BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PORTING TANG TO OPENWRT

PORTOVANIE TANG NA OPENWRT

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR **TIBOR DUDLÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDREJ LICHTNER**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

DUDLÁK, Tibor. *Porting Tang to OpenWRT*. Brno, 2017. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor Lichtner Ondrej.

Porting Tang to OpenWRT

Declaration

Hereby I declare that this term project was prepared as an original author's work under the supervision of Ing. Ondrej Lichtner. The supplementary information was provided by Jan Pazdziora, Ph. D.

. . . .
Tibor Dudlák
January 28, 2017

Contents

1 Introduction 2

2 Encryption 3

3 Escrow 4

4 Tang 5
- 4.1 Binding with Tang 6
- 4.2 Provisioning 6
- 4.3 Recovery 6
- 4.4 Security 7
  - 4.4.1 Man-in-the-Middle attack 7
  - 4.4.2 Compromise the client to gain access to cJWK 7
  - 4.4.3 Compromise the server to gain access to sJWK's private key 7
- 4.5 Building Tang 7
- 4.6 http-parser 8
- 4.7 systemd / xinetd 8
- 4.8 José 8
  - 4.8.1 jansson 9
  - 4.8.2 OpenSSL 9
  - 4.8.3 zlib 9
- 4.9 Server enablement 9

5 Clevis 10
- 5.1 PIN: Tang 10
- 5.2 PIN: HTTP 10
- 5.3 PIN: SSS - Shamir Secret Sharing 11
- 5.4 Binding LUKS volumes 11
  - 5.4.1 Dracut 11
  - 5.4.2 UDisks2 11

6 OpenWRT 12

7 Conclusion 13

Bibliography 14

Chapter 1

Introduction

Nowadays, the whole world uses information technologies to communicate and simply to spread knowledge in form of bits to the other people. But there are personal information such as photos from a family vacation, videos of our children as they grow, contracts, testaments and so on which we would like to protect.

Encryption protects our data even when we do not know that. It protects our conversation privacy, personal information stored all over governmental authorities, bank accounts, in general data transmitted around the internet and stored on our hard drives. Encryption provides process of transforming our information in such way that only trusted person can decrypt data and access it. An unauthorized person might be able to access secured data but will not be able to read the information from them without the proper key. The most important thing is keeping the encryption key (the password) a secret.

The goal of this bachelor thesis will be to port *Tang* server 4, its dependencies, and *Clevis* framework 5 to *OpenWRT* system 6. With accomplishing, this we will be able to automatize process of unlocking encrypted drives on our private respectively home network. There will be no need for any decryption server but only *OpenWRT* device running the *Tang* server itself.

Chapter 2

Encryption

For most of us is common to have a password protected system. But the encrypted disk requires another password (key) to decrypt. Imagine that you came home in mood to enjoy your time and your system ask for password not once but twice. I think this is reason why most of us do not use encryption even when we know it will protect our data. This is what Tang 4 is for and to whom is for because we want automatize things.

Lets look how the encryption is typically done. As you can see in the image below it all starts with desire to keep our data to ourselves and as a secret to the other people. At the most these secrets are stored on our hard drives. Usually we encrypt this secret by using an encryption key. But our secret data might grow in size, and it is time and resource consuming to decrypt and encrypt secret every time encryption key changes or it is compromised. Because of that we wrap encrypted data in the key encryption key and this is what system prompts from us on boot. So changing key encryption key does not affect encrypted data. We can change it whenever we desire to and redistribute new key to all users who are suposed to access this data.

To automatize this we could generate something cryptographicaly stronger than user provided password. Then we store this cryptographicaly stronger random key on some remote system which is can then get from it. This is basically how the Escrow 3 model works.

Chapter 3

Escrow

Before Tang, automated decryption was handled usually with Key Escrow (also known as a “fair” cryptosystem). Client using Key Escrow usually generates a key, encrypt data with it and then stores the key encryption key in a remote server. But it is not as simple as it sounds.

To deliver those keys we desire to store on Escrow server we have got to encrypt the channel on which we distribute the key. Without encrypted link when transmitting keys over not secure network anyone who is listening to the network traffic could immadiately fetch this key. This should signal security risks and of course we do not want third party to acces our secret data. Usually we encrypt channel with TLS or GSSAPI as you can see on Figure 3.1 below. Now it sounds secure to have an encrypted channel isn’t it? Unfortunatelly it is not enough either.

