



SOFTWARE SUPPLY CHAIN SECURITY: A TAXONOMY OF ATTACKS AND A COMPARATIVE ANALYSIS OF PROVENANCE-BASED COUNTERMEASURES

<https://doi.org/10.5281/zenodo.23286483>

Saidrasulov Sh.N.^{1,a}, O'ktamov D.R.^{1,b}, Xamidov J.X.^{1,c}

Nuraliyev Sh.Sh.^{1,d}

¹Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan

Abstract. *This article analyses the security of the software supply chain, which has become one of the fastest-growing attack surfaces in modern information systems. A five-stage reference model of the supply chain with explicit trust boundaries is proposed, and a taxonomy of attacks is derived from it in which every class is bound to a concrete injection point: source tampering, dependency abuse, build injection, artifact substitution and delivery tampering. The verification problem is stated formally as a predicate over signed build provenance, and the resulting conditions are then used as evaluation criteria. On this basis the principal countermeasures — the software bill of materials (SPDX and CycloneDX), signed build provenance (in-toto and the SLSA Build levels), artifact signing and reproducible builds — are compared with respect to the attack classes they actually mitigate rather than the guarantees they are commonly assumed to provide. The analysis is validated against the XZ Utils backdoor (CVE-2024-3094). It is shown that a bill of materials and a valid artifact signature would not have prevented that attack, whereas a hermetic build at SLSA Build Level 3 combined with reproducible builds breaks its central mechanism — the divergence between the released source archive and the publicly auditable repository. The residual gap between declared and auditable build inputs is formalised and proposed as a practical assurance metric. The compliance of the examined mechanisms with the legislation of the Republic of Uzbekistan and with international regulation is substantiated. The results indicate that supply chain assurance should be designed as a verification chain over the build process rather than as an inventory of components.*

Keywords: *software supply chain security, software bill of materials, build provenance, SLSA, in-toto, reproducible builds, dependency confusion, open-source security, continuous integration, trust boundary.*

Annotatsiya. *Ushbu maqolada zamonaviy axborot tizimlarida eng tez kengayib borayotgan hujum yuzalaridan biriga aylangan dasturiy ta'minot yetkazib berish zanjirining xavfsizligi tahlil qilinadi. Ishonch chegaralari aniq ko'rsatilgan besh bosqichli etalon model taklif etiladi va uning asosida har bir sinfi aniq kirish nuqtasiga bog'langan*



hujumlar tasnifi ishlab chiqiladi: manbani buzish, bog‘liqliklarni suiiste‘mol qilish, yig‘ish jarayoniga kod kiritish, artefakti almashtirish va yetkazib berish kanalini buzish. Tekshirish masalasi imzolangan yig‘ish provenansi ustidagi predikat ko‘rinishida formal bayon etiladi va olingan shartlar baholash mezonlari sifatida qo‘llaniladi. Shu asosda asosiy himoya choralari — dasturiy ta‘minot tarkibi ro‘yxati (SPDX va CycloneDX), imzolangan yig‘ish provenansi (in-toto va SLSA darajalari), artefakti imzolash hamda takrorlanuvchi yig‘ish — ularga odatda nisbat beriladigan kafolatlar bo‘yicha emas, balki amalda qaysi hujum sinflarini bartaraf etishi bo‘yicha qiyoslanadi. Tahlil XZ Utils orqa eshigi (CVE-2024-3094) misolida tekshiriladi. Tarkib ro‘yxati va haqiqiy raqamli imzo ushbu hujumning oldini ola olmasligi, SLSA 3-darajadagi germetik yig‘ish takrorlanuvchi yig‘ish bilan birgalikda esa uning asosiy mexanizmini — tarqatilgan manba arxivi bilan ommaviy tekshiriladigan omborxona o‘rtasidagi tafovutni — yo‘qqa chiqarishi ko‘rsatiladi. E‘lon qilingan va tekshirish mumkin bo‘lgan yig‘ish kirishlari o‘rtasidagi qoldiq tafovut formallashtirilib, amaliy baholash ko‘rsatkichi sifatida taklif etiladi. Ko‘rib chiqilgan mexanizmlarning O‘zbekiston Respublikasi qonunchiligi va xalqaro me‘yorlarga mosligi asoslab beriladi. Natijalar yetkazib berish zanjiri ishonchliligini komponentlar ro‘yxati sifatida emas, balki yig‘ish jarayoni ustidagi tekshiruv zanjiri sifatida loyihalash zarurligini ko‘rsatadi.

