BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PORTING TANG TO OPENWRT

PORTOVANIE TANG NA OPENWRT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TIBOR DUDLÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDREJ LICHTNER**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

This thesis describes the encryption and its application to secure the computer's hard drive. It describes the structure of the encrypted disk's partition according to the LUKS specification on Linux operating systems. The thesis focus on describing possibilities of automating the disk decryption process using an external server that enters the process as a third party. It describes the principles of Key Escrow and Tang server. Steps required to compile and configure the Tang server are described too. Also described is Tang server's client – Clevis. The main objective of this work was to port and document the process of porting the Tang server and its dependencies to OpenWrt system, described in this thesis too, which is designed for embedded devices such as WiFi routers. The thesis also includes a documented process of contributing changes and newly created OpenWrt packages to corresponding Open Source projects.

Abstrakt

Hlavným cieľom tejto práce je sprístupnenie serveru Tang na vstavané zariadenia typu WiFi smerovač s plne modulárnym operačným systémom OpenWrt. Tým dosiahneme anonymnú správu šifrovacích kľúčov pre domáce siete a siete malých firiem. Preto táto práca popisuje problematiku šifrovania a jeho využitie na zabezpečenie pevného disku počítača. Oboznámuje čitateľa so štruktúrou šifrovaného diskového oddielu podľa LUKS špecifikácie na operačných systémoch typu Linux. Práca rozoberá možnosti automatizácie odomykania šifrovaných diskov použitím externého servera, ktorý vstupuje do procesu ako tretia strana. Sú v nej popísané princípy serverov Key Escrow a Tang. Dosiahnutie hlavného cieľa je možné vďaka procesu portovania a cross-kompilácie na platforme Linux. Práca obsahuje zdokumentovaný postup prispievania zmien a novo vytvorených balíkov pre OpenWrt do príslušných Open Source projektov.

Keywords

porting, Tang, server, Clevis, client, Escrow, OpenWrt, operating system, embedded device, encryption, LUKS, hard drive, disk partition, encryption key, automation, cross-compiling, package system

Kľúčové slová

portovanie, Tang, server, Clevis, klient, Escrow, OpenWrt, operačný systém, vstavané zariadenie, šifrovanie, LUKS, pevný disk, diskový oddiel, šifrovací kľúč, automatizácia, cross-kompilácia, balíkový systém

Reference

DUDLÁK, Tibor. *Porting Tang to OpenWRT*. Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondrej Lichtner

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. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tibor Dudlák
April 17, 2018

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

Chapter 1

Introduction

We spend our time searching for
security, and hate it when we get it.

John Steinbeck[10]

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

Encryption, as described in chapter 2, protects our data and privacy even when we do not realize that. It provides process of transforming our information in such way that only trusted person or device can decrypt data and retrieve it. An unauthorized party might be able to access secured data but will not be able to read the information from it without the proper key. The most important thing is keeping the encryption key a secret.

With an increasing number of encryption keys to store and protect, there might be necessary to consider using Key management server. This server should provide a secure and persistent service to its clients. One of the possible solutions for persistent Key management is to deploy *Key Escrow* server described in chapter 3. Another solution is server *Tang*, which principles are mentioned in chapter 4. Tang is completely anonymous key recovery service. In contrast to Key Escrow server, Tang does not know any key. It only provides mathematical operation for its clients to recover them.

The goal of this bachelor thesis is to port the *Tang* server, and its dependencies listed in section 4.6 to *OpenWrt* system for *embedded* devices mentioned in chapter 5. With accomplishing this, we will be able to automatize process of unlocking encrypted drives on our private home or small office network, therefore securing our data stored on personal computer's hard drive and/or NAS (Network-attached storage) server if it is stolen. There will be no need for any decryption or even Key Escrow server but the *OpenWrt* device running the *Tang* server itself only.

To achieve this goal . . . **[[thesis tour]]**

Chapter 2

Encryption and how we use it

We may not realize this, but we use encryption every day. The purpose of encryption is to keep us safe when we are browsing the internet or just storing our sensitive information on digital media. In general encryption is used to secure our data, whether transmitted around the internet or stored on our hard drives, from being compromised. Encryption protects us from many threats.

It protects us from identity theft. Our personal information stored all over governmental authorities should be secured with it. Encryption takes care for not revealing sensitive information about ourselves, to protect our financial details, passwords along with others, mainly when we bank online from being defraud.

It looks after our conversation privacy. To be more specific, our cell phone conversations from eavesdroppers and our online chatting with acquaintances or colleagues. It also allows attorneys to communicate privately with their clients and it aims to secure communication between investigation bureaus to exchange sensitive information about lawbreakers.

If we encrypt our laptop or desktop computer's hard drive encryption protects our data in case the computer or hard drive is stolen.

2.1 Security and encryption

Security is not binary; it is a sliding scale of risk management [3]. People are used to mark things, for example good and bad, expensive, and cheap. But we know that people may differ on image/sense. For example, there is no such thing as line or sign which tells us, this part of town is secure, and this is not. The way we reason about security is that we study environment entering or observing it, and we begin to decide whether it is secure or not. Especially at enterprise sector.

Encryption, on its own, might not be enough to make our data or infrastructure secure. Companies define their own security strategies which may include encryption or not at all. It all relies on company's needs or will to take risks in conclusion of getting high gain from them. In any case, encryption is definitely a critical element of security.

According to 2016 Global Encryption Trends Study, independently conducted by the Ponemon Institute, the enterprise-wide encryption in 5 years increased from 15 to 38 percent. Also the ratio of companies with no encryption strategy at all decreased from 37 to 15 percent. More than 50 percent companies are using extensively deployed encryption technologies to encrypt mostly databases, infrastructure and laptop hard drives [11].

2.2 Hard drive encryption

In this thesis we will talk about hard drive encryption... [[**bla ble bli**]]

It all starts, as mentioned, with desire to keep our data to ourselves and as a secret to others. More often than not, these secrets are stored on our hard drives. Let us take a look on how is encryption typically done.

To protect secret data we usually encrypt this data by using an encryption key - see Figure 2.1. However, this secret data might grow in size, and it is time and resource consuming to decrypt and encrypt all our data every time the encryption key changes or it is changed. Because of that, we wrap the encryption key in the key encryption key. This key might be actually user typed pass phrase which system prompts from user when booting or simply when it wants to access encrypted hard drive partition.

