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

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. abstract.en

Keywords

Kľúčové slová

portovanie, Tang, server, OpenWrt, operačný systém, vstavaný, Clevis, klient, Escrow, šifrovanie, LUKS, pevný disk, diskový oddiel, šifrovací kľúč, automatizácia, cross-kompilácia, balíkový systém keywords.en

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

. . . .
Tibor Dudlák
February 5, 2018

Contents

Chapter 1

Introduction

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

John Steinbeck[14]

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 10, 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 11. Another solution is server *Tang*, which principles are mentioned in chapter 12. 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 12.6 to *OpenWrt* system for *embedded* devices mentioned in chapter 13. 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 . . .

Chapter 2

Hard drive encryption

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 [15].

2.2 Hard drive encryption

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 10.1. 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.

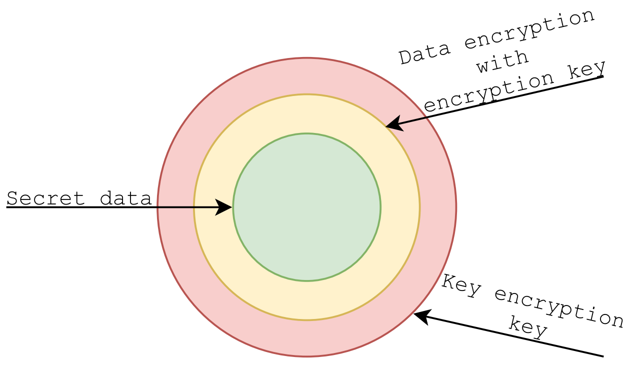


Figure 2.1: How we encrypt data

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 [16]. 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 [13], 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 11 model works.

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.

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

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]]

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 [17] 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 10.2. 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.

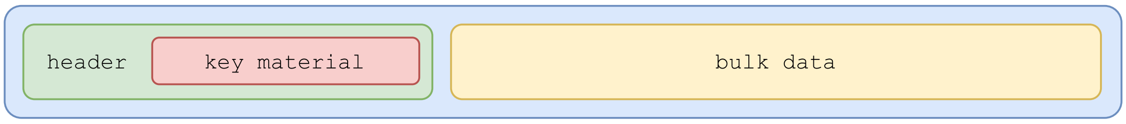


Figure 2.2: LUKS volume structure

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 10.3.

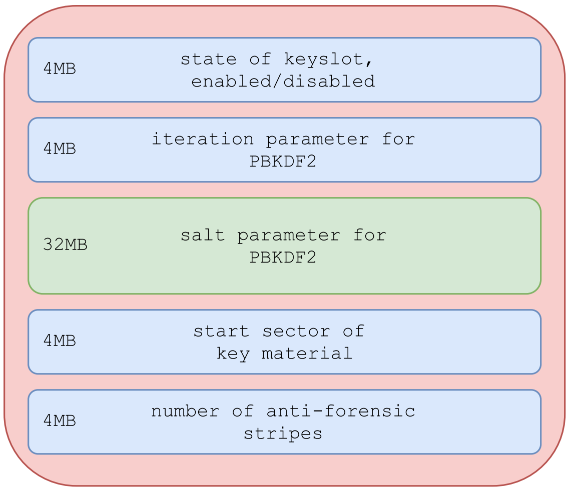

4MB
state of keyslot, enabled/disabled
4MB
iteration parameter for PBKDF2
32MB
salt parameter for PBKDF2
4MB
start sector of key material
4MB
number of anti-forensic stripes

Figure 2.3: LUKS Key Slot

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 E.

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 F.

Chapter 3

Key escrow server

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 11.1 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.

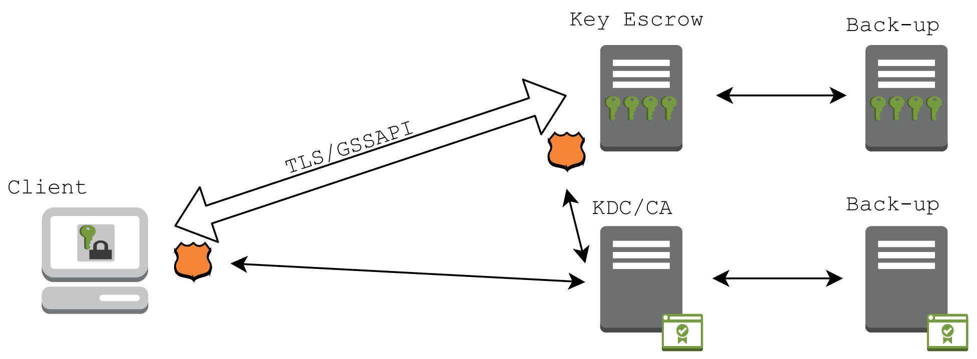


Figure 3.1: Escrow model

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. [[**refactor chapter; add more info**]]

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 12.6.3 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.

