# AURsec

A blockchain aproach to securing software packages

Bennett Piater & Lukas Krismer

Bachelor thesis

Supervisor: Christian Sillaber

University of Innsbruck

## STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

| | | |
|---|---|---|
| Innsbruck 17.05.2017 | ________________________ | ________________________ |
| Location,Date | Bennett Piater | Lukas Krismer |

# Contents



# List of Figures



# List of Tables

# 1 Introduction

The Linux distribution Arch makes it easy to create custom packages and has a very active community. Because of this, there was the need for a place where users could upload their packages for others to use.

ftp://ftp.archlinux.org/income was created, where packages could be put until maintainers that were willing to do so could adopt them, but this delay was too long, so another solution was needed. The next improvement were the Trusted User Repositories, where some privileged users, which were many more than the maintainers of before, were allowed to host their own repositories for anyone to use.

The *Arch User Repository* were the natural evolution: By removing all middlemen, everyone can now upload their packages to one central place. [1]

The AUR is similar to PyPI (Python), NPM (Javascript) and rubygems.org, where all users can share their packages. They all share the problem that submitted packages are not necessarily audited or even checked by anyone.

# 2 Security Issues of the AUR

Indeed, ease of use appears to have been, if not the only, at least the primary design consideration of the Arch User Repository. This creates so many security issues that it is actually quite a task to think through all of them.

### Local Package Creation

One of the most obvious problems is the installation procedure itself. The AUR doesn't host binary packages, which is a good thing. Instead, Arch packages are created locally from a bash file, the so-called `PKGBUILD`, containing metadata like name and version, the URLs and checksums of upstream sources, and functions for the compilation and packaging steps.

The AUR contains these `PKGBUILDS` and possible patches to be applied to the upstream sources in a git repository per package. A package file can be produced by cloning it's repository and using a tool called **`makepkg`** [2], which sources the script, downloads and verifies the sources, and calls the compilation and packaging functions.

This means that users can verify what they are compiling as opposed to blindly trusting binaries created by third-parties, but also that maintainers of AUR packages have a means of executing arbitrary shell commands on users' machines.

This is aggravated by the fact that `PKGBUILDS` can include a `.install` file into the built package, which will be executed *as root* when the package is actually installed. The risk also increases if so-called "AUR helpers" are used. These tools assist the users in installing packages from the AUR by automating the steps and behave like package managers. Some of them (notably **`aurutils`** [7], which is recommended by the authors) allow the users to inspect these files before continuing, but others are very unsafe in that they execute code before giving users the opportunity to inspect it, or decentivize them from doing so.

### The Trust Issue

Another problem is that users are not given any reason to trust the maintainers. Unlike the official repositories, where maintainers are vetted, packages are (often manually) audited before being accepted, and everything must be signed with a trusted OpenPGP key, anyone can create an account and submit a new package to the AUR in a few minutes. There is no

