Hochschule Wismar
University of Applied Sciences Technology, Business and Design
Fakultät für Ingenieurwissenschaften

Master's Thesis

Hardening the Software Supply Chain: Developing a System to Prevent Dependency Confusion Attacks in Cloud Based Continuous Integration and Deployment Processes

Submitted:	September 20, 2022
by:	Henrik Hauser
Supervisor:	Prof. Dr. Nils Gruschka
Secondary Supervisor:	Prof. Dr. Antje Raab-Düsterhöft
External Supervisor:	Sebastian Kurowski

Scope (*Aufgabenstellung*)

The scope of the thesis is to develop a system on a technical level for cloud-based Continuous Integration/Continuous Deployment (CI/CD) processes, which secures these CI/CD processes against Dependency Confusion Attack (DCA)s. This is to be validated against a real-world attack on a cloud-based CI/CD chain. The CI/CD chain is to be structured analogously to the real situation in which the system is to be deployed. The intention is to develop a system that can be realistically used to defend against DCAs and thus provides added value in terms of supply chain security. At the same time, the system should be designed in such a way that developers do not experience any losses in productivity or development ergonomics as a result of its use. In the preparation of the master thesis, the topic of Software Supply Chain Attack (SSCA), as which DCAs are to be classified, cannot realistically be fully addressed due to the sheer number of possible attack vectors.

It is posited that no generalist approach will be applied to the development of the prevention system. The objective is to narrow down the problem for the target application, analyze it, and develop the system based on the results of this analysis. Furthermore, although DCAs exist within the larger context of SSCAs, the objective of the system to be developed relates solely to the prevention of DCAs. These are to be characterized and contextualized to make it possible to develop effective preventive methods. Ultimately, the system will be based on existing technologies and prevention concepts. The aspect that will be originated in the scope of this thesis will be the prevention system in the overall target system, which, in addition to the technical functionality of the prevention system, also includes aspects of integrability and usability.

Kurzfassung

Die Software-Lieferkette ist ein wichtiger Bestandteil der modernen Softwareentwicklung und -distribution. Ähnlich “physischer” Lieferketten stellt eine Unterbrechung der Software-Lieferkette ein großes Risiko für Softwarehersteller und -verbraucher dar. Der “Dependency Confusion”-Angriff beschreibt einen Angriff auf die Software-Lieferkette über Software-Paketmanager auf der Ebene der ”Zulieferer”, wobei Softwarepackages in öffentlichen Package-Registries verwendet werden, welche sich als privat genutzte Packages ausgeben und bösartige Nutzdaten enthalten. Vor allem in automatisierten Systemen haben sich die bestehenden statischen und dynamischen Analysetechniken als ineffektiv oder ineffizient erwiesen, um diesen Angriff zu verhindern. In dieser Thesis wird ein neuartiger Ansatz zur Identifizierung und Blockierung dieser gefälschten Packages entwickelt, der auf Angriffsmerkmalen basiert, die auf der Grundlage einer Analyse des Angriffs entwickelt wurden. Der Ansatz nutzt Blockchain-Technologie, um ein System zur Katalogisierung und Überprüfung der Dateiintegritätsinformationen privater Packages bereitzustellen, um gefälschte Packages zu erkennen und zu verhindern, dass diese das Build-System kontaminieren. Das System wird für die Integration und Verwendung mit der bestehenden CI/CD-Infrastruktur evaluiert, und es wird festgestellt, dass der “Dependency Confusion”-Angriff durch das System wirksam verhindert werden kann, während eine effiziente Nachrüstung in bestehende CI/CD-Systeme möglich ist.

Abstract

The software supply chain is a critical part of modern software development and distribution. Similar to physical supply chains, disruption in the software supply chain poses major risks to software manufacturers and consumers. The “Dependency Confusion” attack describes an attack on the software supply chain via software package managers at the “supplier” level, using software packages on public package registries impersonating privately used packages and containing malicious payloads. Especially within automated CI/CD systems, existing static and dynamic analysis techniques proved to be ineffective or inefficient to prevent this attack. This thesis proposes a novel approach to identify and block these impostor packages, based on attack characteristics developed based on an analysis of the attack. The proposed approach uses blockchain technology to provide a system to catalogue and verify file integrity information of private packages to recognize impostors, and prevent these malicious packages from contaminating the build system. The proposed system is evaluated for integration and use with existing CI/CD infrastructure, and is found to be effective at preventing the “Dependency Confusion” Attack, while being efficiently retrofittable to existing CI/CD systems.

Contents

1 Introduction

1.1 Motivation

Almost every part of our daily lives in 2022 is thoroughly saturated with some degree of interaction with software. This extends from our personal interactions with all manner of appliances, or the ever present smartphones, through to our professional lives. There we interact closely with largely digital, software based processes, that shape the corporate world into a much more efficient and profitable one compared to the pre-digital age. These partly very complex processes themselves mandate similarly complex processes to manufacture the underlying software products. This has led — analogously to comparatively complex products such as automobiles — to a veritable software supply chain. Many parties are involved in the manufacturing of software, adding risk factors to the different stages of the software supply chain, which have to be managed in order to supply a safely usable end product to the consumer.

An important part of this software supply chain is the sourcing of third-party software (including libraries) to help outsource common problems and increase software development efficiency. The components distributed and consumed as such become part of the supply chain of software products. These tools and other third-party software are distributed by so-called "package management systems". Many of these rely on public package sources, where everybody can contribute. Package management systems can also be used "privately" in a software production environment, distributing vendor-specific packages. Solutions like these are especially important in systems that build and distribute software automatically, so called CI/CD environments. These circumstances introduce a number of risk factors along with the desired improvements to development speed.

These risks can be both of a financial dimension, but in case of attacks on critical infrastructure, could also potentially endanger lives. Securing the Software Supply Chain (SSC) is an absolute necessity, be it to prevent monetary damages, or to prevent damages to, in the worst case, the well-being of actual people.

1.2 Thesis Contribution

In the scope of this thesis, one of these risk factors, the so called DCA, will be analyzed, quantified and characterized. The goal is to understand this attack, which is an attack in the basic stages of a SSC, and develop a system, in the form of a software module, which can prevent these kinds of attacks in a cloud-based CI/CD environment. To develop such a system, DCA will be put in the context of the SSC, then characterized through quantitative analysis of past attacks. These characteristics will be used to provide a clear definition of the attack and also serve to develop the aforementioned system. Finally, the system will be evaluated against such an attack exhibiting these characteristics and a conclusion drawn, whether the approach presented can be considered an effective mitigation technique.

1.3 Limitations

This thesis will focus on the development of a system to defend against a specific kind of software supply chain attack, Dependency Confusion. While the general concept of the software supply chain will be explained, and a wider gamut of software supply chain attacks will be used to provide context, this thesis does not posit to provide solutions to defend against any other attack than ones using Dependency Confusion. Additionally, this thesis will, especially in the practical part, focus on a technology stack comprising mainly JavaScript/TypeScript, node.js and npm. This is done to allow a focus on a practical solution to a real problem, which would not be achievable with a more generalist approach for a broader selection of technologies, in which, considering the formal limitations of this thesis, either practical solutions or theoretical groundwork would have to be sacrificed. The technologies used were selected on the basis of familiarity and popularity, and especially on the concrete use case that spurred the work on this thesis.

2 Background

2.1 The Software Supply Chain

2.1.1 What is the Software Supply Chain?

To understand the concept of the Software Supply Chain, it is necessary to understand the concept of a supply chain in a more general sense. The Supply Chain can be defined as a “network of organizations that are involved, through upstream and downstream linkages, in the different processes and activities that produce value in the form of products and services in the hands of the ultimate consumer” [1, p. 17]. In exemplary terms, a paper manufacturer might be a part of a supply chain, which possesses upstream members in the form of logging companies and wood mills, and downstream members such as office suppliers and printing enterprises.

There is strong co-dependence exhibited in the relationships of these members of the supply chain, and the availability of the end-product is dependent on the cooperation between members. These supply chains are characterized and, importantly, separated from vertical integration within a single organization, by containing two or more legally separate entities contributing to the release of an end-product. The supply chain is broadly divisible into four distinct phases (Figure 1):

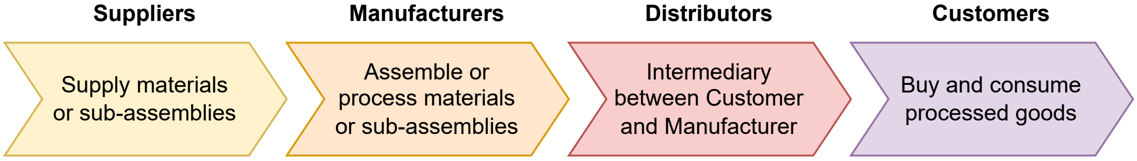

Suppliers
Supply materials or sub-assemblies
Manufacturers
Assemble or process materials or sub-assemblies
Distributors
Intermediary between Customer and Manufacturer
Customers
Buy and consume processed goods

Figure 1: The physical supply chain, from [2, p. 4]

This general view of supply chains maps almost fully into the world of software development and distribution. “Modern” Software Development is far from the cliché of a lone person coding away in a basement [3, p. 566]. Modern enterprise processes,

complex, digitized and including huge amounts of data, make complex software necessary. This kind of software is developed by large, interdisciplinary teams.
Additionally, software development itself often relies on software modules or libraries developed by third parties [4, p. 2]. A basic example would be the use of third-party compilers for code - when developing in a language that has compilers available, it would be added cost and complexity to develop one's own, compared to using a third party tool, such as GCC[1]. This extends to libraries that streamline common tasks such as database interactions, logging and monitoring, and many more. According to Sonatype's Report "2021 State of the Software Supply Chain" [5], year over year (YoY) growth in demand of open source packages (such as libraries and third party tools mentioned above) reached 73% in 2021, while supply YoY growth reached 20%. In numbers, Sonatype reports over 2.2 trillion requests for open source packages. This parallels real-life physical supply chains - there are suppliers that supply materials and sub-assemblies (e.g. Libraries and packages) that make up an integral part of the end-product. Therefore, the development pipeline of modern software can be considered as a form of supply chain. This is referred to as the "Software Supply Chain". Putting it into context with Kilger et al.'s Four Phases, the Software Supply Chain can be broken down and put into the context of physical Supply Chain as follows (Table 1):

Table 1: Comparing physical to software supply chain components

Physical	**Supply Chain Component**	**Software**
OEMs, Raw Material Producers	Suppliers	Library-, Package Developers
Product Manufacturers	Manufacturers	Product Development Teams/Systems
Resellers, Retailers	Distributors	CI/CD Systems, Digital Storefronts
Consumers	Customers	End-Users

Having established the fundamental parallels between a physical supply chain and a software supply chain, a look at the risk profile of both physical and software supply chain is needed. Specifically, the risks at the supplier level up to the manufacturer level will be examined more closely. Christopher [1, pp. 236-237] describes increasing risks to physical supply chains in reducing supplier numbers. This "single sourcing" principle, where one supplier is the sole supplier for an item crucial

[1] https://gcc.gnu.org/

to the manufacture of the end product, poses a significant risk of disruption to the supply chain as a whole. This principle is especially applicable to the software supply chain, since third-party suppliers of libraries and packages (further detailed in subsection 2.2.1) are vital to the development of complex software projects [4, p. 2]. In the following chapters, compromised "suppliers" in the form of third-party packages and libraries will be further explored to ascertain the risks they pose to manufacturers and consumers of software end-products.

The Cybersecurity and Infrastructure Security Agency (CISA) further defines a six-phase Information and Communications Technology (ICT) supply chain lifecycle [6, p. 3], which can be used to detail the typical Software Supply Chain (SSC) and risks pertaining to each of the phases in Table 2.

Table 2: ICT supply chain lifecycle, adapted from [6, p. 3]

No.	Phase	Risks
1	**Design**	Components can be designed with backdoors or malicious functions by third parties or embedded malicious actors
2	**Development and Production**	Malicious Components can be introduced in Soft- and Hardware in manufacturing, assembly and product development
3	**Distribution**	Products and Components can be modified in transport to provide malicious functions
4	**Acquisition and Deployment**	Malicious actors may insert vulnerabilities in acquisition and installation
5	**Maintenance**	Maintenance may allow malicious actors to access and modify components
6	**Disposal**	Improperly disposed items may contain sensitive data and can be accessed after disposal

SSCs are prevalent enough, and pose enough of a target, that an Executive Order by President of the United States Joe Biden in February 2021, to secure the United States' supply chains, explicitly included SSC[2]. This was followed by a publication of the CISA in April, detailing defensive measures against Software Supply Chain Attack (SSCA) [6].

[2] https://www.whitehouse.gov/briefing-room/presidential-actions/2021/02/24/executive-order-on-americas-supply-chains/

2.1.2 Risks in Modern Software Supply Chain Environments

According to Sonatype's "2021 State of the Software Supply Chain" [5], SSCAs experienced an increase of 650% in 2021 alone, an increase of 220% over 2020s 430% increase. In numbers, the global supply of open source packages in popular package managers' registries (for JavaScript, Python, Java and .NET) combined to 37,451,682 components and packages [5, p. 9]. Of the top 10% of these, 29% contained at least one known security vulnerability. An emerging trend also indicates, that these vulnerabilities are not only included due accidental inclusion of vulnerable code, but by malicious actors actively implanting vulnerabilities in these packages [5, p. 11]. This leads to benefits for the attackers, notably being able to exploit vulnerabilities not only once reported, but immediately after the affected software has been released into the wild.

These vulnerabilities are compounded by code reuse. The practice of reuse is not problematic in itself, since there are measurable benefits to not "reinventing the wheel" in most cases, similar to using standard parts such as standardized bolts in traditional supply chains [4, p. 2]. To be able to safely use these standard parts however, it must be ensured that the parts are produced to the standard they claim to follow, so that products aren't subjected to failure caused by inadequate components. Like traditional manufacturing, code manufacturers therefore have to be able to validate software components supplied to them, such that the end product is not affected in its functionality or reliability by inferior components from suppliers [7]. The MITRE ATT&CK Framework specifies "Supply Chain Compromise: Compromise Software Dependencies and Development Tools" Attacks under the ID T1195.001[3]. The MITRE ATT&CK Framework's definition is very broad, and its recommendations also lack all but very basic mitigation methods, citing Software Updates and Vulnerability Scanning.

A large amount of basic risks toward SSC have also previously been discussed by Ohm et al. in [8]. Ohm et al. discuss several attack patterns typical to supply chain attacks, and offer a quantitative analysis of known compromised packages sourced from different package management systems. This paper does not give practical advice on mitigating such attacks. The Dependency Confusion Attack (DCA) falls under the category introduced by the MITRE ATT&CK Framework, and by Ohm et al. as "Injection of Malicious Packages" [8, pp. 34/35], and will be explored further in the following chapters, first giving background on the technical framework in which they are executed, and then exploring options of mitigation and prevention.

[3] `https://attack.mitre.org/techniques/T1195/001/`

Open Source Packages

Often, packages, or modules, as briefly discussed above, are distributed on an Open Source basis. For example, *node.js* itself is an open source project[4], together with *npm*, its package manager[5]. According to Ohm et al. [8, p. 27/28], the software lifecycle of these Open Source projects is typically structured as depicted in Figure 2.

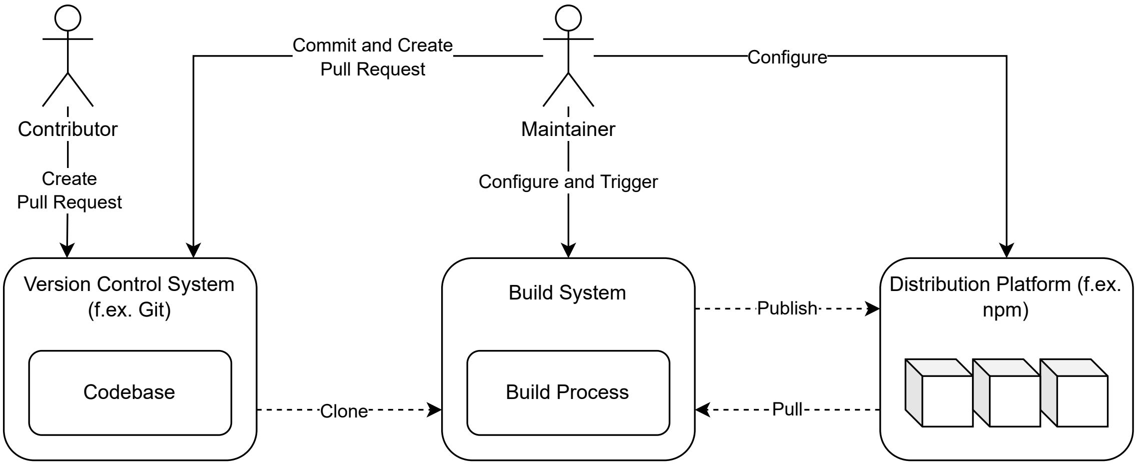


Figure 2: Open Source project development environment, adapted from [8, p. 28]

This means that there are typically two kinds of members to Open Source projects that have more or less direct influence on the codebase of the respective project. *Maintainers* have more broadly spread responsibilities, such as configuring and otherwise directly interacting with the *build system*, and thus also have control over the publication of the resulting binaries. *Maintainers* also have the power to create *Pull Requests*, which are used to alter the *codebase* of the project. The other kind of member, the *Contributor*, can only create *Pull Requests*. If a hypothetical attacker can assume any of these roles, he will have enough leverage to execute a number of attacks, either by injecting malicious code into the *codebase* and having the relevant *Pull Request* approved, or by gaining access as a *Maintainer* and manipulating the *build system*. According to Ohm et al. [8, p. 29], injecting malicious code into the *codebase* and having the *Pull Request* approved may either be accomplished by disguising the contribution as a bug fix or a useful feature [9] and getting it approved via a legitimate third-party (a *Maintainer* of the System), or by obtaining insufficiently secured credentials, such as API Keys, and approving it themselves.

[4] https://github.com/nodejs/node
[5] https://github.com/npm

The latter, as stated by Ohm et al. [8, p. 29] in reference to [10], may also be accomplished by *social engineering.*

Once the package of an open source component has been compromised, it will spread the vulnerability rapidly to consumers of said package. As seen in Table 3, package dependencies in node package manager (npm) (as an example, this functionality also exists in pip, cargo and others) are often set to be installed at the latest version. In this case, a newly malicious package would be installed the next time packages are updated, or the project is initialized (for example in a new environment). With the spread of projects dependent on open source packages, such a vulnerability would spread extremely quickly. For reference, in 2020, more than one Trillion open source packages have been requested from the npm registry, with 2021 projections predicting a 50% YoY growth to 1.5 Trillion [5, p. 8].

2.2 Technical Background

2.2.1 Package Managers

To understand the mechanisms at work, the term “package manager” also has to be clarified. Package managers in some forms have existed since about 1993 [11], initially (and still) used for managing programs in the form of packages for Linux-based systems. These package managers evolved over time to also manage “dependencies” for those programs. These dependencies are yet other programs or libraries, providing more functionalities that the programs do not need to implement themselves [12]. According to the Debian Community, “A package manager keeps track of what software is installed on your computer, and allows you to easily install new software, upgrade software to newer versions, or remove software that you previously installed. As the name suggests, package managers deal with packages: collections of files that are bundled together and can be installed and removed as a group.” [13].

These Operating System (OS)-level Package Managers led to systems based in software development environments. Today, many programming language development platforms ship with a package manager [14] (npm[6] for node.js, cargo[7] for rust, pip[8] for python, rubyGems[9] for ruby, and many more). These package managers

[6] `https://www.npmjs.com/`

[7] `https://doc.rust-lang.org/stable/cargo/`

[8] `https://pypi.org/project/pip/`

[9] `https://rubygems.org/`

serve a similar purpose to the OS-level package managers, with similar dependency-management and automated installation of needed dependencies, but instead of already compiled binaries (programs), they serve modules that can be used with the respective development environment [14]. Very common are “helper”-modules that provide convenience while programming (compare for example lodash[10]), modules that ease interfacing with other systems (compare different clients for DBMS[11][12]), or modules that provide abstractions over other services and implement boilerplate for the user. Software Development Kit (SDK)s for Cloud-Platforms usually fall within the latter category, for example the AWS-SDK[13] or the AWS-CDK[14], which provides abstractions and constructs for developing Infrastructure as Code (IaC). In 2019, the average dependency count of a repository on GitHub was 203, the average dependency count of open source project numbered 180 [15], meaning an average project in these repositories consumes functionality from 203 packages pulled from package management systems. This huge number of dependencies for single projects provides many attack vectors, especially considering that these dependencies may be transient, and as such not immediately apparent to the user or even the developer. This is especially exacerbated by the convenient handling of transient dependencies, also called *peer dependencies*, which is done automatically by the package manager as a core function (see [12]). The developer is not involved in many of these processes by default.

These package management system typically consist of the package management client, used by the developer or Continuous Integration/Continuous Deployment (CI/CD) system locally, and the package registry, where the client pulls packages from (compare Figure 3). The registry supplied as a default is usually a public registry, such as the npm registry[15], but can be configured in the package manager client to privately operated registries[16].

Versioning

The packages distributed with package managers are *versioned*, to enable the automated version management mentioned above. The versioning system in use across

[10] `https://www.npmjs.com/package/lodash`

[11] `https://www.npmjs.com/package/pg`

[12] `https://www.npmjs.com/package/mysql`

[13] `https://aws.amazon.com/sdk-for-javascript/`

[14] `https://aws.amazon.com/cdk/`

[15] `https://registry.npmjs.org`

[16] `https://docs.npmjs.com/cli/v8/using-npm/registry`

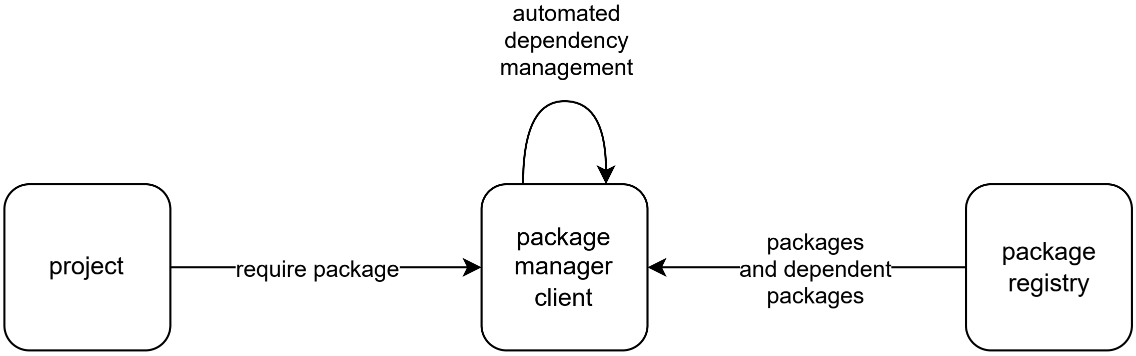

automated dependency management
project
require package
package manager client
packages and dependent packages
package registry

Figure 3: Simplified schematic of package management in software projects

most application package managers is fairly consistent, with npm[17], pip[18], cargo[19] and Bundler (to some extent)[20] all using *semantic versioning*, which is a use case independent specification for use in versioning software [16]. In *Semantic Versioning*, version numbers are divided into three numbers, separated by periods, and optionally extended for additional information about metadata and pre-release versions [16]. The first number denotes a **MAJOR** version, which can include breaking API changes, the second number a **MINOR** version, indicating additional functionality with backward compatibility, and the third number a **PATCH** version, includes such things as backwards compatible bug fixes.

2.2.2 npm

npm is the default package manager for the *node.js* development environment[21]. npm describes the package manager itself (e.g. the program on the user's machine that provides the package management functionalities described in subsection 2.2.1) as well as the associated default package registry[22]. npm provides software in the form of modules for node.js projects, or even fully featured programs that use node.js as a runtime environment[23].

npm uses the semantic versioning convention described in section 2.2.1. Dependencies in projects using npm can have their required versions declared in a number of ways[24] (Table 3):

[17] https://docs.npmjs.com/about-semantic-versioning

[18] https://py-pkgs.org/07-releasing-versioning.html

[19] https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html

[20] https://guides.rubygems.org/patterns/#pessimistic-version-constraint

[21] https://docs.npmjs.com/about-npm

[22] https://www.npmjs.com/

[23] https://docs.npmjs.com/about-npm#use-npm-to---

[24] https://docs.npmjs.com/cli/v6/using-npm/semver#advanced-range-syntax

Table 3: npm versioning syntax for dependencies

Range	Example	Description
Fixed	**1.2.3**	Specifies a fixed version of the dependency. No version other than *1.2.3* would be matched
Tilde-Range	**~1.2.3**	Specifies the most recent **PATCH** version, thus *1.2.9* would be matched, but *1.3.0* would not
Caret-Range	**^1.2.3**	Specifies the most recent **MINOR** version, matching for example *1.20.5* but not *2.0.0*
x/*-Range	**1.2.x** or **1.***	Specifies to match any part of the version not replaced by "*" or "x"
Hyphen-Range	**1.2.3-1.2.9**	Specifies any version in the declared range

This versioning system is implemented with another npm package called "SemVer"[25].
Private Registries: npm can also use package registries apart from the default one. The registry that is to be used has to implement the "CommonJS Compliant Package Registry" specification[26] [17]. This is either handled by a npm-specific implementation (such as verdaccio[27]) or a package registry that handles packages for multiple development environments, such as jFrog Artifactory[28] or AWS CodeArtifact[29]. These registries, which are administrated by private parties (for example companies wishing to distribute their own packages internally) instead of the npm team, are referred to as "Private Registries", since they are usually not reachable for non-members of the parties who own them.

npm package manifest - `package.json`

Dependencies of a project initialized with npm are stored, with other meta information, in the `package.json`, called the "manifest", file. The `package.json` file stores the following information[30]:

- **name**: The name of the project/package
- **version**: The version of the package, according to semantic versioning section 2.2.1

[25] https://docs.npmjs.com/cli/v6/using-npm/semver
[26] http://wiki.commonjs.org/wiki/Packages/Registry
[27] https://verdaccio.org/
[28] https://jfrog.com/artifactory/
[29] https://aws.amazon.com/codeartifact/
[30] https://docs.npmjs.com/creating-a-package-json-file

- **main**: The entry point to the module contained in this project/package
- **license**: The license the project/package is published under
- **dependencies**: A list of packages this project/package depends on; These will automatically be installed when this project/package is installed
- **files**: A list of files and directories that are to be included on publishing the package. Files and directories not included here will not be published
- **scripts**: A list of scripts that will be executed over the lifecycle of this package, see section 2.2.2

In addition to these fields, other information, such as author, operating system compatibility and contributor information, can be included in the package[31] The packages that are specified in the **dependency** field will be installed in the versions referenced there, with respect to possible version ranges specified according to the syntax detailed in Table 3. This installation is usually started at by executing the built-in `npm install` script at the root of the package directory. npm will then pull the respective versions of the packages from the specified package registry and place them in the `node_modules` directory, which is created at the same file system level that the `package.json` file resides at. This usually results in a directory structure similar to the one seen in Figure 4.

```
/......................................................project root
 ├─ node_modules.........................The installed dependencies
 │   └─ lodash...............................an example dependency
 ├─ index.js ............................. The entrypoint/main file
 └─ package.json.............................The package manifest
```

Figure 4: npm project directory structure

npm package lockfile - `package-lock.json`

The `package-lock.json` file exists to describe the "dependency tree" (combining the `package.json` file and the `node_modules` directory) at a single point in time, making it reproducible exactly (removing uncertainties in installing dependencies

[31] `https://docs.npmjs.com/cli/v7/configuring-npm/package-json`

as opposed to installing from `package.json`)[32]. When installing dependencies from the `package-lock.json` file, the exact versions of the dependencies that were installed when the `package-lock.json` file was generated will be installed (compare Listing 1 for an example). (Re)generation of this file occurs whenever `package.json` or `node_modules` is modified by npm[32]. In addition to the exact version number (no ranges are supported in `package-lock.json`), this file includes hashes to verify the integrity of the installed packages (compare Listing 1). The reproducibility afforded by using the `package-lock.json` file makes it ideal for use in automated build environments, where assurance to an exact replica of a known working state of the project is needed[32]. Installation of packages from `package-lock.json` is facilitated by invoking the `npm ci` command.

All these properties make the `package-lock.json` file ideal for usage in automated build environments, such as ones within *CI* Systems (see subsection 2.3.1). The exact reproduction of a “known good” environment (the developer’s) ensures stability of the build process and a functional application.

However, since package versions are locked to one specific version, this approach necessitates very detailed care of the build environment. When using one specific version, all vulnerabilities contained in this version will affect the end product. Using the lockfile as an installation source will ensure that packages match the exact version and package contents on the file system match the hashes, but will not ensure package contents aren’t malicious. Especially when packages need to be upgraded to close vulnerabilities, using the lockfile makes it necessary to monitor each build environment.

The npm lockfile also provides the URL of the tarball (archive) of the package on the registry, which can be queried directly. This URL is contained in the `resolved` field of each dependency recorded in the lockfile (compare Listing 1).

[32] https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json

```
{
    "node_modules/master-trial-package": {
        "version": "0.1.1",
        "resolved": "https://master-registry-domain-365161439830.d.codeartifact.eu-west-1.amazonaws.com:443/npm/master-registry/master-trial-package/-/master-trial-package-0.1.1.tgz",
        "integrity": "sha512-5Vc/6BHE6u3UgKW6xYFWHEhVQSK4wRAVLNDBSMx7frXQOjuEUsYZsKmhJBk5svVz0HrWjbvul2RlB/L3d9pMxQ==",
        "peerDependencies": {
          "aws-cdk-lib": "2.23.0",
          "constructs": "^10.0.0"
        }
    }
}
```

Listing 1: An Example of a Dependency entry in a lockfile

npm scopes

In addition to package names, npm packages can also have a “scope” defined. These scopes are displayed prepended to the package name and separated from it by a slash. Additionally, the scope name itself is prepended by an “@” symbol. A so-called “scoped” package will look like `@scope-name/package-name`[33]. Scopes in npm serve to group packages together, and scope names can be claimed, much like package names. Scoped and unscoped packages are compatible, such that scoped packages can depend on unscoped ones and vice versa. Since no one but the owner of the scope can claim package names within the scope, scoping or claiming a scope is a recommended way for organizations to claim their packages[33]. Scoped packages can also be associated with specific registry. This means that when a scoped package is required by a project, if the scope is associated with a private package registry, the npm client will pull the scoped packages from the specified registry, while pulling unscoped or differently scoped packages from another registry[33] (possibly the default, public one.)

Version Management

In regard to the versioning system introduced in Table 3, there are certain behaviors package registries, default to. In the case of a package being available on a

[33] https://docs.npmjs.com/cli/v8/using-npm/scope

private registry and a public registry, the packages' version numbers will be taken into consideration, and checked against the versions specified in the manifest. If the developer specified the package in his `package.json` dependency list as `internal-package@^1.3.4`, the package registry holding the highest version applicable in the "caret" range specified here (see Table 3), would be chosen to supply that version. For example, if the private registry held the package `internal-package@1.3.4`, and the public registry held the package `internal-package@1.5.0`, the package from the public registry would be chosen. This happens, if the private registry mirrors the package index from the public registry and can therefore supply a higher version than it holds internally.

The popular package management solution jFrog Artifactory[34] provided this behavior without the possibility to change it up until Artifactory version `7.16.3`. If a version for a "latest version" query was found on a public repository, that was higher than the one found on an internal one, the higher version would always be installed [18]. Since version `7.16.3`, jFrog Artifactory allows users to select "priority resolution" for package repositories, making these preferred sources and only pulling packages from there [18]. This is not, however, default behavior.

These vulnerabilities in the installation of newer versions are made more dangerous by using certain features in package managers other than npm, such as the `--extra-index-url` flag in pip. Using this, pip would, even configured for a private package registry, always install packages with higher numbers, if found on the public registry index. Birsan managed to find this used in production code at major companies, by simply searching for the string `--extra-index-url` on public repositories hosted on GitHub [19].

npm Scripts

During the lifecycle of a package, from publish through installation, to starting or stopping (in case the package represents a standalone program), npm offers the package developer a multitude of possibilities to execute scripts in user space. The lifecycle of a package is split into phases represented by keywords, for example the "install" phase, representing the stage where a package has been pulled from the registry and placed in a project structure. Scripts can be executed by npm before, during, and after these phases[35]. This is accomplished by specifying the phase name

[34] `https://jfrog.com/artifactory/`

[35] `https://docs.npmjs.com/cli/v8/using-npm/scripts`

and the script name in the package's manifest (`package.json`) under the `scripts` property, for example:

```
{
    "scripts": {
        "preinstall": "{{ executes BEFORE the `install` script }}",
        "install": "{{ run command to install package }}",
        "postinstall": "{{ executes AFTER `install` script }}"
        }
        }
```

Listing 2: scripts in npm package manifest

The scripts referenced above will be executed by npm in relation to the "install" phase of the package in whose `package.json` the scripts are referenced. These scripts will run in user space, since, as previously discussed in section 2.2.2, NodeJS does not offer a sandboxing feature, and with the level of privilege the NodeJS process itself possesses [20, pp. 996/997]. It should be noted, that npm scripts can be prevented from running by appending the command `--ignore-scripts` to the package installation command. The `--ignore-scripts` command can also be appended to the project-specific `.npmrc` file or added to the global npm configuration[36]. This will, however, prevent some packages from functioning or functioning correctly, since some tools depend on installation scripts to set up their runtime environment[35].

Publishing npm Packages

Once a package has been developed, the author can publish this package to a package registry of his choosing with the `npm publish` command. By default, this is the npm public registry (`https://registry.npmjs.org`)[37]. Depending on the environment used (for example in the case of developing in TypeScript), publishable files must be generated. The package files must be compatible with the CommonJS registry specification[38].