We can not just start sending these keys to server if we do not know that this server is one that it act to be. This server has to have its own identity to trust and we have to authenticate to this server too. Increasing amount of keys implicates need for certification authority server (CA) or Key distribution center (KDC) to manage all of them. With all these keys and at this point only, server can verify if person is permitted to get his or her key, and client is able to identify trusted server. This is a fully statefull process. To sum up an authorized third party may gain access to keys stored on Escrow server under certain circumstances only.

Figure 3.1: Escrow model

Complexity of this system increases the attack surface and for this complex system would be unimaginable to not have backups. Escrow server may store lots of keys from lots of different places and basically we can not afford to loose them.

Chapter 4

Tang

Tang server is open source project implemented in C programming language, and it bind data to network presence. It rely on few other software libraries:

- http-parser 4.6
- systemd / xinetd 4.7
- jose 4.8
  - jansson 4.8.1
  - openssl 4.8.2
  - zlib 4.8.3

Now when we know what Tang rely on and what is for we should be more specific about it and find out how Tang provides its service. What binding data to network presence really means? Essentially it allows us to make some data to be available only when the system containing the data is on a certain, usually secure, network.

Tang server advertises asymmetric keys and client is able to get list off these signing keys 4.8 by HTTP GET request. Now follows the provisioning step. With list of these public keys may process of encrypting data start. Client chooses one of asymmetric keys to generate a unique encryption key with. After this client encrypt data using created key. Once the data is encrypted, the key is discarded. Some small metadata has to be produced as part of this operation. The client should store these metadata to work with them when decrypting. Finally when client want to access encrypted data it must be able to recover encryption key. This step starts with loading stored metadata and ends with simply performing a HTTP POST to Tang server. Server perform its mathematical operation and send result back to client. Finally client has to calculate the key value rather then server calculates it. So the Tang server never knew the value of the key and literraly nothing about its clients.

Figure 4.1: Tang model

On Figure 4.1 you can see the Tang model. It is very similar to Escrow model 3.1 but there are some thing missing. In fact there is no longer need for TLS channel to secure comunication between client and server, and that is because Tang implements the McCallum-Relyea exchange 4.2 as described below.

4.1 Binding with Tang

Client performs an ECDH [13] key exchange in order to generate the binding key. Then client discards its own private key so that the Tang server is the only party that can reconstitute the binding key. To blind the client's public key and the binding key Tang uses a third, ephemeral key. Ephemeral key is generated for each execution of a key establishment process. Now only the client can unblind his public key and binding key.

Figure 4.2: McCallum-Relyea exchange

4.2 Provisioning

The client selects one of the Tang server's exchange keys (sJWK; identified by the use of deriveKey in the sJWK's key_ops attribute). The client generates a new (random) JWK (cJWK). The client performs its half of a standard ECDH exchange producing dJWK which it uses to encrypt the data. Afterwards, it discards dJWK and the private key from cJWK.

The client then stores cJWK for later use in the recovery step. Generally speaking, the client may also store other data, such as the URL of the Tang server or the trusted advertisement signing keys.

$$s = g * S \tag{4.1}$$

$$c = g * C \tag{4.2}$$

$$K = s * C \tag{4.3}$$

4.3 Recovery

To recover dJWK after discarding it, the client generates a third ephemeral key (eJWK). Using eJWK, the client performs elliptic curve group addition of eJWK and cJWK, producing xJWK. The client POSTs xJWK to the server.

The server then performs its half of the ECDH key exchange using xJWK and sJWK, producing yJWK. The server returns yJWK to the client.

The client then performs half of an ECDH key exchange between eJWK and sJWK, producing zJWK. Subtracing zJWK from yJWK produces dJWK again.

Expressed mathematically (capital = private key):

$$e = g * E \tag{4.4}$$

$$x = c + e \tag{4.5}$$

$$y = x * S \tag{4.6}$$

$$z = s * E \tag{4.7}$$

$$K = y - z \tag{4.8}$$

4.4 Security

We can now compare Tang and Escrow. In contrast, Tang is stateless and doesn't require TLS or authentication. Tang also has limited knowledge. Unlike escrows, where the server has knowledge of every key ever used, Tang never sees a single client key. Tang never gains any identifying information from the client.

	Escrow	Tang
Stateless	No	Yes
SSL/TLS	Required	Optional
X.509	Required	Optional
Authentication	Required	Optional
Anonymous	No	Yes

Table 4.1: Comparing Escrow and Tang

Let's think about the security of Tang system. Is it really secure without an encrypted channel or even without authentication? So long as the client discards its private key, the client cannot recover dJWK without the Tang server. This is fundamentally the same assumption used by Diffie-Hellman (and ECDH).