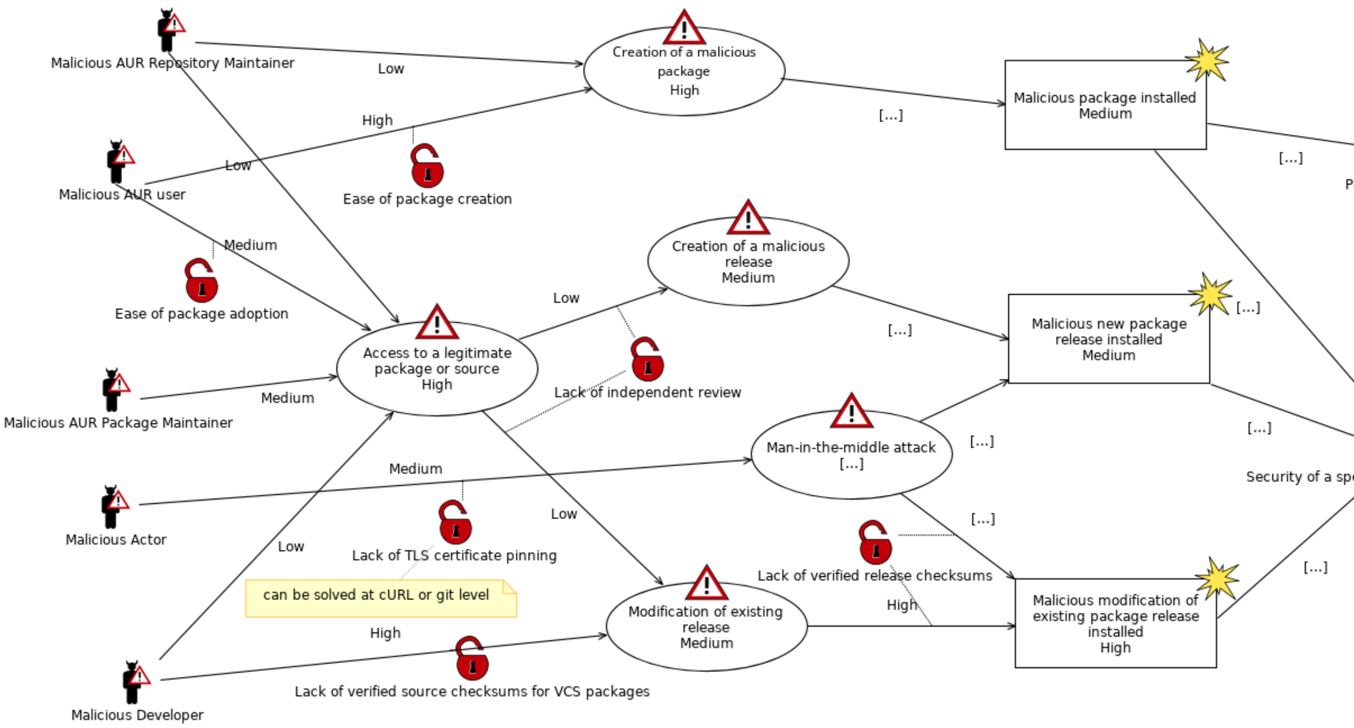


Figure 1: Threat Assessment of the Arch User Repository

admission procedure or audit system and no OpenPGP web of trust in order to minimize the time needed to publish a package or update.

`makepkg` can verify OpenPGP signatures for upstream sources, but the `PKGBUILD` itself could only be signed by using signed git commits, which is sadly not enforced or even officially recommended — and not supported by any AUR helper anyway.

Except when using the AUR helper `bauerbill` [9], which provides a basic user-side trust management system, the only way to be maintain reasonable trust is therefore to manually read every single file, which is cumbersome. Because only highly security-conscious users are willing to put in so much effort before trusting a `PKGBUILD`, most users are left vulnerable by the aforementioned issues.

### Adopting orphan packages

The trust issue is made even worse by the fact that packages can silently and quickly be taken over by other maintainers due to the orphan/adoption system.

When a maintainer wants to stop maintain a package, but the package is still useful and actively developed upstream, he has the option to *orphan* it rather than deleting it. Orphan packages can be *adopted* by any AUR user, at any time, without delay. This feature was designed to minimize update delay, which it does effectively; However, it also makes it easy for malicious agents to take over popular orphaned packages, manipulate them, and immediately orphan them afterwards.

### Concrete Attack Scenarios

We used the CORAS [4] threat modeling language to arrange the security issues in such a way that concrete attack scenarios are intuitive to comprehend and retrace. The resulting threat diagram can be seen in Figure 1.

Many of the AUR's security issues exist by design and are only included for completeness. However, several issues lead to the same situations; This means that security issues further along the right of the diagram tend to be more promising candidates in the search for solvable problems.

This knowledge leads to two concrete attack scenarios that could be preempted without redesigning the AUR, which are outlined below.

### Tampered VCS Sources: Malicious Upstream

In some cases, the user is not adequately protected against malicious (or compromised) upstreams:

The AUR supports so-called *VCS packages* [3], which download sources from a version control system, such as Git or Mercurial, instead of downloading a fixed archive. This relieves the maintainer from updating his `PKGBUILD` for every new commit. `makepkg` will even automatically calculate the up-to-date version number using e.g. tags and commit numbers.