The files published are based on the manifest (`package.json`), where a whitelist of files can be specified under the `files` property (see also section 2.2.2) and the

[36] https://docs.npmjs.com/cli/v8/configuring-npm/npmrc

[37] https://docs.npmjs.com/cli/v8/commands/npm-publish

[38] https://wiki.commonjs.org/wiki/Packages/Registry#Packages.2FRegistry

`.npmignore` and `.gitignore` files, which include a blacklist of files not to be published. The files in the package directory are packed into a gzipped tarball[39] and both a SHA1- and a SHA512[40]-based checksum generated to enable verification of the integrity of the archive[37]. These checksums, together with the tarball, are uploaded to the specified package registry and associated with the package name and version specified in the package manifest. The package can then be consumed by users that have access to the registry.

SSC risks specific to npm

npm brings with it a number of risks to the SSC, some of which are specific to npm. Zimmermann et al. [20] for example, determined in 2019 that up to 40% of packages in the public npm registry depended on known vulnerable code, either directly or through transitive dependencies. They propose, among other reasons, that this issue is caused by heavy code reuse, which is much more prevalent in the npm ecosystem, as opposed to, for example, the Java/maven ecosystem [20, p. 996]. The risks these vulnerabilities introduce into systems is amplified by a lack of privilege separation. All processes, including dependency-own run with the same privilege as the main application [20, pp. 996/997], additionally compounded by the fact that NodeJS as the runtime itself does not offer any kind of sandboxing feature. Pfretzschner and Othmane [21] describe several methods a compromised dependency could use to attack from its node.js environment. Notably, one such method is the leakage of secrets stored in globally accessible variables [21, p. 3], for example security credentials in AWS Lambda, which are often stored in environment variables[41]. This aligns with Ohm et al.'s findings that overall, most malicious packages primary aim is data exfiltration [8, pp. 35/36].

2.2.3 Smart Contracts and Blockchain

According to the German *Federal Financial Supervisory Authority*, Blockchains can be defined as "[...] tamper-proof distributed data structures in which transactions are recorded in chronological order and mapped in an understandable and unalterable form without any centralized control" [22]. This is congruent with the definition

[39] **Tar**: **t**ape **ar**chive - A UNIX archive format, with file archives colloquially referred to as 'tarball', as in a 'ball' of files stuck together with the homonymous tar (bitumen)

[40] Only on lockfiles newer than NodeJS v8.x.x, and depending on the OS: `https://github.com/npm/npm/issues/16938`

[41] `https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html#env_encrypt`

in the white paper of the primordial, blockchain-based, cryptocurrency *Bitcoin* [23]. The transactions referred to in the quote above describe data structures that contain a recipient, a sender, (optionally) data and, with most blockchains, an amount of cryptocurrency [24, 23]. Blockchains use consensus algorithms to ensure that only valid transactions are recorded, and malicious ones are rejected [24]. These transactions and the data within are recorded in a data structure known as a *Merkle tree*. The so-called *Blocks* that make up a Blockchain contain the root of these *Merkle trees*, as well as a timestamp and a hash of the previous block, similar to a linked list [24]. These *Blocks* are shared across a network of *nodes* and are of fixed length. This results in a distributed system of *nodes*, of which every one carries all information about these transactions within them. The records of these transactions on a node are termed a “ledger” [25, p. 35], making the network of nodes all carrying ledgers a distributed ledger.

Additionally, the aforementioned consensus systems ensure that integrity (every ledger must represent the same state of the system, e.g. contain the same transactions) is kept intact across all nodes in the network [26]. The consensus system provides a way to vote on decisions, a decision taken requiring absolute majority (currently, the consensus system for both Bitcoin and Ethereum is Proof-of-Work (PoW), based on solving mathematical problems [26]). This makes the data contained within these transactions on the blockchain highly resistant to manipulation. Specifically, this means manipulating the data contained within the ledger retroactively in this consensus-based system would necessitate control of more than 50% of the nodes in the system [26], in order to gain consensus voting majority and have one’s own modified version of the ledger accepted as a correct one. In the case of Bitcoin, this would mean controlling around 8000 nodes[42] spread around the globe[43], making this a very costly endeavor.

Every user wishing to enact transactions on the blockchain must possess a key pair consisting of a public and a private key, as used for standard asymmetric encryption. The user will then use the private key to sign transactions, which are then spread throughout the network of nodes. The nodes in the network can then use the user’s public key to verify the integrity of the transaction [23]. A so-called “miner” node picks up this transaction, verifies it, and adds it to a block. Once this block has reached its capacity, it is appended to the blockchain. Once the block is appended to the blockchain, the transaction is confirmed and added to the ledger [27, pp. 3]. Miner nodes solve complex mathematical problems in order to be allowed to append

[42] Specifically meaning a computer or server running software that validates transactions and blocks

[43] Bitcoin nodes numbered 15963 on July 2nd, 2022, as checked on `bitnodes.io`

to the blockchain, expending large amounts of computing power and thus energy and resources [28]. As a reward for mining a block, so-called “cryptocurrency” may be paid out (or “minted”) by the protocol [23, 24]. This same cryptocurrency is also used by the user wishing for his transaction to be verified to pay the miner node to do so [28]. Once a block has been added to the blockchain, the network of nodes then needs to verify this new state of the blockchain. This is not to say that cryptocurrency is a necessary feature of a blockchain [29], but commonly associated with it.

Smart Contracts

The concept of “smart” (automatically enforced by hardware or software) contracts predates the emergence of blockchains by a number of years. The first definition of such a pre-blockchain smart contract comes from Nick Szabo in 1994 [30].

Starting with the cryptocurrency/blockchain known as Ethereum in 2014 [24], so-called “Smart Contract” functionality has been added to some blockchains. Smart Contracts in the blockchain ecosystem are pieces of software running on a blockchain-adjacent system that use the blockchain to enforce contractual agreements, by publishing their own code to the blockchain, to make auditing the functionality of the contract possible to everyone, and by executing on a pre-determined set of inputs from involved parties [31, p. 2]. Smart Contracts also make it possible to store data within transactions executed by the Smart Contract [31, p. 2].

This enables a smart contract to provide an interface to the blockchain with additional functionality on top of storing transactions on the blockchain for the user. A smart contract is used by addressing a transaction to the contracts’ address [32, p. 2296]. This also means, having an address on the blockchain, a smart contract has an account on the blockchain, making it a definite entity operating on the blockchain, comparable to the user actually interacting with the smart contract [33]. Once triggered, the code of the contract is executed on every node in the network, using any parameters included in the triggering transaction. Technically, any general purpose computation can be executed as part of a smart contract, but by their blockchain-based nature, their most useful application lies in managing data-driven interactions between participants of the network [34, p. 3].

An example of a smart contract at work could be as a Kickstarter-like crowdfunding agent. In this example, a blockchain called “Cryptochain” with the associated cryptocurrency “Cryptos (C$)” will be used:

Alice wants to crowdfund a project. Her funding goal is 20C$, so the C$ will only be paid out to her, if enough people transfer funds to the smart contract facilitating the crowdfunding.
The smart contract used for this example has three functions:

1. `deposit`: Transfers an amount of C$ to the Contract. Calls the `checkBalance` function after each execution.
2. `checkBalance`: Checks the amount of C$ deposited on the contract against the goal. Calls the `payOut` function if goal is met or exceeded.
3. `payOut`: Transfers the contract’s C$ to the contract issuer Alice

If Alice now deploys this contract, and enough participants call the `deposit` function, once the goal of 20C$ is met, the contract will automatically transfer the C$ to the issuer, Alice.

This functionality is auditable by every user wishing to participate, since the code is published on the blockchain, ensuring the contract isn’t secretly configured to transfer funds to Bob instead. It can also be seen, that the contract is stateful and can “own” assets/cryptocurrency on its associated blockchain.

2.3 Continuous Integration and Deployment Systems

2.3.1 What is Continuous Integration / Continuous Deployment?

CI/CD broadly describes a method, or a set of methods, that aim to automate processes in software development to more quickly deliver applications, or new versions of applications, to customers. They are often visualized as pipelines, where applications automatically traverse multiple stages, to transform code into a usable application [35, p. 2]. These pipelines combine multiple stages that serve to build, test and deliver these applications to their eventual targets. Therefore, CI/CD is

a more concrete conceptual manifestation of the software supply chain itself, automated throughout.
There are broadly three stages associated with a CI/CD pipeline:

1. Continuous Integration (CI)
2. Continuous Delivery (CD)
3. Continuous Deployment (CD)

Continuous Integration is always part of a CI/CD pipeline [35]. Usually, building a CI/CD Pipeline into an existing application development environment also starts by adding *CI*. Functionally, this step will include [35]:

1. **Sourcing** application code with changes committed into it from a specified application repository and/or feature branch
2. **Building** the application with the changes
3. **Testing** the newly built version of the application
4. **Merging** the new version of the application on successful completion of the tests into a release branch in the repository

Notably, these steps can be completed manually, but in *CI* are always automated [36, p. 33/34]. *CI* serves therefore to validate changes made to software made by a developer, by automatically building software, testing it to a set of specified criteria to ensure the changes have not broken the functions of, or the app itself, and then merging those changes on the main branch of the application to be released to *Continuous Delivery*. The result of the *CI* step will be a software version, which has, however, not yet been delivered to a place where it can be used. As described above, both the stage of *Continuous Delivery* and *Continuous Deployment* can be shortened to *"CD"*. CI/CD Pipelines can contain either both, or sometimes only *Continuous Delivery* [37, pp. 105].
Continuous Delivery depends on *CI* being built into the pipeline, which will supply validated code to the repository. *Continuous Delivery* will then involve automatically building the code into an application, running tests if applicable, and releasing a software artifact that is ready to be deployed into a production environment [37, pp. 108]. At this point, the CI/CD pipeline may be finished, and one of the definitions of CI/CD has been satisfied. Drawing the parallel to the ICT Supply Chain lifecycle (see Table 2), this would put the product in phase 2, "Development and

Production". In the analogy of the physical supply chain, at this point, the product has finished its production, and thus ended its manufacturing lifecycle, and is ready to be delivered to Distributors.

This step is accomplished in CI/CD Pipelines by *Continuous Deployment*. *Continuous Deployment* encompasses automated processes to deploy or distribute software artifacts once available [37, pp. 133]. This step relies heavily on testing in the previous ones, since no automated testing is possible once deployed through to production, and the only possible course of action from this point on are automated canaries and deployment rollbacks to previous versions. *Continuous Deployment* then makes the software artifact available to end customers, be that by deploying on their systems via an update utility or being made available in a Software as a Service (SaaS) context [37, pp. 140].

An Example of such a pipeline is shown in Figure 5.

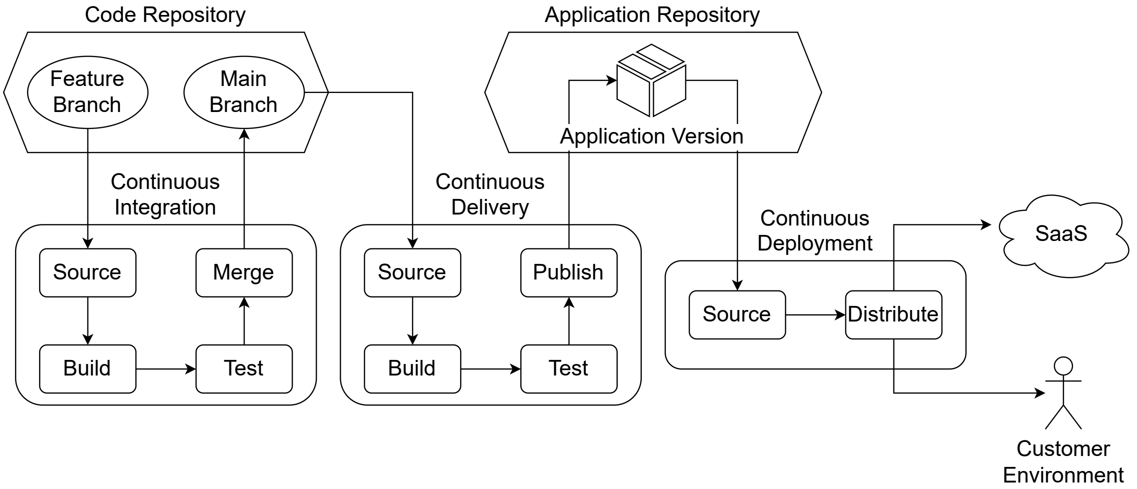


Figure 5: An example of a CI/CD pipeline incorporating three stages

2.3.2 CI/CD Systems implemented

The processes of CI/CD discussed in subsection 2.3.1 and the automation that goes along with it, do not add any value or functionality to the software supply chain as pure concepts, and as such are inextricably linked to technical implementations.

In some cases, only distinct steps of the CI/CD Pipeline may be implemented, for example a system providing only *CI* that needs to be integrated with other systems providing either or both flavor of *CD*. Many software solutions provide integration of the whole CI/CD Pipeline however, some of the more popular ones being

Jenkins[44] and TravisCI[45], both offering environments to automate CI/CD processes across many frameworks and programming languages. These products can either be implemented by the software manufacturer themselves, on self-owned server environments, providing full control over the system environment, or used as managed services, from a plethora of providers, easing integration into existing workflows and removing some or all of the effort of maintaining these systems.

The code repositories themselves can also be self-hosted (using software such as GitLab[46]), or consumed from either a self-managed cloud service (for example AWS CodeCommit[47]) or as a cloud-based SaaS solution, most popularly, for example, GitHub[48]. As of the StackOverflow Developer Survey 2021 [38], the most popular Version Control Software underlying all the above-mentioned services and technologies, is Git[49], being used by 93.43% and 94.41% of all and professional developers respectively.

2.3.3 The Build Step (CI/CD)

The Build Step is central to the *Continuous Integration* and *Continuous Delivery* phases [39, pp. 8]. In the Build Step, code for the application is compiled or otherwise packaged into deployable artifacts. In the case of a compiled program, this includes the compilation itself, as well as the linking of object files and libraries [39, pp. 8]. This is often accomplished by means of a *Makefile* or a *buildspec.* This file specifies the specific steps that are to be taken by the build system to build a software artifact out of the code provided.

In some build Systems, for example AWS CodeBuild[50] or GitHub Actions[51], the Build Step can be supplied with a list of console commands that are necessary to build the software. In these cloud based build environments, tools that are needed to build software, such as for example the Angular CLI[52], which is used to build Angular applications, will not be pre-installed, and after its installation might not be available on subsequent executions, since the underlying virtual machines

[44] `https://www.jenkins.io/`

[45] `https://www.travis-ci.com/`

[46] `https://about.gitlab.com/`

[47] `https://aws.amazon.com/codecommit/`

[48] `https://www.github.com/`

[49] `https://git-scm.com/`

[50] `https://aws.amazon.com/codebuild/`

[51] `https://github.com/features/actions`

[52] `https://angular.io/cli`

are provisioned on-demand and possess only ephemeral storage[53]. This makes it necessary to include installation of these dependencies in every execution of the build step.

In the case of node.js programs, it is also possible to include the build command as a script in the `package.json` file (see section 2.2.2). This is common practice in applications implementing frontend frameworks, such as react[54], Angular[55] or Vue.js[56].

The Build Step also includes pulling any needed dependencies from package registries [39, p. 8]. In the case of a project built with npm as part of the toolchain, the dependencies will need to be installed prior to building the application. The Build System will do this, depending on how it is instructed in the build specification mentioned above, either by executing the `npm i` command, installing dependencies by using the list in the `package.json` file, or, preferably, using the `npm ci` command to install packages using the lockfile (see section 2.2.2), ensuring a consistent state for the build with the developers'.

In case a private registry is to be used to provide packages for the Build, the build system needs to be configured such that it is allowed to access the private repository, and such that will also pull packages from there reliably. In this case, the packages stored on the private repository are often also published to there by their own build systems. Private package registries are often used with large numbers of dependent CI/CD systems to provide private packages, and to also provide better control over versions of publically available packages being used.

2.4 What is a Dependency Confusion Attack?

The Term ”Dependency Confusion Attack” was coined by Alex Birsan in February 2021 [19]. The focus of his research were attacks on/with malicious packages in public package registries (npm, pip, rubyGems) [19]. The goal of the attack (which was very well achieved, infiltrating, among others, Apple and Microsoft) was to inject malicious code into the production environments of the victims via malicious software packages (compare Figure 6).

The injection itself was achieved by naming the packages like internally used packages at the targeted companies. Additionally, the packages were published in a large

[53] See CodeBuild documentation: `https://docs.aws.amazon.com/codebuild/latest/userguide/build-env-ref-compute-types.html`

[54] `https://create-react-app.dev/docs/production-build/`

[55] `https://angular.io/cli/build`

[56] `https://cli.vuejs.org/guide/deployment.html`

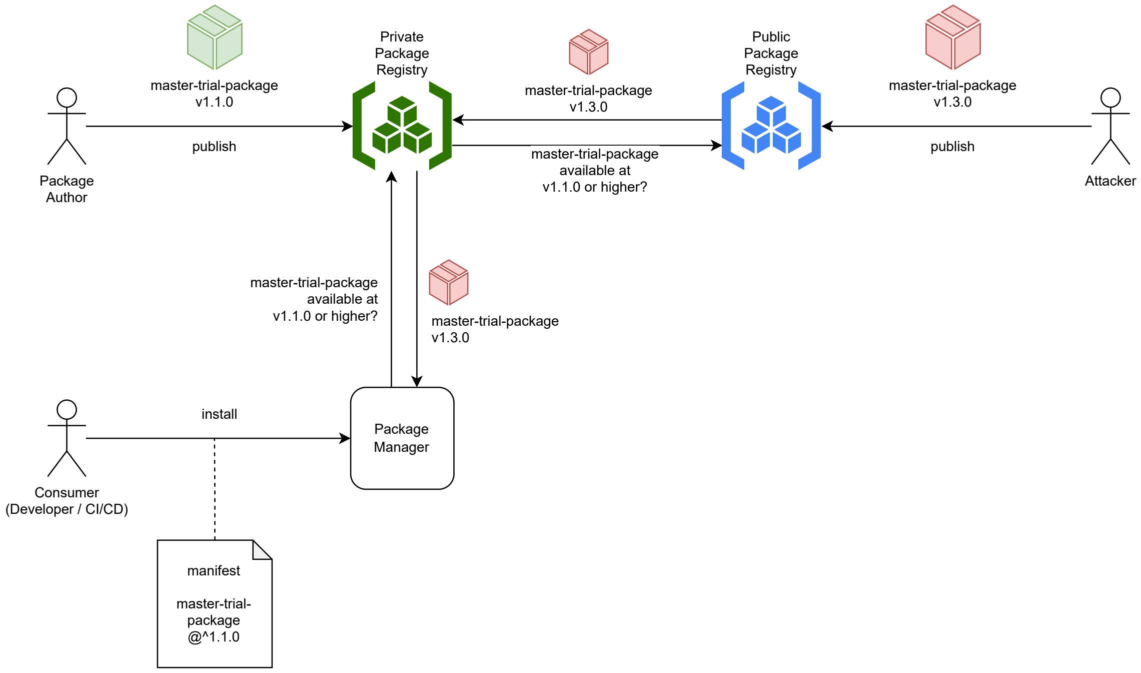


Figure 6: Schematic of a DCA in action

range of version numbers. Package Management Systems, when confronted with the choice of packages with often higher numbers available from outside sources, installed the malicious packages published by the researcher, that provided higher version numbers than the original, internal, packages.
The package names were obtained by scanning public code repositories and comparing the packages used to packages that are available on public package registries; The packages that are left over after subtracting the publically available ones from the whole are privately hosted and distributed. This disclosure of internal package names can happen unintentionally in automated build processes, where private package names can be easily embedded in automatically generated and publically released files. Compare Figure 7: If Pr is a list of packages sourced by OSINT methods from public repositories, subtracting Pu from it will reveal a subset of packages that are not publically known (blue).

These package names are then claimed on public package registries, with many versions with high version numbers published. The package contents that are used to claim these then contain malicious code. npm especially gives package developers the opportunity to execute arbitrary scripts after installing the package via the “postInstall” step in the package delivery process (see section 2.2.2). The pack-

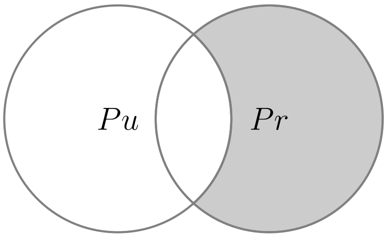


Figure 7: Comparing set of public packages to OSINT sourced set of private package list

age possesses all execution privileges of the respective node.js process providing the runtime environment (see section 2.2.2). Even in well secured networks and development environments, there is usually still the possibility of DNS exfiltration of sensitive data, such as information on network topology or credentials stored on the developer machine itself. Exfiltration of data over DNS requests, while data throughput is low, has been a popular way of exfiltrating sensitive, lightweight, data, such as the aforementioned credentials or private keys, since DNS requests are very rarely blocked from leaving even well secured networks [40, p. 4].

2.4.1 Typo- and Namesquatting

Name- and in a wider sense Typosquatting is an attack pattern that was used as one part of the two-pronged attack described above. **Typosquatting** is a well known attack on dependencies downloaded from package management solutions. One way this is achieved is to approximate the feigned packages name, but introducing spelling mistakes (the eponymous "typo") into it [41, p. 3]. An example of this is a typosquatting campaign started against users of npm in October 2019 [42, p. 4]. In this campaign, 25 variations of the `js-sha3` package were published to the npm public package registry, with variations in name such as `js-shq3` and `js-rha3` (for the full list see [42]), intended to exploit typing mistakes or misremembered package names. All the 25 impostor packages were found to be malicious [42, p. 4]. To prevent this, Taylor et al. [42] propose a program "SpellBound", which is triggered at the package installation step of the npm package lifecycle. Their algorithm examines packages with similar names to the one to be installed, and compares their "popularity score", a metric based on download numbers in a specific timeframe. The expected results are, that similarly named packages, where one is the original and one is the impostor, will have vastly different popularity scores, and the impostor can therefore be flagged as such.

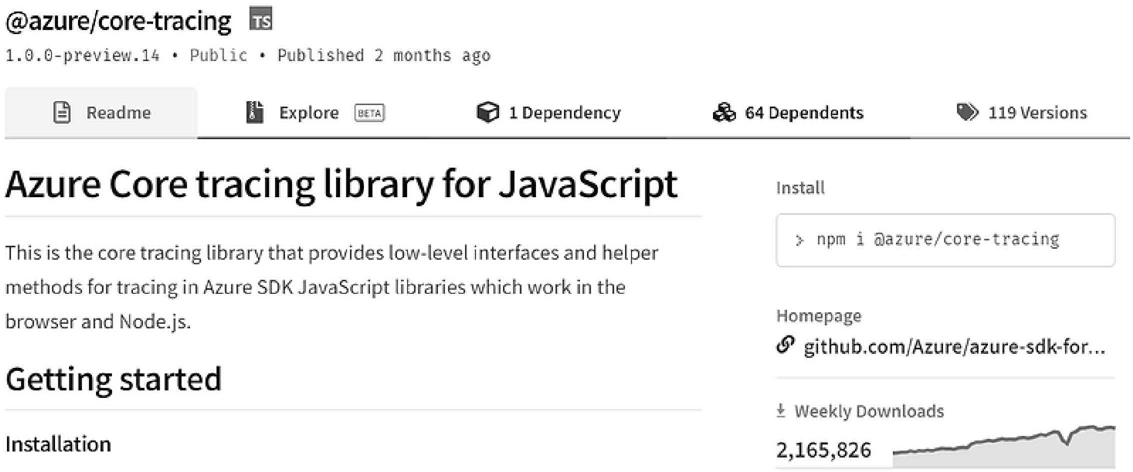
Another way of executing typosquatting attacks is impersonating packages that are normally distributed under a scope. This attack pattern is specific to npm (see section 2.2.2), which uses scopes. Real life examples of this attack pattern can be found in the attack on Microsoft Azure developers in March 2022 [43]. In this case, the attacker(s) claimed package names for their malicious packages that exactly resembled their legitimate counterparts, but lacked the ”@azure” scope. One example is the malicious impersonator `core-tracing`, pretending to be the legitimate `@azure/core-tracing` package. All in all, 218+ Packages have been affected by this attack [43]. This attack also used DNS exfiltration (and HTTP POST requests) to exfiltrate user data.

Namesquatting describes a technique associated with typosquatting, where instead of using misspellings or different spellings in package names, a package that is not present on a package registry has its name claimed by an impostor package (the “squatter”). Namesquatting is one of the core injection vectors of a DCA, and requires prior knowledge about packages that exist on private, but not on public package registries. This approach is unique to the DCA, as a central feature of this attack is to impersonate packages that are not available on public registries, versus typosquatting, which impersonates packages that are already available, but varies the malicious packages’ names [42].

2.4.2 Version Spoofing

Version Spoofing describes the second attack vector in addition to Namesquatting in a DCA. The combination of Namesquatting and Version Spoofing qualifies a ”complete” DCA. With Version Spoofing, the core concept is to publish a malicious package that already in some way name squats an existing package. This malicious package then will have its version number increased, such that it is much higher than the version number of the package that is being impersonated (see also section 2.2.2). This behavior has been central to Birsan’s Proof of Concept (PoC) Attack in 2021 [19], and has been observed in a number of attacks involving name- and typosquatting packages since, notably the very successful attack on Microsoft Azure Developers in March 2022 [43].

Version Spoofing involves exploiting developers either not locking their dependencies’ versions, or, alternatively, software used in private registries automatically trying to get the latest version from a public registry, if available. To accomplish this, the version numbers, in the case of an attack on npm packages versioned using semantic versioning section 2.2.1, several components of the version number are increased substantially over the original packages' version number. Primarily, and as seen in the Azure Developer Attack 2022 [43] (this attack has been chosen as an example, since affected packages could be found to document), the MAJOR part of the version number is set very high - in the case of one of the affected packages, setting it to `99.10.9` for the malicious `core-tracing` package (see Figure 9), impersonating the `@azure/core-tracing` package, which is only available in version `1.0.0` at the time of writing and version `1.0.0-preview.14` at the time of the attack (see Figure 8).



Azure Core tracing library for JavaScript

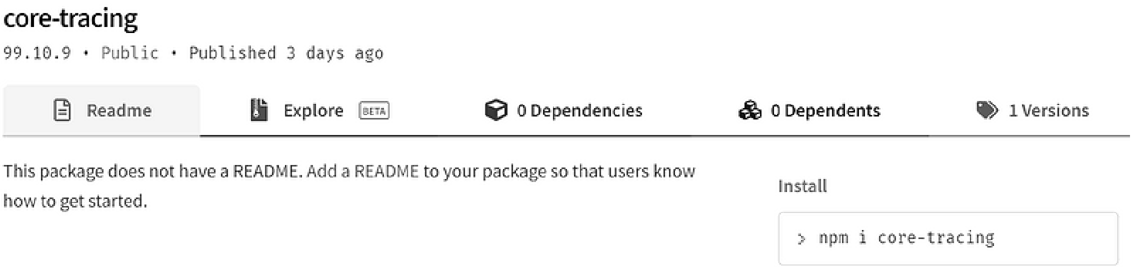
This is the core tracing library that provides low-level interfaces and helper methods for tracing in Azure SDK JavaScript libraries which work in the browser and Node.js.

Getting started

Installation

```
> npm i @azure/core-tracing
```

Figure 8: The original `@azure/core-tracing` package from the Azure Developer Attack 2022 [43]



This package does not have a README. Add a README to your package so that users know how to get started.

```
> npm i core-tracing
```

Figure 9: The impostor `core-tracing` package from the Azure Developer Attack 2022 [43]

In this specific case, the very high version number does not affect the installation, since it already relies on the Typosquatting component of the attack to infiltrate the victim's machine, since the original package is available under the `@azure` scope publically. This attack does also not represent a full DCA, since packages were not

squatted by exact name, but by variations on their original name.
Similarly, the attacker could publish Packages in versions targeting other version specifiers (see Table 3). This way, even if npm is not prompted to install the latest **MAJOR** release, specifications to install **MINOR** or **PATCH** releases can be targeted by the attack, since it is trivial to upload packages specifying arbitrary versions of themselves to npm without changing other contents of the packages.

2.4.3 Dependency Confusion Attacks in Software Supply Chains

Since the PoC of DCA has been published by Birsan in February 2021 [8], there have been a number of copycat attacks, containing actual malicious code. As previously mentioned, in March 2022, Microsoft Azure Developers were the target of an attack, where packages relating to development on the Microsoft Azure Cloud Platform were impersonated. In this attack, more than 218 packages were published that impersonated legitimate packages in the `@azure` scope, exfiltrating data from the affected machines by means of DNS Exfiltration and HTTP Requests [43]. The direct aftermath of Birsan's disclosure of the attack pattern were a number of copycat packages, directly imitating Birsan's approach being published on npm. These packages, numbering around 275, directly copied the approach taken by the initial report, but appeared to not all be malicious in intent [44].
In March 2021 a rise in packages copying the attack was again reported, this time numbering over 700 malicious packages. Additionally, the first variations in attack payload were discovered, in one case exfiltrating the bash history, possibly leaking any credentials entered over the command line, and in one case, opening a reverse shell on the victim's system [45].
Also in March 2021, Sonatype reported a "vigilante actor" flooding the Python Package Index (PyPI) and npm with over 5000 packages using the DCA pattern. These packages contained only code making simple HTTP GET Requests to a Tokyo-based IP address and contained a disclaimer warning about supply chain risks [46]. Up until now, beyond proof of concept and copycats, no damage beyond shock from affected companies has been reported. Since these attacks continue to be launched and continue to effectively infiltrate victims, and considering the runtime privileges especially npm packages are awarded (see section 2.2.2), a successful attack with severe consequences is to be expected, especially considering the far more sophisticated attack by Code White GmbH [47]. A more detailed overview over real-world DCAs, including attack vectors of the malicious payload, is given in section 3.1.

2.4.4 Significance in Cloud Contexts

The public cloud, among which popular hyperscaling cloud providers (AWS, Azure, GCP etc.) are situated, offers many benefits over traditional, on-premise server infrastructure. Among these are severely reduced procurement and maintenance cost for physical machinery and other aspects of capital and operational expenditure, fast and flexible scaling of resources according to demand and potentially huge amounts of computing power for comparatively low prices [48, p. 531].
Due to the nature of these services, in a typical public-cloud context, all resources are only reachable through public and potentially insecure channels - namely, the internet [49, p. 120]. Even considering the numerous technologies on offer to make these connections and accessibility secure, the hardware and the network are ultimately not within the range of influence of the respective customer. This means, among more general considerations about information security, ingress and egress points of information must be considered [50, p. 362]. This holds especially true in the world of Software Build- and Deployment cycles. As previously discussed in subsection 2.1.1, most current software includes other pieces of software, usually in the form of software libraries or packages managed by package managers. This of course also concerns CI/CD systems running on centralized, customer-owned infrastructure, but must be especially considered in a public cloud context, since here, these systems can't simply have their network connections blocked, lest the customer itself wouldn't even be able to access them, and additionally are usually configured to pull packages indiscriminately from public repositories[57]. The inherent reachability of these cloud-based CI/CD systems through the internet, as well as the usually unsafe default settings in regard to package sources make these systems a prime target for DCAs.

2.5 Preventive Methods

Following the discussion of the background and context of DCA in the previous section, this section will outline some methods that are suggested in literature to prevent certain vulnerabilities pertaining to DCA. These methods will be synthesized into requirements for a preventive system in chapter 4, together with characteristics

[57] Compare, for example, jFrog Artifactory: `https://www.jfrog.com/confluence/display/JFROG/Remote+Repositories`

of DCA researched in chapter chapter 3. The methods discussed will be further divided into methods for Detection, Organizational and Operative Methods, Technical Methods and associated Best Practices.

2.5.1 Detection

Detecting Malicious Packages

The detection of malicious packages has been the subject of many research projects. Malicious packages are a problem even outside DCAs, thus there is a broad range of proposed methods of detection, as well as qualitative and quantitative analysis.
Ohm et al. [8] analyzed a large set (174 entities analyzed, 419 identified) of known malicious packages in 2020, of which 109 were npm packages (about 63%, others from PyPI and RubyGems). They were able to determine, that most (56%) malicious packages would trigger their malicious routines on installation [8, p. 33], a behavior mostly found in npm and PyPI packages, which can both trigger arbitrary scripts on installation (`setup.py` for PyPI and see section 2.2.2 for npm).
They also find that 41% of affected packages check for a condition to be met before executing, but this behavior is almost exclusive to malicious packages from npm [8, p. 34]. An important finding of the study is that most malicious packages, 61%, use typosquatting (see subsection 2.4.1) as their infiltration vector [8, p. 34], finding linguistic traps such as British English differences to American English ("color" to "colour" is specifically mentioned) to be used.
Second to being injected by means of typosquatting, Ohm et al. determine that infection of an existing package is used to infiltrate victims' environments [8, p. 35]. On these packages, traces of how the infection was accomplished is often removed, however the risk profile is described in more detail in section 2.1.2.
The objective of these malicious packages was determined to be data exfiltration in more than half of cases observed, followed by "dropping", or download, of a secondary payload [8, p. 37]. Moreover, around half of these packages used obfuscating methods to disguise their true intentions [8, p. 39], and are agnostic to OSs. One important discovery was that there is significant *clustering* to be observed, where a total of 90% of all 174 packages observed belong to a cluster of similar code patterns, which on average comprises 7.3 packages [8, p. 37].
Following these insights, detecting malicious packages introduced by a DCA might be possible via:

1. Detecting script execution side effects on installation; or direct mitigation by disabling scripts
2. Detecting “dropping” by payloads that sideload other malicious dependencies
3. Detecting obfuscation
4. Detecting clusters of suspicious code from a catalog of commonly reused malicious code

In another paper, Ohm et al. propose detecting these malicious packages via the forensic artifacts they introduce into the system environment [51]. Specifically, they observed a significant increase of observables in once-benign packages that exhibited malicious behavior after being infected [51, p. 4]. “Observables” here being forensic artifacts according to the specification of STIX Cyber Observable Objects [52], including forensic artifacts deduced from system calls being made by the suspicious package while installing and running. Based on this observation, they propose an additional step to the *CI* setup in CI/CD environments to be equipped to detect these packages, which in turn will feed the whole build process to Cuckoo[58], a tool for dynamic analysis for file behavior, to detect suspicious observables as part of the build process [51, pp. 5/6]. The results of this analysis are then fed back to the developer, and it is up to him to decide if the process should proceed as normal, or if packages are infected [51, p. 6].
Ohm et al.'s proposal from [51] hence introduces another tool to detect malicious packages after they have been introduced into the system:

5. Dynamic Analysis: Detection of malicious packages based on installation and runtime behavior in a sandbox environment

While the scope of the analysis was limited, they posit that generating warnings based on the number of generated observables is a valid way of identifying malicious packages, especially if benign versions and their number of generated observables are known [51, p. 6].
In yet another report, Ohm et al. advocate for the use of signatures to detect these malicious packages [53]. In a way, this is to be seen as a direct extension of the technique proposed in [8] to use clusters of similar code to detect commonly reused malicious code. The important difference is, that in the original proposal ([8]), expert knowledge and manual labor was used to cluster code. In their proposition in [53], this clustering is automated using Abstract Syntax Trees (ASTs) [53, p.

