Uncertain Certificates

Anton R. Peters

V2.3 9-Jan-2018 (modified English version)

V1.0 15-Mar-2017 (Dutch version)

I. Introduction

One reason for writing this article is, that in my current position I was assigned to work for a major project to replace SHA-1 certificates. This became topical when the Center for Mathematics and Informatics (Amsterdam) together with Google, on 23 February 2017 after 2 years of computing, published the first SHA-1 "collision" [1] - see section “XII. SHA-1” for what this means.

However, in any project I get to deal with security of communication between systems. I find that misconceptions about how the techniques need to be used are widespread. Below I explain some pitfalls that I have encountered in practice, and how to avoid them.

II. PKI

The topic is Public Key Infrastructure (PKI), with private keys and public certificates, which allows two parties - traditionally called Alice and Bob - to encrypt their communications. For this purpose, standard protocols are designed: Transport Layer Security (TLS), formerly known as Secure Socket Layer (SSL), that we use daily in HTTPS connections of our browser (the little lock in the address bar). Not everyone will have to configure PKI daily, so how again does it work?

Secure communication involves a number of different things:

1) **Confidentiality**: can anyone else - spy Zorro – listen in?

2) Data **Integrity**: are there no bits changed *en route* accidently or deliberately? For this, you do not need encryption, it can also be achieved by using a parity bit or Cyclic Redundancy Check like in computer memory; or a hash, such as the control number in your IBAN.

3) **Authentication**: do you know with whom you are talking?

4) **Authorization**: what is the partner you have contact with allowed to do with your data and system, and *vice versa*?

5) **Trustworthiness**: can you trust your communication partner?

The first three goals can be achieved by encrypting your communication. For this, you need a secret key that only Alice and Bob know. Then the problem shifts to this: how can Alice and Bob safely exchange their key? Not over a communication channel: if Zorro is listening, then he can steal the key and decipher their later communication. And if that communication channel is safe enough to exchange a key, then they do not need encryption.

Public Key Infrastructure is a brilliant solution to these issues. It uses 2 keys: with the public one, Alice can encrypt a message. However, the message can not be decrypted with that key. That is only possible with the second key, which Bob keeps secret: he is the only one who can read the encrypted messages.

That such a thing is possible is a mathematical miracle (several, actually). The first and most widely used method was published in 1977 by Rivest, Shamir, and Adleman (RSA). For this purpose, they use the product of 2 large prime numbers, from which the public and the secret key are calculated. See Wikipedia [2]. The code is hard to crack because it is difficult to factor a large number into primes. Probably, every code may be cracked if the spy has enough encrypted data as example;and enough time, computer power, and smart people.

In practice, it's too cumbersome to encrypt all messages with this algorithm. In the TLS protocol [3] when two computer systems set up a connection (handshake), they exchange a one-time session key that they will use for simple symmetric encryption / decryption of their messages. That session key is encrypted with the PKI algorithm so Bob can read it but Zorro can not.

III. Certificates

This works fine, especially for 1: 1 communication, where Alice and Bob know and trust each other. However, the beauty of a Public Key Infrastructure is that everyone can know and use the public key, knowing that only the private key owner (Bob) can read the messages. We use that massively with HTTPS connections to a website: we use the public key of that website. On the Internet, however, the problem arises that we are strangers to one another, and can not be sure who is behind that website - there are ways to redirect traffic to, for example, `www.ing.nl` to your own non-ING server (spoofing). Therefore we started using certificates.

Parties that communicate with each other but do not know each other personally, use a third party – Cesar – whom they both know and trust as Certification Authority (CA). Bob sends the CA a Certificate Signing Request (CSR), containing his public key, and a Subject: a unique name (Distinguished Name), which includes his name (Common Name) and organization ([1]). CA Cesar acknowledges that the public key is indeed from the organization of that name, which he knows. The CA does this by signing the CSR with his own private key. Such a public key signed by the CA is a certificate (according to X.509 standard, see RFC5280: PKIX [4] and RFC6818: Clarifications [7]). Anyone can then check that certificate with the public key of the CA.

That is shifting the problem: how do you know that CA's public key is indeed of the CA? It may be signed by another CA, but eventually there is a "root" CA: the last Certification Authority that everyone knows and from whom everyone has the public key that they trust. PKI works with certificates, so for technical reasons the CA's public key must also be in a certificate. The root CA does this by signing its public key with its private key: a self-signed certificate.

Of course, we can skip all the hassle with CAs, Bob can also give Alice a self-signed certificate and that works well and safely. There is nothing wrong with self-signed certificates.