VCS packages were primarily designed to simplify the installation of up-to-date packages from source, and they do that very well; But they also introduce a big security issue: Since the `PKGBUILD` must not be updated between versions, it cannot contain checksums for the new version, either. This means that users don't have a way to verify the authenticity of the source that they are downloading, unless they can trust the upstream himself, meaning that no-one will notice if the upstream is compromised or makes malicious changes. There is no way to defend against this except to manually audit the upstream sources, which should primarily be the maintainers responsibility.

### Tampered Packages: Malicious Maintainer

Users are also not adequately protected against malicious maintainers:

Because it's so easy to gain access to a package, e.g. by adopting an orphan or simply by creating it, and nothing is verified or audited before publication, It's easy for a malicious agent to modify a package. And because the `PKGBUILD` is not signed or hashed, users will not notice if the build instructions for a specific package version were modified. This allows targeted attacks:

If the time at which a target will update his machine is known and one has access to an AUR package which he is expected to update, malicious code can somehow be introduced into the corresponding `PKGBUILD` within that update window. This could be as simple as changing the checksum if one also has access to the upstream source code (even a very careful user has no chance of noticing this attack), or executing innocuous code in the install file or `PKGBUILD` itself.

The malicious change could be reverted immediately afterwards. If the time frame is short, no other AUR user (and thus, *no-one*) would ever notice. One could only defend against this with a good local trust model, such as that possible with `bauerbill`, or manual cryptographic verification of the git commit to the AUR — assuming that the maintainer signs his commits, which is only very rarely the case.

## 3 The Solution

Defence against the two attacks mentioned above requires the availability of cryptographically secure release hashes for every version of every package; If those were available, an

