Masaryk University
Faculty of Informatics

Security and cryptography in GO

Bachelor's Thesis

Lenka Svetlovská

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Security and cryptography in GO

Bachelor's Thesis

Lenka Svetlovská

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Lenka Svetlovská

Advisor: RNDr. Andriy Stetsko, Ph.D.

Acknowledgement

This is the acknowledgement for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

keyword1, keyword2, …

Contents

List of Tables

List of Figures

1 Introduction

2 Basic Terms

In this chapter, I will define the basic terms which are necessary to an understanding of the text presented in the following chapters. First I will describe the structure of certificates, standard X.509, certificate authority and signing process. I will explain self-signed certificate, PEM and DER formats, and the process of certificate exchange. Finally, I will describe protocols SSL/TLS and TLS_PSK used in the application.

2.1 Certificate

Certificate is digitally signed data structure whose foundation includes a public key of certificate owner. There are some standards of the certificate structure (X.509, EDI, WAP, etc.). The Internet is based on standard X.509 version 3, which was issued by the ITU [1].

2.1.1 Standard X.509

Standard X.509 was originally designed as an authentication framework for X.500 directories [2]. They have a hierarchical structure and the individual attributes can be assigned to individual computers of companies or individual printers. Certificate X.509, see Figure 2.1.

2.1.2 Certificate Authority

The Certification Authority (CA) is an independent third party that issues certificates.

In other words, the CA is an institution that inspects the certificate request and issues whose validity is verifiable using the responsive public key. The CA was created in 1998 in order to ensure secure encryption through a certified public key and guarantee the validity of the digital signature. The CA also ensures their validity revocation. The CA is truly credible, must be impartial and, if it is possible, should be subordinate to any higher CA. Subordination means that it identifies using certificates signed by the higher certificate authority [1].

Root CA, certification authorities at the highest level, uses the certificate signed by themselves (self-signed certificate).

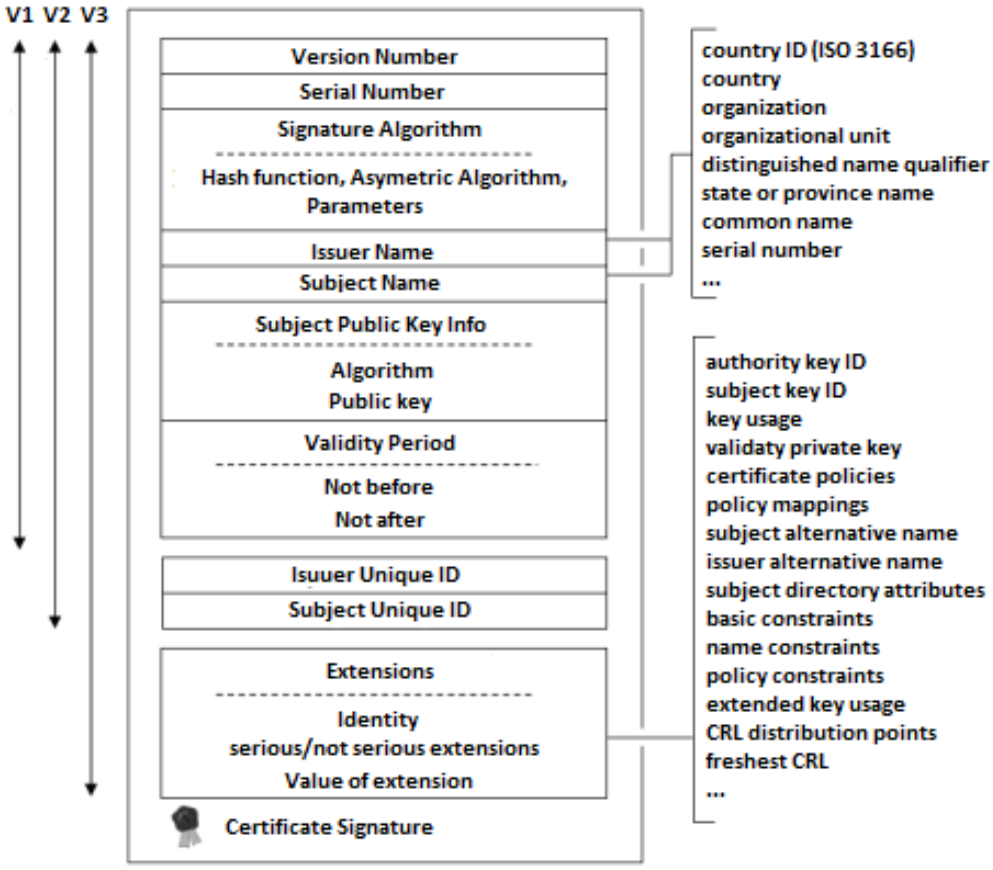


Figure 2.1: Structure X.509 Certificate

2.1.3 Self-signed Certificate

The self-signed certificate is a certificate which the applicant gives by itself. The self-signed certificate has the same data structure as a certificate issues by CA. This certificate is recognized according to identical item Subject and item Issuer.

Creating self-signed certificate is not easy. To create certificate request we need to fill same items, which we do not know. For example, serial number, certificate validity, etc.

As proof of the private key possession by the user, the self-signed certificate uses a digital signature certificate which was made by the private key belonging to the public key in the certificate. Verification of

the signature is carried out through a public key in the actual certificate [1].

