DiffBOM: Does SBOMs Accurately Reflect File System Status of IoT Devices?

Anonymous Author(s)

ABSTRACT

Modern IoT devices running embedded Linux often include various software packages providing key functionalities. However, it has been repeatedly shown that by compromising these software packages, attackers can take control of the whole device. A powerful tool against such software supply chain attack is a software bill of material, or SBOM. An accurate SBOM can help users quickly identify and mitigate potentially compromised software package in an IoT device. But whether SBOMs accurately reflects the content of file systems of IoT devices is largely unknown. The goal of this paper is to determine SBOM coverage, defined as the percentage of files in a file system claimed by an SBOM, in common IoT devices running embedded Linux. We develop DiffBOM, a tool that automatically collects package manager information as the SBOM for the device, compares the information against the file system, and outputs metrics about the coverage.

Using this tool, we discover…

CCS CONCEPTS

• **Security and privacy** → ;

KEYWORDS

template; formatting; pickling

1 INTRODUCTION

Recent times have seen the emergence of a novel attack vector of software called supply chain attack. Instead of targeting the software itself, supply chain attacks target dependencies of software for exploitation [**?** **?**]. As dependencies often works in conjunction with, or act as modules within the target software, vulnerable or malicious code from dependencies can be executed even if the software itself is not vulnerable or malicious. In addition, modern software often depend on various other software packages, which in turn are also dependent on several other packages [**?**], creating a large attack surface. Any software package an attacker manages to exploit affects the security of all other software packages that are either directly or indirectly dependent on it.

Supply chain attacks has already be demonstrated to be feasible and particularly damaging. One recent example is the Log4j vulnerability. Log4j is a popular software package for logging security and performance information on various services and applications [**?**]. A serious vulnerability was discovered in the software package that allows remote code execution [**?**]. Even if the applications and services are not vulnerable, they depend on and runs alongside with the exploitable Log4j software package, opening doors to potential attacks. The popularity of Log4j affected many software packages [**?**] and its effects could persist for years [**?**].

Many IoT devices run embedded Linux with various software packages installed to perform their functions. Thus they are also susceptible to supply chain attacks. Malicious players can compromise or take advantage of vulnerabilities in popular software packages used by IoT devices. As compromised software packages are pulled and installed while building and installing versions of IoT device firmware, devices become vulnerable to attacks. With the increase in deployment of IoT devices in critical areas such as healthcare and infrastructure, mitigating such attacks becomes critical [**?**].

Recent works done on software supply chain security have proposed several ways of auditing software packages provided by popular package managers such as npm and PyPI, thus mitigating such attacks [**?** **?** **?** **?** **?**]. However, these techniques have limited impact in IoT space. Most package managers are designed for personal computer or server users. Users or administrators directly specifies packages to install, and have complete control over how frequently software updates are carried out. The transparency allows users to take actions at moment's notice in case of vulnerability. In contrast, IoT devices often ship as a complete package, so users are unaware of the software components of devices. Updates are often shipped by manufactures as complete image files containing everything from bootloader and OS to software packages. These two factors limit user's ability to quickly take action with their IoT devices even with knowledge of compromised package, and updates addressing such attacks might take longer to apply.

Software Bill of Materials (SBOMs) could be powerful tools in mitigating supply chain attacks on IoT devices. SBOMs exhaustively list all software packages and components bundled with devices as well as information such as software version [**?**], thus eliminating the obscurity of installed software packages on IoT devices. With accurate knowledge provided by high quality SBOMs, users can quickly identify if they are affected by security vulnerabilities and take actions accordingly. However, SBOMs must accurately and completely reflect statuses of software installed on file systems of IoT devices for it to be useful. A poorly constructed SBOM misrepresenting actual states of file systems can lead to false positives or false negatives, defeating its purpose.

Therefore, this paper addresses the problem of if SBOMs accurately reflects file system status of IoT devices. We define SBOM Coverage as the coverage of packages claimed by SBOMs on actual file systems. We propose DiffBOM, a tool comparing versions of SBOMs to versions of actual file systems of IoT devices and provide several metrics on SBOM coverage over versions.

Using DiffBOM, we analyzed versions of selected embedded Linux file systems...

This paper is organized as follows. Section 2 of presents underlying ideas on software supply chain attacks and SBOMs. Section 3 introduces implementation of DiffBOM, followed by section 4 which examines the accurateness of DiffBOM. Section 5 and Section 6 describes and analyze our dataset containing different versions of IoT device file systems. Section 7 reflects on limitations of DiffBOM. Finally, Section 8 presents our conclusion.

2 BACKGROUND

Proxies of SBOM: Providing SBOMs for IoT devices is not common practice. Thus, almost no device in our dataset has available SBOMs for us to analyze. Thus, proxies, or other on-device information encoding the software packages and files present on device, must be used in place of SBOMs in our analysis. Although only accounting for packages installed through package manager, it proves to be a good proxy to SBOMs, as further demonstrated in the Evaluation section of this paper. This enables us to analyze a wide range of devices with or without SBOMs.

Package managers: Even in embedded Linux devices where updates are typically carried out by flashing the whole storage, package managers are used. An example of such package manager for embedded Linux is opkg [**?**]. Like all other Linux package managers, it automates the installation, management, and update of software packages, though it might not be accessible to the end user. Information on installed software packages are stored on the file system, providing a convenient and accurate proxy of SBOMs.

3 DIFFBOM

In this section, we will introduce our idea and implementation of DiffBOM.