[58] `https://cuckoosandbox.org/`

7/8] for source code comparison between packages and a Markov Cluster Algorithm (MCL) to identify said clusters [53, p. 8/9]. They report an $F^1 = 0.99$[59], which made them able to identify seven previously unreported packages on the npm registry that contained malicious code [53, p. 18].
This proposal amends one tool out of the ones already identified previously:

- Automatically detecting clusters of suspicious code by using code signatures generated automatically

With these tools in the toolbox, it is possible to separate them into categories, based on where in the development process or package lifecycle the detection method is applicable, presented in Table 4.

Table 4: Methods of detecting malicious packages

Static Analysis	**Dynamic Analysis**
Detecting obfuscation	Detecting script execution side effects
Detecting clusters of suspicious code (automatically)	Detecting suspicious observables at runtime
Detecting “dropping”	Detecting “dropping”

It should be noted, that detection of “dropping” has been both placed into the categories of static and dynamic analysis, as this could either be accomplished by detection of code signatures typical to “dropping” or at runtime by detecting the downloading of the payloads requested by this code.

Finally, in the Detection section of the MITRE ATT&CK Framework Site For ID T1195.001[60], the recommendations include verification of distributed binaries through hash checking (or “other integrity checking mechanisms”), which could be categorized as a Static Analysis technique. However, this necessitates having a hash of the artifact, and a trusted way to acquire this hash, from a known artifact to check against. This technique will also fit nicely with npms lockfile, since integrity information is stored within them (see section 2.2.2). Checking the hash is a very computationally inexpensive operation, especially when compared to dynamic analysis, and does not require knowledge about clusters of malicious code, as required in some discussed methods of static analysis.

[59] This score combines metrics for precision and recall, e.g. how few false positives and how many true positives compared to a known number could be retrieved

[60] `https://attack.mitre.org/techniques/T1195/001/`

Detecting Dependency Confusion Attack in Progress

Detecting a DCA in progress relies on knowing the patterns of attack vector exploitation of packages that have been compromised with malicious code. Past Attacks have followed very closely the patterns presented in Birsan's original proposal/disclosure of the attack [19]. This includes minimal damage to the infrastructure in which the malicious code has been deployed. The main goal of the original PoC and the following copycats has been, so far, to exfiltrate data from the victims' environments (only Code White GmbH's attack [47] did theoretically include more functionality, but in practice also restricted itself to data exfiltration). This Data exfiltration has been accomplished in the original PoC via DNS exfiltration [19]. This technique has been copied in almost every attack following this pattern so far [43, 44, 45]. In some attacks, the DNS exfiltration has been accompanied by (attempted) data exfiltration over HTTP [46, 43]. A detailed analysis of a number of attacks, especially concerning modes of attack, is provided in section 3.1.

A detection of a DCA therefore, currently, is most likely to be possible by detecting data exfiltration. The two vectors used for this are:

1. DNS exfiltration, using DNS Queries
2. HTTP exfiltration, using HTTP Calls

DNS Exfiltration: DNS exfiltration uses Domain Name System (DNS) Queries to exfiltrate Data. DNS is commonly used to translate Domain Names (e.g. "example.com") into IP addresses (e.g. "93.184.216.34"). To accomplish this, the client trying to access a server, knowing the Domain Name, but not the necessary IP address, sends the Domain Name to a public DNS Server (for example google's "8.8.8.8"). If the IP address is not found there, the request gets forwarded to the Top Level Domain (TLD)'s Name Server, in this case .com's NS. If found at any of these stages, the client obtains the IP and can connect to the Server, if not, the IP lookup fails and will be met with an *NX Domain* Error to the client.

As seen in the description of DNS above, the protocol is not intended to be used for data transfer, since there is no free form "payload" as such. Hence, DNS queries are often excluded from security monitoring, also owing to the vast volume of DNS requests being made.

DNS exfiltration works by way of leveraging the *subdomain*[61] of a DNS Request. Recalling the flow of a DNS request presented above, a query for a subdomain that

[61] The subdomain is part of a Domain Name. The Domain Name is structured as follows: *subdomain.second-level-domain(2LD).top-level-domain(TLD)*. The subdomain can also be called *third-level-domain (3LD)* [54]

is not registered on a public Nameserver will be forwarded to the Nameserver of the next-higher domain. In this case, a DNS request for `not-known.example.com`, where the subdomain `not-known` is not registered on a public Nameserver, would result in a forwarding of the request to the Nameserver of `example.com`. Exchanging this subdomain for a string which contains compressed data, this data can be exfiltrated to this (malicious) Nameserver, potentially in multiple calls, if the data is too large to fit in one string smaller than 64 characters[62]. The Nameserver of, in this case, `example.com` can then retrieve the data from the subdomain label string. For example, a DNS call to `SGVsbG8gV29ybGQhCg.example.com` would let the Nameserver of `example.com` extract a base64 encoded string `SGVsbG8gV29ybGQhCg` from the DNS query, which would let it extract `Hello World!` as data, potentially exposing trade secrets from confidential sources.

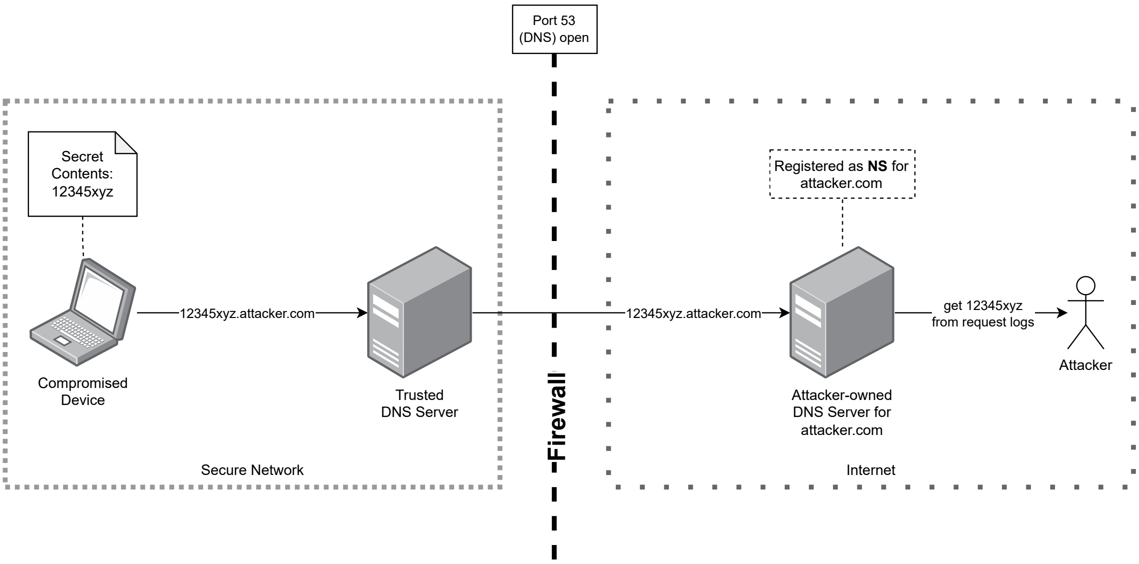


Figure 10: DNS exfiltration Schematic

HTTP Exfiltration: HTTP exfiltration uses standard HTTP calls to exfiltrate data. Therefore, any HTTP Method may be used to exfiltrate data, since HTTP specifies some sort of payload for all HTTP Methods. This includes even HTTP Methods such as GET or OPTIONS, since there are headers or URL parameters that can be filled with arbitrary data[63], as long as the receiving party (a malicious server) knows how to handle these (potentially "illegal" by specifications such as REST) requests.

[62] DNS specification permits each "label", e.g. the parts delimited by periods in the domain, to be at maximum 63 characters long [54]

[63] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`

Detection of these infiltration methods differs heavily in possible avenues of execution. HTTP exfiltration is, in most cases, comparatively easy to detect and prevent. In the case of a CI/CD environment, HTTP traffic can and should be restricted based on a Whitelist basis. This means only allowing HTTP Traffic to and from known domains. In CI/CD environments, restricting access to domains such as code repositories, package registries and possibly other endpoints known to DevOps personnel has to be implemented. In the case that this is not possible, intelligent threat detection software such as Splunk[64] or AWS GuardDuty[65] provide means to monitor for suspicious HTTP (and other) connections.

Detection of DNS exfiltration is much more complex, and, as DNS traffic is usually less heavily scrutinized, not commonly implemented in ready-made solutions. Das et al. [55] propose machine learning models to detect features deemed by them to be typical of DNS exfiltration traffic. Their metrics averaged an $F^1 = 0.96$ [55, p. 740], with a stronger false-negative tendency than false-positive (meaning less restriction on legitimate traffic due to false positives). They also propose another threat model - DNS Tunneling [55, p. 740]. In this case, the malicious code does not exfiltrate data, but requests instructions from the malicious DNS server. These are given by the DNS server by answering the DNS requests with DNS specification compatible TXT records (or A- or AAAA-records)[66]. Their evaluation via PoC proves their proposed ML solution to be effective, but they nevertheless conclude that knowledge about their system might render it ineffective by crafting the malicious code such that it circumvents the detection algorithm by altering its behavioral patterns [55, p. 742]. Nadler et al. [40] propose their solution only to the low-throughput DNS exfiltration problem, since according to their research, it is much harder to detect and there are at least ten known malware families exploiting the method [40, p. 3]. Their proposed approach relies on anomaly detection, which relies on a dataset comprised of a recent history of DNS requests. The system collects DNS traffic and converts it into feature vectors corresponding to domains, and then uses a one-class-classifier to determine if data is exchanged over these vectors [40, p. 6]. The anomalies this is based on are abnormally long requests and responses, encoded payloads, and a large number of unique requests (in terms of subdomain queries) [40, p. 6]. The solution itself is supposed to be implemented on a DNS server as a constantly running process.

[64] https://www.splunk.com/en_us/devops.html

[65] https://aws.amazon.com/guardduty/

[66] All these are DNS record response types. TXT can include arbitrary text, while A and AAAA return IPv4 and IPv6 addresses respectively

Their system proved effective in detecting requests used to exfiltrate data via DNS requests, but initially showed numerous misclassified domains to be blocked. This rate dropped significantly after the algorithm was deployed for more than two days of traffic [40, p. 11]. Their algorithm also proved ineffective against legitimate high-entropy domain names, used by specialty services [40, p. 11].

Ultimately, detecting data exfiltration over HTTP is fairly trivial and integrated in most solutions deployed for threat detection. Air gapping a CI/CD environment on the HTTP traffic level is realistically possible even without employing the use of self-learning dynamically blocking tools, since sources and targets of HTTP traffic should be easy to determine and whitelist.

Doing the same for DNS traffic is far more complex, and in the case of pure DNS exfiltration of data, hard to detect and possible to evade even learning systems [40, 55] by changing the patterns of the attack, especially considering the threat of false positives making it hard to seal the environment for fear of crippling vital services (for example McAfee security services [40, p. 11]). These services might even be other security solutions that use DNS exfiltration to extract information from heavily guarded systems, such as Hidden Objects' "SpyDR"[67].

2.5.2 Organizational/operative

Strong Identity Monitoring

Hensley [56] in his article raises several important points regarding proof of identity in SSCAs. Although he primarily focuses on verifying the identity of actors regarding access to systems and data, he applies this also to suppliers of third-party software, citing the SolarWinds Attack [57]. In this case, attackers compromised the trusted access of a third-party module to attack their victims from a trusted source. This makes the case for stronger and more in-depth authentication of the identity of actors on the side of the party integrating these modules, but also on the side of the provider of the modules. By being able to verify that these modules come from a trusted source or a trusted author, the modules can be implicitly trusted [56, p. 3]. This does require a system of identification where the token or similar mechanism, or the mechanism providing these tokens used to realize the authentication cannot be compromised [56, p. 2].

The problem of clearly identifying authorship is the central reason why attacking open source projects as a malicious maintainer or contributor is so effective, as

[67] `https://youtu.be/oQQ7VvDboPU?t=525`

discussed in section 2.1.2. One reason being that qualifications and references (as in “This contributor can be considered legitimate because he has previously contributed to other projects”) can be faked, and compromising identification tokens of legitimate contributors is possible and used in practice [8, p. 29]. To counter these kinds of attacks, a way of identifying contributors to code that is to be distributed, which is more resilient against these kinds of impersonation attacks is needed.
This concept pairs well with the concept of checking hashes of software artifacts, as suggested by the MITRE ATT&CK Framework in ID T1195.001[68], since a trusted hash of an artifact can only be created by a trusted author, necessitating strong proof of identity.

Best Practices

If there are any private package registries used for npm packages in the development process at all, it is very important to first choose a scope (see section 2.2.2), possibly the company name, and publish all internal packages under that scope [58]. This scope should, immediately after choosing it, be registered with at least the main public npm repository, if not any that can be found. Choosing a scope will make it impossible to publish any packages under that scope if not in the collaborative team, and make it possible to associate a specific scope with a specific registry (in this case the private registry)[69].
Using the lockfile (see section 2.2.2) when building projects as part of a CI/CD pipeline using the `npm ci` command also greatly reduces the risk of pulling packages employing a form of version spoofing (see subsection 2.4.2)[70].
Generally, limiting network access of build systems as much as possible, including public registries (provided publically available packages can be made available through the internal registry), can be a tool to limit possible intrusion vectors.

[68] https://attack.mitre.org/techniques/T1195/001/

[69] https://docs.npmjs.com/cli/v8/using-npm/scope

[70] https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json

3 Characterizing Dependency Confusion Attacks

In the following chapter, real world examples and reports of Dependency Confusion Attack (DCA)s will be used to determine characteristics of these attacks. These characteristics will be used in the following chapter to develop requirements for a system to prevent DCAs.

3.1 Real-World Dependency Confusion Attacks

There are very little to no scholarly publications available analyzing or describing the DCA in ways that could benefit this analysis. This means that an analysis of DCAs has to rely on threat reports and articles describing how to perpetrate this attack as a "security researcher". This does, however, give insights into the typical threat actor, since the latter type of report, or rather tutorials, usually describes ways of exploiting the attack vector. It should also be noted that the attack reports, specifically, are usually published by companies also selling security solutions, so a heavy bias towards highlighting vulnerabilities that can be prevented with the solutions being promoted is to be expected.

On the other hand, bearing in mind the practical premise of this thesis, an in-depth analysis of DCAs would go beyond the intended scope. Thus, a balance has to be struck concerning the depth of analysis and the intended implementation of a preventive system.

The last point to consider before moving into the analysis of the attack, is that the public registries affected by this attack have been ramping up efforts to delete packages used to demonstrate this attack as Proof of Concept (PoC)s[1], so finding packages to analyze might prove hard to impossible, preventing a direct analysis of the attack by ways of dissecting packages.

[1] https://twitter.com/alxbrsn/status/1365308546296995847

3.1.1 A Subset of Attacks

To provide a basis for analysis, a subset of attacks has to be selected. For the reasons provided above, this subset is mostly represented by threat reports, that, for the reasons of either the original malicious packages being already irrecoverably deleted, or the attack details being considered confidential by the victims, cannot be independently verified. Due to the additionally relatively scarce nature of these reports themselves (even though an estimated 63000 DCAs have been reported [59]), this quantitative analysis will have to be heavily augmented by an analysis of the attack architecture in theory.

The Attacks/Reports analyzed will be marked **A***X*, **A** for "Attack" and *X* being a numerical identifier, for reference in developing the characteristics.

Note, that the sources for the attacks described are given at the beginning of each section, and all information thereafter is taken from there.

A1: The Original Proof of Concept

February 2021 [19]

The first attack to consider is the originally published PoC by Alex Birsan in February 2021. In this first-of-its-kind (to be recorded) attack, or rather series of attacks, a large amount of packages used internally by a number of high-profile companies were namesquatted on public package registries.

The attack was perpetrated on registries for JavaScript (node package manager (npm)), Python (PyPi) and Ruby (RubyGems). The Attack consisted of squatting names of internal packages used by different companies, which were acquired by scanning package manifests (see section 2.2.2) on publically available code repositories such as GitHub. These package names were then squatted (the malicious package placed under this name) with numerous version tags on the above-mentioned public registries. The large number of published versions (100 to 1000 per package) is especially notable.

The attack itself consisted of extraditing data about the machine the attack was executed on and the network the machine was part of via DNS Extraction. The Attack proved very successful, infiltrating and extracting data from more than 35 organizations to the date of the report's publication. The exfiltrated data also showed that the affected systems ranged from individual developers to internal (presumably on-premise) and cloud-based build systems as well as vulnerable development pipelines. The packages used for the attack did not contain any of the original code, and merely served to carry the payload of the researcher's DNS extraction code.

A2: First Copycats Appear

February/March 2021 [60, 44, 46]
Shortly after the publication of the research and PoC, the security services and research provider Sonatype initially flagged over 275 packages directly or almost directly copying Birsan's strategy from the PoC on npm. Notably, the same functionality concerning extraction of machine and network data over Domain Name System (DNS) was included in all affected packages. Additionally, the packages only contained this code, removing any of the functionality of the packages whose names were squatted to facilitate the attack. The attacks targeted, according to the naming schemes used, companies such as Apple and Shopify. Identification of the authors was not possible, and association with the targeted companies considered unlikely.
In March of the same year, the number of identified packages targeting the DCA mechanism jumped to over 700. These packages were again discovered and reported by Sonatype, who identified them based on the same DNS extraction mechanisms used in the PoC. At this point, malicious behavior apart from data exfiltration has not yet been identified, suggesting merely copycats trying for bug bounties, similar to what was paid out by many companies for the original PoC. Notably, one of the packages contained additional code that opened a reverse shell on an affected machine (or tried to) [60], marking the first time a deviation from the PoC attack pattern is spotted. The packages were, again, published by pseudonymous authors and appeared to contain no code apart from the malicious payload. Every package published does appear to have numerous versions associated with it, with no apparent deviation in content. The packages have been deleted from the public npm registry and are not available for analysis on registry mirrors either.

A3: Snakes On A Package Index

January 2022 [61]
The copycat attacks in 2021 that were also covered in Sonatype's "State of the Software Supply Chain" [5] were mostly confined to npm as a package index. Some packages using the attack vector of the PoC were published to the Python Package

Index (PyPI) as well, but a marked number of these packages were uploaded top PyPI in January 2022 by a user named “arturlebedev”. 1275 packages were uploaded by this user, all targeting well known, large companies, using package names of internally used packages. Among the companies targeted were Google, Sagepay, Apple, the standard Python setup tools, India’s national biometric identification system “Aadhaar”, and several others.

The packages were deleted within an hour, information about how often these packages were downloaded is not available. The payload of these packages was once again an attempt at exfiltrating fingerprinting information about the machine and network, this time not only over DNS, but also using HTTP. Actual functionality was apparently included in the relevant packages, apart from the malicious exfiltration code. The author “arturlebedev” does not seem to be affiliated with the companies targeted. From the limited information still available after the packages have been deleted, the pattern of dumping a large amount of package versions of one package can be observed here as well.

A4: Virtual Machines, Real Confusion

April 2022 [62]

In April 2022, Sonatype reported a suspicious package that didn’t appear to squat any previously known internal package name on PyPi, ostensibly belonging to the VMware VSphere Automation SDK, part of VMware’s extensive virtualization solution. Further research indicated, however, that this package was named after a previously unknown internal VMware package[2]. The source of the package was determined to be a Ukraine-based security researcher. Vladyslav Kotko, who is not associated with VMware in a development capacity, published the package containing a payload similar to the original PoC in March 2022 under the name “vapi-client-bindings”. No code related to the nominal purpose is contained in the package, only the “call home” functionality using the DNS and HTTP based exfiltration technique. The package name was reclaimed by VMware with the help of the package index shortly after.

[2] `https://github.com/vmware/vsphere-automation-sdk-python/tree/master/lib/vapi-client-bindings`

A5: White Team Caught Red Handed

April/May 2022 [63, 64, 65, 47]
In April and May 2022, ReversingLabs, Heise, jFrog and Snyk all reported a major attack on German companies Bertelsmann, STIHL, Bosch and DB Schenker using the DCA method. The packages published under pseudonyms referencing the victim companies' names constituted a two-stage attack, the first variation seen on the pattern of the original PoC. The attacking packages contained obfuscated code (the first variation) and, also a first, actual dependencies which themselves contained malicious code. In this case, the attackers thus used the original PoCs way of injecting their packages into their victims' environments, and then used the known-exploitable package manager to pull more malicious dependencies, preventing identification of the malicious code by, for example, static analysis before installing the package's dependencies. The attack payload in these secondary packages, however, still appeared to be DNS extraction of fingerprinting data. However, some more functionality in the form of a "self-destruct" function was present. More analysis of the package revealed an additional mechanism consisting of a dropper and a payload. If the malware package manages to infiltrate a system and "call home" via the above-mentioned DNS exploit, a decrypted malicious payload is sent back, and decrypted by the malware "agent". This payload can either be JavaScript or a binary file, which will be executed upon receipt by the malware agent. This is particularly interesting, since the first step of fingerprinting the victim exposes the information necessary to compile native binaries for the victim's machine to the attacker. The payload that was observed in the analysis was a JavaScript file instantiating a command-and-control client that communicated with a server controlled by the attackers.
Analyzing the package further, all sources reported that it appeared to be part of a penetration test performed by German red team security consultancy Code White GmbH, who later owned up to the attack on Twitter[3]. This marks the first time that the malicious payload is more than the DNS extraction of data used in the PoC, and switches it out for a sophisticated attack, that is even hardened against static analysis. Analyzing the reports from jFrog and Snyk does, however, imply that no actual code used in the original packages being impersonated here is contained in the impostor packages.

[3] https://twitter.com/codewhitesec/status/1524016955186982912

3.2 Characteristics of Dependency Confusion Attacks

Based on the examples of real-world DCAs provided in section 3.1, a number of characteristics have been identified, that quantify an attack as a DCA. In the following sections, these characteristics will be named and explained.

3.2.1 C1: Targeting Packages from Private Registries

The first characteristic that can be spotted from all of these attacks is the fact that DCAs, as opposed to for example the closely related Typosquatting Attack (see subsection 2.4.1), target exclusively preexisting packages under their full names. However, these packages being impersonated cannot have already been published/exist on public registries, since the main attack vector is squatting an existing name on a public registry. These names are obtained through Open-Source Intelligence (OSINT)-Methods, such as ones explained in section 2.4. Thus, the packages being targeted for impersonation are packages that are stored on private registries pre-attack.

Characteristic **C1** can be observed in all attacks described in the previous chapter. Especially apparent in attack A1, where these companies were explicitly targeted, but also in A3, A4 and A5, which represented attacks specific to internally used packages from private registries that were obtained - presumably - through OSINT techniques. This is very likely a feature of A2 as well, although the state of facts being this opaque makes this difficult to argue.

3.2.2 C2: Malicious Package Author can not Publish to Private Registry

The second characteristic observable from all real world DCAs presented in the scope of this chapter is the fact that malicious code is injected into the application via an external package. It is important to note, that an attack that would publish a package containing malicious code directly to an internal package registry, would not constitute a DCA. Publishing to a private package registry, for example company-internal systems, requires credentials. The attack in this case would not constitute a DCA, but possibly obtaining Valid Accounts[4] through other attacks.

It is therefore safe to assume, that in case of a DCA the author of the malicious package does not have access (or credentials) to publish directly to the private

[4] MITRE ATT&CK T1078 `https://attack.mitre.org/techniques/T1078/`

package registry, instead having to exploit the behavior of package managers to inject his malicious code through package impostors.

Characteristic **C2** can be observed in all attacks. This can be considered the baseline for all the attacks, considering that, if the attackers had access to the internal registries whose packages were targeted, it would be easier to directly inject malicious packages instead of choosing to inject them by exploiting secondary systems.

3.2.3 C3: Malicious Package Version contains Code Different to Legitimate Version

The third characteristic common to all observed DCAs is the packages' actual code being different to any legitimate version. This is especially apparent, when considering existing versions being impersonated. This characteristic is especially noticeable, since even in non-malicious, explorative, exploits of the DCA technique, there is still code added to the package that differs from the legitimate version. Additionally, it has to be considered that, even if the DCA was employed to facilitate injection of a package into a victims' system, if it contained no additional code (malicious or not), so that a check of the malicious packages' file integrity would not reveal a difference to the legitimate package, the 'malicious' package would not actually constitute a threat, since the function would be exactly the same as the legitimate ones'.

This fact then allows for the assumption, that any actual malicious package being used in a DCA would then be clearly distinguishable from the legitimate package by means of its file integrity information.

Characteristic **C3** can be observed in all attacks mentioned. Attacks A1, A2, A3 and A4 show this in the way that they contain nothing but code used to fingerprint the victim and extract this information via DNS - something that is itself unlikely for a productively used, internal, package - but no actual code in terms of the functionality promised by the name of the package either. Attack A5 is especially interesting to this case, containing sophisticated malware with additional dependencies and a dropper-payload agent. The packages used in A5, however, do not appear to contain any actual productive code either.

3.2.4 C4: Malicious Package Version Author is not the Original Author

The fourth characteristic of a DCA is the difference in authors between the legitimate version of the package versus the impostor package. This does not mean that the actual person or system publishing the package is not the same - this can not be verified - but that the verifiable identity of the authors differs. In essence, this means that legitimate packages are published by authors known to the consumers of the package, while the authors of the impostor package are basically anonymous to the consumers of the package. In most cases, the author of the legitimate package will be known and verifiable in the (likely enterprise) context of the package's use, while the author of the impostor package will not be identifiable within the same context. It can therefore be assumed, that making a package's author verifiable throughout its lifecycle would enable implicit trust in the package itself. In other words, if a package (and all its constituent files) can be traced to an identity of an author throughout building, publishing and consumption of the package, the package can be trusted.

Characteristic **C4** can be observed in all attacks, more clearly in attack A1, A4 and A5, where there is a definite disconnect between the (after the fact) known authors and the victims. However, keeping in mind the pseudonymous nature of the authors in A2 and A3, it can be safely assumed that these authors are very likely not the authors of the existing packages.

3.2.5 C5: Malicious Package is a Version of an Existing Package

The fifth characteristic of a DCA to be observed, is that every impostor package used to facilitate a DCA is a version of an existing package. Importantly, and included in characteristic **C1**, this package cannot exist on public registries. The existing package, as defined by any version of a package identifiable by a package name, having been published on any internally used registry, creates the basis on which to execute the DCA. Without a package to base the attack on by squatting the name and version(s) on a public registry, the DCA cannot be executed. A DCA is therefore only possible by using already published (internal) packages. This means, the attack vectors of a DCA are known to the possible victims beforehand.

Taking this into account, the assumption can be made, that prevention of DCAs is possible by observing only the finite set of previously known internal, legitimate, packages. The number and nature of all possible DCA attack vectors is therefore known a priori, and can be used in conjunction with other characteristics to prevent DCAs.

Characteristic **C5** can be observed in all attacks. Every attack described targeted explicitly internally used packages, which, by the nature of these attacks, existed in a private repository somewhere. The attacking packages therefore constitute versions of an existing package. Additionally, package versions of these malicious packages, were typically distributed in large numbers, meaning many version numbers of these packages were published, that mostly had no original counterpart and were designed to take advantage of package managers' version resolution strategies.

3.2.6 Overview of Characteristics

The characteristics discussed in the previous sections are summed up in Table 5.

Table 5: Overview of DCA characteristics

Index	Description
C1	DCAs target packages that are consumed from private package registries and squat their names on public registries in order to impersonate them
C2	When executing a DCA, the author of the malicious package has no access to publish directly to the private registry and relies on the package manager of the victim to confuse his malicious package on the public registry for the legitimate package in the private registry
C3	Any malicious version of a package is distinguishable from its specific legitimate counterpart by the cumulative integrity of its included files
C4	A malicious package in the scope of a DCA cannot be traced back to a verifiable author in the context of the legitimate package's usage scope
C5	Any malicious package used in a DCA is a version of a previously existing legitimate package from a private registry

4 Developing a System to prevent Dependency Confusion Attacks

4.1 Choosing an Attack Prevention Strategy

After giving an overview of the Dependency Confusion Attack (DCA), looking at existing preventive methods and characterizing the attack based on real-world examples, a strategy to prevent the attack must be chosen with this information in mind. This strategy should be based on the methods of prevention discussed previously, and offer a realistic chance of preventing the DCA while being able to be implemented in a system that can be integrated with existing Continuous Integration/Continuous Deployment (CI/CD) environments.

The final choice of strategy is based on the exclusion of strategies that were considered inappropriate. Detecting a DCA in progress, as discussed in section 2.5.1, was not a strategy considered for the preventive system. As discussed in that section, the payload of the attack may carry any malicious code to be executed on successful infiltration. Thus, a definite attack pattern that could be detected in the execution environment cannot be distilled from the attacks that have been executed so far. Additionally, the malicious payload so far has mostly executed data exfiltration via Domain Name System (DNS), which in itself is hard to detect and prevent (see also section 2.5.1).

Another strategy that was considered, but ultimately not chosen was dynamic analysis. As discussed in section 2.5.1, there are several methods available for dynamic analysis, which all rely on having a sandboxed environment to execute the unknown code in. This would mean providing each CI/CD environment with its own sandboxed environment, which would need to be maintained, and would impact execution times of the CI/CD process. While this may not be a problem for traditional, on premise, infrastructure, in terms of cloud-based CI/CD processes, a much larger number of different CI/CD environments is common, and would complicate the introduction of sandboxing or dynamic analysis infrastructure. Additionally, the typically large number of dependencies in the node package manager (npm) ecosystem (compare subsection 2.2.1) would make the dynamic analysis of each slow and

involved.
The strategy ultimately chosen thus combines aspects of static analysis (see section 2.5.1) and strong identity monitoring. Other aspects of static analysis that were considered, but decided against, were the introduction of static code analysis by clusters of known malicious code and the detection of obfuscation. This choice was made based on the fact, that the actual code contained in the packages that were used in executing the DCA offered little potential for analysis or building a catalog of known clusters of malicious code. A much higher potential for preventing the attack lies in determining if a given package is actually the package it purports to be, or an impostor. To this end, the decision was made to use integrity verification in combination with strong identity practices in order to prevent the DCA.

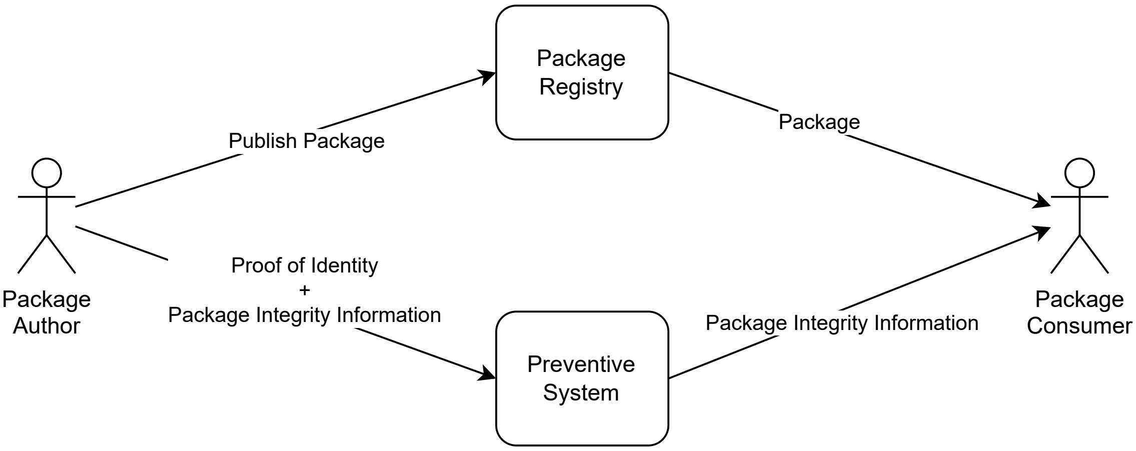


Figure 11: The preventive system's attack prevention strategy at a high level

The strategy that was chosen is shown in Figure 11. In essence, the system allows a package's author to provide integrity information about a package he publishes to a package registry to the system, together with strong proof of identity. This allows the system to provide this information to consumers of the package, who can then use this information to verify the authenticity of the package, while being assured that the information provided to the system is authentic. This concept is used as the basis for the development of the preventive system. In the following sections, additional requirements towards the system are detailed, which include requirements based on the characteristics of the attack itself, the technical framework the system is to be deployed in, as well as functional requirements towards the system. These requirements, which will reflect the preventive strategy chosen here, will then be synthesized into an architecture.

