View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Publikationsserver der Universität Tübingen

Transparency for Control Plane Software

Benjamin Hof Georg Carle

Technische Universität München
{hof, carle}@in.tum.de

***Abstract*—A shift from traditional networking to SDN requires considering the security properties not only of the policies, but also of the software implementing these functions. We propose a transparency-based approach to secure the distribution of the code implementing the software components in SDN-based networks. The goal is to ascertain freedom from *targeted* backdoors for all installed code. To achieve this, the software packages are logged into an append-only log server by the distributor. Monitors verify the correct operation of the log server and are able to detect and warn about irregularities in the software distribution.**

I. Introduction

Security is an important driver for the deployment of SDN. The security of an SDN network depends on the policies implemented by the controller or network operating system. To have confidence in the rule sets emitted by the controller, the operator must assure the integrity of the controller software. As a precondition for running a secure SDN network, the operator must therefore be certain that the software they intend to run is the software that is actually executed on the SDN controller.

To install an SDN controller, the operator would typically start by installing a Linux distribution such as Debian on a server. Using the integrated package manager *APT*, an SDN controller software package can then be installed, customized, and configured. During the life time of the controller, security updates are provided through the distribution. Package downloads for installation and updates are traditionally secured with cryptographic signatures [1].

This mechanism critically depends on the secret key used for signing and is in particular vulnerable to targeted backdoors introduced by the publisher. A backdoor is *targeted*, when it is presented only to a selected victim. To combat this, we propose a transparency-based approach that achieves detection of targeted backdoors, verification of relationship between source code and binary, and developer accountability in case of violations.

II. Background and Related Work

Package-based software distribution is regularly based on *releases*. A release fixes by cryptographic hash the package contents and meta data e.g., dependencies [2]. If any of these change, a new *release file* must be created. In the case of Debian, this release file is signed by the archive server, the authoritative central distribution point. The archive then distributes the packages and meta data to a content distribution network, making it available for installation by clients.

Augmenting systems traditionally based on cryptographic signatures, Merkle tree-based append-only log servers have been proposed to improve security in several applications. They allow to efficiently prove inclusion of elements and historic consistency.

In Certificate Transparency, all certificates are submitted into a public Merkle tree-based log [3]. User agents verify this inclusion. Site operators can now notice unexpected certificates by monitoring the logs, and thereby identify certification authorities misissuing certificates.

In CHANIAC, witness servers act as log for software updates and signing keys [4]. These witness servers cooperate, but must be independent. Approval by multiple developers is required for a release.

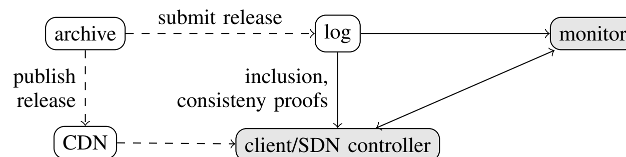

Figure 1. Transparency log for software releases.

III. Design

We propose a design for a transparency system for APT-based distributions such as Debian. In Debian, developers upload signed source code, accompanied by meta data such as build instructions, for compilation by the archive. The archive creates new releases per policy. These are published to a CDN, where clients such as SDN controllers retrieve them. This operation is shown in Figure 1, where the transmission of software package is noted in dashed lines.

We now introduce the transparency system consisting of a public, append-only Merkle tree log, a monitor, and a modified client. Trusted components are marked gray in Figure 1. The log is untrusted and kept trustworthy through the actions of potentially many monitors. The archive submits release information into the log. The monitors verify the correct operation of the log and communicate with the client if there are problems with the log's record keeping. For a new release, clients check if it was published in the log.

The log, monitor and client constitute a pledged transparency overlay as defined and proven by Chase and Meiklejohn [5]. We will describe their respective functions briefly, and present the additional verification functions required to adapt the overlay for our use case.