Kalit so‘zlar: *dasturiy ta‘minot yetkazib berish zanjiri xavfsizligi, dasturiy ta‘minot tarkibi ro‘yxati, yig‘ish provenansi, SLSA, in-toto, takrorlanuvchi yig‘ish, bog‘liqliklar chalkashligi, ochiq kodli dasturiy ta‘minot xavfsizligi, uzluksiz integratsiya, ishonch chegarasi.*

INTRODUCTION

Modern application software is assembled rather than written. A typical project declares a few dozen direct dependencies and inherits several hundred transitive ones, each of which is fetched from a public registry, compiled by an automated build platform and delivered through a distribution channel that the consuming organisation does not control. Every link in that chain is a trust relation, and an adversary who subverts one link inherits the trust that all downstream consumers have placed in it. This is what distinguishes a supply chain attack from a conventional intrusion: the attacker does

not need to breach the target at all, and a single successful compromise propagates automatically to every organisation that installs the affected release.

The empirical scale of the problem is no longer in question. The European Union Agency for Cybersecurity reported that in two thirds of the analysed incidents the adversary targeted the supplier’s code, while the customers themselves had no realistic means of detection [16]. The compromise of the SolarWinds Orion build system in 2020 placed a signed implant in an update distributed to approximately eighteen thousand customers [17]. In 2024 a backdoor was



discovered in XZ Utils, a compression library present in the default installation of most Linux distributions; it received the maximum severity score of 10.0 and was the product of a social-engineering campaign sustained over more than two years [18, 19]. Industry telemetry shows that the volume of deliberately malicious packages published to open registries continues to grow superlinearly: in the second quarter of 2025 alone 16,279 new malicious packages were identified, bringing the cumulative total to 845,204, a 188 per cent increase year on year, with more than half of the samples designed to exfiltrate credentials and access tokens [20].

The response of standards bodies has been rapid. The United States mandated the provision of a software bill of materials for software supplied to federal agencies [13, 14]; the National Institute of Standards and Technology published the Secure Software Development Framework and the cybersecurity supply chain risk management practices [8, 9]; the Open Source Security Foundation released the first stable version of the SLSA specification for build provenance [10]; and the European Union adopted the Cyber Resilience Act, which makes component inventories and vulnerability handling a legal obligation for products with digital elements placed on its market [15].

These instruments are, however, frequently deployed as compliance artefacts rather than as security controls. An organisation generates a bill of

materials, stores it and considers the requirement discharged, without ever asking which class of attack that document is capable of stopping. The literature likewise tends to treat the available mechanisms as a cumulative set of good practices, and the question of the actual coverage of each mechanism against each attack class is seldom posed explicitly. The practical consequence is a systematic misallocation of effort: the cheapest and most auditable control is adopted first, while the controls that address the stages at which the most damaging attacks have historically occurred remain unimplemented.

The aim of this article is to establish that correspondence rigorously. The contribution is fourfold. First, a reference model of the software supply chain with explicit trust boundaries is constructed, and a taxonomy of attacks is derived from it in which each class is defined by its injection point rather than by its technique. Second, the verification problem is stated formally as a predicate over signed build provenance, which converts an informal discussion of trust into a set of checkable conditions. Third, the principal countermeasures are compared against the taxonomy using those conditions as criteria. Fourth, the resulting evaluation is validated retrospectively against the XZ Utils incident, and the residual gap between declared and auditable build inputs is formalised as an assurance metric.

Literature review and methodology. The security of package



distribution infrastructure was first examined systematically by Cappos et al., who demonstrated that the package managers of major Linux distributions were vulnerable to mirror compromise, replay and metadata manipulation even when individual packages were signed [21]. That work established the principle that remains central to the field: signing a component authenticates its author but says nothing about the integrity of the process that delivered it.

Empirical characterisation of real attacks began with Ohm et al., who assembled and analysed a dataset of malicious packages published to the npm, PyPI and RubyGems registries and showed that the overwhelming majority activate during installation and target credentials [2]. Ladisa et al. subsequently produced the most comprehensive systematisation available, an attack tree of more than one hundred vectors against open-source supply chains, validated through expert survey [1]. Zahan et al. approached the problem from the opposite direction, identifying structural weak signals in the npm ecosystem — expired maintainer domains, accounts without multi-factor authentication, packages with a single maintainer and install scripts — that predict exposure independently of any particular vector [3].

On the defensive side, in-toto introduced a general framework for attesting to the steps of a software supply chain, allowing a consumer to verify that every prescribed step was carried out by an authorised actor on the expected inputs

[4]. Sigstore addressed the practical obstacle that had limited the adoption of artifact signing, namely long-term key management, by combining ephemeral certificates bound to an identity provider with a public transparency log [5]. Lamb and Zacchiroli documented the reproducible-builds effort, which makes the build an independently falsifiable claim by requiring that identical inputs yield bit-for-bit identical outputs [22]. Okafor et al. abstracted these efforts into three secure design properties — transparency, validity and separation — and used them to compare existing solutions [6], while Enck and Williams summarised the open problems reported by thirty industry and government organisations, among which the updating of vulnerable dependencies and the trustworthiness of build infrastructure predominate [7].

