypted with the public key in the *copy of the certificate that was stored earlier*, if the server sends back the original unencrypted message we consider it authenticated

If someone intercepts or diverts the connection, they have no hope of passing our random number challenge and providing a valid certificate, without the private key that is computationally infeasible.

The “driving over to the client after lunch with an USB drive” is called an **out of band process**.

Now this solution actually authenticates, and is sometimes used for internal communications. However it is not very practical for several use cases. It would be cumbersome to have to download a certificate every time you need to access a new secure website like a bank or eshop, or worse, waiting for someone to drive over to you house after lunch with an USB drive.

Moreover, you have to ensure that the certificate is not tampered with on the way, which is one of the problems TLS should be solving for us in the first place. There is also the problem of what to do when the certificate expires, or how to revoke it if the private key becomes compromised.

Solution **#3**:

Well, we got something working, but it is not quite ready for use on the landing page of your online store yet. How can we solve it? Well, here is where money comes in.

Meet Bob, he is a well known member of the community, a truly and responsible fellow loyal to a fault, and he comes up with a business. He will create a self signed certificate, and will give it freely to everybody using an out of band process (let's say, during a neighbourhood-wide free-of-charge barbecue), he will then charge you a fee to digitally sign *your* certificate using *his* private key.

See, up until this time every certificate was self signed, this time it is Bob that will sign it, now we’re in a situation where:

- Everybody has the certificate that Bob so generously gave them
- Everybody just trusts Bob (who wouldn't, he saves orphan puppies from fires in his spare time)

Therefore, anyone that has his certificate signed with Bob’s private key can have it authenticated by anyone that has Bob’s certificate (because Bob’s certificate has his public key).

Bob will make sure nobody impersonates anyone. If he receives an email from you saying that you want certain certificate signed for your company, Bob will go to your house personally and ensure that it is your certificate, that it is your company, and that you did sent out that email...you can never be too careful with all these 8 year old evil hackers.

When Bob’s certificates is about to expire he will make sure he gives you a new one.

If anyone wants to revoke his or her certificate they can just tell Bob to put it in his list of revoked certificates, when somebody tries to authenticate you with a certificate they will call Bob to check if that certificate has been revoked.

Well, this is how the real world kind of works, with a few major changes:

- Bob is a **CA (Certification Authority)** such as DigiCert, Comodo, Symantec, GoDaddy, GlobalSign, etc (and no, the don’t save puppies, and sometimes f!@# up really bad)
- The out of band process is not a barbeque. Their certificates already come bundled with your operating system and browser (check /etc/ssl/certs if you’re on Linux)
- The revocation mechanisms are called CRL and OCSP (OCSP was supposed to supersede CRL, but the whole thing is kind of a mess right now)
- You kind of HAVE to trust them

They do charge you though. Depending on the kind of certificate the prices vary.

The process where a third party that both the server and the client trust signs the certificate of the server creates a **chain of trust**. With TLS, we create chains of trust using CAs as our third parties.

Note that your data will not have a stronger or weaker encryption depending on how much you pay, all TLS connections use some form of AES, how strong depends of what the client and server are able and willing to handle/use. For example, many servers refuse to use SSLv3 since the whole POODLE scandal, some older clients do not support the newest encryption algorithms, some servers have not been updated, some clients use Internet Explorer.

Now, you might wonder, if every certificate gives me equally strong encryption, integrity and authentication, why are some more costly than others?

Well, CAs generally charge you more for certificates that will be used on several machines, so for example a certificate for *.talpor.com will cost more than one for www.talpor.com (notice the shameless self-promotion). However, what CAs really sell is not certificates, is trust.

Domain validation vs Extended Validation

You might remember the part where "Bob will make sure nobody impersonates anyone". Well, depending on how hard the CA tries to do this, it will charge more or less. There are three tiers of certification. **Domain Validation, Organization Validation, Extended Validation**.

Domain Validation is the cheapest and most basic form of validation. The CA just tries to make sure that you have some kind of control over the domain that you are trying to get a certificate for. They take a few minutes to validate and setup.

