# BACHELOR PAPER

Term paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Information and Communication Systems and Services

# Automatic Certificate Based Authentication on the Web

By: Benjamin Akhras

Student Number: 1310258008

Supervisor: Dipl-Ing. Mag. Dr. Priv.-doz. Edgar Weippl

Vienna, July 21, 2017

FH University of Applied Sciences
TECHNIKUM WIEN

# Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (for example see §§21, 42f and 57 UrhG (Austrian copyright law) as amended as well as §14 of the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

In particular I declare that I have made use of third-party content correctly, regardless what form it may have, and I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see §14 para. 1 Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, July 21, 2017 Signature

# Kurzfassung

Authentifizierung ist eine wichtige Aufgabe, die die meisten Computersysteme lösen müssen. Stand der Technik für Computersysteme ist eine Multi-Faktor-Authentifizierung. Diese Arbeit vergleicht Wege, um öffentliche Schlüsselinfrastrukturen in IT-Systemen zu automatisieren, um ein solches Multi-Faktor-Authentifizierungssystem zu schaffen. Eine öffentliche Schlüsselinfrastruktur ist ein System, das dazu dient mit Zertifizierungsstellen und den Zertifikate, die sie unterzeichnen, besser umzugehen. Diese Zertifikate stellen den Faktor possession oder ownership dar. Diese Arbeit ist eng mit der Forschung in der Abhandlung "Measuring Password Quality with Natural Language Encoders" verbunden, da in dieser Arbeit der Faktor knowledge, welcher obligatorisch für die in dieser Arbeit beschrieben Multi-Faktor-Authentifizierung ist, beschrieben wird. Besitz oder ownership ist ein Faktor, der den Besitz eines Tokens erfordert. Dieses Token kann in mehreren Formen erscheinen, beispielsweise als Smartcard, die einen privaten Schlüssel enthält oder als Datei auf einem beliebigen Speichergerät. Darüber hinaus werden allgemeine Vorteile einer asymmetrischen Authentifizierung sowie Multifaktor-Authentifizierung besprochen. Weiters wird ein genauerer Blick auf die Automated Certificate Management Environment, Simple Certificate Enrollment Protocol, Mozilla Persona und Google Token Bind und seine wichtigsten Features und Referenzimplementierungen geworfen. Außerdem wurde ein Docker-Container mit einer funktionierenden Implementierung von Token-Bindung in Form eines Nginx-Plugins gebaut. Als Ergebnis werden die notwendigen Schritte um die Verbreitung von Multi-Faktor-Authentifizierung mithilfe eines automatisierten Zertifikatsmanagements zu erhöhen, diskutiert.

**Schlagworte:** Zertifikate, Sicherheit, Authentifizierung, Public-Key

# Abstract

Authentication is an important task that most computer systems need to solve. State of the art computer systems are based on a multi-factor authentication system. This paper compares ways to automate public key infrastructures in IT systems to create such a multi-factor authentication system. A public key infrastructure is a system to better handle certificate authorities and the certificates they sign. These certificates represent the factor possession. This paper is tightly connected to the research made in the paper "Measuring password quality with Natural Language Encoders" as the research there covers the factor knowledge mandatory for the multi-factor authentication addressed in this paper. Possession or ownership is a factor that requires the possession of a token. This token can appear in multiple forms, for example as a smart card holding a private key or only as a file on an arbitrary storage device. Moreover, general benefits of an asymmetric authentication as well as multi-factor authentication are discussed. A closer look on the Automated Certificate Management Environment, Simple Certificate Enrollment Protocol, Mozilla Persona and Google Token Bind and its key features is taken. Furthermore a Docker Container with a working implementation of Token Bind in form of a nginx plugin was built. As a result, this paper discusses the steps necessary to achieve a wider use of multi-factor authentication through the use of an automated certificate management.

**Keywords:** certificate, security, authentication, public key

# Contents

# Contents

# 1 Introduction & Background

Authentication is one of the most complex and important tasks modern computer systems need to solve. Nowadays authentication is achieved with the use of at least two of the factors knowledge, ownership and inherence. In contrast to the research in "Measuring password quality with Natural Language Encoders", where only the factor knowledge was addressed, this paper only discusses the factor possession. This factor requires the possession of a token. This token can appear in multiple forms for example as a smart card holding a private key or only as a file on an arbitrary storage device. For simplicity in the rest of this paper this factor will only be referred to as "token". For the security of an account it is mandatory that it remains secret. To use passwords in a secure manner multiple problems must be addressed.

"Password strength" is a complex issue, to improve it one must first assure that passwords are truly randomly created. Usually some mandatory rules (minimum length, number of digits, etc...) are enforced, but those rules have proven not to be very efficient. This is easily seen with passwords like 'Password1', that follows these rules but still does not provide sufficient complexity. Furthermore the creation and management of passwords is usually seen as a burden by the users. The fact that users are bad at maintaining good passwords, and are often targeted by hackers via social engineering lead to a system where security departments typecast users as insecure. Users are the enemy the security department starts to fight by blocking access to huge parts of the Internet instead of training them in secure usage of it. The users from there start to fight the rules, resulting in further decrease of security[1]. The most well known of those 'rules' is a forced password rotation, that usually results in the password ending up on a Post-it note on the Screen or a rotation of the last character. Nowadays there are more then enough peer reviewed papers proving the inefficiency of these methods[5]. Since a password represents the factor knowledge it is usually something that is easy to remember, something that fits in the memory of a user. But there are several problems with how our brain tries to come up with passwords, mostly that our brain is really really bad at gathering entropy, since nothing our brain chooses is random. The human brain is a per design biased random number generator, and a security system based on a biased random number generator is considered unsafe. This is because cryptographically secure sequences should always be

independent of each other. So if I can think of a truly unbiased random number, or letter, I would first have to completely forget about this first character to think of an unbiased second. This problem is described in the Ironic process theory, that describes a phenomenon whereby if one tries to deliberately suppress a certain thought, in our case the last number, it makes it even more likely to surface in the next.

> Try to pose for yourself this task: not to think of a polar bear, and you will see that the cursed thing will come to mind every minute. — Fyodor Dostoevsky, Winter Notes on Summer Impressions, 1863

To make it more tangible for a user to manage his credentials one can use asymmetric cryptography. He could have a physical token in form of a smart card. Everyone knows how to physically secure goods like money or keys, since humans have been positively selected for that for the last million years.