Two gaps remain in this body of work. The surveys classify attacks and the engineering literature describes controls, but the two are rarely placed in a single matrix in which the coverage of each control against each attack class is stated explicitly and justified. Moreover, the evaluation of such controls is seldom tested against a documented incident in which the defender possessed some of them and was nevertheless compromised. The present work addresses both gaps.

The methodology is comparative and analytical, and proceeds in four steps. A reference model of the supply chain is constructed and its trust boundaries are made explicit. The acceptance of an



artifact by a consumer is then formalised as a conjunction of verification conditions over signed provenance; these conditions, rather than informal notions of maturity, serve as the comparison criteria. The countermeasures are evaluated against the attack taxonomy using those criteria, and the resulting matrix is validated by retrospective analysis of the XZ Utils backdoor, for which the technical mechanism is publicly documented in sufficient detail to establish which controls would and would not have been effective.

Reference model of the software supply chain. The supply chain is

modelled as a directed pipeline of five stages, shown in Figure 1. The source repository holds the revision history and the review policy. The dependency registries supply direct and transitive components. The build platform transforms source and dependencies into artifacts using a toolchain and a build definition, and holds the signing material. The artifact repository stores the produced binaries, container images and signatures. The distribution stage delivers them to consumers through mirrors, update channels and integration into downstream products.

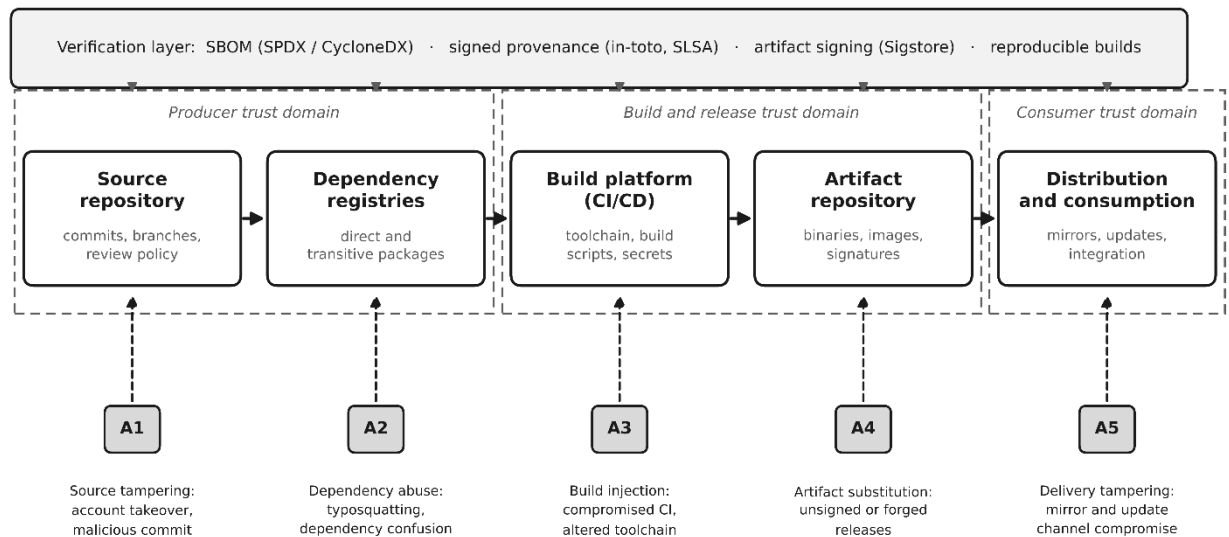


Figure 1. Reference model of the software supply chain with trust boundaries, attack injection points and the verification layer

Three trust domains are distinguished. The producer domain covers the source and the declared dependencies and is governed by the development organisation. The build and release domain covers the build platform and the artifact repository; it is frequently operated by a third party and holds the credentials that make its compromise

particularly damaging. The consumer domain covers distribution and integration and is governed by the receiving organisation. Each boundary is a point at which an assertion made in one domain must be verified in the next, and an attack is precisely a successful assertion that crosses a boundary without being verified.



Above the pipeline lies the verification layer, which does not transform artifacts but produces and checks evidence about them: component inventories, signed provenance, artifact signatures and independent rebuilds. The central argument of this article is that the effectiveness of a control is determined by which boundary its evidence actually crosses, and that this is frequently narrower than assumed.

Formal statement of the verification problem. Let a release be the tuple

$$R = (c, D, B, a, P),$$

where c is the source revision identifier, $D = \{d_1, d_2, \dots, d_n\}$ is the set of resolved direct and transitive dependencies, B is the build definition comprising the platform, the toolchain and the build parameters, a is the produced artifact and P is the provenance attestation issued by the build platform. Let $H(\cdot)$ denote a collision-resistant hash function, pk the verification key of the builder and T the set of builders the consumer is willing to trust.