3.1 Properties

In an ideal world, an SBOM contains information of every and all software packages and files present on file systems of embedded IoT devices. However, this is not always achievable in the real world due to several factors. For example, configuration files are created by some programs when first ran. Often times though, such discrepancies are the results of either buggy SBOM generation tools or poor practices such as manually loading or changing files. These discrepancies inhibit SBOMs' ability of preventing software supply chain attacks. Therefore, we derive SBOM coverage, a metric depicting how much of an actual file system is described by an SBOM, thus how well an SBOM represents a file system. DiffBOM is developed to derive SBOM coverage by comparing an SBOM to contents of a corresponding file system. The tool is aware of semantics of file systems, SBOMs, and proxies of SBOMs.

3.2 System Design

Our implementation of DiffBOM generally follow this three steps: SBOM Parsing, File System Parsing, and Comparison.

In SBOM Parsing, the tool gathers the claimed status of file systems from SBOMs. The tool is designed to be able to accept major SBOM formats, like SPDX, as well as popular proxies of SBOM, such as opkg metadata. With the emergence of the idea of providing SBOMs for enhancing supply chain security, we expect more and more devices to be shipped with SBOMs. Meanwhile, there are many legacy devices with limited manufacture support, and some device makers may be slow in adopting SBOMs. Therefore, supporting multiple SBOM formats and proxies ensures the tool's ability to analyze file systems of a wide range of IoT devices. The tool then correctly read and digest information from different SBOM sources, such as package names and their contained file names, and file hashes, for the Comparison step.

In File System Parsing, the tool parses the actual status of file systems. Images pulled from active devices may have additional runtime changes to the file systems. For example, in initial user setup for routers, some configuration files are changed or created. Thus, analyzing an update package is more ideal. Typical IoT devices we analyzed ship system updates as disk images. Updates are usually done by the OS or bootloader, writing contents of disk images directly to NAND flashes onboard. Thus there must be a procedure of extracting file system information from these disk images. Several runtime or tmpfs directories, such as /tmp/, /run/, /dev/, and /sys/, are ignored. These directories are populated on runtime, so no software packages will be installed on them and ignoring them helps minimizing noise in analysis. Then, the file hierarchy are organized in trees containing information such as file names and file hashes, for comparison with the claimed status of file systems.

In Comparison step, the claimed and actual file system status fetched in previous steps are compared. The tool analyzes the existence of claimed files in the actual file systems, and if possible, compare the hashes of those files. It also handle files claimed but do not exist in actual file systems (the missing files) and files in actual file systems not claimed by any packages (the unclaimed files), and output several metrics on SBOM coverage. We selected the following metrics for analysis: the number of ELF files with mismatching hashes (the changed ELFs), the number of files claimed by two or more packages (the multi-claimed files), the number of missing and unclaimed files, the number of symlinks (the unclaimed symlinks), regular (the unclaimed regular), and ELF (the unclaimed ELF) files in unclaimed files, and the total number of files. The existence of multi-claimed files can signal poorly constructed SBOMs or unrepresentative proxies, since a file is unlikely to be installed by multiple packages. The number of changed ELFs, and percentage of unclaimed and missing files represents the differences between the claimed and actual state of file systems in question. We omit counting directories because a claimed directory does not mean its contents are claimed, and the image building process should create required directories regardless. We further divided the number of unclaimed files into symlinks, regular files, and ELF files, because an unclaimed ELF file signals a more significant difference, while unclaimed regular files and symlinks could be the result of loading custom assets or configuration. For example, the manufacture could be loading a custom web interface to a file system. Although not an ideal practice, it is not as significant as a missing executable file. Similarly, we only check for changed ELF files because a changed regular file could commonly be a configuration file and not as significant as a modified executable.

3.3 Implementation

We implement DiffBOM with Python. Two modules, bomParser, responsible for the SBOM Parsing step described above, and fileTree, responsible for File System Parsing and part of Comparison step, works with the main DiffBOM code.

bomParser currently supports SPDX SBOM format as well as opkg metadata. To parse SPDX SBOM, the tool utilizes spdx-tools python package. spdx-tools detects format of the SBOM, and outputs an object containing information of all packages. bomParser

then convert it into a Python dictionary indexed by the the package name, where each element is a list of dictionaries containing file names and hashes. opkg metadata is contained in a directory. Each package has several files associated to it, with the file names consisting of package names and several kinds of extension. The file with .list extension lists all files associated with the package, thus bomParser reads the content of the file and organize the information in the above format. However, opkg does not provide any file hash information, so changed file detection has to be omitted.

Since different file systems are used for different devices, we chose to manually extract the images first using tools such as unsquashfs or ubi_reader. Then fileTree organizes file system information with an n-ary tree. Each node uses lists to keep track of all its child nodes, and file name, file type, hash, and symlink destination are stored.

To conduct the Comparison step, first the tool goes through the tagging step. It takes the dictionary generated by bomParser and enumerate all of the packages. For each package, each file name is searched in the directory tree generated by fileTree, in a linear and depth-first manner. Our implementation is not sensitive to performance, so this algorithm is chosen for its simplicity. If the file is found, an attribute documenting the package the file is belonged to is modified. If the file is not found, a counter of missing file is incremented. If the file already has the package name attribute modified, the node is added to a set of multi-claimed files. We use set for this purpose because a file might be claimed for more than two times Using set avoids error caused by multiple counting. If hash is present in SBOM information and the file is an ELF file, its hash is also compared and if different, another counter for changed ELF files is incremented.

After the tagging step, the tool then scans the whole file system tree. It recursively goes through each directory in a depth-first manner, counting the number of unclaimed files and symlinks. Then all the counters are outputted in csv format for analysis.

4 EVALUATION

5 DATASET

6 ANALYSIS

7 LIMITATIONS

8 CONCLUSIONS

A APPENDIX

REFERENCES