BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PORTING TANG TO OPENWRT

PORTOVANIE TANG NA OPENWRT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TIBOR DUDLÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDREJ LICHTNER**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

The main objective of this work was to port and document the process of porting the Tang server and its dependencies to OpenWrt system, which is designed for embedded devices such as WiFi routers. 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 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 naportovať a zdokumentovať tento proces sprístupnenia 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

Rozšírený abstrakt

TODO !!! Hlavným cieľom tejto práce je naportovať a zdokumentovať tento proces sprístupnenia 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.

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
May 12, 2018

Acknowledgements

I am using this opportunity to express my gratitude to everyone who supported me throughout the journey of completing this bachelor's thesis. I am thankful for their aspiring guidance, invaluable constructive criticism and friendly advice during the work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues I encountered, and reviewing the thesis text.

I express my thanks to adelton AKA Jan Pazdziora Ph. D. and Ing. Ondrej Lichtner for their support and guidance every time I encountered an issue. I would also like to thank my colleagues Lukáš Slebodník and Stanislav Láznička and all the people who spent time listening to things they did not had to, and even provided some thoughts.

I must not forget to thank my family for supporting my study and friends for not putting my social life at risk. Special thanks to my room-mate PaRTik Segedy, for his curiosity while going through the logs and sources with me when I was feeling helpless despite the fact there might have been more interesting things for him to do (like sleep, for example, although that would be hard with the light on), and Michal Ďurista alongside with Lukáš Balog for grammar and sanity check.

Thank you all.

Contents

Chapter 1

Introduction

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

John Steinbeck[25]

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 or 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. An unauthorized party may be able to access secured data but will not be able to read the information from it without the proper key. With an increasing number of encryption keys to store and protect, it might be necessary to consider using Key management server. One of the possible solutions for persistent Key management is to deploy *key escrow* server described in section 3.1. Another solution is server *Tang*, whose principles are mentioned in section 3.2.

Tang is completely anonymous key recovery service aiming to solve early boot decryption of system volumes encrypted with *LUKS* specification, described in section 2.3. In contrast to Key Escrow server, Tang does not know any keys. It only provides mathematical operation for its clients to recover them.

The goal of this bachelor thesis is to port and document this process of porting *Tang* server to the OpenWrt system. *OpenWrt*, characterized in section 4.1, is Linux-based operating system for *embedded* devices such as wireless routers. Porting packages for the OpenWrt as described in chapter 4 is done with cross-compilation tools available from OpenWrt's buildroot.

The process of porting missing dependencies for the Tang server is described in chapter 5. Work required for the Tang itself is divided into chapter 6 and chapter 7. Section 7.5 sums up not only the limitations that are present on OpenWrt platform but also generic limitation that was discovered while testing the solution.

Result presented in this thesis allows us to automate process of unlocking encrypted drives on our private home or small office network, therefore securing 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 except the *OpenWrt* device running the *Tang* server.

Chapter 2

How we use 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, protecting 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 Encryption and security

Security is not binary; it is a sliding scale of risk management. 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 that this part of town is secure, and this is not. The way we reason about security is by studying environment, entering or observing it, and we begin to decide whether it is secure or not. Encryption, on its own, might not be enough to make our data or infrastructure secure but is definitely a critical aspect of security.

Companies define their own security strategies which may include encryption or may not at all. Security strategies rely on company's needs or will to take risks.

According to 2016 Global Encryption Trends Study, independently conducted by the Ponemon Institute, the enterprise-wide encryption increased from 15 percent in February 2006 to 41 percent in February 2017. Also, the ratio of companies with no encryption strategy at all decreased from 37 to 14 percent in these years. More than 60 percent of companies are using extensively deployed encryption technologies to encrypt mostly databases, infrastructure and laptop hard drives [27].

Passwords are the most common authentication method used for accessing computer systems, files, data, and networks. As stated in article *Solid IT control hygiene* [23], it is important to keep changing them in reasonable time and keep them secret to others. Still, no matter the company's security strategy, we keep seeing them on monitors or desktops 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 even birth date in their passwords [29]. They choose cryptographically 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, 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, we may follow any trustworthy guide[1] on the Internet. More secure way to create passwords would be to generate cryptographically stronger cipher and use it as a password. The only disadvantage is that it is hard to remember [24].

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.

Hard drive encryption is a technology provided by software performing sophisticated mathematical functions or hardware that encrypts the data stored on a hard drive or a disk volume. This technology is used to prevent unauthorized access by unauthorized persons or service to an encrypted data storage without possession of the appropriate key or password. Encrypting the hard drive means providing another layer of security against hackers and other online threats.

To protect secret data, we usually encrypt this data by using an "encryption key" – see Figure 2.1 Hard drive encryption in a nutshell. Every encryption key should be

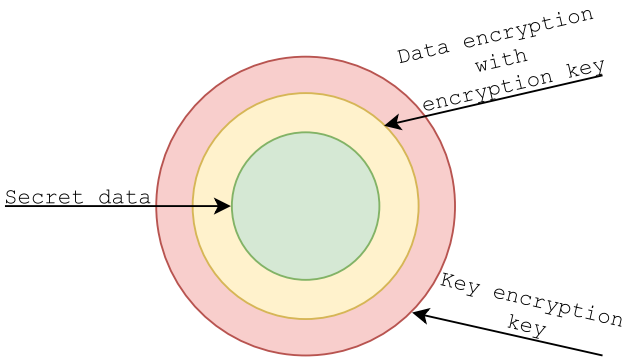


Figure 2.1: Hard drive encryption in a nutshell

unpredictable and unique set of bits able to "scramble" data in a way that it should be impossible to recover data without the key. To satisfy this need, encryption keys are generated by specialized algorithms such as AES (Advanced Encryption Standard[2]) for

[1] https://www.wikihow.com/Create-a-Secure-Password
[2] https://en.wikipedia.org/wiki/Symmetric-key_algorithm#Implementations

symmetric keys and RSA (Rivest–Shamir–Adleman[3]) for asymmetric. Changing encryption key implies that all the data needs to be decrypted with the old key and re-encrypted with the new one, every time the old key is compromised or a change is required. To avoid time-consuming re-encryption whenever key change occurs, the encryption key is then wrapped by the key encryption key.

Key encryption key is mostly generated using the user-provided password. This key encryption key is then used to encrypt the encryption key which does the actual data encryption. Again, the most insecure thing in this key hierarchy would be the user-provided password which we can easily replace with using only cryptographically stronger key. This principle has at least two advantages. Changing the key encryption key does not affect encrypted data, and key can be changed whenever the user desires to, and redistributed to all users or services that are supposed to decrypt this data, giving access to the encryption key.

Hard drive can be encrypted as whole or per partition. Full disk encryption is done in a way that all content on the hard drive except MBR (Master Boot Record) is encrypted. Encrypting MBR would make it impossible to start boot sequence of operating system. Boot sequence would prompt the user for key encryption key in order to load the operating system from encrypted storage.

This could disrupt our daily workflow and might be the reason why most of us do not use hard drive encryption, even when we know it will protect our data. There is a way to automate the hard drive unlocking on early boot with a help from key management system. We can get the key encryption key from some remote system, the Escrow server mentioned in section 3.1, or recover it with Tang described in chapter 3.2. Before that, let us have a look on most common hard drive encryption implementations.

2.2.1 Bit Locker

BitLocker is a closed source, full disk data protection feature that integrates with the operating system Windows Vista and later. It is designed to protect data by providing encryption for entire volumes and addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers. By default, it uses the AES encryption algorithm in cipher block chaining (CBC) or XTS mode with a 128-bit or 256-bit key. CBC is not used over the whole disk; it is applied to each individual sector [17].

2.2.2 LibreCrypt

LibreCrypt is a *LUKS* (Linux Unified Key Setup-on-disk-format) compatible open source "on-the-fly" transparent disk encryption software written mostly in Pascal programming language. This project is based on original FreeOTFE project by Sarah Dean renamed in version 6.2 to LibreCrypt and supports both 32 and 64 bit Windows. LibreCrypt is easy to use even for inexperienced user through its GUI (Graphical User Interface) with support of many languages.

LibreCrypt supports many ciphers including AES (up to 256 bit), Twofish[4] (up to 256 bit), Blowfish[5] (up to 448 bit), Serpent[6] (up to 256 bit). It can create "virtual disks" on

[3] https://en.wikipedia.org/wiki/Public-key_cryptography
[4] https://en.wikipedia.org/wiki/Twofish
[5] https://en.wikipedia.org/wiki/Blowfish_(cipher)
[6] https://en.wikipedia.org/wiki/Serpent_(cipher)

our computer and anything written to these disks is automatically encrypted before being stored on our computer's hard drive[31].

Unfortunately this project seems to be abandoned by its developer on GitHub The source code of LibreCrypt is available at GitHub.

```
https://github.com/t-d-k/LibreCrypt
```

2.2.3 dm-crypt

The device-mapper crypt target (*dm-crypt*) provides transparent encryption of block devices using the kernel crypto API (Application programming interface) supporting ciphers and digest algorithms via standard kernel modules. Device-mapper is a part of the Linux kernel that provides a generic way to create virtual layers of block devices, for example LVM (Logical Volume Manager) logical volumes.

In Fedora and Red Hat Enterprise Linux distributions, user-space interaction with *dm-crypt* is managed by a tool called `cryptsetup`, which uses the device-mapper infrastructure to setup and operate on encrypted block devices. With modern versions of `cryptsetup` (i.e., since 2006), encrypted block devices can be created in two main formats, plain *dm-crypt* format or the extended *LUKS* format.

Plain format has no meta-data on disk. When using any such encrypted device, all the necessary parameters must be passed to `cryptsetup` from the command line, otherwise it uses the defaults, which will only succeed if the device was created using default settings. It derives (generates) the master key from the pass-phrase provided and then uses that to decrypt or encrypt the sectors of the device, with a direct 1:1 mapping between encrypted and decrypted sectors.

In contrast to previous Linux disk encryption solutions, *LUKS* stores all necessary setup information in the partition header, enabling the user to transport or migrate their data more easily.

2.3 Disk encryption with LUKS

LUKS (Linux Unified Key Setup-on-disk-format) is a platform-independent disk encryption specification. *LUKS* was created by Clemens Fruhwirth in 2004 and was originally intended for Linux distributions only. It provides a standard on-disk-format for hard disk encryption, which facilitates compatibility among Linux distributions and provides secure management of multiple user passwords.

Referential implementation of *LUKS* is using a device-mapper crypt target (*dm-crypt*) subsystem for bulk data encryption. This subsystem is not particularly bound to *LUKS* and can be used for plain format encryption as mentioned in subsection 2.2.3.

It is important for us to know this specification a little more due to working with Linux distribution and the implementation of the Tang server.

The advantages of *LUKS* over plain *dm-crypt* are better usability: automatic configuration of non-default crypto parameters and the ability to add, change, and remove multiple pass-phrases. Additionally, *LUKS* offers defenses against low-entropy pass-phrases with salting and iterated PBKDF2[7] (Password-Based Key Derivation Function 2) pass-phrase hashing[13]. With *LUKS*, encryption keys are always generated by the kernel RNG (Random number generator); in contrast to plain *dm-crypt* where one can choose a simple

[7] https://www.ietf.org/rfc/rfc2898.txt

dictionary word and have an encryption key derived from that. One disadvantage of using *LUKS* over plain is that it is readily obvious there is encrypted data on disk; the other is that damage to the header or key slots usually results in permanent data loss. To mitigate this risk the Backup of the *LUKS* header is the best option [5].

2.3.1 Creating LUKS volume

Creating a *LUKS* volume with the `cryptsetup` tool is easy. On Fedora system it can be installed using command:

```
# dnf install cryptsetup
```

Run this command with root privileges. After installation succeeds choose a partition to encrypt. For demonstration, we will encrypt the `/dev/xvdc` patrtition using:

```
# cryptsetup -y -v luksFormat /dev/xvdc
WARNING!
========
This will overwrite data on /dev/xvdc irrevocably.

Are you sure? (Type uppercase yes): YES
Enter LUKS passphrase:
Verify passphrase:
Command successful.
```

But converting existing non-encrypted disk to have full disk encryption if the system is already installed might be quite tough. Hard drive partition must contain the *LUKS* header just before the encrypted data. Let us 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 ask for pass phrase – the key encryption key. To demonstrate this, screen shots with Fedora 28 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 we can encrypt the whole system disk with *LUKS* without unmounting a partition to encrypt. For this purpose *luksipc*, the LUKS In-Place Conversion Tool, was developed. Steps to encrypt a disk using *luksipc* are in the appendix C.

2.3.2 LUKS drive structure

The structure of *LUKS* partition is shown in Figure 2.2 LUKS volume structure for a demonstration. At the the beginning of the device the *LUKS* format uses a meta-data header, also marked as *phdr*, and 8 key slot areas. After header, there is a section with bulk data, which is encrypted with the encryption key. Header contains information about the

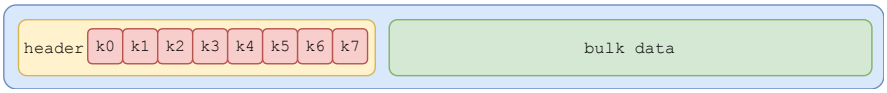


Figure 2.2: LUKS volume structure

cipher used, cipher mode, the key length, a uuid, and a master key check-sum. The pass-phrases stored in the key slots, which structure is shown on Figure 2.3 LUKS Key Slot, are

used to decrypt a single key encryption key that is stored in the anti-forensic stripes. Anti-forensic data storage, as described in article LUKS On-Disk Format Specification Version 1.1 [5], is a feature specially developed for LUKS by Clemens Fruhwirth. The idea of anti-forensic information splitting in *LUKS* is to enlarge the size of every storage region for the encryption key encrypted with one of the user key encryption keys such that all parts of this storage region are required in order to recover the encrypion key.