The consumer accepts the artifact a if and only if the following four conditions hold simultaneously:

- C1 (authenticity of the attestation): $\text{VerifySig}(P, pk) = 1$, that is, the provenance is signed by a key the consumer can validate;
- C2 (binding to the artifact): $\text{subject}(P) = H(a)$, that is, the attestation describes exactly the artifact being installed and not another release;

- C3 (trusted builder): $\text{builder}(P) \in T$, that is, the build was executed on a platform the consumer accepts;

- C4 (completeness of inputs): $\text{materials}(P) \supseteq \{H(c)\} \cup \{H(d_i) : d_i \in D\}$, that is, every input that entered the build is declared in the provenance.

The acceptance predicate is therefore

$$\text{Accept}(a) = C1 \wedge C2 \wedge C3 \wedge C4.$$

A build is called reproducible if the mapping $\text{Build}: (c, D, B) \rightarrow a$ is deterministic, so that two independent executions on distinct infrastructure yield $a_1 = a_2$ and consequently $H(a_1) = H(a_2)$. Reproducibility is the only property in this set that an external party can falsify without any cooperation from the producer, which is what gives it a distinct role in the analysis that follows.

The predicate $\text{Accept}(a)$ establishes who built the artifact and from which inputs the builder claims to have built it. It does not establish that those inputs correspond to the publicly auditable source. Let $\text{Auditable}(c)$ denote the set of inputs that an independent reviewer can derive from the public repository at revision c . The residual gap is then defined as

$$\Delta = \text{materials}(P) \setminus \text{Auditable}(c).$$

Every element of Δ is an input that entered the build and that no external party can inspect. A supply chain is verifiable in the strong sense only when $\Delta = \emptyset$. This quantity is proposed here as a practical assurance metric: it is computable from the provenance document and the repository, it is monotone with respect to the hardening of the build, and, as the case



study below demonstrates, the XZ Utils attack resided entirely within Δ .

Taxonomy of software supply chain attacks. Attacks are classified by injection point rather than by technique, because the injection point determines which verification condition is violated and therefore which control is capable of

detecting the attack. Techniques are volatile and proliferate rapidly; injection points are fixed by the structure of the pipeline. Five classes follow directly from the reference model, and they are summarised in Table 1.

Table 1. Taxonomy of software supply chain attacks by injection point

Class	Injection point	Representative techniques	Documented incidents	Property violated
1	Source repository	maintainer account takeover; deliberately deceptive contribution; gradual acquisition of commit rights; protestware	event-stream (2018); XZ Utils (2024)	Source integrity
2	Dependency registries	typosquatting; dependency confusion; malicious update of a previously legitimate package; self-propagating worm	ua-parser-js (2021); torchtriton (2022); Shai-Hulud npm worm (2025)	Dependency authenticity
3	Build platform (CI/CD)	implant inserted during compilation; altered toolchain; poisoned build script; theft of signing material	SolarWinds SUNBURST (2020); Codecov Bash Uploader (2021)	Build integrity
4	Artifact repository	substitution of a built artifact; release signed with a legitimately held key; repository credential theft	3CX DesktopApp (2023)	Artifact authenticity
5	Distribution channel	mirror compromise; hijacked content delivery network; manipulation of update metadata	polyfill.io (2024)	Distribution integrity



Class A1 covers the introduction of malicious code into the source itself, whether through the takeover of a maintainer account, a deliberately deceptive contribution, or the gradual acquisition of commit rights by an adversary. Class A2 covers the abuse of the dependency resolution mechanism: typosquatting on a popular package name, dependency confusion in which a public package shadows an internal one, and the malicious update of a package that was previously legitimate. Class A3 covers the compromise of the build platform, in which the source is not modified at all and the implant is injected during compilation. Class A4 covers the substitution of the artifact after it has been built, and class A5 the tampering with the delivery channel through which it reaches consumers.

The classes differ sharply in detectability. An attack of class A1 or A2 is present in material that is, at least in principle, publicly readable, and therefore susceptible to review and to automated analysis. An attack of class A3 leaves no trace in the source at all: the repository and the released binary are both internally consistent, and only a comparison between them can reveal the discrepancy. This asymmetry is the reason why the mechanisms examined in the next section are not interchangeable.

Security requirements for a trustworthy supply chain. From the reference model and the acceptance predicate, the following requirements on a supply chain assurance scheme can be stated:

- completeness of the inventory: every component that is present in the delivered artifact must be identifiable, including transitive dependencies and components introduced by the build itself;
- authenticity of origin: for each artifact it must be possible to establish which platform produced it and from which source revision, and this assertion must be cryptographically bound to the artifact;
- integrity of the build process: the build must be isolated, so that neither a concurrent build nor a user-defined build step can influence the result or reach the signing material;
- external falsifiability: the claim that a given artifact corresponds to a given source must be verifiable by a party that trusts neither the producer nor the build platform;
- non-repudiation and transparency: signing events must be recorded in an append-only log so that the issuance of an unexpected signature is detectable after the fact;
- timeliness of response: when a component is subsequently found to be vulnerable, the set of affected products must be determinable without re-examining every build.