4.2 Converting Dependency Confusion Attack Characteristics into Prevention System Requirements

In this section, the characteristics of DCAs explained in chapter 3 will be used to infer requirements to a system used to defend against these attacks. These are the central requirements the system is to be built around, forming the core of the functionality of the system.

Requirements are identified by IDs in the form of `RC-Y`, with **R** signifying “Requirement”, **C** signifying a Requirement derived from a characteristic, and **Y** being a numerical identifier.

Table 6: Requirements derived from characteristics of the attack

ID	**Description**	**Rel. Characteristic**
R-C1	The System must be able to discern publically available packages from ones provided via internal registries	C1
R-C2	The system must be able to read the lockfile or manifest of any project to acquire information about the actual source repository of the package	C1, C2
R-C3	The System must be able to store and access known-reliable integrity information about privately published packages, as well as be able to query information from the registry of any package mentioned in the lockfile or manifest to compare them	C2, C3
R-C4	The System must be able to provide the ability to store and query information about a packages author, making him identifiable.	C4
R-C5	The System must be able to store and access known-reliable information about all versions of privately published packages, as well as be able to retrieve this information from the lockfile or manifest of the project being scanned	C1, C3, C5
R-C6	The System must enable package publishing authors to store integrity and version information about package versions while binding it to a clearly identifiable property of the author	C2, C4

4.3 Identifying Functional Requirements

In this section, functional requirements are identified, which are posed against the system to integrate it in existing environments. Fulfilling these requirements is essential to guarantee actual real world use and impact of the security system to prevent impostor packages being injected instead of legitimate internal packages. Requirements are identified by IDs in the form of `RX-Y`, with **R** signifying "Requirement", **F** for a "Functional" Requirement and **Y** being a numerical identifier.

The functional requirements describe requirements for the functionality of the system and its required behavior in regard to user input (compare Pohl [66, p. 8]). The users, the main source of the functional requirements, should not have to alter their workflows significantly to benefit from the system, to prevent the users from not using the system at all, if its implementation into the existing workflows is too cumbersome. Additionally, the function of the system must allow the verification of integrity information of packages, as well as the ability to store it, to allow for the preventive methods chosen in section 4.1.

Table 7: Functional requirements

ID	Description
R-F1	Authors of internal packages must be able to authenticate to contribute new package versions without providing additional credentials other than their existing domain credentials
R-F2	Authors of internal packages must be able to be authorized based on existing roles or groups
R-F3	The security system must not change the workflow of contributing new package versions other than stopping it if malicious activity has been detected
R-F4	The security system must provide functionality for authorized authors to store integrity information on new package versions identified by name and version
R-F5	The security system must provide functionality for consumers to verify integrity of packages based on package name and version
R-F6	The security system must store package integrity information in such a way that possibilities of manipulation by both in-house and third parties are prevented
R-F7	The security system must provide a way to audit packages that cannot be verified on consumption, for packages that are not developed internally, and thus consumed from sources other than the internal registry legitimately

4.4 System Development

Based on the requirements developed from the characteristics of DCAs (see section 4.2), as well as functional requirements (see section 4.3), the system realizing these requirements was developed. The Development is documented here at a high level in the form of technologies chosen, architectural decisions taken and an acknowledgment of limitations the system faces. Detailed documentation of the development process itself, including details about the programming and testing, has not been included, as it would increase the size of this already large chapter disproportionately to the actual information content, and detract from the more compact and important information of higher-level architectural decisions taken. The source code is available and will be included with this document.

4.4.1 Technologies

The System was developed based on technologies that satisfied the following requirements:

- Availability: The technologies should be readily available and accessible
- Functionality: The technologies should represent a real-life potential to be used (or are already used) in the context of cloud-based CI/CD environments
- Popularity: The technologies used should be relatively popular, both to ensure (ongoing) support and realistic impact of the proposed solution
- Familiarity: The technologies should be known to the author, ideally having already been used, to ensure speedy development work and realistic completion of the project

These criteria ensure a system that is both easily integrable and maintainable by actors other than the author on a technical scale at least.

System Environment

For the reasons detailed in subsection 4.4.1, the following ecosystem was chosen to implement the system:

Table 8: Technologies used to implement the project

Category	**Technology**	**Description**
Runtime	node.js	Very popular server-side JavaScript runtime, intensive use of third-party packages
Package Manager	npm	Supports node.js, the package manager associated with 63% of malicious packages in a recent study (see [8])
Cloud-Vendor	AWS	Supports serverless CI/CD environments with minimal setup, supports IaC with TypeScript/node.js minimizing setup time

The architecture chosen for the implementation of the Proof of Concept (PoC) is shown in Figure 12. This architecture will be used to both implement an attack and, following that, will be hardened and attacked again to evaluate the effectiveness of the system. The architecture is split between repositories on GitHub and repositories as well as other services on AWS. This split is used to ensure that even when potentially infecting services on the AWS Account where the system (and attack) will be implemented, the AWS account can potentially be “nuked”, removing any and all services including the code repositories. The AWS code repositories have been configured to allow the GitHub code repositories to push code to AWS, but no transfer is permitted in the other direction.

AWS: On the AWS infrastructure side, two CI/CD Environments have been configured. One, the Package Pipeline, has been configured to trigger on changes committed with Git to the “main” branch of the associated code repository. This repository has been implemented with AWS CodeCommit, and is served by an “off-site” GitHub repository. When a commit occurs, the repository triggers the AWS CodeBuild service. In CodeBuild, a Build Step has been configured that builds a npm package from the code in the repository and publishes it to an internal registry. This registry has been implemented with AWS CodeArtifact, which provides a CommonJS compliant package registry (see subsection 2.2.2), allowing npm packages to be published and requested from it. This package registry is available from both the Package CI/CD Environment and the Application CI/CD Environment. The Application CI/CD Environment is used to build an Application consuming the package built with the Package CI/CD Environment.

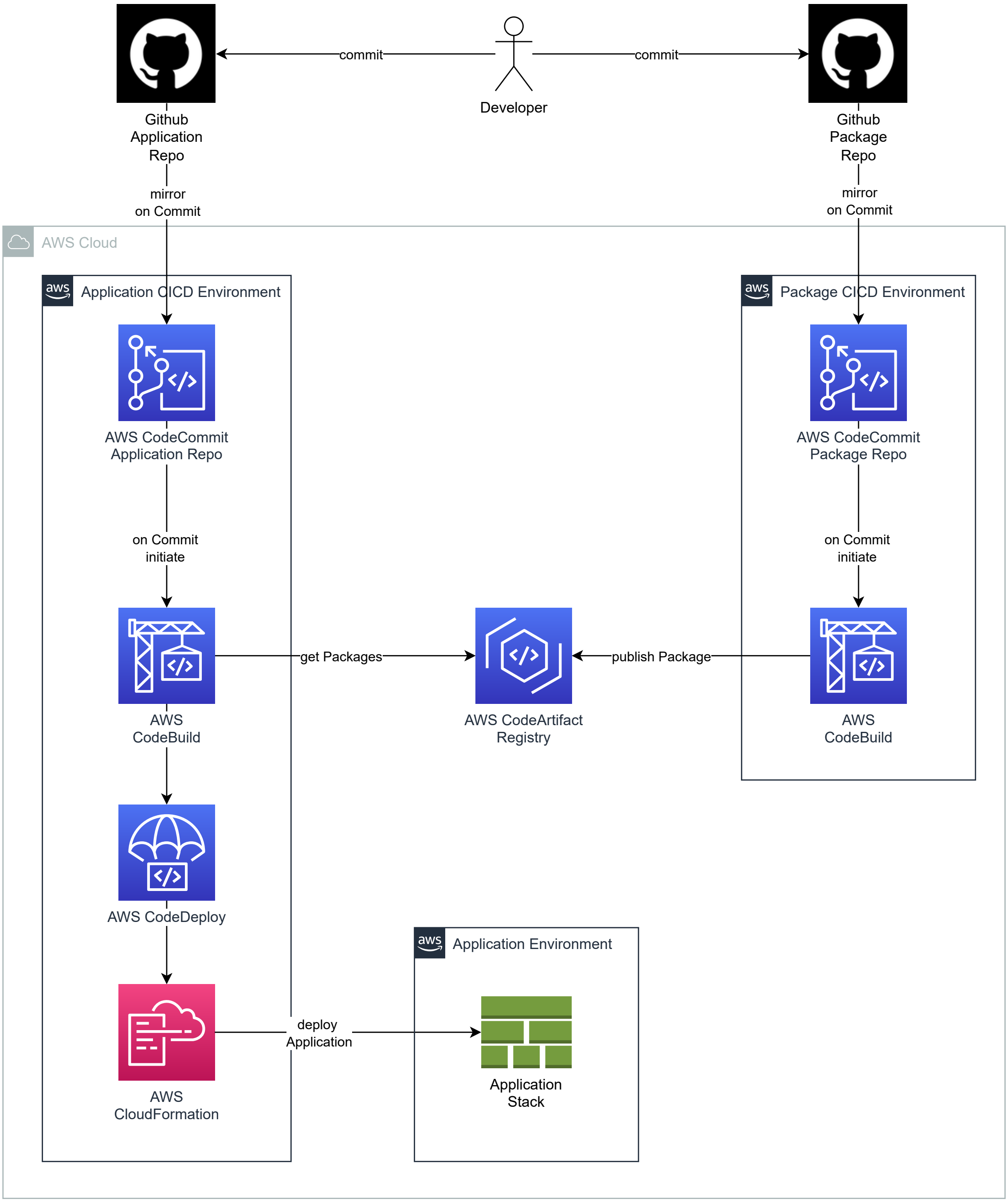


Figure 12: CI/CD Architecture for PoC attack and hardening

Vulnerable Application

The Application being built by the vulnerable CI/CD pipeline is a very simple piece of software used to deploy IaC (using the AWS Cloud Development Kit (CDK) in TypeScript as a technological basis) on the AWS cloud platform. To accomplish this, it consumes a package containing a construct, itself IaC, that serves to implement more functionality. In this scenario, the functionality implemented by the package being consumed can be regarded as a typical reuse of common patterns within corporate development environments. Functionality that is often implemented the same way across many projects is packaged and distributed via internal package management systems. In this case, the actual content of the shared component is irrelevant, so it contains merely an AWS Lambda Function that does not implement any actual logic. The rest of the application is empty, as the function of the application has no bearing on the attack being carried out nor the prevention of it. The Application architecture and system context is shown in Figure 13.

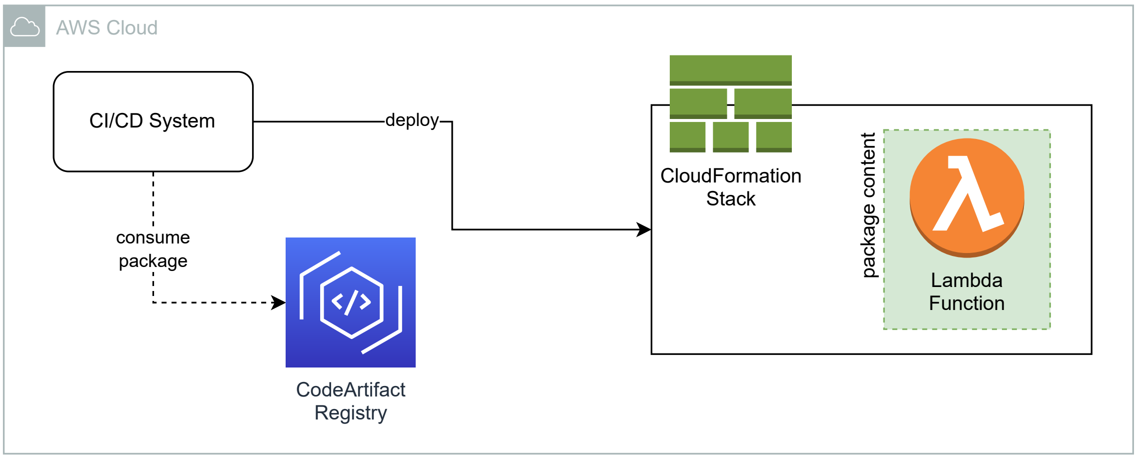
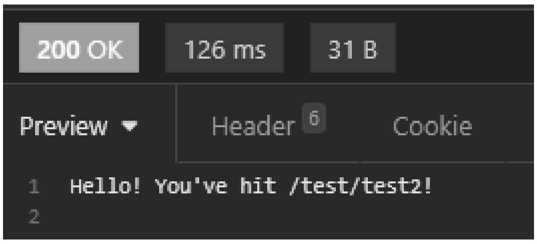


Figure 13: Architecture of the application deployed via the vulnerable CI/CD pipeline

Legitimate Package

As discussed in the details of the vulnerable application above, the package implements an IaC component used in the main application. This component is also implemented with the AWS CDK in TypeScript. To make it distributable and consumable as a package, the application component being distributed is implemented as an AWS CDK construct, describing a reusable component in the vernacular of AWS. The distribution of this package to the internal registry is shown in Figure 12.

To enable carrying out the attack, an impostor package was developed as well (see subsubsection 6.1.1). This package implements the same functionality as the legitimate package, but adds a `postInstall` script to the package manifest (section 2.2.2), outputting a message after the package has been installed.
The legitimate package contains an AWS Lambda function that outputs the message seen in Figure 14 when deployed and hit with an HTTPS Request.



```
Hello! You've hit /test/test2!
```

Figure 14: HTTPS response of the lambda function from the legitimate package once deployed

Vulnerable CI/CD System Environment

The CI/CD pipeline used in this setup contains a simple, three-step, process. The pipeline itself is closely associated with a code repository implementing Git as a versioning system and running on AWS CodeCommit[1] as the service provider.
A commit of new code in this repository triggers the CI/CD pipeline and starts a new run. The repository itself is coupled to an “off-site” repository on GitHub (see also section 4.4.1), which is coupled via SSH Key for authentication and authorization, to push code to the AWS CodeCommit repository. Human user access to the AWS CodeCommit repository is not intended nor provided. The SSH Key used is connected to a user account in the AWS Identity and Access Management (IAM) system, which includes a policy only allowing reading and writing to the specific code repositories (Legitimate Component and Vulnerable Application).
A commit to one of these repositories then triggers the CI/CD pipeline of either the Legitimate Component or the Vulnerable Application. In case of the Legitimate Component, this includes:

1. **Sourcing**: In this step, the code associated with the new commit gets loaded into a temporary compute environment within AWS and packaged into a ZIP-File. This File is then output and made available to use for the next step

[1] https://aws.amazon.com/codecommit/

2. **Building**: In the Build Step, the output (ZIP-File) of the Sourcing step is loaded into an Environment specified in the AWS CodeBuild project definition. AWS CodeBuild is the service used to implement this step. It requires a build environment (in the form of a Docker container image), Buildspec (in the form of a bash script) and the build files (in the form of a ZIP file) to be provided. It will then perform the steps specified in the Buildspec on the build files, using the tools provided in the build environment. In the case of the Legitimate Component, it will install the "aws-cdk" package to the build environment globally to facilitate building the component which uses AWS CDK tooling. It will then authenticate with the internal registry, build the component using the TypeScript transpiler, and publish it to the internal registry using `npm publish`. The Buildspec is pictured in Listing 3

```
npm install -g aws-cdk
aws codeartifact login --tool npm --repository master-registry --domain master-
    registry-domain --domain-owner 365161439830
npm ci
npm run build
npm publish
```

Listing 3: Buildspec of Legitimate Component

In the case of the Vulnerable Application, the CI/CD pipeline includes the following steps:

1. **Sourcing**: This step is the same as with the pipeline of the Legitimate Component. The commit gets zipped and made available to the build environment in the Build Step.
2. **Building**: In the Build Step, the output (ZIP-File) of the Sourcing step is loaded into the defined Build Step. The general behavior is the same as specified in the pipeline of the Legitimate Component. In the case of the Vulnerable Application, the AWS CDK tooling will be installed. It will then authenticate with the internal registry, install the necessary dependencies from the manifest and build the component using the TypeScript transpiler. Lastly, it will synthesize an AWS CloudFormation template from the build files, providing a template to deploy the application in the next step. The Buildspec is pictured in Listing 4

```
npm install -g aws-cdk
aws codeartifact login --tool npm --repository master-registry --domain master-
    registry-domain --domain-owner 365161439830
npm i
npm run build
npx cdk synth
```

Listing 4: Buildspec of Vulnerable Application

4.5 System Architecture

In this Section, the architecture of the preventive system against DCA will be developed and explained. To this end, a functional architecture will be developed, which will specify the functions provided by the system. Building on the functional architecture, a technical architecture will be developed, which will specify how these functions will be implemented to produce a usable program.
Additionally, for the actual development, and in further explanation the short project name of "**K9**" has been chosen. This name is a homophone to "canine" and used in common parlance as a nickname for law enforcement dogs. In the case of the prevention system, it has been chosen since the system "sniffs out" impostor packages similarly to law enforcement dog "sniffing out" contraband.

4.5.1 Functional Architecture

The Chain of Trust

To understand the development of the functional architecture, a concept described as the "Chain of Trust (COT)" is introduced. This concept is used to distinguish the relationships of trust, between the components in the vulnerable system versus the hardened system implementing the proposed solution, along the Software Supply Chain (SSC).
Trust, in this context, means the ability of one component to rely on another, the "trusted component", to supply it with code or other artifacts that do not disguise themselves as other code or artifacts, and do not carry malicious behavioral patterns. This trust is comparable to a supplier-manufacturer-relationship in a physical supply chain (see subsection 2.1.1). In the physical supply chain, the manufacturer relies on the supplier to provide him with components that conform to specifications that are known to both, and enable the vendor, supplied by the manufacturer, to rely on the manufacturer to provide end-products that are manufactured to their specifications. This means, that every actor in this chain relies on another, to conform to standards of some kind. A COT between these actors is thus established. A breach in this COT, for example the supplier supplying components of poor quality, could be resolved in favor of the manufacturer if both parties have agreed to comply with standards beforehand, for example ISO or DIN standards. This introduces the concept of a "Central Authority" to the COT. In this case, the trust of the manufacturer is placed in the standards authority, instead of the supplier. In the same

vein, the vendor can place his trust in the standards authority, by agreeing with the manufacturer to manufacture to certain standards of the standards authority.

Relative Chain of Trust: Within the SSC, the situation is much the same. Specifically, in this section, the COT between the components making up the SSC on which a DCA would be executed will be examined. In the case of a SSC not implementing the proposed system, the COT can be represented as shown in Figure 15.

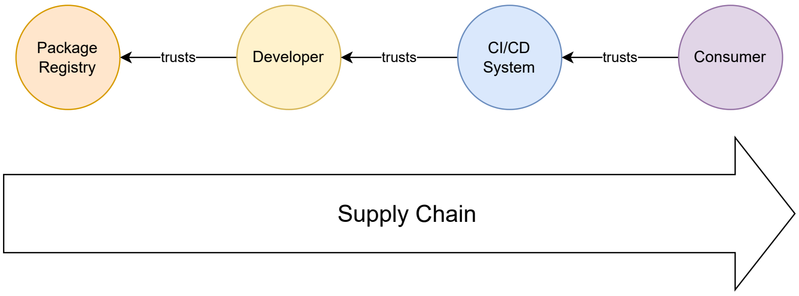


Figure 15: Relative "Chain of Trust" simple schema

In this simplified view, four actors are involved in the chain of trust along the SSC. These are:

1. **Consumer**: Consumes applications or other components produced by the CI/CD System
2. **CI/CD System**: Consumes code from the Developer and produces applications or other components
3. **Developer**: Consumes packages as dependencies from the Package Registry and includes these dependencies in code sent to the CI/CD System
4. **Package Registry**: Holds packages and information about them

The chain of trust starts with the developer consuming packages from the Package Registry. There is an implicit trust established from the developer, for the registry to supply him with the package requested. The developer can check with the Package Registry, if the package that has been delivered is the one requested, by requesting integrity information from the registry. Once the Developer finishes developing, he hands over the code, including dependencies and integrity information about those dependencies to pull from the registry, to the CI/CD System. The CI/CD System trusts the developer to supply it with correct information regarding the dependencies and their integrity information, which it uses to pull packages from the registry, and uses the integrity information provided by the developer to verify that those are the packages the developer intended to include. Once the CI/CD System has finished building the application or component, it provides these to the Consumer. The Consumer trusts the CI/CD System to provide him with an application or component that conforms to his expectations and likely the promises of the developer. This COT does not provide adequate means to detect malicious interference at any point. If the malicious actor manages to infect any stage of the process, be that the Package Registry, the Developer or the CI/CD System, the chain of trust is still intact. In Figure 16 both the “good path”, in which no malicious code is injected (Green, Option 1) and the “bad path”, in which a DCA is performed (Red, Option 2), are shown. In case of Option 2, the Developer (accidentally) consumes a malicious impostor package from the public npm registry. The package version of this malicious packages, as well as integrity information are written to the lockfile (compare section 2.2.2), which is contained within the Application Code. The Application Code is pushed to the CI/CD System, which will pull the package from the public registry. Even though an integrity check is performed, if the lockfile carries information about the malicious package, this integrity check will only ensure that the malicious package is delivered to the CI/CD System. This, in turn, leads to an infected application being produced by the CI/CD System and being delivered to end users.

By being forced to trust only the direct predecessor in the SSC, this system enables bad actors to poison one link of the package delivery system in the SSC, while other members have no means to verify the integrity of packages they consume. In contrast to the physical supply chain, there is no centralized standards authority to refer to.

Absolute Chain of Trust: The concept of the absolute COT introduces one such central authority into the concept. The proposed solution is called the *absolute* COT, since every link in the supply chain can rely on an *absolute* truth outside its direct predecessor. This *absolute* truth is provided by external systems. This is in contrast to the *relative* COT, where every link can only rely on the truth provided by its direct predecessor. The absolute COT proposed here is shown in a simplified version in Figure 17. The actors involved in the absolute COT are the same as those involved in the relative COT, with the addition of a central authority. The process itself still moves along the SSC, involving the Developer pulling packages from the

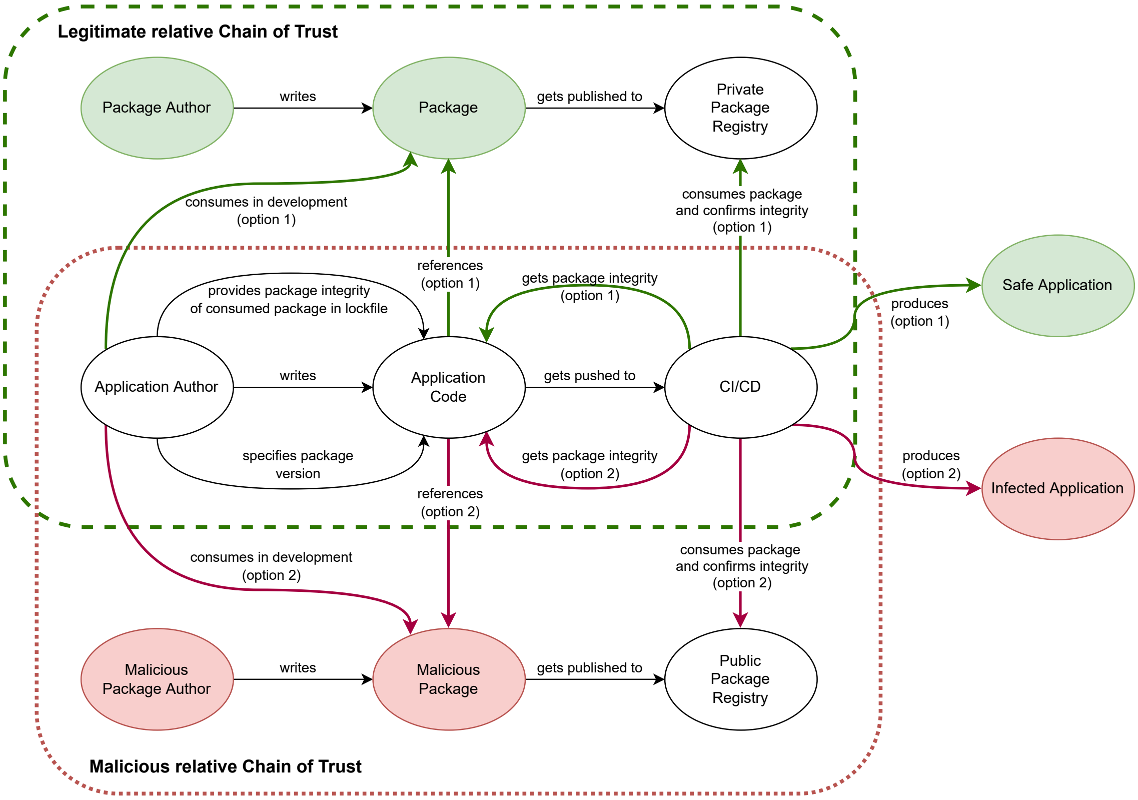

Legitimate relative Chain of Trust
Package Author
writes
Package
gets published to
Private Package Registry
consumes in development (option 1)
consumes package and confirms integrity (option 1)
references (option 1)
gets package integrity (option 1)
provides package integrity of consumed package in lockfile
Safe Application
produces (option 1)
Application Author
writes
Application Code
gets pushed to
CI/CD
specifies package version
gets package integrity (option 2)
references (option 2)
produces (option 2)
Infected Application
consumes in development (option 2)
consumes package and confirms integrity (option 2)
Malicious Package Author
writes
Malicious Package
gets published to
Public Package Registry
Malicious relative Chain of Trust

Figure 16: The relative "Chain of Trust" present in unsecured publishing and consumption of packages, both for the intended way (Option 1) and the possible way of intrusion by a malicious actor (Option 2)

Package Registry and forwarding code with dependency information to the CI/CD System, which in turn produces an application for the Consumer. In the case of the absolute COT, none of these participants in the supply chain need to trust their predecessor to supply them with legitimate material, since they can independently verify everything they're given.

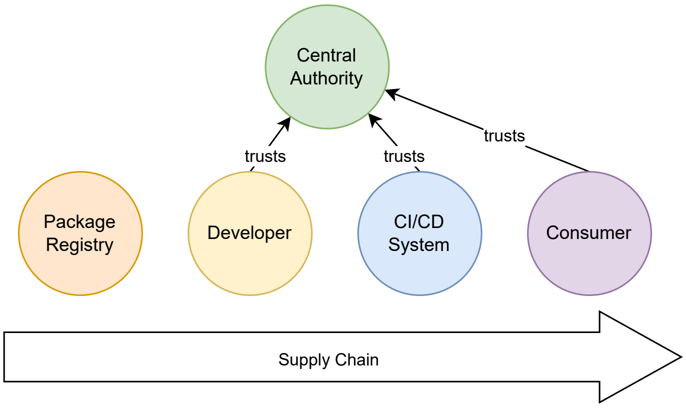


Figure 17: Absolute "Chain of Trust" simple schema

Malicious interference can be detected at any point, given that the Central Authority hasn't been compromised. In the case of the proposed K9 System, the Central Authority's most important component is a Smart Contract on a Blockchain providing integrity information about known internal packages. The decision to use a Smart Contract was taken for reasons such as availability, ease of use and maintainability. A more detailed Architecture Decision Record (ADR) explaining the reasoning can be found in the appendix, under section A.1. The full proposed absolute COT of the K9 System is shown in Figure 18 for the process of building an application including the consumption of dependencies from a private repository source. It can be clearly seen, that in this process, starting with the Developer, no trust is placed in the preceding link in the supply chain. The Developer consumes a package from the Package Registry, but does not rely on the (possibly wrong) registry to provide him with integrity information about the package, instead choosing to check the packages' integrity with information stored on the Smart Contract. Once this requirement is satisfied, the code, containing the package as a dependency, is pushed to the CI/CD System. The CI/CD System, while building the application, does itself not rely on any possible integrity information provided with the code, instead checking the integrity of any dependencies consumed with the Smart Contract, again ensuring that the packages consumed conform to the standard set by the original publisher of the package. This ensures that the application built by the CI/CD System, and being delivered to the Consumer, does not contain any unwanted and possibly malicious code due to violation of trust between the members of the supply chain.

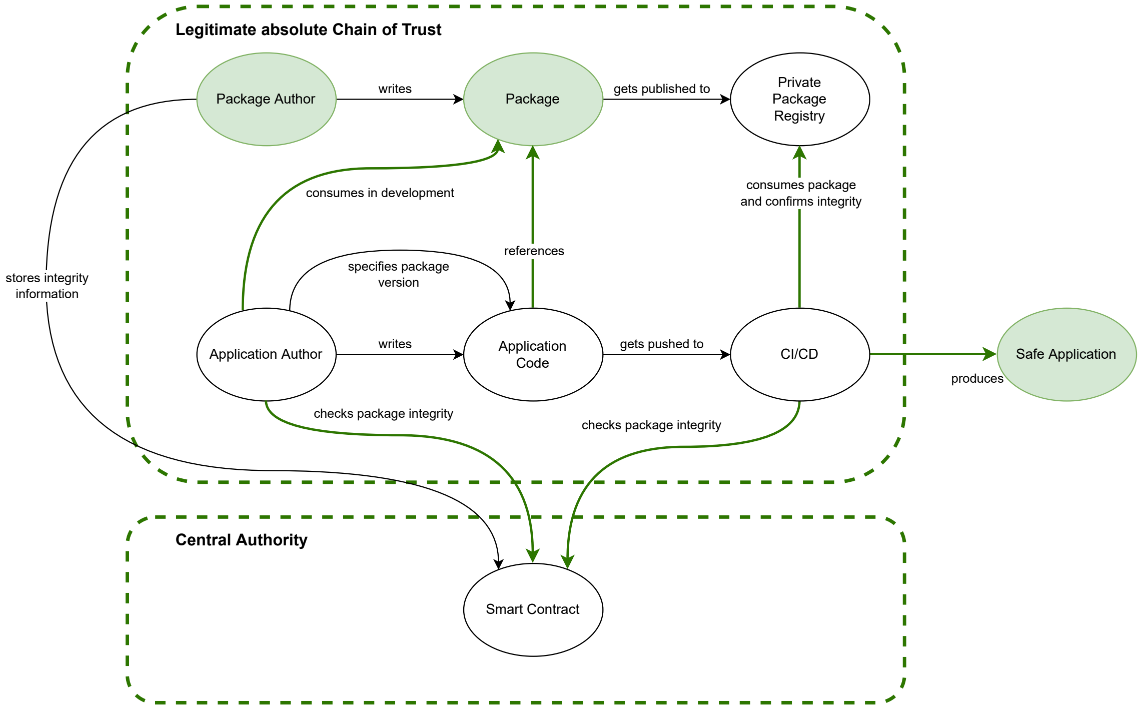


Figure 18: The absolute “Chain of Trust” for the consumption of a package

The proposed K9 System will include both the Central Authority in the Form of a smart contract and augmentations for the build process to enable the use of this smart contract.

Note: The view presented here is from a very high level of observation. The smart contract alone will not secure the whole supply chain, but augments existing security measures. The “distance travelled” of the code from the developer of both the package and the application needs to be secured as well. This is not accomplished by the smart contract, but instead (as shown in section 4.5.2) by existing security measures of the environment where the K9 Agent is to be deployed. These measures include securing the code repositories by leveraging integration between existing enterprise domain controllers with the IAM systems of the repository/cloud provider. In this way, scaling can be ensured and integration with existing systems is simplified. In the same vein, the key(s) used to access the smart contract need to be administered in some way as well. With the proposed deployment of the K9 Agent, this would be accomplished by storing the key(s) with existing cloud services and controlling access with the aforementioned IAM systems of the cloud provider, to ensure that CI/CD systems on the cloud platform can access the keys, while keeping them safe from unauthorized access. This pattern was also used to implement the K9 System's PoC in section 4.5.2. The "full" COT, including the auxiliary systems described here, can be found in the appendix (Figure 39, Figure 40).

Use Case: Publishing a package

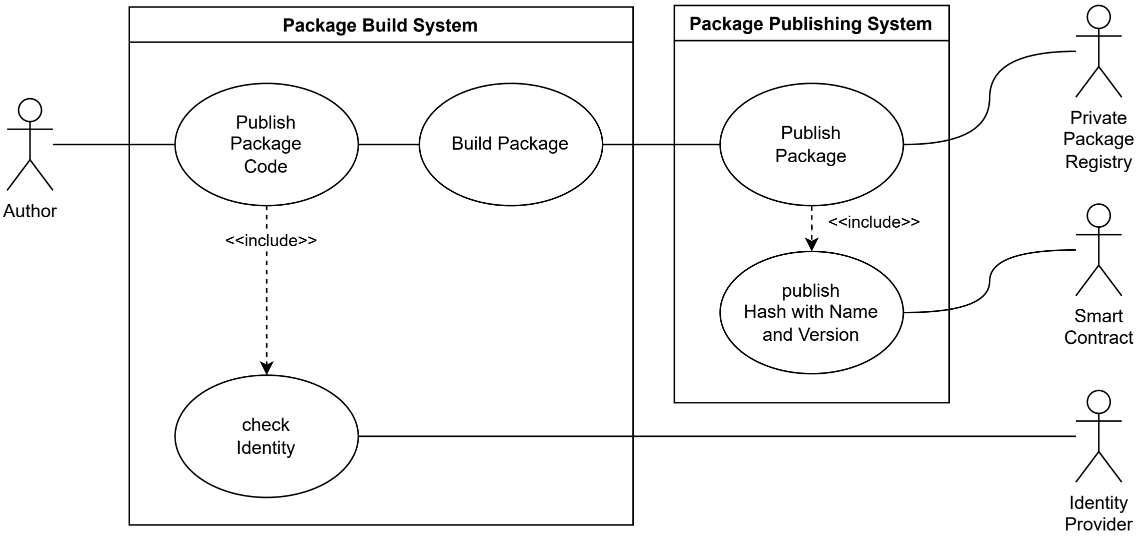


Figure 19: Use Case Diagram: Publishing a package

Functional requirements and requirements based on DCA characteristics are referenced in this use case, depending on which requirement is relevant to the part of the use case being described. Functional requirements are described in section 4.3, while requirements based on characteristics are described in section 4.2. The requirements are identified by their requirement IDs described in these sections.

