Agnostic embedding of digital signatures into programs

Sean Liao
sean.liao@os3.nl

Davide Pucci
davide.pucci@os3.nl

1. Introduction

Back in the early days of Information and Communications Technology (ICT), it was not much of an issue to not care about programs trust. Due to very limited ways of accessing devices, when what we nowadays call Internet was not preminent yet and the only way of interacting with them was to physically sit next to them, security constraints did not represent that much of a topic. The Operating System (OS) was not taught to deal with foreign entities, neither to be modularly extended with new programs, drivers and applications.

During the last thirty years, though, a lot has changed and the usage that was being initially done with such devices does not make that much of a sense anymore: if a device is not connected to the Internet, it is not worth to be used at all. Connectivity has become a major — if not the only big one — requisite, as it represents the the mean through which all the modern technology-related activities are done.

This break change brought along a new side requisite: the ability, from the OS perspective, to discern what is trustworthy and what is not. In fact, relying on connectivity, devices can be extended and customized, installing and removing applications and customizing general OS behavior. Although this represents a very much appreciated development, it brings the drawback to be possibly exploited to carry malevolent data exfiltration, system damages and so forth. Back to the point, the device manufacturer and more specifically the OS has to be able to prove the trustworthiness of every program intention, today more than ever.

The most traditional approach provides trust relying on a Public Key Infrastructure (PKI): any program is verified using public and private key pairs. The authors of the program use their private key to sign it, while the OS verifies the signature using their public key.

2. Problem

In a traditional PKI-based approach, the downside is represented by the fact that the OS has to ship, keep and maintain the PKI with all the trusted public keys: if an author is to be no more trusted, the OS has to release an update to drop the corresponding public key and the final user has to promptly update its installation to welcome the new trust constraint. Also, this approach is heavily dependent to the conception of binary executable: all the programs are supposed to be a single and standalone binary in which the signature can be shipped using areas in the binary format that the OS has predisposed for. Yet this is not always the case: nowadays, the the conception of program is way broader than before: a program can also be a simple script, which is relying on an interpreter — which is a binary, instead — to be run. In this case, which is the signature to be verified? The interpreter can be trusthworthy, but it does not mean the script which it is going to run is too. Then, how to secure the script up?

Also, with virtualization techniques constantly growing by usage, how does this topic applies to containerized applications [7] or even newer schemas such as unikernels [3]? More and more techniques and execution schemas are getting used and still a proper wider approach which can hopefully apply to all those areas is to be identified and/or standardized.

In fact, depending on the area in which the signature is to be inserted, different ways are there to tamper the program and extract the signature, eventually stripping it off, exploiting too permissive OS policies. Furthermore, manufacturers which are proposing and distributing a different OS are, in most cases not fully, adopting different approaches, using different areas in which to put the signature information, e.g., Linux distributions relying on files Extended Attributes (xattrs), functionality ignored but offered specifically by the file system.

This research tries to deepen this lack proposing a novel approach to the problem, which consists of appending data onto the end of the program file, without further needs of dedicated areas in the binary format specification. The way it can be applied on scripts consists instead of payloads appended under the shape of comment chars. The signature this way, regardless of the shape of the program, is always available, sticking to the execution unit that is going to be run, eventually guaranteeing cross trustworthiness between different and co-interacting program files. Furthermore, such an approach would survive to most — if not all — transport methods, e.g., downloading over HyperText Transfer Protocol (HTTP).

This solution resemble what the DigSig [2] and bsign [11] projects have been trying to do, extending it to a more varied environment, which involves different programs shapes.

3. Related Work

Relevant prior work in this area is substantially consisting on whitelisting approaches which discriminate on different information.

3.1. Application-based whitelisting

First group enforces integrity and/or signature based on the application itself. Depending on the underlying OS, there are different solutions. For Linux systems, the Kernel Integrity Measurement Architecture (IMA) [5] project has started in 2005: it checks the integrity of the program by looking at a specific field in the file system extended attributes, with eventual integration with Trusted Platform Module (TPM) using attestation. There is the possibility to modify interpreters to enforce IMA to enable scripts support [1].