A common way to do domain validation is as follows. Lets say that you want a certificate for alice.com. After providing all the necessary information to the CA to generate your certificate (more on that later), the CA will give you a list of emails in the domain associated with the certificate, say for example: admin@alice.com, sysadmin@alice.com, webmaster@alice.com. You are to pick one where a confirmation email will be sent, and here is the catch, *you have no say as to which emails appear in that list*, so you better have access to at least one of those.

Another common method involves probing the WHOIS database for the domain associated with the certificate.

Organization validation and Extended Validation are more expensive, EV is a simple upgrade over OV, so EV certificates are the most expensive ones.

These kinds of validations not only include the usual domain validation checks, they also make several checks to ensure that the company asking for the certificate do exist, physically and legally. These checks can range from phone calls to signed letters from the CEO. Because of this, these kinds of certificates can take several days to come out.

When faced with an EV certificate, browsers have some visual queue (green most of the time) that makes the address bar look more trustworthy, such as the green bar of Internet Explorer, or the big green square with the name of the company of Chrome.

To identify an EV certificate browsers use the CertificatePolicy extension field, each CA has a public list of values for this field that identify their EV certificates.

Step-by-step instructions on how to create certificates on Linux

Now armed with all this knowledge, you want to create your very own certificate for your puppies and cats hospital webpage, how you do need to go about it? Well, here are a few instructions for Linux systems.

Creating certificates to be signed by a CA

The steps are as follows:

1. Create a private key and a **Certificate Signing Request (CSR)**
2. Send the CSR to the CA, keep your private key secret at all times
3. Go through the validation process which will vary depending of the type of validation that you’re going for. You will receive a signed certificate at the end of the process

4. Place the certificate inside your server

5. Configure your application to use the new certificate. For Linux systems this usually means configuring Nginx/Apache2 to use the new certificate and key

6. Conquer the world/Go home for the day (optional)

We will just cover step 1, as the rest are outside the scope of this tutorial.

The CSR is just your public key bundled with a bunch of parameters such as the domain to be authenticated, the encryption algorithm of the key, a digital signature to prevent tampering, the company name, the country, city and state, etc. Sending a CSR to a CA is very much like asking a very specific autograph from them.

Notice that CAs may ignore many fields of the CSR (company name, country, etc) if your certificate will only be domain validated.

So, to **create a private key and its CSR** on Linux you can do:

```
openssl req -nodes -newkey rsa:2048 -keyout example.key -out exampl
```

- openssl: the commonly used program to do TLS related stuff in Linux

- req: openssl command to manage CSRs

- -nodes: set the passphrase of the private key to blank (private keys of TLS may have passphrases)

- -newkey rsa:2048: create a new key, make it RSA with 2048 bits (you may first create key and later feed it to this command as a file path)

- -keyout: where to write the new key

- -out: where to write the CSR

This command will prompt you for some information regarding the CSR, such as the domain (which is called common name), country, company etc. You can leave them all blank except for the common name.

If you want something more **script-friendly** you can use the -subj flag:

```
openssl req -nodes -newkey rsa:2048 -keyout example.key -out exampl
```

If you need to **create a CSR from an existing key** you can use:

```
openssl req -new -key example.key -out example.csr
```

To **create a private RSA key** you can do:

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

Creating self-signed SSL/TLS certificates

To sign your own certificate you can either use an existing CSR and private key, or just create one from scratch.

To **create a self-signed certificate from scratch** you can use this oneliner:

```
openssl req -nodes -newkey rsa:2048 -keyout example.key -out exampl
```

- -x509: write out a self signed certificate instead of a CSR (The technical name for TLS/SSL certificates is X.509 certificates)
- -days: how long will the certificate will be valid. For certificates made from April 1st 2015 onwards browsers do not accept certificates with a validity period spanning more than 39 months (3 years plus 3 months)

This command will prompt you for some information, if you're looking for something without interactivity look up the -subj flag.