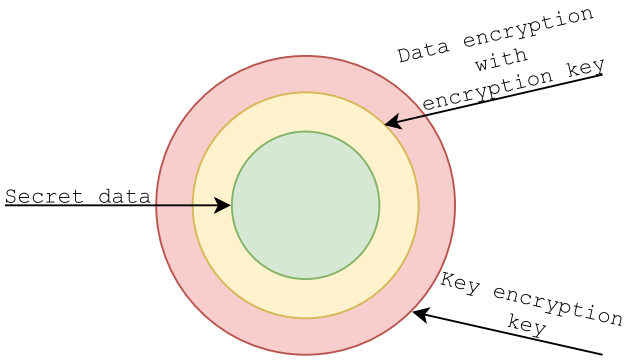

Figure 2.1: How we encrypt data

Changing the key encryption key does not affect encrypted data. It can be changed whenever user desire to, and redistributed the new key to all users or services who are supposed to access this data.

Passwords are the most common authentication for accessing computer systems, files, data, and networks. It is important to keep changing them in reasonable time. One way or another, we still keep them seeing them on monitors or desktop written on sticky notes, and this is absolutely not secure. In fact, users are the most vulnerable part of securing our systems. To aid their memory, users often include part of a phone number, family name, Social Security number, or birth date in their passwords [12]. They choose cryptographicaly weak passwords, dictionary words, which are easy to remember but also easy to guess or to break with brute-force attacks in short period of time. According to Splash Data [9], a supplier of security applications, the most common user selected password in the year 2016 was „123456“. They claim that people continue to put themselves at risk for hacking and identity theft by using weak, easily guessable passwords. To create strong password you can follow any trustworthy guide[1] on the Internet.

More secure way to create key encryption key would be to generate it.cryptographically stronger key encryption key than password provided by user. Then, we store this random key on a remote system, from where we can get or change it later. This is basically how the Key Escrow 3 model works.

[1] https://www.centos.org/docs/4/html/rhel-sg-en-4/s1-wstation-pass.html#S2-WSTATION-PASS-CREATE

For people, is common to have a password protected system. But encrypting hard drive means creating another layer of computer security that could disrupt our daily basis. Imagine, you come home in a mood to enjoy your time, and your system asks for password, not once, but twice just because of encrypted disk. This might be the reason why most of us do not use encryption, even when we know it will protect our data.

2.2.1 LUKS

[[master key hmm refactor text]]

LUKS (Linux Unified Key Setup) is a platform-independent disk encryption specification. It was created by Clemens Fruhwirth in 2004 and was originally intended for Linux distributions.

Referential implementation of LUKS, which was originally meant for Linux, is using a dm-crypt subsystem for bulk data encryption. This subsystem is not particularly bound to LUKS. Alongside Linux implementation exists LibreCrypt, the Windows implementation based on original FreeOTFE [13] project by Sarah Dean.

A LUKS partition can have as many user passwords as there are available key slots, and to access the partition, the user has to provide only one of these passwords [4].

Hard drive with a LUKS partition has notable structure, see Figure 2.2. The entire partition start with the LUKS partition header containing the key material. After header there is section with bulk data, which are encrypted with the master key.

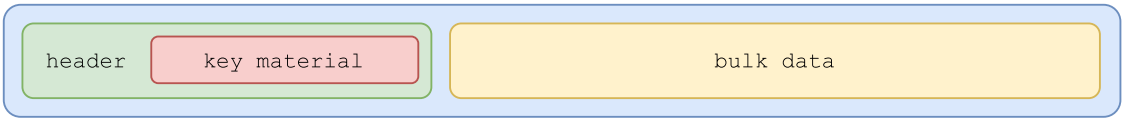

Figure 2.2: LUKS volume structure

Header, also marked as *phdr*, contains information about the used cipher, cipher mode, the key length, a uuid and a master key checksum. Also, the *phdr* contains information about the key slots. Every key slot is associated with a key material section after the *phdr*. When a key slot is active, the key slot stores an encrypted copy of the master key in its key material section. This encrypted copy is locked by a user password. Structure of key slot is on Figure 2.3.

2.2.2 Encrypting with LUKS

Creating a LUKS volume might be quite tough process. Hard drive partition must contain the LUKS header just before the encrypted data. Lets sum up the easier way first.

In case we have not installed our Linux operation system yet, we could simply select an option in time of installation. Then the installation wizard will most likely asks for pass phrase - the key encryption key. To demonstrate this, screen shots with Fedora 25 system installation can be found on appendix B.

If we have already system installed with lots of data on partition, process will probably last longer and the procedure will be more complex. There is no way you can encrypt whole system disk with LUKS without unmounting partition to encrypt. For this purpose was developed *luksipc*, the LUKS In-Place Conversion Tool [2]. Steps to encrypt disk using *luksipc* are on appendix C.

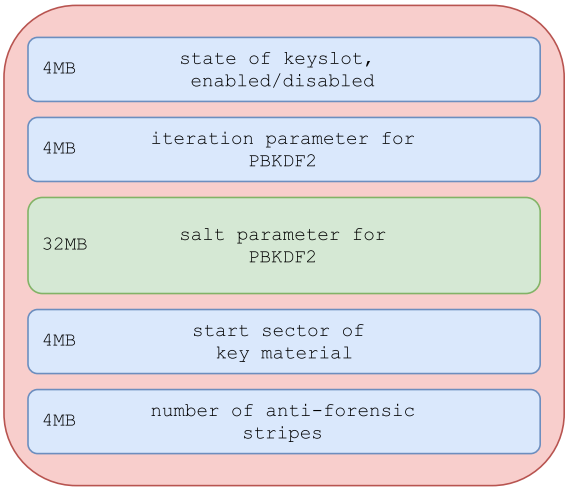

Figure 2.3: LUKS Key Slot

Chapter 3

Key management systems

[[write something LOL]]

Email Security

Before Tang, automated decryption was usually handled by Key Escrow server (also known as a "fair" cryptosystem). A client using Key Escrow usually generates a key, encrypts data with it and then stores the key encryption key on a remote server. However, it is not as simple as it sounds and there are couple of security concerns.

3.1 Escrow security

To deliver these keys we want to store on Escrow server, we have to encrypt the channel on which we distribute them. When transmitting keys over non secure network without encrypted link, anyone listening to the network traffic could immadiately fetch the key. This should signal security risks, and, of course, we do not want any third party to access our secret data. Usually we encrypt a channel with TLS(Transport Layer Security) or GSSAPI (Generic Security Services Application Program Interface) as shown on a Figure 3.1 below. Unfortunatelly, this is not enough to call the communication secure.

We cannot just start sending these keys to the escrow server, if we do not know whether this server is the one it acts to be. This server has to have its own identity to be verified, and the client have to authenticate to this server too. Increasing amount of keys implicates a 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 the client is permitted to get their key, and the client is able to identify trusted server. This is a fully stateful process. To sum up, an authorized third party may gain access to keys stored on Escrow server under certain circumstances only.