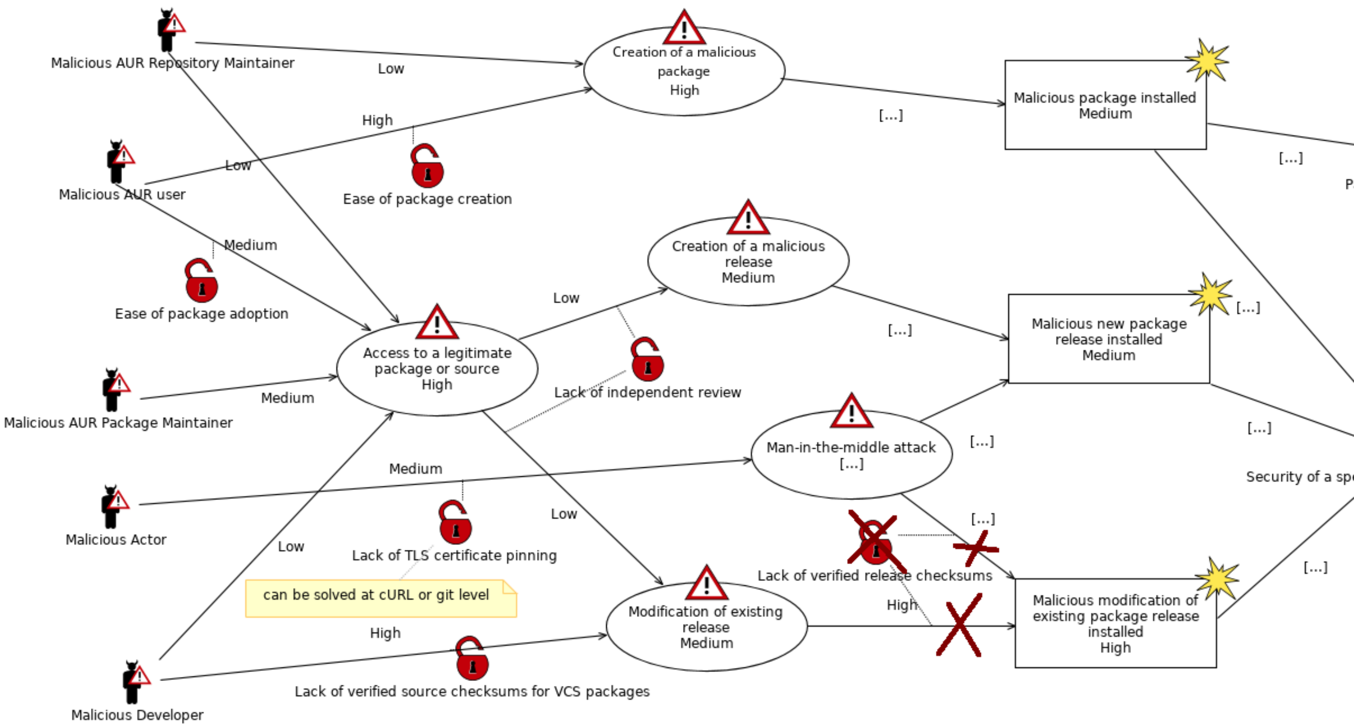


Figure 2: Strategy for Improving the Security of the AUR

attack would result in a hash mismatch and therefore warn the user. However, the AURs design prevents any secure implementation on the server side.

## 3.1 Core Solution

The solution must therefore be to implement a (preferably distributed) database on the user side, which means that there is no single authoritative source. Since the aim is to defend against *targeted* attacks, the assumption being that only few users will encounter a malicious version, this can be worked around by checking the hash against a *consensus* formed by many users.

To make the database as safe as possible, a blockchain is used. The chain contains a smart contract providing securely callable functions. With one of these functions it's possible to commit a hash for a package and version. This hash will be saved in the blockchain only if this user has not committed the same hash before, thereby making it harder to take over the blockchain and get a malicious hash to be the consensus. The consensus is updated after every hash commit. Another function is used to get the current consensus hash and its number of commits for a specific package and version.

This appears to be the first project to use a blockchain as a means to provide distributed verification of (software) downloads.

### Workflow

The following workflow is visualized by Figure 3.

1. First of all a *PKGBUILD* is downloaded and partially executed in a sandbox in order to get the package version. Then, it and any VCS sources are hashed.