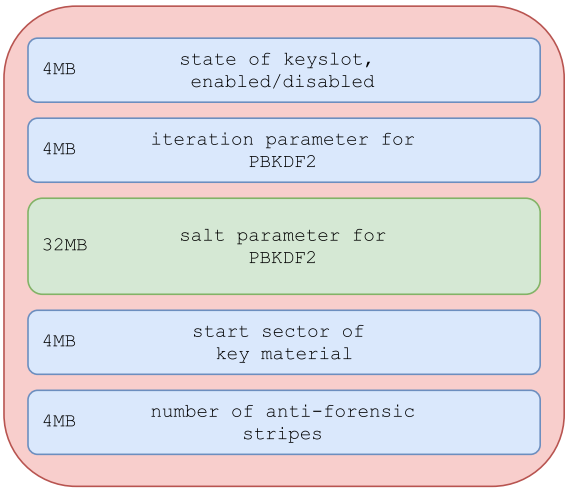


Figure 2.3: LUKS Key Slot

Every key slot is associated with a key material section after the header. 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 or cipher. Supplying this user password unlocks the decryption for the key material, which stores the master key.

2.3.3 Managing LUKS volume

To demonstrate *LUKS* volume management let us shot steps that can be used on our Fedora 27 system. For this demonstration we will use tools available for Fedora 27 such as:

- parted-3.2-28.fc27.x86_64
- cryptsetup-1.7.5-3.fc27.x86_64

To list hard drives for our system we will use the `parted` tool:

```
# parted -l
Model: NVMe Device (nvme)
Disk /dev/nvme0n1: 256GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system  Flags
 1      1049kB  1075MB  1074MB  primary  ext4         boot
 2      1075MB  256GB   255GB   primary
```

The output shows that on our system we have installed an NVMe (NVM Express) volume[8] using the msdos, therefore MBR partition table. The NVMe is a specification for accessing SSDs attached through the PCI Express bus. Our SSD drive has two partitions the first is a boot partition and second should be our *LUKS* partition. To find out if the second partition is *LUKS* volume, we will use cryptsetup tool with option *isLuks*:

```
# cryptsetup isLuks /dev/nvme0n1p1 -v
Device /dev/nvme0n1p1 is not a valid LUKS device.
Command failed with code 22: Invalid argument
```

Unless the -v option is used, this command produces no output. The actual result of the command is returned as an exit code. The output shown reflects that our first partition is not a valid *LUKS* volume. This is expected behavior, encrypting the boot partition would make it impossible for the system to boot.

```
# cryptsetup isLuks /dev/nvme0n1p2 -v
Command successful.
```

At this point, we have successfully identified the *LUKS*volume. To dump the header information of this *LUKS* volume, `cryptsetup` option *luksDump* can be used:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: ENABLED
    Iterations:             415583
    Salt:       3e f9 7d 3b b6 08 60 9a eb dc 52 bb 8e 21 eb bf
                b9 4d 80 a4 70 2d 4e 97 8e 47 c1 a3 04 45 74 d4
    Key material offset:    8
    AF stripes:             4000
Key Slot 1: DISABLED
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

From the listing we can assume that `/dev/nvme0n1p2` has one key encryption key in key slot 0 and has other 7 slots (1-7) disabled. The advantage of *LUKS* is that if a team or group of up to eight people has to work with a common encrypted volume, each team member may use their own password. New user keys may be added to the encrypted volume and old user keys may be removed. To add a new key, we will use `cryptsetup` with option *luksAddKey* as shown:

[8] https://en.wikipedia.org/wiki/NVM_Express

```
# cryptsetup luksAddKey  /dev/nvme0n1p2
Enter any existing passphrase:
Enter new passphrase for key slot:
Verify passphrase:
```

The `cryptsetup` will derive the key encryption key from this passphrase and bind it to the first available key slot in *LUKS* header. The dump of header will now contain information similar to the exhibit below:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: ENABLED
    Iterations:             415583
    Salt:       3e f9 7d 3b b6 08 60 9a eb dc 52 bb 8e 21 eb bf
                b9 4d 80 a4 70 2d 4e 97 8e 47 c1 a3 04 45 74 d4
    Key material offset:    8
    AF stripes:             4000
Key Slot 1: ENABLED
    Iterations:             1247257
    Salt:       44 48 92 df 29 8a df 81 f6 44 f8 66 c5 c2 32 49
                23 76 8a 37 48 85 33 2a 29 10 d8 cc 8f 45 0a 46
    Key material offset:    1520
    AF stripes:             4000
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

At this point, *LUKS* volume is accessible via two different passphrases. To remove the key from *LUKS* header we will use cryptsetup with option *luksKillSlot* as shown:

```
# cryptsetup luksKillSlot /dev/nvme0n1p2 0
Enter any remaining passphrase:
```

To successfully remove key from key slot 0, we have to enter any remaining pass-phrase. In our case, we would enter the newly created pass-phrase which is bound to key slot 1. This will result into having *LUKS* header in state like shown:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
```

```
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: DISABLED
Key Slot 1: ENABLED
    Iterations:             1247257
    Salt:       44 48 92 df 29 8a df 81 f6 44 f8 66 c5 c2 32 49
                23 76 8a 37 48 85 33 2a 29 10 d8 cc 8f 45 0a 46
    Key material offset:    1520
    AF stripes:             4000
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

As stated in section 2.1 password rotation is very important. To change *LUKS* key slot pass-phrase use the `cryptsetup`'s option *luksChangeKey*:

```
#cryptsetup luksChangeKey /dev/nvme0n1p2 -S 1
Enter passphrase to be changed:
Enter new passphrase:
Verify passphrase:
```

Note that pass-phrase to be changed and the slot number must be related to each other. We could see a change in *LUKS* header after command succeeded:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: DISABLED
Key Slot 1: ENABLED
    Iterations:             1630572
    Salt:       cf 54 a9 77 ed 8b c5 75 ca 65 60 6b 31 cb 29 0f
                4e 54 78 8c b1 9a db 3f 2f 6c aa 84 79 da 81 66
    Key material offset:    512
    AF stripes:             4000

Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
```

```
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

More use-cases and examples how to use `cryptsetup` can be found the man page of the tool[9]. Examples shown demonstrate basic key management for *LUKS* volumes and should be sufficient for basic understanding of `cryptsetup`/*LUKS* behavior and usability on the system with an encrypted drive.

2.3.4 LUKS security

In August 2012, Ubuntu Privacy Remix Team did a deep analysis of *LUKS*/`cryptsetup` in Ubuntu environment. `cryptsetup` in version 1.4.1 used by Ubuntu 12.04 LTS has been chosen for the analysis. The team wrote the programs *luksanalyzer* and *hashtest* for the purpose of the analysis.

The analysis lead to conclusion that `cryptsetup` with *LUKS* is a highly secure program for encrypting whole data media or partitions. The encryption algorithms, and other security mechanisms it implements, comply with the current state of the art in cryptography. No back door, or security-related mistakes were found in the published source code. If we use this program in a secure environment, the passwords are strong, and the attacker does not apply highly advanced methods below the layer of the operation system, such as BIOS root-kits, hardware key-loggers or video surveillance, we may assume with high certainty that no one can get access to the data stored in our volumes. A special strong point of `cryptsetup` with *LUKS* is its high power of resistance against dictionary attacks[26].

[9] https://linux.die.net/man/8/cryptsetup

Chapter 3

Automated decryption

As article *Inductive programming meets the real world* [6] states, that we should try to automate everything we can in order to avoid repetitive tasks. Even though disc encryption provides another layer of security to our data, it is used less. Typing one more pass-phrase when accessing some removable storage seems like too much work. More crucial would be full disk encryption. In order to boot the system, we need to have storage with system decrypted – provide another pass-phrase every time the computer is turned off and on again. The Tang server aims to solve struggles with the early boot decryption of system volumes. Before Tang, automated decryption was usually handled by a Key Escrow server.

3.1 Key Escrow

The Key Escrow server (also known as a "fair" cryptosystem) is providing escrow service for encryption keys. A client using Key Escrow usually generates a key, encrypts data with it and then stores the key encryption key on a remote server. Unfortunately there are couple of security concerns.

To transfer the encryption keys we want to store on an Escrow server, we have to encrypt the channel on which we send them. When transmitting keys over an insecure network without an encrypted link, anyone listening to the network traffic could immediately 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 Escrow model. Unfortunately, this is not enough to have the communication secure.

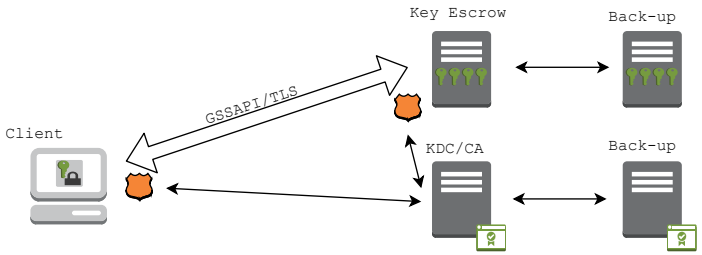


Figure 3.1: Escrow model

This server has to have its identity verified, and the client has to authenticate to this server

too. The increasing amount of keys implicates a need for Certification Authority server (CA) or a Key Distribution Center (KDC) to manage all of them.

Only with the infrastructure in place and keys produced, the server can verify if the client is permitted to get their key, and the client is able to identify a trusted server. This is a fully stateful process.

The complexity of this system increases the attack surface and for such complex system it would be unimaginable not to have backups. The Escrow server may store lots of keys from lots of different places, users and services.

3.2 Tang server

With key escrow, a third party gets copies of a cryptographic key. People might not be comfortable with any third party having this ability and that "technical" problems vex the key escrow solution. Tang's key recovery, on the other hand, lets us just "backup" and restore cryptographic keys anonymously and without any third party possesing our key.

Tang is a very lightweight network service using systemd's socket activation as described in section 6.1. Its purpose is to provide anonymous key recovery to clients over the network. This key recovery can be used with a help of the Tang's client descibed in subsection 3.2.4 Clevis client to unlock data storages with LUKS encryption. The Tang server is an open source project implemented in C programming language.

We can see on Figure 3.2 that the Tang model is very similar to the escrow seen on Figure 3.1 but with some things missing. In fact, there is no longer need for TLS channel to secure

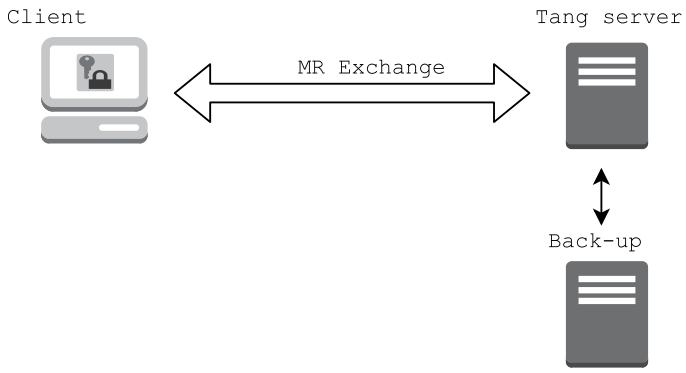


Figure 3.2: Tang model

communication between the client and the server, and that is because Tang implements the McCallum-Relyea exchange as described below.

Tang server advertises asymmetric keys on the network and a client is able to get the list of these signing keys by HTTP (Hypertext Transfer Protocol) GET request. All tang communication is performed using the HTTP protocol with the message body in JWK (JSON Web Key) format defined by RFC 7517 [8]. Getting advertised keys is the first step of the McCallum-Relyea exchange protocol summed up in table 3.1. This protocol has two phases, provisioning and recovery.

Provisioning		Recovery	
client's side	server's side	client's side	server's side
	$S\epsilon_{bindingR}[1, p-1]$ $s = g * S$	$E\epsilon_R[1, p-1]$ $x = c + g * E$	
	← s	x →	
$C\epsilon_R[1, p-1]$ $e = g * C$ $k = g * S * C = s * C$ Discard: K, C Retain s, c		← y $k = y - s * E$	$y = z * S$

*capital is private key; g stands for generate

Table 3.1: McCallum-Relyea exchange protocol

Provisioning The server key pair generation is represented by equation 3.1, capital is private key; lowercase is public key; g stands for generate.

$$s = g * S \tag{3.1}$$

After the server generates key pair, it is advertising the public part of it. The client then 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 public key and JWK is format of the message. The client then generates a new (random) JWK (*cJWK*; c stands for client's key pair).

$$c = g * C \tag{3.2}$$

The client performs its half of a standard ECDH exchange producing *kJWK* – see equation **??**, which it uses to encrypt the data. Afterwards, it must discard the *kJWK* and the private key from *cJWK*.

$$k = s * C \tag{3.3}$$

The client has to store *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 called also binding keys.