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

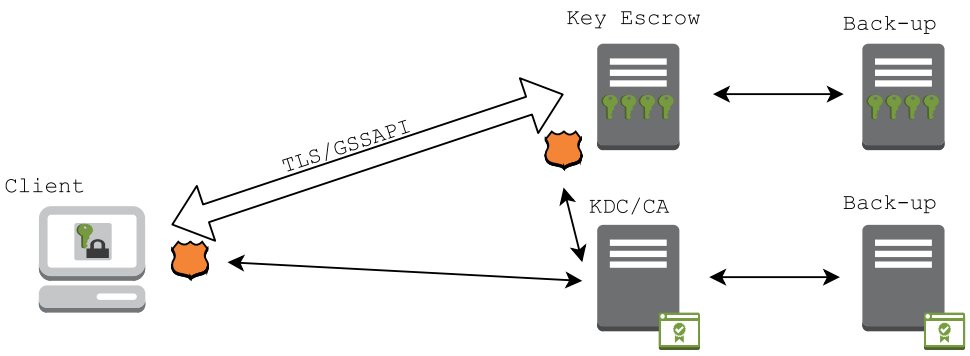

Figure 3.1: Escrow model

Chapter 4

Tang server

[[re-factor entire chapter]] Tang server is an open source project implemented in C programming language, and it binds data to network presence. What does binding data to network presence really mean? Essentially, it allows us to make some data to be available only when the system containing the data is on a particular, usually secure, network.

4.1 Tang - binding daemon

Tang server advertises asymmetric keys and a client is able to get the list of these signing keys 4.6.3 by HTTP (Hypertext Transfer Protocol) GET request. The next step is the provisioning step. With the list of these public keys the process of encrypting data may start. A client chooses one of the asymmetric keys to generate a unique encryption key. After this, the client encrypts data using the created key. Once the data is encrypted, the key is discarded. Some small metadata have to be produced as a part of this operation. The client should store these metadata to work with it when decrypting.

Finally, when the client wants to access the encrypted data, it must be able to recover encryption key. This step starts with loading the stored metadata and ends with simply performing a HTTP POST to Tang server. Server performs its mathematical operation and sends the result back to the client. Finally, the client has to calculate the key value, which is better than when server calculates it. So the Tang server never knew the value of the key and literraly nothing about its clients.

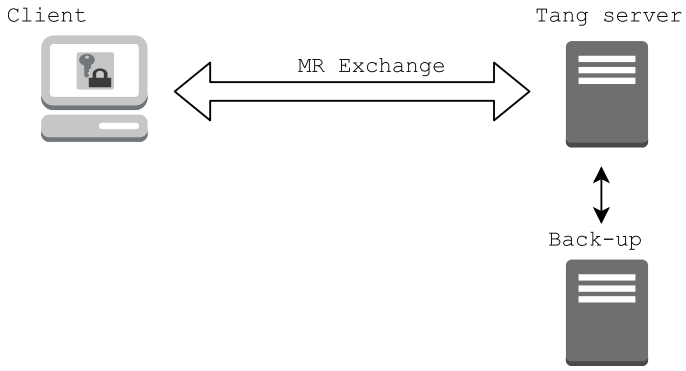

Figure 4.1: Tang model

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 a need for TLS channel to secure comunication between the client and the server, and that is the reason why Tang implements the McCallum-Relyea exchange 4.1 as described below.

4.2 Binding with Tang

A client performs an ECDH key exchange using the McCallum-Relyea algorithm 4.1 in order to generate the binding key. Then the client discards its own private key so that the Tang server is the only party that can reconstitute the binding key. To blind **[[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.

Provisioning		Recovery	
used client's side	server's side	client's side	server's side
	$S \epsilon_R [1, p-1]$	$E \epsilon_R [1, p-1]$	
	$s = gS$	$x = c + gE$	
	$\leftarrow$ s	x $\rightarrow$	
$C \epsilon_R [1, p-1]$			$y = zS$
$e = gC$			$\leftarrow$ y
$K = gSC = sC$		$K = y - sE$	
Discard: K, C			
Retain s, c			

Table 4.1: McCallum-Relyea exchange

4.3 Provisioning

The client selects one of the Tang server's exchange keys (we will call it sJWK; identified by the use of deriveKey in the sJWK's key_ops attribute). The lowercase „s“ stands for server's key pair and JWK is used format of the message. The client generates a new (random) JWK (cJWK; c stands for client's key pair). 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.

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

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

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

4.4 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.

Mathematically (capital is private key; g stands for generate) client's operation:

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

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

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

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

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

4.5 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 4.2: 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).

4.5.1 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.

4.5.2 Compromise the client to gain access to cJWK

It is of utmost importance that the client protects cJWK from prying eyes. This may include device permissions, filesystem permissions, security frameworks (such as SELinux - Security-Enhanced Linux) or even the use of hardware encryption such as a TPM. How precisely this is accomplished depends on the client implementation.

4.5.3 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 file system permissions and the service's policy. An alternative implementation might use hardware cryptography (for example, an HSM) to protect the private key.

4.6 Building Tang

Tang is originally packaged for Fedora OS version 23 and later but we can build it from source of course. It relies on few other software libraries:

- http-parser 4.6.1
- systemd / xinetd 4.6.2
- jose 4.6.3
  - jansson 4.6.4
  - openssl 4.6.5
  - zlib 4.6.6

The steps to build it from source include 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
```

4.6.1 http-parser

Tang uses this parser for both parsing HTTP requests and HTTP responses. The parser can be found on its own GitHub [7].

4.6.2 systemd

systemd 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. **[[Why is systemd needed by tang]]**

4.6.3 José

José [8] 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) [**?**]
- RFC 7516 - JSON Web Encryption (JWE) [**?**]
- RFC 7517 - JSON Web Key (JWK) [**?**]
- RFC 7518 - JSON Web Algorithms (JWA) [**?**]
- RFC 7519 - JSON Web Token (JWT) [**?**]
- RFC 7520 - Examples of ... JOSE [**?**]
- RFC 7638 - JSON Web Key (JWK) Thumbprint [**?**]

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.

4.6.4 jansson

Jansson [6](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

4.6.5 OpenSSL

OpenSSL 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.

4.6.6 zlib

Library zlib [1] is used for data compression.

4.7 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
```

Now we are up and running. Server is ready to send advertisment on demand. **[[Get clevis in here?]]**

4.8 Clevis client

Clevis provides a pluggable key management framework for automated decryption. 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.

4.8.1 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.

4.8.2 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.