4.4.1 Man-in-the-Middle attack

In this case, the eavesdropper in this case sees the client send xJWK and receive yJWK. Since, these packets are blinded by eJWK, only the party that can unblind these values is the client itself (since only it has eJWK's private key). Thus, the MitM attack fails.

4.4.2 Compromise the client to gain access to cJWK

It is of utmost importance that the client protect cJWK from prying eyes. This may include device permissions, filesystem permissions, security frameworks (such as SELinux) or even the use of hardware encryption such as a TPM. How precisely this is accomplished is an exercise left to the client implementation.

4.4.3 Compromise the server to gain access to sJWK's private key

The Tang server must protect the private key for sJWK. In this implementation, access is controlled by filesystem permissions and the service's policy. An alternative implementation might use hardware cryptography (for example, an HSM) to protect the private key.

4.5 Building Tang

Tang is originally packaged for Fedora OS version 24 and later but we can build it from source of course.

doplnit...

To build it from source download source from poject's GitHub or clone it. Make sure you have all needed dependencies installed and then run:

```
$ autoreconf -if
$ ./configure -prefix=/usr
$ make
$ sudo make install
```

Optionally to run tests:

```
$ make check
```

4.6 http-parser

Tang uses this is parser for both parsing HTTP requests and HTTP responses. You can find this parser on its own GitHub [10].

4.7 systemd / xinetd

systemd [12] is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system. systemd provides aggressive parallelization capabilities, uses socket and D-Bus activation for starting services, offers on-demand starting of daemons, keeps track of processes using Linux control groups, maintains mount and automount points, and implements an elaborate transactional dependency-based service control logic.

4.8 José

José [11] is a C-language implementation of the Javascript Object Signing and Encryption standards. Specifically, José aims towards implementing the following standards:

- RFC 7515 - JSON Web Signature (JWS) [5]
- RFC 7516 - JSON Web Encryption (JWE) [3]
- RFC 7517 - JSON Web Key (JWK) [4]
- RFC 7518 - JSON Web Algorithms (JWA) [7]
- RFC 7519 - JSON Web Token (JWT) [6]
- RFC 7520 - Examples of ... JOSE
- RFC 7638 - JSON Web Key (JWK) Thumbprint [4]

JOSE (Javascript Object Signing and Encryption) is a framework intended to provide a method to securely transfer claims (such as authorization information) between parties.

Tang uses JWKs in comunication between client and server. Both POST request and reply bodies are JWK objects.

4.8.1 jansson

Jansson [8](licenced under MIT licence)is a C library for encoding, decoding and manipulating JSON data. It features:

- Simple and intuitive API and data model
- Comprehensive documentation
- No dependencies on other libraries
- Full Unicode support (UTF-8)
- Extensive test suite

4.8.2 OpenSSL

OpenSSL [2] contains an open-source implementation of the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is used by network applications to secure communication between two parties over network.

4.8.3 zlib

Library zlib [1] is used for data compression.

?

4.9 Server enablement

Enabling a Tang server is a two-step process. First, enable and start the service using systemd.

```
$ sudo systemctl enable tangd-update.path

$ sudo systemctl start tangd-update.path

$ sudo systemctl enable tangd.socket

$ sudo systemctl start tangd.socket
```

Second, generate a signing key and an exchange key.

```
$ sudo jose gen -t ’{"alg":"ES256"}’ -o /var/db/tang/sig.jwk

$ sudo jose gen -t ’{"kty":"EC","crv":"P-256","key_ops":["deriveKey"]}’ \
       -o /var/db/tang/exc.jwk
```

Chapter 5

Clevis

Clevis provides a pluggable key management framework for automated decryption [9]. It can handle even automated unlocking of LUKS volumes. To do so we have to encrypt some data with simple command:

```
$ clevis encrypt PIN CONFIG < PLAINTEXT > CIPHERTEXT.jwe
```

In clevis terminology, a *PIN* is a plugin which implements automated decryption. We simply pass the name of supported pin here. Secondly *CONFIG* is a JSON object which will be passed directly to the *PIN*. It contains all the necessary configuration to perform encryption and setup automated decryption.

5.1 PIN: Tang

Clevis has full support for Tang. Here is an example of how to use Clevis with Tang:

```
$ echo hi | clevis encrypt tang '{"url": "http://tangserver"}' > hi.jwe
The advertisement is signed with the following keys:
    kWwirxc5PhkFIH0yE28nc-EvjDY

Do you wish to trust the advertisement? [yN] y
```

In this example, we encrypt the message „hi“ using the Tang pin. The only parameter needed in this case is the URL of the Tang server. During the encryption process, the Tang pin requests the key advertisement from the server and asks you to trust the keys. This works similarly to SSH.