## 1.1 Asymmetric Cryptography

An asymmetric cryptosystem or public-key cryptosystem is a cryptographic method in which, unlike a symmetric cryptosystem, the communicating parties do not need to know a common secret key. Each user creates his own key pair consisting of a secret part (private key) and a non-secret part (public key). The public key allows anyone to encrypt data for the owner of the private key, to verify the digital signatures, or to authenticate it. The private key allows its owner to decrypt encrypted data, generate digital signatures, or authenticate using the public key.

An asymmetric system is about separating roles. With a password all parties needing access to the system need to know the password at some point. The theoretical basis for asymmetric cryptosystems are trap functions. These functions are easy to compute but are virtually impossible to invert without a secret, this secret is the "trap door". The public key is then a description of the function, the private key is the trapdoor. A prerequisite, of course, is that the private key from the public can not be calculated. In order for the cryptosystem to be used, the public key must be known to the communication partner. The security of all asymmetric cryptosystems is therefore always based on the assumption that P is not NP. As a rule, however, these assumptions strongly suggest that they are true. The P-NP problem (also $P \stackrel{?}{=} NP$, P versus NP) is an unresolved problem of mathematics and theoretical computer science, especially of the complexity theory. The question arises as to the relationship between the two complexity classes P and NP. Consider the subset sum problem, an example of a problem that is easy to verify, but whose answer may be difficult to compute. For example, take a list of integers " -2, 5, 42, -3 " and now think of two problems. First you want to know if any nonempty subset of them sums up to 0?
The answer is "yes, because the subset {-2, -3, 5} adds up to zero". This is the verify process and it can be done quickly since it needs only three additions. Second try to add up all possible subsets of numbers, this a much harder task. Unfortunately there is no known algorithm to find

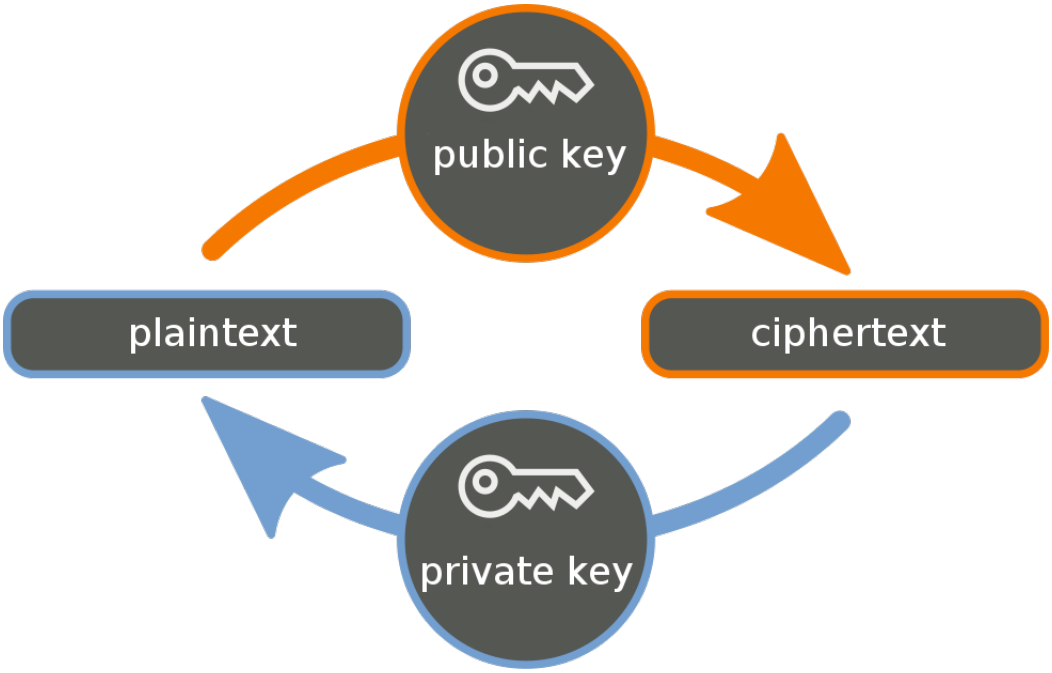


Figure 1: Using a public key to encrypt and a private key to decrypt

the first subset in polynomial time only in exponential time. P = NP would be true if such an algorithm would exist. But if it turned out that P $\neq$ NP, it would mean that there isn't such an algorithm and proof one and for all that there are problems that are harder to compute than to verify. One of the key advantages of asymmetric cryptography is that they reduce the key distribution problem. In the case of symmetrical methods, a key must be exchanged via a secure, tamper-resistant channel before use. But for an asymmetrical connection the public key is not secret and therefore the channel does not need to be secure. The only task remaining is to correlate public and private key, this is sometimes achieved via a "Web of Trust". But deploying user certificates is a very complex and often expensive task. To do so in an ordered manner one needs a public key infrastructure.

## 1.2 Public Key Infrastructure

A public key infrastructure (PKI) is a complex system with multiple tasks relating to the management of certificates. These tasks include the creation, storage, and distribution of digital certificates, all tasks which usually require human action. The task which requires the most intensive human action is the initial creation of the certificate, in most literature as well as in this paper referred to as "enrollment". Since a certificate is used to verify that a particular public key belongs to a certain entity it is necessary that only this entity can enroll for the certificate. The PKI furthermore securely stores and distributes these certificates in a central repository and if necessary also revokes them. The key components of a PKI are:

- A certificate authority (CA) that stores, issues and signs the digital certificates
- A registration authority which verifies the identity of entities requesting their digital certificates to be stored at the CA
- A central directory, a secure location in which to store and index keys
- A certificate management system managing things like the access to stored certificates or the delivery of the certificates to be issued.
- A certificate policy

## 1.3 Trusted Platform Module

A Trusted Platform Module (TPM) is a chip that expands a computer or similar device with basic security features. These functions can serve multiple purposes such as protecting data, or also goals of the post-service control of computer security features. The chip behaves like a fixed smart card, but with the important difference that it is not bound to a concrete user (user instance) but to the local computer (hardware instance).

## 1.4 Integrated Circuit Card

An Integrated Circuit Card (ICC) is a special plastic card with built-in integrated circuit (chip), which contains a hardware logic, memory or even a microprocessor. For security reasons a smart card should be preferred over a TPM, since it is much easier to safely secure a smart card from physical access than the actual computing device.