4.8.3 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).

4.8.4 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.

4.8.5 Dracut

The Dracut unlocker attempts to automatically unlock volumes during early boot. This permits automated root volume encryption. 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.

4.8.6 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 5

OpenWrt system

[[refactor chapter; add more info]] OpenWrt is is perhaps the most widely known Linux distribution for embedded devices especially for wireless routers. It was originally developed in January 2004 for the Linksys WRT54G with buildroot from the uClibc project. Now it supports many more models of routers. Installing OpenWrt system means replacing your router's built-in firmware with the Linux system which provides a fully writable filesystem with package management. This means that we are not bound to applications provided by the vendor. Router (the embedded device) with this distribution can be used for anything that an embedded Linux system can be used for, from using its SSH Server for SSH Tunneling, to running lightweight server software (e.g. IRC server) on it. In fact it allows us to customize the device through the use of packages to suit any application.

Today (May 2017) the stable 15.05.1 release of OpenWrt (code-named „Chaos Calmer“) released in March 2016 using Linux kernel version 3.18.23 runs on many routers.

[[LEDE]] Upstream info link...

5.1 OPKG Package manager

The opkg utility (Open Package Management System) is a lightweight package manager used to download and install OpenWrt packages. The opkg is fork of an ipkg (Itsy Package Management System). These packages could be stored somewhere on device's filesystem or the package manager will download them from local package repositories or ones located on the Internet mirrors. Users already familiar with GNU/Linux package managers like apt/apt-get, pacman, yum, dnf, emerge etc. will definitely recognize the similarities. It also has similarities with NSLU2's Optware, also made for embedded devices. Fact that OPKG is also a full package manager for the root file system, instead of just a way to add software to a seperate directory (e.g. /opt) includes the possibility to add kernel modules and drivers. OPKG is sometimes called Entware, but this is mainly to refer to the Entware repository for embedded devices.

Opkg attempts to resolve dependencies with packages in the repositories - if this fails, it will report an error, and abort the installation of that package.

Missing dependencies with third-party packages are probably available from the source of the package.

5.1.1 OPKG Makefile

Chapter 6

Software portability

Ideally, any software would be usable on any operating system, platform, and any processor architecture. Existence of term „porting“, derived from the Latin portāre which means „to carry“, proves that this ideal situation does not occurs that often, and acctual process of „carrying“ software to system with different environment is most probably required. Porting is process not documented and required if we want to run thing on another platform. Porting is also used to describe procces of converting computer games to became platform independent.

Software porting procces might be hard to distinguish with building software. The reason might be that in many cases building software on desired platform is enough, when software application does not work „out of the box“. This kind of behavior, an application that works immediately after or even without any special installation and without need for any configuration or modification, is ideal. It often happens when we copy application from one computer to another, not realizing that they have processors with same instructions set and same or very simmilar operating system therefore envoroment. But let us be realistic, it depends on many things, such as processor architecture to which was application written (compiled), quality of design, how application is meant to be deployed and of course application's source code.

Number of significantly different central processor units (CPUs), and operating systems used on the desktop or server is much smaller than in the past. However on embeded devices there are still much more various architectures available ARM MIPS RISC CISC x8664. **[[TODO argh]]**

Nowadays, the goal should be to develop software which is portable between preferred computer platforms (Linux, UNIX, Apple, Microsoft). If software is considered as not portable, it could not have to mean immediately that it is not possible, just that time and resources spent porting already written software are almost comparable, or even significantly higher than writing software as a whole from scratch. Effort spent porting some software product to work on desired platform must be little, such as copying already installed files to usb flash drive and run it on another computer. This kind of approach might most probably fail, due to not present dependencies of third party libraries not present on destination computer. Despite dominance of the x86 architecture there is usually a need to recompile software running, not only on different operating systems, to make sure we have all the dependencies present.

To simplify portability, even on distinguished processors with distant instruction sets, modern compilers translate source code to a machine-independent intermediate code. But

still, in the embedded system market, where OpenWrt belongs to, porting remains a significant issue [5].

6.1 OpenWrt's toolchain

Embeded devices are not meant for building on them because they does not have enough memory nor computation resorces as ordinary personal computers do(see Table 6.1). Building on such device would be time consuming and may result to overheating, which could cause hardware to fail. For this particular reason package building is done with cross-compiler.

Cross-compilation is done by set of tools called toolchain and it consists of:

- compiler
- linker
- a C standard library

For porting to „target“ system (OpenWrt) this toolchain has to be generated on „host“ system. The toolchain can be achieved in many different ways. The easiest way is undoubtedly to find a .rpm (or .deb) package and have it installed on our „host“ system. If a binary package with desired toolchain is not available for our system or is not available at all, there might be need to compile a custom toolchain from scratch.[1]

In case of OpenWrt we have available a set of Makefiles and patches called buildroot which is capable of generating toolchain.

6.1.1 Compiler

[[rewrite section compiler]]

In a nutshell compilers translate the high level programming language source code into lower level language such as machine understandable assembler instructions. To do so compiler is performing many operations starting with preprocessing.

Preprocessing supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis. However, some languages such as Scheme[2] support macro substitutions based on syntactic forms.

Lexical analysis, also known as lexing or tokenization, breaks the source code text into a sequence of small pieces called lexical tokens. Lexical analyzer does the scanning (cutting the input text into tokens and assign them a category) and the evaluating (converting tokens into a processed value). Common token categories may include:

- keywords (bound to programming language)
- identifiers (can not be keyword)
- separators
- operators
- literals

[1] tools like crosstool-ng (https://github.com/crosstool-ng/crosstool-ng) may help
[2] https://www.scheme.com/tspl4/

- comments

The set of token categories varies in different programming languages. The lexeme syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it.

Syntax analysis typically builds a parse tree structure according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.

Semantic analysis checks parse tree and does type checking, object binding and definite assignment. Output of semantic analysis is the symbol table or it results into rejecting incorrect programs or issuing warnings.

Code optimization, such as dead code elimination, inline expansion, constant propagation, loop transformation and even automatic parallelization, is done after analysis and it is independent on the CPU architecture.

Code generation is CPU dependent and does the transformation of intermediate language into the target machine language. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and generating machine instructions along with their associated addressing modes.

Cross-compiler is programming tool capable of creating executable file that is supposed to run on a „target“ architecture, in a similar or completely different enviroment, while working on a different „host“ architecture. It can also create object files used by linker. Reason for using cross-compiling might be to separate the build environment from target environment as well. OpenWrt toolchain uses gcc compiler, and it is one of the most important parts of toolchain.