Self-signed certificate is used by software internally. While the applicant generates CSR to be issued by the CA, the public key must be kept by the applicants. In this case, the public key is often maintained like a self-signed certificate. Subsequently it is rewritten by the certificate.

2.1.4 Certificate Signing Request

The certificate applicant can apply CA to sign certificate in two ways: submit a data structure called a certificate request (CSR) or not so [1].

The most common format is PKCS#10. The CSR should include:

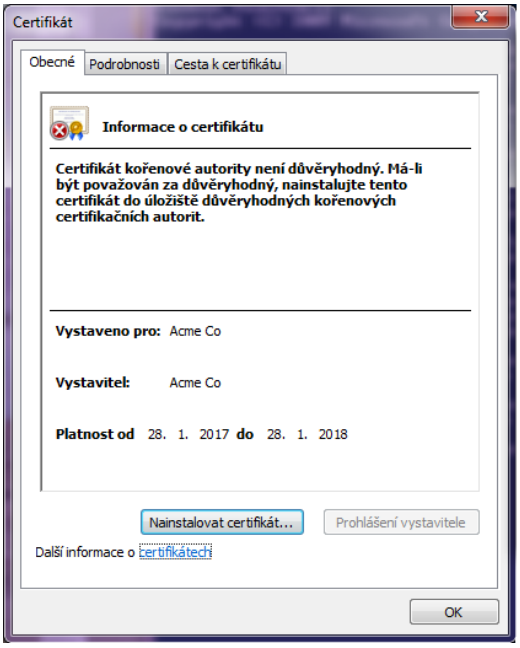
- Applicant ID
- The public key
- Evidence of the possession of the private key
- Other information that the user wishes to insert certificate
- May contain evidence of the generation of paired data
- Data necessary for billing (in case issuance of certificates is paid)
- Passwords for communication with CA:
  - One-time password for issuing the certificate
  - One-time password for certificate revocation
  - Permanent password for personal (non-electronic) communication between user and CA
  - Phrase (in case losing all passwords)

The certificate authority must fill individual items of the certificate before it signs the result digitally. The CA takes CSR, checks completed items and fills the other. The important item is validity period - the time while the certificate is valid. It is shown in two items - not before and not after. The CA can add extensions, such as Subject Alternative Name, information about CA, etc. (see figure 2.1).

2.1.5 Formats PEM and DER

The PEM (Privacy Enhanced Mail) is a Base64 encoded DER certificate. It is designed to be safe for inclusion in ASCII or even rich-text documents. This means that we can simple copy and paste the content

of a pem file to another document and back. Following is a sample PEM file containing a private key and a certificate 2.2.




```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAz2DH3nJePLkL84mVCAgtO8I
MI5cI3e2JKRr/cDpYa/sm7I5WRxeOqtNIN+VU6m
A .... MANY LINES LIKE THAT ..........
sXeRvIEx3lQnC1HFKuJN/8dDcD/tQi2URl4lgpV
8PlZo5bit+nzU2fvJIKOoGLF1NyX18ImApWL8kj
kgL3iXR5t9lbHRsD85zzy3OYBKZBXE/T5cbW4uW
-----END RSA PRIVATE KEY-----

-----BEGIN CERTIFICATE-----
MIIC9jCCAd6gAwIBAgIRAOzV53Pr8ldSJfEtCYN
ODAwWjATMREwDwYDVQQKDAhMZW5rYSA8MzCCASI
A .... MANY LINES LIKE THAT ..........
vZk8K04FuGSxpY7mtPLfeHni9RpxQAoU187/DwU
vpJyk/+8o7e0U3Lil/O7iZjXr3jDSYENc/fj0Hx
r+uQEnxgWgeGyhaB6f15zo6mGY0r+Czkov2gdJH
-----END CERTIFICATE-----
```

Figure 2.2: A sample example of certificate view

A certificate or key must start with header and the number of dashes is meaningful, and must be correct. A single PEM file can contain a number of certificates and a key, for example, a single file with public certificate, intermediate certificate, root certificate or private key [4].

The DER (Distinguished Encoding Rules) is a binary form of ASCII PEM format certificate. All types of certificates and private keys can be encoded in DER format, its information is stored in the binary DER for ASN.1 and applications providing RSA, SSL and TLS should handle DER encoding to read in the information [5].

2.2 Protocols SSL and TLS

Protocols SSL and TLS are used for secure communications between client and server. They create a framework for the use of encryption and hash functions.

SSL was developed by Netscape Communications which published three versions. The first version was only a test, the second has been used in practice. However, it still contained security vulnerabilities, the most important was susceptibility to attack *man in the middle*. The third version was introduced in 1996 and its specification could be found in the document *The SSL Protocol Version 3.0* [6]. Of this version was created TLS protocol which is currently the most widespread and supported [7]. There are three versions which have only minimal differences.

Protocol SSL/TLS provides authentication of the two communicating parties by using asymmetric encryption, message integrity by using MAC and confidentiality by encrypting all communications by selected symmetric cipher.

Protocol SSL/TLS is located between the application and the transport layer reference ISO/OSI model and consists of two main parts. They are *The Record Layer Protocol* (RLP) and *Handshake Protocol* (HP).