## 1.5 Random Number Generation

The most critical part of any private key generation is the randomness the key is inherited from. Usually some prime number, but for the private key to not be recalculable this prime number has to be truly random. There are two types of randomness one must distinguish. On the one hand there are pseudorandom number generators (PRNG). They won't produce a true random number, but if used correctly, for example with the Yarrow algorithm[11], the output will be cryptographically secure. On the other hand there are true random number generators (TRNG) also called hardware random number generators. They use atmospheric noise or thermal noise to generate random numbers. These true random numbers are more secure from a cryptographic point of view, but require dedicated hardware, and are therefore rarely used. In practice cryptographically secure pseudo-random number generator (CSPRNG) are most popular, they are PRNGs with a block cipher applied. A block cipher is turned into a

stream cipher by counting up an integer and encrypting it each time. After $2^{n2}$ colling blocks become more likely and therefore insecure. Today 128-bit blocks are considered secure for most applications, depending of the amount of random data needed.

# 2 Automated Public Key Infrastructure

## 2.1 Certificate Management Environment

The Automated Certificate Management Environment (ACME) is a protocol for the automatic verification of the ownership of an Internet domain and serves the simplified issuance of digital certificates for TLS encryption. This environment made issuing TLS certificates a very cost-effective and automatized process. It was defined by the Internet Security Research Group for use in the Let's Encrypt service.

The protocol is based on JSON-formatted messages exchanged via HTTPS. This API approach is one of the key benefits of ACME, enabling the use of a wide variety of clients to access the service. There is one reference implementation for the client. It is written in python and called "Certbot". But there are various implementations written for the POSIX-shell or the Microsoft Windows "Powershell". These clients usually have additional features as automatic configuration of Server software for the use of the certificate. For example Certbot can configure the apache web server to use the certificate. But even the Powershell version can configure the Microsoft Internet Information Services (IIS) to use the newly issued certificate.

The draft protocol is currently available as an Internet draft[2].

### 2.1.1 Let's encrypt

Let's Encrypt is a certification authority that is based on the Automated Certificate Management Environment[10]. It started operating in 2015 as a beta service and ended this beta phase on April 12, 2016. It offers free X.509 certificates for Transport Layer Security (TLS) for Websites. The certificates are not only free but are created in an automated process, that replaces the complex manual procedures used to create, validate, sign, set, and renew certificates that usually were in place before. A simple way to use and configure TLS for everyone was one of the key factors of Let's encrypt's success.

#### Boulder

This is the first implementation of an ACME-based Certificate Authority. The ACME protocol allows the CA to automatically verify that an applicant for a certificate actually controls an iden-

tifier, and allows domain holders to issue and revoke certificates for their domains.

#### Certbot

Certbot is the reference implementation for an ACME client. It was written by the EFF as part of the Let's encrypt project. It creates the necessary challenge response files for a successful validation of a domain. Based on this validation served via http, the CSR for that domain, also created and sent by Certbot, gets signed by Let's encrypt.

Unfortunately Let's Encrypt announced that they would not support Client Certificates in the near feature[15].

## 2.2 Simple Certificate Enrollment Protocol

The Simple Certificate Enrollment Protocol (SCEP) is a protocol used for enrollment and management operations of certificates in a public key infrastructure. Originally proposed as an IETF Draft by Cisco in the year 2000 the draft was abandoned in the year 2012[14]. Since then a wide variety of appliances and applications implemented the standard in one way or another. Therefore in the year 2015 a new draft was released under a new name but basically with the same concepts. The protocol is designed to manage a secure PKI, as in-band certificate revocation transactions[8].

### 2.2.1 Dogtag Certificate System

Since a PKI is a very complex system the Dogtag Certificate System (DCS) splits the various parts of it into six flexible subsystems. The for this paper important two subsystems of the DCS are:

- The Certificate Authority is the subsystem that provides certificate management functionality for issuing, renewing, revoking, and publishing certificates and creating and publishing Certificate Revocation Lists (CRLs).
- The Registration Authority (RA) is a subsystem that plays the central role in enrollment. It verifies the enrollment requests so that the CA will sign them.

Whereas the following subsystems play little or no role for the research in this paper:

- The Data Recovery Manager (DRM) can be important especially if data is encrypted with the private keys issued by the PKI. If the private key is lost the data otherwise would be lost ultimately.

- Online Certificate Status Protocol (OCSP) is a subsystem that manages an OCSP responder. It manages stored CRLs for CAs and can distribute the load for verifying certificate status.
- The Token Key Service (TKS) manages master keys required for the secure channels needed by the token management system. These secure channels are needed for any privileged operations such as key generation.
- The Token Processing System (TPS) provides the registration authority functionality in the token management infrastructure.

The Certificate Authority and the Local Registration Authority (LRA) are the two main features of Dogtag that are mandatory if one wants to set up a PKI. A Local Registration Authority is, given that it is set up correctly, a well proven way of authenticating enrollment requests in an organization or organization like group. There are even governments acting as a LRA, for example Estonia[7] gives out identity cards with private keys stored on them. But unfortunately it is not feasible for an online enrollment. A local registration authority is a human with an administrative access to the public key infrastructure able to authorize the fulfillment of certificate requests. Sometimes the LRA itself fulfills the request, this usually happens if a private key is generated on dedicated hardware with specialized random number generators (RNG). This private key then gets stored on a smart card handed out to the certificate entity.

# 3 Web Authentication

For decades the cry for some sort of universal client on a typical computer user's machine was heard. With the wide use of cookies a state fullness could be added to the statelessness of HTTP and so the Web Browser became this universal client. But nowadays clients not only support cookies put also TLS client certificates. Unfortunately they are very complicated to use, and the initial setup is even more of a burden, especially compared to a password based login. In the following the TLS Handshakes will be explained and methods to simplify their usage in modern browsers will be discussed.

## 3.1 TLS Handshake

### 3.1.1 Server Authentication Handshake

Fig. 1 shows protocol messages exchanged between a client and server during TLS protocol negotiation (for simplicity, we cover only RSA key exchange mechanism).