6.1.2 Linker

[[refactor]]

With linker we are able to link compile's object files into single executable, library or object file. Linker can do static linking and dynamic linking.

Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is more portable, since it does not require the presence of the library on the system where it runs. Static linking also prevents „DLL Hell“, since each program includes exactly the versions of library routines that it requires, with no conflict with other programs. In addition, a program using just a few routines from a library does not require the entire library to be installed.

Dynamic linking is the process of postponing the resolving of some undefined symbols until a program is run. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these. Loading the program will load these objects/libraries as well, and perform a final linking. Dynamic linking needs no linker.

This approach offers two advantages:

Often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary. If a bug in a library function is corrected by replacing the library, all programs using it dynamically will benefit from the correction after restarting them. Programs that included this function by static linking would have to be re-linked first. There are also disadvantages:

Known on the Windows platform as „DLL Hell“, an incompatible updated library will break executables that depended on the behavior of the previous version of the library if the newer version is not properly backward compatible. A program, together with the libraries it uses, might be certified (e.g. as to correctness, documentation requirements, or performance) as a package, but not if components can be replaced. (This also argues against automatic OS updates in critical systems; in both cases, the OS and libraries form part of a qualified environment.)

6.1.3 C standard library

Most common C standard libraries are: GNU Libc, uClibc musl-libc, or dietlibc. They provide macros, type definitions and functions for tasks such as input/output processing (<stdio.h>), memory management (<stdlib.h), string handling (<string.h>), mathematical computations (<math.h>) and many more[3]. The OpenWrt's cross-compilation toolchain uses musl-libc.

6.2 OpenWrt's buildroot

OpenWrt's buildroot is a build system capable of generating toolchain, and also a root filesystem (also called sysroot), enviroment tightly bound to target. Build system can be configured for any device that is supported by OpenWrt.

The root filesystem in general is a mere copy of the file system of target's platform. In many cases just having the folders /usr and /lib would be sufficient, therefore we do not need to copy nor create the entire target file system on our host .

It is a good idea to store all these things, the toolchain and the root filesystem in a single place. With using OpenWrt's buildroot we will have this covered. Be tidy and pedantic, because cross-compiling can easily become a painful mess!

6.2.1 Buildroot prerequisites

Let us demonstrate what are minimum requirements of space and size of RAM (Random Access Memory) for building packages for Openwrt using its buildroot. For generating an installable OpenWrt firmware image file with a size of e.g. 8MB, we will need at least:

- ca. 200 MB for OpenWrt build system
- ca. 300 MB for OpenWrt build system with OpenWrt Feeds
- ca. 2.1 GB for source packages downloaded during build from the Feeds
- ca. 3-4 GB to build (i.e. cross-compile) OpenWrt and generate the firmware file
- ca. 1-4 GB of RAM to build Openwrt.(build x86's img need 4GB RAM)

These are specifications of embedded device TL-WR842Nv3 a regular wireless router manufactured by TP-LINK which was used to test all packages related to Tang server porting effort:

The actual sum of space required only for buildroot to work correctly which is about 6.4 GB compared to available storage space on wireless router should be demonstrative why is

[3] https://en.wikipedia.org/wiki/C_standard_library#Header_files

Model	TL-WR842N(EU)
Version	v3
Architecture:	MIPS 24Kc
Manufacturer:	Qualcomm Atheros
Bootloader:	U-Boot
System-On-Chip:	Qualcomm Atheros QCA9531-BL3A
CPU Speed:	650 MHz
Flash chip:	Winbond 25Q128CS16
Flash size:	16 MiB
RAM chip:	Zentel A3R12E40CBF-8E
RAM size:	64 MiB
Wireless:	Qualcomm Atheros QCA9531
Antenae(s):	2 non-removable
Ethernet:	4 LAN, 1 WAN 10/100
USB:	1 x 2.0
Serial:	?

Table 6.1: TL-WR842Nv3 specifications

building for embedded devices done with cross-compiling. The another reasons, not to just compare internal storage which might be extended (as shown on apendix TODOapendix), are device's minimalistic RAM size and not that „powerfull" CPU.[[**add more about CPU**]]

6.3 Prepare the enviroment

Before getting to know buildroot we have to have host enviroment ready for buildroot in a way that we will install buildroot's dependencies and then buildroot itself.

6.3.1 Buildroot's dependencies

To be able to work with OpenWrt's bildroot we will need lots of buildroot dependencies first. Please note that following commands have to be run by user with root privileges to install buildroot dependencies on Fedora 27 system:

```
sudo dnf install binutils bzip2 gcc gcc-c++ \
gawk gettext git-core flex ncurses-devel ncurses-compat-libs \
zlib-devel zlib-static make patch unzip perl-ExtUtils-MakeMaker \
perl-Thread-Queue glibc glibc-devel glibc-static quilt sed \
sdcc intltool sharutils bison wget openssl-devel
```

Listing 6.1: Buildroot dependencies installation on Fedora 27

Some feeds might be not available over git but only via other versioning tools like subversion (svn) or mercurial. If we want to obtain their source-code, we need to install svn and mercurial as well:

```
sudo dnf install subversion mercurial
```

Listing 6.2: Other versioning tools installation on Fedora 27

6.3.2 Getting buildroot

There are mirrors containg OpenWrt SDK(Software development kit) buildroots for every released version of OpenWrt system. It is good to consider using OpenWrt's SDK in order to build software application to only one specific release of target system. For example when we are using „stable“ release of OpenWrt 15.05.1 with codename „Chaos Calmer“ on TL-WR842N(EU) device, we should probably end up in in „Supplementary Files“ section of the OpenWrt archives [4] looking for SDK.

If we have host platform which aims to be on the bleeding edge of open-source technologies, such as OS Fedora, we will probably encounter issues with dependencies. These issues are apearing because an older version of the dependencies are required for this old SDK to work and might be not available for our host platform anymore. In this case we would have to install or even build older version of required packages to our host system.

When we start with porting an upstream projet (such as Tang) for latest release of target's sytem on Fedora host system, an upstream bleeding edge buildroot should be the best solution. In our case we will work with latest OpenWrt release LEDE 17.01.4 on our target device, and download the OpenWrt bleeding edge (trunk version) buildroot with git. To download OpenWrt buildroot from upstream repository run command:

```
git clone https://github.com/openwrt/openwrt.git ~\buildroot-openwrt
```

Listing 6.3: Cloning upstream OpenWrt buildroot

6.4 Working with buildroot

After we have enviroment ready to be worked on (all needed dependencies installed and buildroot downloaded on host) let us just remember these simple 4 rules to not break our enviroment in any way.