Record Layer Protocol processes application data, performs fragmentation, compression and data encryption. On the other hand, it decrypts the data again and verifies the checksums. RLP protocol does not care about the type of encryption algorithm or encryption key setting. This information is from HP.

Handshake Protocol is activated immediately after establishment the connection and provides identification of communicating parties, provision of cryptographic algorithms, compression algorithms and other attributes. Then it creates *a master secret* from which are derived encryption keys, initiation vectors and the MAC. The process of the protocol is:

1. The client wants to connect to the server and sends *ClientHello*, which contains the highest number of version supported by SSL/TLS, the number of session (it is empty if it is a new session), the list of supported ciphers and compression methods and a random number.

2. The client waits for a response in the form of a report *ServerHello*, which will contain the highest number of versions of SSL/TLS, which is supported by server and client. It will also contain encryption and compression method, which are selected from the list received in step one, a random number and its public key certificate (the server can also request authentication client).

3. The client verifies the server certificate, if all ciphers are satisfied. Next, he sends a request to server to exchange keys. At the end the server and client share a common 48 bits *premaster secret*. *The master secret* is derived from it. Then of the master secret and random numbers *ClientHello ServerHello* are derived two session keys to encrypt messages and two MAC initiation vectors for use symmetric cipher in CBC mode.

4. The client sends a confirmation selected ciphers. From this moment, the communication has been encrypted and the client sends a message that ends with this phase. In case the server required the client authentication, it has been carried out in this step. Finally, the server sends a confirmation used ciphers and message about the completion of this phase, thereby HP ends.

The used algorithms:
- key exchange: RSA, Diffie-Hellman, ECDH;
- stream symmetric ciphers: RC4 with key length of 40-120 bits;
- block symmetric ciphers: DES, DES40, 3DES, IDEA;
- hashing algorithms: MD5, SHA.

TLS uses public key certificates for authentication. To establish a TLS connection are used symmetric keys (later called pre-shared keys or PSKs), shared in advance among the communicating parties [8].

2.2.1 PSK Key Exchange Algorithm

The cipher suites with the PSK key exchange algorithm use only symmetric key algorithms and are particularly suitable for performance-constrained environments where both ends of the connection can be controlled. The cipher suites are intended for a rather limited set of applications, usually involving only a very small number of clients and servers.

1. The client indicates to use pre-shared key authentication by including one or more PSK cipher suites in the *ClientHello* message.

2. If the TLS server also wants to use pre-shared keys, it selects one of the PSK cipher suites and places it in *the ServerHello* message. The server can provide a "PSK identity hint" in the *ServerKeyExchange* message. If no hint is provided, the *ServerKeyExchange* message is omitted.

3. The client indicates which key to use by including a "PSK identity" in the *ClientKeyExchange* message.

4. The *Certificate* and *CertificateRequest* payloads are omitted from the response.

5. The TLS handshake is authenticated using the *Finished* messages.

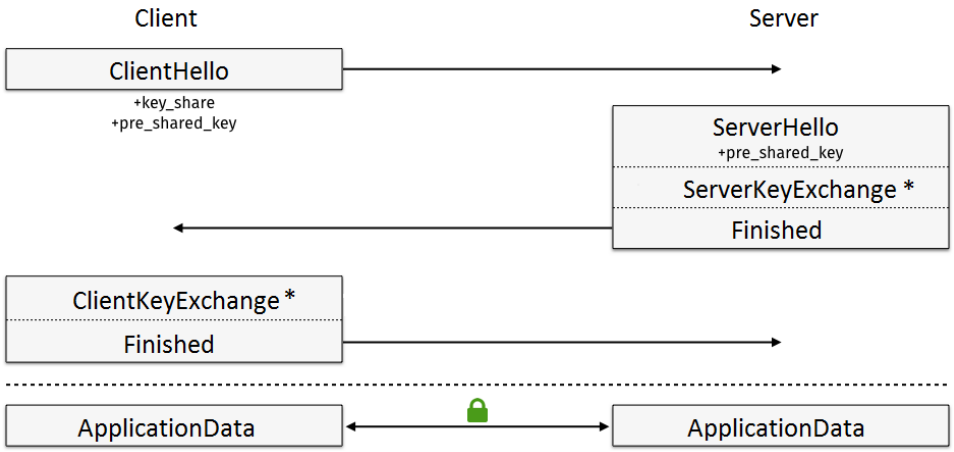


Figure 2.3: TLS_PSK Key Exchange

If the server does not recognize the PSK identity, it may respond with an *unknown_psk_identity* alert message. Alternatively, if the server wishes to hide the fact that the PSK identity was not known, it may continue the protocol as if the PSK identity existed but the key was incorrect: that is, respond with *a decrypt_error alert* [8].

3 Language Go

In this chapter I will explain the main characters of language Go and show how to write the code. Next I will describe Go libraries called packages and the tool Cgo.

3.1 Introduction

Go is a programming language developed at *Google* in year 2007 and announced in November 2009. Many companies have started using Go because of its performance, simplicity, ease of use and powerful tooling. Go programming language is a statically-typed language with advanced features and a clean syntax [9]. It combines the performance and security benefits associated with using a compiled language like *C++* with the speed of a dynamic language like *Python*. It provides:

- garbage collector - memory is cleaned up automatically when nothing refers to it anymore,
- fast compilation time - through effective work with addictions individual parts of the program and simple grammar,
- light-weight processes (via go-routines), channels,
- a rich standard library,
- easy testing - incorporated directly into the core language,
- one of the best documentation - a clear and full of examples.