In Figure 19, the use case of an author publishing a package has been detailed. There are several actors involved in this process:

1. **Author**: The author develops the package and makes changes to it. Once there are changes made that warrant the publishing of a new version of the package, the author commits these changes in a code repository, publishing the package code. To facilitate this, the author must be authenticated with the registry (**R-F1**), otherwise being barred from contributing code to it.

2. **Identity Provider**: The Identity Provider (IDP) can authenticate users based on credentials that are provided to him (**R-F1**). The IDP further can provide information about users, based on claims that are stored in his user registry. These claims can be used to authorize users to do certain things, based on groups or roles that can be applied to users(**R-F2**). In this case, the IDP ensures that users trying to commit code to the package registry belong to a group of users allowed to do so.

3. **Smart Contract**: The smart contract stores information on the blockchain. In this case, the information stored is information about package versions associated with SHA1s representing a hash of a tarball (**R-F4, R-F6, R-C3, R-C4**, see: section 2.2.2) of the files of the package[2]. This information can be provided to the smart contract by authorized user accounts, identified by a key pair.

4. **Private Package Registry**: The Private Package Registry contains the packages that are published by the package publishing system. These are stored and can be distributed by requesting them based on name and version of the package.

These actors correspond to the systems shown in the System Context diagram Figure 20.
The publishing of a package by an author thus includes the following steps:

1. **Publish Code**: The author publishes the code for a new version of the package to the code repository.

2. **Check Identity**: The Package Build System ensures that the author has the right to contribute code to the repository. If this is not the case, the commit is rejected. This step includes the use of an IDP (**R-F1**). This system can verify to the Package Build System that the author has the rights to contribute code to the repository and thus publish new package versions(**R-F2**).

3. **Build Package**: Once it has been verified that the author of the new code is legitimate and allowed to contribute, the build system takes the code and builds it into a new version of the package.

[2] SHA1 was chosen to be compatible with older NodeJS projects, SHA512 hashes are available in current versions of NodeJS, but not guaranteed, see section 2.2.2

4. **Publish Package**: Once the build system finishes building the package, the Package Publishing System publishes the package to the Private Package Registry. This involves packing the package files into a tarball (see: section 2.2.2) and generating a SHA1 over this compressed archive.

5. **Publish Hash, Name and Version**: The SHA1, together with the name and version of the package built in the last step, are published to the smart contract on the blockchain (**R-F4, R-F6**). This way, the package (identified by its name) and the relevant version are associated with the SHA1. This information is stored on the blockchain through the smart contract, ensuring maximum protection against manipulation, and through the architecture of the smart contract, making it available to everybody.

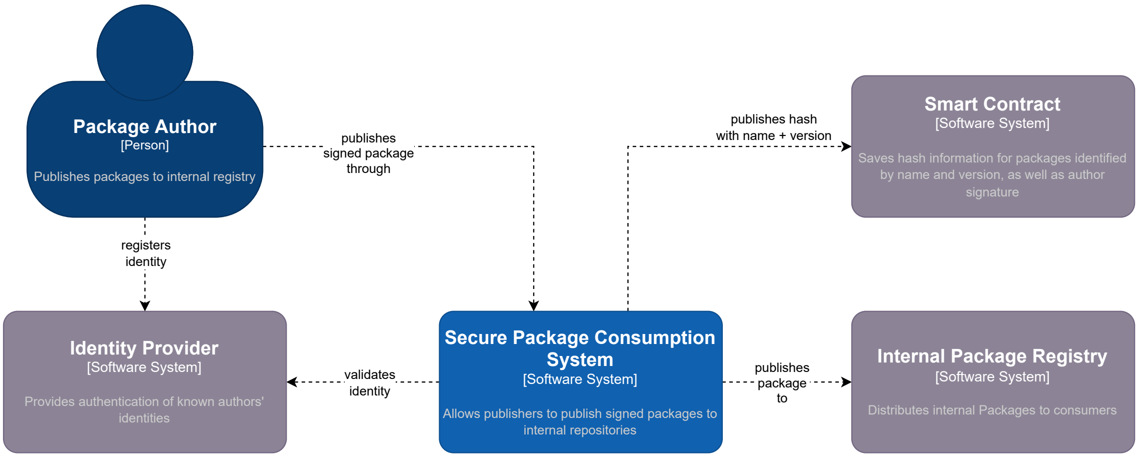

Package Author
[Person]
Publishes packages to internal registry
publishes signed package through
registers identity
Smart Contract
[Software System]
Saves hash information for packages identified by name and version, as well as author signature
publishes hash with name + version
Identity Provider
[Software System]
Provides authentication of known authors' identities
validates identity
Secure Package Consumption System
[Software System]
Allows publishers to publish signed packages to internal repositories
publishes package to
Internal Package Registry
[Software System]
Distributes internal Packages to consumers

Figure 20: C4 System Context Diagram: Publishing a package

Use Case: Consuming a package

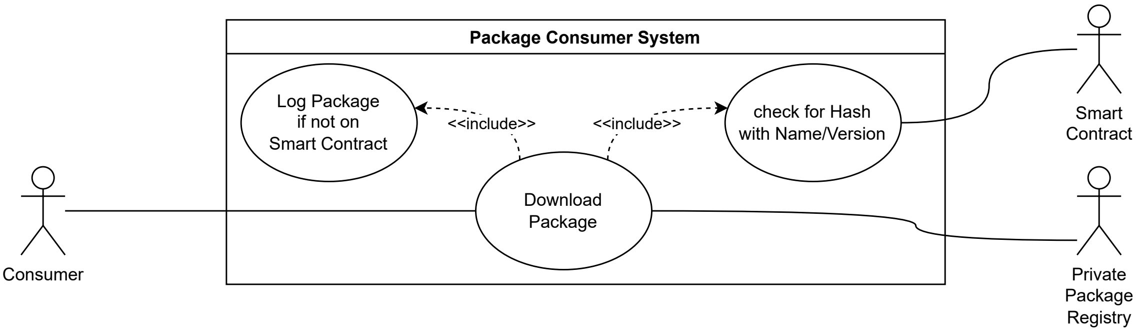


Figure 21: Use Case Diagram: Consuming a package

Functional requirements and requirements based on DCA characteristics are referenced in this use case, depending on which requirement is relevant to the part of the use case being described. Functional requirements are described in section 4.3, while requirements based on characteristics are described in section 4.2. The requirements are identified by their requirement IDs described in these sections.

In Figure 21, the use case of a consumer downloading an internal package has been drawn up. This process includes the following actors:

1. **Consumer**: The consumer downloads packages to use them in his own internal processes. In the context here, the consumer is a CI/CD system automatically pulling packages from the Private Package Registry.
2. **Smart Contract**: The smart contract stores information on the blockchain. In this case, the information stored is information about package versions associated with SHA1s representing a hash of a tarball (**R-F4, R-F6, R-C3, R-C4**, see: section 2.2.2) of the files of the package. The smart contract can provide this information to any user requesting it, only requiring the name and version of the package the information is requested about (**R-F5, R-C4, R-C5**).
3. **Private Package Registry**: The Private Package Registry contains the packages that are published by the package publishing system. These are stored and can be distributed by requesting them based on name and version of the package.

These actors correspond to the systems shown in the System Context diagram Figure 22.
The consumption of a package by a consumer thus includes the following steps:

1. **Download Package**: The Consumer possesses a package name and version and wishes to attain the files associated with this tuple. This step includes the steps “Check for Hash with Name/Version” and “Log Package if not on Smart Contract” and will only conclude successfully if both these steps do not fail. If a hash has been obtained from the smart contract, it will be checked against the hash of the downloaded files. If it matches, the process concludes successfully, if not, an impostor package has been found on the registry and an alarm is raised. If a package is not found on the Smart Contract, it is assumed external and logged for auditing (**R-F5, R-F7, R-C5**).

2. **Check for Hash with Name/Version**: The name and version of the package requested are presented to the smart contract. The smart contract responds with the SHA1 of the package version requested. If the package is not found by name on the smart contract (meaning it is an external package) the step "Log Package if not on Smart Contract" is invoked(**R-F5, R-F7, R-C3, R-C4**). If the version is not found (meaning a malicious version was included in the application), an alarm is raised and the step fails.

3. **Log Package if not on Smart Contract**: Names and Versions of packages not found by name on the smart contract are logged for auditing (**R-F7**). This enables insights into which packages were consumed from external registries and creates a "Software Bill of Materials (BOM)" that can be leveraged for auditing if any vulnerabilities of publically available packages become known.

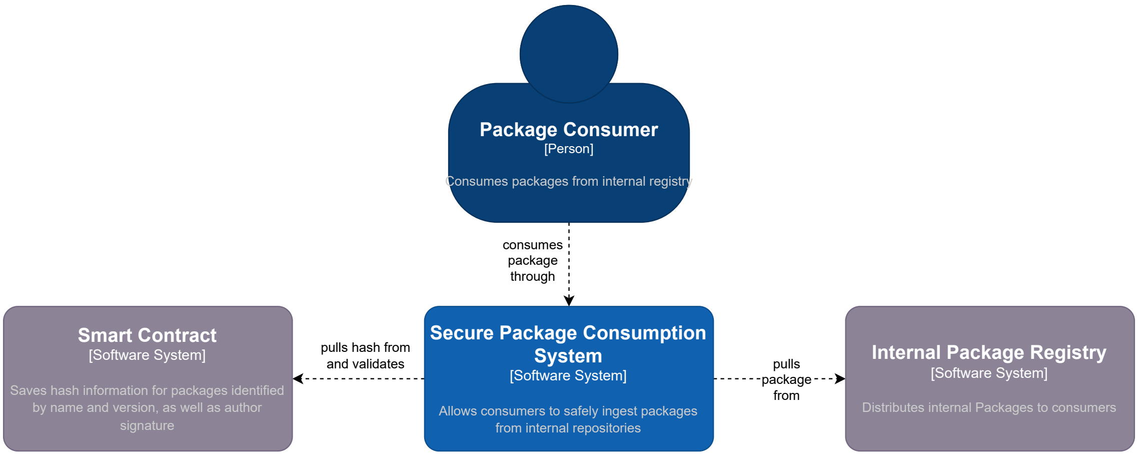

Package Consumer
[Person]
Consumes packages from internal registry
consumes package through
Smart Contract
[Software System]
Saves hash information for packages identified by name and version, as well as author signature
pulls hash from and validates
Secure Package Consumption System
[Software System]
Allows consumers to safely ingest packages from internal repositories
pulls package from
Internal Package Registry
[Software System]
Distributes internal Packages to consumers

Figure 22: C4 System Context Diagram: Pulling a package

4.5.2 Technical Architecture

Smart Contract

The Smart Contract constitutes the centerpiece of the technical architecture. It facilitates persisting data on the blockchain (see subsection 2.2.3), specifically Information about packages that are published to the internal package registry. This is stored in a three-tier Structure (compare Figure 23), consisting of a *Package List* containing all Packages available in the contract, indexed by the md5[3] hash of the package's name[4]. Every instance of such a *Package* includes the name of the package and a list of all available versions of the package. Thus, more than one package can have its versions stored with one smart contract. Each instance of a *Package Version* in this list then contains information pertaining to one published version of the package in the internal registry. This information includes (Table 9):

Table 9: Information stored on the Blockchain via Smart Contract

Identifier	Description
Package Author	the account address of the author/publisher of the package version
Package Name	The name of the package, identifying which package the version belongs to
Package Version	The version number of this particular package version, structured the same as the package version published to the internal registry, according to SemVer section 2.2.1
Package Hash	The SHA1 of the package tarball (compare section 2.2.2), as published to the internal registry

Further, the smart contract allows for a degree of authorization and authentication. The latter is provided via the blockchain networks integrated account system, based on a public-private-key architecture (compare subsection 2.2.3). This ensures that all direct users are part of the network. Authorization is more important in this case, as everybody in the network is allowed to request information from the smart

[3] An MD5 hash has been chosen due to its relatively low character count, which makes it fit into a fixed length data type variable in the smart contract code (Char32), increasing performance and lowering execution cost. Since this hash gets generated from the name and version of the package when inserting its integrity information and in order to retrieve said information, collisions are unlikely, since in both cases, the strings are known previous to being hashed.

[4] Indexing by a hash, whose length is known, reduces memory load and improves performance, which are directly related to cost on the blockchain, while preserving the ability to search ”by name” since the name can be hashed off-chain

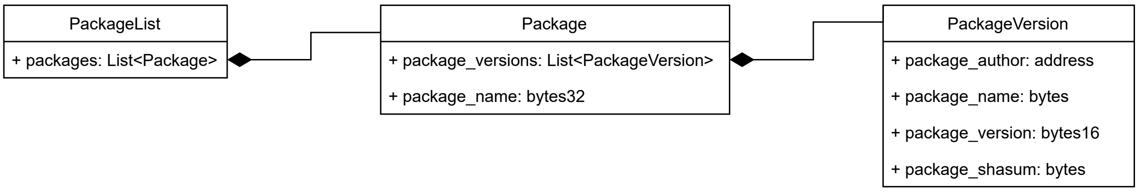

PackageList
+ packages: List<Package>
Package
+ package_versions: List<PackageVersion>
+ package_name: bytes32
PackageVersion
+ package_author: address
+ package_name: bytes
+ package_version: bytes16
+ package_shasum: bytes

Figure 23: Package versioning structure in K9 Smart Contract

contract, but not everybody is allowed to contribute information. To this end, two "user groups" have been established (Table 10):

Table 10: User groups in the K9 Smart Contract

User Group	Description
Owner	The "Administrator" of the smart contract, allowed to make changes and/or unpublish the contract. Only the owner can add and remove contributors to the contract. Additionally, the owner can transfer ownership to another account.
Contributor	Contributors are able to add new packages and package versions to the contract

In addition, the contract keeps a catalog of package names that can be returned to a client to quickly allow the client to ascertain if a given package is included in the contract's storage. In case of such a client request, the whole list is returned instead of individual requests, so that traffic is kept to a minimum, further reducing cost (any request to a smart contract will incur "gas fees", see subsection 2.2.3), since the system will try to check if any of a given manifests dependencies have information about them stored with the contract. Since these dependencies are commonly numbered in the thousands, a single request to the contract to filter out candidates that can then be individually checked is preferable to checking every dependency. To facilitate using the previously mentioned package name list, the package information storage functions, and the package information retrieval functions, the contract provides several methods:

K9 Agent

The K9 Security Agent was developed as a CLI program to be distributed as a compiled binary to the CI/CD environments via a shared, read-only, storage space in

Table 11: Methods of the K9 Smart Contract

Function	Description
storeHash	Stores a SHA1 of a package, together with information about the package. Takes the SHA1, package name, package version, MD5 hash of package name and version and MD5 hash of the package name as arguments. When executed successfully, returns the MD5 hash of name and version
getShasum	Returns package information (in the form of a Package Version (see Table 9)) for a specific version of a package. Takes an MD5 hash of the package name and an MD5 hash of the name and version as arguments.
getAllPackage-Names	Returns the list of all package names to enable the client to filter what packages can be checked against versions stored with the contract.

the form of an AWS S3 Bucket. This decision was reached because it makes it easy to control the distribution of the program, which has no dependencies to be installed with it, to the environments, while keeping the effort of replacing the program being distributed on release of a new version minimal in the development/evaluation phase. This includes the possibility of changing it to a more controllable, compliant process of distributing the tool with a prebuilt build environment when it has reached enough maturity to be used on a wider breadth of projects (compare the ADR in section A.2).
Furthermore, the K9 Security Agent is distributed as one tool that can both publish package version information to the smart contract, and request this information from the contract.

Publishing: In the publishing mode (compare Figure 24), the K9 Agent requires the build environment to build the package beforehand and publish it to the internal registry. Crucially, the output that npm produces in this step needs to be piped into a JSON-File and made available to the K9 Agent. By default, the system is set up to try and use any file named `package-info.json` in the current directory, but this can be overridden with the parameter `--package_info` which can be supplied with a different file name. Following this, the K9 Agent will extract the package name, version number and integrity SHA1 from this file, which represents the integrity of the tarball published to the internal registry. The K9 Agent, which will have to be supplied with the contract and account address in form of a (default, but overridable) `k9-config.json` file or as separate arguments, will then try and authenticate with the contract via a private key. This private key needs to be supplied to the K9 Agent when invoking the program, and cannot be loaded from a file. This is to encourage that the private key is kept in a secure system (in the case of the PoC implementation, AWS Secrets Manager), and not supplied to the build environment in an unsecured file. If the key can successfully authenticate the K9 Agent to the K9 Smart Contract, the package information is published to it. This requires the account the private key is associated with to either have owner or contributor privileges.

Note: In this iteration, the owner of the contract has all the rights of a contributor to the contract, plus administrative rights to the contract. In a mature version of this system, the owner would not have contributor rights, instead being relegated to only administrative duty, while contributors only interact with the contract on a package information publishing level. This would be preferable, as it separates the areas of control more clearly.

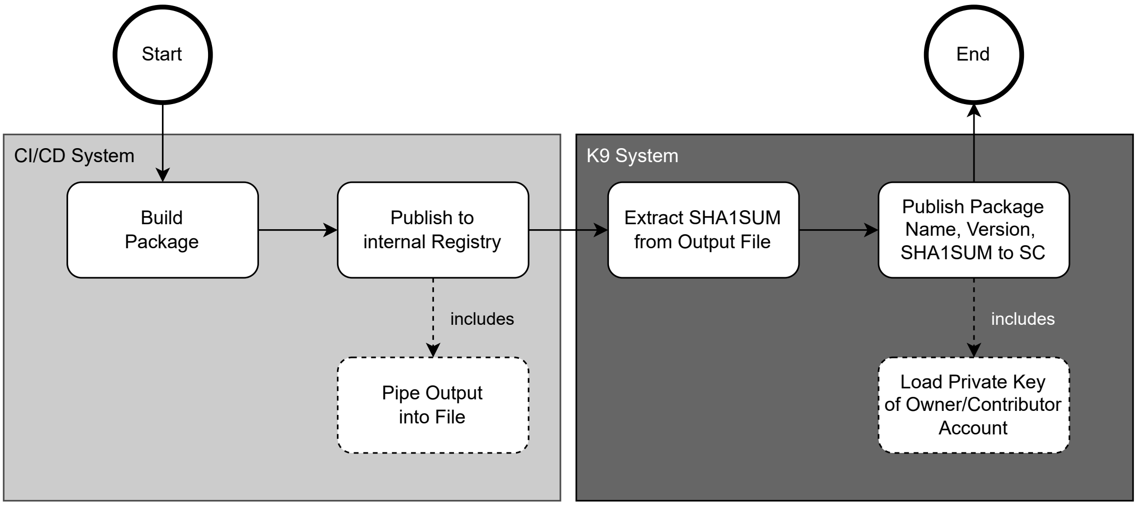

Start
End
CI/CD System
K9 System
Build Package
Publish to internal Registry
includes
Pipe Output into File
Extract SHA1SUM from Output File
Publish Package Name, Version, SHA1SUM to SC
includes
Load Private Key of Owner/Contributor Account

Figure 24: Flow: K9 Agent publishing a package

Consuming: In the consuming mode (compare Figure 25), the K9 Agent begins by scanning for a lockfile, either by looking for the default `package-lock.json` or, if supplied with the `--lockfile` argument, for the supplied file name. If a lockfile isn't found, the K9 Agent looks for a package manifest `package.json` and generates a warning, assuming the unsafe (for automated builds) `npm install` command to be used. The K9 Agent then gets a list of all packages of which information is stored on the smart contract by using the smart contract's `getAllPackageNames` functionality (see Table 11). This list is then used to filter the lockfile of the current application against packages that can be verified against information from the smart contract, implying them to be internally developed and distributed packages. The names and versions of all packages that cannot be verified against the smart contract are cataloged as well and formatted into a JSON-file for auditing or similar, effectively creating a software BOM for external packages. This list is then saved to the file system of the build environment. All internal packages discovered in the lockfile this way are flagged for verification. In the next step, all previously flagged packages are then checked against information about them from the smart contract. For each of these packages, a request is sent to the smart contract on its `getShasum` method, with the version of the package specified in the lockfile. The information returned from the smart contract will lead to either of three outcomes:

1. **Version not found on contract**: The version of the package specified in the lockfile has been published without storing it with the smart contract. This means a version was possibly published on a public repository with an inflated version number. Malicious activity is likely.
2. **Version found on contract, SHA1 mismatch**: The version of the package specified in the lockfile has been published and stored with the smart contract, but the package version content of the consumed package has been modified. Either a malicious version is being consumed from a public registry, or the package has been modified in transit. Malicious activity is likely.
3. **Version found on contract, SHA1 mismatch**: The version of the package specified in the lockfile has been published and stored with the smart contract. The consumed package's contents match the published ones. Malicious activity is unlikely.

If any of the first two possibilities occur, the relevant package will be flagged and an alarm raised. The K9 Agent will check all packages, but output an error to `stderr` and finish with an exit status of `1`, producing an error and breaking the build process.

CI/CD Environment

The architecture of the CI/CD environment is largely the same as the one characterized in section 4.4.1 for the vulnerable CI/CD environment. This is intentional, as the K9 Agent fulfills the general requirement to be easily integrable into existing

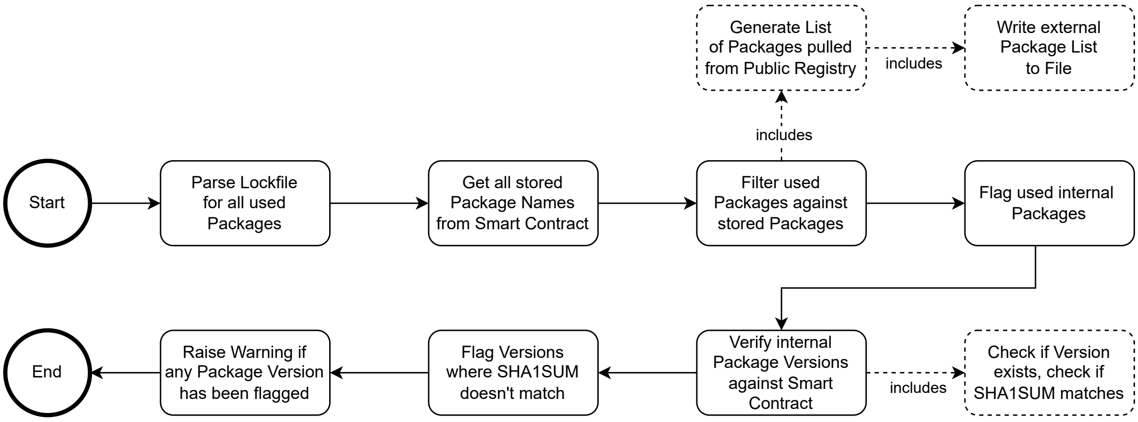

Generate List of Packages pulled from Public Registry
includes
Write external Package List to File
includes
Start
Parse Lockfile for all used Packages
Get all stored Package Names from Smart Contract
Filter used Packages against stored Packages
Flag used internal Packages
End
Raise Warning if any Package Version has been flagged
Flag Versions where SHA1SUM doesn't match
Verify internal Package Versions against Smart Contract
includes
Check if Version exists, check if SHA1SUM matches

Figure 25: Flow: K9 System consuming a package

infrastructure. Thus, only minimal changes to the build environment of the system are needed. This new CI/CD environment is shown in Figure 26. Note, that at this stage, the K9 Agent is distributed via an S3 Bucket, a shared storage space that allows all build environments in the AWS account to (read) access the S3 Bucket to download the K9 Agent binary. In the stage of development and evaluation, this style of distribution was chosen to ease deployment of new versions of the K9 Agent. At a more mature stage of the K9 System, a distribution via pre-built build images is preferred, since the use of a specific build image can be mandated and audited with cloud infrastructure security tools[5]. Compare Figure 41 in the appendix for that architecture.

Changes from the vulnerable architecture are comprised only changes to the buildspec in the build step of the pipeline (AWS CodeBuild in Figure 26). In the build step, several changes have been made to firstly procure the K9 Agent, and secondly access and retrieve the necessary private key to connect to the smart contract, and in the case of publishing, store information on the contract. The private key, host address (entry point for the smart contract interface) and contract address needed to communicate with the smart contract are stored with AWS Secrets Manager, and provided to the build environment via dynamically set environment variables (pointers that only produce values when accessed, storing the value encrypted in AWS Secrets Manager and only exposing it via Tokens that get filled with values at runtime and the values redacted from logs).

Publish Pipeline: In the package publishing pipeline, the buildspec specifies the

[5] For example, AWS Security Hub `https://aws.amazon.com/security-hub/`

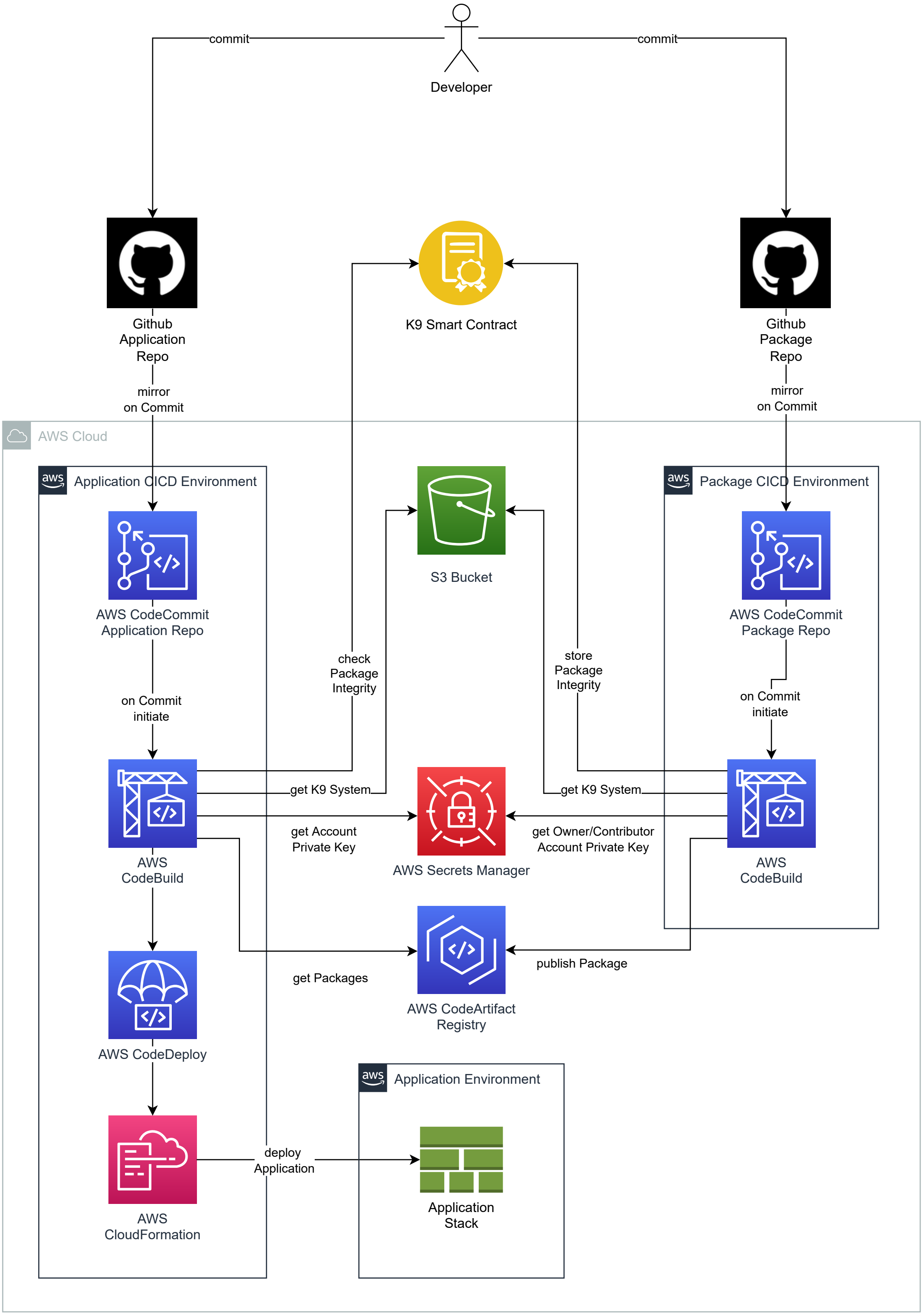


Figure 26: K9 System CI/CD architecture (development/evaluation stage)

following behavior:

1. Install AWS CDK tooling
2. Download the K9 Agent from the shared S3 Bucket
3. Log in to the internal registry
4. Run a check on all dependencies included with the package to be built and published with the K9 Agent
5. Install dependencies and build the Package
6. Publish the package to the internal registry and output resulting information into a JSON-file
7. Publish the information to the K9 Smart Contract using the credentials provided to the build step from AWS Secrets Manager

Note: In the implementation of the publishing step, a preliminary check with the K9 Agent is made against the dependencies of the package to be published. This is to ensure that no impostor packages are included with the package that would then be recorded as “clean” on the smart contract. This does represent a certain “hen-and-egg” problem for the ultimate “precursor” package, so great care must be taken when the first package ever is published with this system.

The buildspec for the *publish*-pipeline is shown in Listing 8 in section 6.2.1. In the listing, the use of tokens for private key, smart contract host (interface address) and contract address can be observed.

Consume Pipeline: In the package consumer/application build pipeline, the buildspec specifies the following behavior:

1. Install AWS CDK tooling
2. Download the K9 Agent from the shared S3 Bucket
3. Log in to the internal registry
4. Run a check on all dependencies included with the application to be built and published with the K9 Agent
5. Install dependencies and build the application

6. Synthesize the application with AWS CDK Tooling

If there are any irregularities detected by the K9 Agent in step 4, the resulting exit code `1` (see section 4.5.2) will cause the whole pipeline to fail, and produce a warning in the build logs for DevOps personnel to interpret the failure.

The buildspec for the *consume*-pipeline is shown in Listing 9.

```
npm install -g aws-cdk
aws codeartifact login --tool npm --repository master-registry --domain master-registry-domain --domain-owner 365161439830
./k9-linux checkall -p $K9_PRIV_KEY -h $K9_HOST_ADD -c $K9_CONT_ADD
npm ci
npm run build
npx cdk synth
```

Listing 5: Buildspec of K9 secured application pipeline

Importantly, the K9 Agent is used to scan dependencies before they are installed. This is achieved by using the `resolved` property (see section 2.2.2) of the lockfile. This URL can be used to infer the registries host address where the package is located (for example `https://registry.npmjs.org/`), which can then be used to access the package metadata API by querieing `<host>/<package_name>/<package_version>`, so for example `https://registry.npmjs.org/master-test-component/0.1.1`, to get the full package metadata as saved on the registry for a specific version of the package. The metadata includes the SHA1 of the package archive, which is then checked by the K9 Agent against the SHA1 stored with the K9 Smart Contract. The integrity of the package archive can thus be checked before downloading the package, ensuring that if a mismatch is detected, no packages with potentially malicious code are downloaded, since the build process would be stopped before proceeding to the installation step (`npm ci`).

5 Limitations and Constraints

When developing the K9 System, while solving the problems associated with preventing the DCA, concessions have to be made in order to facilitate producing a functional system that is realistically deployable and usable within the scope of this document. This produces limitations in regard to the system, both functionally and in terms of usability. In the following section, limitations and constraints that have been identified will be discussed, and pointers to possible solutions given, where possible.

5.1 Securing the Source Code

One constraint of the system, especially in regard to publishing internal packages, is that the "distance travelled" of the code that is to become a package published to the internal registry, is not secured in terms of the K9 System. In this regard, it is assumed, that the entity implementing the K9 System has put measures in place, which ensure that

- Code written by developers and committed to a repository is secured against manipulation on its way to the code repository; Meaning only code intended to by the developer reaches the code repository
- Repositories are secured against unauthorized access from parties other than the package authors; Meaning no code can be committed to the repository by parties other than the original package author

Both of these options would mean possible circumvention of the functionality of the K9 System by exploiting vulnerabilities in the underlying infrastructure that are not related to the DCA the K9 System protects against.
Both of these attack vectors could possibly be defended against by augmenting the K9 System to also validate the source code provided. This could be achieved by storing the developer's account/public key within the smart contract and using a

code signature generated with the developer's private key to validate the source code.

5.2 Administrating the Private Key(s)

Another constraint comes with the nature of the smart contract or blockchain ecosystem which the K9 System integrates into. To interact with the smart contract, a private key representing access to an account on the blockchain must be administered. This cannot be avoided when integrating with blockchain-based services. In the case of the prototype of the K9 System presented here, this key must be manually administered and provided to the system via command line argument. This also means, that personnel interacting with the CI/CD system might have access to the key. In general, the assumption is, that any organization integrating the K9 System would either

- Centralize control over the key to a (possible team of) key administrators; Since asymmetric key pairs are used many software development (-adjacent) services, it is likely that there is a role defined within the organization for administrating them
- Give full control of the CI/CD process for the respective package to the developer of the package, including administrating the private key (DevOps). This way, signing of the code as described in section 5.1 could be accomplished with the same key; To implement this, the K9 Smart Contract's "Contributor List" (see section 4.5.2) could be used

5.3 Using SHA1 Hashes

An important limitation of the K9 System, as described in section 4.5.1, is the use of only SHA1 hashes for integrity checking. This limitation is in place, since npm offers the feature of either using SHA1 or SHA512 hashes in the lockfile, but does not guarantee for the SHA512 hashes to be available on older lockfiles, and on certain platforms[1]. SHA1 has been shown to be not hash collision proof[2] and is mostly deprecated. A necessary feature for future versions of the K9 System would therefore be the automated detection of available SHA512 hashes, and using them

[1] `https://github.com/npm/npm/issues/16938`

[2] `https://shattered.io/`

instead of the SHA1 hashes where possible. This feature has been omitted from the PoC version due to time constraints.