Recovery Mathematically capital letter is private key; g stands for generate. When the client wants to access the encrypted data, it must be able to recover encryption key. To recover *kJWK* after discarding it, the client generates a third ephemeral key (*eJWK*) as the equation 3.4 represent. This key is used to hide the client's public key and the binding key.

$$e = g * E \tag{3.4}$$

The ephemeral key is generated for each execution of a key establishment process. Using *eJWK*, the client performs elliptic curve group addition of *eJWK* and *cJWK*, producing *xJWK* represented in the equation 3.5. The client *POST*s *xJWK* to the server.

$$x = c + e \tag{3.5}$$

The server then performs its half of the ECDH key exchange using *xJWK* and *sJWK*, producing *yJWK* reflected in the equation 3.6. The server returns *yJWK* to the client.

$$y = x * S \tag{3.6}$$

The client then performs half of an ECDH key exchange between *eJWK* and *sJWK*, producing *zJWK* as the equation 3.7 shows.

$$z = s * E \tag{3.7}$$

Subtracting *zJWK* from *yJWK* produces *dJWK* shown in the equation 3.8.

$$k = y - z \tag{3.8}$$

Finally, the client has calculated the key value. So the Tang server never knows the value of the key and literally nothing about its clients.

3.2.1 Security

As shown in table 3.2, Tang compared to Escrow is stateless and doesn't require TLS or authentication. Tang also has limited knowledge. Unlike escrows, where the server has knowledge of every key used, Tang never sees any client keys. 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 3.2: Comparing Escrow and Tang

Thanks to McCallum-Relyea exchange protocol summed up in the table 3.1 Tang is resistant to the man in the middle attack. In case of the eavesdroppers, they see the client send *xJWK* and receive *yJWK*. Since these packets are blinded by *eJWK*, only the party that can un-blind these values is the client itself (since only it has *eJWK*'s private key). Thus, the attack fails.

It is of utmost importance that the client protects *cJWK* and the Tang server must protect the private key for *sJWK*.

3.2.2 Building Tang

Tang is originally packaged for Fedora operating system version 23 and later but we can of course build it from source. It relies on few other software libraries listed in section 5.1.

The steps to build the Tang from sources include downloading the source from project's GitHub or cloning it. Make sure all required dependencies are installed and then run:

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

Optionally tests can be run with:

```
$ make check
```

3.2.3 Server enablement

Enabling a Tang server is a two-step process. First step is to enable the Tang services. Start the key update service which is watching the database directory using systemd:

```
# systemctl enable tangd-update.path --now
```

Enable service using systemd socket activation:

```
# systemctl enable tangd.socket --now
```

After this, systemd will handle the sockets and the Tang's key rotation procedure.

Second, generate a signing key using dependency tool jose and store it in a default directory expected by tang:

```
# jose gen -t '{"alg":"ES256"}' -o /var/db/tang/sig.jwk
```

Do not forget to generate an exchange key:

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

These commands results into change in the Tang's database directory now containing the `sig.jwk` and the `exc.jwk`. The `tangd-update.path` service will trigger regeneration of the cache, stored in `/var/cache/tang/` directory, using `/usr/libexec/tangd-update` script.

Now we are up and running. The server is ready to send an advertisement on client's demand or even using curl:

```
curl -f http://tang.local/adv
```

Tang now advertises its `exc.jwk` key signed using the `sig.jwk`.

3.2.4 Clevis client

Clevis provides a pluggable key management framework for automated decryption and has full support for Tang. It can handle automated unlocking of LUKS volumes. Clevis lets us encrypt data with a 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 the supported pin here. Besides `Tang` *PIN* clevis also supports a *PIN* for performing escrow using `HTTP` or an `SSS` *PIN* to provide a way to mix pins together to provide sophisticated unlocking policies by using an algorithm called Shamir Secret Sharing (SSS).

Second, *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.

PIN: Tang – Here is an example of how to use Clevis with Tang:

```
$ echo hi | clevis encrypt tang '{"url": "http://tang.local"}' > hi.jwe
The advertisement contains the following signing keys:

Apb39FO1vey9FyUe_fEd8lVDABs

Do you wish to trust these keys? [ynYN] y
$ clevis decrypt tang '{"url": "http://tang.local"}' < hi.jwe
hi
```

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 us to trust the keys. This works similarly to SSH.

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

3.2.5 Binding LUKS volumes with clevis

Tang's main purpose is to enable early boot decryption of the *LUKS* volumes and clevis is the client solution for it. Clevis can be used to bind a *LUKS* volume using a *PIN* so that it can be automatically unlocked.

Clevis automatically generates a new, cryptographically strong key using the Tang's advertisement. This key is added to *LUKS* as an additional passphrase. Clevis then encrypt this key using Tang's advertisement, and store the output JWE (JSON Web Encryption) inside the *LUKS* header using *LUKSMeta* library. Here is an example where we bind `/dev/vda2` using the Tang ping:

```
# 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 unlockers described below.

Dracut Install it to Fedora using:

```
# dnf install clevis-dracut
```

The Dracut unlocker attempts to automatically unlock volumes during early boot. This permits the automated root volume encryption. To unlock Fedora, initramfs must be rebuilt after installing Clevis using:

```
# dracut -f
```

Upon reboot, we 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 canceled and boot will continue.

UDisks2 After installation, UDisks2 unlocker runs in a Fedora desktop session. There is no need to manually enable it; just install the Clevis UDisks2 unlocker and restart desktop session.

```
# dnf install clevis-udisks2
```

The unlocker should be started automatically. This unlocker works almost exactly the same as the Dracut unlocker. If we insert a removable storage device that has been bound with Clevis, it 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 4

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 the actual process of „carrying“ software to a system with a different environment is often required. Porting is process required to adapt software designed for specific platform to another platform. Porting is also used to describe process of converting computer games to become platform independent[32].

Software porting process might be hard to distinguish from building software. The reason might be that in many cases, re-building software on the desired platform is enough.

Nowadays, the goal should be to develop software which is portable between preferred computer platforms (Linux, UNIX, Apple, Microsoft). If the software is considered as not portable, it does not have to mean that it is not possible, just that the 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 a desired platform must be little, such as copying already installed files from one computer to another and run/re-build it. This kind of approach might most probably fail, due to not present dependencies of third party libraries on the 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.

Number of significantly different central processor units (CPUs), and operating systems used on the desktop or the server is much smaller than in the past. However, on embedded devices market, there are still much more various architectures available including ARM[1] or MIPS[2].

To simplify portability, even on 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 operating system belongs to, porting remains a significant issue [14].

[1] https://www.arm.com/products/processors
[2] https://www.mips.com/products/classic/

4.1 OpenWrt system

OpenWrt is 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. OpenWrt is a registered trademark which is held by the Software in the Public Interest (SPI) in the name of the OpenWrt project.

Installing OpenWrt system means replacing our 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. A 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. [19]

4.1.1 OpenWrt and LEDE

The LEDE Project ("Linux Embedded Development Environment") is a Linux operating system emerged from the OpenWrt project. Its announcement was sent on 3th May 2016 by Jo-Philipp Wich to both the OpenWrt development list and the new LEDE development list[3]. It describes LEDE as „a reboot of the OpenWrt community" and as „a spin-off of the OpenWrt project" seeking to create an embedded-Linux development community „with a strong focus on transparency, collaboration and decentralisation"[4].

The rationale given for the reboot was that OpenWrt suffered from longstanding issues that could not be fixed from within—namely, regarding internal processes and policies. For instance, the announcement said, the number of developers is at an all-time low, but there is no process for on-boarding new developers and no process for granting commit access to new developers.

At the moment, the latest release of OpenWrt is 15.05.1 (code-named „Chaos Calmer") released in March 2016. LEDE developers continued to work separately on their upstream release and they delivered LEDE „Reboot" with version 17.01.0 on February 22nd 2017.

The remerge proposal vote was passed by LEDE developers in June 2017[5]. After long and sometimes slowly moving discussions about the specifics of the re-merge, with multiple similar proposals but little subsequent action, projects formally announced on LEDE forum in January 2018[6]. OpenWrt and LEDE projects agreed upon their unification under the OpenWrt name. After merge, OpenWrt upstream repository started to show signs of life.

Today (April 2018) the stable LEDE 17.01.4 „Reboot" release of OpenWrt released in October 2017 using Linux kernel version 4.4.92 runs on many routers [20].

4.1.2 Why use OpenWrt

Custom router firmware may be more stable than our hardware's default firmware from the vendor. Not even that, but probably more secure with regular security updates. Besides OpenWrt, there is another open source Linux based firmware available such as Tomato or DD-WRT.

[3] https://lwn.net/Articles/686180/
[4] https://www.phoronix.com/scan.php?page=news_item&px=OpenWRT-Forked-As-LEDE
[5] http://lists.infradead.org/pipermail/lede-adm/2017-June/000552.html
[6] https://forum.lede-project.org/t/announcing-the-openwrt-lede-merge/10217

In the past OpenWrt has supported only CLI (command line interface) configuration, therefore, it was best match for software developers, network admins, or advanced users. A user not acquainted with Linux or even not comfortable with CLI may find the OpenWrt platform hard to use and may turn to other available solutions.

The Tomato firmware is the best match for unexperienced users providing rich GUI and many other features, specifically live „visual“ traffic monitoring, allowing easy visibility on inbound/outbound traffic in real-time. Big disadvantage of the Tomato is that the list of supported devices is quite poor.

DD-WRT on the other hand, is compatible with more routers than any other third party firmware. Compared to Tomato, DD-WRT is reported to have more bugs and less intuitive GUI. The DD-WRT was known as the most feature rich firmware until OpenWrt came along.

OpenWrt turns our router in a fully capable GNU/Linux computer, not just a network „magic“ box. Recently, the OpenWrt platform has come a long way in making itself more accessible to all user levels. It has an Luci[7] Web UI(user interface) now. Users that are less experienced with Linux can easily set up their network using Luci. OpenWrt is capable of running lightweight services like an IRC bouncer or samba/ftp file sharing (some routers have USB ports able to power HDD) or even run software build on our own[28].

The goal of this thesis would be to port a lightweight Tang daemon, described in the section 3.2, to OpenWrt system. Tang server will help us unlock our encrypted volumes while on safe home or office network without a need for extra PC running it but having it on our tiny OpenWrt „server“. With Tang, we do not have to care about typing passphrases over and over to unlock LUKS drives in a safe environment.

4.2 OpenWrt's tool-chain

Compilation is done by set of tools called tool-chain and it consists of:

- compiler
- linker
- a C standard library

Embedded devices are not meant for building on them because they do not have enough memory nor computation resources as ordinary personal computers do. For example device specification see Table 4.1. Building on such device would be time consuming and may result in overheating, which could cause the hardware to fail. For this particular reason, package building is done with cross-compiler.

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

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

[7] https://github.com/openwrt/luci

(<stdio.h>), memory management (<stdlib.h>), string handling (<string.h>), mathematical computations (<math.h>), and many more[8]. The OpenWrt's cross-compilation tool-chain uses musl-libc.

For porting to „target“ system (OpenWrt) this tool-chain has to be generated on „host“ system. The tool-chain can be created in many different ways. The easiest way is undoubtedly to find a .rpm (.deb or any distribution specific package available) package and have it installed on our „host“ system. If a binary package with desired tool-chain is not available for our system or is not available at all, there might be a need to compile a custom tool-chain from scratch[9].

In case of OpenWrt, we have an available set of Makefiles and patches called buildroot which is capable of generating tool-chain.

4.3 OpenWrt's buildroot

OpenWrt's buildroot is a build system capable of generating the tool-chain, and also a root file-system (also called sysroot), an environment tightly bound to the target. The build system can be configured for any device that is supported by OpenWrt.

The root file-system 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 tool-chain and the root file-system 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![3]

4.3.1 Buildroot prerequisites

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

- ca. 200 MB for OpenWrt build system
- ca. 300 MB for OpenWrt build system with additional packages
- 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

Table 4.1 lists the 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.

[8] https://en.wikipedia.org/wiki/C_standard_library#Header_files
[9] tools like crosstool-ng (https://github.com/crosstool-ng/crosstool-ng) may help

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
Antennae(s):	2 non-removable
Ethernet:	4 LAN, 1 WAN 10/100
USB:	1 x 2.0
Serial:	No

Table 4.1: TL-WR842Nv3 specifications

Comparison of available storage space on wireless router to the actual sum of space required only for buildroot to work correctly, which is about 6.4 GB, should demonstrate why the building for embedded devices is done with cross-compiling. Another reasons, not to just compare internal storage which might be extended[10], are device's minimalistic RAM size and low computation capability of the CPU.

4.4 Preparing the host environment

To start with cross-compilation on the host system, we need to set up an environment for it. As the OpenWrt buildroot is a set of scripts, it has run-time dependencies. We need to install these dependencies first. To install buildroot dependencies on Fedora 27 system run:

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

Some packages might be not available over git but only via other versioning tools like svn (subversion) or mercurial. In our case this will not be necessary but if we want to obtain their source-code, we need to install svn and mercurial as well:

```
# dnf install subversion mercurial
```

These commands have to be run by a user with root privileges.

4.4.1 Getting buildroot

OpenWrt system has SDK (Software development kit) buildroots available for every released version of OpenWrt system. It is good to consider using OpenWrt's SDK in order to build the software application for specific release of the target system. For example when we

[10] https://wiki.openwrt.org/doc/howto/extroot

are using a „stable“ release of OpenWrt 15.05.1 with code-name „Chaos Calmer“ on TL-WR842N(EU) device, we should probably end up in „Supplementary Files“ section of the OpenWrt archives[11] looking for the SDK. While porting an upstream projects (such as Tang) for latest target release a bleeding edge buildroot would be the best solution.

If we have a host platform which aims to be on the bleeding edge, such as OS Fedora, we will probably encounter issues with dependencies. These issues are appearing because an older version of the dependencies are required for this old SDK to work and might not be available for our host platform anymore. In this case, we would have to install or even build an older version of required packages. We will work with trunk version buildroot and build for the latest LEDE 17.01.4 release. To clone upstream buildroot run command:

```
$ git clone https://github.com/openwrt/openwrt.git
```

4.5 Working with buildroot

Working with Openwrt's buildroot requires some basic knowledge of the git version control system[12] and processes of upstream project development using GitHub[13]. A short description of how to fork and set up repository can be found in appendix D Setting up the repository. All work related to this thesis has been done using GitHub account Tiboris[14] therefore following command to get our fork of buildroot would be:

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

After we have the environment ready to be worked on, all required dependencies installed and fork of our buildroot downloaded on host system, it would be useful to have these 4 rules in mind, to not break our environment 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

4.5.1 Setting feeds

In OpenWrt, a *feed* is a collection of packages which share a common location. *Feeds* may reside on a remote server, in a version control system, on the local file-system, or in any other location addressable by a single name (path/URL) over a protocol with a supported *feed* method. Setting the *feeds* is the most important step to do before starting cross-compilation. Listing 4.1 shows, a list of usable *feeds* configured by default.

[11] https://archive.openwrt.org/chaos_calmer/15.05.1/ar71xx/generic/
[12] https://git-scm.com/docs/gittutorial
[13] https://guides.github.com/introduction/flow/
[14] https://github.com/Tiboris/

```
src-git packages https://git.openwrt.org/feed/packages.git
src-git luci https://git.openwrt.org/project/luci.git
src-git routing https://git.openwrt.org/feed/routing.git
src-git telephony https://git.openwrt.org/feed/telephony.git
#src-git video https://github.com/openwrt/video.git
#src-git targets https://github.com/openwrt/targets.git
#src-git management https://github.com/openwrt-management/packages.git
#src-git oldpackages http://git.openwrt.org/packages.git
#src-link custom /usr/src/openwrt/custom-feed
```

Listing 4.1: Content of feeds.conf.default

Custom OpenWrt packages are located in `packages` *feed* – see the first line of the listing 4.1. To work with our own fork of the packages *feed*, we can simply change the line to point to our fork repository. Feeds can point to a special *branch* of our choice:

```
src-git packages https://github.com/Tiboris/packages-OpenWrt.git;new_pkgs
```

or commit hash in repository:

```
src-git packages https://github.com/Tiboris/packages-OpenWrt.git^dbdfc99
```

Changing this *feed* to our custom *branch* will reduce effort spent on getting all new dependencies to the buildroot's `feeds/packages` directory.

The feeds are 'managed' with the script available in builroot's `scripts` directory, `feeds`. To download the feeds, run this script with command *update* and option *-a* as shown. Remember to invoke it from the buildroot directory(∼/buildroot-openwrt).

```
$ ./scripts/feeds update -a
```

To make any package available for the build, we shall „install“ it using the same feeds script:

```
$ ./scripts/feeds install <PACKAGENAME>
```

or we can use option *-a* instead of the *<PACKAGENAME>* and make all of the packages available for the build. If for some unknown reason the issued update of packages does not seems to show all the updates, it might be helpful to clean the buildroot's `tmp/` directory[22].

```
$ rm -rf tmp/
```

4.5.2 The menuconfig

For OpenWrt menuconfig provides a simple, yet powerful, environment for the configuration of individual builds. Working with menuconfig is very intuitive. Even the most specialized configuration requirements can be met by using it. Depending on the particular target platform, package requirements, and kernel module needs, the standard configuration process will include modifying:

- Target system
- Package selection
- Build system settings
- Kernel modules

Start the menuconfig interface shown on Figure 4.1 menuconfig by issuing the following command:

```
$ make menuconfig
```

The `make menuconfig` will collect package information first. Afterwards, the following window will appear in our terminal and there we can start configuring the image using menuconfig:

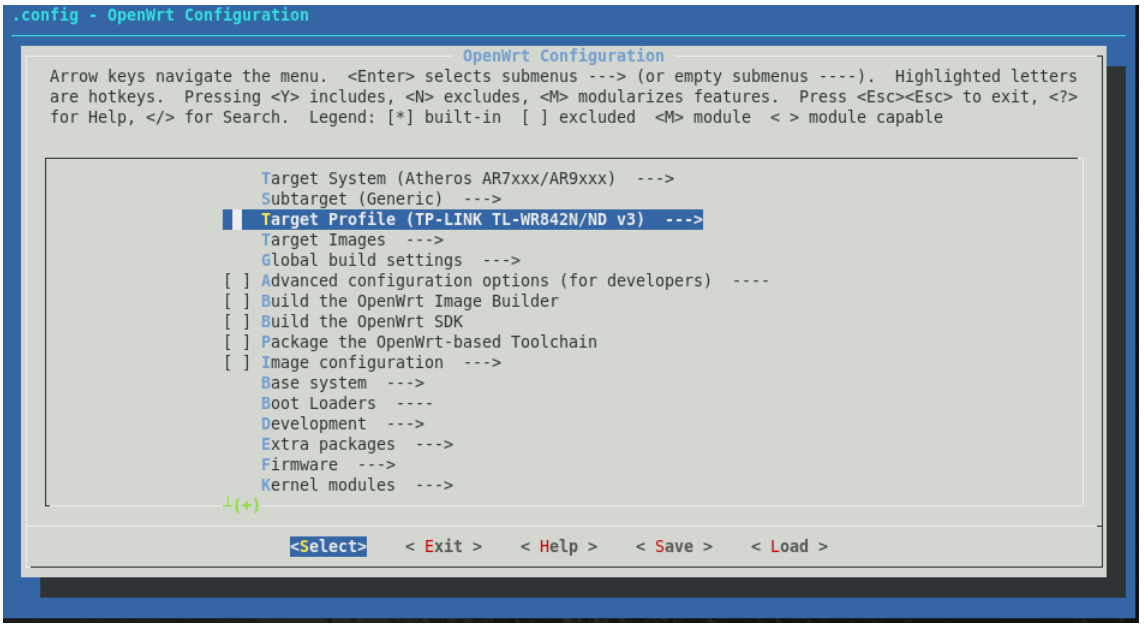


Figure 4.1: menuconfig

Image can be configured with three options: y, m, n which are represented as follows:

- pressing `y` sets the `<*>` built-in label
  This package will be compiled and included in the firmware image file.
- pressing `m` sets the `<M>` package label
  This package will be compiled, but not included in the firmware image file. (E.g. to be installed with opkg after flashing the firmware image file to the device.)
- pressing `n` sets the `< >` excluded label
  The source code will not be processed.

Target system is selected from the extensive list of supported platforms, with the numerous target profiles – ranging from specific devices to generic profiles, all depending on the particular device at hand. In our case, we should browse the `Target Profile` selection and find our targeted device (TP-LINK TL-WR842N/ND v3).

Package selection has the option of either 'selecting all packages', which might be unpractical in certain situations, or relying on the default set of packages will be adequate or make an individual selection. It is here worth mentioning that some package combinations might break the build process, so it can take some experimentation before the expected result is reached. Added to this, the OpenWrt developers are themselves only maintaining

a smaller set of packages – which include all default packages – but, the feeds-script makes it very simple to handle a locally maintained set of packages and integrate them in the build-process.

The final step before the process of compiling the intended image would be to exit the menuconfig tool – this also includes the option to save a specific configuration or load an already existing, and pre-configured, version. Exit the UI, and choose to save the settings. When we save our configuration, the file `.config` will be created in the buildroot directory according to desired configuration [30].

4.5.3 Building single Packages

When developing or packaging software for OpenWrt, it is convenient to be able to build only the package in question (e.g. with package cups):

```
$ make package/cups/compile V=s
```

For a rebuild run:

```
$ make package/cups/{clean,compile,install} V=s
```

It doesn't matter what *feed* the package is located in, this same syntax works for any available package.

If for some reason the build fails, the easiest way to spot the error is to do:

```
$ make package/cups/{clean,compile,install} V=s 2>&1 | \
    tee build.log | grep -i error
```

This pipeline shown invocates the rebuild of the package forwarding the ouput to `build.log` file with `tee`, looking for case insensitive string 'error' with `grep`. Complete guide can be found on OpenWrt Wiki[15].

[15] https://wiki.openwrt.org/doc/howto/build

Chapter 5

Porting the dependencies

Programs don't run in a vacuum but they interact with the outside world. The view of this outside world differs from environment to environment. Things like host-names, 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 become complex fast. If the code depends on some library that is not in the root file-system we need to add it.

Dependencies can be satisfied in two ways: with static libraries or with shared libraries. 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 from the source code on our own. Either way, we end up with one or more binary files and a bunch of header files. There are a few different situations that can happen, but basically everything reduces to having dependencies in buildroot's root file-system.

Having dependencies in a separate folder could be an interesting solution to keep the libraries that we cross-compiled on our own separated from the system libraries. We must then remember to provide to compiler and linker programs the paths where header files and binary files can be found. With static libraries, this information is only needed at compile and link 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 file-system is probably the most common solution. In case of OpenWrt we will use this approach. When everything will be up and running, these libraries must be installed somewhere in the file system of the target platform. It is natural to install dependencies in the target's root file-system, 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. We shall 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) do 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[3].

5.1 Find the dependencies

To port Tang to OpenWrt system we have have all its dependencies available and installed in buildroot. We should do find out if dependencies for our software project, we are about to port, are available for the target's platform. Usually every software project has its pages and all its dependencies should be found there. Let us find these dependencies at Tang's GitHub pages[1] first:

- http-parser
- systemd's socket activation
- José – dependent on:
  - jansson
  - openssl
  - zlib

We will compare versions of all packages required for Tang using rpm information available from the Fedora's *koji* site[2]. All packages available for OpenWrt can be found in OpenWrt GitHub repositories. We will find out that OpenWrt system has already packages openssl, zlib, http-parser, and jansson. The openssl and zlib packages packaged for OpenWrt are in versions already sufficient for Tang.

Search for 'http-parser' does not find this package in base OpenWrt's repository. The reason, is that 'http-parser' is not located in OpenWrt's GitHub repository `openwrt/openwrt` . 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 naming convention is really common in Linux and could be predictable as the `http-parser` is in fact only a library. Compared to *koji* 'http-parser' needs to be updated to version 2.7.1 to satisfy Tang's requirements.

Jansson package was only available in version 2.7 which is too outdated for Tang's dependency José because it requires at least jansson version 2.10.

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 only for the socket activation. We should be able to work with xinetd's socket activation as this package is already packaged for OpenWrt. Finally, as José and Tang will require to add them into package feeds.

Let us clone the fork of `openwrt/packages` repository outside of the buildroot environment:

```
$ git clone git@github.com:Tiboris/packages-OpenWrt.git
```

We will use this fork to have branch for every new package and package change and one special branch `new_pkgs` set up as a feed for buildroot. This will allow us to easily create a upstream pull-request for each package separatelly.

To build the packages we used OpenWrt buildroot[3] with latest upstream commit hash `030a23001b74ede5fa2e6070a8fb04f3feccfbbd`. The OpenWrt buildroot has to have a packages feeds set to point to repository with available OpenWrt packages. We used upstream packages[4] with latest commit hash `580053888235713dd95b96b37169926bffedce0b`.

[1] https://github.com/latchset/tang
[2] https://koji.fedoraproject.org/koji/rpminfo?rpmID=10772363
[3] https://github.com/openwrt/openwrt
[4] https://github.com/openwrt/packages

5.2 Update outdated packages

Finding some of dependencies already available for our desired target platform will definitely make us satisfied. We would agree that starting with something already built for OpenWrt is the best thing to do when we are approaching unknown platform. In following subsections we will use all things we know from section 4.5. Let us start with missing update of José's dependency, package jansson.

5.2.1 Update jansson

Jansson is a C library for encoding, decoding and manipulating JSON data. The latest release of the jansson is v2.11 released on 11th of February 2018[15].

However at the beginning of the Tang porting effort the latest release was v2.10. To update mentioned OpenWrt available jansson package version v2.7 to 2.10 change to package's fork directory and create a new branch for changes:

```
$ cd ~/packages-OpenWrt
$ git checkout -b jansson-update
```

In order to tell the OpenWrt buildroot how to build a program we need to create a special Makefile in the appropriate directory. The appendix E OpenWrt package's Makefile illustrate its content. We shall now find the jansson package Makefile in the repository using for example:

```
$ git grep jansson | grep PKG_NAME
libs/jansson/Makefile:PKG_NAME:=jansson
```

The *PKG_NAME* variable identifies the package for OpenWrt buildroot[21]. List of all available variables can be found on the OpenWrt wiki page[5]. We can assume that the jansson library related files are located in `libs/jansson` directory of the packages repository. Now we can update the Makefile.

We will find variable *PKG_VERSION* and change the old version number (in our case 2.7) to new version number (2.10). This edit will result into changing the *PKG_SOURCE* variable in Makefile. Variables *PKG_SOURCE* and *PKG_SOURCE_URL* are used to identify location of the archive with sources for specified version from where the sources would be downloaded. After the downaload the OpenWrt buildroot checks the file integrity. The *PKG_HASH* and *PKG_MD5SUM* variables serve this purpose. As the new version of archive with sources will be downloaded we need to change *PKG_HASH*/*PKG_MD5SUM* variable as well. For some reason, OpenWrt upstream developers required the use of *PKG_HASH* variable and the bz2 archive for the jansson package. In general it would be sufficient to change only version and the appropriate *PKG_HASH*/*PKG_MD5SUM* variable. We changed the filename extension for *PKG_SOURCE* variable from ".tar.gz" to ".tar.bz2". Now download the archive containing sources and get the archive hash using sha256sum:

```
$ wget http://www.digip.org/jansson/releases/jansson-2.10.tar.bz2 -P /tmp
$ sha256sum /tmp/jansson-2.10.tar.bz2 | cut -d " " -f1
241125a55f739cd713808c4e0089986b8c3da746da8b384952912ad659fa2f5a
```

Last but not least commit the changes and push the changes into our fork.