Go excluded some features intentionally to keep language simple and concise. There is no support for type inheritance, method or operator overloading, circular dependencies among packages, pointer arithmetic, assertions nor for generic programming.

3.2 How to Start

3.2.1 Installation

Golang, the go official website, provides the installer of Go to download free for Windows, Linux, Mac OS X and FreeBSD. To confirm everything is working, open a terminal and type the following:

```
1 go version
```

Upgrading from an older version of Go must be precessed by removing the existing version.

3.2.2 Enviroment

The Go toolset uses an environment variable called GOPATH to find Go source code.

Set up your workplace, for example GOPATH=C:\Projects\Go\.

3.2.3 First Program

Traditionally, the first program in any programming language is called a hello program — a program that simply outputs "Hello World!" to terminal.

Open text editor, create file *hello.go*, and enter the following:

```
1 package main
2 import "fmt"
3 // this is a comment
4 func main() {
5   fmt.Println("Hello World!")
6 }
```

Make sure the file is identical to what is shown here and save it.

3.2.4 Compilation

Go is a compiled language, and like many languages, it makes heavy use of the command line. Open up a new terminal, set up directory where you saved the file *hello.go* and type in the following:

```
1 go run hello.go
```

You should see Hello World! displayed in your terminal.

3.2.5 Arguments

Command-line arguments are a common way to parameterize execution of programs. First must be used go build command to create a binary program.

```
1 go build cmd-arguments.go
2 ./cmd-arguments a b c
```

In code, *os.Args* provides access to raw command-line arguments. The first value is the path to the program, and os.Args[1:] holds the arguments to the program.

```
argsWithProg := os.Args
argsWithoutProg := os.Args[1:]
arg := os.Args[3]
```

Do not forget to import package *os*.

3.3 Packages

In Go, source files are organized into system directories called packages. To develop software applications, writing maintainable and reusable pieces of code is very important. Go provides the modularity and code reusability through its package system. Go encourages programmers to write small pieces of software components through packages, and compose their applications with these small packages.

The packages from the standard library are available at the "pkg" subdirectory of the GOROOT directory. When we install Go, an environment variable GOROOT will be automatically added to our system for specifying the Go installer directory. The Go developer community is very enthusiastic for developing third-party Go packages. When you develop Go applications, you can affect these third-party Go packages [11].

When programmers develop executable programs, they will use the package "main" for making the package as an executable program. The package "main" tells the Go compiler that the package should compile as an executable program instead of a shared library. When programmers build shared libraries, they will not have any main package and main function in the package.

To download third-party Go packages is used command:

```
go get example/exampleLib
```

After installing the exampleLib, put the import statement in programs for reusing the code, as shown below:

```
package db
import (
  "example/exampleLib"
  "example/exampleLib/sub"
```

```
)
func init {
  // initialization code here
}
```

Third-party Go packages are installed by using this command:

```
go install
```

The go install command will build the package “sub” which will be available at the pkg subdirectory of GOPATH.

```
package main
import (
  "fmt"
  "example/exampleLib/sub"
)
func main() {
    sub.Add("dr","Dart")
    fmt.Println(sub.Get("dr"))
    // code here
}
```

3.4 C? Go? Cgo!

Cgo allows Go to interoperate with C code. Go source file imports "C" and it is immediately preceded by comment lines which may contain any C code, including function and variable declarations and definitions. It is necessary to say that there must be no blank lines in between the cgo comment and the import statement.

```
package main
/*
#include <stdio.h>

void hello() {
  printf("Hello World!\n");
}
*/
import "C"

func main() {
  C.hello()
}
```

Cgo allows including own C libraries. The header file *foo.h*, the library implementation *foo.c* and go wrapper *foo.go* are located in the same package $*GOPATH**src**foo*.

```
#cgo CFLAGS: -I/{$GOPATH}/src/foo
#cgo LDFLAGS: -L/{$GOPATH}/src/foo -lfoo
```

Go wrapper contains the explicit paths in the cgo flags because of an environment issue where libfoo wasn't found by the Go linker.

```
go install foo
```

By installing the package everything is prepared for using in Go. Package is called in Go code through *import"foo"*.

In order to use Cgo on Windows, it is firstly needed to install a gcc compiler and have *gcc.exe* in PATH environment variable before compiling with Cgo will work.

4 Go Packages Analysis

This chapter contains the overview, evaluate and compare existing cryptographic libraries based on their measures and expansion measures important for implementation.

4.1 Cryptographic functions

It was necessary to examine features of language Go related to the safety and cryptography, analyze libraries which Go contains and also search for available third-party libraries.

Table 4.1 shows analyzed Go package Crypto created by *Golang* and other Go packages created by third-parties. Individually items mean:

- Name - founder nickname (almost all packages are searched on *https://github.com/* where is not published real name of founder or contributors) and name of package.
- Cryptographic functions - support cryptographic functions, such as symmetric encryption, asymmetric encryption and hash functions (all/sym/asym/hash/unknown).
- Start of project - a year of creating project with package.
- Is still opened - the important measure is if contributors still work on the project (Yes/No).
- Issue tracker system - support of solving bugs, feature requests, tracking todos, and more (Yes/No).
- Documentation accessibility - assessed on a scale of 0-5 points (5 is the highest) according to own experience to work with the package.
- Downloads - number of downloads. In case the package is saved on *https://github.com/*, the information about downloads presents value of Fork. Downloads of package Crypto created by Golang is unknown, this package is automatically installed with installing program Go.
- License - availability of licenses for free (Yes/Payment). The payment indicates the amount of payment per some period. This case did not occur.