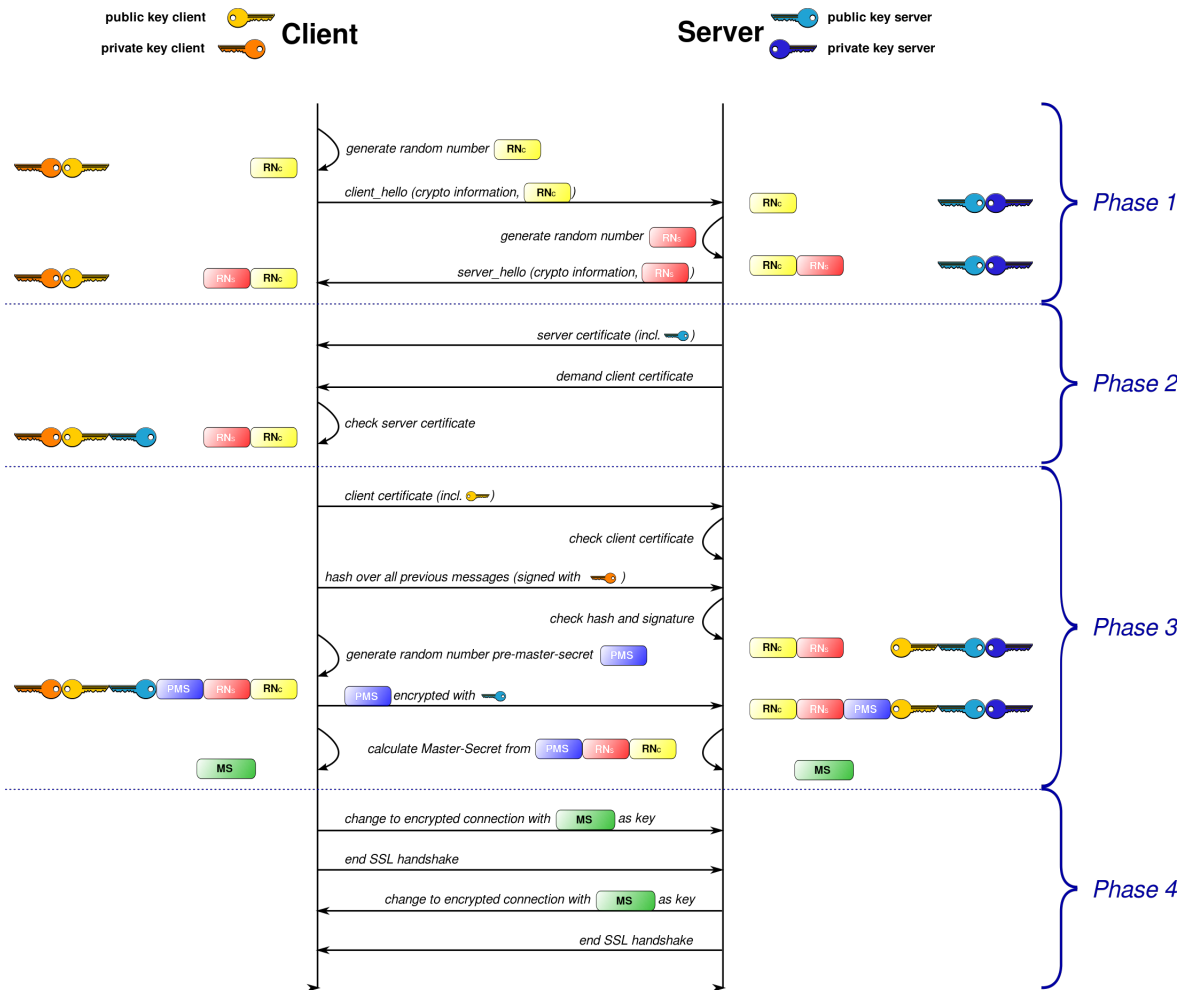


Figure 2: SSL Handshake

Exchanging encrypted and integrity protected application data is only possible if the TLS handshake has already been completed in order to determine the protocol parameters such as supported cipher suites.
Establishing a TLS connection first of all requires a ClientHello message. It consists of a number of cipher suites supported by the client as well as the client‘s randomness which can vary from case to case.
The server responds to the client‘s message with the ServerHello message, which consists of the server‘s randomness. One of the cipher suites from the ClientHello message is also chosen and forms part of the ServerHello message. This selected cipher suite is then required for the TLS session key exchange, as well as for integrity protection and encrypting.
The next step is the Certificate message: Therein the server supplies at least one X.509 certificate that the client will use to establish the certificate chain. In the Certificate message the server will first provide its own certificate with a public key. The server holds the private key to this certificate. After providing the certificate the server will send a ServerHelloDone message. Now a random value, the pre-master secret, is generated by the client. This value should be applied by both the client and the server in order to acquire symmetric keys needed for the cipher suite selected before. Furthermore, the client encrypts the pre-master secret using the server‘s public key received in the server‘s certificate.
The next part of the handshake process is the ClientKeyExchange message, where the pre-master secret is sent to the server in encrypted form. The client also indicates the protection of all following messages by the negotiated cypher suite and symmetric keys to the other party by sending a ChangeCipherSpec message.
Finally, the encrypted Finished message is sent containing a hash of all preceding messages which has to be verified. In order to do so the server compares the decrypted hash to the hash of all preceding handshake messages between server and client. The latter carries out a similar verification using the server‘s Finished message. After the verification is completed, encrypted application data can be exchanged.
The pre-master secret is encrypted by the client using the public key of the server. Therefore only the server holding the corresponding private key is able to decrypt the ClientKeyExchange message. Decrypting this message is necessary for gaining the symmetric keys which are needed for secure further communication. Generally speaking, TLS can resist active and passive network attacks, given that the client uses a public key that belongs to the right server – the one the client wants to communicate with. The authenticity of the public key supplied by the server in the Certificate message is usually verified using public-key infrastructure (PKI)[16].