5.4 Using a (public) Blockchain

Less specifically than described in section 5.2, the choice of using a public blockchain in this implementation of the K9 System also brings with it some constraints and possible risks. For one, every transaction, for example deploying the K9 Smart Contract or storing a version, has to be paid for in the associated blockchain's cryptocurrency. The blockchain used to implement the prototype is the Ethereum blockchain, which between January 2020 and May 2022 showed price fluctuations of up to 341%[3]. The choice was motivated (also compare the ADR section A.1) by the need to use a database or equivalent storage system that is highly resistant to manipulation and easily accessible as well as highly available. A distributed database - which the blockchain can be described as - fits this requirement. The choice of the public Ethereum blockchain was made based on the fact that a distributed database is labor-intensive to set up and maintain, and would have been impossible to also fit in the timeframe allotted to the development of the system in the scope if this document.

If the K9 System were to be deployed in an enterprise context where relying on cryptocurrency to deploy the contract and store package information within it is not an option, alternatives to consider would be managed services such as AWS Managed Blockchain[4] or Amazon Quantum Ledger Database[5] which integrate into existing solutions on, in this example, AWS, without involving the organization in cryptocurrency speculation.

5.5 Using a Blockchain Access Provider

In the implementation of the K9 System presented here, access to functions of the blockchain (including the smart contract) was mediated using a blockchain access provider. This access provider enables interaction with the blockchain without hosting one's own node in the network, instead providing access via HTTPS-based interfaces. In the scope of the PoC this method was used to simplify access to the

[3] Rolling 10-day realized volatility, benchmarked against the S&P 500; taken from `portfolioslab.com` through Statista, `https://www.statista.com/statistics/1278411/ethereum-price-swings/`, last accessed July 4th, 2022

[4] `https://aws.amazon.com/managed-blockchain`

[5] `https://aws.amazon.com/qldb/`

Ethereum Blockchain with the access provider "Infura"[6]. This carries the risk of relying on a third party to mediate access to the blockchain, introducing an additional point of failure. This risk was taken, instead of hosting an own node, for reasons of both restricted monetary and temporal budget in development[7].

In an actual, enterprise level deployment, similar measures to the ones described in section 5.4 might be taken, or the decision to host a full Ethereum node might be made, to mitigate this risk.

5.6 Simplifying OSINT for DCA

The last limitation of the system as implemented is the increased attack surface publically (on the blockchain) distributing the names of internal packages might bring. Discovery of these names is an important part of preparing a DCA (compare section 2.4), meaning public disclosure would bring the risk of malicious actors being able to more easily target these packages.

This is a realistic risk, but at the same time, it is also possible to monitor public registries for these names automatically (since the package names are publically known) and enact defensive measures accordingly, such as sending takedown notices to the involved parties.

[6] `https://infura.io/`

[7] See requirements for hosting an Ethereum node here: `https://ethereum.org/en/developers/docs/nodes-and-clients/run-a-node/`

6 Evaluating the System by Proof of Concept

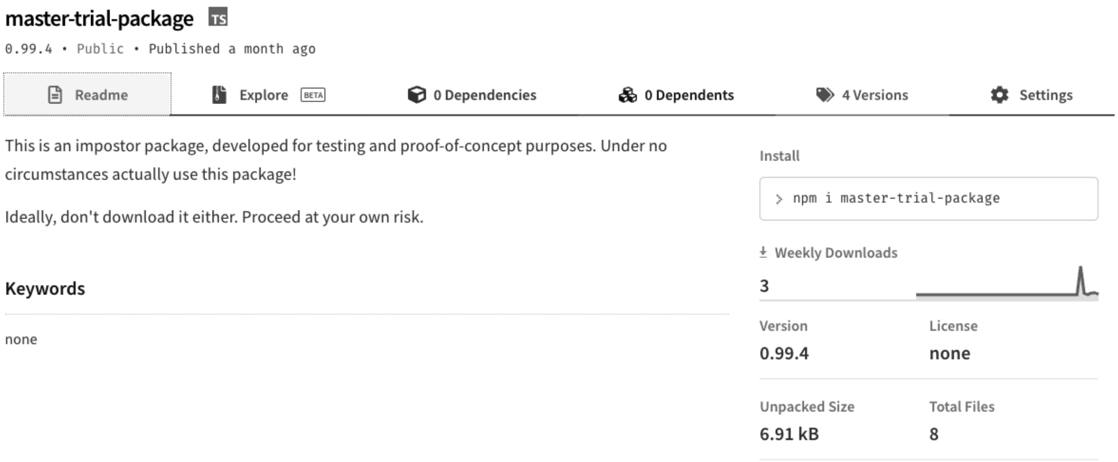
6.1 Proof of Concept Attack on vulnerable Cloud-CI/CD-System

6.1.1 Attack Setup

Malicious Impostor Package

To facilitate the attack, the first step was to prepare a malicious package to distribute and inject into the vulnerable Continuous Integration/Continuous Deployment (CI/CD) system. The package was derived from the original package and modified. The modification consists of an additional script in the “scripts” section of the manifest (see section 2.2.2). In this case, the “postInstall” script was modified, such that on successful consumption of the package by the build system, an output to the *stdout* of the build system is triggered. The successful attack can thus be validated by looking at the build logs of the system (the “scripts” section of the manifest is explored in section 2.2.2). Actual malicious behavior was not implemented, since the proposed system is not intended to scan for malicious behavior, and thus the risks and potential legal problems with providing known malicious software on public platforms are avoided by only including a warning in the build system, preventing harm to unsuspecting users accidentally consuming the package.
Additionally, the manifest file had its version updated, producing one package with the same version as the legitimate package in the internal registry (*0.1.1*), several *PATCH* versions (*0.1.2*, *0.1.3*) and one with a significantly higher *MINOR* version (compare section 2.2.1), *0.99.4*. These packages were then uploaded as public packages to the public node package manager (npm) registry[1]. Additionally, a notice was provided in the description of the package to prevent any potential unsuspecting victims accidentally downloading it, since the attack can only be realistically simulated with a package available to everybody using the public npm registry (compare Figure 27).

[1] `https://www.npmjs.com/package/master-trial-package`



master-trial-package

0.99.4 • Public • Published a month ago

This is an impostor package, developed for testing and proof-of-concept purposes. Under no circumstances actually use this package!

Ideally, don't download it either. Proceed at your own risk.

Keywords

none

Install

```
> npm i master-trial-package
```

Weekly Downloads

3

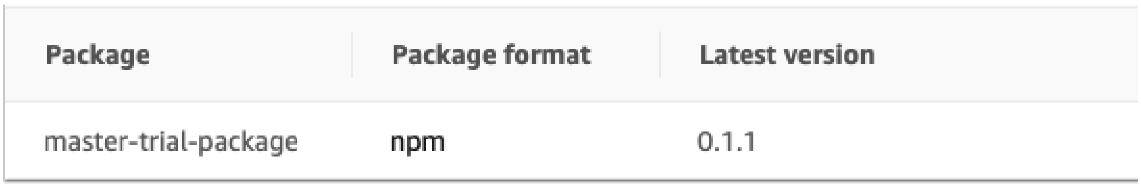
Version	License
0.99.4	none

Unpacked Size	Total Files
6.91 kB	8

Figure 27: Description and statistics for package used in attack on `npmjs.com`

This malicious package contains an AWS CDK compatible construct, which is exported to be used with an AWS CDK TypeScript Application. The construct contains an AWS Lambda Function returning a message that is different to the legitimate packages lambda function ("Hello! You've hit <request path>... But the evil one!" vs. the original's "Hello! You've hit <request path>!"), as well as outputting a message after installation of the package.

The legitimate package is still available on the private registry in its original version, as pictured in Figure 28



Package	Package format	Latest version
master-trial-package	npm	0.1.1

Figure 28: Legitimate package in private registry

Vulnerable Application

The vulnerable application was prepared by developing an example application based on AWS CDK, which deploys an AWS Lambda Function. This Lambda Function is imported from the construct of the Malicious Impostor Package. The architecture and technologies of the vulnerable application are described in detail in section 4.4.1. This Application is then deployed into the vulnerable CI/CD environment described in section 4.4.1.

The application's package manifest is provided in Listing 6 (development dependencies and scripts have been omitted since they are not relevant to the attack on the vulnerable application's side).

```
{
  "name": "master-code",
  "version": "0.1.0",
  "bin": {
   "master-code": "bin/master-code.js"
  },
  "scripts": {
   "build": "tsc",
   "watch": "tsc -w",
   "cdk": "cdk"
  },
  "devDependencies": {
    [...]
  },
  "dependencies": {
   "aws-cdk-lib": "2.23.0",
   "constructs": "^10.0.0",
   "master-trial-package": "^0.1.1",
   "source-map-support": "^0.5.21"
  }
 }
```

Listing 6: Vulnerable Application's Package Manifest (`package.json`)

Note that in this case, installing the vulnerable dependency `master-test-component` in the development environment by the author of the vulnerable application has automatically set it to install newer *MINOR* and *PATCH* versions (see section 2.2.1).

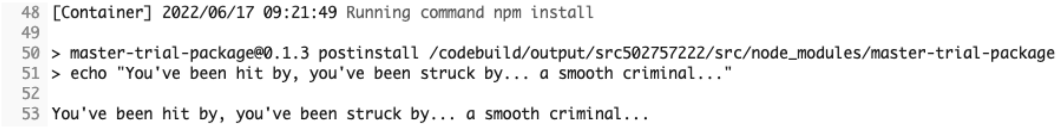
6.1.2 Attack Execution - Naïve Pipeline

With the vulnerable application now listing the dependency in its package manifest, the scenario begins with the developer committing the new version of the application into the CI/CD pipeline via the application repository. To accomplish this, the developer has to be authenticated with the repository. With this Proof of Concept (PoC), AWS CodeCommit is used, which requires authentication and authorization via AWS Identity and Access Management (IAM)[2]. In an enterprise context, this would be accomplished by integrating AWS IAM with the enterprise domain services, for example by mapping Active Directory identities to AWS IAM Roles. In this case,

[2] `https://aws.amazon.com/de/iam/`

the developer might log into the AWS Console with his enterprise Active Directory credentials, and be assigned the AWS IAM role of “CODE_CONTRIBUTOR”, which would allow him to push code to repositories with policies in place to allow proprietors of this role to do so.

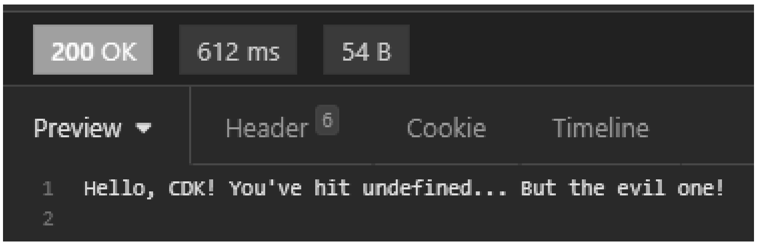
After the repository acknowledges receipt of the new commit, the sourcing and build processes of the vulnerable CI/CD pipeline (see section 4.4.1) will trigger. In the build step, after the installation step, the first sign of a successful attack is the message output by the compromised packages’ postInstall-script (see Figure 29).



```
[Container] 2022/06/17 09:21:49 Running command npm install

> master-trial-package@0.1.3 postinstall /codebuild/output/src502757222/src/node_modules/master-trial-package
> echo "You've been hit by, you've been struck by... a smooth criminal..."

You've been hit by, you've been struck by... a smooth criminal...
```

Figure 29: Message from the impostor package in build step

After the CI/CD pipeline has finished its run, the lambda function contained in the malicious package has been deployed with the application, as can be seen by the response of the lambda function when triggered by an HTTP request (Figure 30)



```
Hello, CDK! You've hit undefined... But the evil one!
```

Figure 30: HTTPS response of the lambda function from the impostor package once deployed via naïve pipeline

At this point, the application has fallen victim to the Dependency Confusion Attack (DCA). Both the build environment and the actual application have been infiltrated by malicious code. At this point, it should be noted that this version of the attack could have also been prevented by setting up the build environment in a specific way. As written in the pipeline, this version of the attack assumes a naïve pipeline to succeed. This means that the pipeline uses a standard `npm install` command instead of the more appropriate `npm ci` (compare Listing 4). Using the latter would have allowed npm itself to use the lockfile to ascertain if the packages pulled from the registry are the same ones mentioned in the lockfile (the lockfile also contains integrity information, compare section 2.2.2). However, this does not diminish the danger of

DCAs. For one, `npm ci` has been added to npm in 2018[3], which means some build pipelines released prior will still not implement `npm ci` (Anecdotal evidence from enterprise DevOps substantiates this claim), or it is not used since lockfiles can be incompatible between versions of npm[4]. The latter point is a security issue in and of itself, but manifests itself in real world DevOps fairly often[5].
On the other hand, another channel for DCAs to execute successfully is via the developer. If the lockfile already contains the checksum of the malicious version, no deviation will be detectable. This attack vector will be explored in the following section.

6.1.3 Attack Execution - Naïve Developer

As previously discussed, one other attack vector for the DCA to execute successfully is the infiltration by ways of the developer already pulling the package from the wrong source. In this case, either:

1. The developer is not logged in to the private registry
2. The package manager gets confused between packages of the same version on different registries and chooses the malicious one
3. The package manager is configured to choose higher versions and pulls a malicious package with a higher version than is actually available from private registries

Either of these options leads to the impostor package being present both in the package manifest and the package lockfile. This second scenario then continues the same as the first one, with the developer committing this "infected" manifest and lockfile into the code repository. The repository is still secured to only accept code from a specific group identified by roles. In this case, compared to the unsecured pipeline of the attack shown prior, the build step has been changed to incorporate the `npm ci` command[6]. This slightly changed buildstep is shown in Listing 7.

[3] `https://github.com/npm/npm/releases/tag/v5.7.1`

[4] `https://github.blog/2021-02-02-npm-7-is-now-generally-available/#changes-to-the-lockfile`

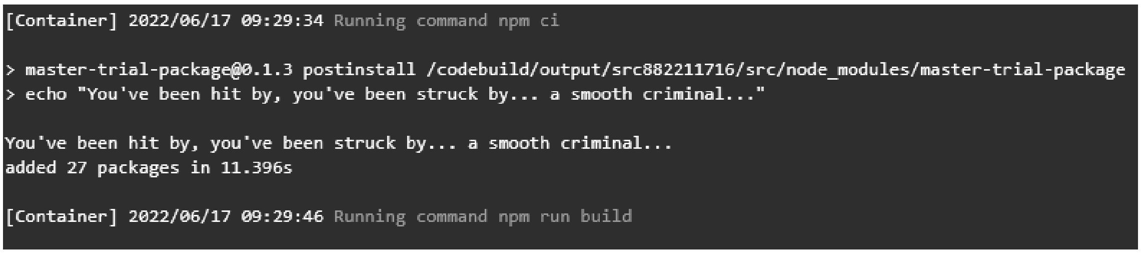
[5] Stack Overflow shows 77 Questions for "npm WARN old lockfile", as well as 155k views for the most popular question for the topic, `https://stackoverflow.com/search?q=npm+WARN+old+lockfile`, last accessed 2022-06-14

[6] `https://docs.npmjs.com/cli/v8/commands/npm-ci`

```
npm install -g aws-cdk
aws codeartifact login --tool npm --repository master-registry --domain master-
    registry-domain --domain-owner 365161439830
npm ci
npm run build
npx cdk synth
```

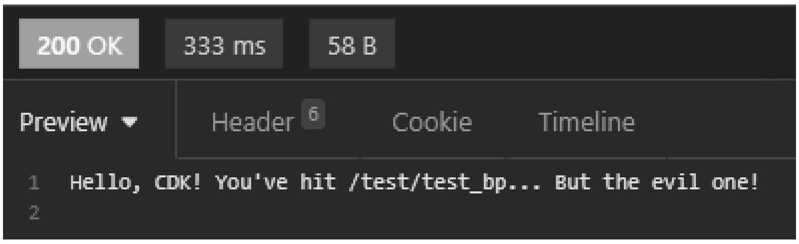
Listing 7: Buildspec of Vulnerable Application with `npm ci`

The sourcing and build process will run their regular course, until at the installation stage, the payload of the malicious package is downloaded with the malicious package itself and executed (see Figure 31). This happens despite npm running an integrity check on the package, since the malicious package has had its integrity information saved to the lockfile already. As expected, the application utilizing the package has been infected as well (compare Figure 32 for the lambda function HTTP response).



```
[Container] 2022/06/17 09:29:34 Running command npm ci

> master-trial-package@0.1.3 postinstall /codebuild/output/src882211716/src/node_modules/master-trial-package
> echo "You've been hit by, you've been struck by... a smooth criminal..."

You've been hit by, you've been struck by... a smooth criminal...
added 27 packages in 11.396s

[Container] 2022/06/17 09:29:46 Running command npm run build
```

Figure 31: Message from the impostor package in build step of best-practice pipeline



```
Hello, CDK! You've hit /test/test_bp... But the evil one!
```

Figure 32: HTTPS response of the lambda function from the impostor package once deployed via best-practice pipeline

6.2 Proof of Concept Attack on hardened Cloud-CI/CD-System

6.2.1 Attack Setup Hardened Pipeline

Publishing a Verified Package Version

To facilitate a secure consumption of the internal package, this package first has to be published in the way outlined in section 4.5.1. The package published as a verified version contains the same code as the vulnerable package published in the Attack PoC that was impersonated by the impostor package (see subsubsection 6.1.1).

In the hardened process, this package is published through a pipeline hardened by the K9 System. In the publishing step, the K9 Agent ensures, that the packages' integrity information is stored with the smart contract, and associated there with the version and package name of the package being published.

The hardened process assumes that the code of the package has a secure way of being delivered to the code repository. The assumption is further, that this security exists in the form of a domain controller, for example an Active Directory service, and authenticating, as well as authorizing, contributors to the repository by integrating with the repository's own cloud IAM system. In this way, it can be assumed that the code will be delivered with unchanging integrity to the code repository. This will also ensure that the chain of trust (see section 4.5.1) is unbroken. At this point, the build step triggers. The build step will bridge the chain of trust to a new authority, the K9 Smart Contract. To facilitate this, the K9 Agent is used in conjunction with a private key that authorizes use of the K9 Smart Contract. The private key itself is also secured with IAM policies, authorizing the CI/CD system containing the build step to retrieve it. In this way, the build step gets code delivered that is guaranteed by the domain controller in conjunction with the cloud IAM system to be the code the author (authenticated with the domain controller) has committed. This code is then built into a package, the integrity information of which is then published to the K9 Smart Contract, using the private key. In this way, the integrity information on the contract can trace a chain of trust back to the original author.

As the K9 Smart Contract has a very high manipulation resistance due to its blockchain-bound nature (see subsection 2.2.3), the integrity information it delivers can be assumed to be true, since the chain of trust up until delivery of the information to the contract has been ensured.

This secure publishing of the package has been implemented within the same infrastructure as the unsecured publishing of the vulnerable package published in the

Attack PoC (see subsubsection 6.1.1).
The key differences in the secure implementations are:

1. Providing the K9 Agent to the package CI/CD pipeline with a shared storage medium (e.g. AWS S3 Bucket)

2. Providing the Private Key and several other pieces of information to the CI/CD environment to use the K9 Agent via a secure interface, protected by IAM policies (e.g. AWS SecretsManager)

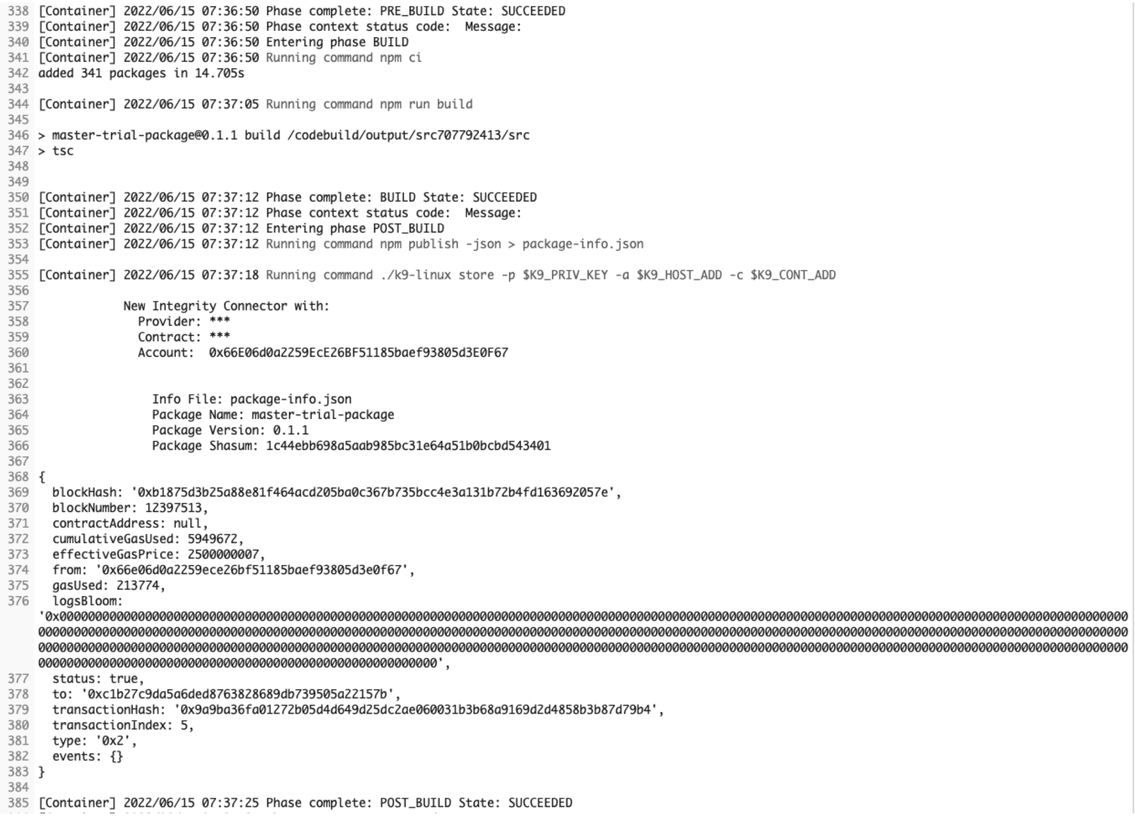
3. Using the K9 Agent with the private key to publish the integrity information

The buildspec for this so modified CI/CD build step is shown in Listing 8. Note, that besides publishing integrity information, the K9 Agent is also used to evaluate the dependencies of the package to be published, to ensure no integrity information of infected packages is stored as “trusted” with the K9 Smart Contract.

```
npm install -g aws-cdk
aws s3 sync s3://k9secured-exe-bucket .
chmod +x ./k9-linux
aws codeartifact get-authorization-token --domain master-registry-domain --domain-
    owner 365161439830 --region eu-west-1 --query authorizationToken --output text >
     registry_token
./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --auth_prefix
    $K9_AUTH_PRE --auth_suffix $K9_AUTH_SUF --registry_prefix $K9_REGI_PRE
aws codeartifact login --tool npm --repository master-registry --domain master-
    registry-domain --domain-owner 365161439830
npm ci
npm run build
npm publish -json > package_info.json
./k9-linux store -p $K9_PRIV_KEY -h $K9_HOST_ADD -c $K9_CONT_ADD
```

Listing 8: Buildspec of K9 secured publishing pipeline

Once these steps are completed, the package information is safely stored with the K9 Smart Contract and thus the version established as a confirmed package version, with associated integrity information. A successful run of this publishing action can be seen in Figure 33. Note, that sensitive information consumed via the secure interface (AWS SecretsManager), such as the private key, has been automatically redacted from the log.



```
[Container] 2022/06/15 07:36:50 Phase complete: PRE_BUILD State: SUCCEEDED
[Container] 2022/06/15 07:36:50 Phase context status code:  Message:
[Container] 2022/06/15 07:36:50 Entering phase BUILD
[Container] 2022/06/15 07:36:50 Running command npm ci
added 341 packages in 14.705s

[Container] 2022/06/15 07:37:05 Running command npm run build

> master-trial-package@0.1.1 build /codebuild/output/src707792413/src
> tsc


[Container] 2022/06/15 07:37:12 Phase complete: BUILD State: SUCCEEDED
[Container] 2022/06/15 07:37:12 Phase context status code:  Message:
[Container] 2022/06/15 07:37:12 Entering phase POST_BUILD
[Container] 2022/06/15 07:37:12 Running command npm publish -json > package-info.json

[Container] 2022/06/15 07:37:18 Running command ./k9-linux store -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD

            New Integrity Connector with:
              Provider: ***
              Contract: ***
              Account:  0x66E06d0a2259EcE26BF51185baef93805d3E0F67


                Info File: package-info.json
                Package Name: master-trial-package
                Package Version: 0.1.1
                Package Shasum: 1c44ebb698a5aab985bc31e64a51b0bcbd543401

{
  blockHash: '0xb1875d3b25a88e81f464acd205ba0c367b735bcc4e3a131b72b4fd163692057e',
  blockNumber: 12397513,
  contractAddress: null,
  cumulativeGasUsed: 5949672,
  effectiveGasPrice: 2500000007,
  from: '0x66e06d0a2259ece26bf51185baef93805d3e0f67',
  gasUsed: 213774,
  logsBloom:
'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000',
  status: true,
  to: '0xc1b27c9da5a6ded8763828689db739505a22157b',
  transactionHash: '0x9a9ba36fa01272b05d4d649d25dc2ae060031b3b68a9169d2d4858b3b87d79b4',
  transactionIndex: 5,
  type: '0x2',
  events: {}
}

[Container] 2022/06/15 07:37:25 Phase complete: POST_BUILD State: SUCCEEDED
```

Figure 33: K9 Agent successfully publishes package version

Including K9 Agent in Application CI/CD Pipeline

With the package publishing pipeline augmented with the K9 System, implementation in the consuming application pipeline is necessary to ensure full protection. The Application code itself is still the same as used in the vulnerable application (see section 4.4.1). The same is applicable for the general layout of the CI/CD pipeline. In order to easily integrate the K9 System in the real world, the modifications to existing CI/CD systems need to be as minimal as possible (**R-F3**, see Table 7). The same assumptions for authentication and authorization as described in section 6.2.1 are made, in order to confirm to **R-F1** and **R-F2**. Through these, an uninterrupted chain of trust can be assumed up until the moment of building the project while consuming packages from internal and external registries. Here, the chain of trust is augmented by use of the K9 Agent, by including integrity information from the K9 Smart Contract to ensure a safe build (considering in this case only internal packages that might be victims to impostors)
The build step will include first checking the available application information, in the form of a lockfile, for known packages, which are assumed to be internal (compare **R-C1, R-C2**, see Table 6). This is accomplished by generating a delta from a list of packages stored on the smart contract (obtained from the same) with the packages in the lockfile. These internal packages are then tested for official availability of their version, and if yes, for their integrity. If all these steps can be completed successfully, the build step will continue as normal.
This secure building of the application has been implemented within the same infrastructure The key differences in the secure implementations are:

1. Providing the K9 Agent to the application CI/CD pipeline with a shared storage medium (e.g. AWS S3 Bucket)

2. Providing the Private Key and several other pieces of Information to the CI/CD environment to use the K9 Agent via a secure interface, protected by IAM policies (e.g. AWS SecretsManager)

3. Using the K9 Agent with the private key to verify the integrity of the consumed packages

4. Providing a way to exfiltrate the software Bill of Materials (BOM) of external packages used into a shared storage medium (e.g. AWS S3 Bucket)

Note: In the case of checking packages instead of publishing them, a private key is used as well. This is necessary, since an account/wallet is needed to interact with the smart contract on the blockchain. In this case, the same private key as used in the publishing step is used. Incidentally, this is a key that belongs to the "administrative" account with regard to the contract. This is **not** a necessity. The account used for obtaining information **from** the contract can be any account associated with the Ethereum blockchain. In this case, an existing account was used for reasons of development and evaluation speed.

The buildspec for this so modified CI/CD build step is shown in Listing 9. Note the "external-packages-compliance-list.json" being copied out of the build environment into a separate S3 Bucket. This is done to allow auditing of the list of external packages consumed (including their respective versions) at a later point in time.

```
npm update -g npm
npm install -g aws-cdk
aws s3 sync s3://k9secured-exe-bucket .
chmod +x ./k9-linux
aws codeartifact get-authorization-token --domain master-registry-domain --domain-owner 365161439830 --region eu-west-1 --query authorizationToken --output text > registry-token',
./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --auth_prefix $K9_AUTH_PRE --auth_suffix $K9_AUTH_SUF --registry_prefix $K9_REGI_PRE
aws s3 cp external-packages-compliance-list.json s3://k9secured-log-bucket
aws codeartifact login --tool npm --repository master-registry --domain master-registry-domain --domain-owner 365161439830
npm ci
npm run build
npx cdk synth --quiet
```

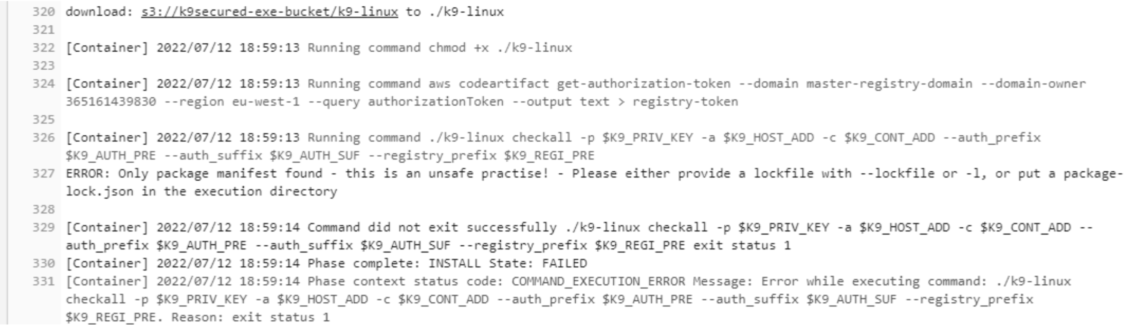
Listing 9: Buildspec of K9 secured application pipeline

6.2.2 Attack Execution - Naïve Pipeline

The first attack pattern, introduced and successfully executed in subsection 6.1.2, relied on the CI/CD pipeline being used without a lockfile. This attack will be tested with the K9 System in use. The prerequisites are the same, the pipeline operating with the not recommended subsubsection 2.5.2 `npm install` command during package installation, and no lockfile being present. This means that no package version or source is fixed. This setup - using a `npm install` command in a CI/CD system - is not commonly not recommended as a best practice[7].

[7] https://snyk.io/blog/ten-npm-security-best-practices/

For this case, the K9 Agent, requiring a lockfile to accurately determine package versions and sources, enforces this best practice. In case only a package manifest (`package.json`) is present, the system operates under the assumption, that `npm ci` is not used in the pipeline. In this case, the K9 Agent will stop the build and produce an error recommending the use of a lockfile (compare Figure 34).



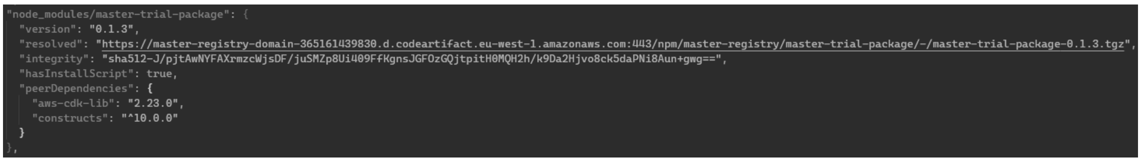
```
download: s3://k9secured-exe-bucket/k9-linux to ./k9-linux

[Container] 2022/07/12 18:59:13 Running command chmod +x ./k9-linux

[Container] 2022/07/12 18:59:13 Running command aws codeartifact get-authorization-token --domain master-registry-domain --domain-owner
365161439830 --region eu-west-1 --query authorizationToken --output text > registry-token

[Container] 2022/07/12 18:59:13 Running command ./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --auth_prefix
$K9_AUTH_PRE --auth_suffix $K9_AUTH_SUF --registry_prefix $K9_REGI_PRE
ERROR: Only package manifest found - this is an unsafe practise! - Please either provide a lockfile with --lockfile or -l, or put a package-
lock.json in the execution directory

[Container] 2022/07/12 18:59:14 Command did not exit successfully ./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --
auth_prefix $K9_AUTH_PRE --auth_suffix $K9_AUTH_SUF --registry_prefix $K9_REGI_PRE exit status 1
[Container] 2022/07/12 18:59:14 Phase complete: INSTALL State: FAILED
[Container] 2022/07/12 18:59:14 Phase context status code: COMMAND_EXECUTION_ERROR Message: Error while executing command: ./k9-linux
checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --auth_prefix $K9_AUTH_PRE --auth_suffix $K9_AUTH_SUF --registry_prefix
$K9_REGI_PRE. Reason: exit status 1
```

Figure 34: The warning generated when using the `npm install` command in a CI/CD environment, stopped by the K9 Agent

In this case, the K9 Agent did not attempt to check the packages in the lockfile, since no lockfile was provided. The usage of the lockfile is enforced as part of the security concept, ensuring fixed versions specified to enable checking them with the K9 Smart Contract. The attack in subsection 6.1.2 can thus be avoided.