Table 4.1: Table of cryptographic libraries in Go

Name	Crypto functions	Start of project	Is still opened	Issue tracker system	Doc. accessibility	Downloads	License
golang/crypto	all	2009	Y	Y	5	-	Y
ory-am/fosite	asym, hash	2015	Y	Y	4	39	Y
square/go-jose	asym, sym	2014	Y	Y	4	70	Y
Shopify/ejson	hash	2014	Y	Y	3	22	Y
mitchellh/go-mruby	asym	2014	Y	Y	4	19	Y
Sermo Digital/jose	all	2015	Y	Y	4	28	Y
mozilla/tls-observato	hash	2014	Y	Y	4	31	Y
square/certigo	hash	2016	Y	Y	3	12	Y
docker/libtrust	all	2014	N	Y	4	26	Y
dedis/crypto	sym, hash	2010	Y	Y	4	17	Y
dchest/blake2b	hash	2012	N	Y	4	6	Y
enceve/crypto	hash	2016	N	Y	4	1	Y

Continued on next page

Table 4.1 – continued from previous page

Name	Crypto functions	Start of project	Is still opened	Issue tracker system	Doc. accessibility	Downloads	License
go-libp2p-crypto	all	2015	Y	Y	4	5	Y
dchest/blake2s	hash	2012	N	N	3	3	Y
benburkert/openpgp	sym, hash	2012	N	N	3	0	Y
minio/sha256-simd	hash	2016	Y	Y	3	14	Y
avelino/awesome-go	unknown	2014	Y	Y	2	2287	Y
ethereum/go-ethereum	all	2013	Y	Y	4	1028	Y
chain/chain	hash	2015	Y	Y	4	113	Y
lightning network/lnd	sym, hash	2015	Y	Y	4	50	Y
square/go-jose	asym, hash	2014	Y	Y	4	70	Y
dgrijalva/jwt-go	asym, hash	2012	Y	Y	4	226	Y
golang/oauth2	asym, hash	2014	Y	Y	4	279	Y

The table 4.1 was created based on information available on following websites [20] [21] [22] [23]. The table does not contain packages which do not support any cryptographic functions. Also, the table does

not contain information from untrusted sources. During the analysis, I discovered sources which are not actual or supported [24].

I searched immediately using keywords as names of functions or words in descriptions of functions. I found a large amount of packages, but a few of them met the necessary requirements such as support of symmetric and asymmetric algorithms or hash operations.

4.2 Expansion measures

The next step of analysis was to select the packages which fulfill the requirements. The important parameters were the support of all cryptographic functions, the topicality of project, the approach to solving problems and no payment license. In the table 4.2, there are shown four libraries which are compared to the base of the other measures:

- Gen. crypto. key - possibilities for generation of cryptographic keys (Yes/No),
- Key size min. - if package could generate cryptographic key, it shows minimal key size in bytes (some packages do not provide this information),
- Key size max. - if package could generate cryptographic key, it shows maximal key size in bytes (some packages do not provide this information),
- X.509 Certificates - work with X.509 certificates (generation of self-signed certificates, support for certificate signing requests, support for different formats and extensions) (Yes/No),
- SSL/TLS - support of higher level protocols such as SSL/TLS (Yes/No),
- TLS_PSK - support of create communication with server using pre-shared key (Yes/No).

Based on observed data I decided to use Crypto package created by *Golang* for implementation.

Crypto supports generation cryptographic keys and their size is 2 to 8192 bytes, as suggests table 1 in [29]. The other important requirement is propped up higher level protocols such as SSL/TLS. Only the Crypto package contains subdirectory worked with TLS connection to the server.

Table 4.2: Filtered table of packages with expansion measures

Package	crypto	jose	go-libp2p-crypto	go-ethereum
Founder	Golang	SermoDigital	libp2p	ethereum
Crypto. functions	all	all	all	all
Start of project	2009	2015	2015	2013
Is still opened	Y	Y	Y	Y
Issue tracker system	Y	Y	Y	Y
Doc. access-ability	5	4	4	4
Downloads	-	28	5	1028
License	Y	Y	Y	Y
Gen. crypto. keys	Y	N	Y	N
Key size min	2	-	unknown	-
Key size max	8192	-	unknown	-
X.509 Certificates	Y	Y	Y	Y
SSL/TLS	Y	N	N	N
TLS_PSK [1]	N	N	N	N

[1] During analysis I found package *Raff/tls-psk* using TLS_PSK cipher suites, I describe it in subsection 5.2.1.

5 Implementation

The theoretical part of my Bachelor thesis was to implement console application to create keys and certificates which are signed by intern certificate authority. In this chapter, I will explain the assignment, the problems and the process of implementation.

5.1 Assignment

For functionality, there are needed several parameters, such as:

- the address of the server with the certificate authority,
- the port connection to the server,
- the pre-shared key to connect to server,
- path to the key store,
- IP address of subsystem,
- the domain name of subsystem and
- identity of a system for what the certificate is created.