Most people believe that self-signed certificates are unreliable and may not be used, so I say it again:

There is nothing wrong with self-signed certificates!

A root CA certificate is by definition a self-signed certificate. A CA is only required as an intermediary between parties who do not know each other. If Alice and Bob do know each other, they can exchange self-signed certificates through some secure way, and then communicate at least as securely as when they use certificates signed by any CA.

IV. CA's

OK, there is something wrong with using self-signed certificates: you can only use them for 1:1 connections.

[1] Since 2000 however one should fill the Subject Alternate Name extension in the certificate. See section 3.1 on p.5 in ref. [3].

If a computer X has to communicate with 10 other computers, they all must include its self-signed certificate in their so-called "truststore" (the database of all certificates you want to use); or X must include 10 self-signed certificates from other hosts in its truststore; or another distribution, depending on who starts the contact. So it's useful if they all use certificates that are signed by the same CA: then they only need to include the CA's (self-signed) certificate in their truststore. The authentication of their own certificates can then be checked by the others and they can communicate securely.

Pitfall 1: use certificates from a public CA within your data center.

It is insane to use certificates signed by a public Certification Authority like Verisign to secure connections between computers on your own Intranet. Then you need to send Verisign a CSR to confirm that your computer A is yours, and you will then receive a certificate that will allow your computer B to determine that your computer A is yours, and Verisign is getting rich laughing. You can do that yourself. Always use your own CA on your Intranet.

In addition, certificates have a limited duration of validity - typically 1 or 2 years. The reason is preventive maintenance. Private keys are bits that on a computer, and anyone who has access can copy them and use them elsewhere. Worse, that is often necessary (see Pitfall 5: copy private key.). If your private key is stolen or cracked, you lost your privacy and integrity, and often you do not even know. Just in case, it is wise to replace certificates regularly.

Pitfall 2: reuse a CSR to renew a certificate.

I have seen administrators have the habit of re-submitting the old Certificate Signing Request to the CA when a certificate expires; they get a new certificate, which they then install with the existing private key. They do not have to create a new keypair. But compare that with the maintenance of an airplane: instead of preventively replacing all old nuts and bolts with new ones, we just turn the old ones back in; whether they will break tomorrow because of fatigue we do not know. The reason for this shortcut is that the entire Key Management Procedure is sometimes as tedious as if you have to cut the new nuts and bolts yourself. Then you choose the easy way.

On the other hand, the limited duration of certificates is unnecessary: as long as private keys are not compromised, you can safely use them and no new certificates are required. If your certificate expires, you can reuse the old CSR as well. However the point is that you are not sure whether your private key has been compromised over time or not.

V. Revoke certificates

If you find that your private key has been compromised, then you have another problem: the whole world has a public certificate in which your CA claims that you are the sole owner of that private key. You can not recall all those public certificates. The solution that has been implemented is that the CA also maintains a Certificate Revocation List (CRL): if you report to the CA that the certificate is no longer valid, then the CA will put it in its CRL. You should hope that your - unknown - communication partners take the trouble to retrieve the CRL and consult it EVERY time that they use your certificate; otherwise they will talk with Zorro unnoticed. In practice, this is usually NOT done.

CRLs are also very outdated: instead one has developed the Online Certificate Status Protocol (OCSP) that allows you to retrieve a status of a certificate from an Internet service.

OK, this is also wrong with self-signed certificates: there is no CA, so no CRL. On the other hand, you use these with 1: 1 connections so you know who your communication partners are and you can let them know that they need to remove your self-signed certificate from their truststore.

This is also an argument against using a public CA within your Intranet: then you need to make an excursion over the Internet to your public CA for each internal connection, in order to check the CRL or OCSP server. Or alternatively you need to regularly, *e.g.* daily, import the CRL of your CA into an internal OCSP server: but then you'll run behind the facts even more.

OCSP also does not work well: the service at the CA receives a massive number of requests and usually can not process the load.

So, one has come up with something else: TLS Certificate Status Request also called "OCSP stapling". In this case, the certificate holder (the website) first presents its certificate again to the OCSP server of his CA, and has it staple a signed time tag to his certificate, indicating that the certificate was still not revoked at that moment. If the client then requests a connection, the service sends that labeled certificate as authentication, and the client can decide if that is good enough and will not himself retrieve the status of the certificate from the OCSP server.