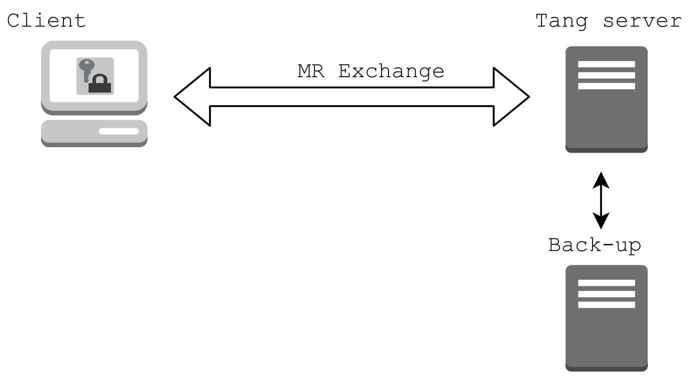


Figure 4.1: Tang model

On Figure 12.1 you can see the Tang model. It is very similar to Escrow model 11.1 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 12.1 as described below.

4.2 Binding with Tang

A client performs an ECDH key exchange using the McCallum-Relyea algorithm 12.1 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. **[[ARROWS]]**

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 12.6.1
- systemd / xinetd 12.6.2
- jose 12.6.3
  - jansson 12.6.4
  - openssl 12.6.5
  - zlib 12.6.6

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 [11].

4.6.2 systemd / xinetd

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 [12] 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) [7]
- RFC 7516 - JSON Web Encryption (JWE) [5]
- RFC 7517 - JSON Web Key (JWK) [6]
- RFC 7518 - JSON Web Algorithms (JWA) [9]
- RFC 7519 - JSON Web Token (JWT) [8]
- RFC 7520 - Examples of ... JOSE
- RFC 7638 - JSON Web Key (JWK) Thumbprint [6]

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 [10](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 a 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.

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

Porting procedure

[[TODO chapter]]

llll

[[Architecture]]

mips

[[Makefiles]]

spec

Chapter 7

Contributing

[[TODO chapter]]

Z

[[mailing lists]]

A

[[github]]

Chapter 8

Conclusion

The Tang 12 server is a very lightweight program. It provides secure and anonymous data binding using McCallum-Relyea exchange 12.1 algorythm.

Clevis 12.8 is a client software with full support for Tang. It has minimal dependencies and it is possible to use with HTTP, Escrow 11, and it implements Shamir Secret Sharing. Clevis has GNOME integration so it is not only a command line tool. Clevis also supports early boot integration with dracut or even removable devices unocking using UDisks2.

To port Tang to OpenWrt system it will be necesarry to port all its dependencies first. The OpenWrt system has already package openssl, zlib, and jansson but only version 2.7 which is too old. So there will be a need for porting jansson. José will require porting and http-parser too. The systemd would be huge effort but tang's requirements are minimal and we should be able to work with xinetd. Finally, some work will be required to port Tang itself. **[[LEDE, contribution, porting]]**

Chapter 9

Introduction

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

John Steinbeck[14]

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 10, 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 11. Another solution is server *Tang*, which principles are mentioned in chapter 12. 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 12.6 to *OpenWrt* system for *embedded* devices mentioned in chapter 13. 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 . . .

Chapter 10

Hard drive encryption

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.

10.1 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 [15].

10.2 Hard drive encryption

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 10.1. 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.

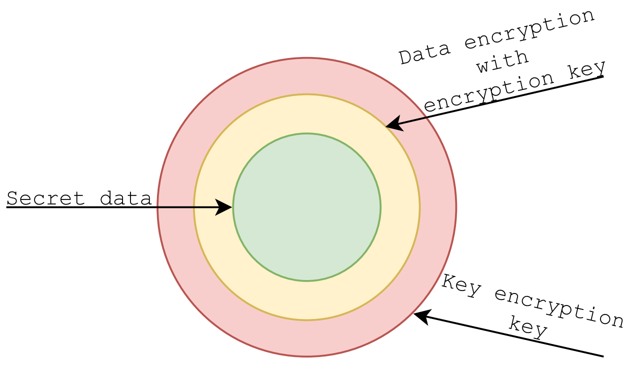


Figure 10.1: How we encrypt data

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 [16]. 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 [13], 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 11 model works.