The application connects to the server with intern certificate authority through TLS_PSK, then it generates the asymmetric RSA key pair and the certification signing request which is sent to the authority. Server sends signed certificate back and it is along with private key saved to the designated key store.

5.2 TLS_PSK

For application I decided to use packages created by Golang. There was a problem because this package does not support TLS with pre-shared key. From the previous research I know that there is one third-party library which is adapted to use TLS_PSK [30].

5.2.1 Raff

Raff/tls-psk library was created 3 years ago and it has not been maintained yet. It does not have any downloads, issue tracker nor active contributors.

For application needed I decided to use this library. Connecting to OpenSSL server was correct but connecting to server with CA was not possible because of unsupported cipher suites. Raff/tls-psk library contains only 4 versions of cipher suites and the others are not supported.

The other solution how to implement TLS_PSK to Go is using Cgo. Cgo allows Go to interoperate code written in C language. How Cgo works I describe in section 3.4.

5.3 Language C in Go

I created methods worked with OpenSSL in language C for initialization, connection, reading from and writing to the server.

5.3.1 Initialization

Initialization consists of four methods from *openssl/ssl.h* and *openssl/err.h*. Methods initialize *s_client*, add algorithms to the table and register the error strings, the available SSL/TLS ciphers and digests.

5.3.2 Connection

Connection to the server is created through TLS_PSK. Connection method creates a new SSL_CTX object as a framework to establish TLS/SSL enabled connections, next it creates a new BIO chain consisting of an SSL BIO (using ctx) followed by a setting of address and pre-shared key. The address consists of host name and port. The pre-shared key is set through *SSL_set_psk_client_callback* with SSL and a callback function as arguments. The purpose of the callback function is to select the PSK identity and the pre-shared key to use during the connection setup phase. The variable with the pre-shared key has to be global. The setting is followed by a connect BIO.

5.3.3 Reading from and writing to server

Created method *C_bio_read* reads 1 byte from BIO until the server sends the null terminator $'\backslash 0'$. Created method *C_bio_write* writes bytes to BIO with the null terminator $'\backslash 0'$ at the end.

5.4 Using C-methods in Go

Methods written in C language have to be called in Go code with prefix '*C.*' and have the correct types of arguments.

C does not have an explicit string type and strings are represented by a zero-terminated array of chars. Conversion between Go and C strings is done with the *C.CString*, *C.GoString*, and *C.GoStringN* functions. These conversions make a copy of the string data [13].

```
// Go string to C string
func C.CString(string) *C.char

// C string to Go string
func C.GoString(*C.char) string

// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string
```

As I wrote in the subsection 5.3.2, the pre-shared key has to be the global variable in C code. To set it from Go code where application takes the pre-shared key as an argument, I used followed commands.

```
//in C
char *psk_key;

//in Go
arg := os.Args
private_key := arg[3]

C.psk_key = C.CString(private_key)
```

5.5 Communication protocol

Communication between the Pairing Client and Server with Certificate Authority (CA Server) is based on null-terminated strings. The protocol looks as follows:

5.5.1 Connection using TLS_PSK

- Pairing client connects to the CA Server using given pairing key and default identity.
- CA Server validates Pairing Client (the identity is ignored).

The connection is provided through Go method *create_connection* with arguments port, the address of the server and pre-shared key. The pre-shared key is set as a global variable in C code (described in section 5.4) and subsequently is called C method to create the connection.

Client and server exchanged *ClientHello - ServerHello*, and *ClientKeyExchange - ServerKeyExchange*. They agreed to use common cipher suite.

5.5.2 Protocol version exchange

- Pairing Client sends supported protocol version.
- CA Server chooses the highest protocol version possible and sends it back.

Currently, is used protocol version only 1. Go method *protocol_version_exchange* calls write and read methods from C code.

```
protocol_version := C.CString("1")
err_write := C.C_bio_write(bio, protocol_version, 1)
server_protocol_version := C.C_bio_read(bio)
if *server_protocol_version != *protocol_version {
  return -1
}
```

5.5.3 Key length

- Pairing Client sends the component/subsystem identification.
- Pairing Client sends information if the key/certificate belongs to the CA or it is end certificate (CA or END).
- CA Server returns key length for the key generation.

There are used C methods for write and read. The subsystem identification (UA) is written to the server together with CA or END information from argument of application. Subsequently, the method reads *key_length* from server and uses standard library functions to convert string to the number from package *"strconv"* and generates the key from package *"crypto/rsa"*.

```
key_len, err := strconv.Atoi(C.GoString(key_length))
checkErr(err)

var key interface{}
key, err = rsa.GenerateKey(rand.Reader, key_len)
```

5.5.4 Certificate signing

- Pairing Client sends generated PEM encoded a certificate request (with domain name in Common Name field).
- Pairing Client sends IP address of the component/subsystem server.
- CA Server responses with the signed certificate (PEM encoded).

