Avoid Supply Chain Attacks Similar to SolarWinds Orion

blog.truesec.com/2021/01/07/avoiding-supply-chain-attacks-similar-to-solarwinds-orions-sunburst

Share

Securing build servers and the development process as a whole is crucial to avoid becoming part of a software supply chain attack. SUNBURST is malware that was spread by breaching the build server for SolarWinds' Orion product. Using threat modeling it is possible to identify mitigations to reduce the risk and improve the security of the development life-cycle.

Summary

Securing build servers and the development process as a whole is crucial to avoid becoming part of a software supply-chain attack. SUNBURST is malware that was spread by breaching the build server for SolarWinds' Orion product. Using threat modeling it is possible to identify mitigations to reduce the risk and improve the security of the development life-cycle.

To be able to mitigate such attacks it is important to first understand the goals of the threat actors. The end goal in such an attack is not to gain access to the supplier (SolarWinds), but rather to leverage the supplier's products to access its customers' systems. Supply chain attacks can affect organizations both in the form of becoming the facilitator of malware to customers (like SolarWinds) and becoming the victim of the malware (like SolarWinds' customers).

The build environment is typically neglected by defenders, and to some extent by attackers. Not only are threats often ignored, but it is perhaps even more common for them to not be evaluated at all. A build server is not the only sensitive component in modern development, and the same neglect is often shown for most parts of the process. Security defenses have historically focused almost exclusively on the runtime environments, with the assumption of the deployed code being trustable (albeit potentially vulnerable). This is not entirely unreasonable, as traditionally that is how attacks happen, but it also leaves threat actors focusing on the supply-chain with great leeway to perform attacks.

Software producers must realize and accept that the **entire development process is sensitive, as it produces the software which is later deployed in sensitive systems**.

Depending on the assets protected by the produced software and development process, different levels of mitigations could be implemented. The focus, energy and money should ideally be spent on the most critical defenses. To help identify those defenses we must evaluate threats and assess risk. Stakeholders and decision-makers must then take security decisions. This goes for all parts of software development and will help you choose appropriate measures.

Attacks on the build environment are most often preventable, but often require work and in many cases must be weighed against productivity. By choosing appropriate mitigations/controls/countermeasures and finding your risk appetite you can most likely find areas that should be improved and perhaps identify a security level you can be comfortable with.

It is likely that both the number of supply-chain attacks and the number of implemented defenses will increase due to the publicity of this attack. The visibility is generally good, as development processes generally need improvement security-wise, but there is also a risk of misdirected prioritisations and focus on specific advanced protections and expensive tooling while leaving the front door open.

In this blog post, we briefly discuss methods for analysing risk in the build environment and discuss some general threats and mitigations for software suppliers.

Introduction

Few have missed the spectacular supply chain attack leveraged against the network infrastructure software company SolarWinds. The attack in question, or rather the malware in question has been dubbed SUNBURST (independently called “Solarigate” by Microsoft).

In short, a threat actor has managed to inject malicious code into SolarWinds software “Orion”, and thus have the malicious code run in their customers’ systems. The attack was done in a fashion where the malicious code was included in the signed packages provided by SolarWinds. A user verifying the downloaded package would assume it was an authentic update, because it was indeed delivered by SolarWinds. Attackers are assumed to have had the ability to inject code as early as October 2019, more than a year before the discovery of the backdoor by FireEye.

SolarWinds describes Orion as “*a powerful, scalable infrastructure monitoring and management platform...*”, so the potential for further escalation due to high privileges was great.

For more information on the behavior of the malware and threat actor please read Truesec´s post on the subject as well as FireEye´s detailed analysis of the malware.

During incident response and research on SUNBURST, a second malware/vulnerability dubbed SUPERNOVA was identified (potentially coming from a different threat actor). There is no indication that a supply-chain attack was used in that case. Rather, it is indicated that the API vulnerability was used to upload a web shell to vulnerable servers.

At the time of writing the exact process of how the malicious code was inserted is not publicly disclosed, but some details are known, and speculations are plentiful. What seems true is that the malicious code was inserted at the build server, prior to signing the release. This would mean that the code was likely not in the source code repository that developers saw but rather added to the build process before signing the binary (this according to SolarWinds).

Regardless of whether the code was in the repository or not, threats and potential attacks towards the build server would be in play for almost all software projects. New details may come to light, hopefully including a detailed report from SolarWinds, but until then we choose to focus on the build server and assume that the code was injected there, or in case the signing is done separately; after build but before signing the artifacts.