No single mechanism satisfies all six. The comparison that follows establishes which subset each of them does satisfy.

Software bill of materials. A software bill of materials is a formal, machine-readable inventory of the components contained in a delivered



artifact together with their relationships. Two formats dominate practice and both have been standardised: SPDX, originating in the Linux Foundation and published as ISO/IEC 5962:2021 [12], and CycloneDX, originating in OWASP and published as ECMA-424 [11]. Their minimum content is harmonised by the inventory of minimum elements issued by

the National Telecommunications and Information Administration [13], so that the choice between them is driven by the intended use rather than by coverage. Table 2 compares them along the dimensions that matter for supply chain assurance.

Table 2. Comparison of the SPDX and CycloneDX bill-of-materials formats

Characteristic	SPDX	CycloneDX
Standardisation	ISO/IEC 5962:2021	ECMA-424 (1st edition, 2024)
Originating organisation	Linux Foundation	OWASP
Primary design goal	licence compliance, extended to security	security and risk analysis from the outset
Serialisation formats	tag-value, JSON, YAML, RDF/XML, spreadsheet	JSON, XML, Protocol Buffers
Vulnerability information	through an external security profile	native (VEX and vulnerability disclosure reports)
Representation of relationships	general relationship graph with a rich type vocabulary	hierarchical dependency tree
Coverage beyond software	profiles for services, datasets and artificial-intelligence models	native inventories for services, machine-learning models and hardware
Typical deployment	regulatory and licence reporting	automated generation inside the build pipeline
Minimum elements of [13]	supported	supported

In terms of the requirements stated above, a bill of materials addresses completeness of the inventory and timeliness of response, and it addresses them well: when a new vulnerability is disclosed, the set of affected products is

obtained by a query rather than by an audit. It addresses none of the remaining four requirements. A bill of materials is a claim about composition that is issued by the producer and is not independently falsifiable; it describes what the producer



believes the artifact contains, which in the presence of an adversary inside the build is precisely the quantity that cannot be relied upon. This is the central limitation that the following subsection addresses.

Build provenance and attestation.

Provenance is a signed statement, emitted by the build platform, that records which artifact was produced, from which inputs and by which process. The in-toto framework generalises this to the whole chain of steps, permitting a consumer to verify that each prescribed step was performed by an authorised actor on the expected materials [4]. The SLSA specification organises the resulting guarantees into a graduated set of Build

levels that can be adopted incrementally [10]. At Build Level 1 provenance merely exists and is distributed; it documents the process and deters mistakes, but it is trivial to forge. At Build Level 2 the build runs on a hosted platform on dedicated infrastructure and the provenance is bound to that platform by a digital signature, which prevents tampering after the build. At Build Level 3 the platform additionally isolates builds from one another and keeps the signing material out of reach of user-defined build steps, which prevents tampering during the build. Table 3 maps these levels onto the attack classes of Table 1.

Table 3. Coverage of the attack classes of Table 1 by the SLSA Build levels

Attack class	Build L1	Build L2	Build L3	Complementary control required
A1 Source tampering	no	no	partial	protected branches, mandatory independent review, signed commits, maintainer vetting
A2 Dependency abuse	partial	partial	partial	bill of materials, pinned dependency hashes, registry admission policy
A3 Build injection	no	partial	yes	hermetic and isolated build, fetching source from version control
A4 Artifact substitution	no	yes	yes	artifact signing with a public transparency log
A5 Delivery tampering	no	yes	yes	signed repository metadata, verification of mirrors at the consumer

Note. “no” — the level provides no mitigation of the class; “partial” — the level restricts the attack or makes it observable but does not prevent it; “yes” — the level prevents the class provided that the provenance is verified before installation.

Two conclusions follow from Table 3. The first is that the progression from Level 1 to Level 3 is not a uniform improvement across the taxonomy: it is

concentrated almost entirely on classes A3 and A4, which are exactly the classes that a bill of materials cannot address. The second is that no Build level, including



Level 3, resolves class A1 on its own, because provenance faithfully records whatever source revision was supplied to it; if the malicious code is in that revision, the provenance will attest to it correctly. Source-level controls — protected branches, mandatory independent review and signed commits — are therefore complementary rather than optional, and the frequent presentation of SLSA as a complete solution is misleading.

Artifact signing occupies a narrower position than is generally supposed. A valid signature establishes that the holder of a given key released the artifact. It does not establish that the artifact corresponds to any particular source, and it is therefore ineffective against every attack in which the legitimate signing identity is itself used, which includes classes A1, A3 and the greater part of A4. Sigstore improves the operational properties of signing substantially by removing long-term key custody and by recording every signing event in a public transparency log [5]; the transparency log is the security-relevant contribution, since it makes an unexpected signature detectable even though it cannot prevent it.