```
$ git commit -a
$ git push --set-upstream origin jansson-update
```

[5] https://wiki.openwrt.org/doc/devel/packages

Before submitting the pull request we should try to build updated package first. The following step is not necessary because update of jansson package has been already merged upstream[6] where the diff can be wieved. But to do so, let us remember that we have special branch set up for buildroot feeds – the `new_pkgs` branch. In general to test updated or new packages we will use this branch to merge our newly created branch into it using:

```
$ git checkout new_pkgs
$ git merge jansson-update
```

After successful merge we have updated jansson package available in our custom feeds. Trigger update with feeds script and make updated jansson available in menuconfig with:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install jansson
```

To finally build updated package we shall run the command:

```
$ make package/jansson/{clean,compile} V=s
```

After successful build the most important part would be to contribute changes to the upstream as we already did through merged pull-request mentioned above. With a knowledge of buildroot and the contribution guidelines effort spent on such updates may be quite minimal.

5.2.2 Update http-parser

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

The latest release available was version v2.7.1 until 9th February 2018 release of v2.8.0 which update will be demonstrated below. Fedora is still using version v2.7.1 with Tang server so compared to last available version on OpenWrt which was v2.3.0 an update is needed. Hoping for the best we first try to update the libhttp-parser to version v2.7.1 to match Fedora version similar way as with jansson. Only updating version of the package may suffice but the case of libhttp-parser as dependency was special as you will notice in subsection 6.2.1 Obstacles of delivering Tang. To upgrade this package we will do same as with jansson package. First make sure that we are still in packages repository and create branch for the change:

```
$ git checkout -b libhttp-parser-update master
```

Let us locate the http-parser Makefile:

```
$ git grep http-parser | grep PKG_NAME
libs/libhttp-parser/Makefile:PKG_NAME:=libhttp-parser
```

Find out the source url to download the archive containing and get the archive hash using sha256sum:

```
$ wget https://github.com/nodejs/http-parser/archive/v2.8.0.tar.xz -P /tmp
$ sha256sum /tmp/v2.8.0.tar.gz | cut -d " " -f1
83acea397da4cdb9192c27abbd53a9eb8e5a9e1bcea2873b499f7ccc0d68f518
```

Please note the file extension in old makefile and download same file-type for upgrade.

Before committing the changes we also found that owner of the repository changed from 'joyent' to 'nodejs' so we addressed these changes as well by editing proper sections in the Makefile. We shall now commit the changes and push them into our fork.

[6] https://github.com/openwrt/packages/pull/4289/files
[7] https://github.com/nodejs/http-parser

```
$ git commit -a
$ git push --set-upstream origin libhttp-parser-update
```

Again following step to merge the libhttp-parser-update branch with feeds branch is not necessary because the submitted pull-request[8] containing these changes has been already merged to the upstream:

```
$ git checkout new_pkgs
$ git merge libhttp-parser-update
```

After package is available in our feeds trigger update with feeds script and make new version of libhttp-parser package available in menuconfig:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install libhttp-parser
```

Finally build an updated libhttp-parser running:

```
$ make package/libhttp-parser/{clean,compile} V=s
```

Unfortunately as we will see in subsection 6.2.1 Obstacles of delivering Tang, the successful build of the updated package may not be enough. Especially when built package is also a dependency for other packages.

5.3 New package José

After updating of jansson and libhttp-parser we are familiar with OpenWrt's Makefiles. As José package is not packaged for OpenWrt now comes the time to write new makefile on our own.

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

- RFC 7520 - Examples of JSON Object Signing and Encryption (JOSE) [18]
- RFC 7515 - JSON Web Signature (JWS) [9]
- RFC 7516 - JSON Web Encryption (JWE) [11]
- RFC 7517 - JSON Web Key (JWK) [8]
- RFC 7518 - JSON Web Algorithms (JWA) [7]
- RFC 7519 - JSON Web Token (JWT) [10]
- RFC 7638 - JSON Web Key (JWK) Thumbprint [12]

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[16].

[8] https://github.com/openwrt/packages/pull/5446

5.3.1 Create José

First, let us create feature branch and directory for new package:

```
$ git checkout -b libhttp-parser-update master
$ mkdir -p utils/jose
```

It does not matter whether new Makefile will be placed whe the libs or utils for the build purposes. We can simply change it as upstream developers would require.

To start with such a work it is good to have some kind of the template. For the José we used the jansson's Makefile as a template. Place this "template" Makefile into `utils/jose` directory and start editing.

Let us go through it from the top to the bottom. This is first non comment line in the file:

```
include $(TOPDIR)/rules.mk
```

Without this include our Makefile would not work so we will leave it as it is. The next are the package name, version and release variables. This have to be the first thing to edit:

```
PKG_NAME:=jose
PKG_VERSION:=10
PKG_RELEASE:=1
```

As we defined the version of the package which we desire, we should visit the José project pages and browse for the release archive. José's upstream releases lives on GitHub[9]. Visit the site and copy link location of the tar.bz2 archive for José release 10. Now download the archive and as we did when updating packages run sha256sum:

```
$ wget -P /tmp \
https://github.com/latchset/jose/releases/download/v10/jose-10.tar.bz2
$ sha256sum /tmp/jose-10.tar.bz2 | cut -d " " -f1
5c9cdcfb535c4d9f781393d7530521c72b1dd81caa9934cab6dd752cc7efcd72
```

This manual step is reflected in the Makefile as shown:

```
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2

PKG_SOURCE_URL:=\
https://github.com/latchset/$(PKG_NAME)/releases/download/v$(PKG_VERSION)/

PKG_HASH:=\
5c9cdcfb535c4d9f781393d7530521c72b1dd81caa9934cab6dd752cc7efcd72
```

The *PKG_SOURCE* variable now contains the value of the source archive name. The *PKG_SOURCE_URL* provides the whole path to archive stored on GitHub server and the *PKG_HASH* is used to verify the file integrity.

```
PKG_INSTALL:=1
PKG_BUILD_PARALLEL:=1

PKG_FIXUP:=autoreconf
include $(INCLUDE_DIR)/package.mk
```

Setting *PKG_INSTALL* to "1" will call the package's original "make install" to the directory defined with *PKG_INSTALL_DIR* variable. The *PKG_FIXUP* performs the important `autoreconf -f -i` for project using autotools. And again without including makefile package.mk the buildroot would not now how to continue build. It is a good practice to divide it into two packages the library and the tool:

[9] https://github.com/latchset/jose/releases

```
define Package/libjose
  SECTION:=libs
  TITLE:=Provides a full crypto stack...
  DEPENDS:=+zlib +jansson +libopenssl
  URL:=https://github.com/latchset/jose
  MAINTAINER:=Tibor Dudlak <tibor.dudlak@gmail.com>
endef

define Package/jose
  SECTION:=utils
  TITLE:=Provides a full crypto stack...
  DEPENDS:=+libjose +zlib +jansson +libopenssl
  URL:=https://github.com/latchset/jose
  MAINTAINER:=Tibor Dudlak <tibor.dudlak\@gmail.com>
endef
```

To add description for both packages we should add defines to Makefile with proper text:

```
define Package/jose/description
    ... description text ...
endef

define Package/libjose/description
    ... description text ...
endef
```

The *Build/Configure* section can be skipped if the source doesn't use configure or has a normal config script, otherwise we can put our own commands here or use *$(call Build/Configure/Default,)* to pass in additional arguments after the comma for a standard configure script.

```
define Build/Configure
    $(call Build/Configure/Default)
endef
```

Every library package should have section *Build/InstallDev*. This section is important for linker and buildsystem especially to work correctly when library would be used as dependency (static libs, header files) for other tools or packages. This section has no use on the target device.

```
define Build/InstallDev
    $(INSTALL_DIR)  $(1)/usr/lib
    $(INSTALL_DIR)  $(1)/usr/include
    $(INSTALL_DIR)  $(1)/usr/include/$(PKG_NAME)
    $(INSTALL_DIR)  $(1)/usr/lib/pkgconfig
    $(CP)   $(PKG_INSTALL_DIR)/usr/lib/lib$(PKG_NAME).so*   $(1)/usr/lib
    $(CP)   $(PKG_INSTALL_DIR)/usr/include/$(PKG_NAME)/*.h  \
            $(1)/usr/include/$(PKG_NAME)
    $(CP)   $(PKG_BUILD_DIR)/*.pc   $(1)/usr/lib/pkgconfig
endef
```

Section it Package/<package_name>/install contains set of commands to copy files into the device file-system represented by the *$(1)* directory. As source we can use relative paths which will install from the unpacked and compiled source, or *$(PKG_INSTALL_DIR)* which is where the files in the Build/Install (not used here) step end up.

```
define Package/libjose/install
    $(INSTALL_DIR)  $(1)/usr/lib
    $(CP)   $(PKG_INSTALL_DIR)/usr/lib/lib$(PKG_NAME).so*   $(1)/usr/lib/
endef
```

```
define Package/jose/install
    $(INSTALL_DIR)  $(1)/usr/bin
    $(INSTALL_BIN)  $(PKG_INSTALL_DIR)/usr/bin/$(PKG_NAME)  $(1)/usr/bin/
endef
```

At the bottom of the file we put *BuildPackage*, a macro setup by the earlier include statements. BuildPackage only takes one argument directly – the name of the package to be built. All other information is taken from the define blocks.

```
$(eval $(call BuildPackage,libjose))
$(eval $(call BuildPackage,jose))
```

At this point our first Makefile for José is ready. Let us merge this changes to feeds and try to build our freshly created package.

```
$ git commit -a
$ git push --set-upstream origin add-jose
$ git checkout new_pkgs
$ git merge add-jose
```

In buildroot run:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install jose
$ make menuconfig
$ make package/jose/{clean,compile}
```

Note that feeds script with an install option will install also missing dependencies of José to be available in buildroot. The new packages will be available in the *Extra packages* section of the menuconfig as you can see on Figure 5.1 Extra packages. José's build takes some

Figure 5.1: Extra Packages

time, since it has openssl library as dependency which also needs to be compiled. The pull request for adding the José package to OpenWrt can be found in its repository[10].

[10] https://github.com/openwrt/packages/pull/4334

Chapter 6

Porting Tang

After successful cross-compilation of José we have all the dependencies packaged except the systemd. The systemd is only one of the many implementations (inetd, launchd, ucspi-tcp, xinetd) of a server providing socket activation.

6.1 Socket activation

Socket activation is a technology provided by a super-server (also called a service dispatcher daemon). It uses very few resources when in idle state and starts other services when needed as well, normally with access to them checked by a TCP wrapper. A service designed for the socket activation would behave as bare CLI application with input read from *stdin* (standard input) and output written to *stdout* (standard output). Tang is exactly this kind of an application and because of that we need to configure socket activation

The xinetd implementation is already packaged for OpenWrt and we will try to configure the Tang to use it later.

6.1.1 xinetd

xinetd listens for incoming requests over a network and launches the appropriate service for that request. Requests come to network ports and xinetd usually launches another daemon to handle the request. This is reflected on Figure 6.1 xinetd socket activation below. xinetd features access control mechanisms such as TCP Wrapper ACLs (access

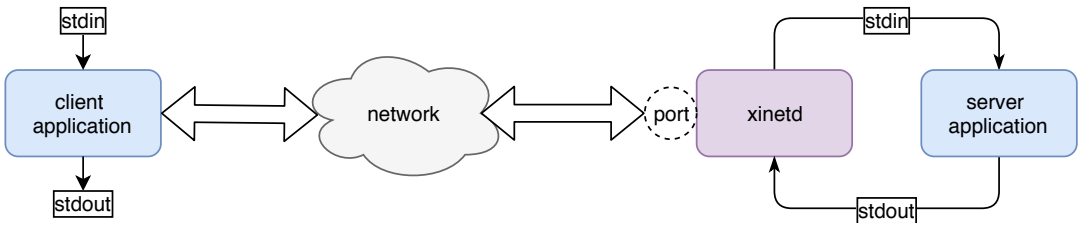


Figure 6.1: xinetd socket activation

control lists), extensive logging capabilities, and the ability to make services available based on time. It can place limits on the number of servers that the system can spawn. xinetd is listening on behalf of the services. Whenever a connection would come in an instance of the respective service will be spawned with using *stdin* and *stdout* of the service application[2].