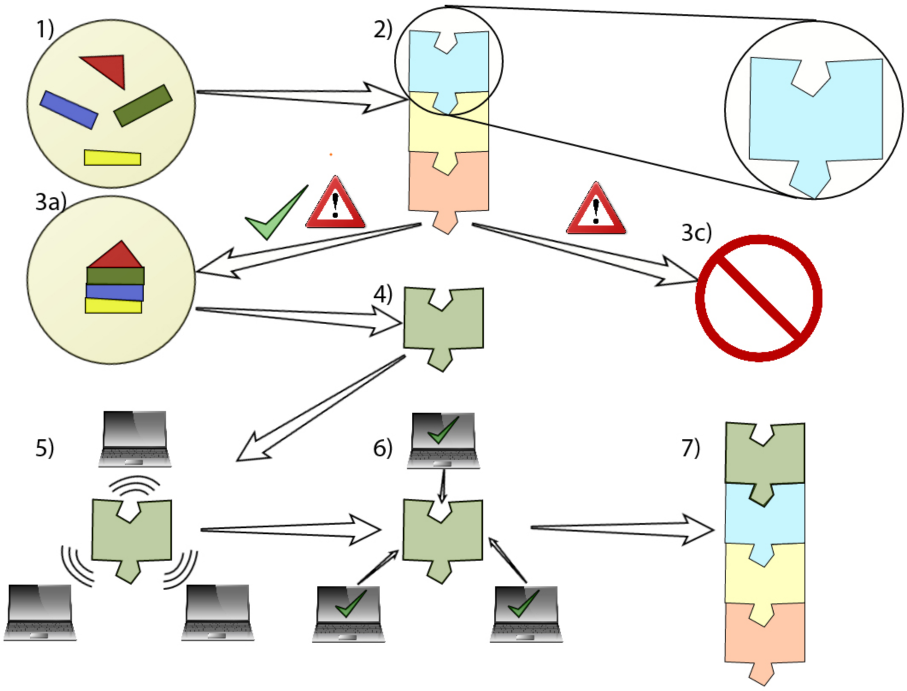


Figure 3: Main Workflow

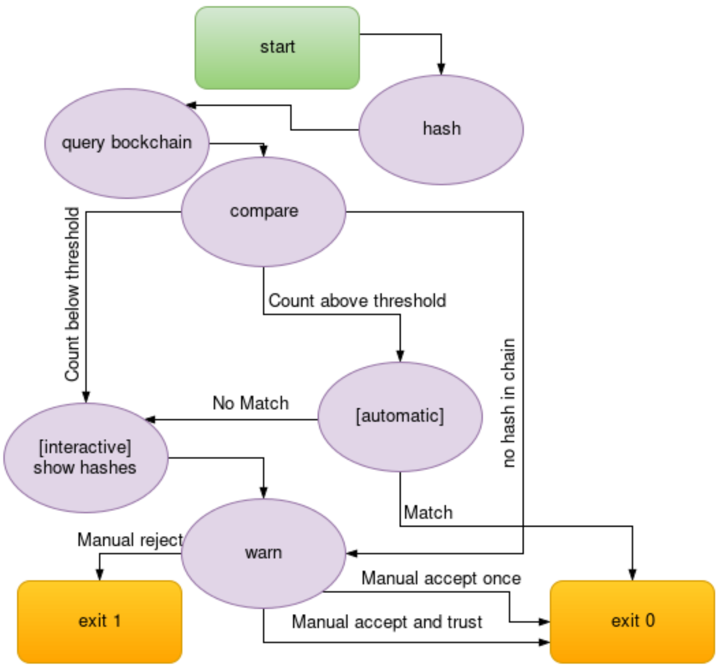


Figure 4: Decision Workflow

2. The resulting local hash is compared with the current consensus hash on the blockchain. [Figure 4].

3. Now the workflow splits into 3 different paths:

    a) The hashes match and the number of commits is over the threshold or the user decides to trust the locally generated hash anyway. *(Followed by step 4.)* The package may be created and installed.

    b) The hashes don't match and/or the number is below the threshold, but the user want to create and install the package without comitting the hash. *(End of the workflow.)* The package may be created and installed.

    c) The hashes don't match and/or the number is below the threshold and the user doesn't want to create the package. *(End of the workflow.)* Our program exits with a non-zero status, so an AUR helper using it would cancel the package installation at this point.

4. The local hash is committed to the blockchain (this is a transaction).

5. All nodes of the blockchain-network get the transaction.

6. The transaction is contained in the next mined block.

7. The block is added to the blockchain.

## 3.2 Detailed Description

### 3.2.1 Blockchain

#### general

The following paragraphs are based on *Blockchain Beyond Bitcoin* [6].

A blockchain is a distributed database system which is not owned by a single user. Every user can see all transactions. If a user wants to add a transaction to the blockchain, the transaction is encrypted and sent to all users. The transaction is then verified. If the majority of the users validate the transaction, the data is added to the blockchain in a block. "Transactions are secure, trusted, auditable, and immutable". Blockchains don't need backups because every user has his own copy which is synchronized with all other blockchains in the network.

**Smart Contracts** are using computerized transaction protocols to execute the terms of the contract which are agreed by the users. They are executed by every miner. Since their code is in the chain, it is guaranteed to be immutable and therefore impossible to manipulate. This means that one is effectively able to run code on the blockchain.

#### specific

Our blockchain has to fulfill several requirements:

- Since we target Arch Linux, it should be easy to install on this platform.
- It needs to provide an API which allows external scripts to work with it.
- It needs to provide smart contracts to maintain the constraints of the hash storage.

- It needs to provide an easy way to create private networks seperate from the main one, because full networks are huge.

The blockchain of choice was Ethereum because it is the only production-ready infrastructure providing smart contracts. It even allows them to be written in a high-level language, *Solidity*, which makes them straightforward to write and understand.

**smart contracts** Aursec has two smart contracts. The first one is a formal contract, which allows the owner of the contract to delete the contract. The second contract is a child of the first. In this contract are the "user-functions", which allows all users to send hash-commits and request the current consensus hash of a versioned package. The Contract allows a user to commit one hash per versioned package. Further commits of hashes of the same versioned package will not be considered. The current consensus of a versioned package is the most often committed hash so the hash of a package is not fixed by the first commit.

**network** Like other blockchains, the Ethereum network is peer-2-peer, but private networks need to provide their own *bootnode* through which peers announce themselves. Our bootnode is active 24/7 on a DigitalOcean droplet provided by our supervisor.

