DiffBOM: Does SBOMs Accurately Reflects File System Status?

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

2 BACKGROUND

3 DIFFBOM

In this section, we will introduce the desired behavior of a DiffBOM tool, as well as introducing our implementation of DiffBOM.

3.1 Desired Behavior

To evaluate if an SBOM accurately reflects the status of a file system, a DiffBOM tool should generally follow this three steps: SBOM Parsing, File System Parsing, and Comparison.

In SBOM Parsing, the tool should gather the claimed status of file systems from SBOMs. The tool should 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 should 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 should parse the actual status of file systems. Since images pulled from active devices may have additional runtime changes to the file systems like initial setup, an update package is a more ideal target with less noise. 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. This could be done either by unpacking them into files and directories on local disk, or by directly analyzing file system on disk images. Several runtime or tmpfs directories, such as /tmp/, /run/, /dev/, or /sys/ should be ignored. These directories are populated on runtime, so no software packages will be installed on them and should be ignored to minimize noise in analysis. Then, information such as file hierarchy, file names, and file hashes should be stored for comparison with the claimed status of file systems.

In Comparison step, the claimed and actual file system status fetched in previous steps should be compared. The tool should analyze the existence of claimed files in the actual file systems, and if available, compare the hashes of those files. It should 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. 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, while the number of changed ELFs, and unclaimed and missing files relative to the total number of 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. 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 custom asset or configuration loading. 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.

3.2 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, and using set avoids error caused by multiple counting. If hash is present in SBOM information and the file is an ELF file, their 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. The tool recursively goes through each directory in a depth-first manner, counting the number of unclaimed files and symlinks. Then all the counters are added to produce the desired metrics. The metrics are outputted in csv format.

4 EVALUATION

5 DATASET

6 ANALYSIS

7 LIMITATIONS

8 CONCLUSIONS

A APPENDIX

REFERENCES