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.

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

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.

10.2.1 LUKS

[[master key]]

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 [17] 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 10.2. 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.

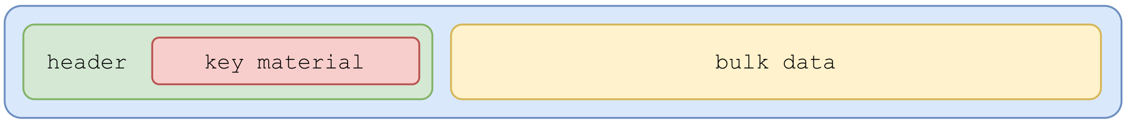


Figure 10.2: LUKS volume structure

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 10.3.

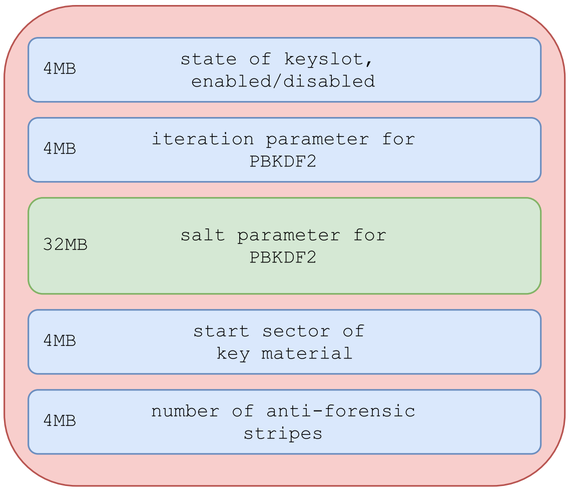

4MB
state of keyslot, enabled/disabled
4MB
iteration parameter for PBKDF2
32MB
salt parameter for PBKDF2
4MB
start sector of key material
4MB
number of anti-forensic stripes

Figure 10.3: LUKS Key Slot

10.2.2 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 E.

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 F.

Chapter 11

Key escrow server

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.

11.1 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 11.1 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.

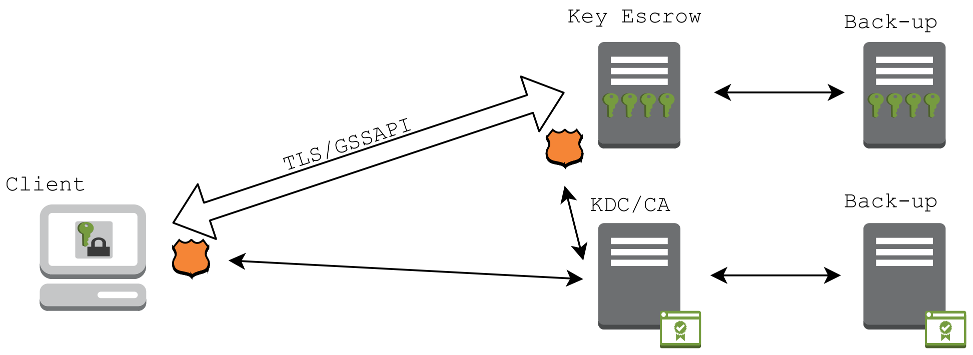


Figure 11.1: Escrow model

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. [[**refactor chapter; add more info**]]

Chapter 12

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.

12.1 Tang - binding daemon

Tang server advertises asymmetric keys and a client is able to get the list of these signing keys 12.6.3 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.

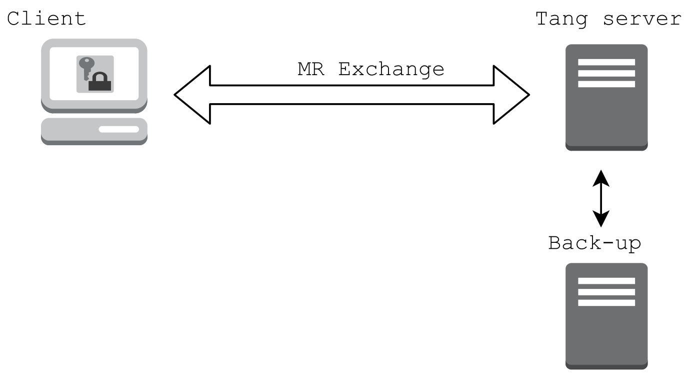


Figure 12.1: Tang model

On Figure 12.1 you can see the Tang model. It is very similar to Escrow model 11.1 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 12.1 as described below.