6.2 Package the Tang

Similarly to José we need to create a new package for OpenWrt. Let us create a branch and `utils/tang` directory where binary programs like Tang belongs to:

```
$ git chechout -b add-tang master
$ mkdir -p utils/tang/
```

The Tang project is owned by same owner on GitHub as José. We should visit the project releases page[1] and get the Tang version v6. Then add following lines to the Makefile similarly as with José's Makefile:

```
include $(TOPDIR)/rules.mk

PKG_NAME:=tang
PKG_VERSION:=6
PKG_RELEASE:=1

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_SOURCE_URL:=\
https://github.com/latchset/$(PKG_NAME)/releases/download/v$(PKG_VERSION)/

PKG_HASH:=1df78b48a52d2ca05656555cfe52bd4427c884f5a54a2c5e37a7b39da9e155e3


PKG_INSTALL:=1
PKG_BUILD_PARALLEL:=1

PKG_FIXUP:=autoreconf

include $(INCLUDE_DIR)/package.mk
```

Do not forget to add the package description which should have section dependencies filled.

- libhttp-parser – used for parsing HTTP requests.
- José – the library and tool for the JavaScript Object Signing and Encryption.
- xinetd (a run-time dependency)
- bash (a run-time dependency)

The actual build proccess of the Tang does not require xinetd's libraries but will be configured to use its socket activation available in run-time. The bash dependency is there for a reason that Tang's `tangd-update` and `tangd-keygen` executables are bash scripts. These scripts are complex and are using data structures that are not available for OpenWrt's default shell - ash. Having run-time dependencies listed in the Makefile will ensure that they are installed to the device before the Tang.

```
define Package/tang
  SECTION:=utils
  TITLE:=tang v$(PKG_VERSION) - daemon for binding data to a third party
  DEPENDS:=+libhttp-parser +xinetd +jose +bash
  URL:=https://github.com/latchset/tang
endef
```

The Tang package will be present in utils section of the `openwrt/packages` repository. Let us add a brief description to our new package using description define:

[1] https://github.com/latchset/tang/releases/

```
define Package/tang/description
    Tang is a small daemon for binding data to the presence of a third party
endef
```

The buildroot should know where to install the Tang's binaries. Let us define a install section and use standard tangd binary location as on Fedora OS:

```
define Package/tang/install
    $(INSTALL_DIR)  $(1)/usr/libexec
    $(INSTALL_BIN)  \
            $(PKG_INSTALL_DIR)/usr/lib/$(PKG_NAME)d*    $(1)/usr/libexec/
endef
```

Let us not forget the last line which allows the actual „magic“ to happen:

```
$(eval $(call BuildPackage,$(PKG_NAME)))
```

We can now merge these changes to feeds and try to build our freshly created Tang package:

```
$ git commit -a
$ git push --set-upstream origin add-tang
$ git checkout new_pkgs
$ git merge add-tang
```

The new packagee tang will be available in the *Extra packages* section of the menuconfig after updating feeds:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install tang
$ make menuconfig
$ make package/tang/{clean,compile}
```

After first try to build the tang package we will encounter the systemd dependency errror:

```
configure: error: Package requirements (systemd) were not met:

No package 'systemd' found

Consider adjusting the PKG_CONFIG_PATH environment variable if you
installed software in a non-standard prefix.
```

We did not defined systemd dependency for Makefile but the cross-compilation of the package will try to configure and compile downloaded sources. Compiler will try to find the systemd dependency as it is defined as dependency in the `configure.ac` file in Tang repository. We shall remove this builtime dependency.

To do so we will remove a requirement for systemd from the Tang's configure.ac and Makefile.am file. These patches are too extensive to be demonstrated. `Makefile_am.patch` and `configure_ac.patch` can be found in submitted pull-request files[2] on GitHub.

To have sources patched before compilation we have to crate a directory for them in the package's feeds repository we forked on branch containing commit adding the Tang and copy them to created directory:

```
$ cd packages-OpenWrt
$ git checkout add-tang
$ mkdir -p utils/tang/patches
```

Patches included in this directory are automatically applied on the sources downloaded from the mirror in the build time.

[2] https://github.com/openwrt/packages/pull/5447/files

To have these changes in our feeds in buildroot commit them and push to add-tang branch. After these changes pushed and merged with *new_pkgs* branch a rebuild of the tang package we will succeed. Now we have Tang package ready to be installed on our device.

We are avoiding troubles with using the libhttp-parser in version 2.8.0. These problems are described in following subsection 6.2.1. The most important part after successful build would be to configure it correctly.

6.2.1 Delivering Tang package

The upstream world is not always ideal. Soon after this porting effort started OpenWrt upstream was in bad shape and almost dead for reasons we described in subsection **??** OpenWrt and LEDE. The active part of OpenWrt developers decided to focus on LEDE project and submitting a pull-request to the OpenWrt was painful. Working with outdated buildroot (or an SDK) was not ideal. We decided to use upstream version of buildroot after re-merge. It solved many issues with outdated dependencies that came into way.

Outdated buildroot caused many issues with dependencies but as we installed older version of them the build could be triggered. The most time consuming thing was to run build over and over and collect linker and compiler errors and adding additional flags into Makefile such as:

```
+CFLAGS += -fhonour-copts
+TARGET_CFLAGS += $(FPIC) -std=gnu99
+TARGET_LDFLAGS += -Wl,-rpath-link=$(1)/usr/lib
```

Same flags and running build over an over were happening with José and with the Tang. In case of Tang there was one additional TARGET_CFLAGS option:

```
-D_GNU_SOURCE
```

Without this option Tang was issuing a compilation errors with implicit declaration of the functions:

```
dprintf()
vdprintf()
```

Using uptream buildroot solved these issues for good.

libhttp-parser dependency has been in its latest version 2.7.1 when the effort started. Unfortunately building the Tang package with libhttp-parser updated to version 2.7.1 failed on dependency and has thrown an error:

```
checking for http_parser.h... no
configure: error: http-parser required!
```

We found out that Fedora's package http-parser contains one patch[3] to add functionality required by the Tang to the http-parser. Actual checking of the header was in configure.ac file of the Tang package:

```
AC_CHECK_HEADER([http_parser.h], [],
        [AC_MSG_ERROR([http-parser required!])], [
#include <http_parser.h>
#ifndef HTTP_STATUS_MAP
```

[3] https://github.com/nodejs/http-parser/pull/337

```
#error HTTP_STATUS_MAP not defined!
#endif
])
```

We first considered applying the very same patch form Fedora package to the http-parser in version 2.7.1 in OpenWrt but in the end the release 2.8.0 solved the dependency error for a *HTTP_STATUS_MAP* macro. On the other hand it brought another issue:

```
Package tang is missing dependencies for the following libraries:
libhttp_parser.so.2.8
```

Adding a symbolic link to the libhttp-parser's install sections of the Makefile will suffice.

```
ln -s libhttp_parser.so.$(PKG_VERSION) libhttp_parser.so.2.8
```

Chapter 7

Configuring the Tang on OpenWrt

Having the installable packages in our buildroot is only the half of the work done. We need to install them on the actual device and setup the environment especially for the Tang server to work correctly. The installation of the packages on OpenWrt is done with the opkg package manager.

The opkg (Open Package Management System) is a lightweight package manager used to download and install OpenWrt packages. These packages could be stored somewhere on device's file-system 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 dnf, yum, apt/apt-get, pacman, emerge etc. will recognize the similarities. It also has similarities with NSLU2's Optware, also made for embedded devices.

Opkg attempts to resolve dependencies with packages in the available repositories/mirrors. If the opkg fails to find the dependency, it will report an error, and abort the installation of selected package.

By removing systemd lines from sources we end up not have automatic updates of the Tang's cache and xinetd's socket activation to set up.

7.1 Install the packages

The packages that have been built in our buildroot are not available in any online mirror yet. To make them available for OpenWrt we should create a pull-request tith the change and work with the community to accept it. Before we do so, we have to make sure, that the built packages are working try installing them and testing their functionality.

To get our newly built packages to the target device running the OpenWrt we can upload them using scp to device's file-system. After successful build, the packages are present in the bin/packages/mips_24kc/packages/ directory (the location is bound to device configuration):

```
$ scp bin/packages/mips_24kc/packages/*.ipk \
    root@192.168.0.1:/root/custom-packages/
```

This command will upload every package built before in builroot to the device's directory `/root/custom-packages/`. The buildroot should contain at least these files:

```
$ ls bin/packages/mips_24kc/packages/
bash_4.4.12-1_mips_24kc.ipk
jansson_2.10-1_mips_24kc.ipk
jose_10-1_mips_24kc.ipk
libhttp-parser_2.8.0-1_mips_24kc.ipk
```

```
libjose_10-1_mips_24kc.ipk
Packages
Packages.gz
Packages.manifest
Packages.sig
tang_6-1_mips_24kc.ipk
xinetd_2.3.15-5_mips_24kc.ipk
```

After the successful upload, connect to the device and install packages. We recommend installing only newly built or updated packages. opkg will resolve known dependencies from the mirrors and will install them. The `/root/custom-pacakges` is not set up as custom opkg feed, thus we need to install tang and ist depencencies manually in order:

```
opkg install /root/custom-packages/jansson_2.10-1_mips_24kc.ipk
opkg install /root/custom-packages/jose_10-1_mips_24kc.ipk
opkg install /root/custom-packages/libhttp-parser_2.8.0-1_mips_24kc.ipk
opkg install /root/custom-packages/tang_6-1_mips_24kc.ipk
```

The opkg tool will resolve other known dependencies and install them as well. After packages are installed we may now proceed to the enviroment setup.

7.2 Setting up the Tang keys

The Tang server is packaged with cripts which help us to generate keys and cache for the Tang daemon. OpenWrt has no realpath available, therefore the `tangd-update` script did not work on OpenWrt.

```
bash: realpath: command not found
```

Simply replacing it with `readlink -f` solved the issue and script with such change is capable of generating cache. We have to create a diff file for this change and add it into `utils/tang/patches` directory.

The OpenWrt Makefile can define section *Package/$(PKG_NAME)/postinst*. This section usually contain a short shell script to tweak the package after installation to make package work out of the box. We can place this short sript to the Tang's Makefile to run the keys and cache generation after installation:

```
define Package/tang/postinst
#!/bin/sh
if [ -z "$${IPKG_INSTROOT}" ]; then
    mkdir -p /usr/share/tang/db && mkdir -p /usr/share/tang/cache
    KEYS=$(find /usr/share/tang/db/ -name "*.jw*" -maxdepth 1 | wc -l)
    if [ "${KEYS}" = "0" ]; then # if db is empty generate new key pair
        /usr/libexec/tangd-keygen /usr/share/tang/db/
    elif [ "${KEYS}" = "1" ]; then # having 1 key should not happen
        (>&2 echo "Please check the Tang's keys in /usr/share/tang/db \
and regenate cache using /usr/libexec/tangd-update script.")
    else
        /usr/libexec/tangd-update /usr/share/tang/db/ /usr/share/tang/cache/
    fi
fi
endef
```

7.3 Configure Tang for xinetd

We need xinetd's socket activation for Tang to work. And to do so we will need a configuration file for the tangd service for xinetd daemons shown in listing 7.1.

```
service tangd
{
    port            = 8888
    socket_type     = stream
    wait            = no
    user            = root
    server          = /usr/libexec/tangd
    server_args     = /usr/share/tang/cache
    log_on_success  += USERID
    log_on_failure  += USERID
    disable         = no
}
```

Listing 7.1: Configuration of Tang service for xinetd

This is configuration to run `/usr/libexec/tangd` after a request comes to port 8888. The server will be spawned with one argument, the cache directory. Please note that we used diffent directory compared to Fedora. We need to have Tang keys stored on persistent data storage. The `/var/` location is only a symbolic link to the `/tmp` directory on OpenWrt device. The developers on IRC channel proposed to use the persistent `/usr/share/` directory for that purpose. The last step before starting the Tang service on OpenWrt would be to setup `/etc/services`:

```
# echo -e "tangd\t\t8888/tcp" >> /etc/services
```

and add the very same thing to post-installation script to have Tang completelly set up after installation:

```
(cat /etc/services | grep -E "tangd.*8888\/tcp") > /dev/null \
    || echo -e "tangd\t\t8888/tcp" >> /etc/services
```

In case of the accidental removal of `/etc/services` file we have to copy the backup of the file from devices ROM and edit it again.