1. Do everything in buildroot as non-root user!
2. Issue all OpenWrt build system commands in the <buildroot> directory, e.g. ~/buildroot-openwrt/
3. Do not build in a directory that has spaces in its full path.
4. Change ownership of the directory where we downloaded the OpenWrt to other than root user

It is good to set upstream...
describe buildroot topology and descibe what every single thing is what for
make will trigger making all packages not like in sdk
difference working with sdk and buildroot that sdk build only packages and buildroot can buld only package but it is designed to build entire image
for specified architecture respectively device from list
do not know how to add there new device but may provide link
configured with make menuconfig shown here firstly select architecture and configuration will be generated into .config :
in sdk .config is pre generated and make menuconfig (need to find out if possible)

[4] https://archive.openwrt.org/chaos_calmer/15.05.1/ar71xx/generic/

on sdk make will trigger only building packages in directory package and it has to be done right way

it is also possible to do this with buildroot but is better to have upstream or own packages there in right directory

screen

make menuconfig is to select all packages which we want to build todo findout if it is neccessary to buld base packages for selected

```
$ make
$ feeds install -a
```

tm tmp is needed if package directory has been edited to regenerate metadata of what (todo find out)

it is better to know what we are installig with feeds install and to choose only required packages for one newly choosed package to build

6.5 Makefiles for OpenWrt's packages

This is very important part of creating new packages therefore porting them to openwrt this file is similar to specfiles on fedora and

```
#include<stdio.h>
#include<iostream>
int main(void)
{
    printf("Chello World\n");
    // comment
    return 0;
}
```

Listing 6.4: Basic C code.

Chapter 7

Porting Tang dependencies

Programs don't run in a vacuum but they interface with the outside world. The view of this outside world differs from environment to environment. Things like hostnames, system resources, and local conventions could be different.

When we start porting a code to a specific target platform, in our case OpenWrt, it is likely that we will face the first problem: satisfying missing dependencies. This problem is easy to solve in principle, but can easily mess things up to a level we wouldn't imagine.

If the code depends on some library that is NOT in the root filesystem, there's no way out but to find them somewhere, somehow. Dependencies can be satisfied in two ways: with static libraries or with shared libraries. If we are lucky, we could find a binary package providing what we need (i.e. the library files and the header files), but most often we will have to cross-compile the source code on our own. Either ways, we end up with one or more binary files and a bunch of header files. Where to put them? Well, that depends. There are a few different situations that can happen, but basically everything reduces to having dependencies in buildroot's root filesystem or in a different folder.

Having dependencies in a different folder could be an interesting solution to keep the libraries that we cross-compiled on our own separated from the other libraries (for example, the system libraries). We can do that if we want but if we do, we must remember to provide to compiler and linker programs with the paths where header files and binary files can be found. With static libraries, this information are only needed at compile and linking time, but if we are using shared libraries, this won't suffice. We must also specify where these libraries can be found at run time.

If we are satisfying the dependencies with shared libraries (.so files) having dependencies in root filesystem is probably the most common solution (and maybe, the best solution). Remember that when everything will be up and running, these libraries must be installed somewhere in the file system of the target platform. So there is a natural answer to the question above: install them in the target's root filesystem, for example in /usr/lib (the binary shared files) and /usr/include (the header files) or in any other path that allow the loader to find those libraries when the program executes. Do not forget to install them in the file system of the actual target machine, in the same places, in order to make everything work as expected. Please note that static libraries ('.a' files) does not need to be installed in the target file system since their code is embedded in the executable file when we cross-compile a program. In case of OpenWrt we will use this approach.

7.1 Find the depencies

To port Tang to OpenWrt system we have have all its dependencies available and installed in buildroot. First thing we should do is some digging and find out if dependencies for our software, we are about to port, are available for target's platform. All packages available for OpenWrt can be found in its github repositories. Let us remember Tang's dependencies first:

- http-parser 4.6.1
- systemd's socket activation4.6.2
- José 4.6.3
  - jansson 4.6.4
  - openssl 4.6.5
  - zlib 4.6.6

After some digging we will find out that the OpenWrt system has already packages openssl, zlib, http-parser, and jansson. Packages openssl and zlib are in versions already sufficient for Tang to get things work.

Package http-parser was little bit tricky. At first try there is a chance that we will not find this package in OpenWrt's repository. The reason might be that we will try to find a exactly „http-parser“ string in OpenWrt's GitHub repository openwrt/openwrt only. First of all, we must not forget that feeds of many packages are living in repository packages nested under openwrt organization. Secondly after searching repository openwrt/packages for http-parser we will find out that it is named as libhttp-parser and in version 2.2.3. This, let us say naming convention, is really common in Linux and could be predictable as the http-parser is in fact only a library. One way or another compared to fedora packages it needs to be updated.

Jansson package was only available in version 2.7 which is too outdated for Tang's dependency José because it require at least jannson version 2.9. We shall not forget that there is a need for updating jansson package for OpenWrt platform as well so we will do updates first.

Packages José, Tang and systemd are not listed in OpenWrt's packages. Porting of the systemd would be huge effort but tang's requirements are minimal and we should be able to work with xinetd's socket activation. Finally, as José and Tang are not listed in packages they will require to add them into package feeds to port Tang itself.

7.2 Update outdated packages

Finding that some of dependencies are already available on our desired target platform will definitelly make us satisfied. We would agree that starting with something already built for OpenWrt is the best thing to do when you are approaching unknown platform. In following subsections we will use all things we know from section 6.4 Working with buildroot. Let us start with missing update of José's dependency, package jansson.

7.2.1 Update jansson

7.2.2 Update http-parser

7.3 Create new packages

7.3.1 Create José

Chapter 8

Porting Tang

8.1 Install dependencies

update feeds feeds install make menuconfig

8.2 Socket activation with xinetd

there is no systemd for OpenWrt need to configure socket activation because tang is not proggramed to use nor create its own sockets

8.2.1 How

to have an example created simple application in C for xinetd

8.3 Cross-compile Tang

do not forget to add xinetd in Dependencies

8.3.1 Concealing http-parser

without upstream patch not working tang

8.3.2 Mysterious José

does makefiles support shell a,b thing???

8.4 Configure Tang with xinetd

/etc/services
: xinetd configuration

Chapter 9

Contribute your work

Everyone can profit from our work...

9.1 Make it look easy

Be tidy create nice files and patch files to be used.

9.2 Follow contribution guide

[[mailing lists]]
    [[github - how]]

Chapter 10

Conclusion

The Tang 4 server is a very lightweight program. It provides secure and anonymous data binding using McCallum-Relyea exchange 4.1 algorythm.