The application creates the template (tmp) based on observed IP address and domain_name from arguments. The template (tmp) is used during creating a new certificate request (CSR).

```
tmp := &x509.CertificateRequest {
  Subject     : pkix.Name{CommonName : ip},
  DNSNames    : []string{domain_name},
  IPAddresses : []net.IP{net.ParseIP(ip)},
}
csr,err:=x509.CreateCertificateRequest(rand.Reader,tmp,key)
```

The template and function *CreateCertificateRequest* are from package *"crypto/x509"*. The function returns bytes array, so to sending the request to the server, it has to be converted to pem string. The request has to be saved in temp file as follows:

```
pem.Encode(csr_file , &pem.Block{Type: "CERTIFICATE REQUEST",
    Bytes: csr})
```

and subsequently, it is read as a string and sent to the server. When the file with request is not needed, it is deleted through:

```
os.Remove("csr.pem")
```

The server answers with signed certificate chain. Signed certificate is saved to file and with generated key are saved to the key store.

5.5.5 Achieving trust

The connection ends when the server sends trust anchor as a CA certificate. After that, it is not possible to write to or read from the server. Correctness of achieving trust is checked up through:

```
strings.HasPrefix(str , "-----BEGIN")
```

5.6 Compilation and run

To run application, it is needed to install a several programs.

1. Go - see how to start with Go in section 3.2
2. GCC - the application use Cgo which needs C compiler
3. OpenSSL - is used in C code
4. libssl-dev - (or openssl-devel) contains compiled dynamic libraries *libssl* and *libcrypto* important for linker options (part of LDFLAGS).
   (a) *libssl* provides the client and server-side implementations for SSLv3 and TLS.
   (b) *libcrypto* provides general cryptographic and X.509 support needed by SSL/TLS [31].

Application is built and run through:

```
go build <name>.go
./<name> <server> <port> <PSK> <pathToKeyStore> <subjectIP>
   <subjectDomainName> <subjectID>
```

For example:

```
go build app.go
./app localhost 10443 aaaa /tmp 192.0.2.0 example.com END
```

6 Evaluation

In this chapter, I will evaluate process of working on this thesis. I will describe the workflow, the problems, ...

6.1 Select Libraries

Before analyzing libraries, I had to learn Go language, install it and try to programming. All important information about this language and working with it is described in the chapter 3.

The Internet provides a big amount of Go packages created by third parties. To the thesis, I selected more than one hundred packages working with cryptographic functions and hash operations. It was necessary to remove packages from untrusted and unsupported sources. The correct packages in table 4.1 are compared based on their actuality (topicality), the approach to solving problems, documentation accessibility, number of downloads and no payment license. The table 4.2 contains 4 libraries selected from table 4.1.

The analysis shows, (do dnesneho datumu) the package suitable all the requirements does not exist. The generation of cryptographic keys (within 2 to 8192 bytes) and support of TLS protocol at the same time is only in package Crypto from *Golang*. It is the main reason why I chose it to use in the implementation.

During the analysis I found package *Raff/tls-psk* using TLS_PSK cipher suites, but the package has not been supporting for 3 years, so it was not possible to use this package in the implementation.

The problem with TLS_PSK solves using *OpenSSL*.

6.2 OpenSSL and Language C

OpenSSL contains an open-source implementation of the SSL and TLS protocols. The libraries, written in the C programming language, implements basic cryptographic functions and provides various utility functions [33].

In the implementation was needed to create methods working with *OpenSSL* in langauge C. The methods are described in section 5.3 and called from Go code using Cgo (explained in section 3.4). .....

During the implementation did not rise any other problem.

6.3 Testing

The necessary step of implementation is testing multiple parts of application and all of the system acting together. I test whether each component fulfills its contract, but also whether they are composed and configured correctly and interact as expected.

6.3.1 Connectivity test

Connectivity test checks behavior of application for the wrong server, port or key. For the wrong server or port, the application returns *BIO_do_connect:* -1 what means the connection failed. For the wrong key, the application writes checking identity and key, *BIO_do_connect:* -1 what means the connection failed for the wrong key.

6.3.2 Step by step testing of communication

I created tests for checking communication in certain moments. Each test has to contain connection and the previous steps of communication to correct test of given step.

For example test of version exchange contains the correct connection to the server.

```
1  func Test_version_exchange_OK(t *testing.T) {
2    t.Log("Testing protocol version exchange... ")
3
4    bio := create_connection("localhost", "10443", "aaaa")
5    if bio == nil {
6      t.Errorf("Error while connecting to server")
7    }
8
9    if ret := protocol_version_exchange(bio); ret == -1 {
10     t.Errorf("Error: different version")
11   } else if ret == -2 {
12     t.Errorf("Error: problem with connection")
13   }
```

```
14    t.Log("Test OK")
15 }
```

Tests of generation certificate request and private key also contain generation with the correct and wrong connection, arguments and key size.

6.3.3 File test

These tests check saving certificates and keys without connection and previous communication with the server to the key store with the existing or non-existing path, correct key or empty interface.

6.4 Cross-compiler and Integration

7 Conclusion

This Bachelor thesis deals with cryptographic tools of Go programming language. Its goal was to evaluate cryptographic libraries and implement the prototype of a client application. The thesis was consulted and developed in cooperation with Y Soft Corporation, a.s.

The Go libraries were overviewed and compared based on their support of symmetric and asymmetric cryptographic algorithms and hash operations. In pursuance of analysis, the four libraries were selected and subsequently examined according to methods for generation of cryptographic keys and self-signed certificates, support for certificate signing requests, different formats, and SSL/TLS. From the report, one library was selected to the implementation.

