Assuring SW Supply Chains Against Indirect Attacks - An Evidence-based Approach

Partha Pal, Aaron Paulos, Rick Schantz
Raytheon BBN Technologies
{Partha.Pal, Aaron.Paulos, Rick.Schantz}@Raytheon.Com

Supply chain attacks are neither a surprise nor unprecedented. However, much has changed from the lifecycle attacks (insertion of code that would activate at a certain time or under certain conditions) of the 1990s. A 2013 MITRE report [1] provided a holistic view of supply chain attacks and a framework for supply chain risk management (SCRM) as part of system security engineering (SSE) together with prevalent attack patterns and potential approaches for assessing malicious insertion in critical components of DoD systems being acquired or sustained. The framework covered the entire end-to-end supply chain for DoD systems, starting from primary hardware and software developers to the Program Office. The attacks considered in the report directly injected code or data into the system of concern and originated at technical and engineering personnel directly involved in the process (in other words, insiders). In contrast, in this paper, we limit ourselves to software products (i.e., binary releases) and narrowly focus on the supply chain segment that starts with source code and ends with the signed binary as a released product, and emerging threats of the kind highlighted by the SolarWinds incident.

The complexity of the software development process has grown as the length and complexity of the modern software stack has grown. In addition to multiple layers and complex dependencies, modern configuration management and build systems like Maven and Gradle combine written code with code imported from external repositories. The percentage of imported code compared to code custom written for a DoD system has gone down significantly. CI/CD and DevSecOps make the software production process even more dynamic, demanding a faster requirement-to-release turnaround. This has widened the possibility of *indirect supply chain attacks:* in order to compromise a product A, the adversary no longer has to be directly involved with the development or production of A, nor does he need to inject code or data into A directly. Attacks of this sort have happened before (e.g., attacks on Github to compromise Linux [2] [3], which could be used to attack software running on Linux). Proof of concept attacks [4] demonstrated the use of naming collision to import malicious external libraries. The SolarWinds incident demonstrated the scope and range of damages such indirect attacks can unleash, and highlighted the utter insufficiency of relying on code signing as the only measure of trust.

The 2021 Executive Order (EO) provides an opportunity to revamp the guideline for acceptance of software, specifically binary releases. To complement the attention paid to assurance/certification/validation of the software product (which primarily concerns the product's functions and security properties), and the OPSEC of the development enterprise (including the integrity of the software development tools and environment), we take the position that *provenance of the binary release* and the *integrity of the process that led to it* must also be considered. Code signing is not sufficient, because it fails to capture the history of the events leading up to the release in a verifiable manner, and certification still is resource heavy and time consuming.

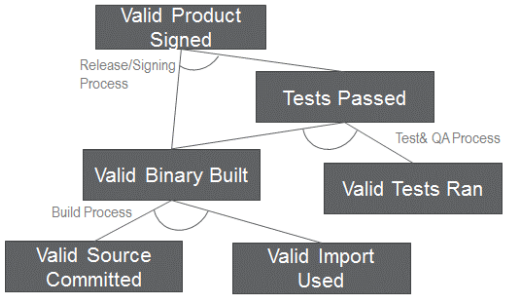

Figure 1: A simplified assurance case for provenance of the software product and the integrity of its production process

Addressing the threat of indirect attacks on binary products accepted into federal systems of critical importance require verifiable evidence that the released binary is the same one that was produced from approved sources and passed required tests/analyses. Figure 1 illustrates the idea using a simplified assurance case. At a high level, a signed binary ***B*** will ideally accompany linked evidence that assert claims like the following:

- The same ***B*** above was used in testing and analyses, and passed all the required tests and scans, and
- The same ***B*** above was produced with only approved imports ***I*** and committed source code ***C***, and
- The ***C*** above includes only code that supports a specified requirement or change request.

As the assurance case structure shows, each node represents a claim about artifacts leading up to the binary product. Only AND connectors are shown, but other connectors e.g., OR, ONE-OF are also possible. There are multiple cases to consider about these claims. First, for some of these claims, we already have the means to substantiate them. For example, user credentials, commit logs and MD5 sums can be used to assert the validity of a source file, or test results can be used to assert that the tested binary passed the tests. Second, there are cases where the evidence can be discerned with some additional, possibly manual, work. For example, assessing source code with specified requirements and models can assert that the committed code does not include anything outside of what is required. Documentation authorizing imports/external repositories and build configuration can attest the validity of imports. Finally, there are cases where we currently lack enough evidence. This situation may arise anywhere, but is critical for claims about lossy or one-way *transformation of artifacts,* i.e., claims that link one or more input artifacts with one or more output artifacts, because these transformations are key pieces of the provenance chain leading to binary product. Consider a simplified build process that takes in a bunch of source files and produces a binary. How can we assert that the sources that went into the build process were actually used to produce the binary that came out of it (SolarWinds software was compromised in this manner)? Without this linkage, we can assert that the sources are all valid, and the binary passes all the tests, but will miss the case that an indirect attack may have caused the build process to use malware-laden shadow source file to build the binary instead of the original source that it took as input.

We thus organize our recommendation into two broad categories: State of Practice (SOP) and R&D. In the R&D category, more work is needed to develop innovative means to assert claims about artifacts, including ways and means to augment lossy/one-way artifact transformations to produce evidence as by-products, as well as means to reduce regular/routine human involvement in assessment of claims. Software watermarking, compiler technology and program analysis techniques can serve as jump-off points for generating machine-usable evidence linked with artifacts generation and transformation. Block chain technology, with its zero-trust framework for maintaining consistent transaction records can deter malicious actors attempting to insert corrupt code and data, and at the same time offers a reliable parts tracking capability in support of the software production process. In the SOP category, we recommend a progressively tiered approach (more stringent evidence for more critical contexts), possibly starting with the status quo of relying on code signing for non-critical software. As the criticality of the software rises, we should demand provenance of the binary and evidence that the integrity of the process that produced the binary was not compromised. For some cases, an audit trail that assigns ownership stakes to vendors and communities that export 3rd party sources and binary may be sufficient. Some situations will be able to afford the time for manual checking. Some situations will call for online (fast)/on-demand checking of the evidence. Time consuming manual evidence extraction and checking is not consistent with the CI/CD and DevSecOps ideals, but the gradual approach we propose is likely to face less resistance, and nicely augments the other techniques to mitigate the supply chain risk (e.g., securing the hardware and software tools, increasing the OPSEC of the vendors, making certification more efficient etc.) being considered and new R&D capabilities we recommend.

References

1. J. F. Miller. Supply Chain Attack Framework and Attack Patterns. 2013
2. C. Campanu. Cannonical GitHub Account Hacked, Ubuntu Source Code Safe. ZDNet 2019
3. D. Goodin. Kernel.org Linux repository rooted in hack attack. The Register.2011.
4. A. Sharma. Researcher hacks over 35 tech firms in novel supply chain attack (in www.bleepingcomputer.com)