As every server purpose is to serve its clients, it needs to have client application. In case of Tang we have Clevis. Clevis 4.8 is a client software with full support for Tang. It has minimal dependencies and it is possible to use with HTTP, Escrow 3, and it implements Shamir Secret Sharing. Clevis has GNOME integration so it is not only a command line tool. Clevis also supports removable devices unocking using UDisks2 or even early boot integration with dracut, which was this thesis inspiration and goal to achieve with only embedded device supported OpenWrt running Tang server.

To port Tang to OpenWrt system it was necesarry to port all its dependencies first. The OpenWrt system has already package openssl, zlib, and jansson but only version 2.7 which was too old. So there was a need for updating jansson to resolve all dependencies for package José. José required porting and after focusing on upstream version porting on this package was straightforward. After struggling with older version, package http-parser known as libhttp-parser in OpenWrt feeds, is now updated to latest upstream version 2.8.0. The systemd would be huge effort but tang's requirements are minimal and we were able to work with xinetd's socket activation. With correct configuration of xinetd and removing dependency for systemd Tang server is running on OpenWrt with some platform specific changes mentioned in chapter 8 Porting Tang.

Bibliography

[1] Adler, M.: *zlib*. [Online] Accessed 1 May 2017.
Retrieved from: https://github.com/madler/zlib

[2] Bauer, J.: *LUKS In-Place Conversion Tool*. [Online] Accessed 3 May 2017.
Retrieved from: http://www.johannes-bauer.com/linux/luksipc/

[3] Bressers, J.: *Security: Everything is on fire!* [Online] Accessed 1 May 2017.
Retrieved from: https://youtu.be/zmDm7J7V7aw?list=PLjT7F8YwQhr--MZrcojlv2lpeBYUlQFkl&t=1058

[4] Fruhwirth, C.: *LUKS On-Disk Format Specification*. [Online] Accessed 2 May 2017.
Retrieved from: https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf

[5] Lehey, G.: *Porting UNIX Software: From Download to Debug*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.. 1995. ISBN 1-56592-126-7.

[6] Lehtinen, P.: *jansson*. [Online] Accessed 1 May 2017.
Retrieved from: https://github.com/akheron/jansson

[7] McCallum, N.: *HTTP-parser*. [Online] Accessed 1 May 2017.
Retrieved from: https://github.com/nodejs/http-parser

[8] McCallum, N.: *jose*. [Online] Accessed 1 May 2017.
Retrieved from: https://github.com/latchset/jose

[9] SplashData: *Worst Passwords of 2016*. [Online] Accessed 15 May 2017.
Retrieved from:
https://www.teamsid.com/worst-passwords-2016/?nabe=4587092537769984:2,6610887771422720:1&utm_referrer=https%3A%2F%2Fwww.google.cz%2F

[10] Steinbeck John, e. b. S. S.; Benson., J. J.: *Of men and their making*. London: Allen Lane The Penguin Press. 2002. ISBN 07-139-9622-6.

[11] Thales: *Global Encryption Trends Study*. [Online] Accessed 1 May 2017.
Retrieved from:
http://images.go.thales-esecurity.com/Web/ThalesEsecurity/{5f704501-1e4f-41a8-91ee-490c2bb492ae}_Global_Encryption_Trends_Study_eng_ar.pdf

[12] Wakefield, R. L.: Network Security and Password Policies. *The CPA Journal*. vol. 74, no. 7. 07 2004: pp. 6–6,8. copyright - Copyright New York State Society of Certified Public Accountants Jul 2004; Last updated - 2011-07-20;

SubjectsTermNotLitGenreText - United States; US.
Retrieved from:
https://search.proquest.com/docview/212314970?accountid=17115

[13] Wikipedia: *Free On The Fly Encryption*. [Online] Accessed 3 May 2017.
Retrieved from: https://en.wikipedia.org/wiki/FreeOTFE