12.2 Binding with Tang

A client performs an ECDH key exchange using the McCallum-Relyea algorithm 12.1 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. **[[ARROWS]]**

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

Table 12.1: McCallum-Relyea exchange

12.3 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.

$$s = g * S \tag{12.1}$$

$$c = g * C \tag{12.2}$$

$$K = s * C \tag{12.3}$$

12.4 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:

$$e = g * E \tag{12.4}$$

$$x = c + e \tag{12.5}$$

$$y = x * S \tag{12.6}$$

$$z = s * E \tag{12.7}$$

$$K = y - z \tag{12.8}$$

12.5 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 12.2: 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).

12.5.1 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.

12.5.2 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.

12.5.3 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.

12.6 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 12.6.1
- systemd / xinetd 12.6.2
- jose 12.6.3
  - jansson 12.6.4
  - openssl 12.6.5
  - zlib 12.6.6

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
```

12.6.1 http-parser

Tang uses this parser for both parsing HTTP requests and HTTP responses. The parser can be found on its own GitHub [11].

12.6.2 systemd / xinetd

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]]**

12.6.3 José

José [12] 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) [7]
- RFC 7516 - JSON Web Encryption (JWE) [5]
- RFC 7517 - JSON Web Key (JWK) [6]
- RFC 7518 - JSON Web Algorithms (JWA) [9]
- RFC 7519 - JSON Web Token (JWT) [8]
- RFC 7520 - Examples of ... JOSE
- RFC 7638 - JSON Web Key (JWK) Thumbprint [6]

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.

12.6.4 jansson

Jansson [10](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

12.6.5 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.

12.6.6 zlib

Library zlib [1] is used for data compression.

12.7 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?]]**

12.8 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.

12.8.1 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.

12.8.2 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.

12.8.3 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).

12.8.4 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.

12.8.5 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.

12.8.6 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 13

OpenWrt system