Looking at the steps in a typical build process, we can easily deduct that there is no single mitigation for all software development-related threats. All components have their own sets of threats and mitigations. Those sets differ for example when it comes to source code, source control, build server, and the actual running artifacts. Luckily, the methods to identify threats and mitigations can be similar.

This blog post will focus on the threats that can enable supply-chain attacks against the build environment. We shall attempt to improve the security posture of a build server by looking at potential threats and mitigation.

We Must Still Patch Our Dependencies

In this specific attack, organisations with bad patching policies seem to have been saved from SUNBURST. Those that had not updated Orion since March 2020, were saved, not due to security hygiene, but rather because of probable lack thereof.

Let us not forget the more likely scenario of "*Using Components with Known Vulnerabilities*" (#9 on OWASP Top Ten). One of the more publicized hacks prior to Sunburst was the Equifax data breach where a failure to keep up to date was the reason for the attack. This is a much more common scenario, but it does not necessarily mean that patching can be done without a thought if the dependency situation is complex. The modern IT/Enterprise world has had to learn patch urgency the hard way after years of conservative or disorganized patch policies, with breaches as a result. The software world will have the same rude awakening as the timeframe between vulnerabilities becoming public and exploitation beginning are becoming shorter and shorter.

In an upcoming blog post, we will be discussing the complexity of modern third-party dependency management for software. In the meantime, we can conclude that software must be reasonably up to date in a supported version and that the best way to reduce risk is to reduce the attack surface by removing unnecessary dependencies, using reputable sources, and isolating risks with the least privilege principles.

Reasons to Attack the Build Server

The build server is the best place to inject code if the output is a binary for customers to download, which is exactly happened to be the case with Orion. It is stealthier than attacking the developer machines, but also gives the attacker fewer options. Managing to inject code prior to signing also raises fewer alarms than if the binary was switched to an unsigned one on SolarWinds' website.

If the target would have been a web application, the actual production server would be the obvious choice to consider. But the production server or runtime environment is also more likely to be further locked down and monitored, making it harder to hide your malicious activity. Once the release is installed in the production system it is less likely that defenses will notice the malicious activity coming from the binary.

The build server is of extreme importance, but often not protected nearly as much as it should be.

An Even More Versatile Target: The Developer Machine

We typically argue that from an attacker's perspective, a developer machine is most often the most attractive target. The developer machine can often control the production system in multiple ways, for example:

- Adding malicious code or vulnerabilities to the version control data (code).
- Modifying the build/deploy environment
- Uploading build artifacts that are included in later steps
- Accessing the actual production environment. Especially in modern cloud-based solutions where the development team is also at least in part the operations team.

Beyond that, the developer machines often have high privileges and limited anti-malware/security tooling active for performance reasons. And developers often download and run tools.

In a software development shop, **the developer's computers are often as valuable or more valuable than a high-level executive's computer**.

In this case, we assume that the build server was targeted, but in many cases accessing a developer's computer gives you access to the build server and is one vector where such an attack can originate.

Identifying and Weighing Threats to the Development Process

At Truesec we often meet development teams to educate developers and analyse software systems. We typically introduce threat modeling early to provide the basis for risk analysis and mitigation selection. Threat modeling is used to identify potential threats to a system. The threats match potential vulnerabilities and those are then coupled with possible preventions and mitigations.

Even though it is common to use a distinction between full prevention and mitigation, we choose to call all security controls "mitigations" in this blog post. We do not generally expect to be able to fully prevent these threats.

Unmitigated vulnerabilities are classified as risks to the system that can then be prioritized or accepted by the organization.