### 3.1.2 Client authentication Handshake

In order to perform a client-authenticated TLS handshake the server also needs the client‘s certificate, which it demands with the CertificateRequest message sent to the client. This message provides the client with the distinguished names (DNs) of certificate authorities (CAs) trusted

by the server. The client can then use this list to select a suitable certificate and send it to the server as part of the client's Certificate message. It may contain a number of various certificates which the server can use to form the certificate chain regarding the CA it trusts.
In case there is no client's certificate or the client does not wish to carry out CCA, the client can provide the server with an empty Certificate message. By doing so the decision whether to realize the handshake even without the client's certificate is left up to the server.
Calculating all prior handshake messages transmitted between client and server and signing it with the client's private key gives proof of the client's access to the private key corresponding to the public key contained in the client's certificate. The signature is transmitted in the client's Certificate Verify message, which is omitted in case the client has already provided the server with an empty Certificate message.
The main advantage of TLS CCA in comparison to authentication methods where a shared secret is disclosed to the server, is that in the TLS CCA process a client proves its ability to access the private key to the server without ever disclosing the private key itself to the server.
Now, the signature provided using the client's private key is bound both to the client's and the server's randomness as well as to the server's certificate and the encrypted pre-master secret sent by the client. Therefore an attacker, who has already retrieved the signature in a MITM attack, cannot reuse it in any other TLS handshake performed either with the legitimate server or any other server.
As a result using TLS CCA makes it impossible for the attacker to impersonate the victim to the legitimate server, even if the attacker can still impersonate the legitimate server. This is still a problem in cases where the attacker's aim is to obtain sensitive data from the user, not from the server. Such an attack could happen, for example, when a user promptly submits sensitive data to the server, without being able to recognize the impersonation attack (e.g., by visually noticing discrepancy in the authenticated environment supplied by the attacker and the personalized environment of the legitimate server). So, in theory at least, TLS CCA can resist even a powerful attack where the legitimate server is successfully impersonated[16]#section-7.4.4.

## 3.2 Renegotiation and Resumption

The renegotiation process allows the server to send the client a "HelloRequest" message, requesting a new TLS handshake from the client. During this renegotiation process all messages transmitted for the new handshake are cryptographically secured by the cipher suite that was already negotiated during the handshake. In practice it could make sense to renegotiate if the client wants to access some backend service and therefore authenticates itself to the server. Since the client will be granted access to additional resources increased security measures, for example another suite, have to be applied.

The client can send a "ClientHello" at anytime either to request a renegotiation which makes actually no sense for the client and should be disabled by the server, or to resume a TLS session. The resumption of a TLS session is performed by including the session identifier from

an older session in the "ClientHello" message. If the TLS server supports this resumption it will reply with a "ServerHello" that also includes this session identifier and a full TLS handshake can follow. Session resumption is not only important for a server saving a TLS operation and therefore lowering load. It is also important for a Client TLS Authentication, especially if the private key the client is authenticating itself with is stored on a smart card or any other external medium where every cryptographic operation can slow down the handshake[16]#section-3.4.

## 3.3 Cookies

A cookie aka magic cookie is a file, or some other form of database the browser can access, that typically contains data about web pages visited by the web browser when surfing the Internet. In general, cookies are used to store information linked to a website or domain locally on the computer for some time and to transmit it to the server upon request[3]. HTTP is a stateless protocol, so web pages are independent of the web server. A web application that interacts with the user through multiple page views must work with tricks to identify the participant across multiple accesses. For this purpose, a unique session identifier can be stored in a cookie from the server, in order to recognize exactly this client in further calls. For example onlineshops can use cookies to collect goods in virtual shopping carts. The customer can place articles in the shopping basket and continue to look around the website, then buy the articles together. The identification of the goods basket or session of the user is stored in the cookie, the article IDs are assigned to this shopping basket or to the session of the user on the web server. This information is evaluated by the server at the time of the order. Cookies also allow the user to individually customize the website, e.g. the language and font size or design of the website in general. Cookies are also necessary to authenticate visitors, with them a so-called "session ID" or "bearer token" is stored. Although the user theoretically has full control over his cookies, he is confronted with the problem that, in a short time, a lot of cookies are created so manually administrating them is not feasible.

But by using cookies for authentication there is a serious security risk of a man-in-the-middle attack. An HTTP cookie is sent with every HTTP request, even after the user has been authenticated, in order to ensure that the HTTP request is from a logged in user. Man-in-the-middle attack should be a thing of the past. By eliminating their attack surface by removing simple password authentication and cookies this could be achieved. But if cookies are still used, for example if the TPM of or smart card cannot handle the amount of requests, the lifetime of these cookies will be drastically reduced in comparison to those handed out by traditional password authentication. Since the renewal of the cookie needs no human interaction this could happen multiple times per minute. The pattern used for this kind of authentication is called "bearer tokens" A bearer of a token is granted access, regardless of the channel over which the token is presented, or who presented it[6].

