Enforcing SBOMs through the Linux kernel with eBPF and IMA

Alex Crawford
EdgeBit, Inc.
San Francisco, California, USA
alex@edgebit.io

Eugene Yakubovich
EdgeBit, Inc.
San Francisco, California, USA
eugene@edgebit.io

Rob Szumski
EdgeBit, Inc.
Richmond, Virginia, USA
rob@edgebit.io

ABSTRACT

At build-time, we generate SBOMs and sign containers. But at run-time, tools like Open Policy Agent and Kyverno only check the signatures or Kubernetes Pod attributes, not the full contents. Let's go beyond only checking container signatures and explore the possibility of checking an entire container against its software bill of materials (SBOM) in real time.

This paper will discuss how features in the Linux kernel – Integrity Measurement Architecture (IMA) and eBPF – could eventually be used on a cluster's nodes to secure running containers and potentially other software. We are aiming to provide this protection without requiring a reboot or installing non-standard kernel modules. This would allow end users to protect machines via a Kubernetes DaemonSet or systemd unit without a need to tweak an operating system image or use a custom kernel.

CCS CONCEPTS

• **Security and privacy** → *Vulnerability scanners*; **Virtualization and security**; *Software security engineering*.

KEYWORDS

Extended Berkeley Packet Filter (eBPF), Software Bill of Materials (SBOM), Integrity Measurement Architecture (IMA), software supply chain security, file integrity

ACM Reference Format:
Alex Crawford, Eugene Yakubovich, and Rob Szumski. 2023. Enforcing SBOMs through the Linux kernel with eBPF and IMA. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23), November 30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3605770.3625206

1 INTRODUCTION

Linux Integrity Management Architecture (IMA) [6] is a lesser-known feature of the Linux kernel which is used to measure and enforce file contents in a high-performance manner. IMA makes use of filesystem extended attributes to record a digest or signature and then checks it against the file contents whenever the file is accessed. This is only useful for static files however, since legitimate modification of file contents (e.g. updating `/etc/resolv.conf`) would cause the IMA digest/signature to fail.

While it's possible to write an IMA policy which protects a subset of files while still allowing the rest to be modified, it wouldn't provide much security since an actor who can maliciously modify file contents can similarly modify extended attributes to match. The extended attributes also need to be protected.

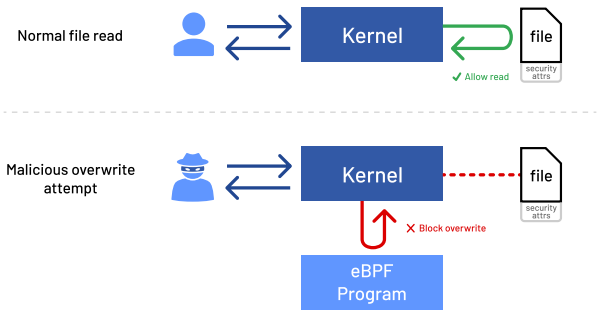

Figure 1: Protecting file integrity and blocking malicious file overwrites

Filesystem extended attributes are typically protected by Extended Verification Module (EVM) [3], which compares a subset of extended attributes against a digest or signature. This mechanism isn't always included in kernel builds though, and isn't currently present in Google's Container-Optimized Operating System or Amazon Linux. There exists another way to potentially limit modification of extended attributes: eBPF.

Extended Berkeley Packet Filter (eBPF) [4] is a facility of the kernel which allows code to be loaded and run within the context of the kernel itself. In order to ensure the integrity and stability of the system, a number of constraints are put in place that limit what eBPF code can do at runtime.

Even with these restrictions, there is support for writing Linux Security Module (LSM) hooks which can reject specific operations ranging from mapping memory to renaming files. Unfortunately, it doesn't appear to support enough of the functionality required to implement the proposed scheme; but with the right changes, expanded under Technical Limitations, it could.

2 ENFORCING SBOMS AT RUNTIME

Once upstream Linux incorporates the required eBPF functionality, an operational team would have the ability to prevent modification to particular files and could then choose to bless only that software which meets a given criteria and know that it won't undergo any further changes, malicious or otherwise. The process of blessing container images or files is done by recording the digest or signature in the file's extended attributes as part of the build pipeline.

A secure production configuration would make this step a prerequisite before deployment to the environment. If any of the blessed files are modified, IMA prevents the file from being used, returning

ACM ISBN 979-8-4007-0263-1/23/11.
https://doi.org/10.1145/3605770.3625206

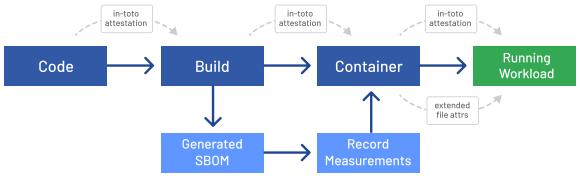

Figure 2: Chain of trust from code to running workload

-EPERM from all file operations. Access to files without a digest or signature could be allowed or denied based on the risk posture of the application.