**interfaces** The Ethereum blockchain provides 2 different interfaces, the IPC (interprocess communication), which provides an interactive javascript shell, and the HTTP RPC (remote procedure call) interface. The IPC interface of our blockchain is deactivated because users don't need to manually use the javascript shell: the only required interaction with the blockchain is through the shell script `aursec-chain` (Section 3.2.6) and the Python script `aursec-tui` (Section 3.3). These scripts communicate with the blockchain trough the RPC interface.

### 3.2.2 aursec-init

Aursec-init is a shellscript which allows the user to create all requirements just by running it. It also allows to overwrite an existing aursec-blockchain with a new one.

**workflow of the initialization:**

1. Create needed folders and markers (markers are needed by TODO)
2. Create the blockchain from our genesis block.
3. Create a new user with a random password which is saved in a file.
4. Create the *Directed Acyclic Graph* for the blockchain, which is a 1GB dataset. The DAG is needed for mining new blocks. [8]
5. Set safe permissions for the above files and folders.
6. Mine a few blocks to trigger the synchronization and to have enough ether to be able to commit a few hashes from the start.

### 3.2.3 aursec — Architecture

`aursec` is the primary tool of our thesis. It implements the workflow of Section 3.1. Users can execute it passing a build folder (containing a `PKGBUILD`) as argument, and it will figure out the package name and version, hash the build files and VCS sources, compare them against the consensus, and present the result to the user.

We designed `aursec` to be a good UNIX citizen:

- Modular design
  Multiple small tools doing one thing and doing it well
- Adherence to the universal interface [5]
  Working on streams of text on `stdin` and `stdout`
- Written in Bash and using existing tools where possible
- Maximizing concurrency using a pipeline and blocking I/O

`aursec` builds a pipeline of two other tools, `aursec-hash` and `aursec-verify-hashes`, which produces lines containing the id and hash of a package as well as the hash representing the current consensus on the blockchain and the number of times that hash was submitted. It then inserts itself into that pipeline and iterates over the lines of items using a *while-read* loop and traverses the state machine (Section 3.1 and Figure 4) for each item.

This architecture has several advantages: It is straightforward to understand because it follows standard UNIX patterns, which also makes it very maintainable. The free 3-level parallelism gained by the pipeline is a very useful advantage in itself, but even more so because all 3 tools are primarily I/O-bound: `aursec-hash` reads and hashes lots of files, `aursec-verify-hashes` constantly queries (and waits for) the blockchain, and `aursec` tends to spend much time waiting for user input. Thus, the concurrency is even more important because it allows work to continue in the background while `aursec` waits for the user. Indeed, the background tasks tend to be finished in most practical situations before the user has had time to inspect the second or third warning.

### 3.2.4 aursec-hash

`aursec-hash` has the straightforward task of producing an ID (`$pkgname-$pkgver-$pkgrel`) and a hash from `PKGBUILD`s.

The id could be parsed from the `.SRCINFO`, which is a plain text file. But VCS packages don't have up-to-date version information in their `PKGBUILD`, which means that it must be interpreted my `makepkg` to update it; This is annoying, but we only source the `PKGBUILD` in a `firejail` sandbox to minimize the inherent risk of executing foreign turing-complete code. This allows us to get an accurate ID for VCS packages, but also to include the actual sources in the hash, thereby compensating for the lack of hashes in the `PKGBUILD` of VCS packages.

The `PKGBUILD` is hashed after stripping it's comments, and VCS sources, if they exist, are hashed using a `find` command. Finally, all hashes are combined by another call to the hash command. Currently, `sha256sum` is used for it's good speed and security.

### 3.2.5 aursec-verify-hashes

This tool fetches the current consensus for every package ID on `stdin`, computes whether it matches with the locally computed hash, and appends that data to the output stream.

Doing this in a separate pipeline step is very worth it because requests from the blockchain are comparatively slow, making the concurrency highly useful.

### 3.2.6 aursec-chain

*aursec-chain* is a shellscript which allows the user to intergate with the blockchain. The script itself communicates with the blockchain through JSON RPC. To provide the user all needed commands, the script has different arguments.