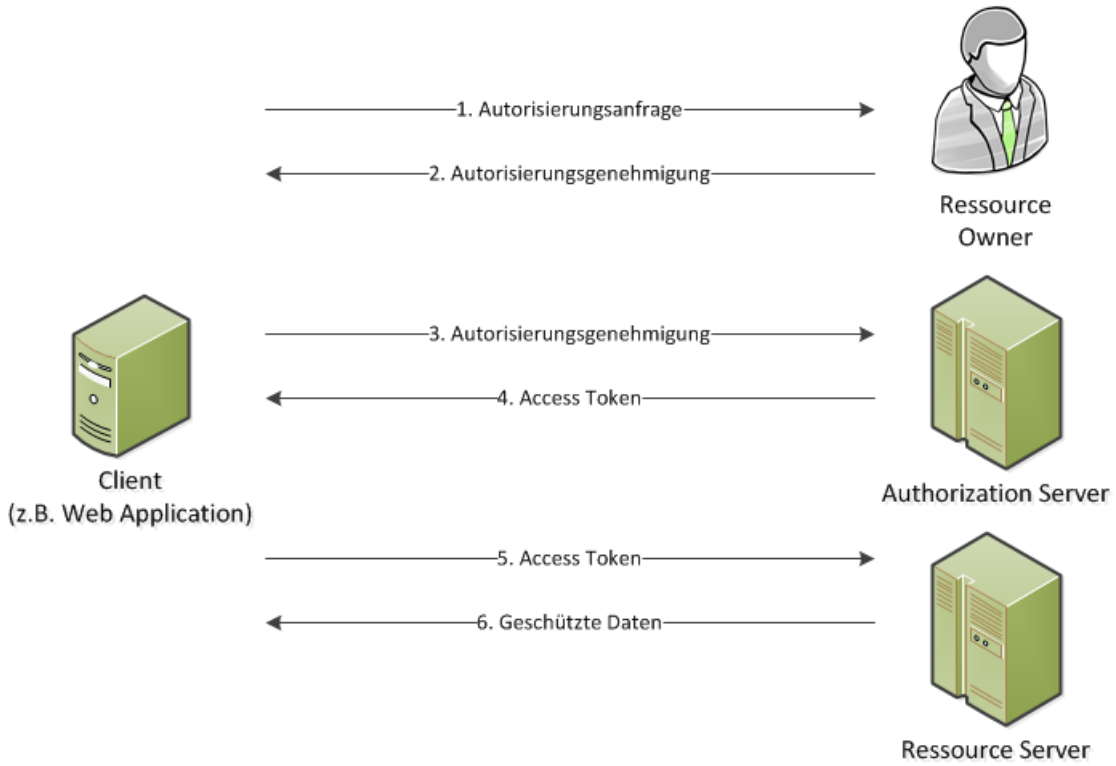


Figure 3: OAuth 2.0 Protocol Flow

## 3.4 OAuth

This Bearer Token concept can be applied to Single Sign On (SSO) as shown by the OAuth project. There are four roles in OAuth 2.0[9]:
Resource Owner: An entity that can grant access to a protected resource if the resource owner is a person, it is called a user.
Resource Server: The server on which the Protected Resources (PR) are located. It is possible to access it based on access tokens. These tokens represent the delegated authorization of the resource owner.
Client: An application that wants to access Protected Resources using the resource owner. The client can be executed either by a server (web application), desktop PC, mobile device or any other device.
Authorization Server: The server authenticates the resource owner and makes access tokens for the scope allowed by the resource owner.

# 4 Alternative Approaches

## 4.1 Persona

In the year 2011 the Mozilla Corporation published the BrowserID protocol[12]. In 2012 at the end of the beta phase it was re-branded as "Persona". From this change on Persona was an option for authentication on virtually every Mozilla Web page. The decentralization of the protocol is the major peculiarity of Persona if compared with other single sign-on solutions. Basically Persona consists of three actors:

- The User
- The Service Provider (SP)
- The Identity Provider (IdP)

Persona is entirely based on the email address of the user as the key for identification of the user. The user wants to use the website of an SP and therefore has to prove his identity to the Identity Provider. The Service Provider (SP) or Relying Party (RP) is the participant who wants to offer his services to the users. The advantage for the SP is that it solves the sensitive parts of the user authentication for him. These sensitive parts include checking, storing and manipulating the mail addresses as well as the passwords. The user then identifies itself with his "Identity Certificate" which is a public key certificate that is sent automatically by the browser[17].

The reference implementation of BrowserID was also hosted by Mozilla and was called "Mozilla Persona". Due to a low reception of this service it was permanently shut down in 2016[4].

A big disadvantage is that the IdP is a single point of failure. If the IdP is faulty, there is a risk that all SPs will be affected by a successful attack and unauthorized access to resources. Another criticism is that IdP could endanger the privacy of their users, by tracking their browser habits. As all single sign on systems it only works as long as the IdP is accessible.

## 4.2 Origin-Bound Certificates

In the year 2012 Google published a paper with the title "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web". One of the goals was to fix disadvantages described earlier for BrowserID. Origin-Bound Certificates (OBC) was later renamed to "Token Bind" as it will be called in this paper from here on. The advantages of Token Bind are

that it is in comparison to Persona a rather simple TLS extension. But it is most important that it allows clients to establish strong authenticated channels with servers and letting it connect with existing authentication tokens like HTTP cookies. The lack of acceptance by the users was one of the key issues of Mozilla Persona. Therefore the usability in concern of already existing infrastructure is one of the most important factors to consider if developing a new certificate based authentication mechanism. These already existing channels can not only be capt but rather remain unchanged. Token Bind will strengthen the authentication against various types of attacks discussed earlier. Since it was developed by Google it was first usable in Google's own browser "Google Chrome" but got adopted by all major browsers by now.

Within the TLS client authentication browsers apply self-signed client certificates, that are generated as necessary by the browser on-the-fly. These certificates do not contain any user-identifying information and simply serve as a foundation for establishing an authenticated channel, which can be re-established in the future.
For each website the browser connects to, it generates an individual certificate, rendering any crosssite user tracking impossible. Therefore these certificates are called Origin-Bound Certificates. These characteristics permit a complete decoupling of certificate generation and use from the user interface. Thanks to TLS-OBC client authentication the existing web user experience stays the same; the modifications are only under the hood. In order to establish a TLS connection with an origin the browser will consistently apply the same client certificate. Therefore the website is able to „bind“ authentication tokens (e.g., HTTP cookies) to the OBC and create an authenticated channel. To do so, it merely needs recording which client certificate should be applied at the TLS layer when the token (i.e., cookie) is submitted back to the server. User identity is established at this layer in the cookie (not in the TLS certificate), as it is generally done on the web today. Thanks to TLS-OBC‘s channel-binding mechanism it is not possible to use stolen tokens (e.g., cookies) over other TLS channels, which makes them useless to token thieves, thus solving a current problem on the web.
Basically, an Origin-Bound Certificate is a selfsigned certificate used by browsers in order to carry out TLS Client Authentication. OBCs differ from normal certificates used in TLS Client Authentication insofar as no interaction with the user is required. This is due to the server always generating and storing just one certificate per origin, which allows the browser to know which certificate it has to use without any input by the user[6].

If a browser does not have an OBC yet it can create one on-the-fly. This newly generated OBC will only be transmitted to the origin it was created for and does not contain any user identifying information as name or email address. It is to prove, in a cryptographically secure manner, that all requests of this session originate from the same client, in the same way TLS from the server side works without an Extended Validation Certificate. This trust is continuous for this client machine, as long as the certificate is valid. From a user point of view, there is no user interface or similar for creating or using Origin-Bound Certificate. Same as for cookies there is no interface to set these manually. The cookie gets set by the browser and is also transmitted by the browser when visiting a specific domain. There is only an interface to delete

cookies, such an interface is as well needed for Origin-Bound Certificates to guarantee privacy. In private browsing no such certificates should be stored either, same as for cookies[6].