6.2.3 Attack Execution - Naïve Developer

The second attack pattern, proven effective against an unsecured CI/CD environment in subsection 6.1.3, is being tested against the pipeline secured with the K9 System. The prerequisites for the developer are the same, being that he allowed either a non-official version or an impostor version from another repository to intrude into the lockfile and his local installation. In this setup, it may even be possible for the private registry to "pull through" impostor versions. In Figure 35, an excerpt from a lockfile is pictured, where an unofficial version that was not published to the private registry was still available from that private registry, due to the registry "pulling through" this version from the public registry. Other infiltration vectors still include the developer not being logged into the private registry on his machine, or the local client of the package manager getting confused between two of the same packages for one version being available on both private and public registry and choosing the malicious one[8].



```
"node_modules/master-trial-package": {
  "version": "0.1.3",
  "resolved": "https://master-registry-domain-365161439830.d.codeartifact.eu-west-1.amazonaws.com:443/npm/master-registry/master-trial-package/-/master-trial-package-0.1.3.tgz",
  "integrity": "sha512-J/pjtAwNYFAXrmzcWjsDF/juSMZp8Ui409FfKgnsJGFOzGQjtpitH0MQH2h/k9Da2Hjvo8ck5daPNi8Aun+gwg==",
  "hasInstallScript": true,
  "peerDependencies": {
    "aws-cdk-lib": "2.23.0",
    "constructs": "^10.0.0"
  }
},
```

Figure 35: A malicious package version ”pulled through” from the private registry in the lockfile

The first possibility, an accidental version upgrade to an illegitimate version, is attempted first. In this case, the version number in the package manifest was deliberately left as it was set when first installing the package with `npm i master-trial-package` (see Listing 10). Note the *caret* range (section 2.2.1) that was automatically set. This package manifest was then used to install all packages and generate a new lockfile locally, on the developers' machine. For this, the developer logged into the private registry before executing the `npm i` command.
After this, a new lockfile has been generated (compare Figure 35), which has “pulled through” the private registry the illegitimate version **0.1.3** of the package, due to the *caret* range specified in the manifest. This means, that the DCA has already succeeded on the developer's machine, and now has to be detected and prevented in the CI/CD environment.

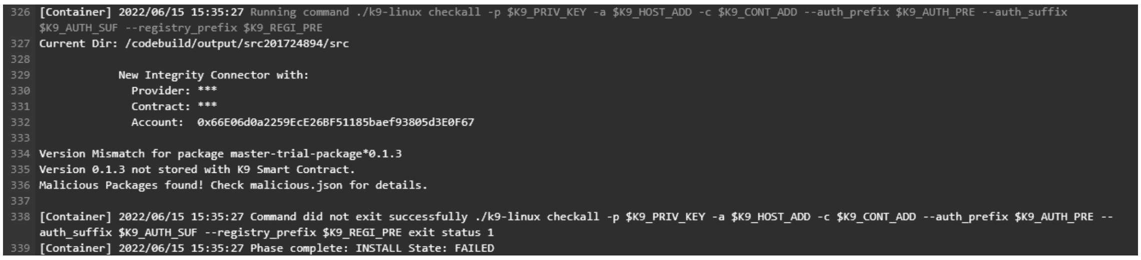
```
[...]
"dependencies": {
  "aws-cdk-lib": "2.23.0",
  "constructs": "^10.0.0",
  "master-trial-package": "^0.1.1",
  "source-map-support": "^0.5.21"
}
[...]
```

Listing 10: Package Version in Manifest of Application

This manifest and lockfile are then committed to the application repository, setting into motion the build process, which has been augmented with the K9 System. The K9 Agent gets invoked before installing the illegitimate, possibly malicious version of

[8] This behavior is not documented in any of npms documentation, but unless other, better, explanations are available, seems to have happened a number of times during the development and evaluation of the K9 System

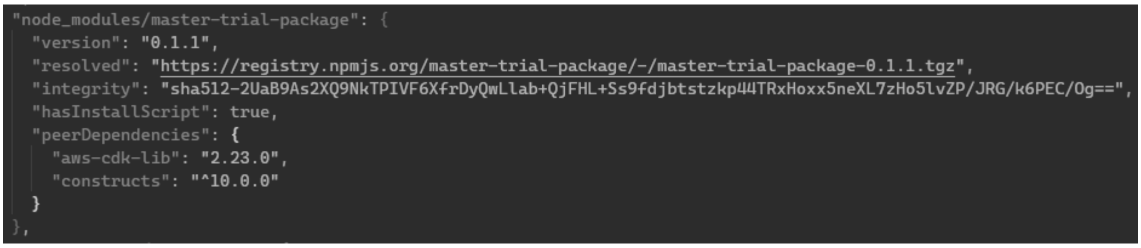
the package and compares the version specified in the lockfile (*0.1.3*) to the version information available on the smart contract (only *0.1.1*). The K9 Agent detects an illegitimate version, and stops the build with an error (see Figure 36).



```
[Container] 2022/06/15 15:35:27 Running command ./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --auth_prefix $K9_AUTH_PRE --auth_suffix
$K9_AUTH_SUF --registry_prefix $K9_REGI_PRE
Current Dir: /codebuild/output/src201724894/src

            New Integrity Connector with:
              Provider: ***
              Contract: ***
              Account:  0x66E06d0a2259EcE26BF51185baef93805d3E0F67

Version Mismatch for package master-trial-package*0.1.3
Version 0.1.3 not stored with K9 Smart Contract.
Malicious Packages found! Check malicious.json for details.

[Container] 2022/06/15 15:35:27 Command did not exit successfully ./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CONT_ADD --auth_prefix $K9_AUTH_PRE --
auth_suffix $K9_AUTH_SUF --registry_prefix $K9_REGI_PRE exit status 1
[Container] 2022/06/15 15:35:27 Phase complete: INSTALL State: FAILED
```

Figure 36: Malicious package version (non-official version) detected and build stopped

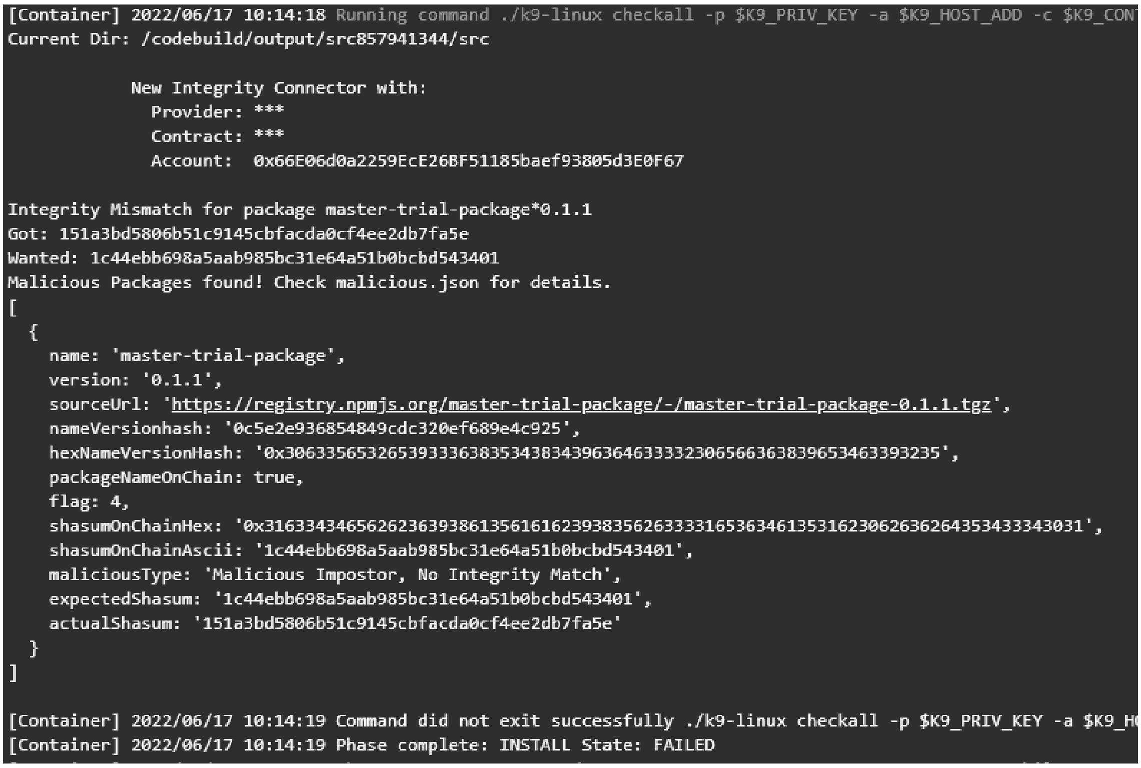
The second possibility, pulling an impostor version from a registry other than the private registry, is attempted after the version confusion. In this case, the version number in the package manifest was set to a fixed value (*0.1.1*). Other than that, a login into the private registry was not conducted. This package manifest was then used to install all packages and generate a new lockfile locally, on the developers' machine, using the `npm i` command.
After this, a new lockfile has been generated (compare Figure 37), which has specified the legitimate version **0.1.1** of the package, but from the public registry (*registry.npmjs.org*).



```
"node_modules/master-trial-package": {
  "version": "0.1.1",
  "resolved": "https://registry.npmjs.org/master-trial-package/-/master-trial-package-0.1.1.tgz",
  "integrity": "sha512-2UaB9As2XQ9NkTPIVF6XfrDyQwLlab+QjFHL+Ss9fdjbtstzkp44TRxHoxx5neXL7zHo5lvZP/JRG/k6PEC/Og==",
  "hasInstallScript": true,
  "peerDependencies": {
    "aws-cdk-lib": "2.23.0",
    "constructs": "^10.0.0"
  }
},
```

Figure 37: A malicious package version from the public registry in the lockfile

This manifest and lockfile are again committed to the application repository. The build process, augmented with the K9 System, starts again. The K9 Agent gets invoked before installing the version impostor of the package and compares the version specified in the lockfile (*0.1.1*) to the version information available on the smart contract (only *0.1.1*). The K9 Agent detects no illegitimate version, and thus requests the integrity information for the package `master-trial-package` at version `0.1.1` from the K9 Smart Contract. The K9 Agent then also requests integrity information from the source registry of the package specified in the lockfile. Once the information has been delivered from both sides, the K9 Agent compares the information from the K9 Smart Contract (nominal) to the information from the registry (actual), detects a mismatch, and stops the build with an error (see Figure 38).



```
[Container] 2022/06/17 10:14:18 Running command ./k9-linux checkall -p $K9_PRIV_KEY -a $K9_HOST_ADD -c $K9_CON
Current Dir: /codebuild/output/src857941344/src

            New Integrity Connector with:
              Provider: ***
              Contract: ***
              Account:  0x66E06d0a2259EcE26BF51185baef93805d3E0F67

Integrity Mismatch for package master-trial-package*0.1.1
Got: 151a3bd5806b51c9145cbfacda0cf4ee2db7fa5e
Wanted: 1c44ebb698a5aab985bc31e64a51b0bcbd543401
Malicious Packages found! Check malicious.json for details.
[
  {
    name: 'master-trial-package',
    version: '0.1.1',
    sourceUrl: 'https://registry.npmjs.org/master-trial-package/-/master-trial-package-0.1.1.tgz',
    nameVersionhash: '0c5e2e936854849cdc320ef689e4c925',
    hexNameVersionHash: '0x3063356532653933363835343834396364633332306566363839653463393235',
    packageNameOnChain: true,
    flag: 4,
    shasumOnChainHex: '0x31633434656262363938613561616239383562633331653634613531623062636264353433343031',
    shasumOnChainAscii: '1c44ebb698a5aab985bc31e64a51b0bcbd543401',
    maliciousType: 'Malicious Impostor, No Integrity Match',
    expectedShasum: '1c44ebb698a5aab985bc31e64a51b0bcbd543401',
    actualShasum: '151a3bd5806b51c9145cbfacda0cf4ee2db7fa5e'
  }
]

[Container] 2022/06/17 10:14:19 Command did not exit successfully ./k9-linux checkall -p $K9_PRIV_KEY -a $K9_H
[Container] 2022/06/17 10:14:19 Phase complete: INSTALL State: FAILED
```

Figure 38: Malicious package version (Impostor with mismatched integrity) detected and build stopped

In both these cases, the K9 Agent, together with the K9 Smart Contract, has detected a mismatch of the legitimate package version the developer intended to be used, and the illegitimate package versions, both at other version numbers and at the actual version number of the legitimate package, that were being injected into the process.

7 Conclusions

7.1 Summary

By analyzing the Dependency Confusion Attack (DCA), five characteristics have been defined that can be used to develop measures to defend against these attacks. The analysis consisted of a study of incident reports and an investigation into malicious packages actually used to execute the attack. These characteristics were then used, in conjunction with requirements of a technical and functional nature, to develop requirements for a system that can successfully prevent DCAs in a cloud-based Continuous Integration/Continuous Deployment (CI/CD) environment.

The system that was developed from the requirements consists of an agent, running locally within the CI/CD environment, and a smart contract, running on the Ethereum blockchain. The agent is used to store integrity information about private packages while these are being published. The same agent can then be used to retrieve this information when the package is consumed, making it possible to spot an impostor package. The integrity information is protected from manipulation by the strong manipulation resistance of the blockchain.

Testing within an environment this system would be deployed in showed promising results. While the control system exhibited a vulnerability to the attack, allowing for a malicious, unofficial version and an impostor of an existing version to penetrate the environment, the preventive system managed to identify an attack in both cases for the protected environment, stopping the build process and preventing any infected packages from reaching the CI/CD system. Additionally, performance, even though not a criterion for this proof of concept, was strong enough to not influence the build time of the application used to facilitate the Proof of Concept (PoC).

While the solution developed was successful in combatting the specified threat, using a public blockchain-based smart contract, while providing manipulation resistance, availability and reliable access, still poses the disadvantage of being reliant of cryptocurrency to publish and interact with it. With cryptocurrency being subject to extreme market swings, caution must be exercised.

Additionally, since file integrity information is used to validate packages against known good information, this system could be extended to be used with other package managers, as well as other software module sharing technologies, such as container repositories.

7.2 Outlook

Having provided a workable solution towards the problem of DCAs, this does not yet represent a finished product, nor one that would be deployable in this form in any sort of scalable context. In this section, several points of possible improvement or addition to the system will be presented, as well as a general outlook on preventing DCAs.

7.2.1 The K9 System: Deployment

The K9 System in its current form - Agent and Smart Contract - is relatively easy to deploy on a small scale, provided the K9 Agent can be distributed to the specific CI/CD environments. This problem has been solved in the development stage of the project by using a shared storage location, allowing the build system to download the binary (see subsection 6.2.3). In the future, this might be streamlined by integrating the K9 Agent into the build image provided to build environments, the practice of in-house build images being prevalent already. This would also allow the enforcement of using the K9 Agent to harden the build process (this concept schematically drawn up in Figure 41).

7.2.2 The K9 System: Widening the Scope

One important limitation of the K9 System in its current form is its specific compatibility with only node package manager (npm) as a package registry. The K9 System supports multiple types of npm compatible registries, by using a configuration file to set parameters it uses to collect meta information about packages, but is not yet compatible with, for example, pip/PyPi or maven packages. This feature would be an attractive addition to the tool set of the K9 System, enabling it to work with a wider number of package management systems.

Additionally, the feature set provided with the K9 System is not limited in its application to package management systems. Integrity management is also a feature usable in the distribution of, for example, software updates to customers and end users, or the distribution of containerized applications. With some modification, the K9 System could be used in these scenarios as well, providing an easy way for the recipient of the container or software update to validate its legitimacy.

7.2.3 The K9 System: Monitoring and Integration

One of the shortcomings of the K9 Agent when deployed in its current form on a wider scale, is the lack of monitorability. If multiple Agents are deployed in different CI/CD environments, staying on top of their status requires looking into the build logs of each. Additionally, there is currently no way to monitor the status of the K9 Smart Contract, be that the currently deployed version, nor what data is stored with the contract.

This lack of monitorability could be solved by the addition of a third component, the K9 Dashboard. This Dashboard might include backend services connecting to the K9 Agents that are currently deployed, and show their status and possible findings, as well as allow access to the lists of external packages consumed by each project. The backend might serve its data via an API, which would allow, on the one hand, a User Interface to be built, which would show this information in an easy-to-use way. This UI could be used to integrate the K9 System into the DevSecOps process, providing a way to monitor usage of internal packages in CI/CD environments and overviews over versions saved on the K9 Smart Contract. A concept of this is shown in Figure 46 and Figure 47 in the appendix. The API might also allow the integration of the K9 System into other Security information and event management (SIEM) solutions, providing a way to extract security metrics and trigger alarms in existing systems.

7.2.4 The K9 System: Integration with Open Source Projects

As explained in section 2.1.2, the Open Source Community does also show problems concerning their authors' credibility and possible injections of malicious versions through their release cycle. The decentralized nature of these projects gives enough leeway to malicious actors to regularly infiltrate these projects and wreak havoc. This decentralized nature, however, also makes these projects ideal candidates for the implementation of a slightly modified version of the K9 System, which would allow Open Source projects to track their releases on the blockchain, and make it easier for users to check the validity of Open Source software they are using.

7.2.5 Academic interest in DCAs

Over the course of the research leading to the development of the K9 System, a large part of the time spent researching was dedicated to analyzing actual attacks. This was due to a marked lack of published research in terms of quantitative analysis of the attacks themselves. This indicates that there is a distinct need for this kind of research, which could be used to facilitate easier and faster development of systems and solutions like the K9 System. The attacks, according to the few sources that are actively monitoring them (most prominently, security solutions providers such as Snyk[1] and Sonatype[2]) do not decrease in number or severity, meaning that actual research into the matter does still hold value.

[1] `https://snyk.io/`

[2] `https://www.sonatype.com/`

A Architecture Decision Records

Note: Please note that in the following Architecture Decision Record (ADR)s[1] the collective “We” is used instead of “I” or passive forms. This should not imply multiple authors, but is meant to evoke the “customers” of the system, similar to user stories in Agile software development.

A.1 Use of Blockchain with Smart Contract as Persistent Data Store

Issue: We want a tool to store software (package) meta information, specifically information about file integrity identified by a name and a version number.

- We want this data store to be as tamper resistant as possible
- We require a high degree of availability of the data store
- We require the data store to provide easy to interface with read and write capabilities
- We want the data store to be hard to write to unauthorized
- We want the data store to be easy to retrieve information from
- We want the data store to be as independent of our infrastructure as possible

Decision: Decided for a smart contract on the Ethereum Blockchain.

[1] The ADR template used is an abridged version of the ADR template by Jeff Tyree and Art Akermann: `https://github.com/joelparkerhenderson/architecture-decision-record/blob/main/templates/decision-record-template-by-jeff-tyree-and-art-akerman/index.md`

Details

Assumptions: We want to be able to trace integrity of software packages without trusting the delivery mechanisms. Distributing and consuming software packages without knowledge about if or how their contents change between being authored and installed in the target environment opens consumers up to dangers from malicious actors and discourages sharing of reusable software components.

Constraints:

- If we choose a tool that needs a large supporting infrastructure, this infrastructure brings maintenance requirements and possibly other security liabilities
- If we choose a tool that requires placing trust in singular links of the "Chain of Trust", this brings liabilities if the link is compromised
- If we choose a tool that cannot be opened to a broader audience, we limit possible distribution of trusted software to this audience

Positions:

- We considered using a local database. This introduces maintenance of a database and removes the possibility of opening up the system to an audience outside, since third parties would have to place trust in a single authority. Additionally, availability varies heavily.
- We considered using a self-managed distributed databases, which would solve availability issues. Since controlling interest would be held by a single party, trust from third parties would still be an issue.
- We considered using a distributed ledger. This would include deploying out own, which would imply heavy development load and possible maintenance implications in the future, but would ease interfacing with third parties and solve the reliability problem. On the other hand, deploying our own ledger can be considered unrealistic due to needing to get third parties onboarded to build a net of trust.
- We considered using an already existing distributed ledger in the form of public blockchains. This included a shortlist of popular and established blockchain technologies such as Bitcoin, Ethereum and Tezos. The requirement for ease of read and write capabilities limited the list further to technologies that offered smart contract capabilities, e.g. possibilities to deploy out own business logic to interface with the blockchain.

Argument: Since the blockchain technology space is highly volatile and fraught with fraudulent activity [67], a technology with a good track record regarding security and legal issues was required. This led to choosing Ethereum, which is an established technology in the space and offers easy development for smart contracts with the Solidity language. It should be noted, that while fraudulent activity and controversy has been perpetrated by use of technology related to the Ethereum technology stack (For example the large amount of variously controversial activity around NFTs between around 2020 and 2022 [68]), none of it was related to the base technology, making the use of the Ethereum platform a reliable choice.
Additionally, the large number of distributed nodes of Ethereum increases reliability[2], and distributes control over the network over a large group of participants.

A.2 Use of a CLI-based Binary as the Security System

Issue: We want a tool to use to publish to and retrieve from the smart contract package version integrity information to ensure the integrity of software packages we publish and consume through an internal package registry.

- We want to integrate the system into Linux-based CI/CD environments, both existing and deployed in future
- We want to be able to distribute the system easily to our CI/CD environments
- We want to easily use the tool with the existing features of the CI/CD environments

Decision: Decided for a CLI-based binary program distributed via shared storage facilities.

[2] 5973 Nodes as of August 2022, see `https://ethernodes.org/`

Details

Assumptions: We want to be able to trace integrity of software packages we install in our CI/CD environments without trusting the delivery mechanisms. Consuming software packages without knowledge about if or how their contents change between being authored and installed presents dangers to our CI/CD environments and the applications they produce, since both may be compromised by malicious software packages being distributed under the guise of benign shared components.

Constraints:

- If we choose a tool that needs to be installed from an untrusted source, this presents the same dangers we seek to avoid
- The tool we choose has external dependencies that need to be installed before using it, this again presents the dangers we seek to avoid
- If we choose a tool that cannot be used from a bash script or the Linux command line, integration is made more difficult than changing or adding some build instructions, which will limit real-world use of the tool

Positions:

- We considered publishing a npm package that is globally installed in the build environment. While this would make integration into the CI/CD environments easy, it would present the danger of the delivery mechanisms being compromised, which is what was sought to be avoided in the first place.
- We considered using a compiled binary of the tool. One option would be to place this binary in a storage location accessible to the CI/CD environments (treated as a directory) with read access from the environment, and copying it at runtime of the build step. This would ease distribution, and satisfy the requirement of being usable with onboard tools of the build step, given that the binary is compiled for the target system.
- We considered using the above-mentioned binary, and distributing it with the Docker image used as the build environment. This would make mandating the distribution of the tool into every build environment possible, which can be accomplished by compliance tools of the cloud provider. On the other hand,

it would mandate building a new build environment image every time a new version of the tool is released.

Argument: The decision taken was to use a compiled binary distributed via shared storage in the form of an AWS S3 Bucket. This includes the caveat of changing this approach once the initial development and evaluation stage has been completed. With the numerous changes and releases of new versions through the development stage, a new build of a build environment image would slow development time too much to evaluate the whole system in a timely manner. Thus, until this stage is completed, a binary compiled for the target environment, distributed via shared, read-only, storage was chosen. The npm package approach was not chosen for the reason detailed above.

Implications: Once the development and evaluation stage has been completed, the distribution of the tool via pre-built build environment images is favored for being mandateable by a governing authority on the cloud platform.

B Diagrams

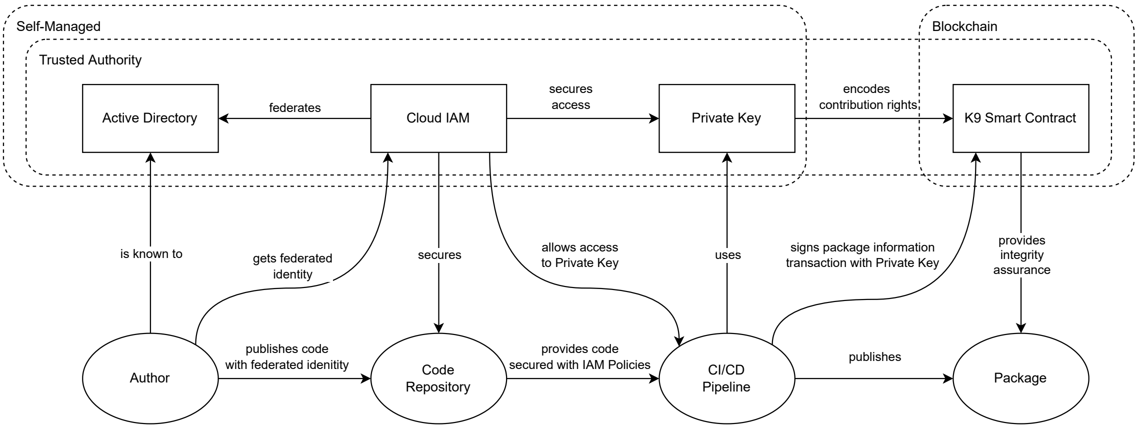


Figure 39: The explicit "Chain of Trust" for the publishing of a package, as planned for the K9 System

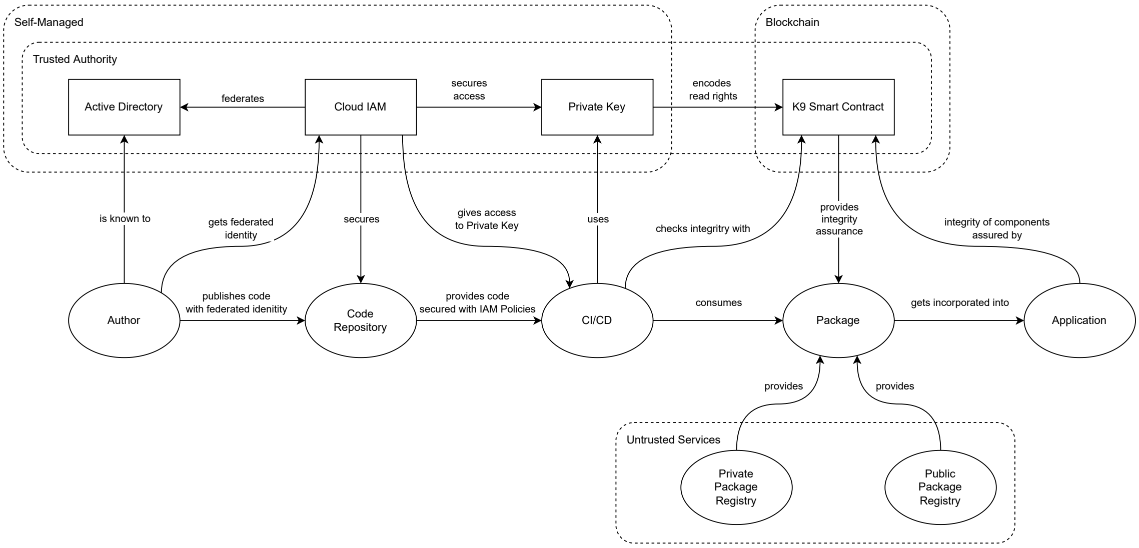


Figure 40: The absolute "Chain of Trust" for the consumption of a package, as planned for the K9 System

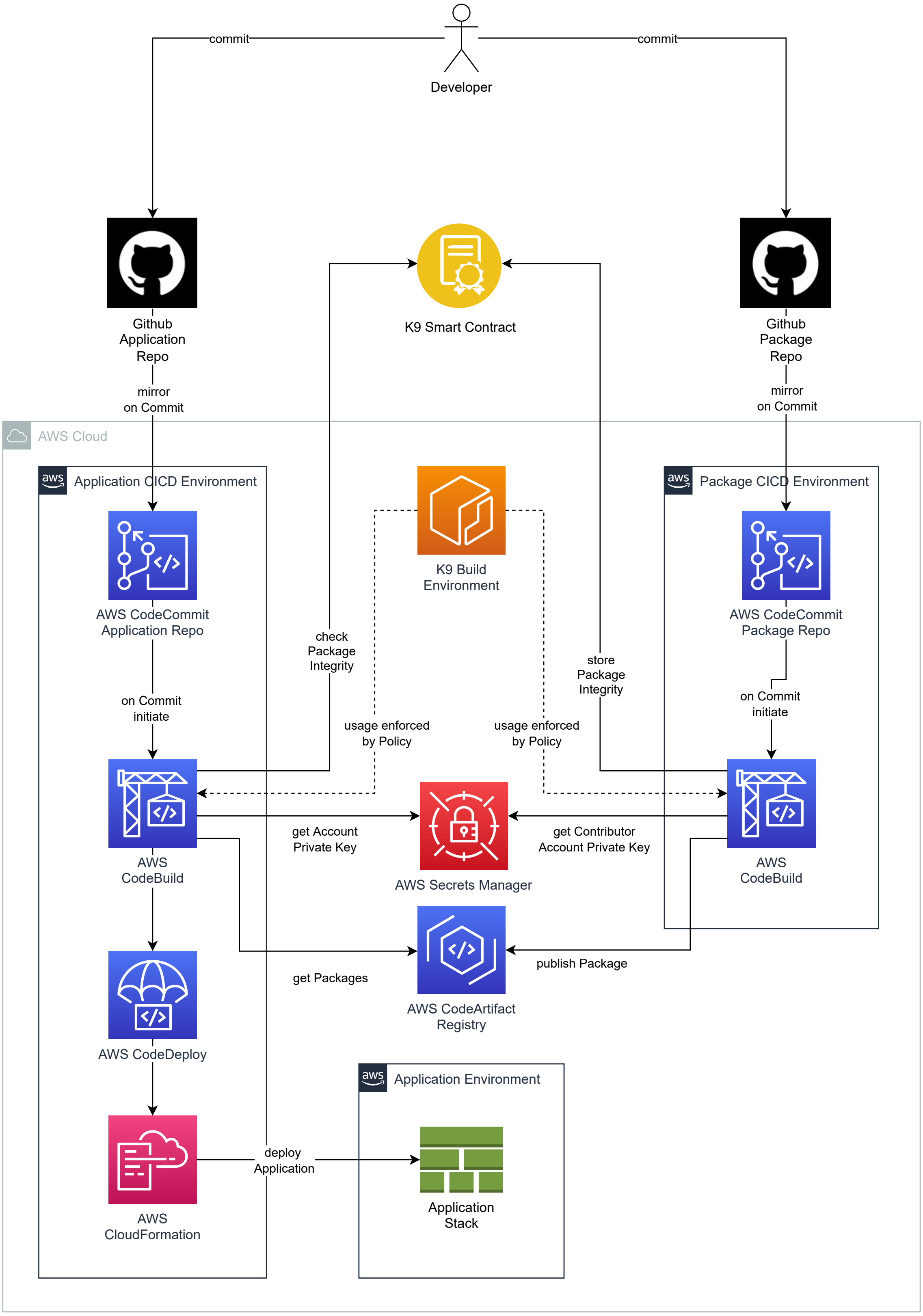


Figure 41: K9 System CI/CD architecture (Maturity)

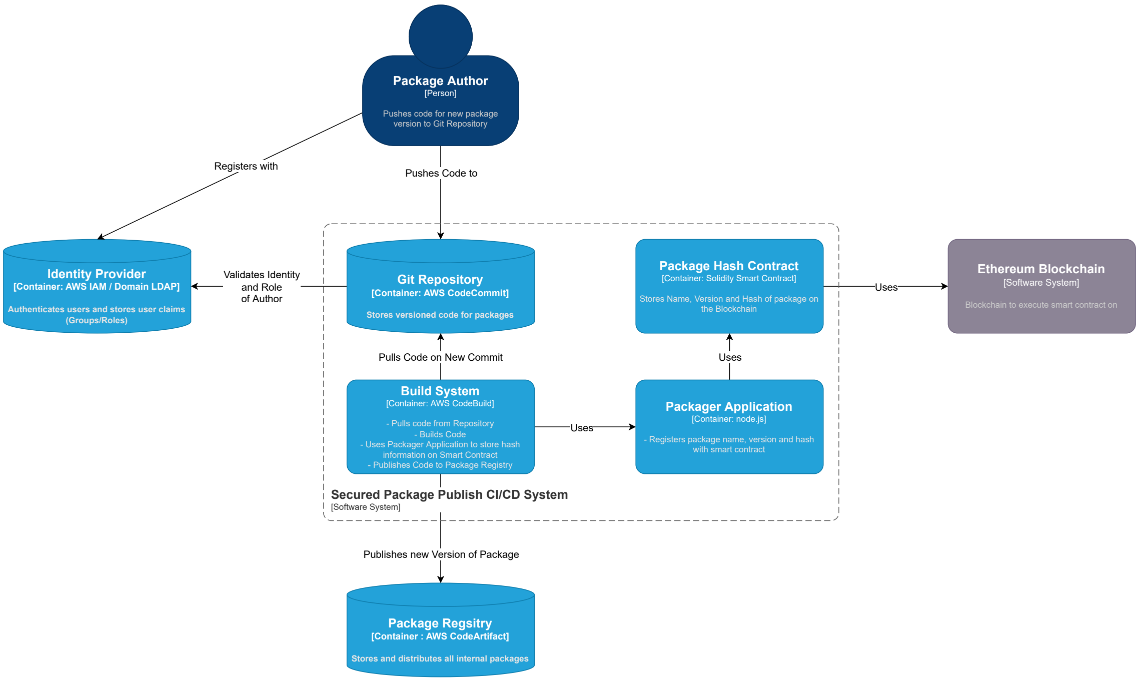

Package Author
[Person]
Pushes code for new package version to Git Repository
Registers with
Pushes Code to
Identity Provider
[Container: AWS IAM / Domain LDAP]
Authenticates users and stores user claims (Groups/Roles)
Validates Identity and Role of Author
Git Repository
[Container: AWS CodeCommit]
Stores versioned code for packages
Package Hash Contract
[Container: Solidity Smart Contract]
Stores Name, Version and Hash of package on the Blockchain
Uses
Ethereum Blockchain
[Software System]
Blockchain to execute smart contract on
Pulls Code on New Commit
Uses
Build System
[Container: AWS CodeBuild]
- Pulls code from Repository
- Builds Code
- Uses Packager Application to store hash information on Smart Contract
- Publishes Code to Package Registry
Uses
Packager Application
[Container: node.js]
- Registers package name, version and hash with smart contract
Secured Package Publish CI/CD System
[Software System]
Publishes new Version of Package
Package Regsitry
[Container : AWS CodeArtifact]
Stores and distributes all internal packages