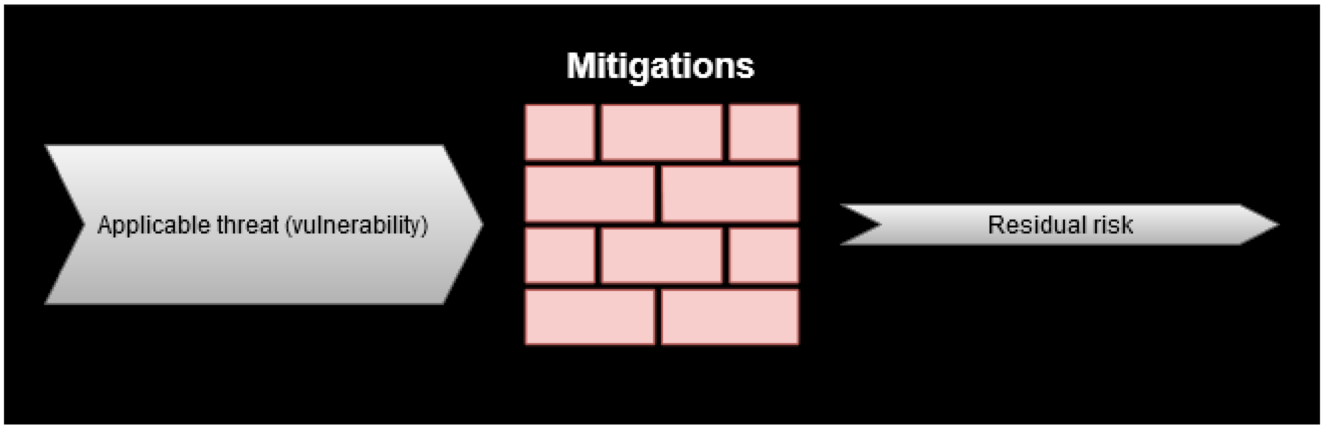

Figure 1: After mitigations are in place, residual risk it likely to remain. That risk can later be fully addressed, mitigated or accepted indefinitely
By using threat modeling we improve our chance of identifying vulnerabilities and spending time and money on the most important mitigations.

When doing threat modeling one typically starts with first visualising data flows and trust boundaries, then discovering and prioritizing threats, and finally deciding preventions and mitigations. Ideally, this process is then repeated throughout the life of the product.

If you are new to threat modeling and are interested in the process, then you can find useful resources in the "Threat Modeling Manifesto". Using threat modeling techniques in some manner helps identify threats and risks at a detailed level. The manner could be ad hoc or use for example the traditional STRIDE method or using a Cyber kill chain. The best way to threat model is to do it. There are several tools available to assist in threat modeling, but often it is best to start to draw on a whiteboard and look at tools later.

In our threat modeling introduction, we often show a typical, but very simplistic diagram of a software production dataflow:

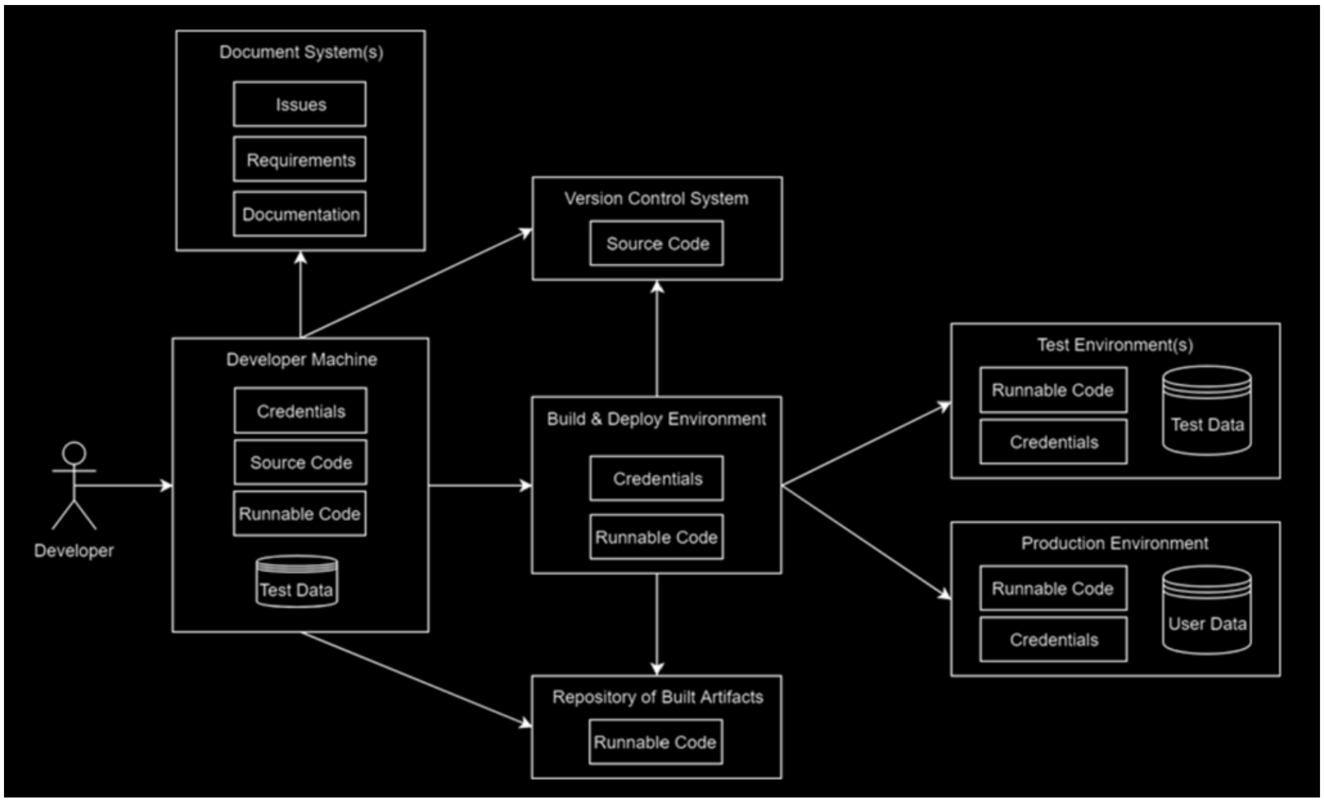

Figure 2: A very simplistic view of a software production dataflow and its assets. The build server is here called "*Build & Deploy Environment*"

There are often more data flows and actors in real life. For example, a developer and/or operator can typically access the systems for administrative purposes. As an exercise you can attempt to draw the data flows and assets in a system you are familiar with. You will likely soon find vulnerabilities to mitigate.

Using this picture as a basis we then use threat modeling to identify potential threats and mitigations for the different processes and steps. The customer can then classify unmitigated threats as risks and prioritize mitigation or accept the risk.

Why Focus on Modeling the Development Process?

The reason for this is not necessarily because attacks against the development/DevOps environment are currently the most common or likely attacks against most organizations (although they are certainly becoming more and more attractive). Rather, the reason has been that focusing on the development process and tooling is a thankful exercise to find high-risk behavior in all development shops we have encountered. There is typically a combination of risk unawareness, automation, and high privileges making it easy for even someone new to threat modeling or security to identify attack vectors. This helps participants quickly see the benefits of threat modeling while improving their own environment in the process.

Assessing the Overall Development Process Maturity

When working with customers, we often recommend a combination of overall maturity measurement and a continuous threat modeling approach. Maturity analysis gives visibility and an overall picture of security practices while being more versatile and pragmatic than compliance audits and similar models (which can still be done if applicable).

For example, the questions and descriptions from the OWASP SAMM model can be useful to assess the maturity level of your organization and indicate typical mitigations. There are other general maturity models such as BSIMM as well as specific models targeting for example embedded systems/IoT.

In OWASP SAMM the process is split into five business functions, where the practices for each is evaluated using questionnaires.

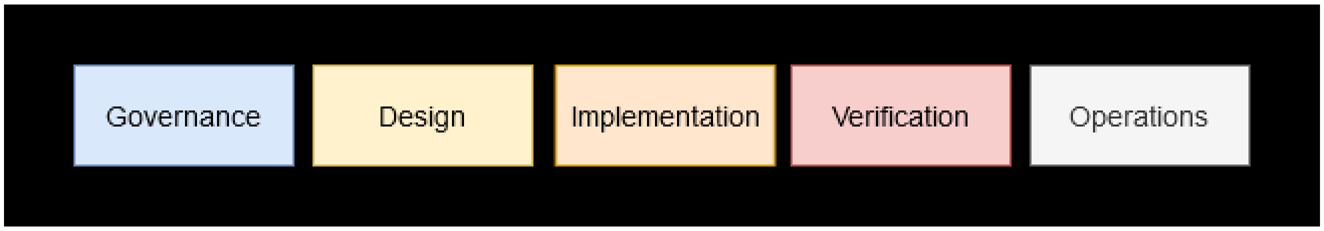

Figure 3: The OWASP SAMM business functions

Such models give an overall picture of maturity but will not lead to finding most threats and risks (nor do they aim to). One can for example note that even the highest maturity level for “Secure deployment” in OWASP SAMM is dependent on trusting a signature that most often is produced at the build server, reflecting that most realistic best practices and build systems do not support a threat model where the build server has been compromised at some level.