Reproducible builds. A build is reproducible when the same source and the same declared dependencies yield a bit-for-bit identical artifact on independent infrastructure [22]. Reproducibility is qualitatively different from the mechanisms discussed above because it does not rest on any assertion by the producer. Any third party may rebuild the artifact and compare hashes; a discrepancy

is proof of divergence between the distributed binary and the published source, and it requires no cooperation and no trust. In the terminology of the formal statement, reproducibility is the only mechanism that renders Δ observable rather than merely declared.

Its cost is correspondingly higher. Achieving determinism requires the elimination of timestamps, build paths, locale-dependent ordering and parallelism-dependent artefacts from the output, which in large code bases is a sustained engineering effort rather than a configuration change. This explains the adoption asymmetry observed in practice: the mechanism with the weakest guarantee, the bill of materials, is the most widely deployed, while the mechanism with the strongest guarantee is the least deployed.

Case study: the countermeasure set against the XZ Utils backdoor. The backdoor introduced into XZ Utils, registered as CVE-2024-3094 with a severity score of 10.0, provides an unusually informative test of the framework developed above, because its mechanism is documented in detail and because the affected ecosystem already employed several of the controls under discussion [18, 19]. An adversary operating under the identity “Jia Tan” contributed to the project over a period exceeding two years, acquired the trust of the maintainer and eventually obtained release authority. Versions 5.6.0 and 5.6.1 of the library were then published containing a multi-stage payload that, on



specific x86-64 Linux systems, hooked the authentication path of OpenSSH through its indirect dependency on the library via systemd, permitting remote code execution by a party holding a particular private key.

The decisive technical detail is the placement of the trigger. The compressed payload files were present in the public repository, where they were presented as test fixtures and were inert. The build script fragment that extracted and activated them, a modified build-to-host.m4, was not present in the repository at all: it existed only in the release archives published by the adversary. The source that the public could inspect and the source that distributions actually compiled were therefore different objects. In the notation introduced above, the attack was located wholly within Δ .

Evaluating each control against this incident yields the following results.

1. A software bill of materials would have recorded the presence of xz-utils 5.6.0 in the affected products. At the time of the attack that version carried no known vulnerability, so the inventory would have produced no signal whatsoever. Its value was realised only after disclosure, when it allowed affected systems to be enumerated rapidly. The bill of materials is thus a response instrument, not a detection instrument.

2. Artifact signing was in place and functioned exactly as designed. The release archives were signed by the legitimate release key held by an actor who had genuinely been granted that

authority. Every signature verified correctly. This is the clearest available demonstration that signing authenticates an identity and not a property of the content.

3. Provenance at SLSA Build Level 1 or Level 2 would have recorded the release archive as the build input and would have been signed by a legitimate platform. Conditions C1, C2 and C3 would all have been satisfied. Condition C4 would also have been satisfied in form, because the archive was indeed declared — but the archive itself was the poisoned material, so the declaration conveyed no protection.

4. A hermetic build at SLSA Build Level 3, configured to fetch the source directly from version control at a pinned revision rather than from a maintainer-supplied archive, would have compiled the repository tree, in which the activating script was absent. The payload would have remained inert. This control breaks the attack, and it does so precisely by forcing Δ towards the empty set.

5. Reproducible builds would have produced a divergence between an independent rebuild from the repository and the binary distributed by the attacker. The discrepancy constitutes positive evidence of tampering and requires no prior knowledge of the payload.

The incident therefore separates the controls sharply. Those that attest to identity or to composition were present and were defeated without being broken, because the adversary operated as a legitimate participant. Those that bind the



artifact to an independently auditable source revision would have detected or prevented the attack. The distinction is not one of maturity or of expenditure but of the object being verified, and it supports the proposition that Δ , rather than the presence of any particular document, is the quantity that should be minimised.

Regulatory compliance. The mechanisms examined above are not merely technical options; in several jurisdictions they have become legal obligations, and their adoption in the Republic of Uzbekistan is supported by the existing legal framework. The Law on Cybersecurity assigns to the subjects of critical information infrastructure the duties of identifying threats, of taking measures to prevent them and of reporting incidents [23]; software supply chain compromise is, by the evidence presented above, among the principal mechanisms by which such infrastructure is reached, and provenance records provide precisely the evidential basis that incident reporting requires. The Law on Informatization establishes the requirements for the protection of information resources and systems irrespective of the technology employed [24], and the Law on Personal Data imposes obligations of lawful processing and of the rectification and erasure of data [25], obligations that cannot be discharged reliably when the integrity of the processing software is itself unverified.