Figure 42: C4 Container Diagram: Package publishing system

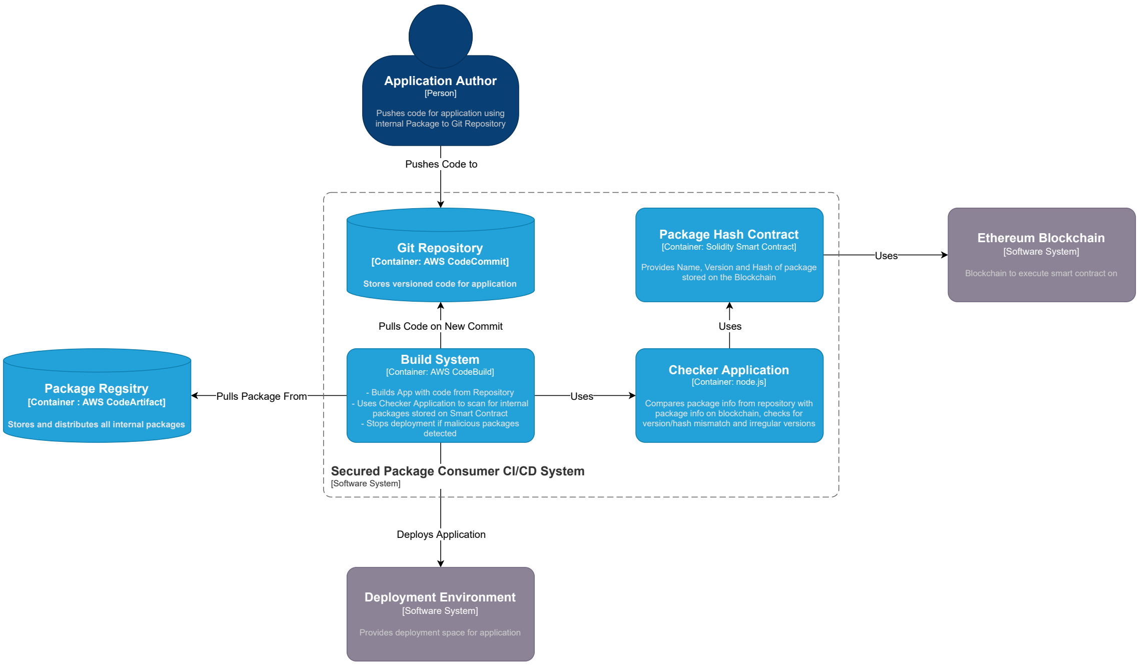

Application Author
[Person]
Pushes code for application using internal Package to Git Repository
Pushes Code to
Git Repository
[Container: AWS CodeCommit]
Stores versioned code for application
Package Hash Contract
[Container: Solidity Smart Contract]
Provides Name, Version and Hash of package stored on the Blockchain
Uses
Ethereum Blockchain
[Software System]
Blockchain to execute smart contract on
Pulls Code on New Commit
Uses
Package Regsitry
[Container : AWS CodeArtifact]
Stores and distributes all internal packages
Pulls Package From
Build System
[Container: AWS CodeBuild]
- Builds App with code from Repository
- Uses Checker Application to scan for internal packages stored on Smart Contract
- Stops deployment if malicious packages detected
Uses
Checker Application
[Container: node.js]
Compares package info from repository with package info on blockchain, checks for version/hash mismatch and irregular versions
Secured Package Consumer CI/CD System
[Software System]
Deploys Application
Deployment Environment
[Software System]
Provides deployment space for application

Figure 43: C4 Container Diagram: consuming a package in building an application

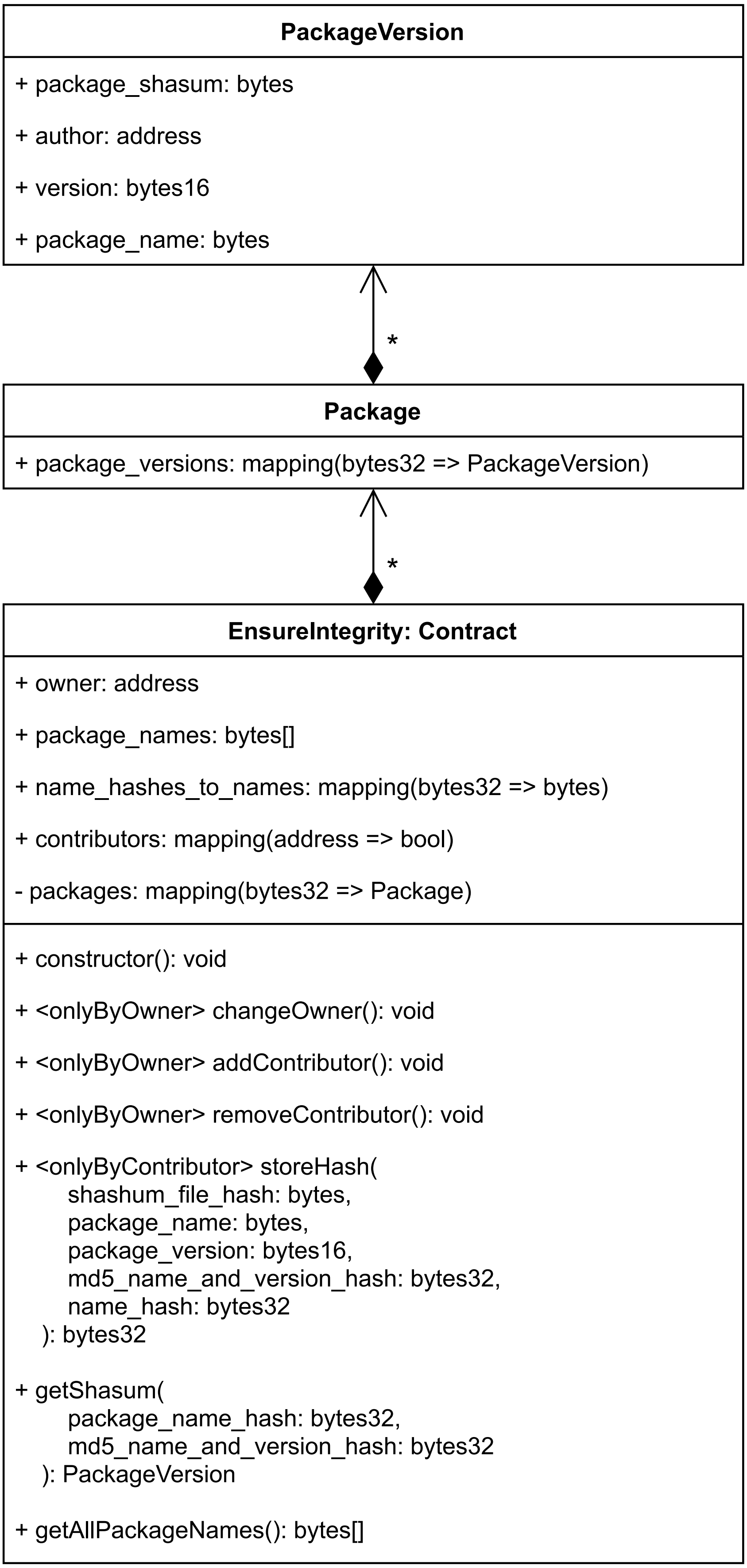

PackageVersion
+ package_shasum: bytes
+ author: address
+ version: bytes16
+ package_name: bytes
*
Package
+ package_versions: mapping(bytes32 => PackageVersion)
*
EnsureIntegrity: Contract
+ owner: address
+ package_names: bytes[]
+ name_hashes_to_names: mapping(bytes32 => bytes)
+ contributors: mapping(address => bool)
- packages: mapping(bytes32 => Package)
+ constructor(): void
+ <onlyByOwner> changeOwner(): void
+ <onlyByOwner> addContributor(): void
+ <onlyByOwner> removeContributor(): void
+ <onlyByContributor> storeHash(
shashum_file_hash: bytes,
package_name: bytes,
package_version: bytes16,
md5_name_and_version_hash: bytes32,
name_hash: bytes32
): bytes32
+ getShasum(
package_name_hash: bytes32,
md5_name_and_version_hash: bytes32
): PackageVersion
+ getAllPackageNames(): bytes[]

Figure 44: Classes: K9 Smart Contract

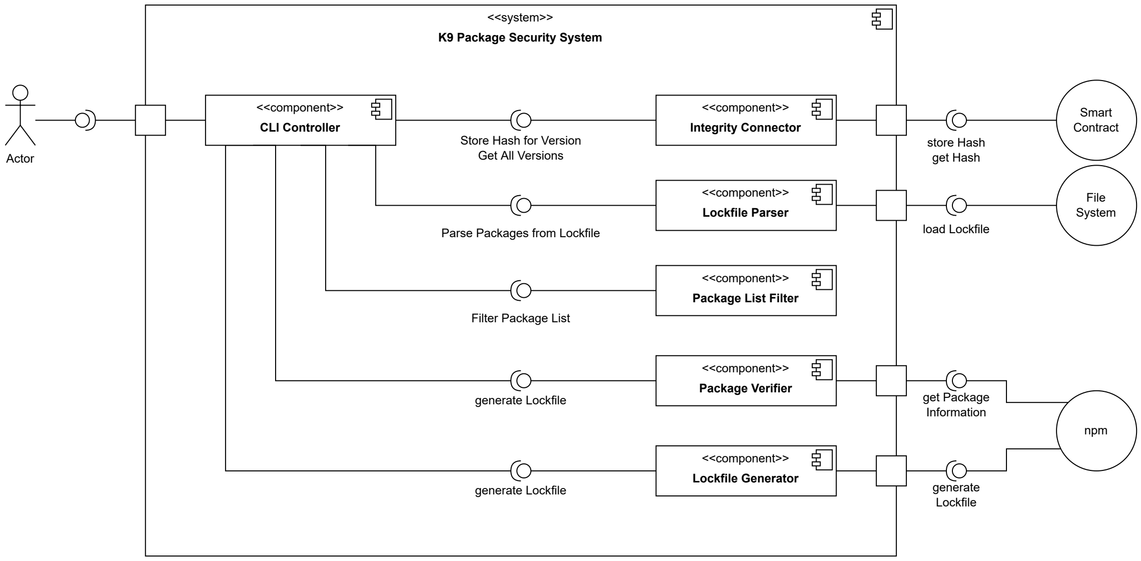
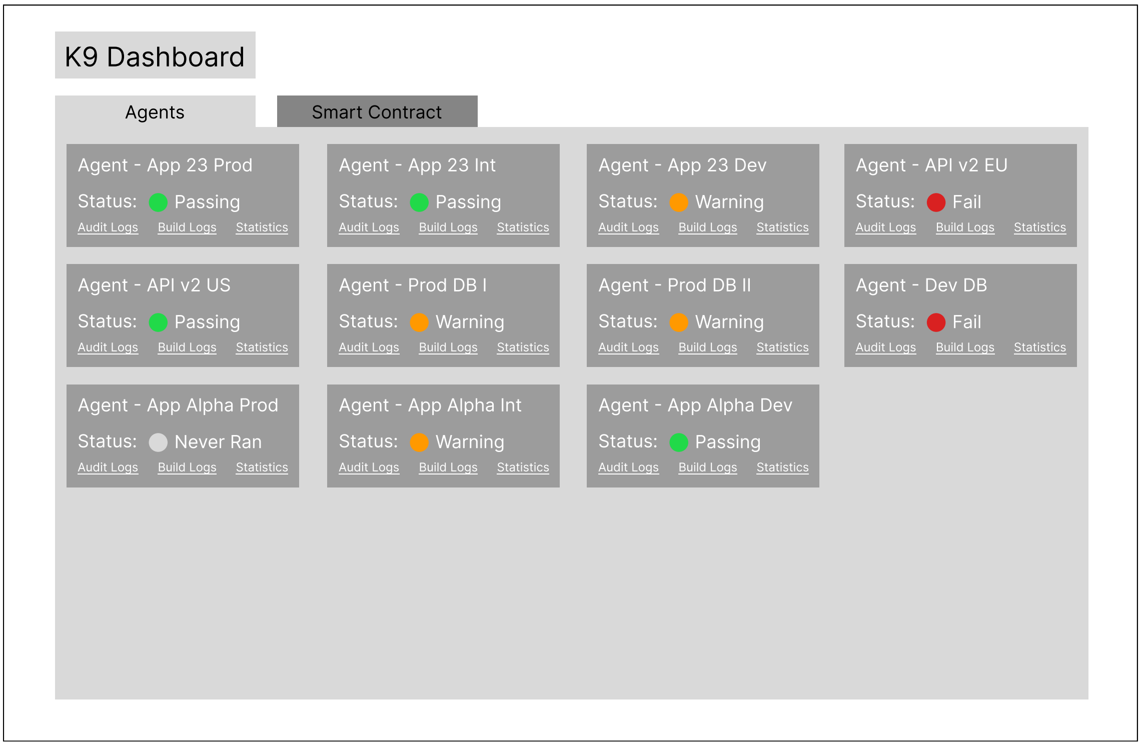


Figure 45: C4 Components Diagram: K9 Agent



Agent - App 23 Prod
Status: ● Passing
Audit Logs Build Logs Statistics

Agent - App 23 Int
Status: ● Passing
Audit Logs Build Logs Statistics

Agent - App 23 Dev
Status: ● Warning
Audit Logs Build Logs Statistics

Agent - API v2 EU
Status: ● Fail
Audit Logs Build Logs Statistics

Agent - API v2 US
Status: ● Passing
Audit Logs Build Logs Statistics

Agent - Prod DB I
Status: ● Warning
Audit Logs Build Logs Statistics

Agent - Prod DB II
Status: ● Warning
Audit Logs Build Logs Statistics

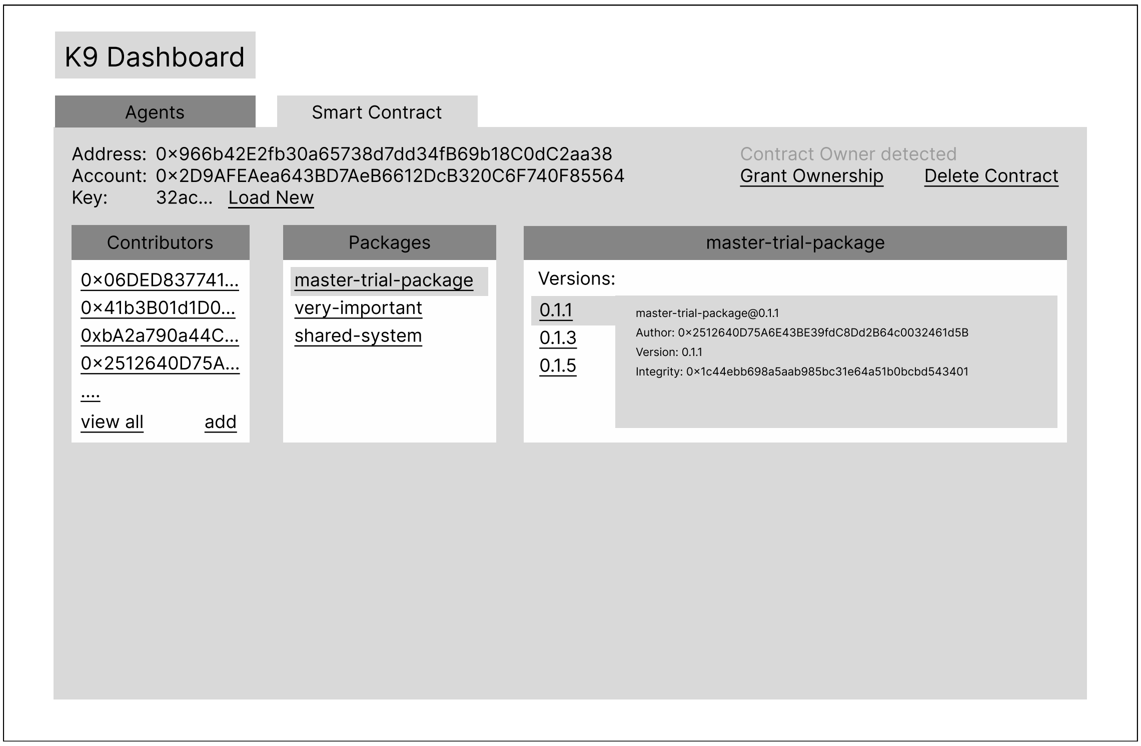
Agent - Dev DB
Status: ● Fail
Audit Logs Build Logs Statistics

Agent - App Alpha Prod
Status: ● Never Ran
Audit Logs Build Logs Statistics

Agent - App Alpha Int
Status: ● Warning
Audit Logs Build Logs Statistics

Agent - App Alpha Dev
Status: ● Passing
Audit Logs Build Logs Statistics

Figure 46: K9 System dashboard concept



Address: 0×966b42E2fb30a65738d7dd34fB69b18C0dC2aa38
Account: 0×2D9AFEAea643BD7AeB6612DcB320C6F740F85564
Key: 32ac...

0×06DED837741...
0×41b3B01d1D0...
0xbA2a790a44C...
0×2512640D75A...
....

master-trial-package
very-important
shared-system

Versions:
0.1.1
0.1.3
0.1.5

master-trial-package@0.1.1
Author: 0×2512640D75A6E43BE39fdC8Dd2B64c0032461d5B
Version: 0.1.1
Integrity: 0×1c44ebb698a5aab985bc31e64a51b0bcbd543401

Figure 47: K9 System dashboard concept

Bibliography

[1] Martin Christopher. *Logistics and supply chain management: creating value-added networks.* 3rd ed. Harlow, England ; New York: FT Prentice Hall, 2005. ISBN: 978-0-273-68176-2.

[2] Christoph Kilger, Herbert Meyr, and Hartmut Stadtler, eds. *Supply Chain Management and Advanced Planning: Concepts, Models, Software, and Case Studies.* 5th ed. 2015. Springer Texts in Business and Economics. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer, 2015. ISBN: 978-3-642-55309-7. DOI: `10.1007/978-3-642-55309-7`.

[3] Daniel Rodríguez et al. "Empirical findings on team size and productivity in software development". In: *Journal of Systems and Software* 85.3 (2012), pp. 562–570.

[4] Steven Raemaekers, Arie van Deursen, and Joost Visser. "An analysis of dependence on third-party libraries in open source and proprietary systems". In: *Sixth international workshop on software quality and maintainability, SQM.* Vol. 12. 2012, pp. 64–67.

[5] sonatype. *2021 State of the Software Supply Chain.* 2022. URL: `https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021`.

[6] Cybersecurity and Infrastructure Security Agency. "Defending Against Software Supply Chain Attacks". In: (2021). Publisher: NIST.

[7] C J Alberts et al. "A Systemic Approach for Assessing Software Supply-Chain Risk". In: *2011 44th Hawaii International Conference on System Sciences.* Kauai, HI: IEEE, Jan. 2011, pp. 1–8. ISBN: 978-1-4244-9618-1. DOI: `10.1109/HICSS.2011.36`. URL: `http://ieeexplore.ieee.org/document/5718996/` (visited on 03/18/2022).

[8] Marc Ohm et al. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment.* Ed. by Clémentine Maurice et al. Vol. 12223. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 23–43. ISBN: 978-3-030-52682-5. DOI: `10.1007/978-3-030-52683-2_2`.

[9] David Gilbertson. *I'm harvesting credit card numbers and passwords from your site. Here's how.* (Accessed on 04/02/2022). Jan. 2018. URL: `https://medium.com/hackernoon/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5`.

[10] Thomas Hunter II. *Compromised npm Package: event-stream.* (Accessed on 04/02/2022). Nov. 2018. URL: `https://medium.com/intrinsic-blog/compromised-npm-package-event-stream-d47d08605502`.

[11] Jeremy Katz. *A brief history of package management.* (Accessed on 03/26/2022). Dec. 2017. URL: https://blog.tidelift.com/a-brief-history-of-package-management.

[12] Steve Ovens. *The evolution of package managers.* July 2018. URL: https://opensource.com/article/18/7/evolution-package-managers.

[13] The Debian Community. *What is a package manager?* URL: https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html.

[14] Diomidis Spinellis. "Package management systems". In: *IEEE software* 29.2 (2012), pp. 84–86.

[15] GitHub. *The State of the Octoverse.* (Accessed on 04/01/2022). Dec. 2019. URL: https://octoverse.github.com/2019/.

[16] Tom Preston Warner et. al. *Semantic Version Specification 2.0.0.* (Accessed on 04/01/2022). URL: https://semver.org/.

[17] Kris Kowal. "CommonJS effort sets JavaScript on path for world domination". In: (Dec. 2009). URL: https://arstechnica.com/information-technology/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination/.

[18] Baruch Sadogursky. "Going Beyond Exclude Patterns: Safe Repositories With Priority Resolution". In: (June 2021). URL: https://jfrog.com/blog/going-beyond-exclude-patterns-safe-repositories-with-priority-resolution/.

[19] Alex Birsan. *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies.* Feb. 2021. URL: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.

[20] Xinyuan Wang. "Small World with High Risks: A Study of Security Threats in the npm Ecosystem". In: *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 995–1010. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.

[21] Brian Pfretzschner and Lotfi ben Othmane. "Identification of dependency-based attacks on node. js". In: *Proceedings of the 12th International Conference on Availability, Reliability and Security.* 2017, pp. 1–6.

[22] German Federal Financial Supervisory Authority. "Blockchain technology". In: *BaFin* (June 2017). URL: https://www.bafin.de/EN/Aufsicht/FinTech/Blockchain/blockchain_node_en.html.

[23] Satoshi Nakamoto. "Bitcoin whitepaper". In: *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)* (2008).

[24] Vitalik Buterin et al. "A next-generation smart contract and decentralized application platform". In: *white paper* 3.37 (2014), pp. 2–1.

[25] Daniel Drescher. *Blockchain basics: a non-technical introduction in 25 steps.* 2017.

[26] Du Mingxiao et al. "A review on consensus algorithm of blockchain". In: *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE. 2017, pp. 2567–2572.

[27] Ujan Mukhopadhyay et al. "A brief survey of cryptocurrency systems". In: *2016 14th annual conference on privacy, security and trust (PST)*. IEEE. 2016, pp. 745–752.

[28] Jingming Li et al. "Energy consumption of cryptocurrency mining: A study of electricity consumption in mining cryptocurrencies". In: *Energy* 168 (2019), pp. 160–168.

[29] Gideon Greenspan. "Ending the bitcoin vs blockchain debate". In: (2015). URL: `http://www.multichain.com/blog/2015/07/bitcoin-vsblockchain-%20debate/`.

[30] Nick Szabo et al. *Smart contracts*. 1994. URL: `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`.

[31] Rosco Kalis and Adam Belloum. "Validating data integrity with blockchain". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 272–277.

[32] Konstantinos Christidis and Michael Devetsikiotis. "Blockchains and smart contracts for the internet of things". In: *Ieee Access* 4 (2016), pp. 2292–2303.

[33] Vitalik Buterin. *Thoughts on UTXOs by Vitalik Buterin, Co-Founder of Ethereum*. 2016. URL: `https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53`.

[34] Loi Luu et al. "Making smart contracts smarter". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 254–269.

[35] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices". In: *IEEE Access* 5 (2017), pp. 3909–3943.

[36] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[37] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[38] *Stack Overflow Developer Survey 2021*. Dec. 2021. URL: `https://insights.stackoverflow.com/survey/2021`.

[39] Brian Fitzgerald and Klaas-Jan Stol. "Continuous software engineering: A roadmap and agenda". In: *Journal of Systems and Software* 123 (2017), pp. 176–189.

[40] Asaf Nadler, Avi Aminov, and Asaf Shabtai. "Detection of malicious and low throughput data exfiltration over the DNS protocol". In: *Computers & Security* 80 (2019), pp. 36–53.

[41] Duc-Ly Vu et al. “Typosquatting and combosquatting attacks on the python ecosystem”. In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2020, pp. 509–514.

[42] Matthew Taylor et al. “Spellbound: Defending against package typosquatting”. In: *arXiv preprint arXiv:2003.03471* (2020).

[43] Shachar Menashe Andrey Polkovnychenko. “Large-scale npm attack targets Azure developers with malicious packages”. In: (Mar. 2022). URL: `https://jfrog.com/blog/large-scale-npm-attack-targets-azure-developers-with-malicious-packages/`.

[44] Ax Sharma. “Sonatype Spots 275+ Malicious npm Packages Copying Recent Software Supply Chain Attacks that Hit 35 Organizations”. In: (Feb. 2021). URL: `https://blog.sonatype.com/sonatype-spots-150-malicious-npm-packages-copying-recent-software-supply-chain-attacks`.

[45] Ax Sharma. “Newly Identified Dependency Confusion Packages Target Amazon, Zillow, and Slack; Go Beyond Just Bug Bounties”. In: (Mar. 2021). URL: `https://blog.sonatype.com/malicious-dependency-confusion-copycats-exfiltrate-bash-history-and-etc-shadow-files`.

[46] Ax Sharma. “PyPI and npm Flooded with over 5,000 Dependency Confusion Copycats”. In: (Mar. 2021). URL: `https://blog.sonatype.com/pypi-and-npm-flooded-with-over-5000-dependency-confusion-copycats`.

[47] Shachar Polkovnychenko Andrey; Menashe. *Npm Supply Chain Attack Targets Germany-based Companies with Dangerous Backdoor Malware*. May 2022. URL: `https://jfrog.com/blog/npm-supply-chain-attack-targets-german-based-companies/`.

[48] Maricela-Georgiana Avram. “Advantages and challenges of adopting cloud computing from an enterprise perspective”. In: *Procedia Technology* 12 (2014), pp. 529–534.

[49] Anca Apostu et al. “Study on advantages and disadvantages of Cloud Computing–the advantages of Telemetry Applications in the Cloud”. In: *Recent advances in applied computer science and digital services* 2103 (2013).

[50] Mazhar Ali, Samee U Khan, and Athanasios V Vasilakos. “Security in cloud computing: Opportunities and challenges”. In: *Information sciences* 305 (2015), pp. 357–383.

[51] Marc Ohm, Arnold Sykosch, and Michael Meier. “Towards detection of software supply chain attacks by forensic artifacts”. In: Proceedings of the 15th International Conference on Availability, Reliability and Security. Ed. by Melanie Volkamer and Christian Wressnegger. Virtual Event Ireland: ACM, 2020, pp. 1–6. ISBN: 978-1-4503-8833-7. DOI: `10.1145/3407023.3409183`.

[52] *Cyber Observable Core Concepts*. Specification. OASIS Open, July 2017. URL: `https://docs.oasis-open.org/cti/stix/v2.0/cs01/part3-cyber-observable-core/stix-v2.0-cs01-part3-cyber-observable-core.html`.

[53] Marc Ohm et al. "Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation". In: *arXiv e-prints* (2020), arXiv:2011.02235. (Visited on 11/04/2020).

[54] *Domain names - implementation and specification.* RFC 1035. Nov. 1987. DOI: `10.17487/RFC1035`. URL: `https://www.rfc-editor.org/info/rfc1035`.

[55] Anirban Das et al. "Detection of Exfiltration and Tunneling over DNS". In: *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA).* 2017, pp. 737–742. DOI: `10.1109/ICMLA.2017.00-71`.

[56] Barry Hensley. "Identity is the new perimeter in the fight against supply chain attacks". In: *Network Security* 2021.7 (2021), pp. 7–9. ISSN: 13534858. DOI: `10.1016/S1353-4858(21)00074-X`.

[57] Center for Internet Security (CIS). "The SolarWinds Cyber-Attack: What You Need to Know". In: (Mar. 2021). URL: `https://www.cisecurity.org/solarwinds`.

[58] Mark Altmann. *How enterprises benefit from scoped npm registries.* 2022. URL: `https://satellytes.com/blog/post/enterprises-benefit-from-scoped-npm-registries/`.

[59] Ax Sharma. *Why are Dependency Confusion Attacks Not Going Away?* 2022. URL: `https://blog.sonatype.com/why-are-dependency-confusion-attacks-not-going-away`.

[60] Ax Sharma. *Newly Identified Dependency Confusion Packages Target Amazon, Zillow, and Slack; Go Beyond Just Bug Bounties.* Mar. 2021. URL: `https://blog.sonatype.com/malicious-dependency-confusion-copycats-exfiltrate-bash-history-and-etc-shadow-files`.

[61] Ax Sharma. *PyPI Flooded with 1,275 Dependency Confusion Packages.* 2022. URL: `https://blog.sonatype.com/pypi-flooded-with-over-1200-dependency-confusion-packages`.

[62] Ax Sharma. *VMware VSphere dependency confusion attempt caught by Sonatype.* Apr. 2022. URL: `https://blog.sonatype.com/vmware-vsphere-dependency-confusion-attempt-caught-by-sonatype`.

[63] Rainald Menge-Sonnentag. *Verwirrung um vermeintlichen Dependency-Confusion-Angriff auf deutsche Firmen.* 2022. URL: `https://www.heise.de/news/Verwirrung-um-vermeintlichen-Dependency-Confusion-Angriff-auf-deutsche-Firmen-7089135.html`.

[64] Paul Roberts. *Update: NPM dependency confusion hacks target German firms.* 2022. URL: `https://blog.reversinglabs.com/blog/npm-dependency-confusion-hacks-target-german-firms`.

[65] Snyk Security Research Team. *Targeted npm dependency confusion attack caught red-handed.* 2022. URL: `https://snyk.io/blog/npm-dependency-confusion-attack-gxm-reference/`.

[66] Klaus Pohl. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam-foundation level-IREB compliant*. Rocky Nook, Inc., 2016.

[67] Weili Chen et al. “Phishing Scam Detection on Ethereum: Towards Financial Security for Blockchain Ecosystem.” In: *IJCAI*. 2020, pp. 4506–4512.

[68] Nir Kshetri. “Scams, Frauds, and Crimes in the Nonfungible Token Market”. In: *Computer* 55.4 (2022), pp. 60–64.

List of Figures

List of Tables

Listings

Acronyms

ADR	Architecture Decision Record. 63, 72, 102
BOM	Bill of Materials. 69, 74, 92
CDK	Cloud Development Kit. 55
CI/CD	Continuous Integration/Continuous Deployment. I, 1, 2, 9, 20–24, 30, 32, 36–38, 48, 52–57, 60, 61, 63–65, 68, 71, 74–76, 80, 83–86, 89, 90, 92–95, 98–100, 104, 105, 108, 119, 120, 125
CISA	Cybersecurity and Infrastructure Security Agency. 5
COT	Chain of Trust. 59–61, 63, 65
DCA	Dependency Confusion Attack. III, IV, 2, 6, 25, 27–31, 34, 39–41, 43–50, 52, 59–61, 65, 68, 79, 82, 86, 87, 95, 98, 99, 101, 119, 121
DNS	Domain Name System. 34–37, 41–43, 45, 48
IaC	Infrastructure as Code. 9, 53, 55
IAM	Identity and Access Management. 56, 64, 65, 85, 86, 89, 90, 92
ICT	Information and Communications Technology. 5, 21, 121
IDP	Identity Provider. 66
npm	node package manager. II, 2, 8–17, 24–27, 29, 31, 33, 38, 40, 41, 48, 61, 72, 80, 83, 86, 88, 95, 99, 105, 106, 119, 121
OS	Operating System. 8, 9, 17, 31
OSINT	Open-Source Intelligence. 44
PoC	Proof of Concept. 27, 29, 34, 36, 39–43, 53, 54, 65, 73, 81, 85, 89, 90, 98, 119

PyPI	Python Package Index. 29, 31
SaaS	Software as a Service. 22, 23
SDK	Software Development Kit. 9
SIEM	Security information and event management. 100
SSC	Software Supply Chain. 1, 2, 5, 6, 17, 59–61
SSCA	Software Supply Chain Attack. 2, 5, 6, 37
TLD	Top Level Domain. 34
YoY	year over year. 4, 8

Theses

Master's Thesis
Hardening the Software Supply Chain: Developing a System to Prevent Dependency Confusion Attacks in Cloud Based Continuous Integration and Deployment Processes

Submitted:	September 20, 2022
by:	Henrik Hauser
Supervisor:	Prof. Dr. Nils Gruschka
Secondary Supervisor:	Prof. Dr. Antje Raab-Düsterhöft
External Supervisor:	Sebastian Kurowski

- The software supply chain, while prividing similar advantages to physical supply chains, presents similar risks in terms of disruption by malicious actors
- The "Dependency Confusion" attack enables threat actors to poison software products by injecting malicious dependencies utilizing existing behaviors in software package management systems
- Existing static and dynamic analysis techniques are inefficient at preventing the attack, considering the large amount of data to be processed
- A novel solution (the "K9 System") using the Ethereum blockchain with smart contracts to store integrity information about private packages to identify impostors is proposed
- The K9 System is proven to be effective at discovering impostor packages based on their file integrity, providing easy access to integrity information, while being highly manipulation resistant and integrable into existing CI/CD environments