To **create a self-signed certificate using an existing CSR and private key** you can do:

```
openssl x509 -req -in example.csr -signkey example.key -out example
```

- x509: openssl command to manage X.509 certificates (aka TLS/SSL certificates)
- -in: the CSR
- -signkey: the private key
- -out: where to write the certificate
- -days: see the previous command

Intermediate certificates

One thing that might baffle you at first is that CAs sometimes give you many certificates. There are practical reasons for this, but the bottom line is that one of those is your certificate, and another one is a root certificate that corresponds to a private key that signed one of the other certificates, which in turn corresponds to a private key that signed another one, and so on and so forth, forming a chain that eventually signs your certificate.

These intermediate certificates are called, well, **intermediate certificates** (Hurra for imagination). What you have to do is simply concatenate the certificates in the correct order, beginning with the certificate for your domain, and ending with the root certificate.

To figure out the correct order check first if your CA provides instructions, or if they gave you an already bundled version of the certificates.

If not you can check the "**Subject**" and "**Issuer**" fields of each certificate. The "Subject" field contains an identifier of the entity associated with the certificate, the “Issuer” contains the identifier of the entity that signed it.

Certificates come encoded in base64, but the following command will decode them:

```
openssl x509 -in example.crt -text -noout
```

Basic constraints

At this point, now that you know that certificates can be chained, you might have come up with a genius plan:

1. Get a very cheap certificate from a CA
2. Locally create a certificate that would be very expensive if it were signed by a CA
3. Use the private key of the cheap certificate to sign the expensive certificate that you created locally
4. Bring the greedy and evil CAs to their knees with your devious and genius stratagem

Well no, you cannot do that: certificates have a field that determines whether or not its private key can sign other certificates (Extensions -> Basic Constraints -> CA: FALSE), and you cannot change this field in the certificate that the CA gives your because it would ruin the signature (recall that the signature is a hash of the certificate contents encrypted with the *private key* of the CA).

Wrap up and useful links

That about does it. There are a few details and a plethora of information and subjects that are not covered here, but from a practical standpoint this encompasess the very basics.

As usual, I leave a list of useful links at the end. If you find an error, typo or another useful link, hop on in the comments and let me know.

- A case to push EV certificates as the defacto standard from DigiCert
- The TLS protocol is far more complex than what I described here
- This is a nice summarization of the certificate authentication process
- There are a lot of nice SSL/TLS tutorials out there
- There are also many step by step instructions out there

Posted on July 19, 2015, 11:26 a.m.

Topics : beginner self-signed tutorial ssl tls selfsigned CA rsa certificate

G+1 0 Tweet Like 43

3 Comments **talPor Solutions - Blog** 1 **Login**

Recommend 3 **Share** **Sort by Best**

Join the discussion…

Kieth Ahren • 2 months ago
Savvy blog post ! BTW , people want a a form , my colleague edited a template version here

> `http://goo.gl/AXwev3`

• Reply • Share ›

wstar • a year ago
THANK YOU. I have wanted this kind of primer regarding SSL and certificate management for some time. I really appreciate that you took the time to put this out there.

• Reply • Share ›

German Jaber wstar • a year ago
You're welcome. I'm glad it helped you.

I'm not sure when you read the article, but I recently added a disclaimer at the beginning warning people about doing security by themselves, you should check it out if it wasn't there when you read it, and if it was keep it on the back of your mind.

1 • Reply • Share ›

ALSO ON **TALPOR SOLUTIONS - BLOG**

Make a metric dashboard for Trello with Django Dashing | talPor
3 comments • 2 years ago•
Pravin Kumar — Hello, Is there a possibility to use dashing inside an existing app in a django project?

talPor Solutions Blog Random Links, Nov 20th -
1 comment • 3 years ago•
Alejandro Rojas — This list is a keeper

talPor Solutions Blog psycopg2 and Django 1.6 - LGDD Case Study -
1 comment • 3 years ago•
Alejandro Rojas — Great to see this blog...I will tell Levi about it to keep off those 1.5 bugs

Docker - Beginner's tutorial | talPor Solutions Blog
16 comments • 2 years ago•
Nan Xiao — "Containers are desing thus, to run an application, not a machine. ", "desing" should be