At the level of state policy, the Strategy of Development of New Uzbekistan for 2022–2026 sets the task of

creating a system of countering cybercrime [26], and the controls analysed here are directly applicable to the software underlying national digital services. Because public information systems are predominantly assembled from open-source components obtained through the same registries discussed above, the taxonomy of Table 1 applies to them without modification.

International regulation is converging on the same instruments and has direct practical consequences for Uzbek software exporters. Executive Order 14028 of the United States made the provision of a bill of materials a condition of federal software procurement [14], with the minimum content specified separately [13]. The Cyber Resilience Act of the European Union, which entered into force on 10 December 2024, requires manufacturers of products with digital elements to identify and document the components they contain, to handle vulnerabilities throughout the support period and to report actively exploited vulnerabilities; the reporting obligations apply from 11 September 2026 and the remaining obligations from 11 December 2027 [15]. Any organisation placing software on the European market after those dates is bound by them irrespective of where the software was developed. The Secure Software Development Framework provides the corresponding engineering practices, grouped as preparing the organisation, protecting the software, producing well-secured software and responding to vulnerabilities [8], and the



supply chain risk management publication extends them to the acquisition of third-party components [9].

It should be emphasised that none of these mechanisms discharges a legal obligation automatically. A bill of materials that is generated but never queried, or provenance that is produced but never verified at installation, satisfies the letter of a requirement while providing no protection. Compliance in substance requires the organisational measures — verification policy, key management, response procedure — that convert the artefacts into controls.

Discussion. The analysis exposes a systematic mismatch between where effort is spent and where the risk lies. The bill of materials is the most widely adopted mechanism because it is inexpensive, because it can be produced after the fact and because its presence is easily audited. Yet it addresses only two of the six requirements, and, as the case study shows, it provides no detection capability against an adversary who is operating inside the trust boundary. Conversely, hermetic builds and reproducibility address the classes of attack that have caused the greatest documented damage, and they are the least adopted, because they require modification of the build infrastructure rather than the addition of a document.

A useful way to summarise the distinction is by the question each mechanism answers. The bill of materials answers what the artifact contains. Provenance answers where it came from.

Signing answers who released it. Reproducible builds answer whether anyone else, starting from the published source, would obtain the same result. Only the last of these is falsifiable by a party external to the producer, which is why it occupies a distinguished position in the framework developed here and why Δ is proposed as the quantity to be minimised.

Several limitations of this study should be stated. The evaluation is qualitative: the coverage assessments in Table 3 are derived analytically from the specifications and from the acceptance predicate, not measured experimentally. The cost of adopting Build Level 3 and reproducibility is acknowledged but not quantified. The validation rests on a single incident, chosen because its mechanism is exceptionally well documented; a broader retrospective study across the dataset of Ohm et al. [2] would strengthen the conclusions. Finally, the analysis addresses the integrity of the supply chain and not the correctness of the components themselves: an artifact may be perfectly verifiable and still contain an exploitable defect introduced in good faith.

These limitations define the direction of further work. The authors intend to measure the engineering overhead of raising a representative university software project from Build Level 1 to Build Level 3, to evaluate the applicability of the proposed metric Δ to the software stack of national digital services, and to examine the interaction between supply chain provenance and the migration to post-quantum signature algorithms, since



the attestations analysed here depend on digital signatures whose long-term validity is itself a supply chain assumption.

Conclusion. This article has analysed the security of the software supply chain and has established a correspondence between attack classes and countermeasures that is usually left implicit. A five-stage reference model with explicit trust boundaries was proposed, and a taxonomy of five attack classes was derived from it in which each class is defined by its injection point and therefore by the verification condition it violates.

The acceptance of an artifact was formalised as a conjunction of four conditions over signed build provenance, and the residual gap Δ between the inputs declared in the provenance and the inputs that an independent party can audit was introduced as a practical assurance metric. The comparison of countermeasures showed that the bill of materials satisfies the requirements of inventory completeness and response timeliness but

provides no detection capability; that the SLSA Build levels concentrate their effect on build injection and artifact substitution while leaving source tampering to complementary controls; that artifact signing authenticates an identity rather than a property of the content; and that reproducible builds alone render Δ externally observable.

The retrospective analysis of the XZ Utils backdoor confirmed this ordering under adversarial conditions: a bill of materials and a valid signature were present and were ineffective, whereas a hermetic build from version control at Build Level 3 together with reproducible builds would have broken the attack by eliminating the divergence between the released archive and the auditable repository. The practical implication for organisations, including the operators of national information systems, is that supply chain assurance should be designed as a verification chain over the build process, with the minimisation of Δ as its objective, rather than as the accumulation of inventory documents.