[[refactor chapter; add more info]] OpenWrt is a 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.

13.1 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.

13.1.1 OPKG Makefile

Chapter 14

Porting procedure

[[TODO chapter]]
llll
[[Architecture]]
mips
[[Makefiles]]
spec

Chapter 15

Contributing

[[TODO chapter]]

Z

[[mailing lists]]

A

[[github]]

Chapter 16

Conclusion

The Tang 12 server is a very lightweight program. It provides secure and anonymous data binding using McCallum-Relyea exchange 12.1 algorythm.

Clevis 12.8 is a client software with full support for Tang. It has minimal dependencies and it is possible to use with HTTP, Escrow 11, and it implements Shamir Secret Sharing. Clevis has GNOME integration so it is not only a command line tool. Clevis also supports early boot integration with dracut or even removable devices unocking using UDisks2.

To port Tang to OpenWrt system it will be necesarry to port all its dependencies first. The OpenWrt system has already package openssl, zlib, and jansson but only version 2.7 which is too old. So there will be a need for porting jansson. José will require porting and http-parser too. The systemd would be huge effort but tang's requirements are minimal and we should be able to work with xinetd. Finally, some work will be required to port Tang itself. **[[LEDE, contribution, porting]]**

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] Jones, e. a.: *JSON Web Encryption*. [Online] Accessed 1 May 2017.
Retrieved from:
http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption

[6] Jones, e. a.: *JSON Web Keys*. [Online] Accessed 1 May 2017.
Retrieved from: http://tools.ietf.org/html/draft-ietf-jose-json-web-key

[7] Jones, e. a.: *JSON Web Signing*. [Online] Accessed 1 May 2017.
Retrieved from:
http://tools.ietf.org/html/draft-ietf-jose-json-web-signature

[8] Jones, e. a.: *JSON Web Tokens*. [Online] Accessed 1 May 2017.
Retrieved from:
https://tools.ietf.org/html/draft-ietf-oauth-json-web-token

[9] Jones, e. a.: *JWT Authorization Grants*. [Online] Accessed 1 May 2017.
Retrieved from: http://tools.ietf.org/html/draft-ietf-oauth-jwt-bearer

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

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

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

[13] 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

[14] 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.

[15] 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