A. Log operation

The proofs provided by the log follow standard Merkle tree operations as described in other hash tree-based systems [3], [5]. Should any of these proofs not succeed, the client has cryptographic evidence of misbehaviour by the log. This evidence can be published out of band and constitutes the starting point of an investigation into the compromise of the distribution infrastructure.

The archive submits the release file, meta data files, and source code to the log server. The log server must respond with signed statements promising inclusion of these items into the Merkle tree, similar to a receipt.

The inclusion promises are distributed via the CDN together with the packages. Using the properties of the log's Merkle tree, clients can now verify that a given release file had been logged by querying the log server for cryptographic proof. This proof contains in particular the signed root of the hash tree.

The client can also verify that the state of the log server is consistent with the state at the time of last contact, i.e. it has operated append-only. To do this, the log is queried for a proof of consistency between the roots of the hash tree at the time of the previous update and the tree root observed in the previous step.

B. Monitor functions

To ensure the honest operation of the archive and the log, monitors continually retrieve tree roots and the elements covered by the hash tree as they become available. When a new release file is downloaded from the log, the following additional validation procedures are executed.

1) Complete coverage: Verify that all elements covered by the release file are logged, including the release file itself, meta data files, and source code. This check ensures that all information the client depends on when installing package is public and immutable.

2) Source availability: For each software package it is checked that the corresponding source code has been logged. This ensures the presence of source code in case of code audits.

3) Version increment: When the contents of a package, its meta data, or its source code change, its version number must be incremented. This ensures no client can be tricked by maliciously meddling with version numbers or dependencies e.g., into installing an older version with known vulnerabilities. Changes to a binary package must be accompanied by a change in build instructions or source code.

4) Authorization: For each changed package source it is verified that it was uploaded and signed by developer authorized for this package.

5) Reproducibility: For each changed package source, the source code is compiled. The hash of the resulting binary package is compared to the hash contained in the meta data covered by the release file.

If any of these checks fail, the monitor issues an alert. Additionally, clients may contact monitors in order to compare their view on the tree. Using the tree roots they have observed respectively, they can request a consistency proof from the log. This proof demonstrates that the trees presented to both parties are consistent with each other.

IV. Evaluation

We implemented a prototype of this system for Debian, based on the Trillian hash tree implementation. In the following, we outline the security properties.

The security of the log system itself is derived from the pledged transparency overlay [5]. Alerts in monitor functions identify misbehaviour attributable to the archive.

Assume there is a backdoor in one of the packages. The client requests proof from the log that the release is covered by the hash tree. By monitor function III-B1, all meta information is present in the log and the corresponding source can be identified. By functions III-B2 and III-B5, the source matching the binary package is available. Since the source is signed by the developer, the backdoor can be attributed to them. Should any other step fail, the archive or log operator is to blame.

Concluding, we achieve the goals lined out in the beginning. By verifying the log inclusion of all code, any backdoor must be public and cannot be targeted, i.e. presented selectively to the victim. The provenance of code can be traced back to a specific developer by logging the signed source code. The relationship between source code and binary is verified by monitor functions.

V. Conclusion

This work presents a transparency system for APT-based distributions of software packages as are often used in Software Defined Networking. We achieve detection of targeted backdoors, ascertain mapping between source code and executable binary, and provide developer attribution in case of malfeasance. This approach provides security properties to softwarized networks that are not achieved in traditional networks. Investigating secure distribution of shared module components e.g., of network operating systems, remains future work.

Acknowledgements

We thank Lukas Schwaighofer for valuable suggestions and Markus Teich for help with programming.

References

[1] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A Look in the Mirror: Attacks on Package Managers," in *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08*.

[2] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable Key Compromise in Software Update Systems," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*.

[3] E. Messeri, A. Langley, E. Käsper, B. Laurie, and R. Stradling, "Certificate Transparency Version 2.0." [Online]. Available: https://tools.ietf.org/html/draft-ietf-trans-rfc6962-bis-24

[4] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds," in *26th USENIX Security Symposium (USENIX Security '17)*.

[5] M. Chase and S. Meiklejohn, "Transparency Overlays and Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*.