```
# cp /rom/etc/services /etc/services
```

Now we shall restart the xinetd daemon using the xinetd script located in `/etc/init.d/` directory to enable tangd service using xinetd's socket activation:

```
# /etc/init.d/xinetd stop
# /etc/init.d/xinetd start
```

To test that service is running we can use the telnet to the device on port defined in the xinetd configuration. The server should advertise its public key. It can be retrieved with simple *GET* request for */adv* content from the server. Try telnet to device write "GET /adv HTTP/1.1" and confirm this with two newlines (return key). The output similar to following should appear:

```
$ telnet 192.168.0.1 8888
Trying 192.168.0.1...
Connected to 192.168.0.1.
Escape character is '^]'.
GET /adv HTTP/1.1

<unknown> GET /adv => 200 (src/tangd.c:85)
```

```
HTTP/1.1 200 OK
Content-Type: application/jose+json
Content-Length: 956

{"payload":"eyJrZXlzIjpbeyJhbGciOiJFQ01SIiwiY3J2IjoiUC01MjEiLCJrZXlfb3BzIjpb
ImRlcml2ZUtleSJdLCJrdHkiOiJFQyIsIngiOiJBR3V2amxUZmpYaDBraWFEa19Tak1vMGhYUm1R
dzFZNkVkNE9yN3Fza2J2c1h6QUlvRTl5ZnRwR2xRVng1OVlxZ1gtR3hQSE8tdzVLVXFmanRGQkVV
ZVByIiwieSI6IkFiNE9NNTBhQ1Y4NkdZVW1PdHBua1VTanNXNUFleENXZG5PZFEyakl0Z1RGNXNq
MG1TSFZCYXhsd2w1N3ZTUEdrSExiRl96SFlUVzlvVzNoTFJyeXRRNmYifSx7ImFsZyI6IkVTNTEy
IiwiY3J2IjoiUC01MjEiLCJrZXlfb3BzIjpbInZlcmlmeSJdLCJrdHkiOiJFQyIsIngiOiJBQVpS
ZmNlNUhLaFYyck1OQzhqLW5iVW1pdlh4NDRjcU1qX2Jvbk5OdnNTcXdBRjhFcGoyTFM0cFpfdUNR
VDJGVDRUSHJnX1Y4VHBKci1PQW41Z05CaUZNIiwieSI6IkFTMjJSdVJZZXVOU1ZtSkdfSmcwSW1n
by1LREd2QVFPZV9Vdk9fT3RZS3lGeUVoSWI1U0FLd3cwSEF0QjJFX2FTMG5pcWV3UUlod1QyanR5
eklCaWdkQU0ifV19","protected":"eyJhbGciOiJFUzUxMiIsImN0eSI6Imp3ay1zZXQranNvb
iJ9","signature":"ADpGOjYeB45MznQOxA6Pw9MXTMYQ649UkRSi_RsP8KPKosl-eA7GmOIiBM
FOoPnCNX-cGjhyDBbQuvESUCJ_3txjANf-srntxFAX5p72Eip-kROGrlCdLrVLnlg36itQUBFx7S
UyB_7I6CX6gdpWyJ-wtro8f2Snu6wwMGl-8V3ylWYr"}
```

The content of the first line from server response should be suspicious to us. According to the RFC 2616 the HTTP header should not contain such line[4].

```
<unknown> GET /adv => 200 (src/tangd.c:85)
```

After some investigation done in the Tang sources we found out that the line contains debug information from the Tang daemon which were written to the /dev/stderr. From this we can assume that xinetd is sending also `/dev/stderr` to the network socket. To bypass this beavior we should write wrapper script which will redirect the `/dev/stderr` output somewhere else. We decided that this output may be helpful for debugging purpose and forwarded it to the log file in the `/var/log/` directory of the device using this script:

```
#!/bin/bash
echo "==================================" >> /var/log/tangd.log
echo `date`: >> /var/log/tangd.log
/usr/libexec/tangd $1 2>> /var/log/tangd.log
```

Let us name this wrapper script the tangdw.

To get Tang to work we need to put invocation of this script in place where tangd binary was configured to. Move the script to the /usr/libexec directory where the tangd binary is to have it in one place and edit the line of xinetd configuration to run wrapper:

```
    server          = /usr/libexec/tangdw
```

The last thing would be to change the script's permissions on device to allow execution.

We did a lot changes to the configuration. The Tang for the OpenWrt platform needs a xinetd configuration which for the ease of use could be packaged with it. The same with the wrapper script in order to have the service working correctly we also need this file. There is a way to add new files for each package and package them for the OpenWrt's opkg to install. To do so we will crate a directory for the files in the package's feeds repository we forked on branch containing commit adding the Tang and copy files there.

```
$ cd packages-OpenWrt
$ git checkout add-tang
$ mkdir -p utils/tang/files
```

As the new files are copied there we need to also edit a *Package/tang/install* section to install these files in proper directories (config file into dedicated `/etc/xinetd.d/`; wrapper into `/usr/libexec/`) as shown:

```
define Package/tang/install
    $(INSTALL_DIR)  $(1)/usr/libexec
    $(INSTALL_DIR)  $(1)/etc/xinetd.d/
    $(INSTALL_BIN)  \
            $(PKG_INSTALL_DIR)/usr/libexec/$(PKG_NAME)d*  $(1)/usr/libexec/
    $(INSTALL_BIN)  ./files/tangdw  $(1)/usr/libexec/
    $(CP)           ./files/tangdx  $(1)/etc/xinetd.d/
endef
```

After all changes done last, we need to to amend commit on a branch, update the feeds in buildroot, rebuild the Tang package, upload and re-install it on our device running the OpenWrt. Due to many changes to the Makefile and the Tang branch in packages see the submitted pull-request[1] which includes all the changes.

7.4 Binding to the Tang on OpenWrt device

Let us now bind the client to the Tang server which is up and running on our OpenWrt:

```
# clevis luks bind -d /dev/nvme0n1p2 tang '{"url":"http://192.168.0.1:8888"}'
The advertisement contains the following signing keys:

Apb39FO1vey9FyUe_fEd8lVDABs

Do you wish to trust these keys? [ynYN] y
Enter existing LUKS password:
```

Clevis will add a new key encryption key to the first available *LUKS* header key slot. After rebuilding the initramfs on our Fedora client the early boot decryption will work.

Clevis client does not have option to unbind encrypted volume bound to Tang server. We have to make sure that the key encryption key we are about to remove from the *LUKS* header key slot is one added by Clevis. To do so please run `cryptsetup` with option *luksDump* before and after binding with the Tang.

```
# cryptsetup luksDump /dev/nvme0n1p2
```

The manual step to remove key from key slot was demonstrated in subsection 2.3.3:

```
# cryptsetup luksKillSlot /dev/nvme0n1p2 1
Enter any remaining passphrase:
```

7.5 Tang's limitations

Let us sum up the limitations that Tang has on OpenWrt platform due to replaced systemd dependency with less capable xinetd implementation of super-server. We also found one platform independent limitation of the Tang's solution to full disk description on early boot.

Key exchange and generation of cache must be manual but can be automated with inotifytools to watch directory for change and run update script automatically. But this mean that the embedded device with such configuration will have another process always running. We decided to not implement automated update of the cache to save the computation resources.

[1] https://github.com/openwrt/packages/pull/5447

xinetd forwarding the stderr output to the socket as well is a little problem. To have tang working correctly we wrote a wrapper running the shell script redirecting the *stderr* into log file. Running some shell script takes some time compared to only running the binary file but for xinetd it is necessary.

Early boot decryption using Wi-Fi network is a platform independent chicken-egg problem caused by information about wireless network being encrypted on system volume. The information needed to connect to wireless network needs to be decrypted but in order to decrypt them automatically we need to connect to network exposed to Tang server. It could be solved using a TPM to store information about such network but it is a security vulnerability and must be considered with a caution.

Chapter 8

Conclusion

Porting to OpenWrt platform is done using cross-compilation tools available throughout the OpenWrt's buildroot. Buildroot is configured using the menuconfig tool for specific device supported by the OpenWrt system. Buildroot is capable to create packages and also the Image of the OpenWrt system ready to be installed to our device. Using outdated buildroots (SDKs) may cause some troubles but it is recommended to use an SDK while porting to older OpenWrt system release.

To port software to the OpenWrt the user have to create a Makefile in a feed corresponding to the software usage. After creating the Makefile for the OpenWrt buildroot the actual build of the package should be triggered before proposing changes to the OpenWrt upstream. If the build cucceeded we have package ready to be tested on our device.

It happens that software requires a dependencies. To be able to build our desired software application we have to check whether these dependencies are available for our target platform, the OpenWrt. In our case Porting tang required to update existing packages in the OpenWrt feeds to newer versions and creating a new dependency package as well.

Every OpenWrt package can be modified before build time. This is done having a new files and diff files in package's directory. Diff files, used to change original sources, are automatically applied when placed in proper diffs directory. New files located in files directory of the package should be processed by directives in package's Makefile.

After successful build of the Tang package we needed also to configure it. The socket activation on xinetd is working similarly like systemd's. We created the new configuration file for tangd service and the wrapper script. Having post install script in the Tang's Makefile will ensure that the Tang will have generated files required for run-time. The only thing that user has to do is to re-start the xinetd daemon.

Every change we did to acomplish goal of this thesis is reflected in separated pull requests against openwrt/packages upstream. The update of the jansson package can be found in pull-request with number 4289[1]. A pull-request containing update of libhttp-parser package has number 5446[2]. New packages for the OpenWrt José and Tang are waiting for community to accept. José can be found in pull-request with number 4334[3]. And finally Makefile for the Tang server package in pull-request 5447[4].

[1] https://github.com/openwrt/packages/pull/4289/files
[2] https://github.com/openwrt/packages/pull/5446/files
[3] https://github.com/openwrt/packages/pull/4334/files
[4] https://github.com/openwrt/packages/pull/5447/files

With correct configuration of xinetd daemon for Tang's socket activation, the Tang server is working on OpenWrt with some platform specific changes and limitations, able to serve its clients on the network.

Bibliography

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

[2] Braun, R.: *xinetd*. [Online] Accessed 1 May 2017.
Retrieved from:
http://web.archive.org/web/20051227095035/http://www.xinetd.org:80/

[3] Dini, F.: *cross compile tutorial*. [Online] Accessed 27 April 2018.
Retrieved from: http://www.fabriziodini.eu/posts/cross_compile_tutorial/

[4] Fielding, R.; Gettys, J.; Mogul, J.; et al.: Hypertext Transfer Protocol – HTTP/1.1. Technical Report 2616. June 1999. obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.
Retrieved from: http://www.ietf.org/rfc/rfc2616.txt

[5] Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.1. *Changes*. vol. 1. 2005: pp. 22–01,. [Online] Accessed 22 April 2018.
Retrieved from: http://tomb.dyne.org/Luks_on_disk_format.pdf

[6] Gulwani, S.; Hernández-Orallo, J.; Kitzelmann, E.; et al.: *Inductive programming meets the real world*. *Communications of the ACM*. vol. 58, no. 11. 2015: pp. 90–99. ISSN 0001-0782. doi:10.1145/2736282.

[7] Jones, M.: JSON Web Algorithms (JWA). Technical Report 7518. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7518.txt

[8] Jones, M.: JSON Web Key (JWK). Technical Report 7517. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7517.txt

[9] Jones, M.; Bradley, J.; Sakimura, N.: JSON Web Signature (JWS). Technical Report 7515. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7515.txt

[10] Jones, M.; Bradley, J.; Sakimura, N.: JSON Web Token (JWT). Technical Report 7519. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7519.txt

[11] Jones, M.; Hildebrand, J.: JSON Web Encryption (JWE). Technical Report 7516. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7516.txt

[12] Jones, M.; Sakimura, N.: JSON Web Key (JWK) Thumbprint. Technical Report 7638. September 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7638.txt

[13] Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical Report 2898. September 2000.
Retrieved from: http://www.ietf.org/rfc/rfc2898.txt

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

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

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

[17] Microsoft: *BitLocker*. [Online] Accessed 21 April 2018.
Retrieved from: https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview

[18] Miller, M.: Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE). Technical Report 7520. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7520.txt

[19] OpenWrt: *About OpenWrt*. [Online] Accessed 27 April 2018.
Retrieved from: https://openwrt.org/

[20] OpenWrt: *LEDE 17.01.1 - First Service Release - April 2017*. [Online] Accessed 27 April 2018.
Retrieved from: https://openwrt.org/releases/17.01/notes-17.01.1

[21] OpenWrt, W.: *Creating packages*. [Online] Accessed 28 April 2018.
Retrieved from: https://wiki.openwrt.org/doc/devel/packages

[22] OpenWrt, W.: *OpenWrt Feeds*. [Online] Accessed 28 April 2018.
Retrieved from: https://wiki.openwrt.org/doc/devel/feeds

[23] Perry, P.: *Solid IT control hygiene*. *Healthcare Financial Management*. vol. 71, no. 2. 2017: pp. 54–55. ISSN 07350732.

[24] 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

[25] 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.

[26] Team, U. P. R.: Security Analysis of Cryptsetup/LUKS. 2012. [Online] Accessed 22 April 2018.
Retrieved from:
https://www.privacy-cd.org/analysis/cryptsetup_1.4.1-luks-analysis-en.pdf

[27] Thales: *Global Encryption Trends Study*. [Online] Accessed 10 May 2018.
Retrieved from: http://enterprise-encryption.vormetric.com/rs/480-LWA-970/images/2017-Ponemon-Global-Encryption-Trends-Study-April.pdf

[28] vpnpick: *DD-WRT vs. Tomato vs. Open WRT?* [Online] Accessed 26 April 2018.
Retrieved from: https://vpnpick.com/dd-wrt-vs-tomato-vs-open-wrt/

[29] 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

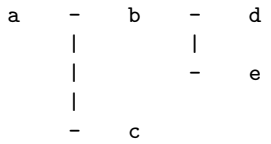
[30] Wiki, O.: *How to build*. [Online] Accessed 26 April 2018.
Retrieved from: https://wiki.openwrt.org/doc/howto/build

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

[32] Wikipedia: *Porting*. [Online] Accessed 1 May 2017.
Retrieved from: https://en.wikipedia.org/wiki/Porting

Appendix A

Compact disk content



Appendix B

Pre-installation enablement of hard drive encryption

B.1 Fedora 28 – disc encryption option selecting

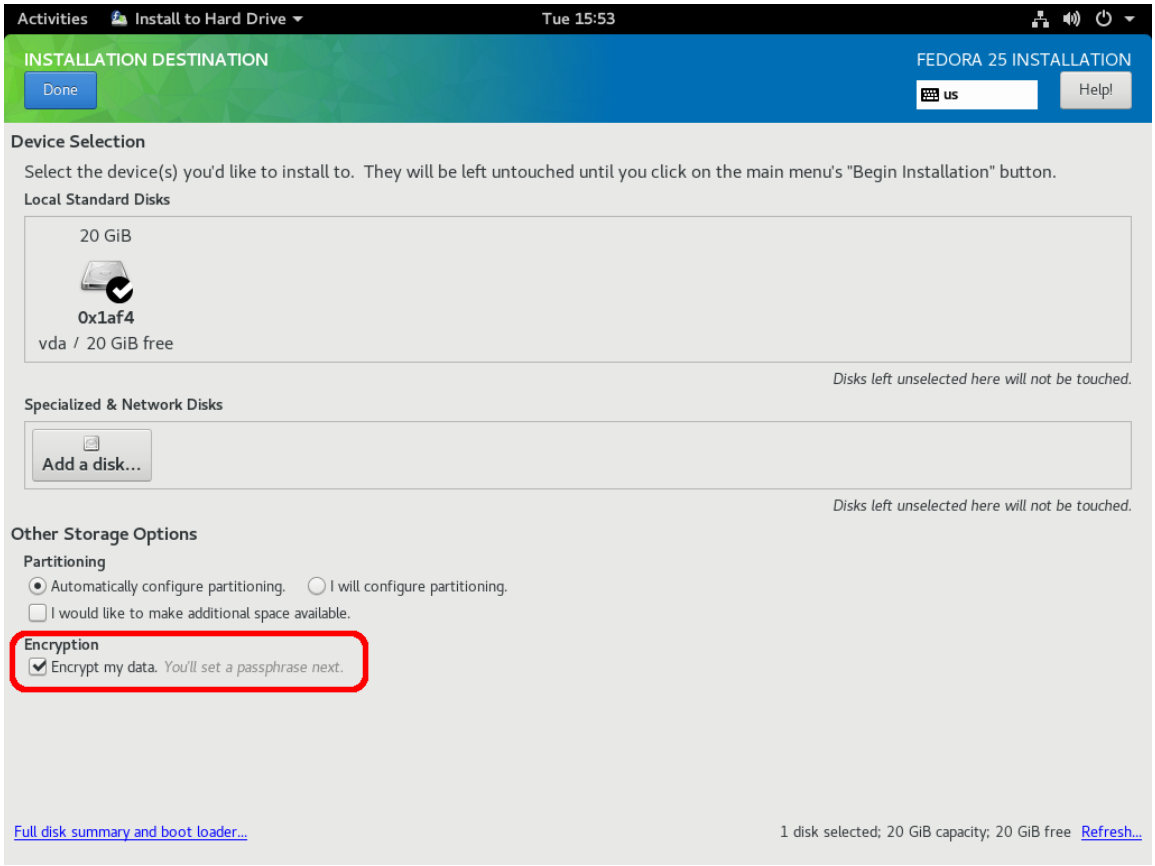


Figure B.1: Checking option

B.2 Fedora 28 – determination key encryption key

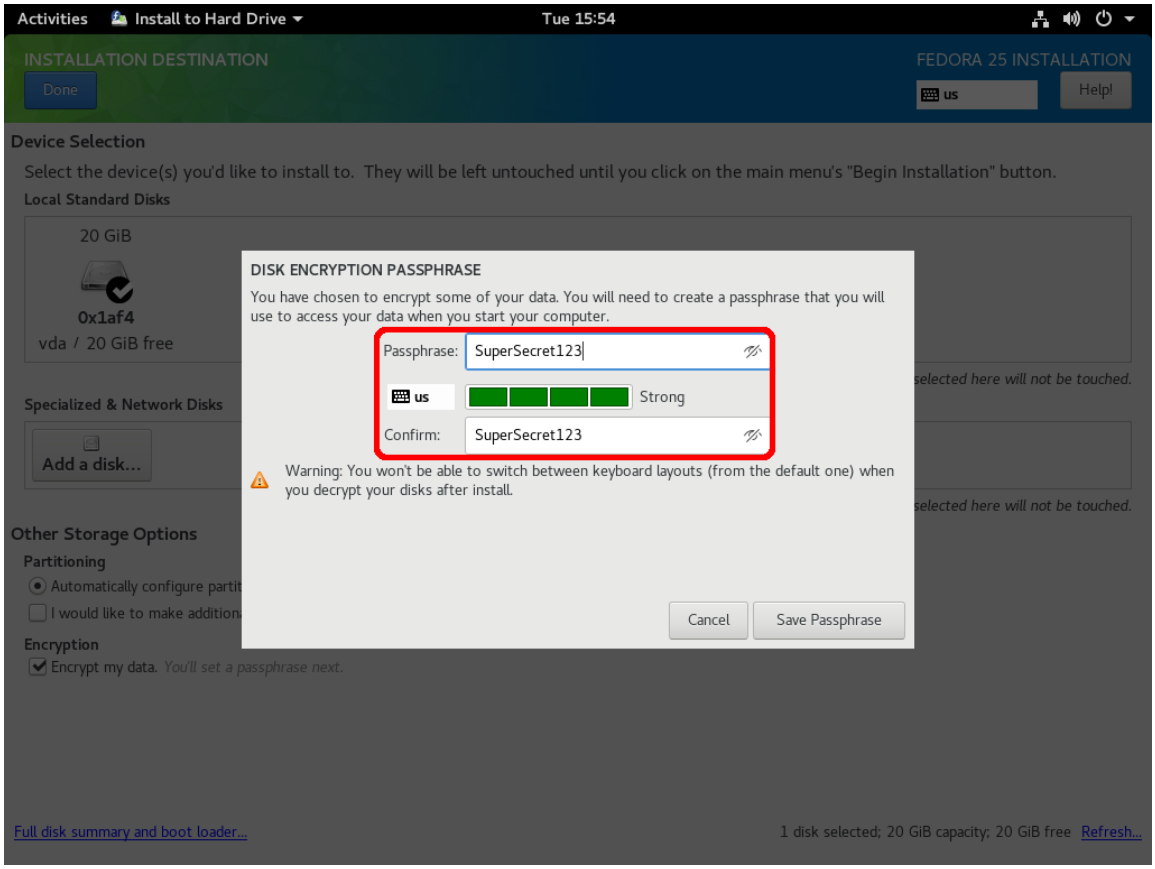


Figure B.2: Determining key

Appendix C

LUKS In-Place Encryption

It takes 4 steps to perform an in place encryption with *luksipc* [1]:

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 key-ring

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 with typing the „p“:

```
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

Setting up the repository

To save our work progress and be able to contribute to the upstream repository we should have a own fork of it. A fork is a copy of a repository that we can manage. It lets us make changes to a project without affecting the original (upstream) repository. We can fetch updates from the upstream or submit changes to the original repository with pull requests. These pull request are generated from the „devel“ branch that we should have in our fork.

To fork OpenWrt's buildroot we should have our GitHub account set up[1], visit the OpenWrt's project upstream repository[2], and click on „Fork“ button in the top-right corner of the page. Before cloning our fork it is recommended to set up git enviroment[3] on host machine and upload the SSH keys[4] to our account to minimize pushing effort to our fork. All work related to this thesis has been done using GitHub account Tiboris[5] therefore output of following commands are bound to it.

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

Please note that this command will clone the *openwrt* repository of the GitHub user *Tiboris* to the *~/buildroot-openwrt* directory on our host.

To catch up with changes made upstream we should consider to configure a remote that points to it. Configured remote allows us to sync changes made in the original repository with the fork and vice versa. Make sure you are in the buildroot directory and add remote[6] called upstream:

```
$ cd ~/buildroot-openwrt
$ git git remote add upstream \
    https://github.com/openwrt/openwrt.git
```

To verify that remote is present run:

```
$ git remote -v
origin  git@github.com:Tiboris/openwrt.git (fetch)
origin  git@github.com:Tiboris/openwrt.git (push)
upstream    https://github.com/openwrt/openwrt.git (fetch)
upstream    https://github.com/openwrt/openwrt.git (push)
```

[1] https://github.com/join
[2] https://github.com/openwrt/openwrt
[3] https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup
[4] https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/
[5] https://github.com/Tiboris/
[6] https://help.github.com/articles/configuring-a-remote-for-a-fork/

With the remote configured correctly we can now sync with upstream repository. To download latest upstream changes use the „fetch“ option and the „merge“ option to apply them to our fork's branch[7].

```
$ git fetch upstream master
remote: Counting objects: 4376, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4376 (delta 1776), reused 1775 (delta 1775),
 pack-reused 2599
Receiving objects: 100% (4376/4376), 1.27 MiB
 | 743.00 KiB/s, done.
Resolving deltas: 100% (2932/2932), completed
 with 809 local objects.
From https://github.com/openwrt/openwrt
 * branch                  master     -> FETCH_HEAD
   36fb0697e2..d089a5d773  master     -> upstream/master
$ git merge upstream/master
```

The detailed description can be found on original GitHub pages[8].

[7] https://help.github.com/articles/syncing-a-fork/
[8] https://help.github.com/articles/working-with-forks/

Appendix E

OpenWrt package's Makefile

This is an example of what OpenWrt Makefile for package could look like. The following makefile is slightly edited makefile from the tutorial[1] on the web.

```
#############################################
# OpenWrt Makefile for helloworld program
#
#
# Most of the variables used here are defined in
# the include directives below. We just need to
# specify a basic description of the package,
# where to build our program, where to find
# the source files, and where to install the
# compiled program on the router.
#
# Be very careful of spacing in this file.
# Indents should be tabs, not spaces, and
# there should be no trailing whitespace in
# lines that are not commented.
#
#############################################

include $(TOPDIR)/rules.mk

# Name and release number of this package
PKG_NAME:=helloworld
PKG_RELEASE:=1

# This specifies the directory where we're going to build the program.
# The root build directory, $(BUILD_DIR), is by default the build_mipsel
# directory in your OpenWrt SDK directory
PKG_BUILD_DIR := $(BUILD_DIR)/$(PKG_NAME)

include $(INCLUDE_DIR)/package.mk

# Specify package information for this program.
# The variables defined here should be self explanatory.
# If you are running Kamikaze, delete the DESCRIPTION
# variable below and uncomment the Kamikaze define
# directive for the description below
define Package/helloworld
	SECTION:=utils
```

[1] https://www.gargoyle-router.com/old-openwrt-coding.html

```
        CATEGORY:=Utilities
        TITLE:=Helloworld -- prints a snarky message
endef

# Uncomment portion below for Kamikaze and later delete DESCRIPTION variable above
define Package/helloworld/description
        If you can't figure out what this program does, you're probably
        brain-dead and need immediate medical attention.
endef

# Specify what needs to be done to prepare for building the package.
# In our case, we need to copy the source files to the build directory.
# This is NOT the default. The default uses the PKG_SOURCE_URL and the
# PKG_SOURCE which is not defined here to download the source from the web.
# In order to just build a simple program that we have just written, it is
# much easier to do it this way.
define Build/Prepare
        mkdir -p $(PKG_BUILD_DIR)
        $(CP) ./src/* $(PKG_BUILD_DIR)/
endef

# We do not need to define Build/Configure or Build/Compile directives
# The defaults are appropriate for compiling a simple program such as this one

# Specify where and how to install the program. Since we only have one file,
# the helloworld executable, install it by copying it to the /bin directory on
# the router. The $(1) variable represents the root directory on the router running
# OpenWrt. The $(INSTALL_DIR) variable contains a command to prepare the install
# directory if it does not already exist. Likewise $(INSTALL_BIN) contains the
# command to copy the binary file from its current location (in our case the build
# directory) to the install directory.
define Package/helloworld/install
        $(INSTALL_DIR) $(1)/bin
        $(INSTALL_BIN) $(PKG_BUILD_DIR)/helloworld $(1)/bin/
endef

# This line executes the necessary commands to compile our program.
# The above define directives specify all the information needed, but this
# line calls BuildPackage which in turn actually uses this information to
# build a package.
$(eval $(call BuildPackage,helloworld))
```

Listing E.1: Makefile for Helloworld.

Appendix F

List of pull-requests

Here is a list of all pull requests realated to the Tang porting effort.

package zlib Created pull-request that was not relevant because of zlib located in main OpenWrt repository see:

`https://github.com/openwrt/packages/pull/4290/files`

package jansson Update of the package jansson was successful and merged upstrem in pull request:

`https://github.com/openwrt/packages/pull/4289/files`

package libhttp-parser This pull request was not relevant and not merged because of http-parser already been located in packages repository under name libhttp-parser:

`https://github.com/openwrt/packages/pull/4304/files`

Closed and not merged due to another pull request merged with bump into same version but without upstream patch containing HTTP_STATUS_MAP macro:

`https://github.com/openwrt/packages/pull/4335/files`

Sucessfully merged bump into libhttp-parser version 2.8.0 containing the upstrem patch with HTTP_STATUS_MAP macro:

`https://github.com/openwrt/packages/pull/5446/files`

package jose Still open (not merged yet) and waiting for review (12th May 2018):

`https://github.com/openwrt/packages/pull/4334/files`

package tang Still open (not merged yet) and waiting for review (12th May 2018):

`https://github.com/openwrt/packages/pull/5447/files`