I think that's a very poor solution: a Certification Authority issues an ID which is valid for some time, but needs to be confirmed over and over again. What is the use of such a certificate? Imagine having a driver’s license valid for 10 years: but every time you step into your car to drive on the digital highway, you must first visit the town hall to get a stamp that your driver's license is still valid .

Limited validity, CRL, OCSP, Stapling: it's always "too little, too late". IMNSHO PKI is "broken by design" at this point. Not that I know a solution for this, otherwise I would not be where I am now.

VI. Authentication

One reason for writing this article was a discussion I had with a colleague. Apparently, I did not pay attention at school, because I thought that the TLS handshake also checks the contents of the certificate, in particular if the certificate has been issued for the name of the service that you call. That does not appear to be the case. The TLS handshake protocol only checks if you have a valid certificate, *i.e.* signed by a known CA and not expired.

It is as if Customs only checks if you have a valid passport, but does not look inside if the picture matches your head. You could have found, borrowed, bought, or stolen someone else's passport: it is not checked (except if it is listed as missing - the CRL).

As already mentioned in section 3.1 on p.5 in the RFC2818: HTTP over TLS, from 2000 [3]:

" If the hostname does not match the identity in the certificate, user oriented clients MUST either notify the user (clients MAY give the user the opportunity to continue with the connection in any case) or terminate the connection with a bad certificate error. Automated clients MUST log the error to an appropriate audit log (if available) and SHOULD terminate the connection (with a bad certificate error). Automated clients MAY provide a configuration setting that disables this check, but MUST provide a setting which enables it."

In RFC7525: TLS Recommendations (2015) is stated in section 6.1 on p.17 [8]:

Host Name Validation

Application authors should take note that some TLS implementations do not validate host names. If the TLS implementation they are using does not validate host names, authors might need to write their own validation code or consider using a different TLS implementation.

It is noted that the requirements regarding host name validation (and, in general, binding between the TLS layer and the protocol that runs above it) vary between different protocols. For HTTPS, these requirements are defined by Section 3 of [RFC2818].

Readers are referred to [RFC6125] for further details regarding generic host name validation in the TLS context. In addition, that RFC contains a long list of example protocols, some of which implement a policy very different from HTTPS."

This refers to pp.21..22 in RFC6125: Service Identity, from 2011 [5]:

6. Verifying Identity Service

This section provides rules and guidelines for implementers of application client software regarding algorithms for verification of application service identity.

6.1. Overview

At a high level, the client verifies the application service's identity by performing the actions listed below (which are defined in the following subsections of this document):

1. The client constructs a list of acceptable reference identifiers based on the source domain and, optionally, the type of service to which the client is connecting.

2. The server provides its identifiers in the form of a PKIX certificate.

3. The client checks each of its reference identifiers against the presented identifiers for the purpose of finding a match.

4. When checking a reference identifier against a presented identifier, the client matches the source domain of the identifiers and, optionally, their application service type ."

Yeah, sure, hahaha.

So it is up to the client application to verify that the subject of the certificate matches the service that the application calls. I can understand this: the underlying security library does not know from where the application has received the certificate. In order to have the application verify the certificate, it must be able to retrieve the details from the certificate via the library, and actually do so. But frequently the application never gets to see the certificate: often TLS is terminated at a proxy. Then there is no identity known to check, never mind to subsequently authorize it (see section “X. Authorization”).

Browsers do check the certificate: MS Edge refuses to make a connection with a site that does not match the certificate; as a user, you can not overrule that. Chrome still gives you that option. Only FireFox will provide you with the necessary information about the certificate and give you the opportunity to accept it.

Pitfall 3: not checking the Subject of the certificate.

My impression is that custom-made applications usually do NOT verify the certificate. So usually authentication is incomplete.

VII. Identity

There is also a problem with the identity for which a certificate is issued. As shown by the old RFC quote, all this is designed for an Internet where computer hosts talk to each other and identify themselves with their hostname. It is and was the default to put host names in the Common Name in the Subject of a certificate. Now that has never been unambiguous. A host has a single hostname (*e.g.* `gandalf` was a popular host name) for which you can issue a certificate. On the Internet it has an IP address for which you could also issue a certificate. But a host can have multiple interfaces and therefore IP addresses – in the past 2 or more ethernet cards, nowadays virtually unlimited Virtual IP Addresses or IP Aliases: another certificate for each? The identity is not for the host as a whole. Furthermore, a host on the network receives from DNS a Fully Qualified Domain Name (*e.g.* `gandalf.example.com`), for which a certificate is required with appropriate Subject. But nowadays, countless DNS entries can refer to one and the same host - a host thus has multiple identities and would require different certificates.