Threats to the Build Server

To simplify matters in this blog post, we will group some threats against the build server as four distinct threats and **mainly discuss a few mitigations for the first two**. There are more threats and mitigations, and typically they would be somewhat specific to the environment. A threat could be defined at a detailed level, for example, a specific protocol, but when doing overall analysis, it is reasonable to group threats broadly.

We would typically introduce STRIDE and similar methods for analysis, but in this blog post, we will skip the process of identifying threats and controls/mitigations and focus on a set of threats and mitigations that are known to be relevant to a build server. Typically, one would also look at for example threats against the availability of the system (DOS risk), which we purposely ignore here.

Threat 1: An unauthenticated attacker gains access to the build server to deliver malware.

In this scenario, the attacker does not initially authenticate as a user that is expected to have access to the build server. Access is gained by using a vulnerability in the build server.

Threat 2: An attacker uses a legitimate account to access the build server to deliver malware.

This is a more complex scenario than the previous threat since we obviously need some group of users to have access to configuring the build server/environment. The attacker could for example have gained this access by phishing account credentials, stealing access tokens at another resource, or by installing malware on the operator’s machine.

Threat 3 (not in focus): An insider uses their legitimate account to access the build server to deliver malware.

In this scenario, the attacker is an actual employee using a verified device. The employee could for example be disgruntled, bribed, or threatened.

Since we will not detail mitigations for this threat, we can just mention that typical mitigation would be requiring multiple individuals to approve sensitive actions, security screening of employees, education, and having a healthy company culture.

Threat 4 (not in focus): An attacker leverages a third-party component to deliver malware.

It is certainly possible that the supply chain attack against SolarWinds’s customers was due to a supply chain attack against SolarWinds using one of their suppliers. When it comes to vendor security, it’s Turtles all the way down. Third-party dependency management is its own hornets’ nest and affects much more than the build server.

Since we will not detail mitigations for this threat here, we can just mention that typical mitigation would be to keep a Software Bill-Of-Materials coupled with threat feeds and alerting, triaging, removing dependencies, and having a good update policy. We will return to this threat in an upcoming blog post, but for now, we assume that manual access to the build server by other means was the vector used.

Potential Mitigations to Threat 1 and Threat 2

There is a multitude of ways a typical build environment can be hardened. Just like in other instances, we prefer if we can model “Defence in-depth”, where multiple mitigations provide layered security. This is useful so that for example if the attacker breaks through one defense, they should be faced with another layer denying the malicious access.

Since the abovementioned threats are broad, there are many potential mitigations. Let us try to walk through ten potential mitigations and see how they affect the threats. Not all mitigations are necessarily compatible with each other and depend on the tooling and environment.

Mitigation 1: Require authentication at the build server.

This mitigation may seem obvious, but until the last few years, it was common for companies to for example run a Jenkins server without authentication. Such behavior can often be traced back to applications not enforcing secure defaults, but rather requiring an administrator to lock down servers. Typically, the organizations would argue that the server was only available on the internal network, making the risk smaller, which leads us to the next mitigation

Mitigation 2: Limit network access to the build server.

This is a primitive limitation, but IP filters and network segmentation still remove much (most) of the network noise and make attacking as well as the discovery of most systems a bit harder. If it is possible to do, it might be a good idea.

Mitigation 3: Classify the build server as a sensitive resource in monitoring.

As mentioned, the build server should be considered of the same level of sensitivity as the actual production servers and therefore receive the same high level of monitoring, alerting, and remediation as the production servers hopefully have.

Mitigation 4: Limit access rights to change build definitions and to access build secrets.

In a modern DevOps setup, very few users should be able to change build definitions at the build server level. At least if such builds have access to sensitive keys such as keys for signing and/or deployment.

It should be noted that many modern DevOps setups have the build definitions defined in the repository (typically in YAML form). This could potentially move the trust boundaries towards the code repository and makes it even more important to have a four-eyes principle/mandatory code review or signoff prior to merging sensitive branches.

In any case, it is typically possible for anyone with access to push code to also run code at the build server during the build process for pull requests. Therefore it is important that pull request builds are built and executed sandboxed environments without access to secrets. That way you can allow the productivity and quality improvement of pull request builds without a high risk of leaking production secrets and data.

Mitigation 5: Use a reputable cloud environment to build/deploy.