The benefits and features of token bind summarized:

- Clients use a different certificate for each origin. Unless the origins collaborate, one origin cannot discover which certificate is used for another.
- Different browser profiles use different Origin-Bound Certificates for the same origin.
- In incognito or private browsing mode, the Origin-Bound Certificates used during the browsing session get destroyed when the user closes the incognito or private browsing session.
- In the same way that browsers provide a UI to inspect and clean out cookies, there should be a UI that allows users to reset their Origin-Bound Certificates

# 5 Results

Dogtag and Let's encrypt both are ways to automate certificate management. But Let's encrypt is not suitable for client certificates, whereas Dogtag is only suitable inside an organization or organization-like institution. OAuth bears several issues concerning man-in-the-middle attacks with their bearer tokens approach, and several privacy issues relating to the centralized IdP infrastructure. Mozilla Persona tried to solve these problems but had multiple shortcoming itself. Even though there are secondary IdP, full security can only be guaranteed under perfect conditions, the "fallback" scenario still has the same privacy issues as OAuth. Most importantly the reception of Persona was very bad since it did not integrate into current work flows as well as it should have. A new approach of combining the strength of these technologies is needed to enable a wider use of certificate based two-factor authentication. A local registration authority is a great approach to identify a person and sign an initial certificate or hand out an initial smart card. But a way to renew, revoke and sign additional certificates in a for the user transparent manner was needed. This approach would use a certificate based two-factor authentication without a smart card, since a smart card based system would most probably not get accepted on the Internet as most users lack readers for these.

| | Dogtag Certificate System | Mozilla Persona | Token Bind |
|---|---|---|---|
| Hardware token | Yes | No | No |
| Active Development | Yes | No | Yes |
| Automated enrollment | No | No | Yes |
| Privacy guaranteed | No | No | Yes |

Token Bind addresses all these issues and is currently in IETF draft status[13]. Unfortunately there is little to no reception of token bind in public. The only user as of now seems to be Google, the creator of the protocol, itself. During the research for this paper it became clear that there are two main reasons for this. First there is no precompiled version of any token bind based implementation. In this paper the nginx webserver was chosen as a token bind ssl terminator. This nginx would be placed in front of an application server, and only the results of the client certificate check would be passed to the application for further authentication processing. The source code doesn't even work with stock openssl, a patch needs to be applied to openssl to make it work. Second there is no real documentation as about how to write an application to use this type of authentication.

On https://github.com/b3n4kh/nginx_token a Dockerfile to create a usable version of this was created for this paper. On https://hub.docker.com/r/b3n4kh/nginx_token/ docker containers are automatically built for this repository. Why docker? Since openssl and nginx are usually core components of a Linux web server these two programs exist in some form or another. To use token bind and not collide with any preinstalled software docker was a reasonable choice.

To configure token bind in nginx these new settings are introduced.

```
## token_binding

syntax: token_binding on|off
default: off
context: http, server
Enables negotiation and verification of the Token Binding protocol.

# Token Binding ID variables (described below) are going to be available when client
    successfully negotiates Token Binding.

## token_binding_cookie

syntax: token_binding_cookie <cookie>|all|none
default: none
context: http, server
Binds selected <cookie> (or all) to clients HTTPS channel and verifies that properly
    bound cookies are received from the client.
```

```
# Because Token Binding ID can be established only over HTTPS, Secure attribute is
    going to be added to cookies bound this way. Also, such cookies are going to be
    removed from HTTP requests and responses.

## token_binding_secret

syntax: token_binding_secret <secret>
default: none
context: http, server
Secret used to bind cookies using token_binding_cookie directive.
```

The Dockerfile necessary to build the docker container can be found in Appendix A.

# Bibliography


[1] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, December 1999.

[2] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. Automatic Certificate Management Environment (ACME). Internet-Draft draft-ietf-acme-acme-06, Internet Engineering Task Force, March 2017. Work in Progress.

[3] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011.

[4] callahad. Transitioning persona to community ownership. http://identity.mozilla.com/post/78873831485/transitioning-persona-to-community-ownership, 2016. 2017-05-09.

[5] National Cyber Security Center. Password guidance: Simplifying your approach. https://www.ncsc.gov.uk/guidance/password-guidance-simplifying-your-approach, 2017. 2017-05-09.

[6] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *21st USENIX Security Symposium*, pages 317–332, 2012.

[7] estonia. Estonia id card. http://www.id.ee/, 2017. 2017-05-17.

[8] Peter Gutmann. Simple Certificate Enrolment Protocol. Internet-Draft draft-gutmann-scep-05, Internet Engineering Task Force, November 2016. Work in Progress.

[9] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. http://www.rfc-editor.org/rfc/rfc6749.txt.

[10] letsencrypt. Let's encrypt. https://letsencrypt.org, 2017. 2017-05-09.

[11] markm. Yarrow algorithm. https://svnweb.freebsd.org/base?view=revision&revision=284959, 2017. 2017-05-17.

[12] millsd. Introducing browserid: A better way to sign in. http://identity.mozilla.com/post/7616727542/introducing-browserid-a-better-way-to-sign-in, 2012. 2017-05-09.

[13] Andrey Popov, Magnus Nystrom, Dirk Balfanz, Adam Langley, and Jeff Hodges. The token binding protocol version 1.0. Internet-Draft draft-ietf-tokbind-protocol-14, IETF Secretariat, April 2017. http://www.ietf.org/internet-drafts/draft-ietf-tokbind-protocol-14.txt.

[14] Max Pritikin, Andrew Nourse, and J Vilhuber. Simple Certificate Enrollment Protocol. Internet-Draft draft-nourse-scep-23, Internet Engineering Task Force, September 2011. Work in Progress.

[15] schnouki. Let's encrypt and client certificates. https://schnouki.net/posts/2015/11/25/lets-encrypt-and-client-certificates/, 2017. 2017-05-09.

[16] Yaron Sheffer, Peter Saint-Andre, and Ralph Holz. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7525, May 2015.

[17] Henning Thiel. Praktische sicherheitsanalyse des mozilla single sign-on protokolls browserid. *Bochum, Germany*, 2014.