**mine** This argument needs more arguments to work.

- `start:` Starts mining.
- `stop:` Stops mining.
- `N blocks:` Wait until mining is stopped and then mines N blocks.
- `auto:` This command is only used by a systemd timer to mine blocks periodically.

**commit-hash** This argument needs two more arguments to work. The first one has to be the versioned package id and the second has to be the locally generated hash of the package. The script then sends a transaction to the blockchain (if enough ether is available). This transaction will be verified by the next mined block.

**get-hash** This argument needs one more argument to work. This argument has to be the versioned package id. Then aursec-chain calls a method which returns the current consensus hash and the number of commits of this hash.

### 3.2.7 Systemd Services

Since Arch Linux uses Systemd as init system and service manager, it was natural for this project to use it as well.

We use it for two things:

**aursec-blockchain.service** This service simply starts the blockchain process with the correct arguments. Using Systemd gives us an easy way to start the blockchain on boot with the correct configuration and a CPU quota to limit the impact on other running programs.

**aursec-blockchain-mine.timer** This timer is used to periodically mine blocks on the chain, thereby making the next wave of hashes available to other users. A Systemd timer works similarly to a Cron job, but with more control, and is easier to provide in a package.

## 3.3 Terminal User Interface

`aursec-tui` is a Python script. The TUI gives a overview over all mined blocks, their hashes, the miner of the block and eventual transactions which are saved in the block.

It is split in two parts. The first part is needed to gain the data from the blockchain and save it, the other part formats the information and displays it. To display the results as fast as possible a thread searches the blockchain in the background. Any additional data will be displayed after the next refresh. The script offers the user two settings for filtering the results.

- `only mine:` only display blocks which where mined by the user himself.
- `only transactions:` only display blocks with transactions.

The settings can be combined with the result that all blocks mined by the user with transactions will be displayed.

## 3.4 Project Management

This project has involved three people. Two students who implemented the tool and their superior. So there had to be a lot of communication. The communication tools were offline meetings, Slack and emails. For filesharing Github is used.

### timeline

From the timeline (Figure 5) can be derived that this project was realized in less then 9 months. The real worktime exposure ratio of the three main-task (reading, programming and writing) differs extremely from the visualized one. In reality we read 50%, programmed 40% and wrote 10% of the worktime. It is also visible, that we didn't manage to complete three milestones in time. This was caused by different circumstances, but we managed to complete the hole project in time cause we adjusted our planing every few weeks.

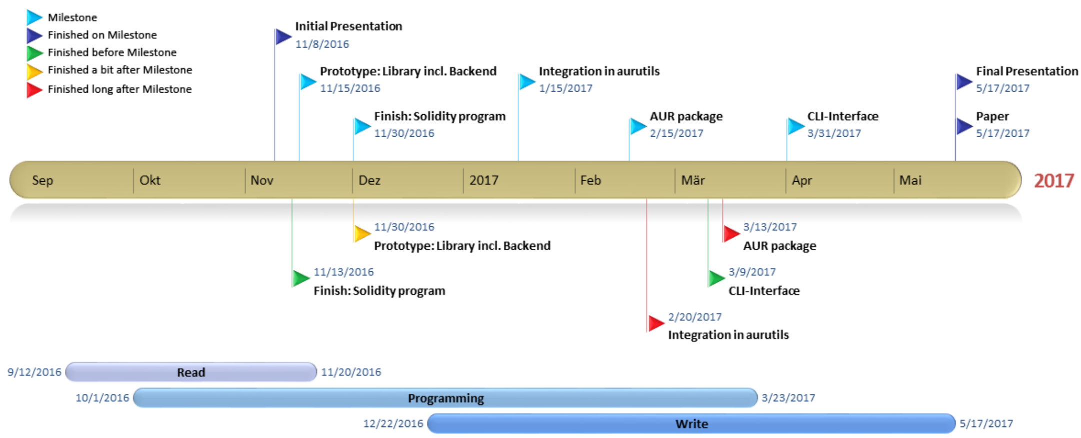


Figure 5: Timeline

# 4 Things we Learned

### Ethereum — Solidity

Programming on a blockchain is a very interesting concept, but it also takes some getting used to. Thankfully, Solidity is a cleanly designed language which abstracts the blockchain away very nicely.