To solve this, early on one has introduced the Subject Alternative Names (SAN) extension: a list of names for which the certificate is also valid; again see RFC2818: HTTP over TLS, section 3.1 on p.5 [3]. Since then (2000), the Common Name in the Subject has been deprecated by the IETF, but it has taken a long time before it became supported. *E.g.* only since Java 7 (2011), the standard keytool of Oracle / Sun supports SANs [10].

Also see RFC6125: Service Identity [5].

Nowadays, we mainly have web services that have URIs that lead through DNS to any host where the web application is listening to port 80 (HTTP), 443 (HTTPS), or another port specifically configured for that application. Therefore, those URIs do not match the host name, and thus require a certificate with another Subject.

VIII. Redundancy

Moreover, we do not want such a web service to be bound to a host: there must be redundancy, so the application must run on multiple hosts. But they all have to support that one certificate issued to identify and authenticate that service.

One solution is to put a reverse HTTP proxy in front of all the application hosts, such as a Load Balancer. The URI for the web service then leads through DNS to a Virtual IP Address on that proxy. The private key for the certificate issued for that URI lives there, and HTTPS is terminated there. The Load Balancer then distributes the traffic destined for the web service over the backend hosts on which instances of the application run.

But sometimes one does not want security termination on a front-end proxy, but the Security office requires that this only happens on the application hosts themselves. One can then also secure the connections between the Load Balancer and the backend web servers with HTTPS: the web servers use host-specific certificates in that case.

IX. Responsibility

Pitfall 4: dump application security on the infrastructure.

Ultimately, it is applications that generate and exchange data, and to ensure integrity and confidentiality of the data, end-to-end encryption between applications must be arranged. That can be done with a Public Key Infrastructure. However, as is apparent from the things described above, this is often delegated to the infrastructure: the underlying Middleware, transport layer, network.

Organizations are looking for a generic solution. Architects do not want everyone to invent a wheel, and good programmers have a reputation of being lazy, and security is difficult so they like to outsource it and purchased applications do not meet specific local standards anyhow. So a generic solution is being built on the common infrastructure layer. But then there are always gaps in the chain. Moreover the generic solution can never meet specific requirements of an application. The infrastructure cannot know who uses it, what data are exchanged, and who has to be allowed access: all that can only be managed at the level of the application.

If the Security Officer, architects, and application builders are doing things really well, they build end-to-end encryption to ensure data integrity and confidentiality and to allow mutual authentication and

authorization. Then the application itself must do PKI. But then it is necessary to distribute the keystore with the private key across all hosts where the application is running.

Pitfall 5: copy private key.

This is not really a pitfall but a design error in the PKI tools IMNSHO. It is necessary to copy the private key, if only as backup, but also for redundancy as described above. But this means that it is also possible to steal the key. There have been appliances for sale keeping the private key in hardware (Hardware Security Module); such a thing often also has a motion detector that destroys the key if someone physically tries to steal the device. But most people use software keys. And bits can always be copied.

So, you should overthink carefully and prescribe it in a Key Management Procedure, who when and how can access a software private key and the truststores, and how to install these securely on multiple hosts; preferably automated. Unfortunately, it will sometimes happen that someone gets root and by hand puts the files wherever they are needed. Then you rely 100% on your employee and 0% on technology.

A better solution is to put the private keys and truststores on a shared disk, and mount it on all nodes that needs it. Then the keys stay in one place. Redundancy / backup is usually taken care of if the shared disk is part of a Storage Area Network.

X. Authorization

So quite a few applications stay far from the security components. They do not check the authentication, so they do not know who they are talking to. Then they can also not authorize anyone to perform certain actions of their service.

Pitfall 6: authentication <> authorization.

PKI at best tells you who you are talking to (and that can be done confidentially): authentication. That by itself is useless if you then not also make a deliberate decision whether you want to do business with that client or service, and if so, what you want to share and what it can or can not do on your system / with your service / with your data: authorization. PKI can not handle that [2]. A lot of people appear to not

[2] There are other possibilities with infrastructure: web servers like Apache have authentication and authorisation modules. Also you can set up a policy server that applications can use for authorizations.

understand this. If you contact them with any certificate, you are welcome and you can do anything. And as we saw above, it is likely that a number of choices have been made beforehand that compromise the security of even authentication.

It is even a design model: a trust zone. An internal CA issues certificates to services that communicate across the Intranet over HTTPS or another TLS protocol. There is only a check at PKI level that you have a certificate issued by a trusted CA: then it's "one of us", and among us everyone is allowed anything. I have seen situations where every host uses the same certificate, which just works in such a construction.