3 MODIFYING A BUILD PIPELINE

An SBOM is used as the source of truth for dependencies used within the application. If an SBOM is not available, an SBOM generation step must be added.

The build environment measures the inputs (source code, compilers, etc.) and those measurements are then compared against values derived from entries in the SBOM. Optional validation against publicly-available release artifacts can be done. If everything checks out, the built asset is blessed with the appropriate IMA digests or signatures and is ready for deployment.

SBOM formats are standardized and all common SBOM generation tools work with this technology stack. Tools such as Syft [1] are popular because they automatically detect a variety of languages and frameworks, ensuring that all dependencies, including transitive ones, are included for measurement.

Software vendors and open source projects are increasingly including SBOM artifacts when new software versions are released. When a library or dependency is imported into a project, its SBOM can be used as a source of truth in addition to an SBOM generated within a build pipeline. The in-toto attestation framework allows for these imported SBOMs to show how they were derived, which chains together all of the elements in your supply chain.

4 ADOPTION AT SCALE

This scheme can be easily implemented due to the increasingly common adoption of SBOMs and the ability to work against the direct-deployable artifact (e.g. a container or Linux package). SBOMs provide the inventory down to the file level and serve as a compliance artifact. The blessed measurements are easy to encode into the container by mutating the extended attributes and map directly back to the SBOM, which itself maps back to the source code.

In production, having all of the enforcement mechanisms within the Linux kernel [5] allows for an extra degree of trust without sacrificing performance. Loading the eBPF program via Kubernetes DaemonSet or container allows for deployment of the protection without manipulating the underlying operating system, preserving the so-called "golden images".

As stated above, this scheme unfortunately cannot be implemented today. The next section highlights the mechanisms that would need to exist in order to implement it, but isn't feasible in today's Linux landscape. For anyone attempting something similar, this would be a good reference and a potential jumping-off point to do a deeper investigation.

5 TECHNICAL LIMITATIONS

The major limitation which prevents eBPF from protecting extended attributes is the fact that they cannot be read from its context. eBPF provides a number of helper functions which allow code to interoperate with the kernel, but there isn't one for reading extended attributes from a given inode. [2]

If such a helper did exist, a simple implementation of this scheme could look at the extended attributes to see if any of them belong to the security class, and deny the file mutation if so.

Pointer-chasing from a given inode might be an option, if it weren't for the fact that extended attributes are filesystem-specific and require calling into filesystem-specific code — disallowed in eBPF's context.

Calling out to a userspace process which does the extended attribute lookup also proved to be a dead end, since LSM hooks don't have a mechanism for synchronous communication. There has been some interest by the development community in adding this functionality, but progress has been slowed by concerns from upstream developers about the security impacts of such a change. Assuming that these issues are addressed though, the full story can be realized.

6 CONCLUSION

Enforcing the integrity of an application workload through its SBOM allows security teams to use the SBOM as a representative of run-time state. SBOMs enriched with vulnerability information on a continual basis provide an accurate picture without requiring invasive monitoring of the running workloads.

Our research has shown that all of the software supply chain tools exist to support this behavior and that the series of Linux kernel modifications required to make the proposed scheme complete are technically viable. It remains to be seen if upstream consensus can be reached to extend eBPF's ability to reach extended file attributes or allow synchronous communication with LSM hooks.

Adoption of this scheme is uniquely ideal for large scale deployment because of the universal APIs offered by kernel-level enforcement and eBPF's ability to modify Linux behavior without requiring custom patches or a heavily customized golden OS image.

REFERENCES

[1] Anchore. [n. d.]. *Syft*. https://github.com/anchore/syft/
[2] Alex Crawford. 2023. Getting acquainted with BPF as a security tool. Retrieved August 30, 2023 from https://edgebit.io/blog/acquainted-with-bpf/
[3] Dmitry Kasatkin and mzohar. 2023. Linux Extended Verification Module (EVM). Retrieved September 20, 2023 from https://sourceforge.net/p/linux-ima/wiki/Home/#linux-extended-verification-module-evm
[4] kernel.org. 2023. BPF Documentation. Retrieved September 20, 2023 from https://docs.kernel.org/bpf/
[5] kernel.org. 2023. Linux kernel. Retrieved August 12, 2023 from https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/security
[6] Gentoo Linux. 2023. Integrity Measurement Architecture. Retrieved September 20, 2023 from https://wiki.gentoo.org/wiki/Integrity_Measurement_Architecture