REFERENCES:

1. Ladisa P., Plate H., Martinez M., Barais O. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains // 2023 IEEE Symposium on Security and Privacy (SP). – IEEE, 2023. – P. 1509–1526.
2. Ohm M., Plate H., Sykosch A., Meier M. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks // Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2020). Lecture Notes in Computer Science, vol. 12223. – Cham: Springer, 2020. – P. 23–43.
3. Zahan N., Zimmermann T., Godefroid P., Murphy B., Maddila C., Williams L. What are Weak Links in the npm Supply Chain? // Proceedings of the 44th International



Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). – IEEE/ACM, 2022. – P. 331–340.

4. Torres-Arias S., Afzali H., Kuppusamy T.K., Curtmola R., Cappos J. in-toto: Providing farm-to-table guarantees for bits and bytes // Proceedings of the 28th USENIX Security Symposium. – USENIX Association, 2019. – P. 1393–1410.

5. Newman Z., Meyers J.S., Torres-Arias S. Sigstore: Software Signing for Everybody // Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS). – ACM, 2022. – P. 2353–2367.

6. Okafor C., Schorlemmer T.R., Torres-Arias S., Davis J.C. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties // Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED). – ACM, 2022. – P. 15–24.

7. Enck W., Williams L. Top Five Challenges in Software Supply Chain Security: Observations from 30 Industry and Government Organizations // IEEE Security & Privacy. – 2022. – Vol. 20, No. 2. – P. 96–100.

8. Souppaya M., Scarfone K., Dodson D. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities: NIST Special Publication 800-218. – Gaithersburg: National Institute of Standards and Technology, 2022. – 36 p.

9. Boyens J., Smith A., Bartol N., Winkler K., Holbrook A., Fallon M. Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations: NIST Special Publication 800-161 Revision 1. – Gaithersburg: National Institute of Standards and Technology, 2022. – 326 p.

10. Supply-chain Levels for Software Artifacts (SLSA), Specification version 1.0. – Open Source Security Foundation, 2023. – URL: <https://slsa.dev/spec/v1.0/>

11. ECMA-424: CycloneDX Bill of Materials Standard, 1st edition. – Geneva: Ecma International, 2024. – 38 p.

12. ISO/IEC 5962:2021. Information technology — SPDX Specification V2.2.1. – Geneva: ISO, 2021. – 192 p.

13. The Minimum Elements For a Software Bill of Materials (SBOM). – Washington: National Telecommunications and Information Administration, United States Department of Commerce, 2021. – 28 p.

14. Executive Order 14028 of May 12, 2021. Improving the Nation's Cybersecurity // Federal Register. – 2021. – Vol. 86, No. 93. – P. 26633–26647.

15. Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements (Cyber Resilience Act) // Official Journal of the European Union. – L, 2024/2847, 20.11.2024.



16. Threat Landscape for Supply Chain Attacks. – Athens: European Union Agency for Cybersecurity (ENISA), 2021. – 32 p.
17. Peisert S., Schneier B., Okhravi H., Massacci F., Benzel T., Landwehr C., Mannan M., Aspinall D., Schaumont P., Bishop M. Perspectives on the SolarWinds Incident // IEEE Security & Privacy. – 2021. – Vol. 19, No. 2. – P. 7–13.
18. CVE-2024-3094: Malicious code in the upstream xz/liblzma tarballs. – National Vulnerability Database, National Institute of Standards and Technology, 2024. – URL: <https://nvd.nist.gov/vuln/detail/CVE-2024-3094>
19. Freund A. Backdoor in upstream xz/liblzma leading to ssh server compromise: oss-security mailing list announcement, 29 March 2024. – URL: <https://www.openwall.com/lists/oss-security/2024/03/29/4>
20. Open Source Malware Index, Q2 2025. – Fulton: Sonatype Inc., 2025. – 14 p.
21. Cappos J., Samuel J., Baker S., Hartman J.H. A Look in the Mirror: Attacks on Package Managers // Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS). – ACM, 2008. – P. 565–574.
22. Lamb C., Zacchiroli S. Reproducible Builds: Increasing the Integrity of Software Supply Chains // IEEE Software. – 2022. – Vol. 39, No. 2. – P. 62–70.
23. On Cybersecurity: Law of the Republic of Uzbekistan of 15 April 2022, No. ZRU-764. – National Database of Legislation of the Republic of Uzbekistan, No. 03/22/764/0313, 16.04.2022. – URL: <https://lex.uz/ru/acts/-5960604>
24. On Informatization: Law of the Republic of Uzbekistan of 11 December 2003, No. 560-II (as amended). – URL: <https://lex.uz/ru/docs/82956>
25. On Personal Data: Law of the Republic of Uzbekistan of 2 July 2019, No. ZRU-547 (as amended). – URL: <https://lex.uz/ru/docs/4396428>
26. On the Strategy of Development of New Uzbekistan for 2022–2026: Decree of the President of the Republic of Uzbekistan of 28 January 2022, No. UP-60. – URL: <https://lex.uz/ru/docs/5841077>