Pitfall 7: public CAs.

Of course, in this age of outsourcing and cloud services, someone will have to talk to an external party.

That must be done securely, so everything has to go over TLS. But that third party uses a certificate from another, public CA. And such an external commercial service really can not use a certificate signed by your private CA. So you have to also put the public root CA certificate into all your truststores. Of course, doesn't everybody do that with well-known CAs? MS-Windows comes with a full list of root CA certificates preinstalled too, so that MS-Edge can speak HTTPS with all banks and webshops. That only makes things safer.

Right?

Well, then suddenly everyone who has a certificate from that CA gets into your trust zone. And such a public CA loves to sell a certificate to anyone who wants it. Also to `werobyourbank.ru` .

But such a certificate from a CA that everyone uses can be trusted, can it not?

Yes, for a certain degree of authentication. It is up to the recipient to decide if he wishes to grant authorizations. And a certificate is certainly not a Statement of Good Behavior. But a lot of people seem to think it is.

Pitfall 8: intermediate CAs.

Not every "leaf" certificate (for the end user) is signed by a Root CA. Often there is a chain of 1 or more Intermediate CAs. One reason may be that for security, one does not want to put the Root CA onto the network at all: then a cumbersome manual procedure is required to get CSRs on and certificates from the system, so this is only done for subordinated CAs which then issue the "leaf" certificates. Or one wants to distinguish sub-domains using separate Intermediate CAs. This will be a nuisance after a reorganization,

or worse, if a department is sold: forever certificates will remain in circulation for which you remain the responsible Root CA, but you are no longer in control of their issue. It seems better to use multiple independent Root CAs that issue leaf certificates directly.

XI. No CAs?

Above yet another design error became apparent of PKI and the use of a public CA, namely that the CA will sign a certificate for any domain, and that anyone who accepts the CA in his truststore must accept all those certificates.

With a protocol such as DNS-based Authentication of Named Entities: DANE (RFC6698: DANE for TLS (2012) [6], RFC7671: DANE Operations (2015) [9]), one can try to replace CAs by DNS. But DNS also has fundamental security issues, which one tries to remedy by Domain Name System Security Extensions (DNSSEC).

XII. SHA-1

The issue is as follows.

In a TLS handshake, message hashes are transmitted (Message Authentication Code); a certificate also contains a hash of the Subject and of the Signature. By recalculating or comparing hashes, one can check that those data have not been compromised. A commonly used hash function is called SHA-1, related to MD5. It is known that it is not completely safe.

A hash is a summary of another, larger piece of data; SHA-1 hashes have a length of 160 bits. There are therefore "only" 2 ^ 160 = 10 ^ 48 different SHA-1 hashes. But there are many, many more possible data files. So there are countless blocks of data that give the same hash. Therefore, if you find such a so-called "collision", you do not know which of the two is the data sent, so it is no longer guaranteed that the data is not tampered with. The chance is extremely small, and you can not abuse it to forge anything specific: but such a first collision has been found now [1].

That has been coming for a while, and the CAB Forum, a consortium of Certification Auhorities and Browser manufacturers, has decided not to issue any certificates with SHA-1 hashes anymore after 1 January 2016, and that browsers will no longer accept SHA-1 starting from 1 January 2017 [11], [12].

This is why everyone has been busy replacing SHA-1 certificates with new ones using the safer SHA-2 (usually 256 bits).

XIII. Conclusion

As my story hopefully shows, the SHA-1 hash function is not the big problem. The way in which a Public Key Infrastructure is implemented and used is often a greater risk.

Links

1. https://www.cwi.nl/news/2017/cwi-and-google-announce-first-collision-industry-security-standard-sha-1
2. https://en.wikipedia.org/wiki/RSA_(cryptosystem)
3. HTTP over TLS: https://tools.ietf.org/html/rfc2818
4. PKIX Certificate and CRL Profile: https://tools.ietf.org/html/rfc5280
5. Service Identity: https://tools.ietf.org/html/rfc6125
6. DANE for TLS: https://tools.ietf.org/html/rfc6698
7. RFC5280 Clarifications: https://tools.ietf.org/html/rfc6818
8. TLS Recommendations: https://tools.ietf.org/html/rfc7525
9. DANE Operations: https://tools.ietf.org/html/rfc7671
10. http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html
11. https://cabforum.org/wp-content/uploads/Baseline_Requirements_V1_1_9.pdf
12. https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.4.2.pdf