# List of Figures

# Abbreviations

**CA** Certificate Authority

**ACME** Automated Certificate Management Environment

**OCSP** Online Certificate Status Protocol

**CRL** Certificate Revocation Lists

**SCEP** Simple Certificate Enrollment Protocol

**PKI** Public key infrastructure

**SSO** Single Sign On

**LRA** Local Registration Authority

**DCS** Dogtag Certificate System

**IaC** Infrastructure as Code

**IaaS** Infrastructure as a Service

**CSPRNG** Cryptographically Secure Pseudo-Random Number Generator

# A Appendix

```
FROM debian

MAINTAINER Benjamin Akhras "b@akhras.at"

ENV NGINX_VERSION 1.13.0

COPY ssl.patch /usr/ssl.patch
COPY ngx_token_binding /usr/ngx_token_binding

RUN GPG_KEYS=B0F4253373F8F6F510D42178520A9993A1C052F8 \
  && CONFIG="\
␣␣--prefix=/etc/nginx␣\
␣␣--sbin-path=/usr/sbin/nginx␣\
␣␣--modules-path=/usr/lib/nginx/modules␣\
␣␣--conf-path=/etc/nginx/nginx.conf␣\
␣␣--error-log-path=/var/log/nginx/error.log␣\
␣␣--http-log-path=/var/log/nginx/access.log␣\
␣␣--pid-path=/var/run/nginx.pid␣\
␣␣--lock-path=/var/run/nginx.lock␣\
␣␣--http-client-body-temp-path=/var/cache/nginx/client_temp␣\
␣␣--http-proxy-temp-path=/var/cache/nginx/proxy_temp␣\
␣␣--http-fastcgi-temp-path=/var/cache/nginx/fastcgi_temp␣\
␣␣--http-uwsgi-temp-path=/var/cache/nginx/uwsgi_temp␣\
␣␣--http-scgi-temp-path=/var/cache/nginx/scgi_temp␣\
␣␣--user=nginx␣\
␣␣--group=nginx␣\
␣␣--with-http_ssl_module␣\
␣␣␣␣␣␣␣␣--with-openssl=/usr/src/openssl-1.1.0e␣␣␣␣\
␣␣␣␣␣␣␣␣--add-module=/usr/src/ngx_token_binding␣␣␣␣\
␣␣--with-http_realip_module␣\
␣␣--with-http_addition_module␣\
␣␣--with-http_sub_module␣\
␣␣--with-http_dav_module␣\
␣␣--with-http_flv_module␣\
␣␣--with-http_mp4_module␣\
␣␣--with-http_gunzip_module␣\
␣␣--with-http_gzip_static_module␣\
␣␣--with-http_random_index_module␣\
␣␣--with-http_secure_link_module␣\
␣␣--with-http_stub_status_module␣\
␣␣--with-http_auth_request_module␣\
```

```
␣␣--with-threads␣\
␣␣--with-stream␣\
␣␣--with-stream_ssl_module␣\
␣␣--with-stream_ssl_preread_module␣\
␣␣--with-stream_realip_module␣\
␣␣--with-http_slice_module␣\
␣␣--with-compat␣\
␣␣--with-file-aio␣\
␣␣--with-http_v2_module␣\
␣␣" \
  && addgroup --system nginx \
  && adduser --disabled-password --system --home /var/cache/nginx --shell /sbin/
      nologin --ingroup nginx nginx \
  && apt-get update \
    && apt-get install -y build-essential zlib1g-dev libpcre3 libpcre3-dev libbz2-
        dev libssl-dev curl gnupg libperl-dev libc6-dev gcc \
  make \
  libxslt1-dev \
  && curl -fSL http://nginx.org/download/nginx-$NGINX_VERSION.tar.gz -o nginx.tar.gz
       \
  && curl -fSL http://nginx.org/download/nginx-$NGINX_VERSION.tar.gz.asc  -o nginx.
      tar.gz.asc \
  && curl -fSL https://www.openssl.org/source/openssl-1.1.0e.tar.gz -o openssl.tar.
      gz \
  && export GNUPGHOME="$(mktemp␣-d)" \
  && found=''; \
  for server in \
  ha.pool.sks-keyservers.net \
  hkp://keyserver.ubuntu.com:80 \
  hkp://p80.pool.sks-keyservers.net:80 \
  pgp.mit.edu \
  ; do \
    echo "Fetching␣GPG␣key␣$GPG_KEYS␣from␣$server"; \
    gpg --keyserver "$server" --keyserver-options timeout=10 --recv-keys "$GPG_KEYS"
        && found=yes && break; \
  done; \
  test -z "$found" && echo >&2 "error:␣failed␣to␣fetch␣GPG␣key␣$GPG_KEYS" && exit 1;
       \
  gpg --batch --verify nginx.tar.gz.asc nginx.tar.gz \
  && rm -r "$GNUPGHOME" nginx.tar.gz.asc \
  && mkdir -p /usr/src \
  && mv /usr/ngx_token_binding /usr/src/ngx_token_binding \
  && tar -zxC /usr/src -f openssl.tar.gz \
  && patch /usr/src/openssl-1.1.0e/ssl/t1_lib.c < /usr/ssl.patch \
  && tar -zxC /usr/src -f nginx.tar.gz \
  && rm openssl.tar.gz \
  && rm nginx.tar.gz \
  && cd /usr/src/nginx-$NGINX_VERSION \
  && ./configure $CONFIG --with-debug \
  && make -j$(getconf _NPROCESSORS_ONLN) \
```

```
  && mv objs/nginx objs/nginx-debug \
  && ./configure $CONFIG \
  && make -j$(getconf _NPROCESSORS_ONLN) \
  && make install \
  && rm -rf /etc/nginx/html/ \
  && mkdir /etc/nginx/conf.d/ \
  && mkdir -p /usr/share/nginx/html/ \
  && install -m644 html/index.html /usr/share/nginx/html/ \
  && install -m644 html/50x.html /usr/share/nginx/html/ \
  && install -m755 objs/nginx-debug /usr/sbin/nginx-debug \
  && ln -s ../../usr/lib/nginx/modules /etc/nginx/modules \
  && rm -rf /usr/src/nginx-$NGINX_VERSION \
  # forward request and error logs to docker log collector
  && ln -sf /dev/stdout /var/log/nginx/access.log \
  && ln -sf /dev/stderr /var/log/nginx/error.log

COPY nginx.conf /etc/nginx/nginx.conf
COPY nginx.vh.default.conf /etc/nginx/conf.d/default.conf

EXPOSE 80
EXPOSE 443

STOPSIGNAL SIGQUIT

CMD ["nginx", "-g", "daemon␣off;"]
```