VIRTUSLAB

Supply Chain Attacks: How They Work and How to Defend Your Codebase against Them

When I was choosing the topic for my master's thesis, software supply chain security seemed both an interesting and important issue. However, multiple months of sleepless research later, I realised my work had only brushed the surface of the problem.

The problem which, if ignored, can easily cause anything ranging from a minor bug to a literal disaster. Should we be scared? What can we do to be safe? This article will do its best to answer these questions briefly, while still doing justice to how serious the danger is. As a bonus, I will also mention how a build tool called Bazel can help in the fight.

Capture Checking

What is a software supply chain?

Let us start with something far from the world of software - aviation. From the passengers' perspective, any technical problem is simply a "broken airplane". Initial diagnostics might point to issues with its wings, engines, or any major part of the aircraft. However, each component consists of hundreds or thousands of smaller elements. A huge miracle of engineering (because that is what airplanes are) worth millions may be rendered out of order by a damaged half-inch rivet manufactured on another continent, many years ago. Even the smallest element has an impact on how the entire plane operates; its quality is dependent on multiple factors, like raw materials and tools used.

Software supply chain development is not much different from building an airplane, baking a cake, or any production process. It is easy to think of a project as a bunch of code and perhaps some compiled artifacts. However, programs never exist in a void - even a simple Python script needs to use an interpreter, which in turn requires an operating system. Diving deeper takes us to the machine that everything runs on and all its components. The aforementioned supply chain is relatively short and consists of elements from trusted providers like Python or Microsoft (though, of course, they are not invulnerable to attacks). However, the situation is rarely this simple. A typical corporate codebase uses multiple libraries, each of which is its own project with its own dependencies. A minor typo committed into a simple library may spill down the chain and end up as a major issue for millions of users. Keeping the entirety of such a chain under control is simply impossible, forcing developers to put some trust into the quality of libraries, compilers, interpreters, and other similar components.

Complicated? Everything becomes even more so when we introduce security into the equation. If a single, small bug is capable of causing noticeable problems in some distant supply chain links, it is terrifying to think about what an intentional interference can cause. No matter how scary it seems, it is important to realise that not only is it possible, but it does happen and causes real harm. Whenever an action like that is malicious, it deserves the title of a "supply chain attack".

How to attack a software supply chain?

Despite the provocative nature of this chapter's name, this is by no means a guide on how to sabotage someone. Nevertheless, the knowledge of how supply chains are attacked is one of the best tools in defending them. There are many possible strategies - some of them are quite simple, others consist of a great number of small actions and preparations. Moreover, various approaches can be mixed and matched to increase the odds of success. While being only the tip of an iceberg, the few attack types described here should be enough to get the gist of how vulnerable the software supply chain is. Unlike building an aircraft, a process with a clear start and finish, software development usually goes on for years on end (work on Emacs has not ended despite it being already half a century old), giving potential attackers a lot of time to prepare their actions.

When wanting to attack a software development project, the simplest approach is to introduce malicious changes directly into the code. This danger is especially significant in open source projects, which are the unquestionable cornerstone of software worldwide, inviting everyone to become a contributor. Due to the nature of supply chains, a minor interference in a library maintained by a volunteer community can result in major issues with every project depending on it, be it as small as student assignments or as large as commercial operating systems. This is a perfect environment for attackers, since no access to their main target itself is required - a codebase may have the strictest protections in place, but still inherits some malicious functionalities from its dependencies, either direct or transitive. A good example of this approach is an attack on XZ Utils, a command-line set of tools for file compression. One of the contributors helped develop it for two years, gaining everyone's trust. At one point, they introduced a backdoor, making it possible to remotely gain access to machines where the malicious software version was installed. As a result, the backdoor was present in every Linux distribution containing this particular version of the tools, therefore affecting huge operating systems without interfering with them directly.

A popular alternative to modifying the code is to attack publishing mechanisms. Source files remain unchanged, but the artifacts available for download are built from another repository. This can be achieved by taking control of automatic publishing mechanisms, or by manually building and uploading a malicious version, usually using stolen keys required to access package repositories like PyPI (for Python), Maven (for JVM-based languages) or npm (for JavaScript).

Becoming a contributor in a popular library repository is a good strategy. Depending on the project size, attackers can enjoy various benefits - big projects make it easier to hide among large numbers of commits, while small projects have fewer people to notice someone's ulterior motives. In any way, a perfect situation for an attacker would be to become the only person working on a codebase, but unless the project is old and forgotten, an opportunity like that rarely arises. This is where something called "typosquatting" comes into play. Instead of introducing changes into an existing software

project, attackers create their own with a similar name and wait for programmers to make typos when defining project dependencies. It is usually a clone of the original in order to have all the functionalities the users expect, but with some malicious modifications. To improve the odds of a successful attack, some additional measures can be implemented, for instance, creating a trustworthy-looking webpage or publishing programming tutorials where the fake library is being used. Moreover, misspelled name usage can be forced by contributing it directly into an open codebase, mixing this method with the previous one.

Another strategy for introducing a bad library to supply chains is to create one that has no suspicious traits at first. This method is neither easy nor fast, but can be especially difficult to detect. The process starts with finding an unfulfilled need, like a programming language functionality the users lack, or perhaps some set of utilities to help software developers be productive. The attacker then creates a library that solves these problems and updates it regularly. It contains nothing suspicious, so even if an ambitious user performs a detailed scrutiny of the project's codebase, nothing is there to be found. After gaining a satisfying number of users, one update can surprisingly contain some malicious code, hidden between important fixes and features. When the library users update it past this version, the supply chain gets exposed to unwanted traits that the attacker implemented.

