AURsec

A blockchain aproach to securing software packages

Bennett Piater & Lukas Krismer

Bachelor thesis
Supervisor: Christian Sillaber

University of Innsbruck

Contents

List of Figures

List of Tables

1 Introduction

The linux distribution Arch has one of the most active communities. This is the reason why there was the need for a place, where users could upload their own packages in a repository. ftp://ftp.archlinux.org/income was born. But the packages where not available for other users till a Package Maintainer atopted them. The progression were the Trusted User Repositories, where some users where allowed to host their own repositories for anyone to use. On this base the AUR (Arch User Repository) was evolved. The AUR is similar to PyPI (Python), npm (Javascript) and ruby gems (Ruby), where all users can share their tools. The problem by all of them is, that the packages are just checked by the community and by no higher instance. [1]

2 Security Issues of the AUR

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 comands 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` [4], 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 GNU Privacy Guard key, anyone can create an account and submit a new package to the AUR in a few minutes. There is no admission procedure or audit system and no GnuPG web of trust in order to minimize the time needed to publish a package or update.

`makepkg` can verify GnuPG 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` [6], 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

VCS Packages: Malicious Upstream

[3]

Tampered Packages: Malicious Maintainer

3 The Solution

3.1 Core Solution

The solution for a few security-problems is a secure database, which contains hashes of versionized packages. Before a package is installed, the locale generated hash can be compared with the one in the database. This guaranty that the loaded package is the same as the package loaded by most of the other users.

To make the database as safe as possible, a blockchain is used. On this blockchain is a smart contract, which allows to call functions. With one of these functions it is possible to commit hashes of versionized packages. This hash will be saved in the blockchain if this user has not committed a other hash for the same versionized package before. Another function is used to get the current consensus hash and its number of commits of a versionized package. This is the first application which uses a blockchain to secure downloads.

Workflow

1. First of all a *PKGBUILD* is downloaded and partially executed in a virtual area. Then this data get hashed.

2. The resulting local hash becomes compared with the current consensus hash of the versionized package of the blockchain [Figure 2].

3. Now the workflow splits into 3 ways.
    - a) The hashes match and the number of commits is over the threshold or the user decides to trust the local generated test. The package is created and installed. *(Followed by step 4.)*
    - b) The hashes don't match and/or the number is below the threshold but the user want to create and install the package. *(End of the workflow.)*
    - 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.)*

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

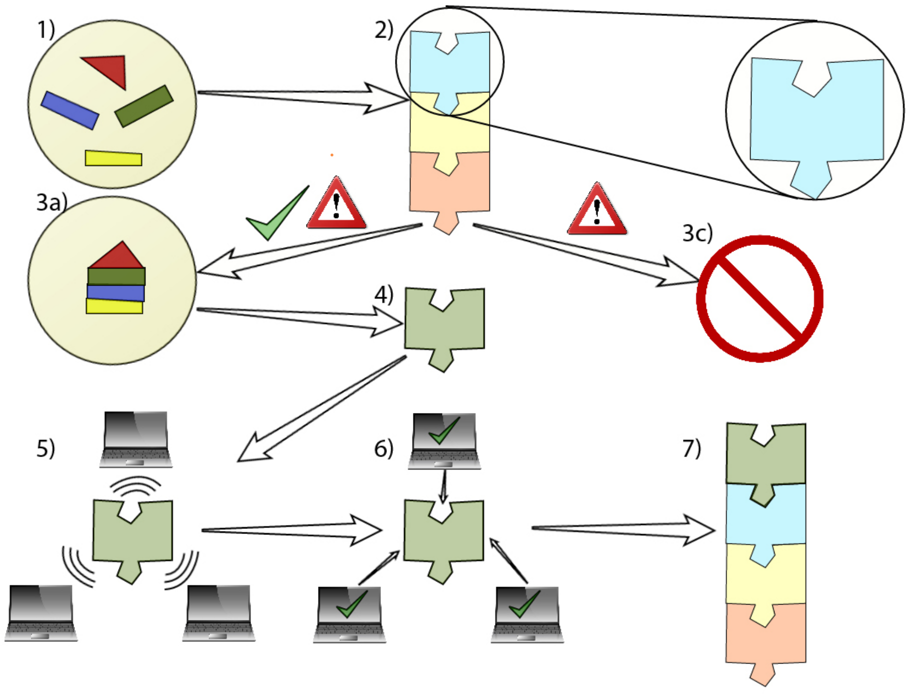

Figure 1: Main Workflow

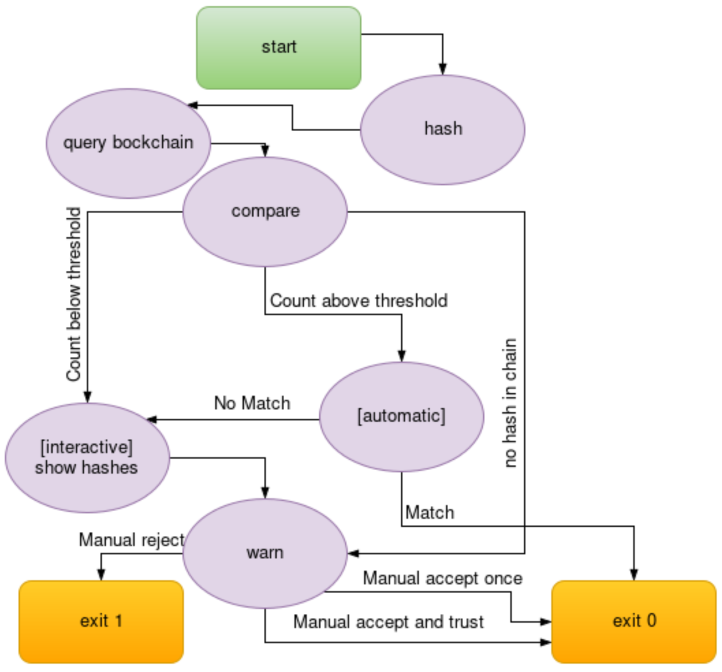

Figure 2: Decision Workflow

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

The blockchain has to fulfill different requirements.

- The blockchain mainly has to be installed on arch-linux computers. So the installation should be easily on this platform.
- The blockchain has to have a interface, which allows to use the blockchain in background cause the blockchain should be started with a systemd service.
- The blockchain has to support smart contracts for the functional requirements (mainly get/commit hashes).
- The blockchain should support private networks, cause the aursec-network should only contain transactions related to aursec.
- This private network should be easily build.

The blockchain of choice was a ethereum-blockchain. The main reason for this blockchain, are the solidity smart contracts. These contracts are easy 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 versionized package.

network Currently the nodes connect to the network over a bootnode. This bootnode runs on a 24/7 server.

interfaces The Ethereum-blockchain uses 2 different interfaces. The IPC (interprocess communication) and the RPC (remote procedure call) interface. The IPC interface of the aursec-blockchain is deactivated by default, cause the user shouldn't have the need of a console which is attached to the running node. The user interact with the blockchain through shellscripts especially aursec, aursec-aursync-wrapper, aursec-chain and the pythonscript aursec-tui [see 3.2.3]. 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. Creating needed folders and markers (markers are needed by TODO) and set the right rights.
2. Creating the blockchain based on a genesis-block.
3. Creating a new user with a random password which is saved in a file.
4. Creating the DAG, which is a 1GB dataset. The DAG is needed for mining the blocks. [5]
5. Mine a few blocks to have enough ether to be able to commit a few hashes at the start.

3.2.3 aursec-chain

3.3 Terminal User Interface

3.4 Project Management

4 Things we Learned

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

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

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