Using cloud-based solutions could make many mitigations simpler by trusting the provider and focusing on their compliance. For many projects that are not highly sensitive, this is a sensible choice unless the security level of inhouse operations is quite a bit higher than average. On the other hand, trust boundaries move, and we now might introduce new threats related to for example Cloud Act and international espionage. In many cases that are within the organization's risk appetite and therefore accepted.

Role-based access control and the ability to limit access to deployment tokens are typically available in cloud build environments. But it also makes network-based controls mentioned earlier harder or sometimes impossible.

It makes the build serverless obscure, which also affects the risk. While “Security through obscurity” is often misconstrued as meaning “obscurity is bad for security”, obscurity often serves an important purpose as an added defense. Obscurity should not be your only means of security, but it certainly helps to delay attacks in many cases. In the SUNBURST attack, the attackers relied heavily on obscurity and being clandestine both in terms of network behavior and code.

Mitigation 6: Add manual intervention steps before releasing/deploying.

Making the build system or process require a manual step such as a multi-party sign-off or separate signing ceremony reduces the risk of new releases being sneaked out or keys being used without knowledge. In case Continuous deployment is used this is obviously not possible, but from a threat perspective, it is quite easy to argue that full Continuous Deployment most often requires the organization to either have a very sophisticated development pipeline with regards to security control or to accept a risk that may be too big.

In the case of SUNBURST, it is likely that multiple manual intervention steps were in fact required and that this mitigation to some extent was in place.

Mitigation 7: Use *Privileged administration* requirements to access the build server.

This mitigation is connected to Mitigations 3 and 4. We typically don’t want project members to be able to access sensitive parts of a build server at all. In the case of an on-premise build server, we might not want them to be able to access the server at all unless a special circumstance happens. For those circumstances, as well as for general maintenance, we could introduce a privileged access strategy for the sensitive resource. We want to limit the actions that can be performed and monitor those actions.

One such step could be to require specific accounts, devices, or virtual machines to access the environment, making it harder for an attacker to utilize an identity, developer machine, or just network access to perform the attack.

Ideally, we want to achieve what is often somewhat confusingly called *zero trust*. This often means *limited trust* and is in fact a more dynamic *least privilege* setup where the identity of the accessor is not enough to grant access, but rather a risk-based approach takes context such as how the access happened.

Microsoft (among others) has useful resources and recommendations regarding privileged administration and privileged access strategy in their Security best practices documentation.

Mitigation 8: Sign build output and verify signatures.

If the build/release process requires a signed build it reduces the risk of incorrect binaries being deployed to the target system. It likely does not protect from replay attacks where an old and vulnerable build is reused, but nonetheless is useful to verify that in fact, the owner of the key has signed the data.

In the case of SUNBURST, the code was in fact signed. Either automatically in the build process or manually after the binaries were produced. And since the malware was included in the signed output, it is easy to conclude that the utility of signing is significantly reduced if the signer cannot guarantee that the data is authentic. Blindly signed data guarantees that the signer put their name on it, but not that they actually read what they signed. This is like how people typically handle Terms of Service agreements or cookie notices, and they are all symptoms of the same problem: *It’s just too much to verify.* This leads us to our next mitigation.

Mitigation 9: Verify that the release output matches the code repository prior to signing/releasing.

This is something that very few do, simply because it is quite hard to achieve. In this case, we want to be certain that the output of the build process is what we expect. We want to see that the build function *b(code)* leads to the expected binary, even though the build process and output data is typically highly opaque.

In other words, we require

```
b_m1(code)==b_m2(code)
```

for different independent build machines *m1* and *m2*. It could be enough that the output is not binary equal, but sufficiently similar if the differences can easily be verified to not have a significant effect (perhaps a timestamp as metadata).

To achieve this, we typically want something called Reproducible builds or Deterministic builds/compiling. The idea is that the same code and build process should render binary equal outputs. Intuitively that would be something one could expect, but compilers and build processes are often environment-dependent and often attach metadata such as agent names and timestamps to the build. Achieving reproducible builds has historically been difficult, but even in environments like Java and .net it is often possible to achieve, although it might include resetting timestamps or comparing them while allowing a time window.

If the build process is fully deterministic, we can then check out the source code at a separate machine, perform the build and compare the content or hash value of the binaries to verify bit for bit equality.