[16] 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

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

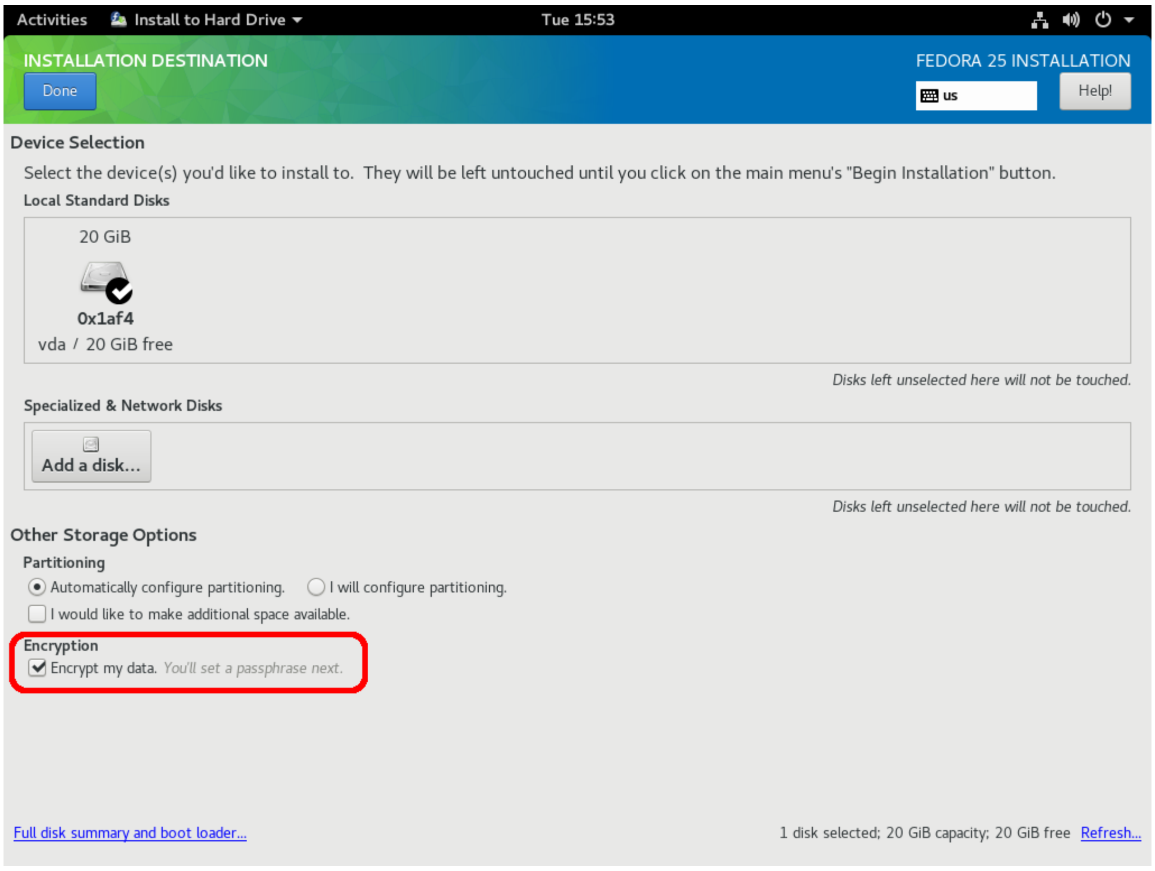
Appendix A

Disk content

Appendix B

Hard Disk Encryption With LUKS

B.1 Fedora 25 - disc encryption option selecting



Activities | Install to Hard Drive | Tue 15:53

INSTALLATION DESTINATION

Done

FEDORA 25 INSTALLATION

us

Help!

Device Selection

Select the device(s) you'd like to install to. They will be left untouched until you click on the main menu's "Begin Installation" button.

Local Standard Disks

20 GiB

0x1af4

vda / 20 GiB free

Disks left unselected here will not be touched.

Specialized & Network Disks

Add a disk...

Disks left unselected here will not be touched.

Other Storage Options

Partitioning

[✓ Automatically configure partitioning.] I will configure partitioning.

I would like to make additional space available.

Encryption

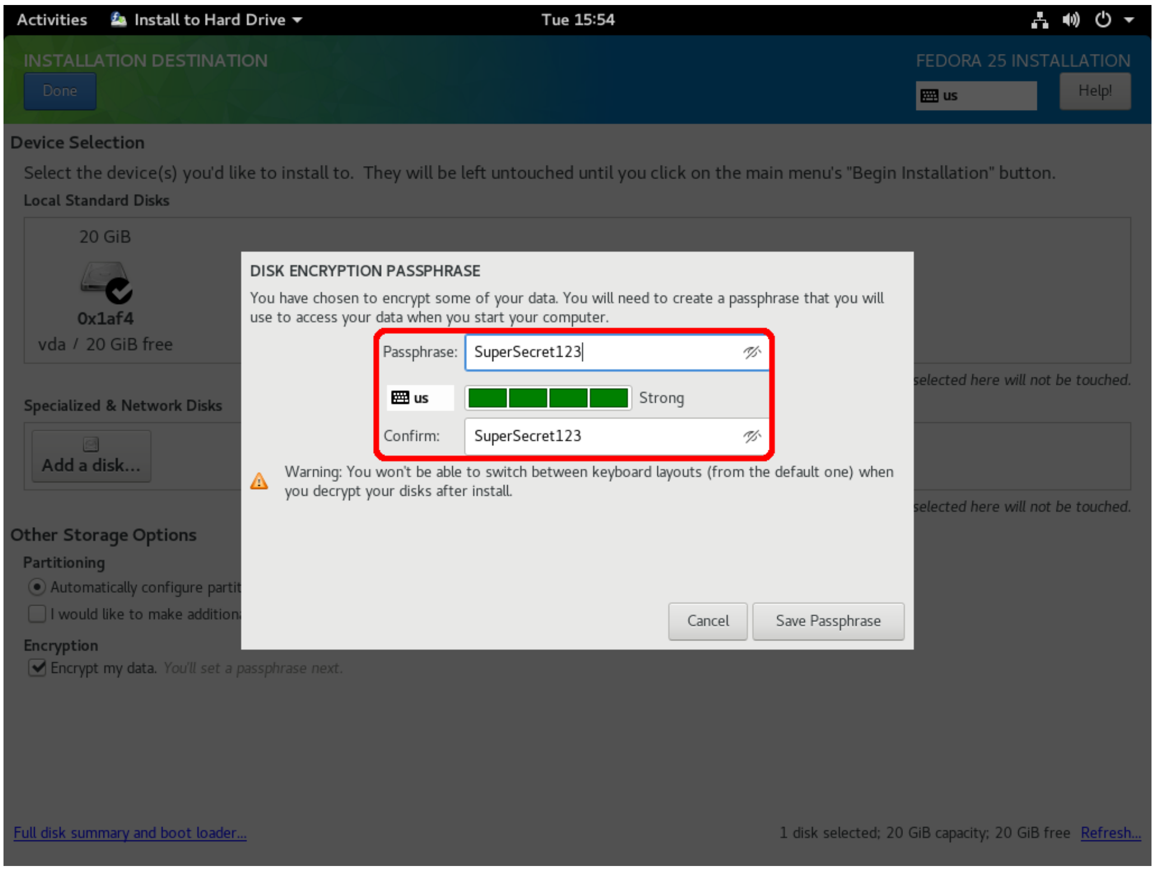
[✓ Encrypt my data.] *You'll set a passphrase next.*

Full disk summary and boot loader...

1 disk selected; 20 GiB capacity; 20 GiB free Refresh...

Figure B.1: Checking option

B.2 Fedora 25 - determination key encryption key


Activities   Install to Hard Drive   Tue 15:54

INSTALLATION DESTINATION   Done

FEDORA 25 INSTALLATION   us   Help!

Device Selection

Select the device(s) you'd like to install to. They will be left untouched until you click on the main menu's "Begin Installation" button.

Local Standard Disks

20 GiB
0x1af4
vda / 20 GiB free

...selected here will not be touched.

Specialized & Network Disks

Add a disk...

...selected here will not be touched.

Other Storage Options

Partitioning

Automatically configure partit...

I would like to make addition...

Encryption

Encrypt my data. *You'll set a passphrase next.*

DISK ENCRYPTION PASSPHRASE

You have chosen to encrypt some of your data. You will need to create a passphrase that you will use to access your data when you start your computer.

Passphrase: SuperSecret123

us   Strong

Confirm: SuperSecret123

Warning: You won't be able to switch between keyboard layouts (from the default one) when you decrypt your disks after install.

Cancel   Save Passphrase

Full disk summary and boot loader...   1 disk selected; 20 GiB capacity; 20 GiB free   Refresh...

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

[[vyskusat doplnit...]]

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
```