Microsoft's project Integrity Policy Enforcement [8] for Linux systems has started in 2020 and represents a Linux security module to do hash-based integrity measurement.

The aforementioned DigSig [2] and bsign [11] projects started in 2002 and got shutted down in 2009: the idea resided into a kernel module which enforced signed-only binaries execution, by adding hash and signatures in an extra Executable and Linkable Format (ELF) section.

Gatekeeper [1] is a security feature added from MacOS version 10.7.3 to enforce signed applications.

AppLocker, on the other hand, is the solution adopted by Microsoft for Windows: enabled from Windows 7, it enforces discriminating on path, execution context and signature.

In the cloud world, there are two known solutions: Docker Content Trust (DCT) [4], adopted since 2015 for signing Docker containers and to be enforced during containers build and execution, and Binary Authorization [6] service from **GCM!** (**GCM!**), developed since 2018, to verify Docker containers signature at execution phase by including an attestation.

3.2. Path and attributes -based whitelisting

The second relevant group of solutions is based on application location (path) or other attributes.

SELinux [10] is a Linux security module whose development has started in 2000, which enforces access control based on file system attributes.

RedHat, in 2016, started the File Access Policy Daemon [9], which is a Linux security module which can integrate with SELinux to enforce path-based whitelisting.

Canonical AppArmor is also a Linux security module developed in 1998 and still under active development, to enforce path-based access control constraints.

Regarding Windows, there are several third-party applications to achieve whitelisting, being the following the most relevant:

- Ivanti Application Control
- McAfee Application Control
- Trend Micro Application Control
- Faronics Anti-Executable
- Kaspersky Whitelisting
- Airlock Application Whitelisting
- Thycotic Application Control

1. https://marc.info/?l=linux-kernel&m=111682497821375&w=2

4. Research Question

This section presents our main research question and sub-questions that follow from it.

Is the suffix-based signature a feasible solution for storing and validating signed code?

This question can be divided into these sub-questions:

1) Does it work for compiled executables?
2) Does it work for scripts and interpreted runtimes?
3) How does it compare to already existing solutions?
4) Can it work as an extension to existing framework?

5. Methodology

We implement and test the hypothesis that checksums and signatures can be attached to arbitrary executables and scripts without affecting normal operation. We survey and select method(s) to enforce verification before execution. Different methods likely needed for binary vs scripts. Possible candidates are as a security module with integration into Linux IMA, hijacking libc, and hooking into system calls directly with eBPF. Finally, we compare trade-offs with existing systems identified in related work.

References

[1] Apple. *Safely open apps on your Mac*. 2019. URL: https://support.apple.com/en-us/HT202491.

[2] Ericsson Research Canada. *DigSig (Digital Signature... in the Kernel) and DSI (Distributed Security Infrastructure)*. 2009. URL: http://disec.sourceforge.net/.

[3] Michael De Lucia. *A Survey on Security Isolation of Virtualization, Containers, and Unikernels*. May 2017.

[4] Docker. *Content trust in Docker*. 2020. URL: https://docs.docker.com/engine/security/trust/content_trust/.

[5] Dsafford et al. *Integrity Measurement Architecture (IMA)*. 2020. URL: https://sourceforge.net/p/linux-ima/wiki/Home/.

[6] Google. *Binary Authorization*. 2020. URL: https://cloud.google.com/binary-authorization.

[7] Docker Inc. *What is a Container?* 2018. URL: https://www.docker.com/resources/what-container (visited on 04/20/2020).

[8] Microsoft. *Integrity Policy Enforcement (IPE)*. 2020. URL: https://microsoft.github.io/ipe/.

[9] Redhat. *File Access Policy Daemon*. 2020. URL: https://github.com/linux-application-whitelisting/fapolicyd.

[10] Redhat. *What is SELinux?* 2019. URL: https://www.redhat.com/en/topics/linux/what-is-selinux.

[11] Marc Singer. *Bsign*. 2002. URL: https://github.com/digsig-ng/bsign-mirror.