This of course requires us to trust the source code repository and the build definitions (which may be defined therein). Threats against the repository are not covered in detail here, but typically the distributed nature of modern version control such as *git* coupled with mandatory code reviews and the ability to require signed commits helps us reduce those risks.

Mitigation 10: Supply chain attestation and validation.

Now we are reaching the more futuristic solutions that aim to mitigate the risks for the entire software supply chain. This type of mitigation is something that even fewer apply but will likely become more popular in the future.

One such framework is in-toto. In-toto aims to make each step and component of the product to be verifiable, where each step of the process is signed and combined into a full verified chain. The verifiable binaries and their attestation must then be delivered in a safe manner to the receiving party which can then verify that the content is to be trusted prior to deployment.

For continuous deployment scenarios, this last-mile verification is typically achieved by combining in-toto with tools from The Update Framework (TUF). In other cases, signing the verified content before release could be a reasonable solution.

Another interesting tool aiming to help with supply chain verification is Gossamer, which is mainly designed for open-source PHP projects but has relevant ideas for other ecosystems. Gossamer uses signatures coupled with cryptographic ledgers mapping the changes to the code/project. Beyond that, there is a possibility for third party attestation of updates/changes. Gossamer does not so far have the same clearly stated goal as in-toto of verifying the development process in whole but could possibly be used for that purpose.

Mapping Threats to Mitigations

Let us map the threats to mitigations and see if the mitigation is **at least partly useful** against the threat. The mitigations will to a large extent be the same for several threats, but some specific mitigations are needed for each threat. This is of course not an entirely objective analysis and depends on the situation, but let us try to do it in a general fashion:

	Threat 1: An unauthenticated attacker gains access to the build server to deliver malware	**Threat 2:** An attacker uses a legitimate account to access the build server to deliver malware
Mitigation 1: Require authentication at the build server	**Yes**	**No**. The attacker has a valid identity
Mitigation 2: Limit network access to the build server	**Yes.** The attacker now also needs network access	**Yes.** The attacker now also needs network access

Mitigation 3: Classify the build server as a sensitive resource in monitoring	**Yes.** More likely to identify the breach	**Yes.** More likely to identify the breach
Mitigation 4: Limit access rights to change build definitions and to access build secrets	**Yes**	**Yes**. Unless a privileged account is accessed
Mitigation 5: Use a reputable cloud environment for build/deploy	**Yes**	**Yes**
Mitigation 6: Add manual intervention steps before release/deploy	**Yes**	**Yes**
Mitigation 7: Use Privileged administration requirements to access the build server	**Yes**	**Yes**. Unless a privileged account and device is accessed in a manner not detected
Mitigation 8: Sign build output and verify signatures	**No,** but useful for threats against the software delivery service	**No,** but useful for threats against the software delivery service
Mitigation 9: Verify that the release output matches the code repository prior to signing/releasing	**Yes**, if manual interventions are also in place	**Yes**, if manual interventions are also in place
Mitigation 10: Supply-chain attestation and validation	**Yes**	**Yes**

We can draw some conclusions from this:

- Even though most mitigations are useful, they rarely cover the entire threat. They sometimes need to be in combination and should often be layered.
- Covering the basics when it comes to server security and monitoring of the build server helps partly mitigate several threats
- Reducing access to the build server and build definitions greatly reduces risks. Least privilege and zero trust are as usual something to strive towards.
- Full supply chain attestation and validation is attractive and will hopefully be easier to integrate in the future

Conclusion

Have we solved DevOps security and supply chain attacks now?

The obvious answer is “no”. We have only looked at some threats and mitigations towards the build server, which is just one component in a modern development flow. Threat actors continuously change attack vectors, switching methods if for example SQL injections and phishing become harder. Large unaccounted for risks in the development process will become even more attractive for the determined and resourceful actors. The reward for them is too great to ignore.

There are many ways to threat model a development process, but the important thing is that we do it in some form. We must account for supply-chain attacks both in terms of being fed malicious code from other parties and in terms of becoming the facilitator for malicious code to customers/users. The probability for such an attack is hard to predict but it will probably increase in the near future.

The good news is that we can come a long way if we cover the basics and stick to the least privilege model and a defined process. And there are exciting tools and processes for software supply chain validation in the market already, and more to come.

There is no reason to panic for most software suppliers. Identify the risks, evaluate protections, and don't forget to focus on securing the actual application you are supplying as well.

Cybersecurity Secure Development