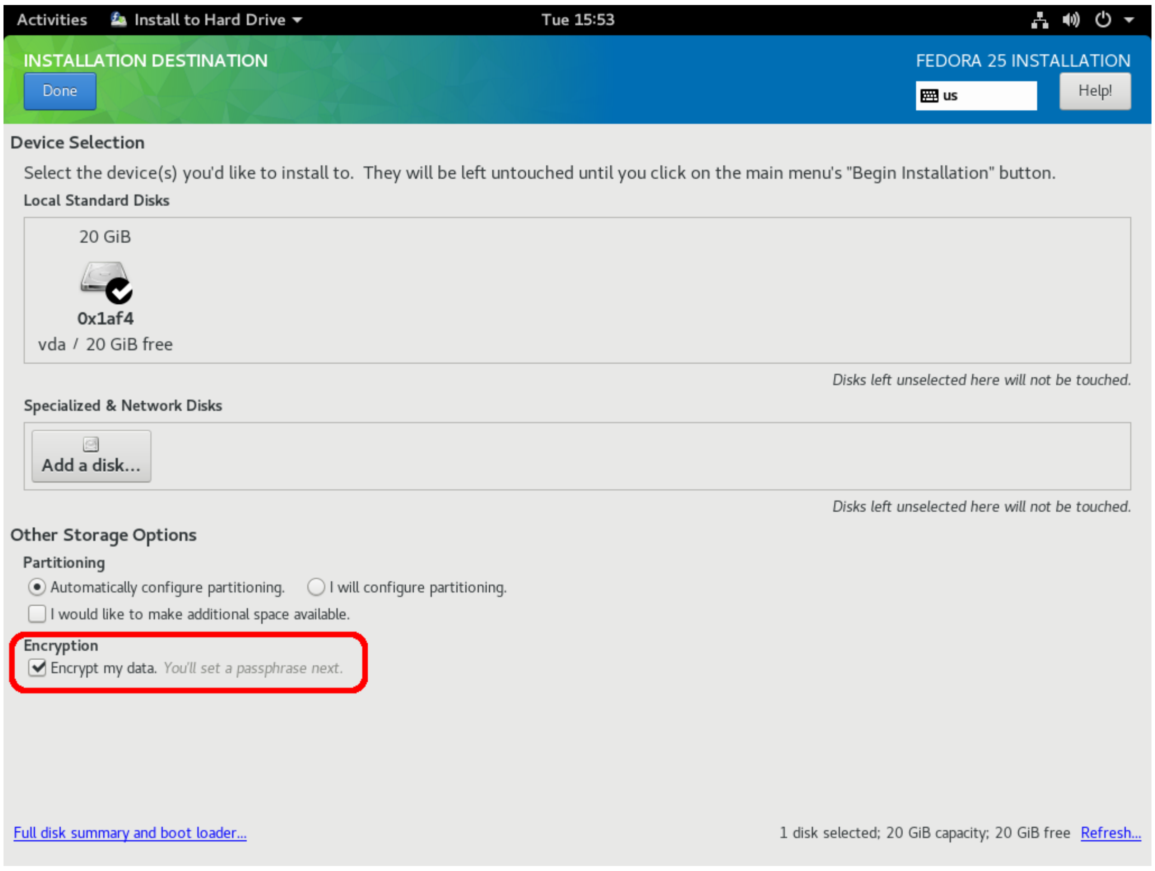
Appendix D

Disk content

Appendix E

Hard Disk Encryption With LUKS

E.1 Fedora 25 - disc encryption option selecting



Device Selection

Select the device(s) you'd like to install to. They will be left untouched until you click on the main menu's "Begin Installation" button.

Local Standard Disks

20 GiB
0x1af4
vda / 20 GiB free

Disks left unselected here will not be touched.

Specialized & Network Disks

Add a disk...

Disks left unselected here will not be touched.

Other Storage Options

Partitioning

● Automatically configure partitioning. ○ I will configure partitioning.

☐ I would like to make additional space available.