Lastly, supply chain attacks may target those with the most control over a project's codebase - its members. While this can be done by a direct action like blackmail or bribery, from a software development standpoint a programmer is one with the tools they use. Among these are IDEs - applications in which the code is being written. They usually share a trait uncommon for other programs - plugins. Overall, the possibility of installing extensions that add functionalities to an IDE is extremely useful. However, developers have to trust every plugin they have installed, since even a single compromised (or intentionally malicious) plugin can make the whole application perform undesirable actions. Since IDEs are often given high permissions in operating systems, their attackers can enjoy almost unlimited access to computers used in development, including making changes in code written by the users. If the developer in question is trusted by their team, these modifications might not even undergo the review process. This issue becomes even more serious in recent times, as nowadays it is typical for software developers to have some kind of an artificial intelligence agent installed in their IDE. If for any reason the model becomes malicious, having broad permissions will allow it to cause serious damage in the codebase and on the affected machine.

How to defend a software supply chain?

Software supply chain attacks are generally considered difficult to defend against. However, there are many methods that can greatly reduce the chances of becoming a victim, some of them very simple to implement. Moreover, it is important to remember that all protective measures improve the security of not only the software project itself, but also the supply chain as a whole.

First and foremost, keeping software development processes secure requires having strict verification procedures in place. While annoying at times, having every change in the code be reviewed by at least one other person is essential. Allowing shortcuts like self-review or pushing directly to production branches makes it much easier for potential attackers to sneak malicious functionalities into the codebase. Likewise, reviewers should never accept changes without at least a quick glance, searching for suspicious parts like unnecessary dependencies. As mentioned before, the “I trust this developer” approach is dangerous, since the programmers themselves are vulnerable to various attacks.

Another strategy is to make sure the project and its contributors are protected against other, simpler types of attacks. Even the biggest and most elaborate successful supply chain attacks usually start their execution by gaining some form of basic access through means like finding weak passwords or exploiting basic system misconfigurations. Companies and organisations (including open source project owners) should regularly update and verify access rights and enforce strong account protection. This is an area where every individual developer can affect how secure the entire ecosystem is, meaning that being cautious should be highly encouraged.

Lastly, a way to improve software security is to actively look for any signs of an attack taking place, both inside a particular project, and in all other supply chain links. Software supply chain attacks are rarely instantaneous - the larger the operation, the more steps and time it requires. This gives the defenders the time to notice an attack and stop it, before it causes any significant harm. Even if a project is already fully compromised, realising that fact can greatly limit how much other links of the supply chain are affected. If more people perform security checks on a software, the chance of finding any dangers increases. Open source projects are more exposed to malicious interference, but they also let anyone search for signs of an attack. This creates an environment where companies are incentivised to use their own resources to help protect open software, effectively helping the whole supply chain remain secure.

To improve transparency and help respond to attacks in complicated software supply chains, it is advised to create a document called Software Bill of Materials (SBOM). A SBOM contains detailed information about all direct and transitive dependencies used in a project. When a library is affected by an attack, having a SBOM present (and regularly

updated) makes it much easier to notice dependence on an infected version. Moreover, in order to protect its software supply chains, the European Union is introducing a Cyber Resilience Act (CRA), which makes it a legal obligation to generate and publish machine-readable SBOMs for released software. This greatly improves the chances of transparency becoming an industry standard.

How can Bazel help?

Publishing a software codebase is enough to make third-party verification available. But the verification should also be made as easy and reliable as possible. Every aspect where the project is not fully deterministic makes the process more complicated and therefore less secure. To address this issue, the idea of “Reproducible Builds” was conceived. According to this concept, building a given source code should always produce the exact same results, regardless of any external factors. When this requirement is met, third parties can compare published artifacts bit by bit with those generated locally - any difference may point to an interference and can be investigated further. This is especially helpful against attacks on package repositories. However, applying this concept in practice can prove difficult, as even something as trivial as a timestamp can already make the artifacts vastly different.

A build tool called Bazel makes implementing build reproducibility much easier. While created with big monorepos in mind, it can also benefit small projects. Bazel builds are hermetic by default - unless explicitly told to do so, they do not depend on external factors like locally installed compilers or libraries. Instead, the tool downloads everything by itself according to its configuration files. Those are a part of the repository and can be uploaded into version control systems - two people having the same source code revision will have their builds configured in the same exact way, leading to similar artifacts. Going even further, when a launcher tool called Bazelisk is used, projects can define the Bazel version the builds should be run with, removing yet another non-deterministic factor from the process. When the Reproducible Builds concept becomes easy to implement, more projects are created in a secure way, improving the software supply chain as a whole.

As a nice bonus, among other functionalities like gathering licenses to ship with the product, a SBOM-generating Bazel extension is in active development. At the time of writing this article, this set of tools is still a fresh solution, but shows that Bazel’s capabilities for securing the software supply chain are constantly improving.

Conclusion

To summarise the biggest takeaways:

- Supply chain security is essential for software integrity,
- There are many methods available for attackers,
- Open source projects are easier to attack, but they also provide far more opportunities for third parties to verify them and detect any interference,
- Defending is not hopeless and starts from basic good practices every day,
- Bazel makes projects easier to scrutinise, allowing third parties to quickly detect attacks.

Overall, I hope this article clarifies what the software supply chain is all about and sparks interest in helping to secure it.