In practice, writing for Ethereum turned out to be enjoyable. Solidity reads like a half-way mix between C and Javascript, and most things that we rely on in this project happen automatically: Guaranteeing that the code cannot be modified, the ACID properties of transactions, etc. This means that Ethereum, and Solidity in particular, are ideally suited for secure, "intelligent" (self-enforcing) databases.

Because of that, our code reads largely like a pseudo-code description of the algorithm itself, making it easier to maintain and verify. Solidity even supports some automatic formal verification, but not yet for `structs`, so we cannot make use of it for our program.

### JSON RPC

The JSON PRC is used by the aursec-ethereum-blockchain for remote access to the block. Methods can be called or transactions can be sent by sending a JSON-Object to the block chain. The answer from the block chain is also always a JSON-Object. One big advantage is the easy parsing from and to JSON-Objects both in Bash and Python. The best example is visible in the code of aursec-tui (Section 3.3).

#### Bash

Using Bash as a programming language is interesting. The syntax can be weird, even arcane; At the same time, we were often surprised by the advanced features that are provided directly inside the language, such as string substitutions, regular expression matching or associative arrays. In addition, the `coreutils` are very powerful; To our knowledge, only the Python standard library offers comparable functionality.

Apart from getting used to the uncommon syntax, we found that the most important prerequisite for writing larger programs in Bash was to think in streams: Bash is not strictly imperative or functional, and it's certainly not object-oriented. Functions and programs can only return non-integer values as text on `stdout`, and it is often useful to provide them their input on `stdin` as well. We quickly found out that the most efficient way to structure programs is to use `while-read` loops, which iterate over an input stream of text.

Embracing this design philosophy results in the natural use of highly concurrent pipelines that turn out to be very easy to understand, maintain and extend, far more so than equivalent imperative or object-oriented versions. Readers familiar with Java can compare this to Java 8's `java.util.stream` API.

Writing safe and correct code is as hard as in C, mostly due to the lack of exception handling or a sensible alternative. We work around that using `set -e`, which cancels a program whenever an unused return value is non-zero, and exit handlers for cleanup.

Bash doesn't provide or recommend a canonical testing framework, but associative arrays and `eval` allowed us to write our own system for basic unit tests with named test cases, commands to execute, and expected invariants. We used it to great effect in narrowing down the best `firejail` sandbox ruleset, e.g. preventing `makepkg` from writing to folders other than `pwd`.

#### Python-Urwid

Urwid is a *Terminal User Interface* (TUI) library for Python. It provides ready-made widgets which make it easy to create structured user interfaces.

## 5 Evaluation

# References


[1] ArchWiki. Arch User Repository — Arch Wiki. https://wiki.archlinux.de/title/Arch_User_Repository, 2017. accessed March 17, 2017.

[2] ArchWiki. Creating packages — Arch Wiki. https://wiki.archlinux.org/index.php/Creating_packages, 2017. accessed March 18, 2017.

[3] ArchWiki. VCS Package Guidelines — Arch Wiki. https://wiki.archlinux.org/index.php/VCS_package_guidelines, 2017. accessed March 18, 2017.

[4] Hogganvik Ida Stølen Ketil Dahl, Heidi E.I. Structured semantics for the coras security risk modelling language, 2007.

[5] Peter H. Salus. *A Quarter Century of UNIX*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[6] Sarah Underwood. Blockchain beyond bitcoin. *Commun. ACM*, 59(11):15–17, October 2016.

[7] Alad Wenter. aurutils: helper tools for the AUR. https://github.com/AladW/aurutils, 2016-2017. accessed March 18, 2017.

[8] Ethereum Wiki. Ethash-DAG — Ethereum Wiki. https://github.com/ethereum/wiki/wiki/Ethash-DAG, 2017. accessed March 20, 2017.

[9] xyne. Bauerbill: Extension of Powerpill with AUR and ABS support. http://www.xyne.archlinux.ca/projects/bauerbill/, 2015-2017. accessed March 18, 2017.