Encryption

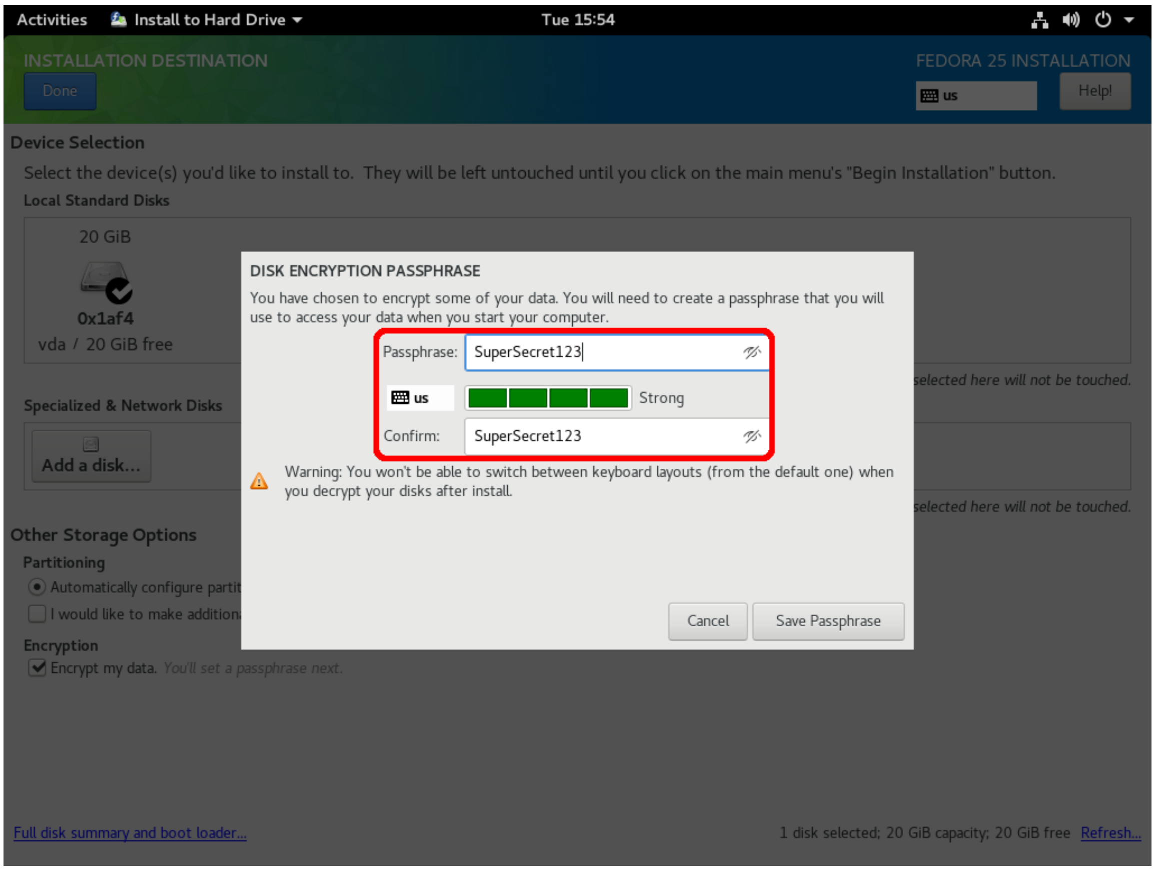
☑ Encrypt my data. *You'll set a passphrase next.*

Full disk summary and boot loader...

1 disk selected; 20 GiB capacity; 20 GiB free Refresh...

Figure E.1: Checking option

E.2 Fedora 25 - determination key encryption key


Activities   Install to Hard Drive   Tue 15:54

INSTALLATION DESTINATION   FEDORA 25 INSTALLATION

Done   us   Help!

Device Selection

Select the device(s) you'd like to install to. They will be left untouched until you click on the main menu's "Begin Installation" button.

Local Standard Disks

20 GiB

0x1af4

vda / 20 GiB free

selected here will not be touched.

Specialized & Network Disks

Add a disk...

selected here will not be touched.

Other Storage Options

Partitioning

Automatically configure partit

I would like to make addition

Encryption

Encrypt my data. *You'll set a passphrase next.*

DISK ENCRYPTION PASSPHRASE

You have chosen to encrypt some of your data. You will need to create a passphrase that you will use to access your data when you start your computer.

Passphrase: SuperSecret123

us   Strong

Confirm: SuperSecret123

Warning: You won't be able to switch between keyboard layouts (from the default one) when you decrypt your disks after install.

Cancel   Save Passphrase

Full disk summary and boot loader...   1 disk selected; 20 GiB capacity; 20 GiB free   Refresh...

Figure E.2: Determining key

Appendix F

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

F.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
```

F.2 Step 2 - resizing

[[vyskusat doplnit...]]

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
```

F.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
```

F.4 Step 4 - adding key

DO NOT FORGET to add key to LUKS volume:

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