The prototype of a client application is able to generate asymmetric keys and certificate signing requests, connect to a server application using TLS_PSK cipher suite and communicate with the server. The application is prepared to integrate to the company system.

The Bachelor thesis had a great personal benefit for me. I expanded my knowledge of Go programming language, OpenSSL, and the certificates. I also gain a great asset in designing and developing the application and I improved my programming skills in the languages C and Go.

....

....

Bibliography

1. DOSTÁLEK, Libor; VOHNOUTOVÁ, Marta. *Velký průvodce infrastrukturou PKI*. Computer Press, Albatros Media as, 2016.
2. SCHMEH, Klaus. *Cryptography and public key infrastructure on the Internet*. John Wiley & Sons, 2006.
3. SINGH, Simon; KOUBSKÝ, Petr; ECKHARDTOVÁ, Dita. *Kniha kódů a šifer: tajná komunikace od starého Egypta po kvantovou kryptografii*. Dokořán, 2003.
4. HOW TO SSL. *PEM Files*. 2017. Available also from: `http://how2ssl.com/articles/working_with_pem_files/`. [Online; 21-02-2017].
5. BAKKER, Paul. *ASN.1 key structures in DER and PEM*. 2014. Available also from: `https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem`. [Online; 21-02-2017].
6. FREIER, Alan; KARLTON, Philip; KOCHER, Paul. The secure sockets layer (SSL) protocol version 3.0. 2011.
7. OPPLIGER, Rolf. *Security technologies for the world wide web*. Artech House, 2003.
8. ERONEN, Pasi; TSCHOFENIG, Hannes. *Pre-shared key ciphersuites for transport layer security (TLS)*. 2005. Technical report.
9. DOXSEY, Caleb. *Introducing Go*. O'Reilly Media, 2016.
10. HARA, Mikio. *golang/go*. 2017. Available also from: `https://github.com/golang/go/wiki`. [Online; 27-02-2017].
11. VARGHESE, Shiju. *Understanding Golang Packages*. 2014. Available also from: `https://thenewstack.io/understanding-golang-packages/`. [Online; 27-02-2017].
12. GOLANG. *Command cgo*. 2017. Available also from: `https://golang.org/cmd/cgo/`. [Online; 24-03-2017].
13. GERRAND, Andrew. *The Go Blog*. 2011. Available also from: `https://blog.golang.org/c-go-cgo`. [Online; 25-03-2017].
14. DOGAN, Jaana Burcu. *golang/go*. 2016. Available also from: `https://github.com/golang/go/wiki/cgo`. [Online; 25-03-2017].

15. CHISNALL, David. *The Go programming language phrasebook*. Addison-Wesley, 2012.
16. BALBAERT, Ivo. *The way to go: a thorough introduction to the Go programming language*. iUniverse, 2012.
17. SUMMERFIELD, Mark. *Programming in Go: creating applications for the 21st century*. Addison-Wesley, 2012.
18. HARRIS, Alan. *Go: Up and Running*. O'Reilly Media, 2015.
19. KOZYRA, Nathan. *Mastering concurrency in go*. Packt Publishing, 2014.
20. GOLANGLIBS. *Security*. 2017. Available also from: `https://golanglibs.com/top?q=security`. [Online; 27-02-2017].
21. GOLANGLIBS. *Cryptography - Go libraries and apps*. 2017. Available also from: `https://golanglibs.com/category/cryptography?sort=top`. [Online; 27-02-2017].
22. GO DOC. *crypto - GoDoc*. 2017. Available also from: `https://godoc.org/golang.org/x/crypto`. [Online; 27-02-2017].
23. GOLANG. *Packages*. 2017. Available also from: `https://golang.org/pkg/`. [Online; 27-02-2017].
24. PURE-GO-LIBS. *Pure Go Libs*. 2017. Available also from: `http://golang.cat-v.org/pure-go-libs`. [Online; 27-02-2017].
25. GOLANG. *Package crypto*. 2012. Available also from: `https://golang.org/pkg/crypto/`. [Online; 18-11-2016].
26. SERMODIGITAL. *SermoDigital/jose*. 2015. Available also from: `https://github.com/SermoDigital/jose`. [Online; 18-11-2016].
27. LIBP2P. *libp2p/go-libp2p-crypto*. 2015. Available also from: `https://github.com/libp2p/go-libp2p-crypto`. [Online; 18-11-2016].
28. ETHEREUM. *ethereum/go-ethereum*. 2013. Available also from: `https://github.com/ethereum/go-ethereum`. [Online; 18-11-2016].
29. HINEK, M Jason. On the security of multi-prime RSA. *Journal of Mathematical Cryptology*. 2008, vol. 2, no. 2, pp. 117–147.
30. RAFF. *raff/tls-psk*. 2014. Available also from: `https://github.com/raff/tls-psk`. [Online; 27-02-2017].
31. OPENSSL. *openssl/openssl*. 2016. Available also from: `https://github.com/openssl/openssl`. [Online; 02-05-2017].

32. DIE.NET. *Section 3: library functions - Linux man pages*. 2017. Available also from: `https://linux.die.net/man/3/`. [Online; 02-05-2017].

33. FOUNDATION, Inc. OpenSSL. *OpenSSL*. 2017. Available also from: `https://www.openssl.org/docs/man1.0.2/`. [Online; 02-05-2017].