Alternatively, you can manually load the advertisment using the adv parameter. This parameter takes either a string referencing the file where the advertisement is stored, or the JSON contents of the advertisment itself. When the advertisment is specified manually like this, Clevis presumes that the advertisement is trusted.

5.2 PIN: HTTP

Clevis also ships a pin for performing escrow using HTTP. Please note that, at this time, this pin does not provide HTTPS support and is suitable only for use over local sockets. This provides integration with services like Custodia.

5.3 PIN: SSS - Shamir Secret Sharing

Clevis provides a way to mix pins together to provide sophisticated unlocking policies. This is accomplished by using an algorithm called Shamir Secret Sharing (SSS).

5.4 Binding LUKS volumes

Clevis can be used to bind a LUKS volume using a pin so that it can be automatically unlocked.

How this works is rather simple. We generate a new, cryptographically strong key. This key is added to LUKS as an additional passphrase. We then encrypt this key using Clevis, and store the output JWE inside the LUKS header using LUKSMeta.

Here is an example where we bind /dev/vda2 using the Tang ping:

```
$ sudo clevis bind-luks /dev/sda1 tang '{"url": "http://tang.local"}'
The advertisement is signed with the following keys:
        kWwirxc5PhkFIH0yE28nc-EvjDY

Do you wish to trust the advertisement? [yN] y
Enter existing LUKS password:
```

Upon successful completion of this binding process, the disk can be unlocked using one of the provided unlockers.

5.4.1 Dracut

The Dracut unlocker attempts to automatically unlock volumes during early boot. This permits automated root volume encryption. Enabling the Dracut unlocker is easy. Just rebuild your initramfs after installing Clevis:

```
$ sudo dracut -f
```

Upon reboot, you will be prompted to unlock the volume using a password. In the background, Clevis will attempt to unlock the volume automatically. If it succeeds, the password prompt will be cancelled and boot will continue.

5.4.2 UDisks2

Our UDisks2 unlocker runs in your desktop session. You should not need to manually enable it; just install the Clevis UDisks2 unlocker and restart your desktop session. The unlocker should be started automatically.

This unlocker works almost exactly the same as the Dracut unlocker. If you insert a removable storage device that has been bound with Clevis, we will attempt to unlock it automatically in parallel with a desktop password prompt. If automatic unlocking succeeds, the password prompt will be dissmissed without user intervention.

Chapter 6

OpenWRT

Chapter 7

Conclusion

Bibliography

[1] Adler, M.: *zlib*. [Online].
Retrieved from: https://github.com/madler/zlib

[2] Eric A. Young, T. J. H.: *OpenSSL*. [Online].
Retrieved from: https://github.com/openssl/openssl

[3] Jones, e. a.: *JSON Web Encryption*. [Online].
Retrieved from:
http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption

[4] Jones, e. a.: *JSON Web Keys*. [Online].
Retrieved from: http://tools.ietf.org/html/draft-ietf-jose-json-web-key

[5] Jones, e. a.: *JSON Web Signing*. [Online].
Retrieved from:
http://tools.ietf.org/html/draft-ietf-jose-json-web-signature

[6] Jones, e. a.: *JSON Web Tokens*. [Online].
Retrieved from:
https://tools.ietf.org/html/draft-ietf-oauth-json-web-token

[7] Jones, e. a.: *JWT Authorization Grants*. [Online].
Retrieved from: http://tools.ietf.org/html/draft-ietf-oauth-jwt-bearer

[8] Lehtinen, P.: *jansson*. [Online].
Retrieved from: https://github.com/akheron/jansson

[9] McCallum, N.: *Clevis*. [Online].
Retrieved from: https://github.com/latchset/clevis

[10] McCallum, N.: *HTTP-parser*. [Online].
Retrieved from: https://github.com/nodejs/http-parser

[11] McCallum, N.: *jose*. [Online].
Retrieved from: https://github.com/latchset/jose

[12] Poettering, L.; Sievers, K.; Hoyer, H.; et al.: *systemd*. [Online].
Retrieved from: https://en.wikipedia.org/wiki/Systemd

[13] Wikipedia: *Elliptic curve Diffie–Hellman*. [Online].
Retrieved from:
https://en.wikipedia.org/wiki/Elliptic_curve_Diffie%E2%80%93Hellman