Appendix A

Disk content

Appendix B

Hard Disk Encryption With LUKS

B.1 Fedora 25 - disc encryption option selecting

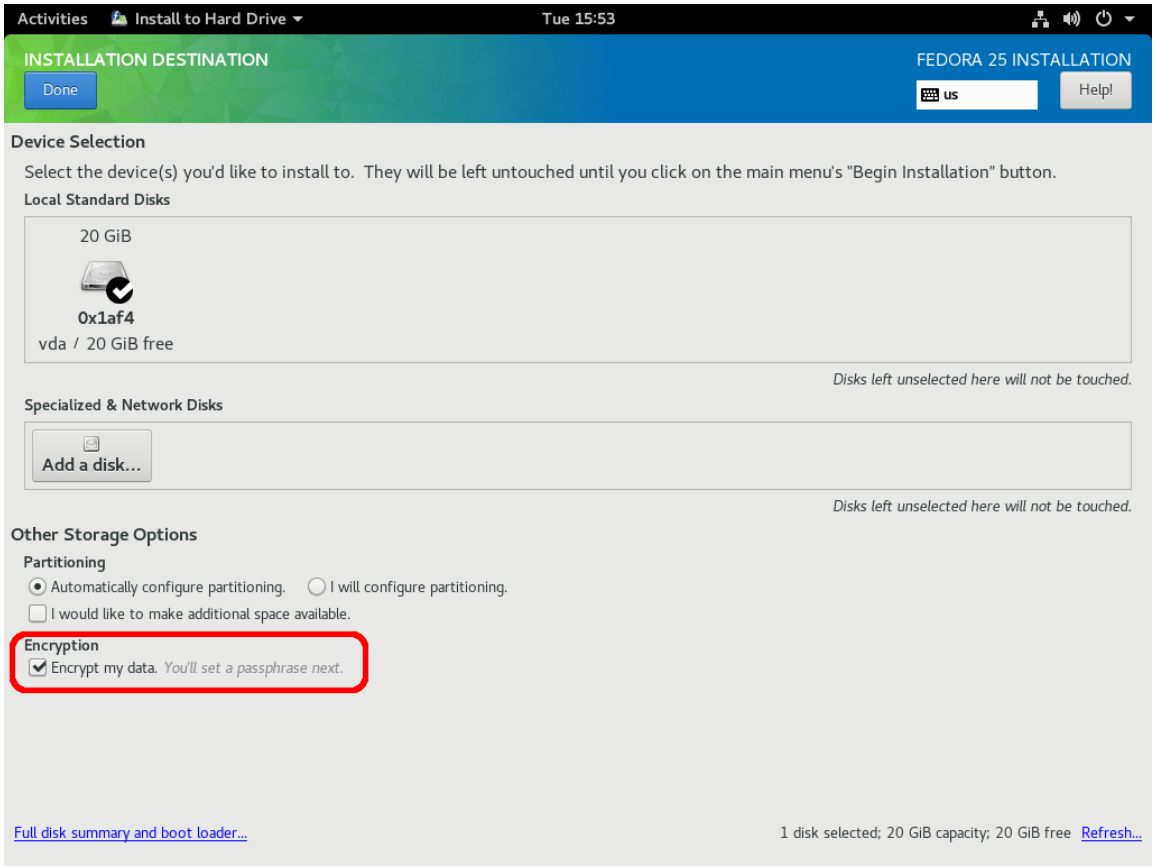

Figure B.1: Checking option

B.2 Fedora 25 - determination key encryption key

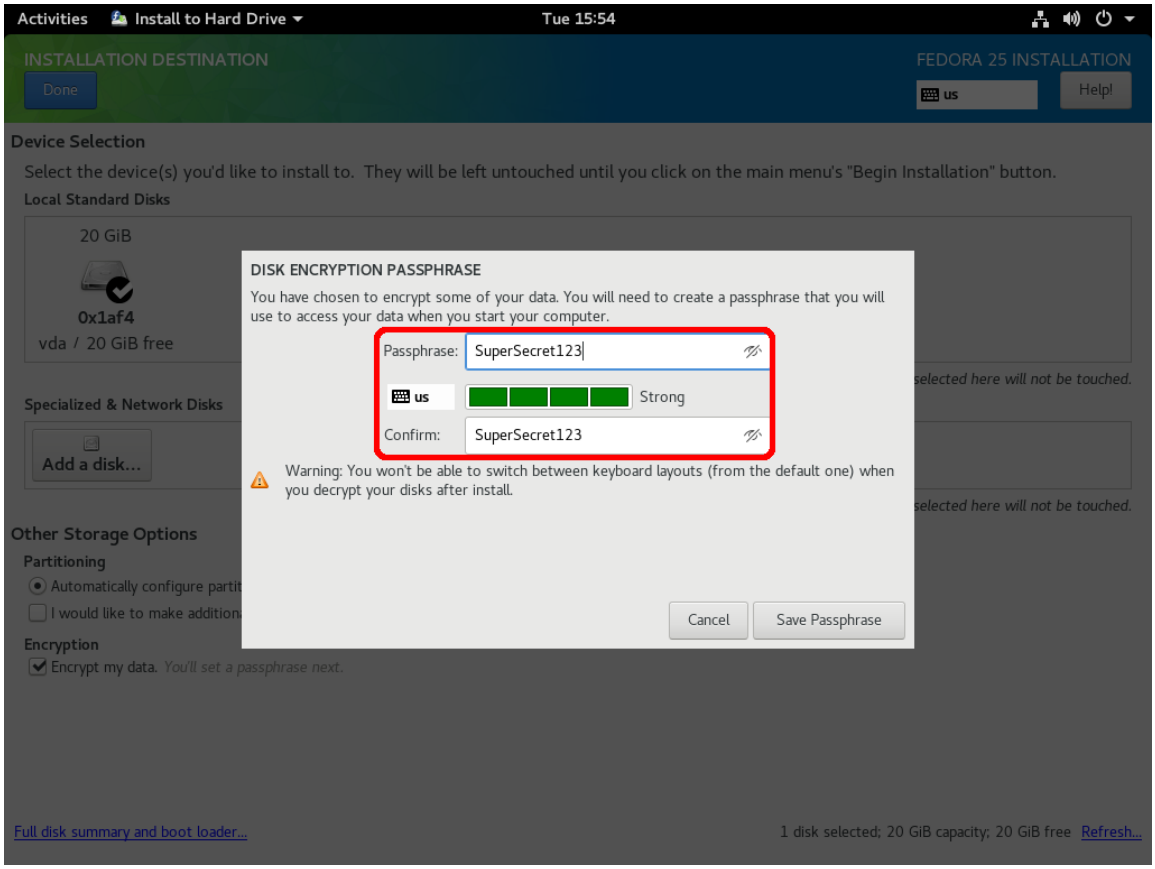

Figure B.2: Determining key

Appendix C

LUKS In Place Encryption

It takes 4 steps to perform an in place encryption with *luksipc* :

1. Unmounting the filesystem
2. Resizing the filesystem to shrink about 10 megabytes (2048 kB is the current LUKS header size – but do not trust this value, it has changed in the past!)
3. Performing luksipc
4. Adding custom keys to the LUKS keyring

C.1 Step 1 - unmounting

There should not be any problems unmounting partition, unless you want to encrypt / - the root partition, which in our case (to lock whole disk) will be necessary. To do so we need to restart our computer and boot any other or live distribution capable of completing these next steps.

```
# umount /dev/vda2
```

C.2 Step 2 - resizing

There are plenty tools for re-sizing, essentially for partitioning as whole (fdisk, e2fsck, etc.). Demonstrating how this is done for ext 2, 3, 4 here:

```
# e2fsck /dev/vda2
# resize2fs /dev/vda2 -s -10M
```

Delete and recreate shrinked partition with fdisk:

```
# fdisk /dev/vda Welcome to fdisk (util-linux 2.23.2).
Changes will remain in memory only, until you decide to write them.  Be careful
before using the write command.
Command (m for help):
```

Check the partition number:

```
Command (m for help):  p Disk /dev/vda:  407.6 GiB, 437629485056 bytes, 854745088
sectors Units:  sectors of 1 * 512 = 512 bytes Sector size (logical/physical):
512 bytes / 4096 bytes I/O size (minimum/optimal):  4096 bytes / 4096 bytes Disklabel
type:  dos Disk identifier:  0x5c873cba Partition 2 does not start on physical
sector boundary.
```

```
Device Boot Start End Blocks Id System /dev/vda1 * 2048 1026047 512000 83
Linux /dev/vda2 1026048 1640447 307200 8e Linux LVM
```

C.3 Step 3 - encrypting

After this, luksipc comes into play. It performs an in-place encryption of the data and prepends the partition with a LUKS header. Firt we have to download luksipc or install it with package manager.

```
$ wget https://github.com/johndoe31415/luksipc/archive/master.zip
$ unzip master.zip
$ cd luksipc-master/
$ make
```

Now run it with parameters like:

```
# ./luksipc -d /dev/vda2
```

luksipc will have created a key file /root/initial_keyfile.bin that you can use to gain access to the newly created LUKS device:

```
# cryptsetup luksOpen -key-file /root/initial_keyfile.bin /dev/vda2 fedoradrive
```

C.4 Step 4 - adding key

DO NOT FORGET to add key to LUKS volume:

```
# cryptsetup luksAddKey -key-file /root/initial_keyfile.bin /dev/vda2
```